51 Data Flow

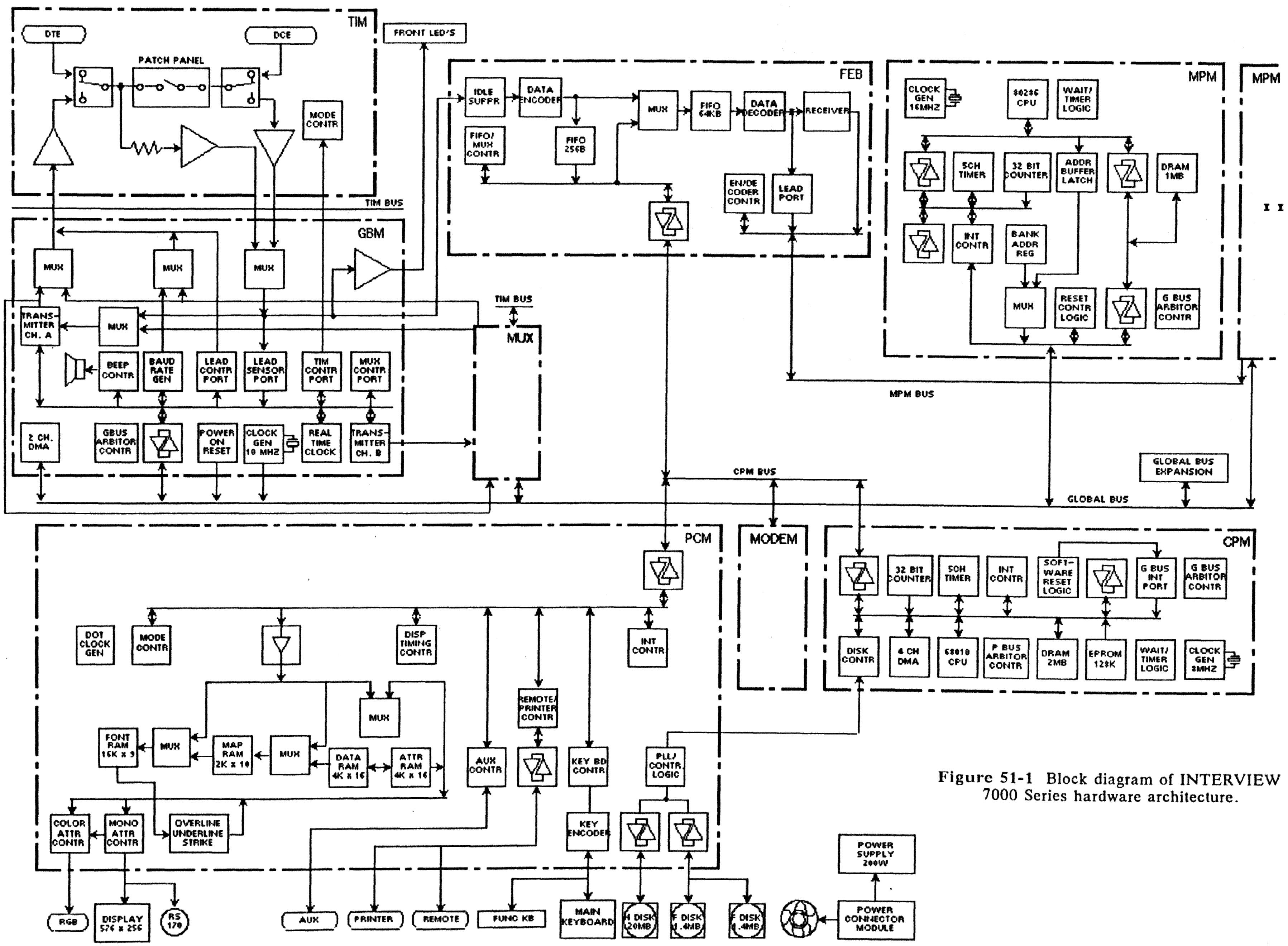


Figure 51-1 Block diagram of INTERVIEW 7000 Series hardware architecture.

51 Data Flow

Figure 51-1 is a block diagram showing the components on each of the six types of logic board in the INTERVIEW 7000 Series. The components on the TIM (Test Interface Module) also are shown. Figure 51-2 indicates the flow of data among the various functional components of the unit.

51.1 Two Types of CPU

The brain of the INTERVIEW is the Motorola 68010 processor on the CPM (Central Processing Module). See Figure 51-1. The 68010 processor controls operations in the unit not directly under control of the user program. 68010 operations include fetching power-up software and initialization routines from the EPROM, controlling disk I/O, and maintaining setup and statistics screens. The operating system in the 68010 is pSOS.

An Intel 80286 processor controls the operation of the MPM (Main Processor Module). The MPM does all higher level processing of receive data. The board also generates the transmit data to be sent out in emulate mode. The 80286 uses a basic, multitasking real-time executive operating system.

An INTERVIEW 7000 and 7200 *TURBO* may have from one to three MPMs, each with its own 80286 CPU. The INTERVIEW 7500 and 7700 *TURBO* always have three MPMs.

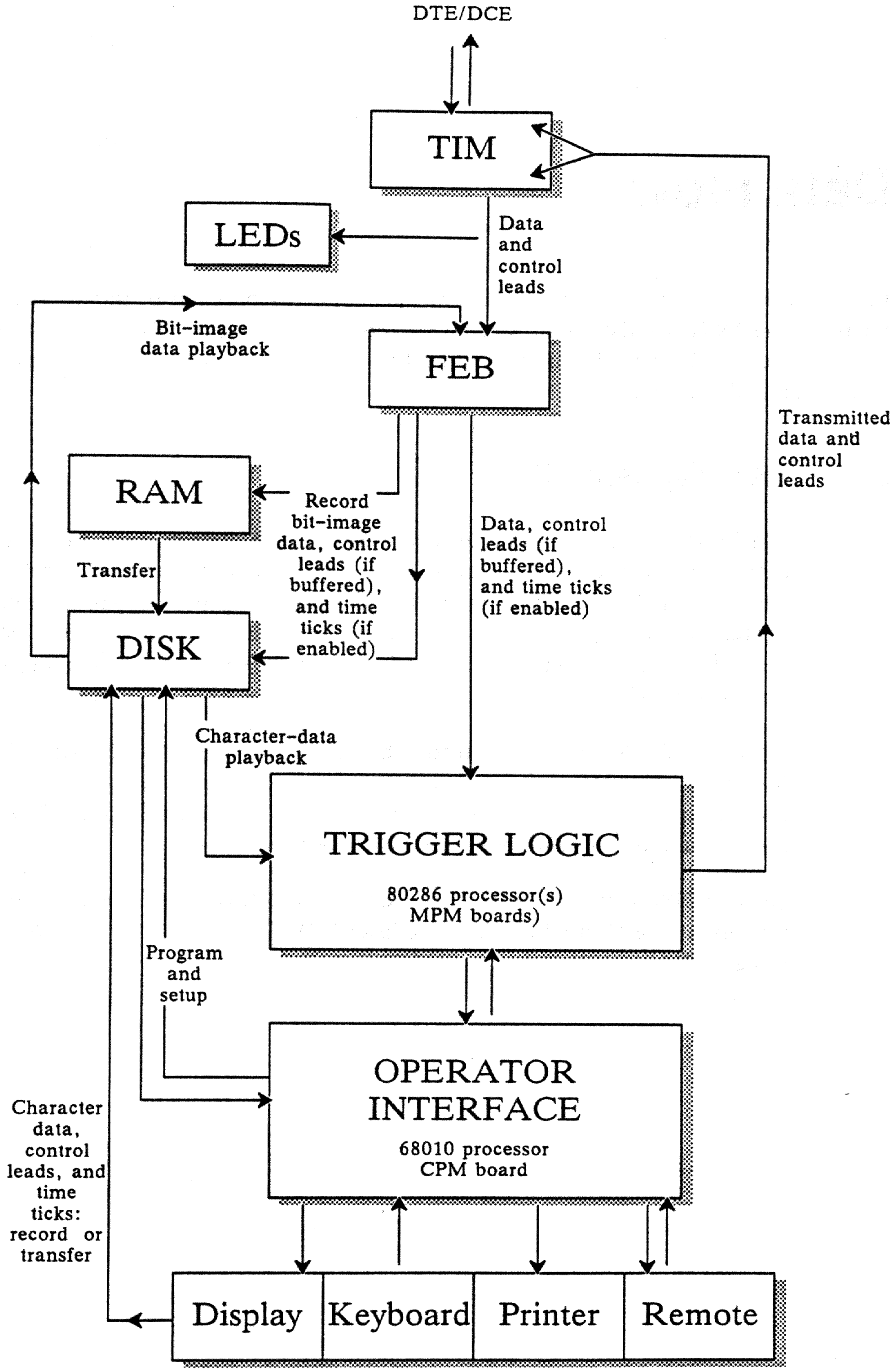


Figure 51-2 INTERVIEW 7000 Series functional diagram.

The 80286 operates on software located in the DRAM on the MPM. See Figure 51-1. This software is the user program—setups, trigger menus, protocol spreadsheet, and protocol state machines (layer packages)—translated and compiled by the CPM and loaded into the MPM. The program will tell the MPM how to process the data, what trigger conditions to look for in the data stream, etc.

The CPM polls the MPM continuously to see if data is available to be output to the printer or the plasma display. This data includes character data, trace data, prompts, and values to be posted to the statistics screens.

While the CPM accesses the MPM on a regular basis, there is no access in the reverse direction. That is, *the user program running on the MPM has no direct access to the CPM*. The user cannot write to one of the menu screens, for example.

51.2 Front-End Buffer

Note in Figure 51-2 that the front-end buffer (FEB) lies squarely between the line interface and (1) the recording medium and (2) the program logic on the MPM. This means that control leads may or may not be recorded and may or may not be seen by the trigger-menu and spreadsheet conditions—depending on the FEB setup (see Section 7).

Once control leads and time ticks (that is, the original timing values) are recorded alongside character data, they are locked in. Since the FEB is not on the playback path for character data, FEB selections do not apply.

Bit-image data, however, does pass through the FEB during playback. Except for the **Idle Suppress** field, FEB selections apply. This means that control leads and time ticks, if recorded with the data, *must* be enabled in order for the program logic to detect them.

Not only characters but also leads and time ticks, if enabled in the FEB setup, are captured automatically in the display buffer (that is, the screen buffer or character RAM).

Data, time ticks, and control leads are encoded in a special storage format by a data-encoder chip on the FEB board. See Figure 51-1. The encoded data is buffered to be sent to the PCM (Peripheral Control Module) for recording and to the MPM for processing.

The encoding process is driven by clock pulses on the line interface. This means that in the absence of external clock (or, if the INTERVIEW is emulating DCE, in the absence of internal clock), neither line data, time ticks nor EIA leads will be recorded or presented to the receivers and to the program logic.

52 Program Main

Softkey–selectable programming "tokens" entered by the user on the Protocol Spreadsheet are translated automatically into C during the initial compiler phases after [RUN] is pressed. Trigger Menu setups also are translated into C. When the translation is complete, the compiler takes over and converts the C code into object code. The C variables and routines used by the translator are documented throughout this volume.

Briefly, the translator makes the following conversions: it turns TESTs into tasks; STATE names into labels; STATEs into *waitfor* clauses; CONDITIONS into *waitfor* expressions that include event variables; and ACTIONS into statements and routines, also inside of *waitfor* clauses.

Then the translator creates a program *main* function that calls every task in the program.

52.1 Translating a Simple Test into C

Suppose that the following simple program, intended to sound the INTERVIEW's alarm at 1 P.M., has been entered on the Protocol Spreadsheet.

```
STATE: sample1
   CONDITIONS: TIME 1300
   ACTIONS: ALARM
```

When the user presses [RUN], *roughly* the following C coding (with some extraneous code removed for clarity) is generated and then compiled:

```
extern fast_event fevar_time_of_day;
extern volatile unsigned short crnt_time_of_day;
task
{
   main ()
   {
      state_sample1:
      waitfor
      {
         fevar_time_of_day && (crnt_time_of_day == 1300):
         {
            sound_alarm ();
         }
      }
   }
} dtest_0;
main ()
{
   dtest_0 ();
}
```

Note that the translator has assigned *state_sample1* to a default TEST named *dtest_0*. It converted the TEST into a task and placed *state_sample1* inside of the task. Then it created a program *main* function and used the program *main* to call every test/task in the program. The tasks appear in the task list in the same order in which they appear in the spreadsheet program. In this instance there was only one task to call.

If you try to enter the program above on the spreadsheet entirely in C, in the first place you will have to surround it with a pair of curly braces. Then it will not compile. The translator does not look inside of curly braces (except to expand constants). It simply lifts up the braced C regions and places them intact into its translation of the softkey portion of the program, before adding a program *main*—even when, as in this instance, a program *main* already is included in a C region. The two *main* functions conflict here, and the compiler issues the error message, "*Error 109: Function main redefined.*"

If we were to remove the *main* function from our C version, the program would compile but it still would not *work*. Here's why. When the translator looks at a program made up entirely of C code, it doesn't see anything. So it creates a program *main* with a task-list that is empty. The task that is declared in the program above (*dtest_0*) is never called.

The rule, then, is that a Protocol Spreadsheet program containing tasks written in C must always have at least one softkey STATE (with its implied task) that calls all the tasks.

52.2 A Minimum of One Softkey *State*

Here is a Protocol Spreadsheet test that works and yet has the minimum number of softkey tokens—one. Note that we have given the task *dtest_0* a new name, since the translator will declare the task-name *dtest_0* as the default test for our new softkey state, *task_list*.

```
{
  extern fast_event fevar_time_of_day;
  extern volatile unsigned short crnt_time_of_day;
  task
  {
     main ()
     {
        state_sample1:
        waitfor
        {
           fevar_time_of_day && (crnt_time_of_day == 1300):
           {
              sound_alarm ();
           }
        }
     }
  } c_test;
}
  STATE: task_list
        {
           c_test ();
        }
```

And here is the program as it is actually compiled. Note that the translator has added a program main that calls *dtest_0* (which in turn calls *c_test*).

```
extern fast_event fevar_time_of_day;
extern volatile unsigned short crnt_time_of_day;
task
{
   main ()
   {
      state_sample1:
      waitfor
      {
         fevar_time_of_day && (crnt_time_of_day == 1300):
         {
            sound_alarm ();
         }
      }
   }
} c_test;
task
{
   main ()
   {
      state_task_list:
      {
         c_test ();
         waitfor              /* This empty waitfor is automatically generated in any state
   {                             that does not contain a waitfor. */
         }
      }
   }
} dtest_0;
main ()
{
   dtest_0 ();
}
```

52.3 Writing the Test Entirely in C

The INTERVIEW is equipped with tools—namely, the *#pragma hook 0* preprocessor directive and linkable-object (LOBJ) files—that make it possible to write a version of the test completely in C.

> **NOTE:** For more information on *#pragma hook* directives, see Section 56.4. Refer also to Section 13.3(P) on linkable-object files.

Write the following C code to an ASCII file (*hook_ctest.s*) using the Protocol Spreadsheet editor's WRITE/U command. Then delete the code from the spreadsheet. Go to the File Maintenance screen and and create a linkable-object file (*hook_ctest.o*) using the Compile command.

```
#pragma hook 0 "c_test();"
extern fast_event fevar_time_of_day;
extern volatile unsigned short crnt_time_of_day;
task
  {
    main()
      {
        state_sample1:
        waitfor
          {
            fevar_time_of_day && (crnt_time_of_day == 1300):
            {
              sound_alarm();
            }
          }
      }
  } c_test_task;
c_test()
  {
    c_test_task();
  }
```

Notice that the "hook" is a call to the routine *c_test*. This routine's only purpose is to start the task, *c_test_task*. A task name is always local to a linkable-object file and never *directly* copied from it. If you try to call the task directly in the *#pragma hook 0* directive, therefore, the spreadsheet program (shown below) will not compile. Since the task name is local to the file, the following error message will be displayed: *"Error 140: Unresolved reference c_test_task."* The rule for including tasks in a linkable-object file, then, is to let the *#pragma hook 0* directive call a routine which starts the task(s).

> **NOTE:** Since task names are local to a file, the definition of *c_test_task* also cannot be located in a referenced LOBJ file different from the one in which it is called.

The Protocol Spreadsheet program required to execute the test consists of a single line:

```
OBJECT: "hook_ctest.o"
```

When translated, the program looks like this:

```
#pragma object "hook_ctest.o"
main()
  {
    c_test();
  }
```

Notice that the routine *c_test* is located within the top-level program *main*. The hook text from a *#pragma hook 0* directive is always put at the end of *main*'s task list. At this point, since *c_test* has not been previously declared, it is assumed to be an *extern* function (not a task) that returns an *int*. The linkable-object file(s) referenced in the spreadsheet program will be searched for the routine's definition.

53 Regions in Spreadsheet

C language can be embedded in a Protocol Spreadsheet program at several access points. A C region can be opened at the top of the program, or in an OBJECT, CONSTANTS, LAYER, TEST, STATE, CONDITIONS, or ACTIONS block.

At these points, simply begin the C region with an opening curly brace. Make your entry and terminate it with a closing curly brace.

The remainder of this section describes C code blocks related to the spreadsheet components, from largest to smallest.

53.1 Layer and Test

The *main* function of a *task* is the highest level function that may be programmed by the user of the INTERVIEW 7000 Series. The keyword *task* in a C region corresponds to the TEST: softkey token on the Protocol Spreadsheet. Typing TEST: **keyboard_alarm** on the spreadsheet is the equivalent of the following C coding:

```
task
{
   #pragma layer 1
   main()
   {
      /* declarations, state-labels, and statements go here */
   }
}
layer_1_test_keyboard_alarm;
```

The INTERVIEW is multitasking, so more than one task/test may be defined. All tasks/tests run concurrently if they are included in the task list created by the translator when it generates the program *main* function. See Section 52, Program Main, for an explanation of how this automatic program *main* is created.

Layers have no existence in C independent of the tasks that they contain. When a user enters the LAYER: token on the spreadsheet followed by a layer number, the C translator prefixes that number to the name of each task that follows. Note in the example above that the test name **keyboard_alarm** was given a *layer_1_test* prefix.

The C translator also issued the preprocessor directive *#pragma layer 1*. The compiler uses this layer declaration to distribute tasks efficiently among 80286 processors. This pragma is an optimizing feature and is not strictly required in the body of the task.

The C translator does nothing else with the layer number other than convert it into a prefix to the task name and construct the *#pragma* directive.

The layer number does, of course, determine many of the branching *softkey* selections that will be available to the user who is not programming in C. The C programmer will find that none of the variables or routines mentioned in this manual is specific to a particular layer. A variable or routine that is supplied, for example, by the X.25 Layer 3 personality package (at the time that the package is loaded in via the Layer Setup screen) will still be available inside of a task that nominally belongs to Layer 1 or Layer 2.

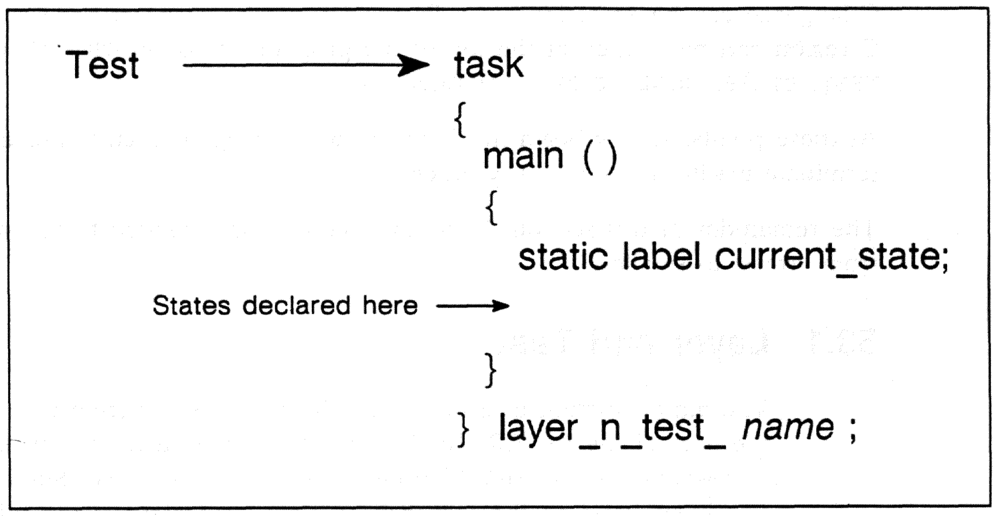


Figure 53-1 C equivalent of a spreadsheet test.

53.2 State, Enter State, and Next State

A STATE on the Protocol Spreadsheet is a label in C, used as a target of a *goto* statement. Typing STATE: **alarm_on** on the spreadsheet is the equivalent of this C coding, placed inside of the braces that follow the task *main*:

```
static label current_state;
state_alarm_on:
current_state = state_alarm_on_loop;
{
    /* statements go here */
    goto (current_state);
    state_alarm_on_loop:
    waitfor
    {
        /* condition clauses go here, each comprised of expression, colon(:), and statements */
    }
    goto (current_state);
}
```

Note that the C translator has taken STATE: **alarm_on** on the Protocol Spreadsheet and produced two state labels, *state_alarm_on* and *state_alarm_on_loop*. The first state label is followed by statements that will be executed immediately upon entering the state. The "loop"-state label always introduces a *waitfor* construction. Both states end in a statement to *goto (current_state)*.

The translator's version of a state includes overhead to cover all cases, including special cases. The loop state is not strictly required, and a streamlined version of the basic state coding that eliminates the extra state will work in most instances:

```
static label current_state;
state_alarm_on:
{
   /* declarations and statements go here */
   waitfor
   {
      /* condition clauses go here, each comprised of expression, colon(:), and statement(s) */
   }
   goto (current_state);
}
```

Note these points about states created entirely by the programmer:

- A *goto* statement cannot be used inside of a *waitfor* construction.
- You must use a *break* statement to exit the *waitfor* construction.
- You may dispense with the *current_state* variable and *goto* a state label, in which case the opening and closing parens may be omitted.

(A) Declaring States

The state name followed by the colon (:) is itself a label declaration and does not require an additional declaration.

(B) Enter State

The C translator puts a *waitfor* construction into every "loop" state. If you want a statement to be executed immediately without waiting for an event, you may place that statement in the nonloop state, outside of the *waitfor* statement. The following is an example of a state in which the *sound_alarm* routine is executed immediately.

```
static label current_state;
state_alarm_on:
current_state = state_alarm_on_loop;
{
   sound_alarm();
   goto (current_state);
   state_alarm_on_loop:
   waitfor
   {
   }
   goto (current_state);
}
```

The example above is the equivalent of this spreadsheet entry:

```
STATE: alarm_on
   CONDITIONS: ENTER_STATE
   ACTIONS: ALARM
```

A hybrid version also may be created:

```
STATE: alarm_on
{
   sound_alarm();
}
```

The *sound_alarm* function is executed immediately, since the translator places it above the *waitfor*. When you enter a CONDITIONS: block on the spreadsheet, you move inside a *waitfor*—unless you place your C region immediately following an ENTER_STATE.

An ENTER_STATE condition may cause the translator to generate an *if* statement in the nonloop state (above the *waitfor* state). Here is a spreadsheet example:

```
STATE: alarm_on
  CONDITIONS: ENTER_STATE
     COUNTER anyname EQ 3
  ACTIONS: ALARM
```

This is the C version:

```
static label current_state;
state_alarm_on:
current_state = state_alarm_on_loop;
{
   if (counter_anyname.current  == 3) sound_alarm();
   goto (current_state);
   state_alarm_on_loop:
   waitfor
   {
   }
   goto (current_state);
}
```

And here is a hybrid version:

```
STATE: alarm_on
{
   if (counter_anyname.current  == 3) sound_alarm();
}
```

(C) Next State

The C translator supplies the statement "*goto (current_state)*" at the bottom of every state that it codes. If *current_state* has been redefined and if the program reaches the bottom of the state, the *goto* statement will redirect the program toward a new state label. That is how the program is redirected into *state_alarm_on_loop* in this translator's version of **STATE: alarm_on**:

```
static label current_state;
state_alarm_on:
current_state = state_alarm_on_loop;
{
   goto (current_state);
   state_alarm_on_loop:
   waitfor
   {
   }
   goto (current_state);
}
```

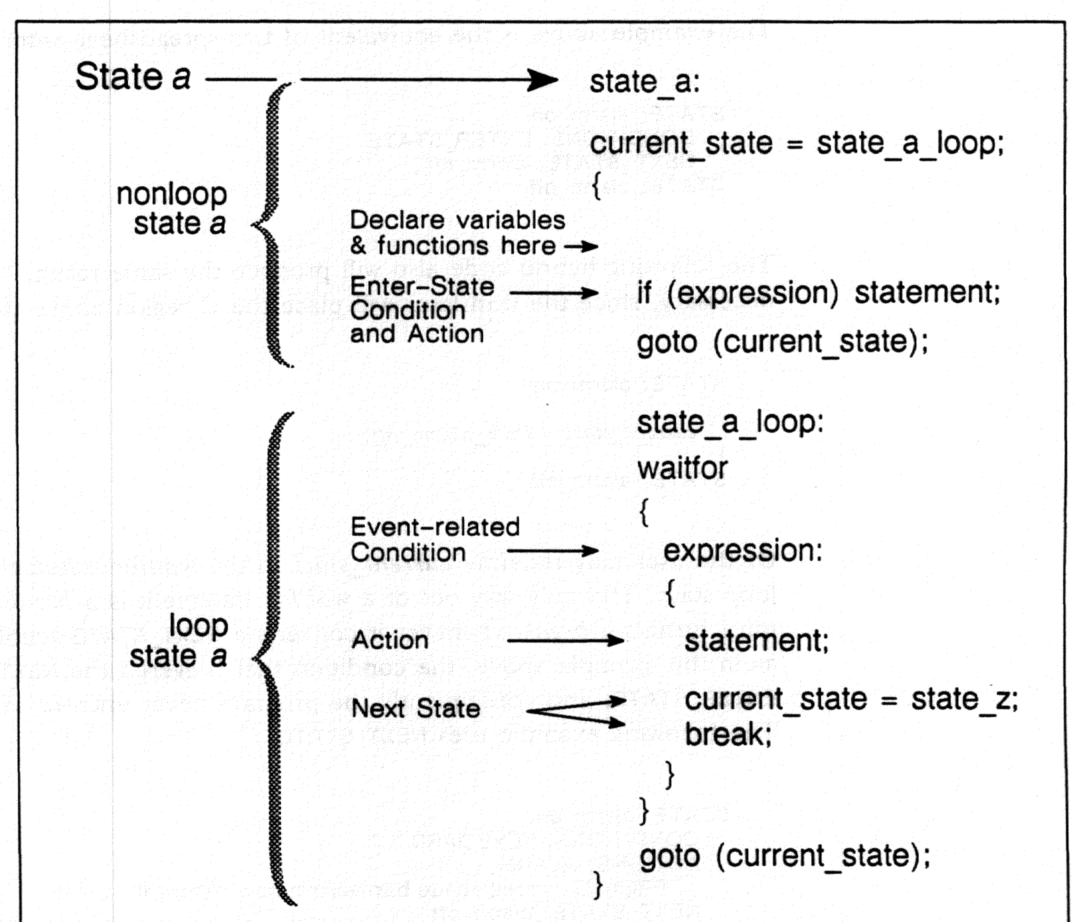


Figure 53-2 Basic C structure of a spreadsheet state.

If the user wants to redefine *current_state*, he may do so in the nonloop state, in which case the loop (*waitfor*) state will be bypassed:

```
static label current_state;
state_alarm_on:
current_state = state_alarm_on_loop;
{
   current_state = state_alarm_off;
   goto (current_state);
   state_alarm_on_loop:
   waitfor
   {
   }
   goto (current_state);
}
state_alarm_off:
/* etc. */
```

The example above is the equivalent of this spreadsheet entry:

```
STATE: alarm_on
  CONDITIONS: ENTER_STATE
  NEXT_STATE: alarm_off
STATE: alarm_off
```

The following hybrid code also will produce the same result. No *break* is necessary, since the translator will place the C region above the *waitfor*.

```
STATE: alarm_on
{
  current_state = state_alarm_off;
}
STATE: alarm_off
```

Or the user may redefine *current_state* in the *waitfor* statement itself, inside the loop state. The only way out of a *waitfor* statement is a *break*, so the translator must furnish a *break* whenever it converts a NEXT_STATE action into C (unless, as in the example above, the condition that triggered the NEXT_STATE action was ENTER_STATE, and consequently the program never entered the *waitfor* loop). The following example uses NEXT_STATE:

```
STATE: alarm_on
  CONDITIONS: KEYBOARD " "
  ACTIONS: ALARM
    PROMPT "press space bar--alarm now disabled"
  NEXT_STATE: alarm_off
STATE: alarm_off
  CONDITIONS: KEYBOARD " "
  ACTIONS: PROMPT "press space bar--alarm is activated"
  NEXT_STATE: alarm_on
```

Here is the C version:

```
static label current_state;
state_alarm_on:
current_state = state_alarm_on_loop;
{
  goto (current_state);
  state_alarm_on_loop:
  waitfor
  {
    keyboard_new_any_key && (keyboard_any_key == ' '):
    {
      sound_alarm();
      display_prompt ("press space bar--alarm now disabled");
      current_state = state_alarm_off;
      break;
    }
  }
  goto (current_state);
}
```

```
state_alarm_off:
current_state = state_alarm_off_loop;
{
   goto (current_state);
   state_alarm_off_loop:
   waitfor
   {
      keyboard_new_any_key && (keyboard_any_key == ' '):
      {
         display_prompt ("press space bar--alarm is activated");
         current_state = state_alarm_on;
         break;
      }
   }
   goto (current_state);
}
```

Various hybrid versions are possible. Here is one:

```
STATE: alarm_on
   CONDITIONS:
   {
      keyboard_new_any_key && (keyboard_any_key == ' ')
   }
   ACTIONS:
   {
      sound_alarm();
      display_prompt ("press space bar--alarm now disabled");
      current_state = state_alarm_off;
      break;
   }
STATE: alarm_off
   CONDITIONS:
   {
      keyboard_new_any_key && (keyboard_any_key == ' ')
   }
   ACTIONS:
   {
      display_prompt ("press space bar--alarm is activated");
      current_state = state_alarm_on;
      break;
   }
```

53.3 Conditions and Actions

When a condition is translated into C code by the INTERVIEW, the resulting expression is enclosed in braces at the top of a *waitfor* statement. The only exception to this rule is the ENTER_STATE condition—see Section 53.2(B), above.

The conditional expression is followed by a colon and then by the statement that constitutes the action to be taken when the condition is true. If more than one action is coded, braces must be used to form a statement block. See Figure 53-3.

Typing CONDITIONS: KEYBOARD " " on the spreadsheet is the equivalent of this C coding, placed inside of the braces that follow the reserved word *waitfor*:

```
keyboard_new_any_key && (keyboard_any_key == ' '):
{
   /* action-statements or routines go here */
}
```

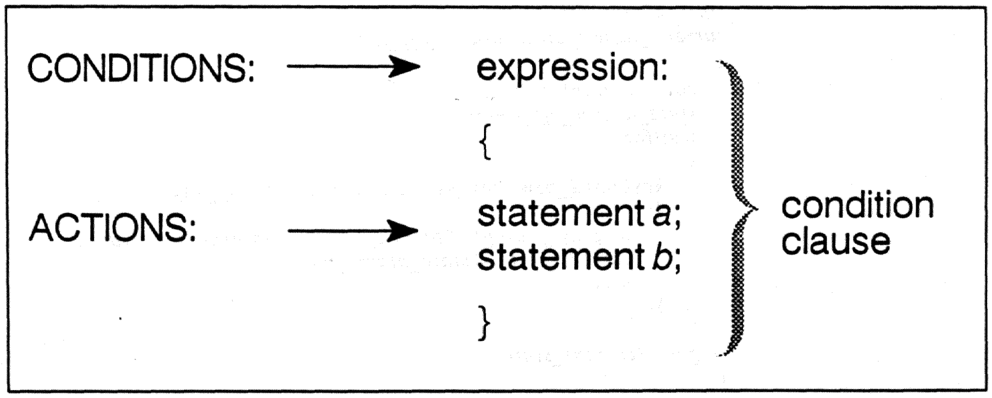


Figure 53-3 The translator converts the Condition-and-Action "trigger" into a condition clause inside of a *waitfor* statement.

(A) Multiple Condition Clauses

Following the semicolon that terminates the statement (or following the statement block), you may enter another condition clause. These clauses correspond to triggers on the Trigger menus or conditions-and-actions blocks inside a state on the Protocol Spreadsheet. Multiple condition clauses may be placed inside of one *waitfor* construction. (There is only one *waitfor* statement per state.)

Here is an example of a state with two "triggers":

```
STATE: keyboard_prompt
  CONDITIONS: KEYBOARD "1"
  ACTIONS: ALARM
    PROMPT "You have pressed the 1 key."
  CONDITIONS: KEYBOARD "2"
  ACTIONS: ALARM
    PROMPT "You have pressed the 2 key."
```

A version in C would have two condition clauses:

```
state_keyboard_prompt:
waitfor
{
  keyboard_new_any_key && (keyboard_any_key == '1'):
  {
    sound_alarm();
    display_prompt ("You have pressed the 1 key.");
  }
  keyboard_new_any_key && (keyboard_any_key == '2'):
  {
    sound_alarm();
    display_prompt ("You have pressed the 2 key.");
  }
}
```

If you are mixing spreadsheet tokens with C, place condition clauses inside of STATE: blocks. Any C region at the top of a State block is placed above the automatic *waitfor* statement. You must therefore supply your own *waitfor* word, since a condition clause is syntactically valid only inside of a *waitfor*. An example follows.

```
STATE: keyboard_prompt
{
   waitfor
   {
      keyboard_new_any_key && (keyboard_any_key == '1'):
      {
         sound_alarm();
         display_prompt ("You have pressed the 1 key.");
      }
      keyboard_new_any_key && (keyboard_any_key == '2'):
      {
         sound_alarm();
         display_prompt ("You have pressed the 2 key.");
      }
   }
}
```

A word of warning is in order. When your program executes this code, it will find itself stuck in a *waitfor* statement beneath the label *state_keyboard_prompt*. If you want to exit this *waitfor*, you must execute a *break* in a statement block in one of the condition clauses. Once you have broken outside of the *waitfor*, you may *goto* another state.

If you add softkey CONDITIONS, ACTIONS, or NEXT_STATE blocks to the state above, they will be placed inside a different *waitfor* statement, the one that is created automatically inside a state called *state_keyboard_prompt_loop*. See Section 53.2 (particularly Figure 53-2). What may look like a single state on the spreadsheet really will be two different states which never are active at the same time.

(B) Multiple Expressions

Expressions may be logically *and*ed (&&) or *or*ed (||) together inside a condition clause. Here is the spreadsheet version of a CONDITIONS block with two expressions:

```
CONDITIONS: KEYBOARD "2"
   FLAG keyboard_disabled 0
ACTIONS: PROMPT "You have pressed the 2 key."
```

Inside the condition clause in C, the translator supplies a double ampersand (&&) to connect the keyboard expressions with the flag expression:

```
keyboard_new_any_key && (keyboard_any_key == '2')
&& (flag_keyboard_disabled.current == 0):
{
   display_prompt ("You have pressed the 2 key.");
}
```

Inside a CONDITIONS block, the translator is able to *and* a softkey condition correctly with a C expression. Note that the user types the C expression without a terminating colon. The translator will supply one later:

```
CONDITIONS: KEYBOARD "2"
{
   flag_keyboard_disabled.current == 0
}
ACTIONS: PROMPT "You have pressed the 2 key."
```

The *and*ing is also successful when the C expression is placed above the softkey condition inside the CONDITIONS block:

```
CONDITIONS:
{
   flag_keyboard_disabled.current == 0
}
   KEYBOARD "2"
ACTIONS: PROMPT "You have pressed the 2 key."
```

If you want to insert a comment into a Conditions block, remember that the translator does not look inside of C regions (except to expand constants). It will take the comment and *and* it with the rest of the expressions in the Conditions block. Since a comment is not a C expression, the program will not compile: see Section 53.3(D). Note in the following example that a 1 has been inserted inside the C region along with the comment in order to make the code compile and in order to make the expression "true."

```
CONDITIONS:
{
   /* This comment will be anded with the keyboard expression. */ 1
}
   KEYBOARD "2"
ACTIONS: PROMPT "You have pressed the 2 key."
```

(C) Event Variables

The translator converts most Conditions blocks on the Protocol Spreadsheet into two or more expressions linked by the logical *and* operator (&&). The keyboard condition in the examples above was typical: KEYBOARD "2" on the spreadsheet became a pair of expressions logically *and*ed in C.

The first expression, *keyboard_new_any_key*, is an event variable. Event variables are very important in the INTERVIEW implementation of C, and the programmer should observe the following rules of thumb:

1. *An event variable usually is paired with a nonevent variable.* At the moment an event variable comes true in a *waitfor* construction, all nonevent (or "status") variables attached to that event variable are evaluated for truth or falsity. Whenever any keyboard key is struck, the event variable *keyboard_new_any_key* comes true. At that moment, the nonevent expression *keyboard_any_key* == '2' is evaluated to determine whether it is true or false.

2. *A* waitfor *statement must include at least one event expression.* A *waitfor* statement without an event variable will not compile. There must be some event that *might* transpire to cause the nonevent expressions to be evaluated.

3. *An event variable may appear alone in an expression.* It is possible (though unusual) to have an event expression that is not *and*ed with a nonevent expression. When the translator converts CONDITIONS: DTE GOOD_BCC into C, for example, the resulting expression is this simple event variable:

   fevar_gd_bcc_td:

4. *A nonevent variable also may appear alone.* It also is possible (though the translator does not do this inside of *waitfor* statements) to have a nonevent expression that is not *and*ed with an event expression—as long as there is an event expression somewhere in the *waitfor* construction. The following program will compile and work:

```
{
    extern fast_event keyboard_new_any_key;
    extern volatile unsigned short keyboard_any_key;
}
        STATE: keyboard_prompt
        CONDITIONS:
        {
            keyboard_new_any_key && (keyboard_any_key == '1')
        }
        ACTIONS: PROMPT "You have pressed the 1 key."
        CONDITIONS:
        {
            keyboard_any_key == '2'
        }
        ACTIONS: PROMPT "You have pressed the 2 key."
```

   In this example, *keyboard_any_key == '2'* is not *and*ed with an event variable. As a result, it is attached automatically to the event variable *keyboard_new_any_key* in the Conditions block above. If there had happened to be other event variables in the state, it would have been attached to them as well; so that when any event in the state came true, *keyboard_any_key == '2'* would be evaluated.

   NOTE: Other event variables in the state would cause *keyboard_any_key* to be *evaluated*, but would not necessarily cause it to be *updated*. Event variables are guaranteed to update only their associated nonevent variables. In the example above, *keyboard_any_key* is an associated nonevent variable for the event variable *keyboard_new_any_key*.

5. *Two event variables may not be combined.* Two event variables may never be combined in a condition clause, since two events never are simultaneous. Since all spreadsheet conditions have event variables associated with them—counter conditions have the *counter_name_change* event variable, for example—it might seem impossible to combine a counter with another

condition in a single CONDITIONS block. In fact, in the case of a few special combinable conditions—buffer-full, counter, flag, and EIA are examples—the translator will sometimes omit the event variable. When two or more combinable conditions are combined, the translator uses a first come, first served rule that is explained in Section 54.3, Programming Considerations.

(D) Evaluating Nonevent Expressions

Nonevent expressions are true if they have a nonzero value. In the following program, the "trigger" will sound the alarm when any keyboard key is struck because all of the nonevent expressions are nonzero:

```
{
  extern fast_event keyboard_new_any_key;
}
    STATE: boolean
    CONDITIONS:
    {
      keyboard_new_any_key && 1 && 99 && 10003
    }
    ACTIONS: ALARM
```

This version never will sound the alarm, because one of the *and*ed components is zero:

```
{
  extern fast_event keyboard_new_any_key;
}
    STATE: boolean
    CONDITIONS:
    {
      keyboard_new_any_key && 1 && 0 && 10003
    }
    ACTIONS: ALARM
```

Relational expressions like *keyboard_any_key* == '2' and logical expressions connected by && (like those above) and || are defined automatically to have the value 1 if true and 0 if false.

(E) Multiple Statements

Statements may be blocked together inside a condition clause. Here is the spreadsheet version of an ACTIONS block with two statements:

```
CONDITIONS: KEYBOARD "2"
ACTIONS: PROMPT "You have pressed the 2 key."
   ALARM
```

The C version is a condition clause with two routines, *display_prompt* and *sound_alarm*, inside a block or compound statement:

```
keyboard_new_any_key && (keyboard_any_key == '2'):
{
  display_prompt ("You have pressed the 2 key.");
  sound_alarm();
}
```

A hybrid version, part spreadsheet language and part C language, will work:

```
CONDITIONS: KEYBOARD "2"
ACTIONS: PROMPT "You have pressed the 2 key."
{
   sound_alarm();
}
```

The hybrid example as it stands will not allow you to declare routines and variables, because the translator will place these declarations in a statement block beneath the *display_prompt* routine. For declarations, move the C region to the top of the Actions block; or use double braces to open a new statement block lower down, since declarations are legal following the left brace that introduces any compound statement.

53.4 Example of Complete C Program

Some of the examples in the previous pages of this section were incomplete, in that they included variables that were not declared, or they lacked a softkey STATE that could generate a proper program main. The following is an extended example that compiles and runs. It includes many of the pieces that formed the shorter examples in this section. It is written for the Protocol Spreadsheet *as completely as possible* in C. (See Section 52.3 on how to write a program completely in C.)

```
{
   extern fast_event keyboard_new_any_key;
   extern volatile unsigned short keyboard_any_key;
   task
   {
      main()
      {
         static label current_state;
         state_alarm_on:
         current_state = state_alarm_on_loop;
         {
            goto (current_state);
            state_alarm_on_loop:

         waitfor
            {
               keyboard_new_any_key && (keyboard_any_key == ' '):
               {
                  sound_alarm();
                  display_prompt ("press space bar--alarm now disabled");
                  current_state = state_alarm_off;
                  break;
               }
            }
            goto (current_state);
         }
         state_alarm_off:
         current_state = state_alarm_off_loop;
         {
            goto (current_state);
            state_alarm_off_loop:
            waitfor
            {
               keyboard_new_any_key && (keyboard_any_key == ' '):
```

```
                        {
                          display_prompt ("press space bar--alarm is activated");
                          current_state = state_alarm_on;
                          break;
                        }
                    }
                    goto (current_state);
                }
            }
        }
        layer_1_test_keyboard_alarm;
    }
        STATE: task_list
        {
            layer_1_test_keyboard_alarm();
        }
```

53.5 Summary of C Regions

The translator removes the outer braces from a C region and places it into one of the six basic levels of source code shown in Figure 53-4.

(A) Declarations

Declare your variables and routines in a C region, delimited by curly braces { and }, at the top of your program or at the top of a Constants, Layer, Test, State, or Actions block. Declare a variable preceded by its type descriptors and followed by a semicolon, as in these examples:

```
{
   extern fast_event keyboard_new_key;
   extern fast_event keyboard_new_any_key;
   extern fast_event fevar_time_of_day;
   short minutes;
}
```

We have not bothered to declare routines in most of the examples in the manual, since it is not necessary. In the absence of a declaration, the compiler assumes that the routine is external and that it returns an integer. In nearly all cases, this assumption works. In the few cases where a routine returns a *long* (*get_68k_phys_addr* is an example), it must be declared.

1. *Automatic declaration.* In cases where the translator declares a variable automatically, the user does not have to declare the variable himself. For example, a KEYBOARD condition, when entered via softkey, will declare the variable *keyboard_new_key* automatically for the entire program. When a variable has been declared twice in a program block, the program may not run. Instead, the compiler will put up a message such as the following: *Error 110: keyboard_new_key redeclared.* In software version 5.00 and in earlier software, the compiler flagged double declarations and aborted the compilation.

   Sometimes it is difficult to keep track of the exact version of a variable that the translator is declaring. Some external variables have been improved for the use of C programmers, and we have documented the newer version in our tables and in many of our examples. The translator may still use an older version of the variable.

In an earlier software release, for example, the variable *extern event keyboard_new_key* was speeded up and renamed *extern fast_event keyboard_new_key*. The translator still uses the older name to declare the variable.

The variable *keyboard_new_any_key* is a still more recent improved version of *keyboard_new_key*—improved in that it detects the striking of non-ASCII keys as well as the ASCII set. The translator never declares *keyboard_new_any_key* automatically.

Similarly, the translator uses an older version of *extern fast_event fevar_eia_changed*. The older version is *extern event evar_eia_changed*. In the earlier software, compiler error messages such as "*keyboard_new_key redeclared*" and "*Variable fevar_eia_changed undeclared*" will inform you what the translator is doing in each instance.

2. *Legal declaration*. Declarations are legal following the left brace that introduces any compound statement. Figure 53-4 shows that when the user opens a braced C region following a TEST:, STATE:, or ACTIONS: keyword, the translator removes the outer braces from the C region and plants the C code just inside the left brace at Level 2, 4, and 6 of the source code. Declarations therefore are valid at the top of these regions.

   Declarations should be grouped at the top of any region, since they are not allowed in a statement block below an executable statement. This program will not compile, because the *sound_alarm* routine precedes a declaration:

```
{
   extern fast_event fevar_eia_changed;
}
     STATE: lead_changes
        CONDITIONS:
        {
           fevar_eia_changed
        }
        ACTIONS:
        {
           sound_alarm();
           int lead_changes;
           lead_changes ++;
        }
```

   Declarations never are legal at Level 5 (Figure 53-4)—that is, preceding the colon in a condition clause inside a *waitfor* statement. Declarations always are legal at Level 1, since there are no executable statements at that level.

   The set of variables listed as *extern* cannot be declared below Level 1. *Extern* has a specialized meaning at the task level or lower: it is used to "forward-declare" a variable without actually reserving storage space. The variable must be declared again (but not as *extern*) in the body of the task.

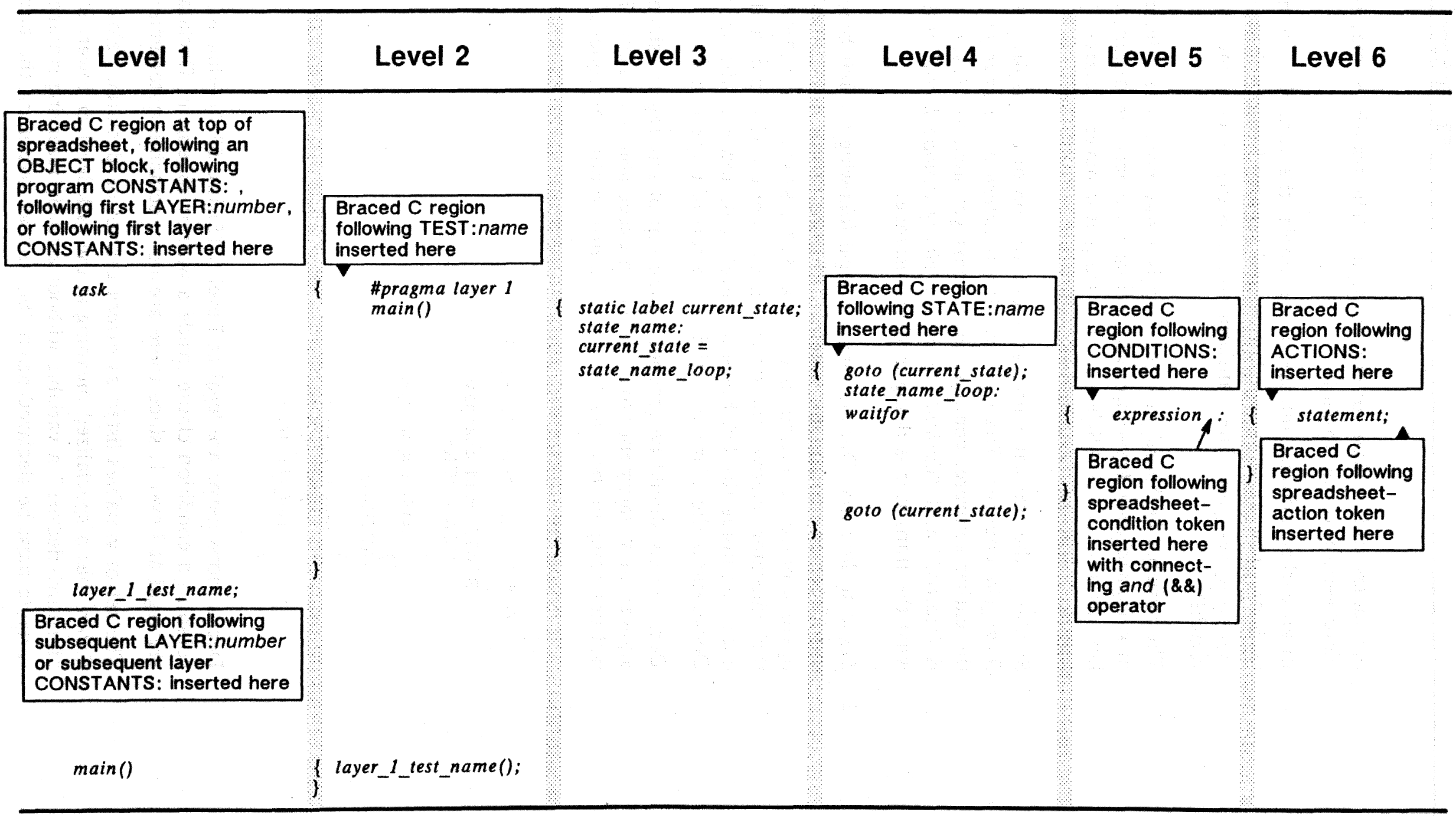


Figure 53-4 The translator removes the outer braces from a C region and places it into one of six basic levels of source code. The "telescoping" of the braces indicates the scope of declarations. A variable or routine declared for Level 1 is declared for the remainder of Level 1 and across all levels to the right.

3. *Scope.* The "telescoping" of the braces in Figure 53-4 indicates the scope of declarations. A variable or routine declared for Level 1 is declared for the remainder of Level 1 and across all levels to the right. This means that a variable or routine declared at the top of Level 1 will be global throughout the program. You can force a declaration to the top of Level 1 by placing it in braces (1) at the top of the Protocol Spreadsheet; (2) before or after an OBJECT block; (3) inside a CONSTANTS block above the Layer level; (4) inside the first LAYER block on the spreadsheet; or (5) inside the CONSTANTS block in the first LAYER block.

   Here is an example of a global declaration:

```
{
    extern fast_event fevar_eia_changed;
}
LAYER: 1
    TEST: leads
        STATE: init
            CONDITIONS:
            {
                fevar_eia_changed
            }
            ACTIONS: PROMPT "Status of a lead has changed."
```

   A variable or routine declared at Level 1 (Figure 53-4) is declared for subsequent layers and tests, whether the subsequent layer is higher or lower. The concept of higher and lower layers is relevant to softkey entry on the Protocol Spreadsheet, but is not carried over into the source code. To the compiler, a TEST in Layer 2 and a TEST in Layer 3 are simply concurrent tasks. The task that is first in the program is compiled first. That is the only meaning of "higher" and "lower" to the compiler.

   A variable or routine may have its scope limited to a particular Test, State, or Actions block. A variable or routine also may be redeclared at different levels. Given more than one valid declaration, the lower or *nearer* one applies.

4. *Initialization.* A variable must be of the *static* storage class to pass its value into a *waitfor* statement. Declarations at Level 1 of the source code (Figure 53-4) are always *static*, whether or not they are declared so. A variable that is initialized at Level 4 (Figure 53-4) must be declared as *static* by the programmer if the initialized value is to be used inside a *waitfor*.

(B) Statements

Executable statements may occur at four levels (Figure 53-4) in the source code: at Level 2 of the program *main* function, where the function is defined; at Levels 3 and 4, where the task *main* function is defined; and at Level 6, inside a *waitfor* statement. The programmer has no access to Level 3. To access Level

4, the programmer may open a C region just beneath the STATE: *name* identifier. He may access Level 6 by opening a braced C region below the ACTIONS: keyword.

Levels 1 and 2 are reserved for declarations. The program *main* function executes statements at Level 2 (see the bottom of Figure 53-4), but this function is accessible only to the translator.

54 Events

In Run mode, the user program in the INTERVIEW moves from program STATE to program STATE. In each state a set of conditions is tested, with one or more actions the result of a particular condition coming true.

In the INTERVIEW's implementation of C, a "state" is a special control structure called a *waitfor* clause that is placed in the program directly following a *label* named for the state. Program movement is controlled by *goto* statements that reference these labels.

Each *waitfor* clause defines a set of interrupts ("events") that it is waiting for. When a *waitfor* clause is active and an interrupt/event occurs that is defined in that clause, the entire clause is processed. All of the conditions in the clause are tested and appropriate actions (statements, operations, routines) are executed.

The *waitfor* clause is a mechanism designed specifically for the data-communications testing environment, in which the program must interact at high speed with a variety of unpredictable inputs.

54.1 Example of Event: *fevar_time_of_day*

In the *waitfor* clause in an earlier example (Section 52 of this volume), the condition was this:

```
fevar_time_of_day && (crnt_time_of_day == 1300)
```

Once every minute, the CPM sends an interrupt to the MPM. This interrupt takes the form of a *fevar_time_of_day* event.

If the program includes a *fevar_time_of_day* condition, the interrupt each minute will cause the variable *crnt_time_of_day* to be updated.

If the current state includes a *fevar_time_of_day* condition, the interrupt each minute will satisfy that condition. At the same time all other conditions in the clause, including non-event (that is, non-interrupt-driven) conditions such as *crnt_time_of_day == 1300*, will be tested.

The relationship between an event variable such as *fevar_time_of_day* and its associated nonevent variable (in this case, *crnt_time_of_day*) can be summarized as follows: the event variable anywhere in the program causes the nonevent variable to be updated each time the event occurs. The event variable in the currently active *waitfor* loop causes the nonevent condition to be tested each time the event occurs.

Figure 54-1 illustrates this relationship, as well as the relationship between an event and a nonassociated variable. The figure shows, for example, how an EIA event might cause the time-of-day variable to be checked but not updated; and how a time-of-day event might cause the EIA-status variable to be checked but not updated. "Event" in the figure means event variable, while "variable" means nonevent variable.

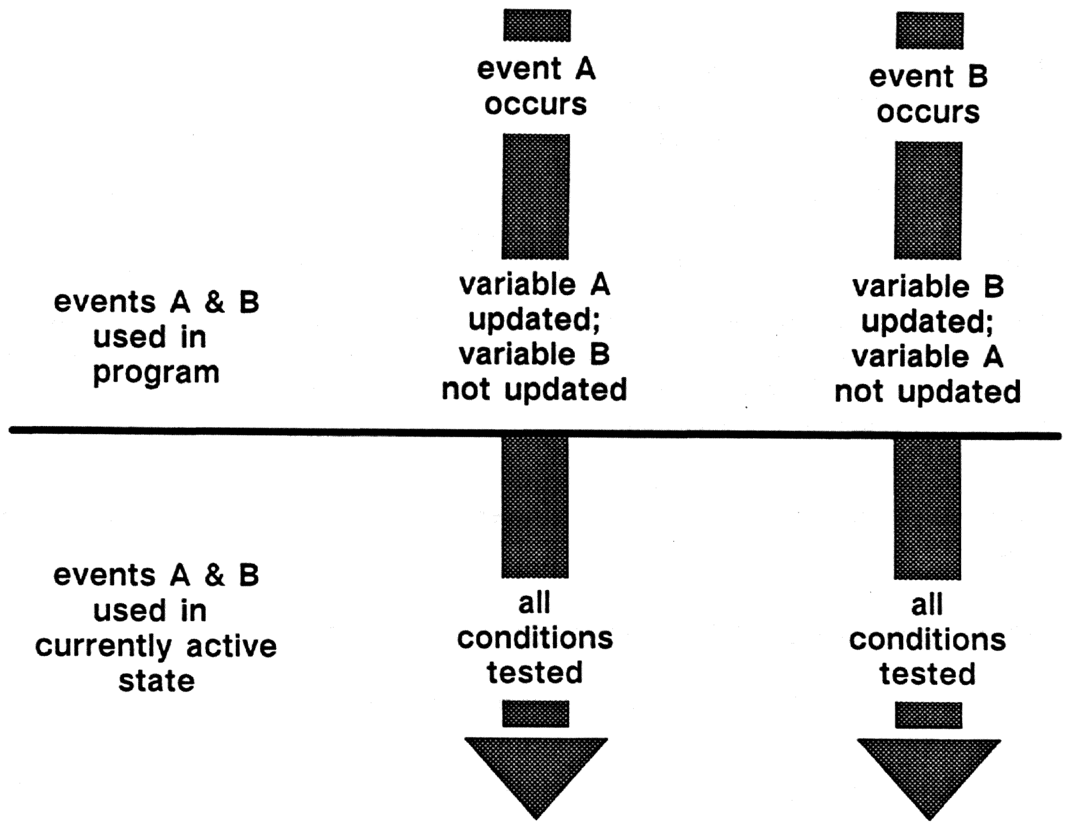


Figure 54-1 This figure is meant to show the effect of event A on its associated variable (variable A) as well as its effect on a nonassociated variable (variable B).

54.2 Various Origins of *waitfor* Events

Interrupts sent to the MPM from the CPM include *fevar_time_of_day* and *keyboard_new_key*. Interrupts sent to the MPM by the SCC (Serial Communications Controller) chip in the FEB include *fevar_rcvd_char_td*, *fevar_gd_bcc_rd*, and *fevar_eia_changed*. Some interrupts are sent to the user program by the protocol state machines in the layer packages. Examples are *dce_frame* and *dte_packet*.

Interrupts also can be generated by the program itself. The program sends an interrupt in the form of a "signal." *counter_name_change* and *flag_name_change* are events that are signaled by the program itself, since the program is in charge of all counter and flag increments, decrements, and sets.

54.3 Programming Considerations

By itself in a *waitfor* clause, *crnt_time_of_day == 1300* never can be true, since only interrupts/events cause the nonevent conditions in the clause to be processed. On the other hand, *counter_name_change && flag_name_change* never can return true, since two events cannot occur simultaneously.

Because two events never are simultaneous, the programmer (and the built-in translator) has a decision to make whenever two nonevent conditions, such as *counter_name.current == 3* and *flag_name.current == 5*, are *and*ed together. If the programmer writes *counter_name_change && (counter_name.current == 3) && (flag_name.current == 5)*, the condition may be true when *counter_name.current* transitions to 3 but it never will be true when *flag_name.current* transitions to 5, since there is no interrupt to cause the condition to be checked at that moment. If an interrupt (*flag_name_change*) is tied to *flag_name.current*, then *counter_name.current* transitioning to 3 will not be detected.

When the user combines a flag condition with a counter condition on a single Trigger Setup menu, the translator solves the dilemma of which event to "wait for" by generating a two-pronged *waitfor* condition that is approximately the following:

```
(counter_name_change && (counter_name.current == 3) &&
(flag_name.current == 5)) || (flag_name_change &&
(counter_name.current == 3) && (flag_name.current == 5)):
```

On the Protocol Spreadsheet, the translator simply attaches the appropriate event variable to the first softkey condition listed. If the user enters

```
CONDITIONS: COUNTER name EQ 3
  FLAG name 101
```

the translator converts this to *(counter_name_change && (counter_name.current == 3) && (flag_name.current == 5)*. The user is then free to repeat the combined condition, reversing the order of the elements (and therefore invoking the *flag_name_change* interrupt) the second time around.

NOTE: The examples in Section 54.3 above are somewhat simplified. The actual translator versions are made more complicated by the inclusion of *counter_name.old* and *flag_name.old* variables that are explained in Section 62.

55 Receiving and Transmitting Data

As the INTERVIEW monitors the data source (line or disk), it signals the arrival of each character by an event variable (*fevar_rcvd_char_rd* or *fevar_rcvd_char_td*) and it stores each character momentarily in a variable (*rcvd_char_rd* or *rcvd_char_td*) accessible by the user. Data can be taken from the line in this form and copied into memory or into an interlayer message buffer. BOP-framed data is copied automatically into an interlayer ("IL") buffer.

The user transmits data from the INTERVIEW by creating a transmit-data structure and then referencing the structure in an *ll_transmit* routine. Or the user may copy the data into an interlayer buffer (or simply reference the data in the buffer) and then call out the buffer in an *ll_il_transmit* routine.

The IL buffers have several advantages as a storage medium for data. First, they are reusable. They are allocated dynamically and erased automatically unless the user takes steps to maintain them. Without these reusable buffers, data in Run mode would quickly eat up all of the memory in the unit.

Second, IL buffers support linked lists. There are routines that will start a list, insert data at the top of a list, and append data to the bottom of a list. Linked lists are well suited to layered-protocol transmissions, where the transmit string is built incrementally as the transmission moves down the layers.

55.1 Locating Data in an IL Buffer

When a BOP frame is placed automatically in an IL buffer, a data primitive is created automatically and the event variable *m_lo_ph_prmtv* is signaled. The segment number of the IL buffer is recorded in the variable *m_lo_ph_il_buff*. The offset from the start of the buffer to the start of the data is recorded in the variable *m_lo_ph_sdu_offset*. This offset is always 32 bytes. What is considered data at higher layers may have a larger offset, since each layer's data begins farther into the frame. See Figure 55-1 for an illustration of a gradually shrinking "service data unit" (SDU) and a gradually expanding SDU offset.

The first memory location in the first of the sixteen IL buffers is 03a80000, the first location in the second buffer is 03b00000, the first location in buffer #3 is 03b80000, buffer #4 starts at 03c00000, and so on through 04200000. Each of these addresses is 32 bits. The high-order 16 bits is the 80286 segment number (03a8, 03b0, 03b8, 03c0, etc., through 0420). This is the number that the software passes around when it wants to identify an IL buffer, simply because 16 bits are easier and faster to pass around than 32 bits, the low-order 16 of which are always zero when we are discussing the starting location of each buffer.

When we want to look at data in the buffer, we need to reference not a 16-bit segment number but a 32-bit address. So we cast the segment number (always a *short*, 16 bits) into a *long* and move the number over to its high-order position, sixteen bits to the left. We add 32 to the number to bypass the header information for the buffer. Then we cast the new *long* as a character pointer. Here, for example, is *m_lo_ph_il_buff* converted into a pointer to the first byte in a frame:

```
char * m_frame_ptr;
m_frame_ptr = (void*)(((long)m_lo_ph_il_buff << 16) + 32);
```

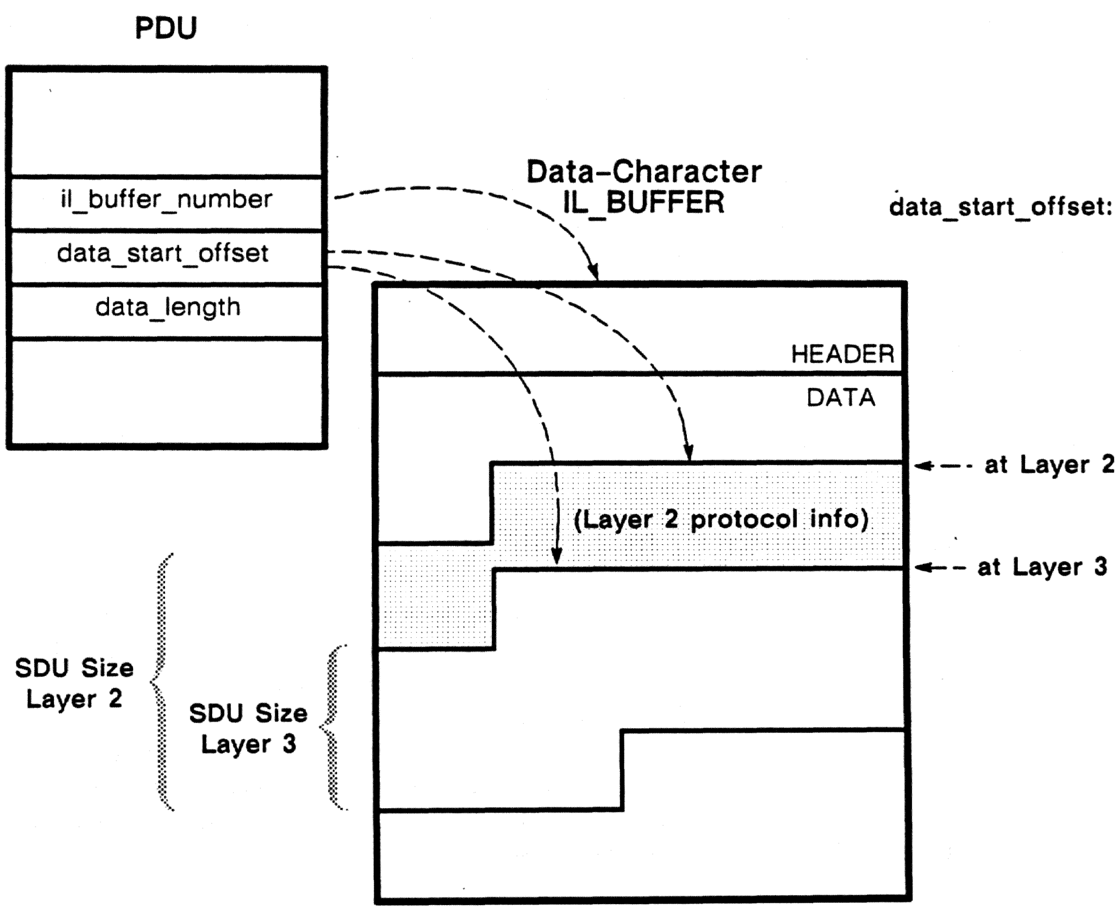


Figure 55-1 When an IL buffer is passed upward, the data offset changes and the data length changes, but the buffer itself does not change.

55.2 Monitor Path vs. Receive Path

The variables *m_lo_ph_prmtv*, *m_lo_ph_il_buff*, and *m_lo_ph_sdu_offset* are part of a set of monitor services that handle IL buffers in both monitor and emulate modes. These variables are updated for data on either data lead. The layer packages use these variables to generate the protocol traces. The translator uses them to implement spreadsheet condition-tokens such as PH_TD_DATA IND and DTE INFO.

Another set of variables are maintained in emulate mode and are updated for data on the receive side only. These variables have names that reveal their obvious relationship to the monitor set: *lo_ph_prmtv*, *lo_ph_il_buff*, *lo_ph_sdu*, etc. These receive-side variables are used by the translator to implement spreadsheet condition-tokens such as PH_DATA IND and RCV INFO.

Whenever a BOP frame is placed automatically in an IL buffer during an emulate run, events *m_lo_ph_prmtv* and *lo_ph_prmtv* both are signaled. The segment number of the same IL buffer is recorded in two variables, *m_lo_ph_il_buff* and *lo_ph_il_buff*.

55.3 Passing a Buffer Upwards

Layer 1 stores data in IL buffers and passes these buffers to Layer 2 automatically, as we have seen. If a Layer 2 personality package is loaded in from the Layer Setup screen, the second data byte in the buffer (the 34th byte overall) is checked to determine the frame type. If the contents of the buffer is an Info frame, a data primitive is created automatically and the event variable *m_lo_dl_prmtv* is signaled. The segment number of the IL buffer is recorded in the variable *m_lo_dl_il_buff*. This is the same segment number that was stored previously in *m_lo_ph_il_buff*.

The offset from the start of the buffer to the start of the data—Layer 2 or data link (DL) data—is recorded in the variable *m_lo_dl_sdu_offset*. This offset is always 34 in MOD 8. This number represents the 32-byte buffer header plus a 2-byte frame header that is of no interest to Layer 3, which will use *m_lo_dl_il_buff* and *m_lo_dl_sdu_offset* to construct its packet trace.

The size of the data component in the buffer is stored in the variable *m_lo_dl_sdu_size*. This number will be 2 bytes smaller than the variable *m_lo_ph_sdu_size*.

If no layer packages are loaded, none of the buffer-handling services are provided automatically at Layer 2 or higher. The programmer can provide the services "manually" as indicated above.

If layer packages are loaded, monitor-path variables (those variables whose names begin with *m_*) are updated automatically in order to drive the protocol traces. Receive-path variables such as *lo_dl_prmtv*, *lo_dl_il_buff*, and *lo_dl_sdu* are generated as needed by GIVE_DATA actions entered by the user on the Protocol Spreadsheet. Otherwise it is up to the C programmer to maintain these variables. For example, the user passing an IL buffer up to Layer 3 might write this code:

```
lo_dl_il_buff = lo_ph_il_buff;
lo_dl_sdu = (lo_ph_sdu + 2);
pdu_ptr->data_length = (pdu_ptr->data_length - 2);
signal (lo_dl_prmtv);
```

The same updates of variables and the same signal would be generated if the user called a *send_dl_prmtv_above* routine, as follows:

```
_set_maint_buff_bit (lo_ph_il_buff, &l2_relay_baton);
send_dl_prmtv_above (lo_ph_il_buff, l2_relay_baton, lo_ph_sdu + 2, pdu_ptr->data_length - 2,
0x45);
```

The *send_dl_prmtv_above* routine requires an SDU size value. There is no receive-path variable (equivalent to *m_lo_ph_sdu_size* on the monitor path) that maintains this value. Determine the SDU size from the *data_length* variable located in the *pdu*-structure. In the examples above, *pdu_ptr* is a structure pointer. The SDU size, therefore, is referenced as *pdu_ptr->data_length*. Refer to Section 63.1 for more information on the *pdu* structure.

> NOTE: Do not use *m_lo_ph_sdu_size* for receive-path routines such as *send_dl_prmtv_above*. It is not updated reliably at the same moment that other receive-path variables are updated.

0x45 is the code for a DL_DATA IND primitive.

55.4 Layer 1 Transmit

Line transmissions are accomplished through L1 transmit routines. Shown below is a program that ends in an *l1_il_transmit* routine. This routine puts the data contents (the service data unit or "SDU," not the buffer header) of an IL buffer out onto the line.

Note that there is a set of routines leading up to the transmit routine. This set of routines is necessary to get a buffer, to start a linked list inside the buffer, and finally to insert several chunks of data into the list before it is transmitted.

```
{
 unsigned short bufnum;
 unsigned short baton;
 unsigned short list_hd_offset;
 static unsigned char data[] = "((FOX))";
 static unsigned char pkt_hdr[3] = {0x10,0x07,0};
 static unsigned char frm_hdr[2] = {0x03, 0};
 int length;
 unsigned short transmit_tag = 1;
}
```

```
STATE: fox
  CONDITIONS: KEYBOARD " "
  ACTIONS:
  {
    _get_il_msg_buff(&bufnum,&baton);
    _start_il_buff_list(bufnum,&list_hd_offset);
    length = sizeof(data) -1;
    _insert_il_buff_list_cnt(bufnum,list_hd_offset,&data[0],length);
    _insert_il_buff_list_cnt(bufnum,list_hd_offset,&pkt_hdr[0],3);
    _insert_il_buff_list_cnt(bufnum,list_hd_offset,&frm_hdr[0],2);
    l1_il_transmit(bufnum,baton,list_hd_offset,transmit_tag);
  }
```

The transmit string will look like this on the INTERVIEW's data display:

EX NU QL BL NU THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG 0123456789,[G]

(A) Segment Number

The *l1_il_transmit* routine required four arguments as input. First, it required the segment number of the IL buffer that was intended to be transmitted. This number was supplied by the *_get_il_msg_buff* routine, and we called the number *bufnum*. There are a total of sixteen numbered IL buffers available to the program.

(B) Relay Baton

The second argument was the number of the "relay baton" or "maintain bit." This relay baton was supplied by the *_get_il_msg_buff* routine, and we called the variable that held the number *baton*. A relay baton is passed down automatically with every send or transmit routine and serves to hold the buffer until it has been processed by the next layer (or transmitted by Layer 1). Then the baton is freed.

There are sixteen numbered relay batons available *for each IL buffer*. At the moment that all sixteen batons (or maintain bits) are free, the buffer is returned automatically to the pool of free IL buffers and its contents are no longer available to the program.

In many applications—X.25 Layer 2 and Layer 3 personality packages, for example—an extra maintain bit is reserved (via the *_set_maint_buff_bit* routine) each time a buffer is sent down. This extra maintain bit is held onto in case a frame or packet must be resent, and is not freed (in a *_free_il_msg_buff* routine) until the outstanding frame or packet has been acknowledged.

(C) List-Header Offset

In addition to buffer number and baton number, the *l1_il_transmit* routine also requires as input the offset from the start of the buffer to the linked-list header. This offset is supplied at the moment the linked list is started by the *_start_il_buff_list* routine. In the program above we called this offset *list_hd_offset*.

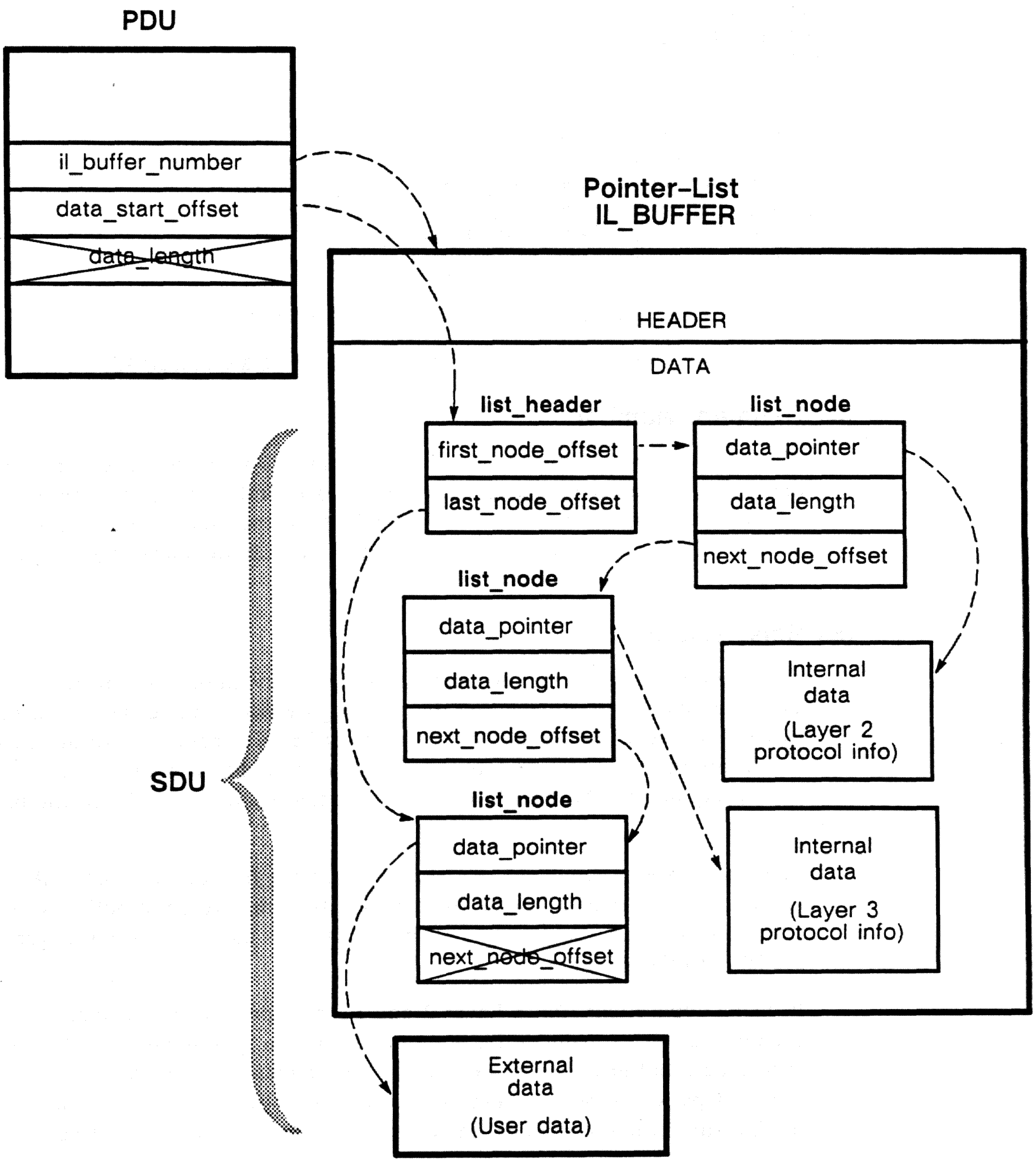


Figure 55-2 When an IL buffer is passed downward, the data-start offset gives the location of the list header. This list header and the various pieces of the transmission (the list nodes) are threaded together.

Figure 55-2 illustrates how the list header ties the linked list together by identifying the offsets to the first and last nodes. A list node is created by each *_insert_il_buff_list_cnt* or *_append_il_buff_list_cnt* routine. The program in Section 55.4 has three *_insert_il_buff_list_cnt* routines. The IL buffer that is transmitted therefore has three list nodes.

(D) Transmit Tag

The fourth argument in the *ll_il_transmit* routine is a "transmit tag" that determines the type of BCC to be appended to the transmission. This variable is stored in the 32-byte header of each IL buffer. Refer to the structure *il_buffer* in the table of OSI structures, Table 63-1.

A transmit tag of 1 means a good BCC and 2 means a bad BCC. 3 causes an aborted transmission.

55.5 Passing a Buffer Between Tasks

At this point we need to modify our *ll_il_transmit* program to allow different layers—which are simply separate concurrent tasks in the programming architecture—to contribute list nodes to the IL buffer intended for transmission. The resulting transmit string will be the same as before, but three different tasks will have contributed data components to the transmitted buffer. In our new program, a Layer 4 task will provide the fox message, Layer 3 will provide the *_insert_il_buff_list_cnt* routine that references the 3-byte packet header, and Layer 2 will provide the insert routine that references the 2-byte frame header.

How do the separate layer tasks communicate with each other so that the right buffer is accepted at the moment it is handed down? They relay information in the same way that tasks always communicate, by signals that are detected throughout the program as event variables. When Layer 4 sends an IL buffer down in a *send_n_prmtv_below* routine, an event variable at Layer 3 (*up_n_prmtv*, not shown in the program below but implied nevertheless in the N_DATA REQ condition) comes true and at the same time updates the variables *up_n_il_buff* and *up_n_sdu*. Layer 3 can use these variables to identify the new IL buffer and to determine the offset to the list header in that buffer. With this information, Layer 3 can insert its own list node into the buffer before passing it down to layer 2.

Here is the program, followed by a few explanatory comments:

```
{
 unsigned short bufnum;
 unsigned short l4_baton;
 unsigned short l3_baton;
 unsigned short l2_baton;
 unsigned short list_hd_offset;
 static unsigned char data[] = "((FOX))";
 static unsigned char pkt_hdr[3] = {0x10,0x07,0};
```

```
  static unsigned char frm_hdr[2] = {0x03,0};
  int length;
  extern volatile unsigned short up_n_il_buff;
  extern volatile unsigned short up_dl_il_buff;
  extern volatile unsigned short up_n_sdu;
  extern volatile unsigned short up_dl_sdu;
}
LAYER: 4
 STATE: fox
  CONDITIONS: KEYBOARD " "
  ACTIONS:
  {
   _get_il_msg_buff(&bufnum,&l4_baton);
   _start_il_buff_list(bufnum,&list_hd_offset);
   length = sizeof(data) -1;
   _insert_il_buff_list_cnt(bufnum,list_hd_offset,&data[0],length);
   send_n_prmtv_below(bufnum,l4_baton,list_hd_offset,0,0x64,0);
  }
LAYER: 3
 STATE: packet_header
  CONDITIONS: N_DATA REQ
  ACTIONS:
  {
   _insert_il_buff_list_cnt(up_n_il_buff,up_n_sdu,&pkt_hdr[0],3);
   _set_maint_buff_bit(up_n_il_buff,&l3_baton);
   send_dl_prmtv_below(up_n_il_buff,l3_baton,up_n_sdu,0,0x44,0);
  }
LAYER: 2
 STATE: frame_header
  CONDITIONS: DL_DATA REQ
  ACTIONS:
  {
   _insert_il_buff_list_cnt(up_dl_il_buff,up_dl_sdu,&frm_hdr[0],2);
   _set_maint_buff_bit(up_dl_il_buff,&l2_baton);
   send_ph_prmtv_below(up_dl_il_buff,l2_baton,up_dl_sdu,0,0x24,0);
  }
```

In the send–primitive routines, the hex values 64, 44, and 24 identify the primitives as *data* requests. See, for example, the values of *up_n_prmtv_code* in Table 63-4.

Note that there is no longer an *l1_il_transmit* routine in the program. When Layer 2 executes a *send_ph_prmtv_below* routine, Layer 1 handles the transmit function automatically.

The *send_ph_prmtv_below* routine does not have a transmit–tag argument that allows us to specify the BCC. Since the *l1_il_transmit* routine, which has a transmit–tag input, is being handled automatically, it is not immediately clear how you would send the transmit string with a bad BCC. Here is one way. Instead of the *send_ph_prmtv_below* routine at Layer 2, use the *l1_il_transmit* routine as follows:

```
l1_il_transmit(up_dl_il_buff,l2_baton,up_dl_sdu, 2);
```

The 2 in the argument represents the transmit tag for a bad BCC.

If it seems strange to be using an *ll_il_transmit* routine at Layer 2, remember that none of the variables or routines is really layer-specific. In C, layers are simply concurrent tasks.

A "realistic" implementation of this program might be made somewhat more complicated by two additional elements. One or more *_open_space_in_il_buff* routines might be used so that, as far as possible, text data could be copied into the buffer where it would then be erased when the buffer was freed. (One of the advantages of IL buffers is that the space inside them can be recycled.)

Another complication is that for the same transmission, more than one linked list might be started in a single buffer. The example under the *_insert_il_buff_list_cnt* routine in Section 63.3(A) shows Layer 2 accepting a buffer from Layer 3 and starting a new linked list. This allows Layer 3 to reconstruct its original linked list in case a packet-resend is needed.

55.6 Sample Transmit Program: Sync or Async Echo

This application monitors incoming data for text strings bounded by ␂ and ␃ or ␗. It copies these strings into an IL buffer and then echoes them back out onto the line, preceded by two ASCII sync characters. The program will work in most data formats as long as ASCII ␂ and ␃ are included.

The program may be modified for EBCDIC ␖, ␂, ␃, and ␗. Use received-character variables *fevar_rcvd_char_rd* and *rcvd_char_rd* for data received on RD.

```
{
  extern fast_event fevar_rcvd_char_td;
  extern volatile unsigned short rcvd_char_td;
  unsigned short number, length;
  unsigned short il_buffer_number, relay_baton, data_start_offset;
  unsigned char echo_string[100] = {'␖', '␖'};
}

STATE: look_for_stx
  CONDITIONS:
  {
     fevar_rcvd_char_td && rcvd_char_td == '␂'
  }
  ACTIONS:
  {
     number = 2;
     echo_string[number] = rcvd_char_td;
     number++;
  }
  NEXT_STATE: construct_echo_string
STATE: construct_echo_string
  CONDITIONS:
  {
     fevar_rcvd_char_td
  }
  ACTIONS:
  {
     echo_string[number] = rcvd_char_td;
```

```
            number++;
            if ((rcvd_char_td == 'Ex') || (rcvd_char_td == 'EB'))
            {
               length = number;
            }
        }
        CONDITIONS: RECEIVE GOOD_BCC
        NEXT_STATE: transmit_echo_string
    STATE: transmit_echo_string
        CONDITIONS: ENTER_STATE
        ACTIONS:
        {
            _get_il_msg_buff(&il_buffer_number, &relay_baton);
            _start_il_buff_list(il_buffer_number, &data_start_offset);
            _insert_il_buff_list_cnt(il_buffer_number, data_start_offset, echo_string, length);
            l1_il_transmit(il_buffer_number, relay_baton, data_start_offset, 1);
        }
        NEXT_STATE: look_for_stx
```

55.7 Sample Transmit Program: BOP Echo

When **Format:** BOP is selected on the Line Setup screen, every frame that is received at the line interface is placed in an IL buffer and passed up to Layer 2. This sample program makes a pointer to the I-field in the most recent IL buffer received at Layer 2, and then it echoes the data back out in the C equivalent of a SEND INFO action. If you try this program, be sure to load the X.25 or SDLC package at Layer 2.

```
{
    char * data_ptr;
    extern volatile unsigned short rcvd_frame_buff_seg;
    extern volatile unsigned short rcvd_frame_sdu_offset;
    extern volatile unsigned short rcvd_frame_sdu_size;
    struct send_frame_structure
    {
        unsigned char addr_type;
        unsigned char frame_type;
        unsigned char nr_type;
        unsigned char ns_type;
        unsigned char p_f_type;
        unsigned char bcc_type;
        unsigned char addr_value;
        unsigned char cntrl_byte;
        unsigned char nr_value;
        unsigned char ns_value;
    };
    struct send_frame_structure frame;
    unsigned short number, baton, offset;
}
LAYER: 2
    STATE: echo
        CONDITIONS: RCV INFO
        ACTIONS:
        {
            data_ptr = (void *)(((long)rcvd_frame_buff_seg << 16) + rcvd_frame_sdu_offset);
            _get_il_msg_buff(&number, &baton);
            _start_il_buff_list(number, &offset);
            _insert_il_buff_list_cnt(number, offset, data_ptr + 2, rcvd_frame_sdu_size - 2);
            frame.bcc_type = 1;
            send_frame(number, baton, offset, &frame);
        }
```

56 C Basics

```
                    ** Protocol Spreadsheet **

LAYER: 1
  TEST: bsc_one
   {static label prev_state;}
     STATE: polling
       CONDITIONS: RECEIVE ONE_OF "␗␃"
       ACTIONS: SEND "␖␖␐/" GOOD_BCC
       {prev_state = state_polling;}
       NEXT_STATE: ack0
     STATE: ack0
       CONDITIONS: RECEIVE ONE_OF "␗␃"
       ACTIONS: SEND "␖␖␐70" GOOD_BCC
       {current_state = prev_state;
        break;
       }

 F 1      F 2      F 3      F 4      F 5      F 6      F 7      F 8
LAYER:   TEST:   STATE:   CONDS:    NEXTST:
```

Figure 56-1 Using C to return to the previous state.

56 C Basics

C programming language as implemented in the INTERVIEW 7000 Series is based on the current ANSI recommendations. It contains several extensions to the language which enhance its utility in protocol testing, notably multi–tasking.

C is intended as an aid to INTERVIEW users who have advanced programming knowledge. A sophisticated programming tool, C can be applied to testing requirements which are not met by Protocol Spreadsheet selections. C is useful, for instance, in the analysis and "intelligent" manipulation of variable data strings anticipated within a complex protocol. Additional applications of C are the creation of customized protocol and program trace displays.

Figure 56-1 provides a means of returning to whatever state was the former state, without you the programmer knowing which state was previously active. This "go to previous state" function is not a standard spreadsheet feature. The example employs Bisync protocol to demonstrate the usefulness of this capability. The test begins in a state called **polling**. Here, an ACK1 is sent whenever the end of any received data is encountered, and the test passes to the state called **ack0**. This time when the end of received data is encountered, an ACK0 is sent, and the test returns to whatever state it was in formerly.

The first C region is the declaration of the variable *prev_state*, which allows the variable to be used anywhere within the test. In the second C region, the variable *prev_state* is initialized to the name of the active state. The third C region shows the transition of the test to the previously active state. Depending on the contents of the *prev_state* variable, the former state could be one of any number of states. This capability means that, as the programmer expands the simple test, the state ack0 can be used again and again as a utility state from which the test returns to the former state, removing the need for repetitive spreadsheet entry.

56.1 Notable Variations in C

The AR version of C varies in certain respects from the ANSI standard. Notable exceptions to the standard are outlined below. A full set of implementation–defined variations appear in Appendix K.

(A) Reserved Words

The following two reserved words, in addition to those covered in the ANSI standard, are included in C:

task

waitfor

(B) Predeclared identifiers

The following type identifiers are always predeclared. They are not defined in any *#include* files, nor are their definitions required in any program. Thus they are part of the INTERVIEW C lexicon, even though they are not reserved words and therefore do not appear in the language summary in Appendix K.

event

fast_event

label

(C) Floating Point Notation

Since Floating Point Notation is not required in the protocol testing environment and since corresponding calculations could degrade processing speed, floating point notation is omitted from the AR implementation of C. Fixed point calculations, however, are performed.

(D) Values Returned from C Functions

Functions declared within AR's implementation of C may only return values for data types which are 1, 2, or 4 bytes long. Consequently, a function cannot legally return most structure or union types.

56.2 Editing a C Program

Entries in C are made on the Protocol Spreadsheet, accessed from the Main Program screen. All editing functions available on the spreadsheet can be applied to C coding. Refer to Section 26 for a description of these editing functions.

56.3 Error Reporting in C

Most syntax errors made on the Protocol Spreadsheet are indicated by strike-through of the text where the error occurs. This facilitates correction of entries as you create a test.

Errors which appear in C coding are not indicated by the editor. However, when the program is compiled (when you press [RUN]), the errors will be noted. If there are errors in the program, the INTERVIEW will automatically revert to the Protocol Spreadsheet rather than run the program.

(A) Locating Errors

The cursor is automatically positioned near the first error when the INTERVIEW reverts to the Protocol Spreadsheet. A diagnostic message about the error will be displayed at the top (second line) of the screen. Errors pertaining to the general syntax of the spreadsheet are explained in text. Errors noted by the C pre-processor or compiler are displayed as numbers, with explanatory text if the filename *sys/error_text* is accessible at the moment on a disk. (The file should always be accessible in units with hard disks.) These numbered messages are listed in Appendix A3.

Press GO-ERR again to move down through the spreadsheet to the next error. When you press GO-ERR and there are no more errors, the message "No More Errors" will be displayed.

56.4 Preprocessor Directives

The INTERVIEW supports preprocessor directives *#define* and *#include*. The full set of ANSI preprocessor directives are supported on the INTERVIEW. Included among these directives are *#if*, *#else*, *#ifdef*, *#ifndef*, and *#undef*. (Refer to the ANSI Recommendation for a discussion of these directives.) Implementation-defined *#pragmas* are also preprocessor directives. *#pragma object* and *#pragma hook* are two of the AR *#pragmas*. As the name implies, preprocessor directives are processed before the program in which they appear is compiled.

Preprocessor directives are easy to recognize, since they are always preceded by a pound sign (#). Spaces are significant to the meaning of the directives, since other delimiters are generally not used. Note also that a semi-colon cannot be used to terminate a preprocessor directive. Instead, a directive is terminated by a hard Carriage Return or some indicator of line continuation. Press RETURN to terminate the directive (no indication of the Return will appear on the screen). Type \ (backslash) and press RETURN at the end of the line on the screen to indicate that the directive continues on the next line. You may also allow text to wrap to the next line by continuing to type. (Wrapped lines are indicated on the screen by the highlighted symbol ■.)

(A) #define

The *#define* directive gives you the convenience of replacing frequently referenced items with a text string of any length.

1. *Placement*. A *#define* directive may be placed at the beginning of a logical line anywhere in a legal C region. The eight valid positions for C regions on the Protocol Spreadsheet are shown in Figure 53-4. The *#define* directive may also be placed in a separate *#include* file. Use the *#include* directive as explained in (B) to invoke the file and make the macro-substitutions it indicates in your main program file.

2. *Format.* The directive follows this format:

   #define identifier string

   For example, if you enter the following line of code,

   #define message The quick brown fox 12345

   the identifier message (wherever it appears exactly as written in the file being acted upon) is replaced in subsequent lines of code by the string The quick brown fox ....12345. The replacement, the macro-substitution, is performed before the code is compiled. When you enter the *#define* directive, leave a space between the directive (*#define*) and the identifier. There should be no spaces in the identifier. The space following the identifier indicates that the next ASCII character (or blank) starts the replacement string. Spaces are allowed and are considered part of the string. Terminate the string (and the directive) as described at the beginning of this sub-section.

3. *Nesting.* *#define* substitutions may be nested. Of course, the nested replacements must be described by a *#define* directive which precedes the *#define* for the replacement text which contains them.

   There is one exception to nesting identifiers—the macro substitution will not be performed when the identifier occurs in a string. In the example below, the programmer tries to nest MAXTRIES within the definition of MESSAGE:

   ```
   #define MAXTRIES 3
   #define MESSAGE "Maximum retransmissions is MAXTRIES."
   ```

   A call to *displayf(MESSAGE);* causes the following to be displayed:

   ```
   Maximum retransmissions is MAXTRIES.
   ```

   This is certainly not what the programmer intended.

(B) #include

#include files, when invoked in a program, are read into the program file before the program is compiled. As a result, your program has access to commonly used items such as subroutines (input/output and string operations, for example), global variables, constants, and structures without your having to enter or modify the required code repeatedly.

1. *Format.* The format for the directive is as follows:

   #include <filename>

   or

   #include "filename"

#include files follow standard naming conventions. See Section 13.2(E). As an added convention, the suffix *.h* is appended to the end of the name (as in the filename *stdio.h*).

2. *Search rules for #include files.* The delimiters you use to surround the filename determine how the INTERVIEW searches its filing system for the file.

- The <> delimiters are intended for files which are supplied by AR. When these delimiters are used, the following directories—and only the following directories—are searched, in the order given:

  1. */sys/include* on current drive (indicated on File Maintenance screen)
  2. The directory named as the current directory on the File Maintenance screen (provided that the current directory is not the root directory for FD1, FD2, or hard disk)
  3. */usr/include* on current drive (indicated on File Maintenance screen)
  4. *FD1/sys/include*
  5. *FD2/sys/include*
  6. *HRD/sys/include*
  7. *FD1/usr/include*
  8. *FD2/usr/include*
  9. *HRD/usr/include*

  NOTE: The directory names are given in the format which the INTERVIEW interprets as the absolute path from the root directory of the disk named before the first slash. So *HRD/sys/include* means */sys/include* on the hard disk.

- The " " delimiters are intended for user-created files. The same directories are searched for the filename, but they are searched in the following order:

  1. The directory named as the current directory on the File Maintenance screen (provided that the current directory is not the root directory for FD1, FD2, or hard disk)
  2. */usr/include* on current drive (indicated on File Maintenance screen)
  3. */sys/include* on current drive (indicated on File Maintenance screen)
  4. *FD1/usr/include*
  5. *FD2/usr/include*
  6. *HRD/usr/include*
  7. *FD1/sys/include*
  8. *FD2/sys/include*
  9. *HRD/sys/include*

If you have used the same filename for an include file in more than one directory, the file which is actually read in as a result of an *#include* directive will be from the first directory searched which contains that filename. The delimiters you use, then, can make a difference in the file selected for inclusion.

The filename enclosed in <> or " " delimiters may be a relative pathname. The highest directory in the pathname must reside in the current directory or in one of the */include* directories. In response to an *#include "disk_io/stdio.h"* directive, for example, the INTERVIEW first looks for a *disk_io* subdirectory in the current directory on the File Maintenance screen and then for an *stdio.h* file in that subdirectory. If the file is not found, the search for the relative pathname continues according to the sequence designated for " " delimiters.

If the file is not located in any of these directories, an error message is returned to the operator.

(C) #pragma object

Use the *#pragma object* directive to access the compiled routine definitions in a linkable-object file. The OBJECT block-identifier discussed in Section 24.4 may also be used for this purpose. (Also see Section 13.3(P) on creating a linkable-object file—displayed as type LOBJ in the directory listings on the File Maintenance screen).

1. *Placement*. Place the *#pragma object* directive inside any legal C region on the Protocol Spreadsheet. Except for those containing the *static* attribute, routine definitions from an LOBJ file always have global scope. It makes sense, therefore, to position the directive at the top of your spreadsheet program along with other global declarations and definitions.

2. *Format*. The format for the *#pragma object* directive is as follows:

   #pragma object "filename.o"

   A *#pragma object* directive references only one LOBJ filename, but you may include as many directives as you wish.

   The relative or absolute pathname of the linkable-object file is enclosed in quotation marks.

3. *Search rules for linkable-object files*. As your spreadsheet program compiles, the INTERVIEW's filing system is searched for the linkable-object files referenced in *#pragma object* directives.

   - If the referenced LOBJ filename begins with *FD1/*, *FD2/*, or *HRD/*, the INTERVIEW interprets it as the absolute pathname and makes only that one search.

   - Pathnames beginning with a / indicate that the root directory on each drive should be the beginning point of the search. The drives are searched in the following order: current drive, FD1, FD2, and HRD.

- Otherwise, the name may be a one-word filename, or a relative pathname which includes the directories leading to the file. The highest directory in a relative pathname must reside in the current directory or in one of the */lib* subdirectories. The following directories—and only the following directories—are searched, in the order given:

  1. current directory on the current drive (indicated on the File Maintenance screen)
  2. */usr/lib* on the current drive
  3. */sys/lib* on the current drive
  4. *FD1/usr/lib*
  5. *FD2/usr/lib*
  6. *HRD/usr/lib*
  7. *FD1/sys/lib*
  8. *FD2/sys/lib*
  9. *HRD/sys/lib*

If the pathname is not located in any of these directories, the program will not compile and an error message will be returned to the operator.

4. *How #pragma object works.* When the source of code for the Compile command is FILE, the LOBJ which results usually defines user-created routines. These routine definitions may be "linked," or combined, as needed with your spreadsheet program. This means that routines called within your active program do not always have to be defined on the Protocol Spreadsheet or in *#include* files.

   NOTE: An LOBJ file may also contain *#pragma hook* directives. See Section (D) below. If a *#pragma object* directive references an LOBJ file which contains *#pragma hook* directives, the "hooks" within that file are ignored. Since Compile SPREADSHEET always generates *#pragma hooks*, use the OBJECT block-identifier to reference the resulting LOBJ file.

   (a) *Referenced linkable-object files searched for routine definitions.* If a spreadsheet program calls a routine for which no definition is provided, the LOBJ files referenced in *#pragma object* directives are searched in the order in which they appear on the Protocol Spreadsheet. If a routine is defined in more than one referenced LOBJ file, the definition in the first LOBJ file listed on the Protocol Spreadsheet will be used.

   If the routine definition is not found in the spreadsheet program or in any referenced linkable-object file, the compilation will abort. When you go to the Protocol Spreadsheet and look for error messages, the routine name will appear as an unresolved reference.

(b) *Compiled routine definition combined with compiled spreadsheet.* When the routine's definition is located, the compiled code is copied from the LOBJ file and combined with the compiled code of the spreadsheet program.

Routine definitions in an LOBJ file may reference additional routines not defined within the same file. If these indirectly-referenced routines also are not defined on the Protocol Spreadsheet, the LOBJ files are searched again.

Routine definitions containing the *static* attribute are local to the LOBJ file. A *static* routine will be copied from the file only if it is included in the definition of another routine.

NOTE: Use *#pragma object* directives in your active spreadsheet program only. Do no incorporate them in code that will be compiled and saved as an LOBJ file. Although the code will compile, no search fro routine definitions in referenced LOBJ files will be performed.

(c) *Efficiently uses memory.* Using *#pragma object* to reference routines in linkable-object files, assists in using the INTERVIEW's memory and spreadsheet buffer efficiently.

- Only the definitions for routines actually called within the current spreadsheet program are copied into memory from the LOBJ file. All other code within the file is ignored.

- When commonly utilized routines are defined in linkable-object files, space in the spreadsheet buffer otherwise dedicated to this purpose can be used for additional programming.

- Since the code in LOBJ files has already been compiled, the INTERVIEW can support a larger program without a corresponding increase in compilation time.

NOTE: Additional *#pragma* preprocessor directives utilized by the INTERVIEW are discussed in other sections of the manual. Refer to Section 61 on Display Window and Trace, for example, for information on the *#pragma tracebuf* directive. Except for *#pragma hook* (below), these other *#pragmas* should be part of the active spreadsheet program, not part of a linkable-object file.

(D) #pragma hook

The *#pragma hook* directive allows compiled C code within a referenced linkable-object file to be automatically combined with the compiled code of an active spreadsheet program. There are eight types of *#pragma hook* directives—hook_types zero through seven. All types may be system-generated during the Compile operation when the source of code is SPREADSHEET, but the resulting linkable-object file always contains at least one hook_type zero.

The programmer also uses hook_type zero (*#pragma hook 0*). For this reason, *#pragma hook 0* will be the focus of the following discussion. The primary purpose of *#pragma hook 0* is to "force" a routine to be called and executed as part of a spreadsheet program, even though no explicit call to the routine is made on the Protocol Spreadsheet. The spreadsheet program *may* also call the routine, but keep in mind that it will be executed twice—once because of the call on the spreadsheet and once because of the call made via the *#pragma hook 0* directive.

1. *Format*. Create hooks on the Protocol Spreadsheet and then write them to a file using the WRITE/U editor command. Before typing your hook on the spreadsheet, press EDIT to prevent the editor from placing a strike-through over the text.

   The format for the *#pragma hook 0* directive is as follows:

   #pragma hook hook_type "routine_name();"

   Follow the directive with a space and enter a decimal (not hexadecimal) constant to identify the *hook_type*.

   After the hook_type, enter another space, and then the *hook text*—C code that calls the routine you want combined with your spreadsheet program. The call to the routine is placed inside quotation marks and includes required syntax—parentheses for the arguments and a semi-colon to complete statement punctuation.

   NOTE: Task names are always local to a linkable-object file and never *directly* copied from it. The hook text, therefore, cannot reference a task. The rule for exporting tasks from a linkable-object file is to let the *#pragma hook 0* directive call a routine which starts the task(s). See Section 5. following and Section 52 for examples.

   More than one *#pragma hook 0* directive may be present in a single LOBJ file, but each directive calls only one routine.

2. *Routine definitions.* Typically, the definition for the routine called in the directive is located within the same linkable-object file. It may, however, be in another LOBJ file as long as both files are referenced via OBJECT block-identifiers on the Protocol Spreadsheet.

   The definition of the hook-text routine may also reference a task (which must be defined in the same file) or it may reference additional routines not defined within the same file. The rules in Section (C) above for indirectly referencing routines apply.

   Definitions for most of the *extern* routines included in this manual are not strictly required.

3. *Accessing hooks.* If you want the hook text combined with your program, use the OBJECT block-identifier to reference the LOBJ file. If you use the *#pragma object* directive to reference the file, the "hooks" within that file will be ignored.

4. *Hooks are added to task list of program main.* As your program compiles, referenced linkable-object files are searched for hooks. When a hook_type zero directive is found in the file, the hook text is automatically added to the bottom of the task-list in the top-level *main*. If a referenced LOBJ file contains more than one "hook," they will be added to the task list in the order in which they appear in the file.

   NOTE: The order of tasks and hooks in the task-list indicates the order in which *main* initiates tasks and executes hook routines. It does not necessarily indicate the order in which the actions in tasks or hooks are processed.

5. *Execution of hooks.* Recall that the *main* function is system-created during compilation. Refer to Section 52, Program Main. Because *main* simply initiates the execution of each task listed, the (hook-text) routine essentially runs concurrently with the tests in your spreadsheet program.

   Since the hook text is a routine, and not a task, it must actually be executed by *main*, not simply started. The definition of the routine determines when, or whether, any subsequent hooks will be executed by *main*.

   - If the routine's definition references a task, as in the example below, *main* returns quickly, leaving the routine to execute the task. Then *main* begins execution of the next hook in the task list.

     ```
     #pragma hook 0 "example();"
     extern fast_event fevar_time_of_day;
     extern volatile unsigned short crnt_time_of_day;
     ```

```
task
 {
  main()
   {
    state_alarm_at_one:
    waitfor
     {
       fevar_time_of_day && (crnt_time_of_day == 1300):
        {
         sound_alarm();
        }
     }
   }
 } example_task;
example()
 {
  example_task();
 }
```

- If the routine's purpose is *not* to start a task (or tasks), then *main* has to execute all the code. The more code there is, the longer it will be before *main* can return to execute the next hook.

  If the definition includes a *waitfor*, as in the following example, any subsequent hooks will never get executed. Instead, *main* will continue to wait for the specified event.

```
#pragma hook 0 "example();"
extern fast_event fevar_time_of_day;
extern volatile unsigned short crnt_time_of_day;
example()
 {
  waitfor
   {
    fevar_time_of_day && (crnt_time_of_day == 1300):
     {
        sound_alarm();
     }
   }
 }
```

56.5 Data Types

(A) Precisions

When a variable is declared, the compiler allocates space in memory according to the *type* declaration that precedes the variable name. There are three sizes (or *precisions*) of data allowable in 80286 memory, and three corresponding data types. A *char* is allotted one byte of memory. A *short* is given two bytes, while a *long* reserves four bytes of memory. Shorts and longs are varieties of *int* or integer, and the type descriptions *short int* and *long int* are permitted. The type *int* used by itself is the same as *short int*.

(B) Signed and Unsigned Types

All three precision types may be *signed* or *unsigned*. Signed and unsigned data types are *stored* identically, but treated differently in arithmetic operations. Specifically, they differ in the way they undergo type conversion, comparison, division, and right shifting.

1. *Type conversion.* The following declarations store the same value in memory:

```
signed char a = -6;
unsigned char b = -6;
```

In both cases, the byte stored in memory will be the two's complement of 00000110, or 11111010. (The two's complement is the one's complement + 1.) This bit pattern translates as hex *fa* or ASCII *z*. The *displayf* routine in the following program will write two *z*'s to the screen:

```
{
   signed char a = -6;
   unsigned char b = -6;
}
   STATE: data_type
      CONDITIONS: ENTER_STATE
      ACTIONS:
      {
         displayf ("%c%c" , a, b);
      }
```

When you lengthen the *chars* to *shorts*, however, they behave differently. The *unsigned char* is left-padded with zeroes. The *signed char*, having a leftmost bit equaling 1, is left-padded with ones. This left-padding with ones is called "sign extension."

A *char* is converted to a *short* automatically when a *%d*, *%u*, or *%x* conversion is applied to it, so the following example illustrates the difference between the conversion of *signed* and *unsigned* types:

```
{
   signed char a = -6;
   unsigned char b = -6;
}
   STATE: data_type
      CONDITIONS: ENTER_STATE
      ACTIONS:
      {
         displayf ("%x%x ", a, b);
      }
```

The variable *a* will be seen to extend to hex *fffa*, which is *fa* left-padded with eight ones. The unsigned variable *b* will have been extended by eight zeroes and will appear unchanged as *fa*.

If the *%x* conversion specifiers in the example above are replaced by *%d*, the resulting signed-decimal conversion will show *a* equaling -6, *b* equaling 250. The *signed char* will have survived the type-lengthening with its original negative value intact.

Because they can be lengthened without changing their values, signed variables should be used for any arithmetic operations. Other differences between signed and unsigned variables, not reflected in Table 56-1, are the following:

2. *Comparison*. If the leftmost bit of a signed variable is 1, then the variable has a negative value and the expression *variable* > *0* is false. If the leftmost bit of an unsigned variable is 1, the variable is positive and *variable* > *0* is true.

3. *Division and modulus*. If the leftmost bit of a signed variable is 1, the two's complement of the variable rather the stored value will be used in any division or modulus operation.

4. *Right shifting*. When a right-shift (>>) operator is used on a signed variable, a 1-bit is shifted in at the left. When the same operation is performed on an unsigned variable, a 0-bit is shifted in.

Table 56-1 shows the ranges of values that are produced by *displayf* and *printf* routines when the valid conversion specifiers—*%c*, *%d*, *%ld*, and so on—are applied to the various *signed* and *unsigned* data types. Frequently it makes no difference whether a variable is declared as *signed* or *unsigned*. When a variable undergoes type conversion, however, as in the case of a *char* given a decimal or hex conversion, there is a significant difference.

Table 56-1
Data Types: Ranges of Values Displayed and Printed

type	char conversion (%c)	signed decimal conversion		unsigned decimal conversion		hex conversion	
		short (%d)	long (%ld)	shor%u)	long (%lu)	short (%x)	long (%lx)
char[1]	␀ to ▒	0 to 255	–	0 to 255	–	0 to ff	–
signed char[1]	␀ to ▒	-128 to 127	–	0 to 127 and 65408 to 65535	–	0 to 7f and ff80 to ffff	–
unsigned char[1]	␀ to ▒	0 to 255	–	0 to 255	–	0 to ff	–
int	–	-32768 to 32767	–	0 to 65535	–	0 to ffff	–
signed int	–	-32768 to 32767	–	0 to 65535	–	0 to ffff	–
unsigned int	–	-32768 to 32767	–	0 to 65535	–	0 to ffff	–
short	–	-32768 to 32767	–	0 to 65535	–	0 to ffff	–
signed short	–	-32768 to 32767	–	0 to 65535	–	0 to ffff	–
unsigned short	–	-32768 to 32767	–	0 to 65535	–	0 to ffff	–
long	–	–	-2147483648 to 2147483647	–	0 to 4294967295	–	0 to ffffffff
signed long	–	–	-2147483648 to 2147483647	–	0 to 4294967295	–	0 to ffffffff
unsigned long	–	–	-2147483648 to 2147483647	–	0 to 4294967295	–	0 to ffffffff

[1] Through "integral promotion," *char* is converted automatically to *int* in a *%d*, *%u*, or *%x* conversion.

(C) Static Storage Class

A variable must be of the *static* storage class to pass its value into a *waitfor* statement. Declarations at the Program, Layer, or Test level (Level 1 in the source code diagram in Figure 52–4) are *static* even if they are not explicitly declared so. The same is true of a character array initialized by a string (see Section 56.7).

A variable that is initialized at the State level must be declared as *static* by the programmer if the initialized value is to be used inside a *waitfor*.

The following program will display a value of 8 on the prompt line when the operator presses the spacebar:

```
STATE: pass_initialized_value
{
   static int initialized = 8;
}
   CONDITIONS: KEYBOARD " "
   ACTIONS:
   {
      displayf ("%d ", initialized);
   }
```

If you removed the word *static* from the declaration, the initialized value would not be passed into the condition clause and the program would display 0 or a "garbage" number instead of 8.

56.6 Operator Precedence

In an expression with more than one operator, operations are prioritized according to the ranking of operator precedence in Table 56-2. The operator with the highest precedence is at the top of the table. Precedence decreases as you move down.

Consider this example:

```
STATE: precedence
{
   int a;
   a = 3 * 4 + 2;
   displayf ("%d", a);
}
```

Because multiplicative operators (*, /, and %) have higher precedence than additive operators (+ and –), the *3 * 4* operation is performed first. Then 2 is added to the product of 3 and 4, and finally the sum is assigned to the variable *a*. (Assignment operators have very low precedence.) The result of the program is that *a* is displayed as 14. Compare this example:

```
STATE: precedence
{
   int a;
   a = 3 * (4 + 2);
   displayf ("%d", a);
}
```

Table 56-2
Operator Precedence[1]

Operator	Type of Operator	Associativity
()	primary expression	left to right
[] . -> ++ --	postfix	left to right
++ -- sizeof & * + - ~ !	unary	right to left
(type)	cast	left to right
* / %	multiplicative	left to right
+ -	additive	left to right
<< >>	bitwise shift	left to right
< > <= >=	relational	left to right
== !=	equality	left to right
&	bitwise AND	left to right
^	bitwise exclusive OR	left to right
|	bitwise inclusive OR	left to right
&&	logical AND	left to right
||	logical OR	left to right
? :	conditional	right to left
= *= /= %= += -= <<= >>= &= ^= |=	assignment	right to left
,	comma	left to right

[1] Operators on the same line have the same precedence; rows are in order of decreasing precedence.

Here the additive operation is performed before the multiplicative, since the parentheses that denote a primary expression (see Table 56-2) have the highest precedence of all. The result of this program is that decimal 18 is displayed.

Given operations with the same precedence, left-to-right or right-to-left "associativity" (see the right column in Table 56-2) indicates which is performed first. This order of processing is significant for an expression such as *36 / 6 / 2*, where the associativity is left to right.

Associativity is very important in assignment operations, which are always interpreted in a right-to-left direction. Consider this example:

```
STATE: right_to_left_associativity
{
  int a = 4;
  int b = 1;
  a = b;
  displayf ("%d", a);
}
```

The result of this program is that 1 is displayed, not 4. Right-to-left associativity also explains why the following program does not compile.

```
STATE: right_to_left_associativity
{
   int a = 3;
   3 = a;
   displayf ("%d", a);
}
```

A constant never can have a value assigned to it, even if the value equals the constant.

56.7 Strings

A string is a sequence of characters enclosed in double quotes. This is an example of a string:

```
"hello"
```

A string is an expression of the type *pointer*, and may be used anyplace in the program that is appropriate for a pointer. For example, a pointer is appropriate as the argument of a *displays* routine:

```
displays ("hello");
```

The string in this statement does two things during compilation: it writes the character string "hello" in memory, and it *points* to the first character in the string. The string "hello" becomes a 4-byte address that you can examine by displaying it as a *long* hexadecimal:

```
displayf ("%lx", "hello");
```

(A) Using a String to Initialize an Array

Note that the pointer represented by "hello" in the examples above is not stored anywhere and therefore can be used only once. The string pointer "hello" could have been stored as a pointer to the first character in an array, as in this example:

```
char string_array [] = "hello";
displayf (string_array);
```

Stored in this manner, the pointer can be used repeatedly.

An array like *string_array* that has been "initialized" by a string shares many of the traits of standard arrays, but it has unique characteristics as well.

1. *Data type.* A string may only initialize an array whose elements are of the type *char*.

2. *Null termination.* A string is always terminated by a null character. This null terminator is appended by the compiler, not the programmer.

3. *Size*. All arrays must declare their size, in any of three ways. The programmer may declare the length inside of brackets, as in this example:

```
char array [5];
```

Or he may leave the brackets empty and provide a list of initializers, inside of curly braces, from which the compiler can determine the size of the array:

```
char initializer_list_array [] = {'h', 'e', 'l', 0x6c, 'o'} ;
```

The third method of indicating size is to leave the brackets empty and initialize the array with a string, as in our original example of a string initializer:

```
char string_array [] = "hello";
```

The compiler will add a terminating null-character to this string, and calculate an array size of six. To verify that the compiler counts one more character than the user has entered, you may try the following test. Note that the *sizeof* operator will return the length of any array:

```
STATE: display_size_of_string
{
   char string_array [] = "hello";
   int compiler_count = sizeof(string_array);
   displayf ("%d",compiler_count );
}
```

4. *One-dimensional array*. Whereas arrays in general can be multidimensional, a string-initialized array always has one dimension.

(B) Valid Strings

1. *ASCII and control*. With a few exceptions, all ASCII characters, including control characters, are valid in a string. The exceptions are ␀, ␊, ", and \. These characters are liable to be misinterpreted by the compiler. Null (␀) and linefeed (␊) will be taken to indicate a new logical line in the program. Double-quote (") will be mistaken for the end of the string. Backslash (\) will be misinterpreted as the start of an escape sequence.

   If one of these characters is included in a string, the program may not compile. If not, you will be returned to the Protocol Spreadsheet. The following message will be displayed for nulls or linefeeds: *"Error 718: Newline inside string."* For quotation marks, the message is *"Unclosed AR "C" region."* Depending on their placement in the string, backslashes may or may not generate an error. Even when compilation succeeds, however, they will not be interpreted correctly.

Table 56-3
C String Non-Literals

Non-literal	Meaning	ASCII character	Hex character
\a	bell	BL	07
\b	backspace	BS	08
\f	form feed	FF	0C
\n	linefeed †	LF	0A
\r	carriage return	CR	0D
\t	horizontal tab	HT	09
\v	vertical tab	VT	0B
\'	single quote	'	27
\"	double quote †	"	22
\\	backslash †	\	5C
\###	octal representation	any ASCII character	00 – FF
\x###	hex representation	any ASCII character	00 – FF

† These characters require non-literal entries in INTERVIEW strings. The others may be entered as ASCII characters, non-literals, or hexadecimal characters.

2. *Non-literals*. Most characters in strings are interpreted literally. Each of the invalid characters listed above, therefore, needs a non-literal representation. Non-literals are preceded by a backslash. The compiler converts these non-literals to their one-byte numeric value.

   To include a null (or any ASCII) character in a string, use the octal or hexadecimal representation shown in Table 56-3. Hex and octal numbers take up to three digits, so use leading zeroes if necessary. Otherwise, a subsequent digit may be interpreted as part of the value. Suppose, for example, you want to create the string "NUabc". You initialize an array as follows:

   ```
   char string[] = "\x0abc";
   ```

   The string will be stored as "+c" (hexadecimal characters AB 63). The correct declaration was *char string[] = "\x000abc"*. In octal, the null would be written *000*.

   Please note that a string that has a null character somewhere other than at the end will be difficult to display or print completely. Display and print routines that take strings as input typically begin at the pointer position and continue until they encounter a terminating null. If, as in the last example, a null is encountered at the beginning of the string, execution of the routine will end before anything has been displayed or printed.

   Provide precision to the *%H* conversion specifier to override null termination of a string while displaying a string in hex: see Section 60.3(C).

3. *Constants.* Spreadsheet constants may be included in strings. An example of a spreadsheet constant is the fox message represented as «FOX». See Section 25 on Constants.

   The C translator expands constants both inside and outside of C regions before the code is preprocessed.

4. *Hexadecimal characters.* ASCII characters, including the control characters, may be entered in strings as hexadecimal characters via the [HEX] key. Hex representation is considered literal. That is, you may not enter ASCII characters which require non-literal representation in strings as hexadecimal characters. The sequence of characters comprising a non-literal may be entered as hexadecimal characters. Double backslash (\\), for example, may be entered as $^{5}c^{5}c$.

(C) String Routines

There are several C routines in the INTERVIEW that display or print strings. See Section 63 on "Print" and Section 60 on "Display Window and Trace" for detailed descriptions of the *prints*, *displays*, and *traces* routines, as well as other display and print routines that use the *%s* conversion specifier.

There is also a pair of routines, *index* and *rindex*, that search inside of strings for particular characters. These routines are defined (with examples) in Section 67.

56.8 Recommended Sources

The following sources provide accurate, in-depth information on C Programming Language:

1. *ANSI Document X3J11/86-098.* Proposed American National Standard for Information Systems—Programming Language C.

   NOTE: When approved, the number for the ANSI document will change to: *ANSI Standard X3.159-198X.*

2. Darnell, Peter A., and Margolis, Philip E. *Software Engineering in C.* New York: Springer-Verlag, 1988.

3. Harbison, Samuel P., and Steele, Guy L., Jr. *C: A Reference Manual.* 2d ed. Englewood Cliffs: Prentice-Hall 1987.

4. Kernighan, Brian W., and Ritchie, Dennis M. *The C Programming Language.* 2d ed. Englewood Cliffs: Prentice-Hall, 1988.

57 Variables

57.1 Creating or Accessing C Variables

Softkey–selectable programming "tokens" entered by the user on the Protocol Spreadsheet are translated automatically into C during the initial compiler phases after [RUN] is pressed. (Then the C code in turn is compiled into object code.) The C variables used by the translator are documented throughout this volume.

C regions available to the user at every level of spreadsheet programming (see Section 53) provide direct access to these variables.

An example of a user–accessible variable is *keyboard_new_key*, used in the following program to sound an alarm whenever any ASCII–keyboard key is pressed.

```
{
  extern fast_event keyboard_new_key;
}
     STATE: anykey
        CONDITIONS:
           {
              keyboard_new_key
           }
        ACTIONS: ALARM
```

The C regions also allow the user to work with variables of his own creation.

Here is an example of a user–created variable named *minutes* that is used to count minutes elapsed since the beginning of Run mode. The C program displays this "counter" on the prompt line of the Run–mode screen.

```
{
  extern fast_event fevar_time_of_day;
  short minutes;
}
     STATE: run_mode_minutes
        CONDITIONS:
        {
           fevar_time_of_day
        }
        ACTIONS:
        {
           minutes++;
           pos_cursor (0,0);
           displayf ("Duration of run = %4d minutes", minutes);
        }
```

The first C region in the example "declares" the variables *fevar_time_of_day* and *minutes*. The first of these variables is an event variable that is built into the system software. All event variables in an active State-block are polled constantly. Once every minute, *fevar_time_of_day* returns true.

The second variable, *minutes*, is created by the program itself—that is, by the user. The declaration in effect creates the variable: it causes 16 bits in memory ("short" = 16 bits) to be dedicated to information stored under the name *minutes*.

The second C region in the example is placed inside the Actions block. The statement *minutes++* causes the value that is stored in the 16 bits dedicated to *minutes* to increment. The function *pos_cursor (0,0)* places the cursor in the leftmost column on the second line of the display screen (the Prompt line). The *displayf* function writes a text message to the display screen, beginning at the current cursor position. In the text message itself, "%" will be replaced by the current value of the variable *minutes*. "4" means that four columns on the screen will be dedicated to the value, and "d" means that the value will be expressed in a decimal number.

57.2 Declaring Variables

Declare your variables and routines in a C region, delimited by curly braces { and }, at the top of your program or at the top of a Constants, Layer, Test, State, or Actions block. Declare a variable preceded by its type descriptors and followed by a semicolon, as in these examples:

```
{
    extern fast_event keyboard_new_key;
    extern fast_event keyboard_new_any_key;
    extern fast_event fevar_time_of_day;
    short minutes;
}
```

A variable may have its scope limited to a particular Test, State, or Actions block. A variable also may be redeclared at different levels. (In software revision 5.00 or earlier, it may not be redeclared at the same level.) Given more than one valid declaration, the lower or *nearer* one applies.

The rules governing the placement of variable declarations are laid out in detail in Section 53.5(A).

(A) Naming Variables

1. *Legal names*. The first letter of a variable name may be either a letter or an underscore. Following characters may be letters, numbers, underscores, or dollar signs.

   Reserved words (indicated in **boldface** type in Appendix K) may not be used as variable names.

2. *Naming conventions*. Generally speaking, variables that begin with *dte_* or *dce_* are used by the software to test DTE and DCE conditions. Variables that begin *rcvd_* are used to test RECEIVE (or RCV) conditions. Variables that begin *m_* are used by the layer packages to construct the protocol traces.

(B) Modifiers

1. *Data type*. The data type for each variable precedes the variable name in the declaration. All standard data types except *float* are supported in the INTERVIEW 7000 Series. Standard data types and their sizes and ranges are given in Table 56-1.

2. *Preassigned modifiers*. When you declare a user-accessible external variable, be sure to use the modifiers which precede the data type for that variable as listed in variable tables throughout this volume.

57.3 Comparing a Variable to a Value

User-accessible and user-created variables may be tested as part of any standard C expression.

The following is an example of a user-invented variable called *anykey* that is declared with a default value of zero, incremented by the operator pressing any ASCII-keyboard key, and checked for a value of 3 by an *if* statement after each depression of a key. An alarm will sound on the third keystroke.

```
{
   extern fast_event keyboard_new_key;
   short anykey;
}
     STATE: press_key
        CONDITIONS:
        {
           keyboard_new_key
        }

        ACTIONS:
        {
           anykey++;
           if (anykey == 3) sound_alarm ();
        }
```

The next example uses a built-in, user-accessible variable called *crnt_time_of_day* and checks it for a particular value. This 16-bit variable stores the time of day in hours and minutes. The Condition in the program (the event variable *fevar_time_of_day*) is true once per minute. The Action each time the condition is true is to check *crnt_time_of_day* for a value of 1129. At 11:29 AM, an alarm will sound.

```
{
  extern fast_event fevar_time_of_day;
  extern volatile unsigned short crnt_time_of_day;
}
     STATE: alarm_clock
        CONDITIONS:
        {
           fevar_time_of_day
        }
        ACTIONS:
        {
           if (crnt_time_of_day == 1129) sound_alarm();
        }
```

57.4 Checking a Variable in a *Waitfor* Clause

Please note that the following variation on the preceding example does not produce the same result. *The alarm will never sound* if this version of the program is run:

```
{
  extern volatile unsigned short crnt_time_of_day;
}
     STATE: alarm_clock
        CONDITIONS:
        {
           crnt_time_of_day == 1129
        }
        ACTIONS:
        {
           sound_alarm();
        }
```

Note that the time-of-day condition that was lodged in an *if* statement in the previous example has now been placed in a Conditions block. Conditions blocks on the Protocol Spreadsheet are converted to *waitfor* clauses (see Section 53.3), not *if* statements, when the program is translated automatically into C coding.

Waitfor clauses work very differently from *if* statements and other conditional control structures in C.

(A) Event vs. Nonevent Variables

Two kinds of variables may be used inside of these *waitfor* clauses—event variables and nonevent variables. When a state is active, event variables in that state are checked regularly during routine polling by the CPU. When an event variable (such as *fevar_time_of_day*) is polled and returns a value of true, conditional statements containing nonevent variables (such as *crnt_time_of_day*) also are checked for truth or falsity. *In the absence of an event variable being polled and returning a value of true, a statement about a nonevent variable inside of a Conditions block* (waitfor *clause) never can be true.*

Since there is no event variable in the Conditions block (*waitfor* clause) above, the nonevent variable *crnt_time_of_day* is never even checked.

(B) Translation of Softkey Tokens Into Variables

You could have written the "alarm clock" program using only softkey entries, as follows:

```
STATE: alarm_clock
   CONDITIONS: TIME 1129
   ACTIONS: ALARM
```

In this case, the C translator will convert the Conditions block into a *waitfor* clause that uses the event variable *fevar_time_of_day* to check the nonevent variable *crnt_time_of_day* once a minute. Here is the translator's version of the Conditions and Actions blocks:

```
{
   waitfor
   {
      fevar_time_of_day && (crnt_time_of_day == 1129):
      {
         sound_alarm();
      }
   }
}
```

(C) Example of A Nonevent Condition "Waiting For" An Event

The next example illustrates the interplay of event variables and nonevent variables in a *waitfor* clause.

```
{
   extern fast_event keyboard_new_key;
   short anykey;
}
      STATE: press_key
         CONDITIONS:
         {
            keyboard_new_key
         }
         ACTIONS:
         {
            anykey++;
         }
         CONDITIONS:
         {
            anykey == 3
         }
         ACTIONS: ALARM
```

This program looks similar to a previous one in which the operator hit three keys and the alarm sounded. Here, however, the alarm does not sound until the fourth keystroke. The variable *anykey* begins the test at zero, and increments (*anykey++*) with every keystroke. But remember what a condition such as *anykey == 3* in a *waitfor* clause really means. It means that the condition will be true when the variable equals three and an event (such as a keystroke) occurs that causes the variable to be checked. On these terms, the condition is not satisfied until the fourth event.

(D) User-Created Event Variables

The user can create his own event variable simply by declaring a new variable with the modifiers *extern event*. Once the event variable has been declared, he can use the *signal* function to indicate that the event has occurred. Here is an example of an event variable called *check_number* that causes the nonevent variable *number* to be checked—and sounds the alarm when the value of *number* satisfies the condition.

```
{
  short number = 3;
  extern event check_number;
}
    STATE: user_created_event
    {
      signal (check_number);
    }
      CONDITIONS:
      {
        check_number && (number == 3)
      }
      ACTIONS: ALARM
```

(E) Rules and Cautions

To sum up the discussion of event and nonevent variables, here are a few rules of thumb:

1. *If* statements, *for* loops, *while* loops, and other conditional control structures may not be used in Conditions blocks (that is, in *waitfor* clauses). They may be used in State blocks, above (or in the absence of) Conditions blocks; and they may be used in Actions blocks.

   (Placing an *if* statement at the top of the State block, above any *waitfor* clauses, is how the translator converts ENTER_STATE softkey conditions into C.)

2. Event variables are designed for use in Conditions blocks (*waitfor* clauses) only. It makes no sense to use an event variable in an *if* statement, *while* loop, etc., since there is no possibility that the event will be true at the precise moment the statement is being processed.

3. A Conditions block (*waitfor* clause) that lacks an event variable can never come true.

One other word of caution about the importance of event variables: please note that the following program will not sound the alarm even if the operator presses a key while the time is 11:29 AM.

```
{
   extern fast_event keyboard_new_key;
   extern volatile unsigned short crnt_time_of_day;
}
      STATE: alarm_clock
         CONDITIONS:
         {
            keyboard_new_key && (crnt_time_of_day == 1129)
         }
         ACTIONS: ALARM
```

The reason this program doesn't "work" is that all variables begin Run mode at zero. Often a particular event variable must return true before a particular nonevent variable will be updated. The nonevent variable *crnt_time_of_day* is updated only when the event variable *fevar_time_of_day* is entered in the *waitfor* clause and returns true. In the example above, the operator pressing the key will cause *crnt_time_of_day* to be checked; but in the absence of *fevar_time_of_day*, the value of *crnt_time_of_day* remains always at zero.

57.5 Checking and Displaying Equivalent Values of a Variable

Variables may be checked and displayed as octal, decimal, hexadecimal, and ASCII–character values. Decimal comparison and display is the default.

(A) Checking Equivalent Values

To compare a variable to an octal value, precede the value with a zero (0). No prefix is necessary to make a decimal comparison. To compare a variable to a hexadecimal value, precede the value with 0x or 0X. To check whether a variable matches an ASCII character, enter the character in between single quotes.

The alarm will sound in the example below, since all of the values entered to the right of the equal signs are equivalent.

```
{
   char foxtrot = 'f';
}
      STATE: compare_equivalent_values
      {
         if ((foxtrot == 0146) && (foxtrot == 102) && (foxtrot == 0x66) && (foxtrot ==
         'f')) sound_alarm();
      }
```

Note that the data type *char* in the declaration simply means that the variable is composed of 8 bits. The designation *char* does not say anything about the comparison mode or the display mode. (Data types *short* and *int* = 16 bits; *long* = 32 bits.)

(B) Displaying Equivalent Values

Variables may be displayed in a variety of data formats via the *displayf* function. The full set of display conversions is given in Table 61-7. The program below generates a representative sample of display formats. When the program is run, the prompt line on the display screen will look like this: 152 106 6a 6A j 6A.

```
{
    char juliet = 'j';
}
        STATE: display_equivalent_values
        {
            displayf ("%o  %d  %x  %X  %c  %#u ", juliet, juliet, juliet, juliet, juliet,
            juliet);
        }
```

57.6 Isolating Bits from a Variable Value

Some variables are bit-oriented. That is, one bit (or perhaps a small field of bits) may have significance that is independent of the surrounding bit values. The variable *current_eia_leads* (refer to Table 60-1), for example, uses 7 bits to store the on/off status of seven separate EIA leads, plus an eighth bit to store the status of any lead that is patched to the UA input jack (see Section 10.3). If you want to check this variable to determine the status of DTR (for example) you need to determine whether the bit that represents DTR (the fifth bit from the right or the fifth least significant bit in the variable) is set to 1 (DTR off) or zero (DTR on). How can you isolate this bit from the surrounding bits in order to determine its status?

The tool for isolating a bit in a C variable is the "care mask," a group of bits (usually expressed in hexadecimal) in which the bit(s) under scrutiny is set to 1 and all other bits to zero. The care mask for DTR is 0x10 (or 16 in decimal notation). The binary version, 00010000, shows that only the DTR bit is set to 1. When this care mask is *and*ed (via the "&" operator) with the variable *current_eia_leads*, only two results are possible, depending on whether the DTR bit in *current_eia_leads* is 1 or 0.

With DTR on, suppose that the combination of all lead statuses gives *current_eia_leads* a value of e6 in hex—11100110 in binary. The effect of *and*ing this variable with the care mask for DTR will be as follows:

```
  11100110
& 00010000
  00000000
```

Now turn DTR off, and the result of the *and*ing will be this:

```
  11110110
& 00010000
  00010000
```

The seven "don't care" zeroes in the care mask guarantee seven zero-bits in the result (because 0 & 1 = 0 and 0 & 0 = 0). So the result of the *and*ing must be either 0 if the DTR bit is 0 (on), or hex 10 (decimal 16, binary 00010000) if the DTR bit is 1 (off).

This C program will detect DTR on:

```
{
   extern fast_event fevar_eia_changed;
   extern const volatile unsigned short current_eia_leads;
}
      STATE: check_dtr_on
         CONDITIONS:
         {
            fevar_eia_changed
         }
         ACTIONS:
         {
            if ((current_eia_leads & 0x10) == 0) sound_alarm();
         }
```

If you try to run this program, make sure of the following:

1. The Front-End Buffer Setup menu should be configured to buffer control leads.

2. If you are not connected to a device that provides clock, the Line Setup menu should be configured to provide internal clock. EIA leads are clocked through the front-end buffer before they reach the program logic.

3. After the program enters Run mode, use a single-wire patch cord to connect the +12V output pin on the test-interface module to the DTR lead. The alarm should sound as soon as the patch is made.

A slightly different condition inside of the *if* statement will detect DTR off:

```
if ((current_eia_leads & 0x10) == 0x10) sound_alarm();
```

The DSR bit is the fourth least significant bit in the *current_eia_leads* variable, so the care mask for DSR is 0x08 (binary 00001000). The following *if* statement will detect DSR on:

```
if ((current_eia_leads & 0x08) == 0) sound_alarm();
```

This *if* statement will detect DTR on and DSR on:

```
if ((current_eia_leads & 0x18) == 0) sound_alarm();
```

This *if* statement will detect DTR off and DSR on:

```
if ((current_eia_leads & 0x18) == 0x10) sound_alarm();
```

The last condition simply means that you care (1=care) about DTR and DSR and you want DTR to be 1 (off) and DSR to be 0 (on).

57.7 Pointing to an Address

Some routines require an address as input. The *displays* (display-string) routine, for example, requires a CPU memory address as its argument. When executed, the routine will begin to display characters that it finds at the specified address and at subsequent addresses, one by one, until a null is encountered. A memory address is four bytes (32 bits) and is declared as a *long*.

```
{
   long any_cpu_address;
}
   STATE: display_string
   {
      displays (any_cpu_address);
   }
```

Many of the important addresses needed by the user and by the program can be found inside of interlayer ("IL") message buffers. When BOP-framed data is monitored, it is copied automatically into IL buffers. Each time a frame is buffered, a data primitive is created automatically and the event variable *m_lo_ph_prmtv* is signaled. The segment number of the IL buffer is recorded in the variable *m_lo_ph_il_buff*. This segment number can be converted into an address.

Here, for example, is a program that looks for a DTE data packet, converts *m_lo_ph_il_buff* into a four-byte address that points to the first data position, and displays the data contents of the packet.

```
{
   long first_data_address;
   extern volatile unsigned short m_lo_ph_il_buff;
}
LAYER: 3
   STATE: display_data
      CONDITIONS: DTE DATA
      ACTIONS:
      {
         first_data_address = ((long) m_lo_ph_il_buff << 16) + 37;
         displays (first_data_address);
      }
```

The IL buffer is illustrated in Section 63 of this manual, and the procedure for converting the buffer-segment number into a memory address is explained in detail in Section 63.1(C). Briefly, we have cast the segment number (a *short*, 16 bits) into a *long* and moved the number over to its high-order position in the CPU address, sixteen bits to the left. Then we added 37 to the number to bypass the header information for the buffer (32 bits) and the frame and packet headers (5 bits).

Each address in memory stores 8 bits, so the second byte in the data field of the data packet would be *first_data_address + 1*, the fourteenth byte would be *first_data_address + 13*, and so on.

57.8 Creating a Character Pointer

For most of the variables in a C program, the address is not important to the user or to the program. The user does not need to know the address in order to declare the variable, perform operations on it, and compare its value to other values. In general, addresses of variables are solely the concern of the compiler.

In the case of a routine such as *displays*, the address is what is important. The value that is stored at the address is not so important, since the routine will go to the address and begin displaying the data whatever the value (as long as the value is displayable).

There is another kind of variable for which both the address and the value stored at the address are important. These variables are called pointers. The user creates a pointer by typing an asterisk (*) just following the data type in a declaration, as in this example:

```
char * packet_type_ptr;
```

The variable *packet_type_ptr* is a four-byte memory address just as *first_data_address*, declared as a *long* in the previous example, was a four-byte address—even though *packet_type_ptr* is declared as a *char*. The data type *char* preceding the asterisk simply means that the amount of data pointed to is eight bits.

Once you use an asterisk to declare the variable a pointer, you can access the address directly as *packet_type_ptr* or you can access the value stored at that address as * *packet_type_ptr*. A *displays* routine would accept *packet_type_ptr* as input, while a *displayc* or *displayf* routine would expect * *packet_type_ptr*.

With the X.25 personality package loaded at Layers 2 and 3 (via the Layer Setup screen), the following program goes to the memory location pointed to by *packet_type_ptr* and checks its value to determine whether the packet in the buffer is a Clear request.

```
{
   extern volatile unsigned short m_lo_ph_il_buff;
   extern event dte_packet;
   char * packet_type_ptr;
}
   STATE: search_for_dte_clear
      CONDITIONS:
      {
         dte_packet
      }
      ACTIONS:
      {
         packet_type_ptr = (void *) (((long) m_lo_ph_il_buff << 16) + 36);
         if (*packet_type_ptr == 0x13) sound_alarm();
      }
```

The pointer *packet_type_ptr* is a *char*, but you could just as easily point to a *short* (16 bits) or a *long* (32 bits). If you increment an address, you get the next address, 8 bits farther in memory. If you increment a *char* pointer, you also get the next address. If you increment a *short* pointer, you add two increments to the memory address. In effect you move the pointer two places. If you increment a *long* pointer, you move the pointer by four addresses, 32 bits.

In the example above, the integer *m_lo_ph_il_buff* is cast as a pointer (*void* *) after it is cast as a *long*. This is to avoid a compiler error ("Warning 31: Illegal implicit integer-to-pointer conversion") when the new value of *m_lo_ph_il_buff* is assigned to *packet_type_ptr*.

57.9 Pointing with Subscripts

When it is preceded by an asterisk (*), the pointer *packet_type_ptr* returns the character value that it points to, as we have just seen. Another way to return this value is to omit the asterisk and add a subscript: *packet_type_ptr[0]*. This mechanism allows you to access an array of values without moving the pointer.

For example, the transmission header ("TH") in a FID2 SNA information field is six bytes long. If you establish a pointer to the first TH byte (TH0), you can use subscripts to access any other byte in the field without moving the pointer. The following program checks the values of two bytes in the TH field (corresponding to "DAF" and "OAF") before freezing the data display and sounding an alarm.

```
{
   extern volatile unsigned short m_lo_ph_il_buff;
   char * th;
}
LAYER: 2
   STATE: th_pointer
      CONDITIONS: DTE INFO
         ACTIONS:
         {
            th = (void *) (((long) m_lo_ph_il_buff << 16) + 34);
            if ((th[2] == 5) && (th[3] == 1))
            {
               ctl_capture_td (0x10);
               ctl_capture_rd (0x100);
               sound_alarm ();
            }
         }
```

57.10 Creating a String

Strings are used in INTERVIEW programming mainly for transmissions and for messages to the operator ("prompts"). In the following program, the compiler decodes the string "QWERTYUIOP" from ASCII to hex, stores it in memory as a series of contiguous values, adds a null to it, returns the address of the first character, "Q," and then assigns this address to the variable *keyrow*:

```
{
  long keyrow;
}
STATE: assign_string_address_to_variable
{
  keyrow = "QWERTYUIOP";
}
```

The variable *keyrow* now is the four-byte address of "Q" in the string. You can see this address for yourself by using either *"QWERTYUIOP"* or *keyrow* as the argument in a *displayf* routine:

```
displayf ("%lx ", "QWERTYUIOP");
```

or

```
displayf ("%lx ", keyrow);
```

Either version will display a CPU address (hex 04400000) on the second line of the Run-mode screen.

The string can be displayed in a simple *displays* routine, since that routine expects a four-byte address as input:

```
displays ("QWERTYUIOP");
```

or

```
displays (keyrow);
```

If you want to access individual characters in the string, declare a pointer:

```
char * keyrow = "QWERTYUIOP";
```

With a pointer you can display the entire string or a single character—the seventh character, "U," in this example:

```
displays (keyrow);
displayc (keyrow[6]);
```

Declaring the string an array has virtually the same effect as declaring it a pointer:

```
char keyrow [] = "QWERTYUIOP";
```

The name of the array still is the address of the first character in the string and so may be used in a *displays* routine; and individual characters still may be specified by a subscript:

```
displays (keyrow);
displayc (keyrow[6]);
```

The only difference is that the array name is a constant whose value is assigned in a declaration and cannot be changed, while the pointer is a variable and may be incremented, assigned a new value, and so forth, while the program is running.

57.11 Comparing Strings

A string comparison in C may be conducted as follows. First, create a pointer in the manner described in Section 57.8, or else simply declare one of the pointers to line data that is provided in the set of user-accessible variables. Example: *extern volatile unsigned char * m_packet_ptr.*

Next, create an array that represents the search string you will try to match against the line data. For example:

```
char search_string [] = "\xa";
```

Create a trigger to look for a line event (such as the event variable *dte_packet*) that will initialize the pointer.

```
{
   extern volatile unsigned char * m_packet_ptr;
   char search_string [] = { 0x10, 0x04, 0x0b };
   extern event dte_packet;
}
LAYER: 3
     STATE: match_packet_string
        CONDITIONS:
        {
           dte_packet
        }
```

Compare the pointer-value with the first element of the search string. If a match is found, increment the pointer and compare the new value to the second element of the search string; and so on. If a match is found for every element of the string, take an appropriate action.

```
ACTIONS:
{
   if (search_string [0] == * m_packet_ptr)
   {
      m_packet_ptr ++;
      if (search_string [1] == * m_packet_ptr)
      {
         m_packet_ptr ++;
         if (search_string [2] == * m_packet_ptr) sound_alarm ();
      }
   }
}
```

Here is the same Actions block, only this time the variable *element* replaces the numeral in the subscript to *search_string*, and the same variable is added as a subscript to *m_packet_ptr*. This coding may be modified easily for any length string. For a 9-byte string, for example, simply change the 3 in the *if* statement to 9.

```
ACTIONS:
{
    element = 0;
    while (search_string [element] == m_packet_ptr [element])
    {
        if (search_string[element++] == 3)
        {
            sound_alarm();
            break;
        }
    }
}
```

57.12 Accessing a Variable Inside of a Structure

A structure is a mechanism that makes repetitive declarations of similar variables unnecessary. For example, there are twelve variables associated with any given counter created in the program. One variable is the current value of the counter, one is the last sampled value, another is the highest sampled count, another the total of all the sampled values, another the number of samples taken, and so forth. If the user creates four counters via the spreadsheet softkeys, the C translator does not declare 48 separate variables (4 x 12). Instead the translator declares a structure for counters—called *counter_struct*—that declares each of the twelve variables once, as follows:

```
{
  struct counter_struct
  {
      unsigned long current;
      unsigned long last;
      unsigned long maximum;
      unsigned long minimum;
      unsigned short sample_count;
      unsigned long total_high;
      unsigned short total_low_low;
      unsigned short total_low_high;
      unsigned short out_of_range;
      unsigned short changed;
      unsigned long prev;
      unsigned long old;
  };
```

Then the translator declares each of the user's four counters as having the structure *counter_struct*:

```
struct counter_struct dte_good_bcc, dte_bad_bcc, dce_good_bcc, dce_bad_bcc;
```

In effect the translator has declared all 48 variables. Suppose the user wants to access one of these variables. He may wish to display the total value of a counter whose current value no longer is the total value (since the counter may have been sampled—and therefore cleared—several times). As long as the total is less than 65,536, the entire number will reside in the seventh variable in the *counter_struct* structure, *total_low_low*. If the counter in question is *dce_good_bcc*, he will access this "total" variable under the name *dce_good_bcc.total_low_low*.

Here is a sample trigger that displays this variable whenever the operator presses [T]:

```
STATE: display_total_dce_good_bcc
   CONDITIONS: KEYBOARD "Tt"
   ACTIONS:
      {
         displayf ("Total DCE good BCC's = %d", dce_good_bcc.total_low_low);
      }
```

Refer to Section 62.1 for more detail on the structure of counters.

57.13 Creating a Structure Pointer

We have just seen how a structure can be created to store and access data conveniently. A structure can also be used as a multibyte pointer that is superimposed on data that has been stored previously.

In our example we will declare the structure of an IL buffer and then point this structure at a newly received IL buffer.

The precise structure of an IL buffer is given in the following declaration. Note that there are 32 bytes devoted to header information and the remaining 4K bytes are available for data.

```
{
  struct il_buffer
  {
     unsigned short lock;
     unsigned short maintain_bits;
     unsigned short buffer_size;
     unsigned short transmit_tag;
     unsigned short receive_tag;
     unsigned long char_buff_frame_start;
     unsigned long char_buff_frame_end;
     unsigned short tick_count_high;
     unsigned short tick_count_mid;
     unsigned short tick_count_low;
     unsigned short available_space_offset;
     unsigned short bytes_remaining;
     unsigned long bcc_indicator;
     unsigned char data [4064];
  };
```

The next step is to create a pointer that has the structure of *il_buffer*. First, declare the structure of *il_buffer*, as indicated above. Then declare *buffer_ptr* as a structure-pointer, as follows:

```
struct il_buffer * buffer_ptr;
```

The next step is to wait for an INFO frame to be monitored. When the the frame data has been buffered and *m_lo_ph_il_buff* has been updated with the new buffer-segment number, assign the first address of this buffer to *buffer_ptr*.

```
buffer_ptr = (void *) ((long) m_lo_ph_il_buff << 16);
```

Now a structure has been created around the most recent upward-moving IL buffer. This means that rather than moving a pointer around in the IL buffer, you can access elements in the buffer directly. The *tick_count_low* variable, for example, would be called *buffer_ptr->tick_count_low*. (The -> operator is used in place of the dot operator in structure-pointers.)

The first element of the *data* string would be called *buffer_ptr ->data[0]*. Here is a program that displays on the prompt line the fifth data element (the packet-*type* byte) in the IL buffer for Info frames monitored on DTE.

```
{
   extern volatile unsigned short m_lo_ph_il_buff;
   struct il_buffer
   {
      unsigned short lock;
      unsigned short maintain_bits;
      unsigned short buffer_size;
      unsigned short transmit_tag;
      unsigned short receive_tag;
      unsigned long char_buff_frame_start;
      unsigned long char_buff_frame_end;
      unsigned short tick_count_high;
      unsigned short tick_count_mid;
      unsigned short tick_count_low;
      unsigned short available_space_offset;
      unsigned short bytes_remaining;
      unsigned long bcc_indicator;
      unsigned char data [4064];
   };
   struct il_buffer * buffer_ptr;
}
   LAYER: 2
      STATE: monitor_il_buffers
         CONDITIONS: DTE INFO
         ACTIONS:
         {
            buffer_ptr = (void *) ((long) m_lo_ph_il_buff <<16);
            pos_cursor (0,0);
            displayf ("%02x ", buffer_ptr->data[4]);
         }
```

58 Routines

This manual documents the C routines that are "external" to the C program—that is, defined elsewhere than in the program. Most of these routines are used by the C translator when it converts softkey-selectable programming "tokens"—most commonly those tokens that are appropriate to Actions blocks—entered by the user on the Protocol Spreadsheet. Some, like the Disk I/O routines, are associated with no spreadsheet conditions or actions and can be accessed only in C regions on the spreadsheet.

58.1 Declarations

In most of the examples in the manual, we have not bothered to declare routines since it is not necessary. In the absence of a declaration, the compiler assumes that the routine is external and that it returns an integer. In nearly all cases, this assumption works. In those rare cases when the routine returns another data type (the stats-display routine *get_68k_phys_addr*, for example, returns a *long*) it must be declared.

58.2 Arguments

An argument is an input that the user provides when he calls a routine. Arguments are placed inside of parentheses just following the routine name, as in this call to the *pos_cursor* routine: *pos_cursor (1,5);*

This routine requires two arguments in order to position the cursor in one of 1,088 possible character positions. The first argument selects one of the seventeen horizontal rows. The second argument selects one of the sixty-four vertical columns.

Many routines in the INTERVIEW library have arguments whose names end in the letters *ptr* or *pointer*. If you look at the synopsis for the *displays* routine, for example, you will see that the only argument is something called *string_ptr*. This is an address argument. The user enters a four-byte address as argument when he calls the displays routine, and the routine goes to this address and begins displaying data until a null (or other nondisplayable character) is encountered.

Pointers are four-byte addresses. The following call to the *displays* routine will go to the location of *m_packet_info_ptr* (the first byte of user data in a packet) and begin displaying data until a nondisplayable character is encountered:

```
displays (m_packet_info_ptr);
```

Array names also are four-byte addresses. The following example will display the characters in the array *string*:

```
char string [] = "QWERTY";
displays (string);
```

A string of characters declared inside of double-quotation marks is really a four-byte address that points to the first character in the string. In the function call *displays ("qwertyuiop")*, *"qwertyuiop"* qualifies as a string pointer and therefore satisfies the formal definition of the routine.

Many routines have no arguments and are called with empty parentheses:

```
sound_alarm();
```

Do not omit the parentheses. Without them, *sound_alarm* is a variable instead of a routine.

58.3 Returns

In addition to performing various operations, many routines include a *return* function that, at the end of the routine, stores a user-defined value in a memory location. As an example, we will look at an X.25 routine called *l3_window_full.*

The *l3_window_full* routine is declared automatically by the translator after the user has made a WINDOW FULL softkey entry. The synopsis for *l3_window_full* shows how it is declared:

```
extern unsigned char l3_window_full (path_number);
```

The routine is declared as a *char* because at the end of the routine, a return function will store a *char*-sized value (8 bits) in memory. If the packet window is full, the stored value will be nonzero. If the packet window is not full, the value will be zero.

The stored value is accessed any time you call the routine in your program. If you want to test for the window being full, you can enter this line of code:

```
if (l3_window_full(path_number) != 0) sound_alarm();
```

Here is a simpler coding for the same test:

```
if (l3_window_full(path_number)) sound_alarm();
```

This coding works for the same reason that *if (1) sound_alarm();* or *if (!0) sound_alarm();* will sound the alarm. Nonzero constants, variables, and expressions are true in C and cause statements to be executed inside of *if*, *while*, and other control constructions. Constants, variables, and expressions that equal zero are false and prevent statements in control structures from being executed.

If a routine is declared as a *short*, a *short* will be set aside in memory and any value returned by the routine (via a *return* function) will be stored there. If the routine is declared a *long*, a *long* will be reserved. If the routine is declared *void*, no space will be reserved in memory and a call to return a value will not be successful.

58.4 User-Defined Routines

The following coding will blank out the prompt line near the top of the INTERVIEW run-mode display.

```
pos_cursor(0,0);
displays ("                                                  ");
```

If you code these two routines each time you display a user-prompt, you can always be sure that the prompt line will be blank and that each prompt will overwrite the previous prompt completely. The only problem is that the two routines are laborious to type in.

A better way is to declare a routine that executes the two "subroutines" automatically.

Declare a routine with its arguments inside parentheses and its body—the list of statements or subroutines that the routine is intended to perform—inside a pair of curly braces.

```
void blank_prompt_line()
{
   pos_cursor(0,0);
   displays ("                                                  ");
}
```

Now you can blank out the line simply by typing this:

```
blank_prompt_line();
```

Suppose you wanted a routine that blanked the prompt line and generated a new prompt. The new prompt will be the argument for the routine:

```
void new_prompt (string_pointer)
char string_pointer [];
{
   pos_cursor(0,0);
   displays ("                                                  ");
   pos_cursor(0,0);
   displays (string_pointer);
}
```

Now you can generate a prompt against a blank background with this simple routine:

```
new_prompt ("This prompt will overwrite any previous prompt");
```

NOTE: User routines may be declared and defined outside of the current spreadsheet program—in include files or linkable-object files. See Section 56.4.

58.5 Example Routines

We will provide three examples that will help illustrate how routines are created.

(A) Example Routine: Temporary Prompt

Here is a user-defined routine that blanks the prompt line, displays a new user-defined prompt, and then waits a user-defined interval before blanking the prompt line again. The routine is called *temporary_prompt*. The two inputs are 1) the new prompt, and 2) the number of seconds that you want the prompt to remain on the display.

The routine incorporates one external routine, *timeout_restart_action*, discussed in Section 69.3 of the section titled "Other Library Tools," and one internal routine, *blank_prompt_line*, discussed above.

```
{
   struct
   {
      unsigned long event_id;
      unsigned short event_id_uid;
   }
   timeout_prompt;
   void blank_prompt_line()
   {
      pos_cursor(0,0);
      displays ("                                                    ");
   }
   void temporary_prompt (string_pointer, seconds)
   char string_pointer [];
   char seconds;
   {
      blank_prompt_line();
      pos_cursor(0,0);
      displays(string_pointer);
      timeout_restart_action (&timeout_prompt, seconds * 1000, blank_prompt_line);
   }
}
STATE: test_temporary_prompt
   CONDITIONS: KEYBOARD " "
   ACTIONS:
   {
      temporary_prompt("This prompt will self-destruct in 4 seconds.", 4);
   }
```

Note that the *blank_prompt_line* routine is embedded inside the *timeout_restart_action* routine, which in turn is embedded inside the *temporary_prompt* routine.

Note also:

The structure *timeout_prompt* is needed by the *timeout_restart_action* routine. The structure is explained in Table 69-1.

The two arguments in the *temporary_prompt* routine are declared outside the body of the routine (that is, outside of the curly braces). As a result, they are not redeclared each time the routine is called.

Timeout timers increment in milliseconds, so the user's *seconds* entry is multiplied by 1,000.

(B) Example Routine: Display Binary Value of Byte

The next sample routine takes a user-defined 8-bit value as input and expands it into a binary display of ASCII 1's and 0's. The routine, called *display_binary*, uses the & ("and") operator to isolate each bit and turn it into a "1" or "0" in an ASCII string called *binary_string*. See Section 57.6 for a discussion of the & operator.

The condition-and-action program that follows the declaration of *display_binary* uses the routine to expand the packet-type byte in each DCE packet.

```
{
 extern volatile unsigned char * m_packet_ptr;
 extern event dce_packet;
 char binary_string [8];
 void display_binary (hex_value)
 char hex_value;
 {
  if ((hex_value & 0x80) == 0) binary_string[0] = '0';
  else binary_string[0] = '1';
  if ((hex_value & 0x40) == 0) binary_string[1] = '0';
  else binary_string[1] = '1';
  if ((hex_value & 0x20) == 0) binary_string[2] = '0';
  else binary_string[2] = '1';
  if ((hex_value & 0x10) == 0) binary_string[3] = '0';
  else binary_string[3] = '1';
  if ((hex_value & 0x08) == 0) binary_string[4] = '0';
  else binary_string[4] = '1';
  if ((hex_value & 0x04) == 0) binary_string[5] = '0';
  else binary_string[5] = '1';
  if ((hex_value & 0x02) == 0) binary_string[6] = '0';
  else binary_string[6] = '1';
  if ((hex_value & 0x01) == 0) binary_string[7] = '0';
  else binary_string[7] = '1';
  displayf ("\n%s", binary_string);
 }
}
  STATE: binary
   CONDITIONS: { dce_packet }
   ACTIONS:
   {
    display_binary (m_packet_ptr[2]);
   }
```

(C) Example Routine: Compare String Against Line Data

Here is a routine called *strcmp* that matches a user-entered string to line data, beginning at a point in the line data that the user specifies. The arguments are the string itself and a pointer to the beginning of the line data.

When the user enters his string inside double quotes, the compiler writes the string into memory, appends a zero (null), and returns a pointer to the first character in the string. The *strcmp* routine uses this zero to determine when the match is complete.

If a complete match is found, the *return(1)* routine breaks out of the while loop, so the *return(0)* never is executed. A routine that returns 1 (or nonzero) inside of an *if* condition will make the condition true.

The sample program that uses the *strcmp* routine looks on the DCE side for a data packet with a user-data field that begins "␍␊PASSWORD." This string occurs on the "HDLC/X.25 Data Sample" diskette, DSK-951-007-1, shipped with your INTERVIEW. Be sure to load in the Layer 2 and Layer 3 X.25 packages if you try out this program. The Layer 3 package will provide you with your line-data pointer (*m_packet_info_ptr*).

```
{
 extern volatile unsigned char *m_packet_info_ptr;
 int element;
 int strcmp (user_string_ptr, line_data_ptr)
 char user_string_ptr [];
 char * line_data_ptr;
 {
   element = 0;
   while (user_string_ptr[element] == line_data_ptr[element])
       {
         if (user_string_ptr[++element] == 0)
         return (1);
       }
   return(0);
 }
}
LAYER: 3
   STATE: match_user_data_field
     CONDITIONS: DCE DATA
     ACTIONS:
     {
       if (strcmp("\x0d\x0aPASSWORD", m_packet_info_ptr))
       sound_alarm();
     }
```

59 Monitor/Transmit Line Data

The external variables and routines in this section are available for use by the programmer to monitor and transmit data. Their use on the Protocol Spreadsheet is not limited to any particular layer, though normally they belong at Layer 1.

The variables and routines approximate Layer 1 spreadsheet-generated conditions and actions. Refer to Section 28 for more detailed explanations of the purposes of specific conditions and actions. Sometimes the name of the variable or routine is sufficient for identifying its related spreadsheet token. When this is not the case, the information is provided below.

59.1 Structures

Use the structure *xmit_list*, shown in Table 59-1, when transmitting line data via the *l1_transmit* routine. Refer to *l1_transmit* in Section 59.3(B) for an example of how to use this structure.

Table 59-1
Transmit Structures

Type	Variable	Value (hex/decimal)	Meaning
Structure Name:	xmit_list		Structure of a transmit list for *l1_transmit* routine. Declared as type *struct*. Reference member variables of the structure as follows: *xmit_list.string_length*.
unsigned char *	string		pointer to the location of the transmit string—the transmit string is declared separately
unsigned short	string_length	*0-ffff/0-65535*	length of the transmit string

59.2 Variables

(A) Monitoring Events

1. *Emulate or monitor mode.* Layer 1 events include characters received, good or bad BCC's, aborts, parity errors, and framing errors. All event variables in Table 59-2 containing a *_td* or *_rd* suffix are valid in either emulate or monitor mode. These event variables are *fevar_rcvd_char_rd, fevar_rcvd_char_td, fevar_gd_bcc_rd, fevar_gd_bcc_td, fevar_bd_bcc_rd, fevar_bd_bcc_td, fevar_abort_rd, fevar_abort_td, fevar_parity_rd, fevar_parity_td, fevar_frm_error_rd, fevar_frm_error_td,* and *fevar_rcv_buffer_full.* The variable *fevar_frm_error_rd,* for example, equates to DCE FRAMING_ERROR (or RECEIVE FRAMING_ERROR when you are emulating DTE).

   You can use both *td* and *rd* variables relating to the same event in one conditions block. Suppose you want count all bad BCC's, from either side of the line. Enter the following CONDITIONS/ACTIONS block:

   ```
   CONDITIONS:
   {
    fevar_bd_bcc_td || fevar_bd_bcc_rd
   }
   ACTIONS: COUNTER bad_bcc INC
   ```

   Using spreadsheet tokens, the same test needs two CONDITIONS/ACTIONS blocks:

   ```
   CONDITIONS: DTE BAD_BCC
   ACTIONS: COUNTER bad_bcc INC
   CONDITIONS: DCE BAD_BCC
   ACTIONS: COUNTER bad_bcc INC
   ```

   Use *fevar_rcv_buffer_full* and its associated status variable, *rcv_buffer_full,* to monitor the status of the character buffer. The moment the buffer is full, *fevar_rcv_buffer_full* comes true and the value of *rcv_buffer_full* transitions from zero to a non-zero value. Then, new data begins to overwrite the old data. The softkey equivalent of *fevar_rcv_buffer_full* is the layer-independent condition BUFFER_FULL when it appears alone in a conditions block. When BUFFER_FULL is combined with another condition, in most cases the other condition will supply the event variable and only the status test will be used. See Section 27 for a discussion of this and other layer-independent conditions and actions.

Table 59-2
Monitor/Transmit Variables

Type	Variable	Value (hex/decimal)	Meaning
extern fast_event	fevar_rcvd_char_rd		True for each character received on RD. Line Setup configured for emulate or monitor mode.
extern fast_event	fevar_rcvd_char_td		True for each character received on TD. Line Setup configured for emulate or monitor mode.
extern fast_event	fevar_gd_bcc_rd		True when a good BCC is calculated for an RD block or frame. Line Setup configured for emulate or monitor mode.
extern fast_event	fevar_gd_bcc_td		True when a good BCC is calculated for a TD block or frame. Line Setup configured for emulate or monitor mode.
extern fast_event	fevar_bd_bcc_rd		True when a bad BCC is calculated for an RD block or frame. Line Setup configured for emulate or monitor mode.
extern fast_event	fevar_bd_bcc_td		True when a bad BCC is calculated for a TD block or frame. Line Setup configured for emulate or monitor mode.
extern fast_event	fevar_abort_rd		True when an abort is detected in an RD frame. Line Setup configured for emulate or monitor mode.
extern fast_event	fevar_abort_td		True when an abort is detected in a TD frame. Line Setup configured for emulate or monitor mode.
extern fast_event	fevar_parity_rd		True when a parity error is detected for an RD byte. Line Setup configured for emulate or monitor mode.
extern fast_event	fevar_parity_td		True when a parity error is detected for a TD byte. Line Setup configured for emulate or monitor mode.
extern fast_event	fevar_frm_error_rd		True when an async framing error is detected for an RD byte. Line Setup configured for emulate or monitor mode.
extern fast_event	fevar_frm_error_td		True when an async framing error is detected for a TD byte. Line Setup configured for emulate or monitor mode.

Table 59-2 (continued)

Type	Variable	Value (hex/decimal)	Meaning
extern fast_event	fevar_xmit_cmplt		True when the INTERVIEW puts a transmission out onto the link. Line Setup configured for emulate mode only.
extern fast_event	fevar_rcv_buffer_full		Returns true at the moment the character buffer fills with data and will begin to overwrite existing data. Line Setup configured for emulate or monitor mode.
extern volatile unsigned short	rcv_buffer_full	0	not full
		1	full
			Line Setup configured for emulate or monitor mode.
extern unsigned short	rcvd_char_td		Most recent TD character is stored in this variable. Line Setup configured for emulate or monitor mode.
		0-ff/0-255	data character (lower byte in 16-bit data word in data buffer)
		100/256	good or bad BCC
		101/257	flag
		102/258	sync
		103/259	abort
extern unsigned short	rcvd_char_rd		Most recent RD character is stored in this variable. Line Setup configured for emulate or monitor mode.
		0-ff/0-255	data character (lower byte in 16-bit data word in data buffer)
		100/256	good or bad BCC
		101/257	flag
		102/258	sync
		103/259	abort
extern unsigned char	td_modifier		Most recent modifier byte for a TD data character. This is the upper byte in the 16-bit data word reserved for each data character in the data buffer. Line Setup configured for emulate or monitor mode.
		1	data—initial value (always included in value of *td_modifier*)
		2	alternate code set
		4	underline (rd character)
		8	reverse image
		10/16	hexadecimal
		20/32	low intensity
		40/64	blink
		80/128	strike-thru (parity error)

Table 59-2 (continued)

Type	Variable	Value (hex/decimal)	Meaning
extern unsigned char	rd_modifier		Most recent modifier byte for an RD data character. This is the upper byte in the 16-bit data word reserved for each data character in the data buffer. Line Setup configured for emulate or monitor mode.
		1	data (always included in value of *rd_modifier*)
		2	alternate code set
		4	underline (rd character)—initial value of *rd_modifier*
		8	reverse image
		10/16	hexadecimal
		20/32	low intensity
		40/64	blink
		80/128	strike-thru (parity error)

2. *Emulate mode only.* One variable is valid in emulate mode only, since it monitors an emulate action. "SENDing" a transmission means queuing a transmission to send. The layer protocol (the RTS–CTS handshake, for example, at Layer 1) may delay the actual transmission. The fast-event variable *fevar_xmit_cmplt* will not come true until the transmission actually has been sent. Use this condition to start accurate response-time measurements.

   If you try to use *fevar_xmit_cmplt* in monitor mode, you will be returned to the main program menu. When you go to the Protocol Spreadsheet and search for errors, the following message will be displayed: "*Error 140: Unresolved reference fevar_xmit_cmplt.*"

(B) Status Variables

Status variables are those in Table 59-2 that do not include *event* in the Type column. Their associated event variables guarantee that they are updated and tested.

1. *Distinguishing character types.* Suppose you're monitoring the DCE side of the link. Every time a character is detected, the event *fevar_rcvd_char_rd* comes true, regardless of whether or not the character will be stored in the character buffer. Not all characters are "data" characters. A character also may be a flag or the second byte in a block-check, for example. *fevar_rcvd_char_rd* (or *fevar_rcvd_char_td*) does not distinguish character types.

   Character type is stored in the high byte of *rcvd_char_rd* or *rcvd_char_td*. For data characters, the high byte is zero. The low byte contains the actual value of the character.

For a "non-data" character, hereafter referenced as a special symbol, the high byte of *rcvd_char_rd* is a non-zero value. The low byte specifies a special symbol to be displayed on the data screen, overwriting or replacing the character. The special symbols are [S] (sync), [G] (good BCC), [B] (bad BCC), [A] (abort), and [F] (flag). See Table 59-2.

Notice on Table 59-2 that the value for good BCC and bad BCC is the same. Use *fevar_gd_bcc_rd* and *fevar_bd_bcc_rd* event variables to distinguish between good and bad BCC's (or data BCC's in DDCMP). Likewise, use *fevar_gd_bcc2_rd* and *fevar_bd_bcc2_rd* to differentiate between good and bad header BCC's in DDCMP. Refer to Section 75 for DDCMP variables.

Aborts are not automatically reflected in *rcvd_char_rd* and *rcvd_char_td*. When seven consecutive 1-bits are received in 7E-framed protocols, the controller chip generates an interrupt. The bits, however, are not stored in memory. In this case, use *fevar_abort_rd* or *fevar_abort_td* to detect the interrupt. When this event variable transitions to true, it updates *rcvd_char_rd* (or *rcvd_char_td*) to indicate an abort.

Use *rcvd_char_td* and *rcvd_char_rd* to monitor received characters, independent of whether or not they will be buffered. The following condition detects RD data characters only:

```
CONDITIONS:
{
  fevar_rcvd_char_rd && (!(rcvd_char_rd & 0x100))
}
```

2. *Attributes*. Data characters and special symbols in the character buffer are available for normal or enhanced display on the data display-screen. Access the data display by pressing DATA on the first rack of Run-mode softkeys, or by selecting it as the initial Run-mode display on the Display Setup menu.

   The current attributes for RD data are stored in *rd_modifier*. Table 59-2 shows how the various attributes are coded. The initial value of *rd_modifier* is always five. This value means that the character is data (1) on the RD (4) side. RD data is always underlined. TD data is never underlined. The initial value of *td_modifier*, therefore, is one.

   You may change some attributes by using spreadsheet tokens (or their equivalent C routines). The Layer 1 ENHANCE action allows you to control reverse-image, blink, hexadecimal, and low intensity enhancements. This action also updates *rd_modifier*, *td_modifier*, or both.

   When an RD data character is written to the character buffer, the value of *rd_modifier* is written to the high byte of a two-byte data event-word. The data character, found in *rcvd_char_rd*, is written to the low byte. See Section 59.3(C) on the format of character-buffer event words.

NOTE: The attributes in *rd_modifier* and *td_modifier* do not apply to special symbols. *rd_modifier* and *td_modifier* always reflect the attributes last assigned to data. Underlining applied to (RD) special symbols on the data display–screen comes from a bit in the special receive–event word. See Table 59-3.

59.3 Routines

Unless noted otherwise, the routines discussed below apply when the Line Setup menu shows either emulate or monitor mode.

(A) Controlling Data Display

ctl_enhance_td

Synopsis

```
extern void ctl_enhance_td(enhance_type_status);
unsigned short enhance_type_status;
```

Description

This routine turns various enhancements of the data display on and off on the DTE side. It also updates the variable *td_modifier*. The softkey equivalent of this routine is the ENHANCE DTE action on the Protocol Spreadsheet.

Inputs

There is one two–byte parameter. The high byte identifies the type of enhancement to be controlled: blink (40), low intensity (20), hexadecimal representation (10), and reverse image (08). The low–order byte indicates the status of the enhancement. To indicate a given enhancement is on, the second byte has the same value as the first. If the enhancement is to be turned off, the value of the second byte is zero. For example, if you want to turn blink on, the parameter value is 0x4040. To turn blink off, it is 0x4000.

Multiple enhancements can be controlled with one action by using hexadecimal addition of the parameters, as in the example for *ctl_enhance_rd*.

Example

Assume X.25 protocol for this example. You want to enhance the packet type byte on the DTE side with a blinking, reverse image.

```
LAYER: 1
    STATE: enhance_packet_type
       CONDITIONS:  DTE STRING "[F][F]((XXXXXXX0))[X][X]"
       ACTIONS:
       {
        ctl_enhance_td(0x4040);
        ctl_enhance_td(0x0808);
       }
```

```
CONDITIONS: DTE STRING "[F][F]((XXXXXXX0))[X][X][X]"
ACTIONS:
{
 ctl_enhance_td(0x4000);
 ctl_enhance_td(0x0800);
}
```

ctl_enhance_rd

Synopsis

extern void ctl_enhance_rd(enhance_type_status);
unsigned short enhance_type_status;

Description

This routine turns various enhancements of the data display on and off on the DCE side. It also updates the variable *rd_modifier*. The softkey equivalent of this routine is the ENHANCE DCE action on the Protocol Spreadsheet.

Inputs

See *ctl_enhance_td*.

Example

Assume X.25 protocol for this example. You want to enhance the packet type byte on the DCE side with a blinking, reverse image.

```
LAYER: 1
    STATE: enhance_packet_type
        CONDITIONS: DCE STRING "[F][F]((XXXXXXX0))[X][X]"
        ACTIONS:
        {
          ctl_enhance_rd(0x4848);
        }
        CONDITIONS: DCE STRING "[F][F]((XXXXXXX0))[X][X][X]"
        ACTIONS:
        {
         ctl_enhance_rd(0x4800);
        }
```

ctl_capture_td

Synopsis

extern void ctl_capture_td(status);
unsigned short status;

Description

This routine turns on and off the presentation of DTE data to the screen—that is, it stops or "freezes" the display—and capture of data to the screen buffer (character RAM). Unlike the Manual Freeze mode initiated by the [FREEZE] key, however, the "capture off" action does not allow you to scroll through the buffer while the test continues. The softkey equivalent of this routine is the CAPTURE DTE action on the Protocol Spreadsheet.

Inputs

The only parameter is the status of capture, on (0x00) or off (0x10). Turning capture off freezes the display.

Example

Assume X.25 protocol for this example. You want to turn capture off as soon as the cause byte is displayed in a Clear packet on the DTE side. Capture will be resumed when the spacebar is pressed.

```
LAYER: 1
    STATE: find_cause
      CONDITIONS: DTE STRING "[F][F]((XXXXXXX0))⊠⊠13⊠"
      ACTIONS:
      {
       ctl_capture_td(0x10);
      }
      CONDITIONS: KEYBOARD " "
      ACTIONS:
      {
       ctl_capture_td(0x00);
      }
```

ctl_capture_rd

Synopsis

extern void ctl_capture_rd(status);
unsigned short status;

Description

This routine turns on and off the presentation of DCE data to the screen—that is, it stops or "freezes" the display—and capture of data to the screen buffer (character RAM). Unlike the Manual Freeze mode initiated by the FREEZE key, however, the "capture off" action does not allow you to scroll through the buffer while the test continues. The softkey equivalent of this routine is the CAPTURE DCE action on the Protocol Spreadsheet.

Inputs

The only parameter is the status of capture, on (0x00) or off (0x100). Turning capture off freezes the display.

Example

Assume X.25 protocol for this example. You want to turn capture off as soon as the cause byte is displayed in a Clear packet on the DCE side. Capture will be resumed when the spacebar is pressed.

```
LAYER: 1
    STATE: find_cause
        CONDITIONS: DCE STRING "[F][F]((XXXXXXX0))⊠⊠13⊠"
        ACTIONS:
        {
         ctl_capture_rd(0x100);
        }
        CONDITIONS: KEYBOARD " "
        ACTIONS:
        {
         ctl_capture_rd(0x00);
        }
```

outsync_action

Synopsis

```
extern void outsync_action(side);
unsigned short side;
```

Description

The *outsync_action* routine applies to synchronous format only. This routine sends one of the receivers (TD or RD) out of sync and initiates a search for sync. The softkey equivalent of this routine is the (PROTOCL) OUT_SYN action on the Protocol Spreadsheet.

Inputs

The only parameter identifies which side of the line is to go out of sync, 0 for the DTE side, 1 for the DCE side.

Example

To display DTE-protocol information only, initiate sync each time a start-of-text character is found. The results of this routine are similar to turning capture off and on, but here the display does not have to be turned on again. It resumes automatically with sync.

```
LAYER: 1
    STATE: go_out_of_sync
        CONDITIONS: DTE STRING "SX "
        ACTIONS:
        {
         outsync_action(0);
        }
```

(B) Transmitting

Use the following routines in emulate mode only. If you try to call one of these routines in monitor mode, you will be returned to the main program menu. When you go to the Protocol Spreadsheet and search for errors, a message like the following will be displayed: "*Error 140: Unresolved reference l1_il_transmit.*"

l1_transmit

Synopsis

```
extern void l1_transmit(count, struct_send_string_ptr, xmit_tag);
unsigned short count;
struct xmit_list
   {
    unsigned char * string_ptr;
    unsigned short string_length;
   };
struct xmit_list * struct_send_string_ptr;
unsigned short xmit_tag;
```

Description

The *l1_transmit* routine sends a specified string with a user-determined BCC.

Inputs

The first parameter is the number of strings to be sent.

The second parameter is a pointer to a structure which in turn identifies the location and length of each string.

The third parameter is a transmit tag which includes a BCC in bits 0–2: good (001), bad (010), or abort (011). Bits 3–7 are reserved for future use. Integers may be used to indicate the value of the transmit tag: good (1), bad (2), and abort (3).

Example

Assume you want to send a fox message at Layer 1 inside of an X.25 data packet with a good block check. You might have 2 strings, one with the Layers 2 and 3 header information, and one with the fox message. You would send these strings as follows:

```
{
 unsigned char headers [] = {0x01, 0x00, 0x10, 0x04, 0x00};
 unsigned char message [] = "((FOX))";
 struct xmit_list
  {
   unsigned char * string;
   unsigned short string_length;
  };
 struct xmit_list send_string [] = {&headers[0], 5, &message[0], sizeof(message) - 1};
}
```

```
LAYER: 1
    STATE: send_message
      CONDITIONS: KEYBOARD " "
      ACTIONS:
      {
       l1_transmit(2, &send_string[0], 1);
      }
```

l1_il_transmit

Synopsis

```
extern void l1_il_transmit(il_buffer_number, relay_baton, data_start_offset, transmit_tag);
unsigned short il_buffer_number;
unsigned short relay_baton;
unsigned short data_start_offset;
unsigned short transmit_tag;
```

Description

This routine sends a designated interlayer message buffer out onto the line.

Inputs

The first parameter is the interlayer message buffer number.

The second parameter is the maintain bit used to hold the buffer while the send operation is performed at Layer 1.

The third parameter is the offset from the beginning of the buffer to the service data unit (SDU).

The fourth parameter is a transmit tag which includes a BCC in bits 0–2: good (001), bad (010), or abort (011). Bits 3–7 are reserved for future use. Integers may be used to indicate the value of the transmit tag: good (1), bad (2), and abort (3).

Example

Send the same text as in the example for *l1_transmit*. The softkey equivalent of this routine is the SEND action on the Protocol Spreadsheet. Refer to Section 63.3(A) for a description of the *_get_il_msg_buff*, *_start_il_buff_list*, and *_insert_il_buff_list_cnt* routines.

```
{
 unsigned short il_buffer_number;
 unsigned short relay_baton;
 unsigned short data_start_offset;
 unsigned char message [] = "01\x0001004\x000((FOX))";
}
```

```
LAYER: 1
     STATE: send_message
       CONDITIONS: KEYBOARD " "
       ACTIONS:
       {
        _get_il_msg_buff(&il_buffer_number, &relay_baton);
        _start_il_buff_list(il_buffer_number, &data_start_offset);
        _insert_il_buff_list_cnt(il_buffer_number, data_start_offset, &message[0],
            (sizeof(message) - 1));
        l1_il_transmit(il_buffer_number, relay_baton, data_start_offset, 1);
       }
```

idle_action

Synopsis

```
extern void idle_action(character);
unsigned char character;
```

Description

Only for format SYNC, the *idle_action* routine allows you to change the idle–line condition applied by the INTERVIEW. The softkey equivalent of this routine is the (PROTOCL) IDLE_LN action on the Protocol Spreadsheet.

Inputs

The only parameter is a character or numeric value representing the idle character.

Example

X.21 or X.21BIS idles different characters in various states, FF, NU, +, for example. To signal a change in protocol state, you might change the idle character to +:

```
LAYER: 1
     STATE: change_idle_character
       CONDITIONS: KEYBOARD " "
       ACTIONS:
       {
        idle_action('+');
       }
```

set_tcr_b

Synopsis

extern void set_tcr_b (tcr_register_mask, tcr_register_value);
unsigned char tcr_register_mask;
unsigned char tcr_register_value;

Description

This routine clamps the transmit line to 0 (space) or 1 (mark), or unclamps it so that *transmit* routines may be executed. In X.21, steady zero will signal a clear request/indication or a clear confirm, while steady 1 will indicate one of the call-ready or call-setup states. In other contexts, the routine simply initiates and terminates a *break*.

Inputs

The first parameter is the mask that is *and*ed with the current TCR register to turn the current values of bits 3 and 4 (counting 1-8 from the right) to zero. This mask is always 0xf3.

The second parameter contains the new values of bits 3 and 4 that will be written to the register. The three available parameters are 0x08 to clamp the line to zero, 0x0c to clamp the line to 1, and 0x04 to unclamp the line and permit data transmissions.

Example

This program will generate a 250-millisecond break when the operator presses the [BRK] key.

```
{
 extern fast_event keyboard_new_any_key;
 extern volatile unsigned short keyboard_any_key;
}
     STATE: generate_break
        CONDITIONS:
        {
         keyboard_new_any_key && (keyboard_any_key == 0x1e3)
        }
        ACTIONS: TIMEOUT break RESTART 0.250
        {
         set_tcr_b (0xf3, 0x08);
        }
        CONDITIONS: TIMEOUT break
        ACTIONS:
        {
         set_tcr_b (0xf3, 0x04);
        }
```

(C) Writing to Character RAM

For the sake of speed, the 64–Kbyte character buffer uses a shorter data word than the 32–bit word in the Display Window and traces. Refer to Table 61-4. A sixteen–bit event word is reserved for each character in the 64–Kbyte character buffer.

Table 59-3 shows the format of event words. Two kinds of event word should be distinguished: data and special receive.

1. *Data Event–Words*. Data event–words may contain enhancement attributes in the high byte. Whereas attributes comprise 24 bits of a *long* in the Display Window and the traces, in the character buffer they are contained in only 8 bits. Data words in the character buffer, therefore, include a less flexible set of attributes. Color attributes, for example, are not directly available in words written to the character buffer. See Section 16, Color Display, for an explanation of how reverse, blink, and low enhancements in the character buffer may be mapped to colors in the RGB output. Table 59-3 lists the available attributes.

   The character is located in the low 8 bits. Its value can range from hexadecimal 0 through FF.

2. *Special–Receive Words*. The high byte in special–receive words determines the symbol (from the special graphic character font) that will overlay the character contained in the low byte. The symbols that may be written to the character buffer are good BCC's, bad BCC's, aborts, flags, and sync. One bit, the td/rd indicator, controls on which side the symbol will be displayed. Symbols on the RD side are underlined, as all RD data is. Notice in Table 59-3 that the td/rd indicator bit is the same one that controls the underline enhancement in data event–words.

   The value in the low byte is meaningless in the context of special–receive words. The special symbol will overlay or replace the character. Its value, nevertheless, can range from hexadecimal 0 through FF.

Table 59-3
Character Buffer 16-Bit Word

Type	Mask (hex)	Input (hex)	Meaning
data			*data-event word:*
	0100	0100	the low byte contains data
	0500	*add 0100 to the following:*	td/rd indicator:
		0000	td character
		0400	rd character (underlined)
	ff00	*add modified value of td/rd indicator to one (or a combination) of the following:*	enhancements: † (enhancements apply to data indicated in low byte)
		0000	normal
		0200	alternate code set
		0800	reverse image
		1000	hexadecimal
		2000	low intensity
		4000	blink
		8000	strike-thru (parity error on character)
special receive			*special receive-event word:*
	8300	0200	special receive-event word
		8200	reserved
	8700	*add 0200 to the following:*	td/rd indicator:
		0000	td character
		0400	rd character (underlined)
	bf00	*add modified value of td/rd indicator to one of the following:*	special event: (symbols for these events overlay the data indicated in low byte)
		0800	good CRC
		1000	bad CRC
		1800	abort
		2000	flag
		2800	sync
		3000	bad CRC2 (DDCMP)
		3800	good CRC2 (DDCMP)
reserved	0700	0400	reserved
reserved	0f00	0800	reserved

† Selecting rd (0400) for the td/rd indicator results in the data being underlined. The underline enhancement shares the same bit. It has been omitted from the list of enhancements to avoid an error from double counting.

The routines for writing 16–bit event words to the character buffer are *add_event_to_buff* and *add_array_to_buff*. These routines may be used when the Line Setup menu shows either emulate or monitor mode.

add_event_to_buff

Synopsis

```
extern unsigned int add_event_to_buff(event_word);
unsigned int event_word;
```

Description

The *add_event_to_buff* routine writes the specified input to the 64–Kbyte character buffer.

Inputs

The only input is a 16–bit event–word to be written to the buffer. Table 59-3 lists the coding of event words.

Returns

A one is returned if the event was successfully added to the character buffer. If the routine failed, zero is returned.

Example

To display only SDLC frames with an address of hexadecimal c2, enter the following spreadsheet program:

```
LAYER: 1
{
 extern unsigned short rcvd_char_td;
 extern unsigned short rcvd_char_rd;
}
     STATE: init
        CONDITIONS: ENTER_STATE
        ACTIONS: CAPTURE BOTH OFF
        NEXT_STATE: address
     STATE: address
        CONDITIONS: DTE STRING "FF"
        ACTIONS:
        {
         if(rcvd_char_td == 0xc2)
           {
            add_event_to_buff (((short)td_modifier << 8) + rcvd_char_td);
            ctl_capture_td(0x00);
           }
        }
        CONDITIONS: DTE STRING "FF"
        ACTIONS: CAPTURE DTE OFF
```

```
CONDITIONS: DCE STRING "FF"
ACTIONS:
{
 if(rcvd_char_rd == 0xc2)
   {
    add_event_to_buff (((short)rd_modifier << 8) + rcvd_char_rd);
    ctl_capture_rd(0x00);
   }
}
CONDITIONS: DCE STRING "FF"
ACTIONS: CAPTURE DCE OFF
```

add_array_to_buff

Synopsis

```
extern unsigned int add_array_to_buff(array_ptr, count);
unsigned short * array_ptr;
unsigned char count;
```

Description

The *add_array_to_buff* routine writes specified elements of an array to the 64-Kbyte character buffer.

Inputs

The first parameter is the location of the array to be written to the character buffer. The array consists of 16-bit *shorts*.

The second parameter is the number of elements in the array to be written. The number of elements which can be written to the buffer must be in the range 0-16. Elements in the array must adhere to the format of event words shown in Table 59-3.

Returns

The result of the *add_array_to_buff* routine is all or nothing. A one is returned when all requested elements of the array are successfully added to the character buffer. If the routine fails, zero is returned and nothing is written to the buffer.

Example

To display on the Data Screen only X.25 packets with an LCN of 004, enter the following spreadsheet program. (This program displays the DTE side of the line only. Additional programming similar to that entered would include DCE data.)

```
LAYER: 1
{
 unsigned short dte_array [100];
 unsigned short lcn;
 extern unsigned short rcvd_char_td;
}
     STATE: init
        CONDITIONS: ENTER_STATE
        ACTIONS: CAPTURE BOTH OFF
        NEXT_STATE: address
     STATE: address
        CONDITIONS: DTE STRING "[F][F]"
        ACTIONS:
        {
         dte_array [0] = (0x0100 + rcvd_char_td);
        }
        NEXT_STATE: frame_type
     STATE: frame_type
        CONDITIONS: DTE STRING "((XXXXXXX0))"
        ACTIONS:
        {
         dte_array [1] = (0x0100 + rcvd_char_td);
        }
        NEXT_STATE: gfi
        CONDITIONS: DTE STRING "((XXXXXXX1))"
        NEXT_STATE: address
     STATE: gfi
        CONDITIONS: DTE STRING "[X]"
        ACTIONS:
        {
         dte_array [2] = (0x0100 + rcvd_char_td);
         lcn = ((unsigned int)rcvd_char_td & 0x0f) << 8;
        }
        NEXT_STATE: lcn
     STATE: lcn
        CONDITIONS: DTE STRING "[X]"
        ACTIONS:
        {
         dte_array [3] = (0x0100 + rcvd_char_td);
         lcn += rcvd_char_td;
         if(lcn == 0x0004)
           {
            add_array_to_buff(dte_array, 4);
            ctl_capture_td(0x00);
            current_state = state_eof;
           }
         else
            current_state = state_address;
         break;
        }
     STATE: eof
        CONDITIONS: DTE STRING "[F][F]"
        ACTIONS: CAPTURE DTE OFF
        NEXT_STATE: address
```

60 EIA

The Test Interface Module (TIM) located in the rear of the INTERVIEW determines the EIA leads available for monitoring and control (Section 10). The variables and routines in this section apply to RS-232, V.35, and RS-449 interface modules. The X.21 module is treated separately in Section 70.

To use the C variables and routines explained in this section, enable EIA leads by selecting **Buffer Control Leads: YES** on the FEB Setup menu. See Section 7.1(B). If no other source for clock is provided, use internal clock (Line Setup menu).

The variables and routines approximate Layer 1 EIA spreadsheet-generated conditions and actions. Their use on the Protocol Spreadsheet is not limited to any particular layer, though normally they belong at Layer 1.

60.1 Variables

With an RS-232, V.35, or RS-449 TIM installed, you may monitor RI, DSR, DTR, CD, CTS, RTS, and UA. The lead names in RS-449 are slightly different: see Table 60-1.

The fast-event variable *fevar_eia_changed* detects a change in EIA leads. It does not establish which lead(s) has changed. Two associated variables, *current_eia_leads* and *previous_eia_leads*, indicate the status of the seven leads. These are two-byte (*short*) variables. Each lead is represented by a different bit in the *short*. Some bits are unused. Table 60-1 lists the mask that can be used to isolate each lead.

Whenever a lead changes, the value in *current_eia_leads* is written to *previous_eia_leads*. Then *current_eia_leads* is updated.

(A) Masking To Detect a Change in a Given Lead

To test whether or not a given lead changed, RTS for example, while disregarding its status, enter the following condition on the Protocol Spreadsheet:

```
CONDITIONS:
{
fevar_eia_changed && (((current_eia_leads ^ previous_eia_leads) & 0x80) == 0x80)
}
```

Select a mask value from the list in Table 60-1 to indicate which lead you care about. Specify multiple leads with a mask derived via hexadecimal addition.

Table 60-1
EIA Variables

Type	Variable	Value (hex/decimal)	Meaning
extern fast_event	fevar_eia_changed		True when the status changes for an EIA lead (non-data). Line Setup configured for emulate or monitor mode.
			RS-232/V.35: (RS-449)
extern const volatile unsigned short	current_eia_leads	4	RI (IC)
		8	DSR (DM)
		10/16	DTR (TR)
		20/32	CD (RR)
		40/64	CTS (CS)
		80/128	RTS (RS)
		200/512	UA
			A value in this list, when *anded* (&) with *current_eia_leads*, equals zero if the lead is *on*. Example: STATE: rts_on { *if ((current_eia_leads & 0x80) == 0) sound_alarm();* }
			Note: This variable will store EIA status if (1) internal or external clock is supplied and (2) EIA leads are enabled on FEB Setup. Line Setup configured for emulate or monitor mode.
extern const volatile unsigned short	previous_eia_leads		Same values as *current_eia_leads*. Updated only after logic has had a chance to compare current and previous leads. Line Setup configured for emulate or monitor mode.

The mask for RTS is 0x80. In the example, the event *fevar_eia_changed* updated *current_eia_leads*. The new *current_eia_leads* was *bitwise-exclusive-OR*ed with *previous_eia_leads* to identify all the leads that changed. Then the result was *bitwise AND*ed with the RTS mask to determine if RTS was among the leads that changed. If this result was equal to the mask, the lead changed.

(B) Masking For the Status of a Lead

You may also test the current status of a lead, independent of any change. *And* the mask with *current_eia_leads*, as in this *if* statement testing for RTS "on":

```
STATE: test_for_rts_on
  {
  if((current_eia_leads & 0x80) == 0)  sound_alarm();
  }
```

If the result is zero, the lead is on. If the result equals the mask, the lead is off. "On" means that a lead is more positive than +3 volts with respect to signal ground. "Off" implies only that a lead is not at or above the "on" threshold, not necessarily that a minus threshold has been attained.

(C) Detect Change and Current Status

The two examples shown above could be combined to test for RTS changing from off to on:

```
CONDITIONS:
{
  (fevar_eia_changed && (((current_eia_leads ^ previous_eia_leads) & 0x80) == 0x80) &&
      ((current_eia_leads & 0x80) == 0))
}
```

This example approximates the translator's version of the spreadsheet-token condition **EIA RTS ON** when it appears alone in a conditions block. When an **EIA** condition is combined with another condition, in most cases the other condition will supply the event variable and only the EIA status test will be used.

60.2 Routines

You may control RS-232 EIA leads in emulate mode only. When the Line Setup menu shows **Mode:** EMULATE DCE, you control CTS, CD, and DSR. An EMULATE DTE selection gives you control over RTS and DTR. Entries on the Interface Control menu may be used to set the leads' initial status (Section 10.6).

ctl_eia

Synopsis

```
extern void ctl_eia(on_mask, off_mask);
unsigned short on_mask;
unsigned short off_mask;
```

Description

The *ctl_eia* routine allows you to control the status of up to three of nine possible leads. Which leads you control depends on your emulation mode. The softkey equivalent of this routine is the **EIA** action on the Protocol Spreadsheet.

Inputs

The first parameter indicates which leads you want to turn on. Each bit in the parameter controls a given lead: RTS/CTS (01), DTR/DSR (02), CD (04), AUX0 (10), AUX1 (20), AUX2 (40), AUX3 (80). Wherever there is a *zero* in the first parameter, the corresponding lead will be turned on. A one in this parameter will *not* cause any lead to be turned off. A value of 0xff will mean *don't care* (no action).

The second parameter indicates which leads you want in the "off" condition. Each bit in the parameter controls a given lead: RTS/CTS (01), DTR/DSR (02), CD (04), AUX0 (10), AUX1 (20), AUX2 (40), AUX3 (80). Wherever there is a *one* in the second parameter, the corresponding lead will be turned off. Zeroes in this parameter do *not* turn leads on. A value of 0 will mean *don't care* (no action).

> **NOTE:** If both bytes are attempting to control the same lead, the off parameter will override the on parameter.

Example

Suppose your emulate mode is EMULATE DCE. As a DCE, you control the CTS, DSR, and CD leads. (An attempt to control the status of RTS or DTR will fail, since the DTE controls these leads.) When RTS is raised, you want to turn CTS on; when RTS drops, turn CTS off.

```
LAYER: 1
     STATE: control_cts
        CONDITIONS: EIA RTS ON
        ACTIONS:
        {
         ctl_eia(0xfe, 0x00);
        }
        CONDITIONS: EIA RTS OFF
        ACTIONS:
        {
         ctl_eia(0xff, 0x01);
        }
```

61 Display Window and Trace

The C structures, variables, and routines detailed in this section control the type and location of certain displays on the INTERVIEW. These displays can be grouped into three categories.

The first display area is the prompt line, the second line on all Run-mode screens.

The second type of display utilizes the Display Window, available as a selection on the Display Setup portion of the Line Setup menu, or conditionally accessible via softkey during Run mode. To write to the Display Window, use the *pos_cursor* (or *restore_cursor*) and *displayc*, *displayf*, or *displays* routines. When using Display Window, you may position the cursor before output is generated on the screen.

The third type of display utilizes one or a combination of the eight available trace buffers. Trace screens are conditionally accessible via softkey during Run mode. Seven user-traces appear as choices under the User Trace selection on the Display Setup menu. The remaining trace is Program Trace, also an option on Display Setup. Program Trace enables you to track any or all layers, one or all tests, and movement between states. To write to any of the eight trace-screens, use the *tracec*, *tracef*, and *traces* routines.

> **NOTE:** You may not use the *pos_cursor* routine to position the cursor on any trace screen. New lines (or blank lines) may be generated via the "\n" specifier.

Attributes—color, underlining, and font, for example—may be assigned to characters in the Display Window and all of the Trace buffers.

> **NOTE:** Color attributes are applied to the RGB output signal, not to the plasma screen.

61.1 Current Display Mode

A group of variables keeps track of softkey movement from one display screen to another (see Table 61-1). When you move from the Display Window to the Program Trace screen, for example, the *fast-event* variable *display_screen_changed* indicates the change of display. The coded value for Display Window now is stored in *prev_display_screen*, and the value for Program Trace can be found in *crnt_display_screen*.

These variables also distinguish between Run mode and Freeze mode. This distinction is important since some keys on the keyboard are mode-dependent. In Freeze mode, for instance, cursor keys automatically become operational for scrolling through the buffer. The programmer will want to avoid using these keys as user-input when *crnt_display_screen* indicates that the unit is in Freeze mode.

Table 61-1
Current Display Variables

Type	Variable	Value (hex/decimal)	Meaning
extern fast_event	display_screen_changed		True when Run-mode display-screen is changed, or when Run/Freeze mode is changed. Value in *crnt_display_screen* is stored in *prev_display_screen*, and *crnt_display-screen* is updated. Line Setup configured for emulate or monitor mode.
extern unsigned short	crnt_display_screen		Contains current display screen (low byte) and indicates whether unit is in Run mode or Freeze mode (high byte). Line Setup configured for emulate or monitor mode.
			display-screen
		0	no display
		1	single-line data
		2	dual-line data
		3	single-line data with leads
		4	dual-line data with leads
		11/17	tabular statistics
		12/18	graphic statistics
		21/33	Display Window
		31/49	Program Trace
		41/65	Layer 1 Protocol Trace
		42/66	Layer 2 Protocol Trace
		43/67	Layer 3 Protocol Trace
		44/68	Layer 4 Protocol Trace
		45/69	Layer 5 Protocol Trace
		46/70	Layer 6 Protocol Trace
		47/71	Layer 7 Protocol Trace
		51/81	User Trace 1
		52/82	User Trace 2
		53/83	User Trace 3
		54/84	User Trace 4
		55/85	User Trace 5
		56/86	User Trace 6
		57/87	User Trace 7
		61/97	TIM package standard stats
		62/98	TIM package aux
			Run/Freeze mode (bit 9)
		100/256	Freeze mode
		0	Run mode

Table 61-1 (continued)

Type	Variable	Value (hex/decimal)	Meaning
extern unsigned short	prev_display_screen		Contains previous display screen (low byte) and indicates whether unit was in Run mode or Freeze mode (high byte). Line Setup configured for emulate or monitor mode.
			display-screen
		0	no display
		1	single-line data
		2	dual-line data
		3	single-line data with leads
		4	dual-line data with leads
		11/17	tabular statistics
		12/18	graphic statistics
		21/33	Display Window
		31/49	Program Trace
		41/65	Layer 1 Protocol Trace
		42/66	Layer 2 Protocol Trace
		43/67	Layer 3 Protocol Trace
		44/68	Layer 4 Protocol Trace
		45/69	Layer 5 Protocol Trace
		46/70	Layer 6 Protocol Trace
		47/71	Layer 7 Protocol Trace
		51/81	User Trace 1
		52/82	User Trace 2
		53/83	User Trace 3
		54/84	User Trace 4
		55/85	User Trace 5
		56/86	User Trace 6
		57/87	User Trace 7
		61/97	TIM package standard stats
		62/98	TIM package aux
			Run/Freeze mode (bit 9)
		100/256	Freeze mode
		0	Run mode

61.2 Prompt Line

Access to the prompt line is always available via the *display_prompt* routine, or its softkey equivalent, the PROMPT action. Attributes may not be assigned to a prompt created via either of these methods. (To create a prompt with attributes, use the *pos_cursor* and *displayf* routines.) Prompts appear on whatever screen is active at the time the prompt is written, including trace screens. With one exception, movement to another display erases the prompt. The only screen which retains the most recent prompt is the Display Window.

You may also position the cursor to the prompt line in the Display Window via the *pos_cursor* routine. The initial position of the cursor in the Display Window is at the beginning of the prompt line—row zero, column zero. Anything written to this cursor position in the Display Window will appear as a prompt on any one of the other display screens (assuming one of them is active at the time the message is written). Position the cursor below the prompt line for messages intended for the Display Window only.

Trace buffers retain no record of prompts. When you write to a trace screen, the initial position of the cursor is the line immediately below the prompt line—row one. Since you may not position the cursor in trace buffers, all messages written to trace buffers are appended at the end of the buffer. You may never access the prompt line via *tracef* (or *tracec* or *traces*) routines.

61.3 Display Window

The Display Window preserves one screen, including the prompt line, of user-entered messages. When the end of the display screen is reached, the previous messages are overwritten, beginning at row one (the line below the prompt line).

> NOTE: Use the keyboard variables and the *send_key* routine explained in Section 69, Other Library Tools, to program the Run-mode use of ⇩ and ⇧ in the Display Window. (For other Run-mode screens, these keys control the playback speed of disk data.)

(A) Variables

There are variables accessible to the user which provide information about the Display Window. Table 61-2 lists the variables and their possible values. Two variables indicate the current position of the cursor: *current_line* stores the row number and *current_col* stores the column number. To find out which attributes are active in the Display Window, check the values stored in *window_color* and *window_modifier*. Color is stored in the high byte of the two-byte variable *window_color*. Enhancements are stored in the low byte. The current font code can be found in *window_modifier*.

> NOTE: Attributes assigned via the *%m* conversion specifier (refer to *tracef*-routine input) to characters in trace buffers will not alter the values of *window_color* and *window_modifier*. These variables refer to the Display Window only.

The variable *display_window_buffer* provides the user with access to the display-window buffer. This variable is an array of 1,088 *longs*. Each element in the array contains one byte of character data and three bytes of attributes. The attributes are determined by the current values of *window_color* and *window_modifier*.

Table 61-2
Display Window Variables

Type	Variable	Value (hex/decimal)	Meaning
extern unsigned short	current_line	*0-10/0-16*	Contains the current row number of the cursor position in the Display Window. Line Setup configured for emulate or monitor mode.
extern unsigned short	current_col	*0-3f/0-63*	Contains the current column number of the cursor position in the Display Window. Line Setup configured for emulate or monitor mode.
extern unsigned short	window_color		Two-byte variable. Current color selections are indicated in the low byte. Current enhancements are indicated in the high byte. Written to by %*m* conversions. Attributes are copied into data words in Display Window. Line Setup configured for emulate or monitor mode.
			Isolate bits of interest via *bitwise and*ing (&) of mask with variable. Compare result to value column. For example, underline attribute mask = 0x100. Therefore *window_color & 0x100* equals *0* (underline off) or *0x100* (underline on). Line Setup configured for emulate or monitor mode.
			background color mask = 7 (bits 1-3):
		0	black
		1	blue
		2	green
		3	cyan
		4	red
		5	magenta
		6	yellow
		7	white
			foreground color mask = 0x38 (bits 4-6):
		0	black
		8	blue
		10/16	green
		18/24	cyan
		20/32	red
		28/40	magenta
		30/48	yellow
		38/56	white

Table 61-2 (continued)

Type	Variable	Value (hex/decimal)	Meaning
	(window_color continued)		*color blink mask = 0x40 (bit 7):*
		0 40/64	no blink blink
			color strike-thru mask = 0x80 (bit 8):
		0 80/128	no strike-thru strike-thru
			overline mask = 0x100 (bit 9):
		0 100/256	no overline overline
			blank mask = 0x200 (bit 10):
		0 200/512	no blank blank
			underline mask = 0x400 (bit 11):
		0 400/1024	no underline underline
			reverse image mask = 0x800 (bit 12):
		0 800/2048	no reverse image reverse image
			hex mask = 0x1000 (bit 13):
		0 1000/4096	no hex hex
			low intensity mask = 0x2000 (bit 14):
		0 2000/8192	no low intensity low intensity (RS-170 output)
			monochrome blink mask = 0x4000 (bit 15):
		0 4000/16384	no monochrome blink monochrome blink
			monochrome strike-thru *mask = 0x8000 (bit 16):*
		0 8000/32768	no monochrome strike-thru monochrome strike-thru

Table 61-2 (continued)

Type	Variable	Value (hex/decimal)	Meaning
extern unsigned char	window_modifier		Contains the current modifiers. Line Setup configured for emulate or monitor mode.
			font mask = 7 (bits 1-3):
		0	ASCII
		1	special graphic character set (refer to Table 61-5)
		2	primary font—code selected on Line Setup
		3	alternate font—current implementation is for call-setup phase in X.21 (ASCII)
		7	hexadecimal
extern unsigned long	display_window_buffer [1088]		Array of 32-bit words that make up the one-screen Display Window. Each word contains three bytes of attributes and a one-byte character. Refer to Table 61-4. Line Setup configured for emulate or monitor mode.

(B) Structures

Once the data word is written to the buffer as an element in the *display_window_buffer* array, it can be accessed and written to—and therefore changed—the same as any other location in memory. There is an *extern* array, *display_window_index_buffer[17]*, which provides a method of informing the display controller on the CPM card that the display needs to be updated. The structure of this array is shown in Table 61-3.

Each element in the *display_window_index_buffer* array represents a horizontal row or line in the Display Window. Each element is a structure with two variables, *mpm* and *cpm*. The first variable in the structure, *mpm*, increments automatically whenever a line in the display-window buffer is updated by a display routine. (If you write to the buffer directly without using one of the display routines, you must increment this variable "manually.") Its particular value at any moment is not important. What is significant is whether or not the value of the second variable in the structure, *cpm*, is the same as *mpm*. The processor on the CPM compares these two variables (for each line) periodically to determine if a line in the Display Window needs to be rewritten. If the values of the two variables do not match, it means that a line updated in memory now needs to be updated by the CPM display-controller software. After the display is changed, the value of *mpm* is copied automatically into *cpm*.

Table 61-3
Display Window Buffer Structures

Type	Variable	Value (hex/decimal)	Meaning
Structure Name:	**display_window_index_buffer [17]**		An array of structures used for detecting changes to the display-window buffer. There are seventeen elements in the array, one for each line in the Display Window. When a change is made to a line in the display-window buffer, the corresponding element in the array is accessed. If a *displayf* routine changes line 3, *display_window_ index_buffer[3].mpm* is automatically incremented. The CPM detects the difference between *display_window_index_buffer [3].mpm* and *display_window_index_buffer [3].cpm* and updates line 3 in the Display Window. Declared as type *extern struct*. You must increment an *mpm* variable manually when you write *directly* (not via a *displayf* routine) to the Display Window.
unsigned char	mpm	*0-ff/0-255*	When the MPM updates a line in the display-window buffer, this variable is incremented.
unsigned char	cpm	*0-ff/0-255*	The CPM checks the value of this variable against the value of *mpm*. If they are different, the value in *mpm* is copied into *cpm*. The updated line in MPM is then presented on the display-window screen.

(C) Routines

You may position the cursor before output is generated on the screen via the *pos_cursor* and *restore_cursor* routines. The *pos_cursor* routine positions the cursor at the row and column you specify. The *restore_cursor* routine returns the cursor to a previous location.

One routine, *displayf*, allows you to add attributes to messages in the Display Window, including the prompt line. These attributes are listed in Table 61-4.

displayc

Synopsis

```
extern void displayc(character);
const char character;
```

Description

The *displayc* routine outputs a single ASCII character to the Display Window screen. The placement of the output on the screen may be controlled via the *pos_cursor* routine. Attributes may not be used in *displayc*.

Inputs

The parameter value may be given as a hexadecimal, octal, or decimal constant; as an alphanumeric constant inside of single quotes; or as a variable. A hexadecimal value must be preceded by the prefix 0x or 0X; an octal value must be preceded by the prefix 0. If no prefix appears before the input, the number is assumed to be decimal. Valid numeric entries are 00 to 127, decimal. An alphanumeric character placed between single quotes will be output as is to the display.

Example

The *displayc* entries on the left output the character given on the right, at the cursor location on the Display Window screen:

displayc('a');	a
displayc(65);	A
displayc(0x65);	e
displayc(065);	5

displayf

Synopsis

```
extern int displayf(format_ptr, . . . );
const char * format_ptr;
```

Description

The *displayf* routine writes output to the Display Window screen, under control of the string pointed to by *format_ptr* that specifies how subsequent arguments are converted for output. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but otherwise ignored. The *displayf* routine returns when the end of the format string is encountered. The placement of the output on the screen may be controlled via the *pos_cursor* routine.

Inputs

The format is composed of zero or more directives: ordinary characters (not %), which are copied unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments. Each conversion specification is introduced by the character %. After the %, the following appear in sequence:

- Zero or more *flags* that modify the meaning of the conversion specification. The flag characters and their meanings are:

  - The result of the conversion will be left-justified within the field.

  + The result of a signed conversion will always begin with a plus or minus sign.

  space If the first character of a signed conversion is not a sign, a space will be prepended to the result. If the *space* and + flags both appear, the *space* flag will be ignored.

  # The result is to be converted to an "alternate form." For d and i conversions, the flag has no effect. For o conversion, it increases the precision to force the first digit of the result to be a zero. For x (or X) conversion, a nonzero result will have 0x (or 0X) prepended to it. For u conversions, the argument is displayed in small hex characters. For example, displayf ("%#u", 258); yields ${}^{0}{}_{1}{}^{0}{}_{2}$. For c and s conversions, if the argument contains a newline character, it is displayed as ␊.

- An optional decimal integer specifying a minimum *field width*. If the converted value has fewer characters than the field width, it will be padded on the left (or right, if the left adjustment flag, described above, has been given) to the field width. The padding is with spaces unless the field width integer starts with a zero, in which case the padding is with zeros.

- An optional *precision* that gives the minimum number of digits to appear for the d, i, o, u, x, and X conversions, the maximum number of characters to be written from an array in an s conversion, or the number of characters to be written from an array in an H conversion (overriding the usual null-termination of strings). The precision takes the form of a period (.) followed by an optional decimal integer; if the integer is omitted, it is treated as zero. The amount of padding specified by the precision overrides that specified by the field width.

- An optional h specifying that a following d, i, o, u, x, or X conversion specifier applies to a *short int* or *unsigned short int* argument (the argument will have been promoted according to the integral promotions, and its value shall be converted to *short int* or *unsigned short int* before printing); or an optional l specifying that a following d, i, o, u, x, or X conversion specifier applies to a *long int* or *unsigned long int* argument. If an h or l appears with any other conversion specifier, it is ignored.

- A character that specifies the type of *conversion* to be applied. (Special AR extensions have been added.) The conversion specifiers and their meanings are:

d, i, o, u, x, X

The *int* argument is converted to signed decimal (d or i), unsigned octal (o), unsigned decimal (u), or unsigned hexadecimal notation (x or X); the letters abcdef are used for x conversion and the letters ABCDEF for X conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no characters.

c — The *int* argument is converted to an *unsigned char*, and the resulting character is written.

s — The argument shall be a pointer to a null-terminated array of 8-bit *chars*. Characters from the string are written up to (but not including) the terminating null character: if the precision is specified, no more than that many characters are written. The string may be an array into which output was written via the *sprintf* routine. (If the string pointed to is an array which has been written via the *stracef* routine, you must use *%b* rather than *%s* to display it.)

p — The argument shall be a pointer to void. The value of the pointer is converted to a sequence of printable characters, in this format: *0000:0000*. There are always exactly 4 digits to the right of the colon. The number of digits to the left of the colon is determined by the pointer's value and the precision specified. Use this conversion to display 80286 memory addresses. The 16-bit segment number will appear to the left of the colon and the 16-bit offset to the right.

% — A % is written. No argument is converted.

\n — Displays ␤. No argument is converted.

H — displays a character array (pointed to by the argument) as small hex characters. If precision is specified, it is used as the length of the array (overriding the usual null-termination of strings).

b The argument shall be a pointer to an array of 32-bit words. Characters from the string are written up to (but not including) the terminating word containing a null character: if the precision is specified, no more than that many words are written. If the string pointed to is an array into which output was written via the *stracef* routine, you must use *%b* rather than *%s* to display it. (To display the information in an array written to via *sprintf*, use *%s*.)

m The argument is a *long* integer that indicates attributes to be assigned to subsequent characters. Attributes stay "on" until they are specifically turned "off" with another *%m* conversion specifier. The lowest-order byte contains primarily font code. The next higher byte is not used to set attributes. (In the display-window buffer, this second byte is reserved for character coding.) The third byte holds color information. The high byte indicates which enhancements should be invoked.

Attributes are written automatically to *window_color* and *window_modifier* variables, then copied into subsequent 32-bit data words in the Display Window. Table 61-4 shows the format of this 32-bit word.

Attributes may not be assigned as a one-byte value. Even if you want to alter only one attribute setting, color for example, you must include it as part of a *long*. Append an "L" at the end of the hexadecimal code specifying attributes to indicate the value is a *long*.

NOTE: If you are specifying an attribute in a lower-order byte of the *long*, color for example, and you want the high byte (or bytes) to be zero, you may omit the high byte provided you have the "L" appended at the end of the hexadecimal code. The high byte (or bytes) will be left-padded with zeroes. For example, 0x200000L is converted to 0x00200000L. Associated characters will be displayed on a color monitor as green on a black background, as dictated by the hexadecimal 20 in the third byte. Enhancements are controlled in the high byte, now assigned a value of zero. Any enhancements previously turned "on" will be turned "off."

If a conversion specification is invalid, the behavior is undefined.

If any argument is or points to an aggregate (except for an array of characters using *%s* conversion or any pointer using *%p* conversion), the behavior is undefined.

In no case does a nonexistent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is expanded to contain the conversion result.

Returns

The *displayf* routine returns the number of characters displayed.

Example

To display a date and time in the form "Sunday, July 3, 10:02," where weekday and month are pointers to strings:

```
LAYER: 1
{
 unsigned char weekday [10];
 unsigned char month [10];
 unsigned short day;
 unsigned char hour;
 unsigned char min;
}
     STATE: output_to_display_window
        CONDITIONS: KEYBOARD " "
        ACTIONS:
        {
         displayf( "%s, %s %d, %.2d:%.2d\n", weekday, month, day, hour, min);
        }
```

sprintf

The *sprintf* routine is similar to the *displayf* routine. *displayf* writes output with or without attributes directly to the *Display Window*. *sprintf,* fully documented in Section 64.3, writes output to a *character array* in which attributes are not supported. This routine is useful for writing formatted output to a display, printer, or file.

See also *stracef* in Section 61.4(C).

Table 61-4
Display Window/Trace Buffer 32-Bit Data Word

Bit	Mask (hex)†	Input (hex)††	Meaning
			Modifier attributes, font for example, are contained in the low byte of the 32-bit word.
1-3	0000000<u>7</u>L		<u>Font</u>:
		000000<u>00</u>L	ASCII
		000000<u>01</u>L	special graphic character set (refer to Table 61-5)
		000000<u>02</u>L	primary font—code selected on Line Setup
		000000<u>03</u>L	alternate font—current implementation is for call-setup phase in X.21 (ASCII)
		000000<u>07</u>L	hexadecimal
4	000000<u>08</u>L		<u>Special character indicator</u>: (used in trace buffer only; should not be altered by user)
		000000<u>00</u>L	only value in *modifier* in trace buffer header
		000000<u>08</u>L	Character is not displayable but contains control info used internally by the trace logic. When a "\n" is included in a *tracef* routine, for example, a new line is generated, but nothing is displayed on the trace screen. The *tracef* routine automatically sets this bit before the 32-bit word is written into *trace_buf.array*.
5-8	000000<u>f0</u>L	000000<u>00</u>L	unused, but should be zero
9-16	0000<u>ff</u>00L	0000<u>00</u>00L	*Character data* is contained in the second byte of the *long* word. Input should be 00 in all %*m* conversions.

† Use the masks to change attributes of characters in the Display Window or trace buffer. In the Display Window, characters are represented in the second byte of the *longs* that comprise the 1,088 array elements in *display_window_buffer*. In the *trace_buf* structure, the characters are represented in the second byte of the *longs* that make up the *trace_buf.array*. To change one attribute of a character while leaving the others unchanged:

display_window_buffer[position] = ((display_window_buffer[position] & (~attribute-mask)) | input);

To change only the font of the twenty-first character in the trace buffer from its current setting to the special graphic font, for example:

l2_trbuf.array[20] = ((trace_buf.array[20] & (~0x00000007L)) | 0x00000001L);

*And*ing the character with the mask will indicate the current setting of an attribute:

If *(l2_trbuf.array[20] & 0x00000007L)* equals 2, then the 21st character in the Trace 2 user-trace buffer is being displayed in the font selected on the Line Setup menu.

†† In *displayf* routines, the %*m* conversion specifier writes input to the *window_color* and *window_modifier* variables. These variables are copied into subsequent data words in the Display Window. In *tracef* routines, the %*m* conversion specifier writes input to *trace_buffer_header*. The header is then copied into each subsequent data word in the buffer. Combine attributes via hexadecimal addition.

Table 61-4 (continued)

Bit	Mask (hex)	Input (hex)	Meaning
			Color is contained in the third byte of the *long*. Combine color attributes via hexadecimal addition.
17-19	00070000L		Background color:
		00000000L	black
		00010000L	blue
		00020000L	green
		00030000L	cyan
		00040000L	red
		00050000L	magenta
		00060000L	yellow
		00070000L	white
20-22	00380000L		Foreground color:
		00000000L	black
		00080000L	blue
		00100000L	green
		00180000L	cyan
		00200000L	red
		00280000L	magenta
		00300000L	yellow
		00380000L	white
23	00400000L		Color blink:
		00000000L	no blink
		00400000L	blink
24	00800000L		Color strike-thru:
		00000000L	no strike-thru
		00800000L	strike-thru
			Enhance attributes, underlining for example, are contained in the high byte of the *long*. Combine enhancements via hexadecimal addition.
25	01000000L		Overline: (for monochrome and color)
		00000000L	no overline
		01000000L	overline
26	02000000L		Blank:
		00000000L	monochrome display, color display
		02000000L	monochrome no display, color display
27	04000000L		Underline: (for monochrome and color)
		00000000L	no underline
		04000000L	underline

Table 61-4 (continued)

Bit	Mask (hex)	Input (hex)	Meaning
28	08000000L		Monochrome reverse image:
		00000000L	no reverse image
		08000000L	reverse image
29	10000000L		Hex:
		00000000L	no hex
		10000000L	hex
30	20000000L		Monochrome low intensity:
		00000000L	no low intensity
		20000000L	low intensity (RS-170 interface)
31	40000000L		Monochrome blink:
		00000000L	no blink
		40000000L	blink
32	80000000L		Monochrome strike-thru:
		00000000L	no strike-thru
		80000000L	strike-thru

Table 61-5
Special Graphic Character Set†

Display	Input (hex/decimal)	Display	Input (hex/decimal)
˥	0	┐	1a/26
ʃ	1	┼	1b/27
‾	2	─	1c/28
_	3	│	1d/29
⟫	4	┬	1e/30
⟪	5	┴	1f/31
⁙	6	┤	20/32
[S]	7	├	21/33
⊠	8	▨	22/34
[F]	9	▒	23/35
[A]	a/10	█	24/36
[G]	b/11	▨	25/37
[B]	c/12	▒	26/38
[M]	d,11/13,17	▌	27/39
↵	e/14	▨	28/40
…	f/15	▒	29/41
·	10/16	▌	2a/42
˥	12/18	▨	2b/43
ʃ	13/19	▒	2c/44
‾	14/20	▬	2d/45
_	15/21	▨	2e/46
[B]	16/22	▒	2f/47
┘	17/23	▬	30/48
└	18/24	*(space)*	31/49
┌	19/25		

† Written to the Display Window or a trace buffer when low (modifier) byte of 32-bit data word = 0x01.

Table 61-5 (continued)

Display	Input (hex/decimal)	Display	Input (hex/decimal)
¥	80/128	コ	9a/154
。	81/129	サ	9b/155
「	82/130	シ	9c/156
」	83/131	ス	9d/157
、	84/132	セ	9e/158
・	85/133	ソ	9f/159
ヲ	86/134	タ	a0/160
ァ	87/135	チ	a1/161
ィ	88/136	ツ	a2/162
ゥ	89/137	テ	a3/163
ェ	8a/138	ト	a4/164
ォ	8b/139	ナ	a5/165
ャ	8c/140	ニ	a6/166
ュ	8d/141	ヌ	a7/167
ョ	8e/142	ネ	a8/168
ッ	8f/143	ノ	a9/169
ー	90/144	ハ	aa/170
ア	91/145	ヒ	ab/171
イ	92/146	フ	ac/172
ウ	93/147	ヘ	ad/173
エ	94/148	ホ	ae/174
オ	95/149	マ	af/175
カ	96/150	ミ	b0/176
キ	97/151	ム	b1/177
ク	98/152	メ	b2/178
ケ	99/153	モ	b3/179

Table 61-5 (continued)

Display	Input (hex/decimal)	Display	Input (hex/decimal)
ヤ	b4/180	Ä	ce/206
ユ	b5/181	Å	cf/207
ヨ	b6/182	É	d0/208
ラ	b7/183	æ	d1/209
リ	b8/184	Æ	d2/210
ル	b9/185	ô	d3/211
レ	ba/186	ö	d4/212
ロ	bb/187	ò	d5/213
ワ	bc/188	û	d6/214
ン	bd/189	ù	d7/215
゛	be/190	ÿ	d8/216
・	bf/191	Ö	d9/217
ç	c0/192	Ü	da/218
ü	c1/193	¢	db/219
é	c2/194	£	dc/220
â	c3/195	B	dd/221
ä	c4/196	₧	de/222
à	c5/197	ƒ	df/223
å	c6/198	á	e0/224
ç	c7/199	í	e1/225
ê	c8/200	ó	e2/226
ë	c9/201	ú	e3/227
è	ca/202	ñ	e4/228
ï	cb/203	Ñ	e5/229
î	cc/204	ª	e6/230
ì	cd/205	º	e7/231

Table 61-5 (continued)

Display	Input (hex/decimal)	Display	Input (hex/decimal)
¿	e8/232	i	ed/237
⌐	e9/233	¨	ee/238
¬	ea/234	§	ef/239
½	eb/235	•	f0/240
¼	ec/236		

displays

Synopsis

```
extern void displays(string_ptr);
const char * string_ptr;
```

Description

The *displays* routine writes output to the Display Window screen, under control of the string that is pointed to by *string_ptr*. The *displays* routine returns when the end of the string is encountered. The placement of the output on the screen may be controlled via the *pos_cursor* routine. Attributes may not be used in *displays*.

Inputs

The input is a pointer to a string composed of zero or more ordinary characters. Octal or hexadecimal values also may be included in the string, with octal preceded by \ and hex by \x. Pad each value to three integers with leading zeroes.

Example

The following entry

```
pos_cursor( 0, 0 );
displays("End of test.");
```

produces the following output on the prompt line:

```
End of test.
```

The following coding produces the same output:

```
pos_cursor( 0, 0 );
const char * string_ptr;
string_ptr = "End of test.";
displays (string_ptr);
```

display_prompt

Synopsis

```
extern void display_prompt(string_ptr);
const char * string_ptr;
```

Description

The *display_prompt* routine displays a designated string at the beginning of the prompt line. The cursor is automatically positioned at row zero, column zero. Once the prompt is written, the cursor is returned to its previous position. The softkey equivalent of this routine is the PROMPT action. The prompt is visible on whichever display screen is active at the time the prompt is written. The most recent prompt is retained in the Display Window. Attributes may not be used in *display_prompt*.

Inputs

The input is a pointer to a string composed of zero or more ordinary characters. Octal or hexadecimal values also may be included in the string, with octal preceded by \ and hex by \x. Pad each value to three integers with leading zeroes.

Example

Refer to the example provided for the *displays* routine. The same string could be output to the same position without calling the *pos_cursor* routine:

```
display_prompt("End of test.");
```

or

```
const char * string_ptr;
string_ptr = "End of test.";
display_prompt (string_ptr);
```

pos_cursor

Synopsis

```
extern unsigned int pos_cursor(row, column);
unsigned char row;
unsigned char column;
```

Description

This routine positions the cursor on the Display Window screen by row and column numbers.

> **NOTE:** The *pos_cursor* routine may not be used to position the cursor on trace screens.

Inputs

The first parameter is the row number. Possible values: 0–16. (The top line of the screen is reserved for header information and cannot be written to.)

The second parameter is the column number. Possible values: 0–63.

Returns

The *pos_cursor* routine returns the previous cursor position in the form of an *unsigned int*. The high byte contains the row number; the low byte identifies the column number.

Example

To position the cursor at the far left edge of the prompt line on the Display Window, enter zero for both parameters.

```
LAYER: 4
    STATE: write_to_display
       CONDITIONS: KEYBOARD " "
       ACTIONS:
       {
        pos_cursor(0,0);
        displays("Display on prompt line.");
       }
```

restore_cursor

Synopsis

extern void restore_cursor(position);
unsigned int position;

Description

The *restore_cursor* routine returns the cursor to a previous position.

> **NOTE:** The *restore_cursor* routine may not be used to position the cursor on trace screens.

Inputs

The only input is an *unsigned int* in the same form that is used by the returned value of the *pos_cursor* routine. The high byte identifies the row number. The low byte identifies the column number.

Example

Suppose the cursor is located in the middle of the Display Window. You want to write a message to the prompt line, but return to your previous location on the screen to continue your display.

```
{
unsigned int previous;
}
    STATE: display
      CONDITIONS: KEYBOARD " "
      ACTIONS:
      {
       pos_cursor(8,0);
       displays("This line begins on row 8, column 0 of the Display Window.");
       previous = pos_cursor(0,0);
       displays("This sentence is on the prompt line.");
       restore_cursor(previous);
       displays("This sentence begins on row 8, column 58 of the Display Window, the
       position of the cursor at the time pos_cursor(0,0) was called.");
      }
```

61.4 Program and User Traces

Unless their sizes are increased, Program Trace and the User Traces retain a maximum of 4096 characters, equivalent to four full screens when every character space is used. (See Section (B)2. below on increasing the size of trace buffers.) When a buffer's limit is reached, new characters written to the end of the buffer force out the same number of characters from the beginning of the buffer. The prompt line is not part of these buffers. Messages are appended to the end of the buffers. In Freeze mode you may scroll through the buffer using the cursor keys.

You write messages to the User Traces only by using C routines. The Run-mode softkeys for User Traces—USER TR, TRACE 1, TRACE 2, TRACE 3, TRACE 4, TRACE 5, TRACE 6, TRACE 7—appear when the buffers are used in a program.

(A) Variables

There are no *extern* variables associated exclusively with Traces.

(B) Structures

1. *Declaring trace buffers.* The trace routines that write to any of the trace buffers require a pointer to the appropriate trace buffer as input. To point to one of the trace buffers, you must first have declared it as a structure. The structure that is common to trace buffers is named *trace_buf*. This structure is already declared in a file called *trace_buf.h* located in the *HRD/sys/include* directory. The *trace_buf* structure contains another structure, *trace_buffer_header*, which also is declared in the *trace_buf.h* file. (These structures are explained in Table 61-6.) Use the *#include* pre-processor directive to include both declarations in your program.

   There are eight trace buffers available (including the Program Trace), each one having its own display screen. All are instances of the *trace_buf* structure. Declare each one you use as an *extern struct*, as in this example:

   ```
   extern struct trace_buf l1_trbuf;
   ```

   The names of all the trace buffers are listed in Table 61-6.

Table 61-6
Trace Buffer Structures

Type	Variable	Value (hex/decimal)	Meaning
Structure Name: trace_buffer_header			Structure of a header for trace buffers. Declared as type *extern struct*. Declared automatically if a softkey-entered TRACE action is taken. Contained in the structure of the trace buffer. Declaration contained in file named *HRD/sys/include/trace_buf.h*. Written to by *%m* conversion specifier. Because it is an extern structure, values of component variables should not be altered directly by the user. In some instances, e.g., altering array size, the result could be a crash.
unsigned short	logical_end	*0-fff/0-4095*	end of data within the buffer. Maximum value is one less than the *array_size*.
unsigned short	logical_end_wrap_count	0 *non-zero*	trace buffer is not full trace buffer is full. As new lines are written to the end of the trace buffer, lines at the beginning of the buffer are removed.
unsigned char	modifier		Special-character indicator bit and bit 8 must be zero. For other specific values and their meanings,see Table 61-4.
unsigned char	color	*0-ff/0-255*	For specific values and their meanings, see Table 61-4.
unsigned char	enhance	*0-ff/0-255*	For specific values and their meanings, see Table 61-4.
unsigned short	write_lock	*0-ffff/0-65535*	prevents two processes from writing to the same buffer at the same time. Should not be altered by user or future access to the trace buffers may be locked out.
unsigned short	array_size	1000/4096	size of buffer; at present only one value
unsigned char	line_size	*0-3f/0-63*	number of characters in last line in buffer
unsigned char	spare	0	reserved for future use
Structure Name: trace_buf			Structure of a trace buffer. Declared as type *extern struct*. Declared automatically if a softkey-entered TRACE action is taken. Declaration contained in file named *HRD/sys/include/trace_buf.h*.
struct trace_buffer_header	hdr		structure of the trace-buffer header described above
unsigned long	array [4096]		array of data words in the buffer

Table 61-6 (continued)

Type	Variable	Value (hex/decimal)	Meaning
Structure Name:	**prog_trbuf**		Structure of the Program Trace buffer, an instance of the *trace_buf* structure declared in file named *HRD/sys/include/trace_buf.h*. Declared as type *extern struct trace_buf*. Declared automatically if a softkey-entered TRACE action is taken. Writing to this buffer makes the Run-mode PROG TR softkey appear.
struct trace_buffer_header	hdr		structure of the trace-buffer header described above
unsigned long	array [4096]		array of data words in the buffer
Structure Name:	**l1_trbuf**		Structure of one of seven user trace buffers, an instance of the *trace_buf* structure declared in file named *HRD/sys/include/trace_buf.h*. Declared as type *extern struct trace_buf*. Writing to this buffer causes the Run-mode TRACE 1 softkey appear.
struct trace_buffer_header	hdr		structure of the trace-buffer header described above
unsigned long	array [4096]		array of data words in the buffer
Structure Name:	**l2_trbuf**		Structure of one of seven user trace buffers, an instance of the *trace_buf* structure declared in file named *HRD/sys/include/trace_buf.h*. Declared as type *extern struct trace_buf*. Writing to this buffer causes the Run-mode TRACE 2 softkey appear.
struct trace_buffer_header	hdr		structure of the trace-buffer header described above
unsigned long	array [4096]		array of data words in the buffer
Structure Name:	**l3_trbuf**		Structure of one of seven user trace buffers, an instance of the *trace_buf* structure declared in file named *HRD/sys/include/trace_buf.h*. Declared as type *extern struct trace_buf*. Writing to this buffer causes the Run-mode TRACE 3 softkey appear.
struct trace_buffer_header	hdr		structure of the trace-buffer header described above
unsigned long	array [4096]		array of data words in the buffer
Structure Name:	**l4_trbuf**		Structure of one of seven user trace buffers, an instance of the *trace_buf* structure declared in file named *HRD/sys/include/trace_buf.h*. Declared as type *extern struct trace_buf*. Writing to this buffer causes the Run-mode TRACE 4 softkey appear.
struct trace_buffer_header	hdr		structure of the trace-buffer header described above
unsigned long	array [4096]		array of data words in the buffer

Table 61-6 (continued)

Type	Variable	Value (hex/decimal)	Meaning
Structure Name: I5_trbuf			Structure of one of seven user trace buffers, an instance of the *trace_buf* structure declared in file named *HRD/sys/include/trace_buf.h*. Declared as type *extern struct trace_buf*. Writing to this buffer causes the Run-mode TRACE 5 softkey appear.
struct trace_buffer_header	hdr		structure of the trace-buffer header described above
unsigned long	array [4096]		array of data words in the buffer
Structure Name: I6_trbuf			Structure of one of seven user trace buffers, an instance of the *trace_buf* structure declared in file named *HRD/sys/include/trace_buf.h*. Declared as type *extern struct trace_buf*. Writing to this buffer causes the Run-mode TRACE 6 softkey appear.
struct trace_buffer_header	hdr		structure of the trace-buffer header described above
unsigned long	array [4096]		array of data words in the buffer
Structure Name: I7_trbuf			Structure of one of seven user trace buffers, an instance of the *trace_buf* structure declared in file named *HRD/sys/include/trace_buf.h*. Declared as type *extern struct trace_buf*. Writing to this buffer causes the Run-mode TRACE 7 softkey appear.
struct trace_buffer_header	hdr		structure of the trace-buffer header described above
unsigned long	array [4096]		array of data words in the buffer

2. *Sizing trace buffers*. There is a preprocessor *#pragma* which allows the user to configure the size of the data array in each user trace buffer. The syntax is TRACE-NUMBER SIZE TRACE-NUMBER SIZE. . . . Trace number 0 refers to the Program Trace buffer, and trace-number "*" allows all trace-buffer arrays to be set at once. All sizes are given in terms of four-byte array elements.

   The example below first sets all trace-buffer arrays to 16,000 elements, and then down-sizes array number 3 to 2,048 elements.

   ```
   #pragma tracebuf * 16000 3 2048
   ```

   When a trace buffer is declared, its array will have the size specified in the *#pragma tracebuf* directive. If the buffer was not referenced in a *#pragma tracebuf* directive, its array size will default to 4,096. The maximum size for a trace-buffer array is 16,381 elements. If you specify a size that is too small or too large, an error message will be displayed.

(C) Routines

The four trace routines are *tracec*, *tracef*, *stracef*, and *traces*. These routines are defined below. The softkey TRACE action is built on the *tracef* routine.

The first argument in three of the trace routines is the address of the trace buffer into which you want output written. Each time you call a trace routine, *tracef* for example, variables in the named trace–buffer structure are updated. Those variables which store attributes are updated when the *%m* conversion specifier is used in the *tracef* routine parameter. When *%m* is not present, the routine applies the attributes currently stored in the *color*, *modifier*, and *enhance* variables.

The second argument in all trace routines is the character, string, or format pointer to the data that will be written to the selected trace buffer.

The *tracef* routine allows you to add attributes to messages on the Program Trace screen and User Traces. These attributes are listed in Table 61-4.

Each trace operation appends output to the end of the trace buffer. You may not use the *pos_cursor* routine to position the cursor on any trace screen. New lines (or blank lines) may be generated via the "\n" nonliteral. Put the "\n" nonliteral at the end of the string to generate a leading blank line on the selected trace screen:

```
tracef(&prog_trbuf, "This trace message will generate a leading blank line.\n");
```

During real–time display, this line moves just ahead of the freshest trace message and continuously overwrites the oldest one. If you put the "\n" sequence at the beginning of the format string, no leading blank line will help you distinguish new messages from the old:

```
tracef(&prog_trbuf, "\nThis message will not generate a leading blank line.");
```

tracec

Synopsis

```
extern void tracec(trace_buffer_ptr, character);
extern struct trace_buf * trace_buffer_ptr;
const char character;
```

Description

The *tracec* routine outputs a single ASCII character to the trace screen indicated.

Inputs

The first parameter is a pointer to the trace buffer into which the character will be written.

For the second parameter, see the *displayc* routine.

Example

In this instance, output will be written to the Program Trace screen.

```
{
#include <trace_buf.h>
extern struct trace_buf prog_trbuf;
}
LAYER: 2
    STATE: display_to_prog_tr
        CONDITIONS: KEYBOARD " "
        ACTIONS:
        {
         tracec(&prog_trbuf, 'a');
         tracec(&prog_trbuf, '\n');
         tracec(&prog_trbuf,65);
         tracec(&prog_trbuf, '\n');
         tracec(&prog_trbuf,0x65);
         tracec(&prog_trbuf, '\n');
         tracec(&prog_trbuf,065);
        }
```

When the user views the PROG TR screen, the output will look like this:

```
a
A
e
5
```

tracef

Synopsis

```
extern int tracef(trace_buffer_ptr, format_ptr, . . . );
extern struct trace_buf * trace_buffer_ptr;
const char * format_ptr;
```

Description

The *tracef* routine writes output to a specified trace screen, under control of the string, pointed to by *format_ptr*, that specifies how subsequent arguments are converted for output. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but otherwise ignored. The *tracef* routine returns when the end of the format string is encountered.

Inputs

The first parameter is a pointer to the trace buffer into which the output will be written.

For the second parameter, see the *displayf* routine. Placement of "\n" in the format string of a call to *tracef* generates a blank new line on the selected trace screen. (In a *displayf* routine, the newline character does not blank the new line.)

Attributes are written via the *%m* conversion specifier to *trace_buf.hdr.modifier*, *trace_buf.hdr.color*, and *trace_buf.hdr.enhance*. The attributes are copied from these variables into subsequent 32-bit data words in the Program Trace and User Traces. Table 61-4 shows the format of this 32-bit word.

Returns

The *tracef* routine returns the number of characters displayed, or a negative value if the unit is in freeze mode.

Example

This program traces X.29 PAD-control messages in DTE and DCE data packets. The letters "DCE" are underlined in the DCE trace lines.

```
LAYER: 3
{
   #include <trace_buf.h>
   extern struct trace_buf l3_trbuf;
   extern unsigned char * m_packet_info_ptr;
   extern unsigned short m_packet_lcn;
   unsigned char pad_ctrl_msg;
}
   STATE: packet_type
      CONDITIONS: DTE DATA Q= 1
      ACTIONS:
      {
         pad_ctrl_msg = m_packet_info_ptr[0];
         tracef (&l3_trbuf, "DTE LCN:%.3x PAD MSG:%.2x\n", m_packet_lcn,
            pad_ctrl_msg);
      }
      CONDITIONS: DCE DATA Q= 1
      ACTIONS:
      {
         pad_ctrl_msg = m_packet_info_ptr[0];
         tracef (&l3_trbuf, "%mDCE%m LCN:%.3x PAD MSG:%.2x\n", 0x04000000L,
            0x00000000L, m_packet_lcn, pad_ctrl_msg);
      }
```

stracef

Synopsis

```
extern void stracef(array_ptr, string_ptr);
unsigned long array_ptr;
const char * string_ptr;
```

Description

The *stracef* routine is similar to the *tracef* routine, except that *stracef* writes output to a *variable*, while *tracef* writes output to a trace screen. The output is under control of the string pointed to by *string_ptr* that specifies how subsequent arguments are converted for output. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but otherwise ignored. The *stracef* routine returns when the end of the format string is encountered.

The *stracef* routine differs from *sprintf* in that it generates an array of *longs*, whereas *sprintf* generates an array of *chars*. When the *stracef* array is written to a trace buffer (or to the Display Window) it carries its predefined attributes along with it. An *sprintf* array, by contrast, will receive the attributes that are active in the buffer at the moment.

At the end of the output string, there will be a null character with the Special Character Indicator bit set in its *modifier* attribute-byte.

Inputs

The first parameter is a pointer to the variable into which output will be written. The array which will hold output must be declared as a *long*. By allocating 32 bits for each element, the array may accommodate attributes assigned via the *%m* conversion specifier. Attributes comprise 24 bits of the *long*. The preferred form of the declaration is *unsigned long name [100]*. The size and name of the array are user-determined.

For the second parameter, see the *displayf* routine.

Example

This program traces X.29 PAD-control messages for DTE and DCE data packets. The resulting trace is identical to the one generated by the example under *tracef*. Note that attributes that are turned on in an *stracef* array do not need to be turned off after the array has been brought, via the *%b* conversion specifier, into a *tracef* format string.

```
LAYER: 3
{
  #include <trace_buf.h>
  extern struct trace_buf l3_trbuf;
  extern unsigned char * m_packet_info_ptr;
  extern unsigned short m_packet_lcn;
  unsigned char pad_ctrl_msg;
  unsigned long source[4];
}
  STATE: packet_type
    CONDITIONS: DTE DATA Q= 1
    ACTIONS:
    {
      stracef (source, "%s", "DTE");
    }
    NEXT_STATE: pad_msg_trace
    CONDITIONS: DCE DATA Q= 1
    ACTIONS:
    {
      stracef (source, "%m%s", 0x04000000L, "DCE");
    }
    NEXT_STATE: pad_msg_trace
  STATE: pad_msg_trace
    CONDITIONS: ENTER_STATE
    ACTIONS:
    {
      pad_ctrl_msg = m_packet_info_ptr[0];
      tracef (&l3_trbuf, "%b LCN:%.3x PAD MSG:%.2x\n", source, m_packet_lcn,
        pad_ctrl_msg);
    }
    NEXT_STATE: packet_type
```

traces

Synopsis

```
extern void traces(trace_buffer_ptr, string_ptr);
extern struct trace_buf trace_buffer_ptr;
const char * string_ptr;
```

Description

The *traces* routine writes output to a specified trace screen, under control of the string that is referenced by *string_ptr*. The *traces* routine returns when the end of the string is encountered.

Inputs

The first parameter is a pointer to the trace buffer into which the output will be written.

For the second parameter, see the *displays* routine.

Example

In this instance, output will be written to the TRACE 1 screen.

The following entry

```
{
#include <trace_buf.h>
extern struct trace_buf l1_trbuf;
}
LAYER: 1
    STATE: any
        CONDITIONS: KEYBOARD " "
        ACTIONS:
        {
         traces(&l1_trbuf, "End of test.");
        }
```

produces the following output on the TRACE 1 trace screen:

End of test.

The following coding produces the same output:

```
{
#include <trace_buf.h>
extern struct trace_buf l1_trbuf;
}
LAYER: 1
    STATE: any
        CONDITIONS: KEYBOARD " "
        ACTIONS:
        {
         const char * string_ptr;
         string_ptr = "End of test.";
         traces (&l1_trbuf, string_ptr);
        }
```

61.5 Attributes

Attributes are written to the Display Window and to the trace buffers in 32-bit words that include 8 bits of character data (the second-lowest byte) and 24 bits of attributes. The format of the 32-bit data word, given in Table 61-4, is the same for the Display Window and for the trace buffers.

In *displayf* routines, the *%m* conversion specifier writes input to *window_color* and *window_modifier* variables. These variables are then copied into data words written to the Display Window by string pointers in this and subsequent *displayf* routines. See Figure 61-1.

In *tracef* routines, the *%m* conversion specifier writes input to the *trace_buffer_header* structure for a particular user-trace buffer. The header is then copied into each data word written to the particular user buffer by string pointers in this and subsequent *tracef* routines. See Figure 61-2.

(A) Applying Attributes As Data Is Buffered

There are two ways an attribute may be assigned to a character in the Display Window. One way uses the *%m* conversion specifier to assign attributes to the *window_color* and *window_modifier* variables. This program, for example, includes a *displayf* routine that uses the *%m* conversion specifier to write underlined data to the Display Window:

```
STATE: apply_attribute_to_window_color_variable
   CONDITIONS: ENTER_STATE
   ACTIONS:
   {
    pos_cursor (1,0);
    displayf ("%mThis data is underlined in the Display Window.", 0x04000000L);
   }
```

The chart in Table 61-4 shows the hex value 04000000L in the "input" column alongside the underline attribute. This means that when the value 0x04000000L is input to the conversion specifier *%m*, an underline attribute is applied to the current *displayf* string and any that follow until the attribute is turned off. The underline attribute actually is applied to the external *window_color* variable. See Table 61-2. The *window_color* and *window_modifier* variables lend their attributes to every character that is written in a format string to the Display Window. In Run mode if the user presses the softkey for DSP WND, he will see his underlined string. Subsequent characters or strings written to the Display Window also will be underlined.

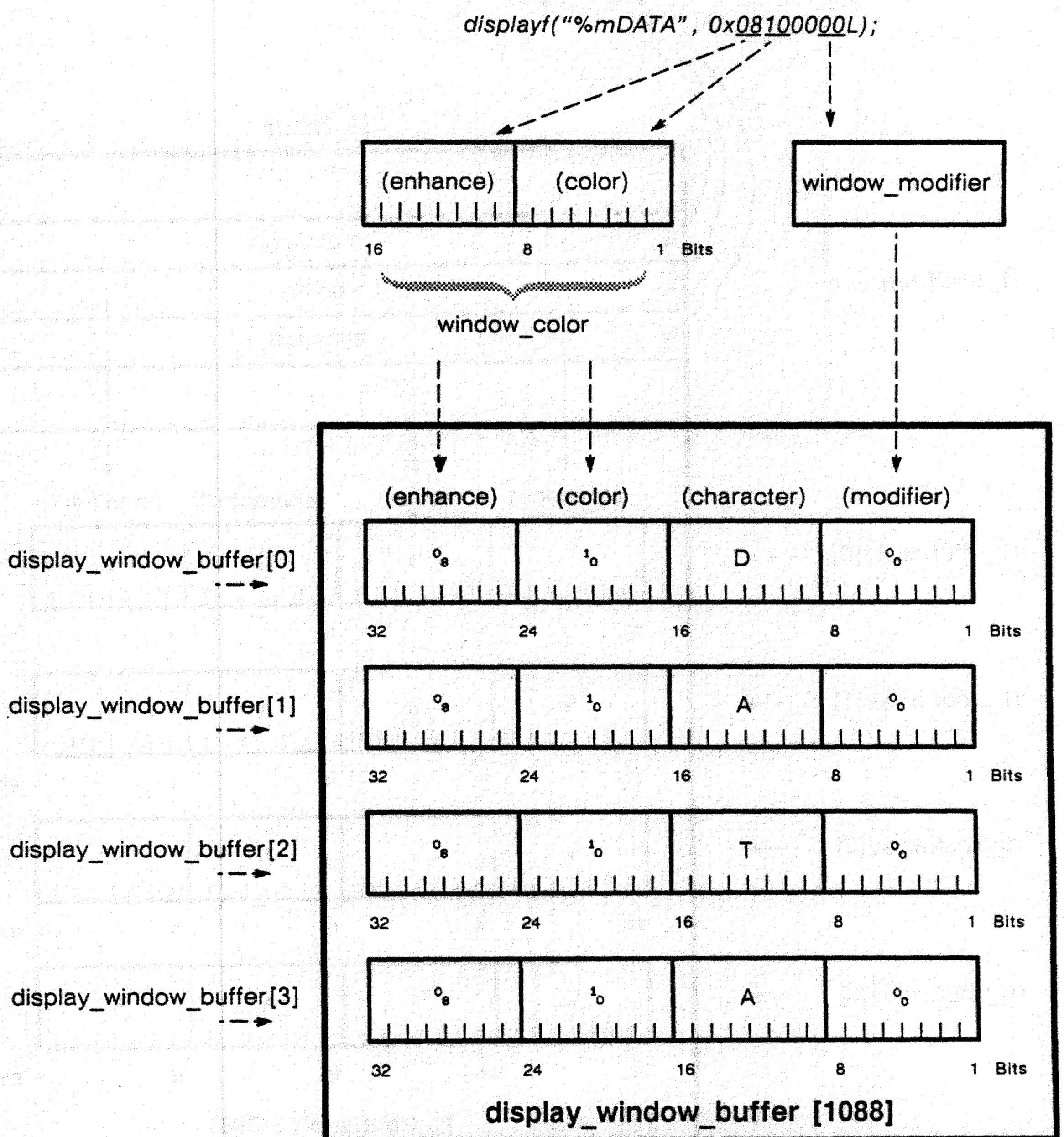


Figure 61-1 When a *displayf* routine is called, the attributes assigned via the *%m* conversion specifier are stored in two *extern* variables, accessible to the user. Both color and enhance attributes are contained in *window_color*. The low byte in *window_color* indicates the color; the high byte contains enhancements. In this example, the following attributes will be assigned to characters written to the Display Window: reverse-image enhancement, green-on-black color, and ASCII font. Before a character is written to the Display Window, it is combined in a *long* with its attributes, as mapped in the figure.

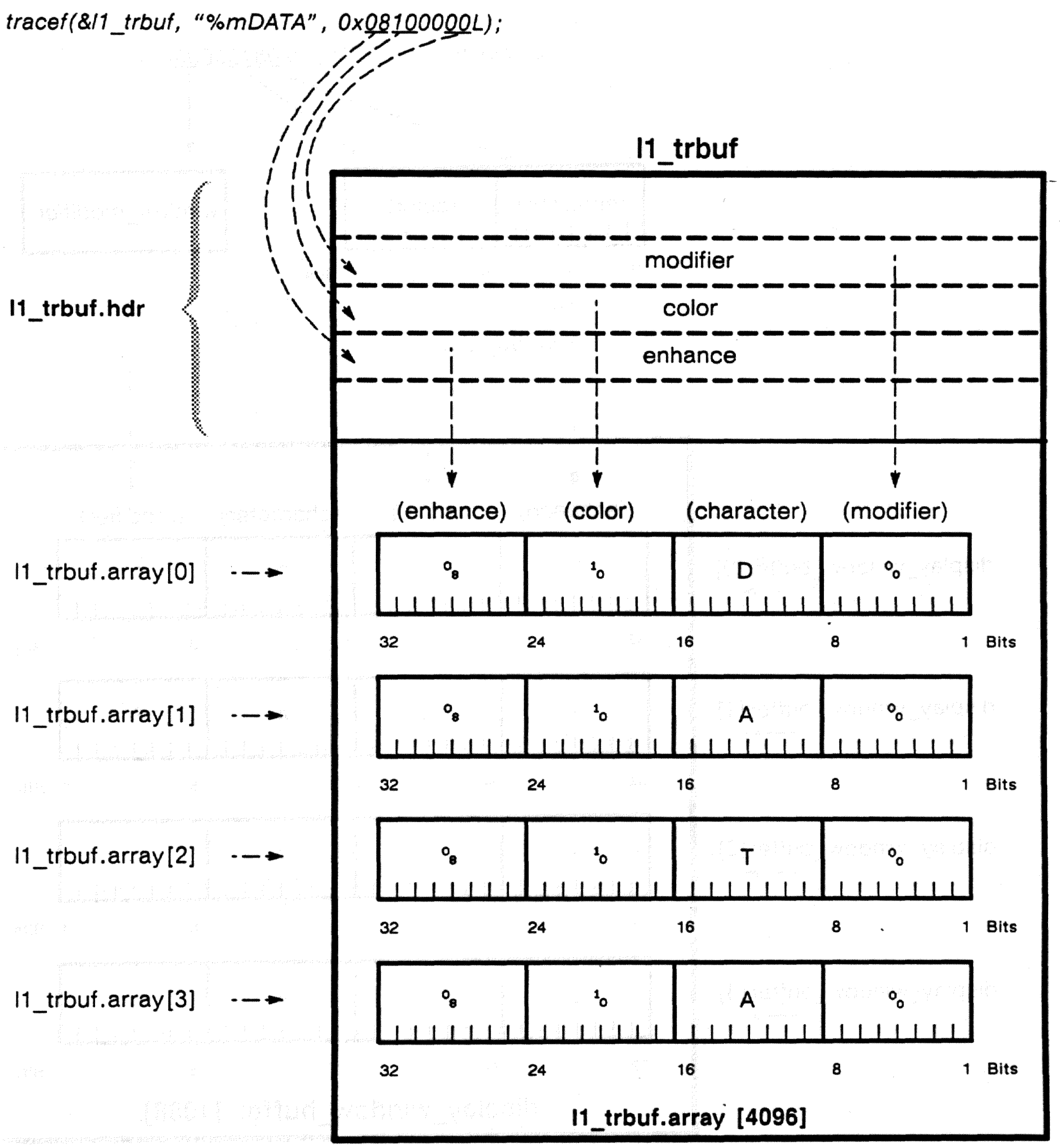


Figure 61-2 When a *tracef* routine is called, the attributes assigned via the *%m* conversion specifier are stored in three variables in the trace-buffer header of a designated buffer. In this example, *l1_trbuf.hdr* holds the following attributes: reverse-image enhancement, green-on-black color, and ASCII font. Before a character is written to the buffer, it is combined in a *long* with its attributes, as mapped in the figure.

The same attribute could be applied to a string in any of the user-trace buffers, as follows:

```
{
 #include <trace_buf.h>
 extern struct trace_buf l1_trbuf;
}
     STATE: apply_attribute_to_header
        CONDITIONS: ENTER_STATE
        ACTIONS:
        {
         tracef (&l1_trbuf, "%mThis data is underlined.", 0x04000000L);
        }
```

Only the header for the TRACE 1 display is affected by this *%m* conversion. Only the TRACE 1 buffer is written to. When other trace buffers are subsequently written to, the strings will not receive underlining as a result of the attributes applied above to the TRACE 1 header.

(B) Applying Attributes to Buffered Data

The Display Window is an array of 1,088 *long int*egers, each including one byte of character data and three bytes of attributes. The character data is generated by strings in display routines. The attributes for each character are derived from the current *window_color* and *window_modifier* values at the time the character is written to the display-window buffer.

Once the data word is written to the buffer as an element in the array, it can be accessed and written to—and therefore changed—the same as any other location in memory. In the example that follows, a string is written to the Display Window without underlining. Then, as a result of a keyboard input from the operator, the first 32-bit word in the string (containing the first character, the letter "T") is given a new value that includes the underline attribute.

```
{
 extern unsigned long display_window_buffer[1088];
 extern struct
        {
         unsigned char mpm;
         unsigned char cpm;
        }
 display_window_index_buffer[17];
}
     STATE: apply_attribute_directly_to_display_window
        CONDITIONS: ENTER_STATE
        ACTIONS:
        {
         pos_cursor(1,0);
         displayf ("This data is not underlined.");
        }
```

```
CONDITIONS: KEYBOARD " "
ACTIONS:
{
 display_window_buffer[64] = ((display_window_buffer[64] & ~0x04000000L) |
   0x04000000L);
 display_window_index_buffer[1].mpm ++;
}
```

Incrementing *display_window_index_buffer.mpm* is necessary to alert the processor on the CPM card (containing the display–controller software) that the program has changed the contents of the Display Window. Refer to Table 61-3 for an explanation of this structure.

The bitwise *and*ing and *or*ing in the example are necessary if you want to change certain bits in the word without affecting others. Note that the value whose complement (~) is *and*ed with *display_window_buffer* element #64 is the "mask" for the underline attribute in Table 61-4; and the value to the right of the *or* operator (|) is the "input" value for the underline attribute.

Table 61-7
Conversion Specifiers

Specifier	Argument type	Conversion Type
%b	integer-array pointer	array of *long* integers. 2nd byte of each *long* is displayed as character. 1st, 3rd, and 4th bytes interpreted as attributes. Array begins at pointer, ends at element containing null character and Special Character bit = 1.
%i	integer	signed decimal representing 15-bit value
%c	unsigned character	unsigned character
%#c	unsigned character	newline character displayed as $^{L}{}_{F}$ rather than acted on
%d	integer	signed decimal representing 15-bit value
%ld	integer	signed decimal representing 31-bit value
%H	character-array pointer	character array indicated by argument appears as small hex characters. (Precision as to number of characters becomes length of the array, overriding usual null-termination of strings.)
%m	integer	*long* integer not displayed or printed, but written to attribute header-variable for Display Window or for one of the trace buffers
%o	integer	unsigned octal representing 16-bit value
%lo	integer	unsigned octal representing 32-bit value
%#o	integer	unsigned octal representing 16-bit value, preceded by 0
%#lo	integer	unsigned octal representing 32-bit value, preceded by 0
%p	integer	unsigned hexadecimal (lower-case letters) representing 32-bit value, with a minimum 5 digits displayed and a colon between the 4 right-hand digits and the 1-4 left-hand digits. Useful for displaying CPU segment number and offset.
%s	character-array pointer	array of characters beginning at pointer and ending at null terminator or at array-length precision, whichever occurs first
%#s	character-array pointer	newline character displayed as $^{L}{}_{F}$ rather than acted on
%u	integer	unsigned decimal representing 16-bit value
%lu	integer	unsigned decimal representing 32-bit value
%#u	integer	hex *characters* (example: $^{B}{}_{F}{}^{E}{}_{5}$) representing 16-bit value
%#lu	integer	hex *characters* (example: $^{B}{}_{F}{}^{E}{}_{5}{}^{3}{}_{0}{}^{1}{}_{3}$) representing 32-bit value

Table 61-7 (continued)

Specifier	Argument type	Conversion Type
%x	integer	unsigned hexadecimal (lower-case letters) representing 16-bit value
%lx	integer	unsigned hexadecimal (lower-case letters) representing 32-bit value
%#x	integer	unsigned hexadecimal (lower-case letters) representing 16-bit value, preceded by 0x
%#lx	integer	unsigned hexadecimal (lower-case letters) representing 32-bit value, preceded by 0x
%X	integer	unsigned hexadecimal (upper-case letters) representing 16-bit value
%lX	integer	unsigned hexadecimal (upper-case letters) representing 32-bit value
%#X	integer	unsigned hexadecimal (upper-case letters) representing 16-bit value, preceded by 0x
%#lX	integer	unsigned hexadecimal (upper-case letters) representing 32-bit value, preceded by 0x
%\n	none	displays an ↵
%%	none	displays a %

61.6 Protocol Trace Buffers

There are two Protocol Trace buffers, one dedicated to Layer 2 and the other to Layer 3 data. Run-mode softkeys for accessing these traces—PROTOCL, L2TRACE, and L3TRACE—appear when personality packages are loaded in at Layers 2 and 3. The prompt line is not part of these buffers.

The size of each Protocol Trace buffer is 65,536 bytes. Of this total, two bytes are dedicated to the buffer header and two bytes to the trailer. The usable size of a Protocol Trace buffer, therefore, is 65,532 bytes. When a buffer's limit is reached, new characters written to the end of the buffer force out the same number of characters from the beginning of the buffer. In Freeze mode you may scroll through the buffer using the cursor keys.

You cannot write directly to the Protocol Trace buffers. Monitor the position within the buffers, as well as the wrap count, using the variables and structures discussed below.

(A) Variables

The addresses of the variables in Table 61-8 identify the physical location of the beginning and end of each Protocol Trace buffer. The beginning position is at the first data byte in the buffer. The end is just after the last data byte.

Table 61-8
Protocol Trace Buffer Variables

Type	Variable	Value (hex/decimal)	Meaning
extern unsigned char	l2pp_trbuff		First data byte in the Layer 2 Protocol Trace buffer. Address of this variable—segment number plus offset—will indicate the *physical* location of the first data byte, two bytes from the beginning of the buffer. Line Setup configured for emulate or monitor mode.
extern unsigned long	l2pp_trbuff_end		First byte in the two-byte trailer of the Layer 2 Protocol Trace buffer—i.e., after the last data byte. Address of this variable—segment number plus offset—will indicate the *physical* location of the end of the data area, hexadecimal FFFE bytes from the beginning of the buffer. Line Setup configured for emulate or monitor mode.
extern unsigned char	l3pp_trbuff		First data byte in the Layer 3 Protocol Trace buffer. Address of this variable—segment number plus offset—will indicate the *physical* location of the first data byte, two bytes from the beginning of the buffer. Line Setup configured for emulate or monitor mode.
extern unsigned long	l3pp_trbuff_end		First byte in the two-byte trailer of the Layer 3 Protocol Trace buffer—i.e., after the last data byte. Address of this variable—segment number plus offset—will indicate the *physical* location of the end of the data area, hexadecimal FFFE bytes from the beginning of the buffer. Line Setup configured for emulate or monitor mode.

(B) Structures

The structure variables in Table 61-9 contain the high and low bytes of a beginning and ending offset and wrap-count in the Layer 2 and Layer 3 Protocol Trace buffers. Create a logical beginning (or ending) offset within a buffer by combining the two offset-variables relating to a beginning (or ending) position into a single, two-byte offset. Add the resulting offset to the address of *l3_trbuff* to identify the *physical* address of a *logical* location.

The example below uses *#define* preprocessor directives for determining beginning and ending offsets in the Layer 3 Protocol Trace buffer. When *get_l3pp_value_end* is encountered in a program, for example, each of the two "end" offset-variables is cast into a *long* and, if necessary, shifted left to its appropriate position in an offset. Then the two variables are added together.

```
#define get_l3pp_value_begin
(((unsigned long)(l3pp_trbuff_ctl.begin_off_hi) << 8) +
((unsigned long)(l3pp_trbuff_ctl.begin _off_lo)))

#define get_l3pp_value_end
(((unsigned long)(l3pp_trbuff_ctl.end_off_hi) << 8) +
((unsigned long)(l3pp_trbuff_ctl.end_off_lo)))
```

When the ending offset, in this example, is added to the address of *l3_trbuff*, the result is the address of the *logical* end in the buffer:

```
unsigned long end_address;
end_address = &l3_trbuff + get_l3pp_value_end;
```

You may also use the offsets and wrap counts to determine how much data is currently in the buffer. Include the wrap count in the high two bytes of a four-byte offset. Then subtract the beginning offset from the ending offset.

```
#define get_l3pp_value_begin
(((unsigned long)(l3pp_trbuff_ctl.begin_wrap_hi) << 24) +
((unsigned long)(l3pp_trbuff_ctl.begin_wrap_lo) << 16) +
((unsigned long)(l3pp_trbuff_ctl.begin_off_hi) << 8) +
((unsigned long)(l3pp_trbuff_ctl.begin_off_lo)))

#define get_l3pp_value_end
(((unsigned long)(l3pp_trbuff_ctl.end_wrap_hi) << 24) +
((unsigned long)(l3pp_trbuff_ctl.end_wrap_lo) << 16) +
((unsigned long)(l3pp_trbuff_ctl.end_off_hi) << 8) +
((unsigned long)(l3pp_trbuff_ctl.end_off_lo)))

unsigned long end, begin, count;
end = get_l3pp_value_end;
begin = get_l3pp_value_begin;
count = end - begin;
```

Table 61-9
Protocol Trace Buffer Structures

Type	Variable	Value (hex/decimal)	Meaning
Structure Name:	**lpp_trbuff_ctl**		Declared as type *struct*. The variables contained in this structure monitor logical location in a Protocol Trace buffer. Reference structure variables as follows: *lpp_trbuff_ctl.begin_off_hi*.
unsigned char	begin_off_hi	*0-ff/0-255*	High byte of a 2-byte offset from the *physical* beginning of the Protocol Trace buffer to a *logical* beginning in the buffer. Range of the two-byte offset is 2 through hexadecimal FFFE.
unsigned char	begin_off_lo	*0-ff/0-255*	Low byte of a 2-byte offset from the *physical* beginning of the Protocol Trace buffer to a *logical* beginning in the buffer. Range of the two-byte offset is 2 through hexadecimal FFFE.
unsigned char	begin_wrap_hi	*0-ff/0-255*	High byte of a 2-byte count of the number of times a *logical* beginning has wrapped through the Protocol Trace buffer.
unsigned char	begin_wrap_lo	*0-ff/0-255*	Low byte of a 2-byte count of the number of times a *logical* beginning has wrapped through the Protocol Trace buffer. It will have a value of zero only once. Once the count reaches hexadecimal FFFF, it will wrap to one.
unsigned char	end_off_hi	*0-ff/0-255*	High byte of a 2-byte offset from the *physical* beginning of the Protocol Trace buffer to a *logical* end in the buffer. Range of the two-byte offset is 2 through hexadecimal FFFE.
unsigned char	end_off_lo	*0-ff/0-255*	Low byte of a 2-byte offset from the *physical* beginning of the Protocol Trace buffer to a *logical* end in the buffer. Range of the two-byte offset is 2 through hexadecimal FFFE.
unsigned char	end_wrap_hi	*0-ff/0-255*	High byte of a 2-byte count of the number of times a *logical* end has wrapped through the Protocol Trace buffer.
unsigned char	end_wrap_lo	*0-ff/0-255*	Low byte of a 2-byte count of the number of times a *logical* end has wrapped through the Protocol Trace buffer. It will have a value of zero only once. Once the count reaches hexadecimal FFFF, it will wrap to one.
Structure Name:	**l2pp_trbuff_ctl**		An instance of the *lpp_trbuff_ctl* structure, declared as type *extern struct lpp_trbuff_ctl*. The variables contained in this structure monitor logical location in the Layer 2 Protocol Trace buffer. Has the same structure as *lpp_trbuff_ctl*. Reference structure variables as follows: *l2pp_trbuff_ctl.begin_off_h*.

Table 61-9 (continued)

Type	Variable	Value (hex/decimal)	Meaning
Structure Name:	**l3pp_trbuff_ctl**		An instance of the *lpp_trbuff_ctl* structure, declared as type *extern struct lpp_trbuff_ctl*. The variables contained in this structure monitor logical location in the Layer 3 Protocol Trace buffer. Has the same structure as *lpp_trbuff_ctl*. Reference structure variables as follows: *l3pp_trbuff_ctl.begin_off_h*.

(C) Routines

There are no routines associated exclusively with Protocol Traces.

62 Counters, Timers, and Accumulators

62.1 Counters

The translator declares the following structure for counters that are entered as softkey tokens on the Protocol Spreadsheet:

```
struct counter_struct
{
  unsigned long current;
  unsigned long last;
  unsigned long maximum;
  unsigned long minimum;
  unsigned short sample_count;
  unsigned long total_high;
  unsigned short total_low_low;
  unsigned short total_low_high;
  unsigned short out_of_range;
  unsigned short changed;
  unsigned long prev;
  unsigned long old;
};
struct counter_struct counter_name={0,0,0,-0ul};
```

The first eight counter variables in the structure are used to calculate statistical values whenever the counter is sampled. See Table 62-1. Three of the variables—*counter_name.current*, *counter_name.prev*, and *counter_name.old*—come into play each time the counter is incremented, decremented, or set to a particular value.

Counters are internal program variables and counter interrupts are strictly program-generated signals, so the C programmer is free to ignore this structure and maintain counts and statistics in a different way. Please note, however, that the 68010 CPU expects this counter structure when it polls the 80286 periodically for statistical values to display in columns on the tabular and graphic stats screens.

(A) Current, Previous, and Old Values

When a counter is incremented, decremented, or set to a specific value on the Protocol Spreadsheet, the program does not signal a *counter_name_change* interrupt automatically. First it verifies that the new value of the counter really is a *change* from the previous value. See Table 62-2. For this comparison, the program needs to maintain two variables, *counter_name.current* and *counter_name.prev*.

Table 62-1
Counter Structures

Type	Variable	Meaning
Structure Name:	**counter_struct**	Structure of a counter. Declared as type *struct*. Declared automatically if a program counter is used. Program counters assigned to structure as follows: struct counter_struct counter_*name*. Reference a structure variable as follows: counter_*name*.current.
unsigned long	current	This value of the counter is acted on directly by program actions.
unsigned long	last	Last sampled value; displayed on the tabular statistics screen.
unsigned long	maximum	Maximum value of all samples; displayed on the tabular statistics screen.
unsigned long	minimum	Minimum value of all samples; displayed on the tabular statistics screen. Should be initialized as ~0ul.
unsigned short	sample_count	Number of samples.
unsigned long	total_high	High four bytes of an eight-byte counter total.
unsigned short	total_low_low	Low two bytes of an eight-byte counter total. This two-byte variable counts to 65,535.
unsigned short	total_low_high	Bytes 3 and 4 of an eight-byte counter total.
unsigned short	out_of_range	Number is out of range, either incremented beyond the range or decremented below 0; should not be factored into averages.
unsigned short	changed	For future use.
unsigned long	prev	When converting a counter action to C, the translator compares *prev* with *current* to determine whether counter has changed. Then *prev* is updated to *current* and *counter_*name*_change* is signaled.
unsigned long	old	When converting a counter condition to C, the translator compares *old* with *current* to determine whether true condition is new (transitional). After program logic has examined counter, *old* is updated to *prev*.

Here, for example, is the C translation of the simple action COUNTER example SET 5.

```
counter_example.current = 5;
if (counter_example.prev != counter_example.current)
  {
    counter_example.old = counter_example.prev;
    counter_example.prev = counter_example.current;
    signal (counter_example_change);
  }
```

Table 62-2
Counter Variables

Type	Variable	Meaning
extern event	counter_*name*_change	True when the named counter is incremented, decremented, or set to new value. This event will not be triggered unless a spreadsheet *condition* names the counter. Line Setup configured for emulate or monitor mode.

It is clear from the translation that the variable *counter_example.prev* is used to limit the number of *counter_example_change* interrupts to those cases where the current value of the counter really has changed.

What is *counter_name.old* used for? We will preface the answer by citing the behavior of the counter in the following spreadsheet example.

```
STATE: threshold_condition
   CONDITIONS: KEYBOARD " "
   ACTIONS: COUNTER spacebar INC
   CONDITIONS: COUNTER spacebar GE 7
   ACTIONS: ALARM
```

Each time you press the space bar while this program is running, the counter will increment, but no matter how many times you press the space bar the alarm will only sound once. It will sound on the seventh keystroke, the *first time* the counter is greater than or equal to 7. If the program had a decrement or set action that lowered the counter to less than 7, the alarm would sound again when the counter reached the 7 threshold.

The translator accomplishes this threshold condition by coding the *waitfor* clause as follows:

```
counter_spacebar_change && (! (counter_spacebar.old >= 7)) && (counter_spacebar.current >= 7);
```

Since *counter_spacebar.prev* was used (and then updated to "current") in the *if* statement that sent the *counter_spacebar_change* interrupt, the "old" value is required in the *waitfor* condition to insure a "transitional" or "threshold" counter condition.

(B) Sampling a Counter

Here is the translator's version of a counter sample action:

```
counter_name.last = counter_name.current;
if (counter_name.current > counter_name.maximum)
  {
    counter_name.maximum = counter_name.current;
  }
if (counter_name.current < counter_name.minimum)
  {
    counter_name.minimum = counter_name.current;
  }
counter_name.sample_count++;
{
  unsigned long temp;
  temp = (counter_name.current & 0x0000ffff) + counter_name.total_low_low;
  counter_name.total_low_low = temp;
  temp = (counter_name.current >> 16) + counter_name.total_low_high + (temp >> 16);
  counter_name.total_low_high = temp;
  counter_name.total_high += temp >> 16;
}
counter_name.current = 0;
```

In order to establish an average value for all samples, a grand total for current values at the time of each sampling must be maintained. Since an ordinary INTERVIEW current counter is 32 bits, the counter that maintains the grand total of current counts must be larger (64 bits). There is no data type this large in C, and so the "total" counter is distributed among three variables and the somewhat complicated coding involving the *temp* variable is required to add the current counter to this composite counter.

(C) Updating the Statistics Screen

The CPM polls the MPM continuously to see if data is available to be output to the printer or the plasma display. This data includes character data, trace data, prompts, and values to be posted to the statistics screens.

In order to know where on the statistics screens the values for the particular counters (and timers and accumulators) should be placed, the 68010 CPU on the CPM needs some help from the program (that is, from the MPM). This help is in the form of a "stat message" that the translator (or the programmer) codes once at the beginning of the program. The stat message is a structure that the MPM sends to the CPM. See Table 62-3. The stat message says, in effect, "Here is the address of a counter structure. When you access this structure during the running of the program in order to pull out the current, last, maximum, minimum, total, and sample-count values, display those values on the row of the tabular stats screen where the user has typed *spacebar*" (for example).

Table 62-3
Counter, Timer, and Accumulator Structures

Type	Variable	Value (hex/decimal)	Meaning
Structure Name: stat_msg			Structure of a stat message. A stat message is sent once for each named counter, timer, or accumulator. Declared as type struct. Declared automatically if a softkey-entered COUNTER is used as a condition, or if softkey-entered COUNTER, TIMER, or ACCUMUL action is taken. Program stat messages assigned to structure as follows: struct stat_msg *name*. You must assign values to the elements of the structure. Reference a structure variable as follows: *name*.type.
unsigned short	op_type	0a00/2560	Register statistics objects from the MPM to the CPM. Other values and meanings for future use.
unsigned short	type	0 0100/256 0200/512	accumulator counter timer
unsigned long	object_name		The MPM (80286) address of a counter, timer, or accumulator name, converted to CPM (68010) format. To get an object_name address, enter: *name*.object_name = get_68k_phys_addr("*name_of_counter*");
unsigned long	object_address		The MPM (80286) address of a counter, timer, or accumulator structure, converted to CPM (68010) format. To get a structure address for a counter, enter: *name*.object_address = get_68k_phys_addr(&counter_*name_of_counter*);

Here is a C program that causes the current value of a counter named "key" to increment on the tabular-statistics screen each time an ASCII-keyboard key is struck.

```
{
 struct
   {
     unsigned short op_type;
     unsigned short type;
     unsigned long object_name;
     unsigned long object_address;
   } stat_msg;
 extern unsigned long get_68k_phys_addr();
```

```
  struct counter_struct
    {
      unsigned long current;
      unsigned long last;
      unsigned long maximum;
      unsigned long minimum;
      unsigned short sample_count;
      unsigned long total_high;
      unsigned short total_low_low;
      unsigned short total_low_high;
      unsigned short out_of_range;
      unsigned short changed;
      unsigned long prev;
      unsigned long old;
    };
  struct counter_structure counter_key;
  extern fast_event keyboard_new_key;
}
  STATE: update_stat_screen
      {
        stat_msg.op_type = 2560;
        stat_msg.type = 256;
        stat_msg.object_name = get_68k_phys_addr("key");
        stat_msg.object_address = get_68k_phys_addr(&counter_key);
        send_stat_message(&stat_msg);
        waitfor
          {
            keyboard_new_key:
              {
                counter_key.current++;
              }
          }
      }
```

The variable *stat_msg.object_name* is a pointer to the name of the counter that the user has entered on the protocol spreadsheet. The program gives this name to the CPM, and expects the CPM to locate the name among the names that the user has entered on the tabular or graphic statistics menu. The delivery to the CPM of a pointer to the stats–menu name and a pointer to the counter structure is the purpose of the stat message. The message allows the CPM to correlate a line on the statistics results screen with an actual program counter (or timer or accumulator).

NOTE TO C PROGRAMMERS: When the translator creates a counter variable it adds the prefix *counter_* to the spreadsheet name, but the programmer who is working primarily in C and is not making use of spreadsheet counters can name the counter any way he wishes, with or without the prefix. Similarly, the string that is communicated to the CPM in *stat_msg.object_name* ("key" in the example above) must agree with the name on the stats menu, but it need not bear any resemblance to the name of the counter structure.

NOTE ALSO: In most of the examples in this manual, we have not bothered to declare routines since it is not necessary. In the absence of a declaration, the compiler assumes that the routine is external and that it returns an integer. In nearly all cases, this assumption works. *get_68k_phys_addr()* returns a *long*, however, and must be declared.

62.2 Timers

The translator declares the following structure for timers that are entered as softkey tokens on the Protocol Spreadsheet:

```
struct timer_struct
{
  unsigned long current;
  unsigned long last;
  unsigned long maximum;
  unsigned long minimum;
  unsigned short sample_count;
  unsigned long total_high;
  unsigned short total_low_low;
  unsigned short total_low_high;
  unsigned long start_tick_value;
  unsigned short running;
  unsigned short changed;
};
```

There are no timer conditions in the software (since timeouts provide the time-triggering function), and therefore all of the variables in the structure serve as data for the CPM when it updates the stats screens. See Table 62-4. A stat message must be sent so the CPM can correlate a line on the statistics results screen with the correct program timer. The stat message is documented in the previous section on counters. The timer stat message is different only in respect that the *stat_msg.type* element should be set to 512 instead of 256.

Timer restart, continue, and stop actions are explained in this section. The clear action is simply a matter of changing the elements in the structure to zero (except for *timer_name.minimum*, which becomes the one's complement of zero).

(A) Time Ticks

Time ticks are timed increments of either of two hardware counters in the INTERVIEW. The programmer can select which of the two timing mechanisms to use for a given timer.

One tick-counter is on the FEB card and is used to time-stamp incoming data and EIA leads. The intervals between ticks is determined on the FEB Setup menu. Ticks can be enabled/disabled on the same menu. The current value of this counter is available in a variable called *ll_tick_count*. See Table 62-5. The current value always reflects the number of ticks since the program entered Run mode. The number of ticks may or may not equate to the amount of time in Run mode, since ticks are also encoded in playback data and the playback rate is subject to "local conditions" such as playback speed and idle suppression.

FEB time ticks are the most precise timing mechanism in that they have a resolution to 10 microseconds. They also represent the most durable method of timekeeping, since they preserve the original data timings even during playback.

Table 62-4
Timer Structures

Type	Variable	Value (hex/decimal)	Meaning
Structure Name:	**timer_struct**		Structure of a timer. Declared as type *struct*. Declared automatically if a program timer is used. Program timers assigned to structure as follows: struct timer_struct timer_*name*. Reference a structure variable as follows: timer_*name*.current.
unsigned long	current		Current value of timer, not updated while timer is running. Values are in microseconds rounded to tick-unit on FEB Setup screen.
unsigned long	last		Value of last sample; displayed on the tabular statistics screen.
unsigned long	maximum		Maximum value of all samples; displayed on the tabular statistics screen.
unsigned long	minimum		Minimum value of all samples; displayed on the tabular statistics screen. Should be initialized as ~0ul.
unsigned short	sample_count		Number of samples.
unsigned long	total_high		High four bytes of an eight-byte timer total.
unsigned short	total_low_low		Low two bytes of an eight-byte timer total.
unsigned short	total_low_high		Bytes 3 and 4 of an eight-byte timer total.
unsigned long	start_tick_value		Tick-count in microseconds when timer was started, restarted, or continued. For line-related conditions at Layer 1, this value is stored in *l1_tick_count*; for non-line conditions, use *get_wall_time_286_ticks* routine.
unsigned short	running	0	Stopped. This variable is polled and a zero stops the timer from incrementing and sets the current value to *timer_name.current* (understood as microseconds).
		~0	Running. All 1's in this variable causes the timer to increment, showing a value that equals (wall-time ticks - *timer_name.start_tick_value*) + *timer_name.current*.
unsigned short	changed	~0	For future use.

Table 62-5
Timer Variables

Type	Variable	Meaning
extern unsigned long	l1_tick_count	This variable counts ticks from the start of Run mode. Tick=sec, msec, etc., depending on FEB setup. Subtract early value from later value to create a timer. ACTIONS: { *displayf* ("%ld msecs ", (*l1_tick_count* – *timer_name.start_tick_value*));} Add to *start_of_run_time* to determine more precise current time for time-stamping events. Line Setup configured for emulate or monitor mode.
extern unsigned long	start_of_run_date	Date when Run mode entered. Byte 1 (low byte) indicates day; byte 2 stores month; and bytes 3 and 4 indicates year. May be used to time-stamp events. See also *start_of_run_time*. Line Setup configured for emulate or monitor mode.
extern unsigned long	start_of_run_time	Time when Run mode entered. Byte 1 (low byte) indicates seconds; byte 2 stores minutes; and byte 3 indicates hours. May be used to time-stamp events. See also *start_of_run_date* and *l1_tick_count*. † Line Setup configured for emulate or monitor mode.

† In the example below, the *displayf* (or *tracef*) routine uses timer variables to time-stamp good BCCs on the DCE side. (Similar programming could determine the current date.) The tick unit selected on the FEB Setup menu is seconds. Adjust the program as needed for other tick units.

```
{
 extern unsigned long start_of_run_date, start_of_run_time, l1_tick_count;
 unsigned short seconds, hours, minutes, tick_mins, tick_secs, tick_hours;
 #define SECS(run_time) (unsigned short)(run_time & 0xff)
 #define MINS(run_time) ((unsigned short)(run_time >> 8) & 0xff)
}
      STATE: time
         CONDITIONS: DCE GOOD_BCC
         ACTIONS:
         {
          tick_secs = l1_tick_count % 60;
          tick_mins = (l1_tick_count + SECS(start_of_run_time)) / 60;
          tick_hours = (tick_mins + MINS(start_of_run_time)) / 60;
          displayf("Time: %.2d:%.2d:%.2d\n",
                 (unsigned short)(((start_of_run_time >> 16) & 0xff) + tick_hours)%24,
                 (MINS(start_of_run_time) + tick_mins)%60,
                 (SECS(start_of_run_time) + tick_secs)%60);
         }
```

The other tick–counter is on the MPM and is referred to as the wall–time clock. This clock ticks once per millisecond and drives the timers displayed on the statistics results screens—at least while they are incrementing. At the moment a timer stops incrementing, the programmer can reach in and replace the incremented value with a timer value based the FEB tick–counter instead.

The current value of this wall–time tick–counter is available to the program via the *get_wall_time_286_ticks* routine. The current value always reflects both the number of ticks and the actual elapsed time ("wall time") since the program entered Run mode.

(B) Running

While it increments on the stats screen, a timer always is driven by wall–time ticks. To start a current timer incrementing, first you must have sent a stat message to correlate the timer structure with a timer line on the stats screen. At that point the simple statement *timer_name.running = -0* will start the timer. The value of the timer at any given time while it is running will be the MPM (wall–time) ticks minus the *timer_name.start_tick_value* plus any *timer_name.current* value.

To stop a timer, change *timer_name.running* to zero. The current column of the timer will immediately display the value of *timer_name.current* (zero, unless you have done something in your program to calculate the current value of the timer). The stats display will interpret *timer_name.current* as a value in microseconds and convert it to the unit selected for that timer line.

(C) Restart

The translator has two different versions of the timer restart action, depending on what condition precipitated the action. The first version is used if the condition was data–related (or EIA–related) *and* time ticks are enabled on the FEB Setup menu. Here is this data–timer version:

```
unsigned long temp;
convert_tick_count (l1_tick_count, &temp);
timer_name.current = 0;
timer_name.start_tick_value = temp;
timer_name.running = -0;
```

The *convert_tick_count* routine converts *l1_tick_count* into microseconds and stores the result in *temp*. The value of *temp* is assigned immediately to *timer_name.start_tick_value*. When the 68010 sees that *timer_name.running* equals the one's complement of zero, it subtracts the start–tick value from the l1–tick count and displays the difference in the current column of the timer line. Since the start–tick value was derived a moment before from the l1–tick count, the difference will be zero. The current column on the stats screen should begin a timer at zero following a restart.

A slightly different version of the program is used if the condition was nondata-related or if time ticks are disabled in the FEB. The *convert_tick_count* routine is not used and the following routine is used in its place:

```
get_wall_time_286_ticks (&temp);
```

This routine returns the current value of the wall-time tick-counter, in milliseconds zero-padded to microseconds. It stores the value in *temp* and the program proceeds as above.

(D) Continue

The timer-continue action is very similar to the restart. There are just two differences. One, the action is enclosed in an if statement that verifies that *timer_name.running* equals zero—that the timer actually is stopped, in other words; and two, *timer_name.current* is not set to zero, but retains the value it received the last time the timer stopped.

(E) Stop

Here is one of the two versions of a timer stop action:

```
if (timer_name.running != 0)
  {
    unsigned long temp;
    convert_tick_count (l1_tick_count, &temp);
    timer_name.current += temp - timer_name.start_tick_value;
    timer_name.running = 0;
  }
```

In this translation, the start-tick value is subtracted from the current tick count, and any pending current value (held over if the timer was continued) is added in. The result is a new *timer_name.current* value. This value is posted to the stats screen as soon as the 68010 sees *timer_name.running = 0*.

The other version of the stop action uses *get_wall_time_286_ticks* instead of *convert_tick_count*.

(F) Sample Action

The code that produces the sample action is identical to the code that sampled a counter. See Section 62.1(B). The *timer_name.sample_count* variable's not equaling zero causes minimum, maximum, and average values to be displayed.

62.3 Accumulators

Shown below is the structure of an accumulator as the translator declares it (and as the 68010 accesses it to update the statistics screens). Also refer to Table 62-6. Note that there is no current value, since an accumulator neither counts nor times. There are no "previous" and "old" values, because in its spreadsheet implementation an accumulator never is tested in a Conditions block.

```
struct accumulator_struct
{
  unsigned long last;
  unsigned long maximum;
  unsigned long minimum;
  unsigned short sample_count;
  unsigned long total_high;
  unsigned short total_low_low;
  unsigned short total_low_high;
  unsigned short changed;
};
struct accumulator_struct accumulator_name={0,0,~0ul};
```

Here is the translator's version of an accumulate action when the object of the accumulation (selected by the user) was the maximum sampled value of a counter named *framechar*.

```
accumulator_name.last = accumulator_framechar.maximum;
if (accumulator_name.last > accumulator_name.maximum)
  {
    accumulator_name.maximum = accumulator_name.last;
  }
if (accumulator_name.last < accumulator_name.minimum)
  {
    accumulator_name.minimum = accumulator_name.last;
  }


accumulator_name.sample_count++;
  {
    unsigned long temp;
    temp = (accumulator_name.last & 0x0000ffff) + accumulator_name.total_low_low;
    accumulator_name.total_low_low = temp;
    temp = (accumulator_name.last >> 16) + accumulator_name.total_low_high + (temp >> 16);
    accumulator_name.total_low_high = temp;
    accumulator_name.total_high += temp >> 16;
  }
accumulator_name.changed = ~0;
```

A stat message must be sent so the CPM can correlate a line on the statistics results screen with the correct accumulator. The stat message is documented in the previous section on counters. The accumulator stat message is different only in respect that the *stat_msg.type* element should be set to 0 instead of 256.

The *accumulator_name.sample_count* variable's not equaling zero causes minimum, maximum, and average values to be displayed.

Table 62-6
Accumulator Structures

Type	Variable	Meaning
Structure Name:	**accumulator_struct**	Structure of an accumulator. Declared as type *struct*. Declared automatically by program when the user softkey-enters an ACCUMULATE action. Specific accumulator assigned to structure as follows: struct accumulator_struct accumulator_*name*. Reference a structure variable as follows: accumulator_*name*.last.
unsigned long	last	Value of last sample; displayed on the tabular statistics screen.
unsigned long	maximum	Maximum value of all samples; displayed on the tabular statistics screen.
unsigned long	minimum	Minimum value of all samples; displayed on the tabular statistics screen. Should be initialized as -0ul.
unsigned short	sample_count	Number of samples.
unsigned long	total_high	High four bytes of an eight-byte accumulator total.
unsigned short	total_low_low	Low two bytes of an eight-byte accumulator total.
unsigned short	total_low_high	Bytes 3 and 4 of an eight-byte accumulator total.
unsigned short	changed	For future use.

62.4 Routines

get_68k_phys_addr

Synopsis

```
extern unsigned long get_68k_phys_addr(variable_ptr);
unsigned char * variable_ptr;
```

Description

This routine converts the address of a specified variable in the 80286 processors (MPM boards) to 68010 (CPM) format. This routine must be declared.

Inputs

The only parameter is the address to be converted.

Returns

The *get_68k_phys_addr* routine returns the converted address.

Example

See *send_stat_message* routine.

send_stat_message

Synopsis

```
extern void send_stat_message(struct_stat_msg_ptr);
struct stat_msg
  {
   unsigned short op_type;
   unsigned short type;
   unsigned long object_name;
   unsigned long object_address;
  };
struct stat_msg * struct_stat_msg_ptr;
```

Description

The *send_stat_message* routine sends the stat message structure to the 68010 CPU (CPM board). The current use of this routine sends the addresses of program counters, timers, and accumulators in the 80286 processors (MPM boards) to the CPM board where the tabular and graphic statistics displays are located.

The routine is called only one time in a program for each named counter, timer, or accumulator. Entering COUNTER as a condition or action (or TIMER or ACCUMUL as actions) via softkey on the Protocol Spreadsheet automatically declares the counter named and sends the stat message.

Inputs

The only parameter is a pointer to the structure of the stat message. For an explanation of the elements of the stat message, see Table 62-3.

Example

You plan on incrementing a counter named "dte_info" when a DTE Info frame is detected.

```
{
  struct
   {
    unsigned short op_type;
    unsigned short type;
    unsigned long object_name;
    unsigned long object_address;
   } stat_msg;
```

```
 struct counter_structure
 {
  unsigned long current;
  unsigned long last;
  unsigned long maximum;
  unsigned long minimum;
  unsigned short sample_count;
  unsigned long total_high;
  unsigned short total_low_low;
  unsigned short total_low_high;
  unsigned short out_of_range;
  unsigned short changed;
  unsigned long prev;
  unsigned long old;
 };
 struct counter_structure counter_dte_info = {0, 0, 0, -0ul};
 extern unsigned long get_68k_phys_addr();
}
LAYER: 2
     STATE:send_stat_message
        CONDITIONS: ENTER_STATE
        ACTIONS:
        {
         stat_msg.op_type = 2560;
         stat_msg.type = 256;
         stat_msg.object_name = get_68k_phys_addr("dte_info");
         stat_msg.object_address = get_68k_phys_addr(&counter_dte_info);
         send_stat_message(&stat_msg);
        }
        NEXT_STATE: count_info
     STATE: count_info
        CONDITIONS: DTE INFO
        ACTIONS:
        {
         counter_dte_info.current++;
        }
```

get_wall_time_ticks

Synopsis

```
extern void get_wall_time_ticks(ticks_68k_format_ptr);
unsigned long * ticks_68k_format_ptr;
```

Description

The *get_wall_time_ticks* routine gets the number of wall-time ticks (in CPM storage format) from the time [RUN] was hit. The wall clock gives millisecond resolution rounded to microseconds.

Inputs

The only input is a pointer to the location where the returned time-tick value will be stored.

Example

```
{
 unsigned long ticks;
}
LAYER: 2
     STATE: get_ticks
        CONDITIONS: KEYBOARD " "
        ACTIONS:
        {
         get_wall_time_ticks(&ticks);
        }
```

get_wall_time_286_ticks

Synopsis

extern void get_wall_time_286_ticks(ticks_286_format_ptr);
*unsigned long * ticks_286_format_ptr;*

Description

The *get_wall_time_286_ticks* routine gets the number of wall–time ticks (in MPM storage format) from the time [RUN] was hit. The wall clock gives millisecond readings rounded to microseconds. Use this routine prior to setting the *start_tick_value* in a timer action when **Time Ticks:** OFF has been selected on the Front–End Buffer Setup screen. Also use this routine to derive the *start_tick_value* if the condition is not line-related, e.g., KEYBOARD, even when time ticks are enabled on the FEB Setup menu.

Inputs

The only input is a pointer to the location where the returned time–tick value will be stored.

Example

```
{
 unsigned long ticks_286;
}
LAYER: 3
     STATE: get_ticks
        CONDITIONS: KEYBOARD " "
        ACTIONS:
        {
         get_wall_time_286_ticks(&ticks_286);
         displayf ("%lu", ticks_286);
        }
```

convert_tick_count

Synopsis

```
extern void convert_tick_count(mpm_format_ticks, converted_ticks_ptr);
unsigned long mpm_format_ticks;
unsigned long * converted_ticks_ptr;
```

Description

The *convert_tick_count* routine converts a designated tick count into microseconds.

Use this routine to derive the *start_tick_value* for a timer action if ticks are enabled on the FEB Setup menu and the condition is line-related, e.g., RCV INFO.

Inputs

The first parameter is a designated tick count as long as it is in MPM storage format. It may be any of the layer tick counts. The unit of the *l1_tick_count* (and other layers' tick counts) value is determined on the Front End Buffer menu.

The second parameter is a pointer to the location where the returned tick count converted to microseconds will be stored.

Example

```
{
 extern unsigned long l1_tick_count;
 unsigned long converted_ticks;
}
LAYER: 1
     STATE: convert_ticks
        CONDITIONS: RECEIVE GOOD_BCC
        ACTIONS:
        {
         convert_tick_count(l1_tick_count, &converted_ticks);
         displayf ("%lu", converted_ticks);
        }
```

63 OSI

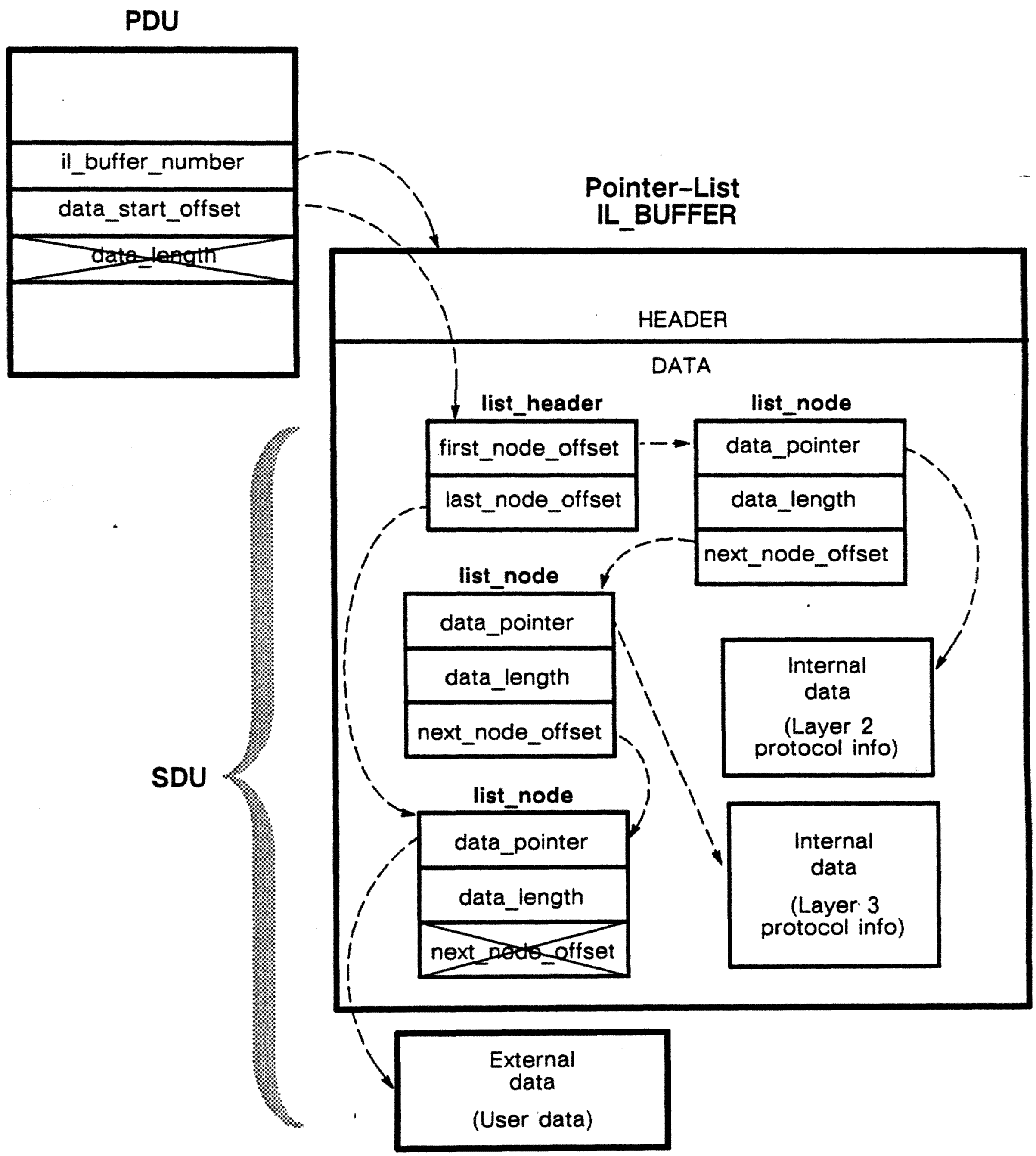


Figure 63-1 Primitive Data Unit and sample Pointer-List Buffer being passed *down* the layers.

63 OSI

The most convenient tools for handling protocol headers while data is moving down and up the layers in the INTERVIEW are the spreadsheet SEND and GIVE_DATA actions in the various protocol packages. For instances when a protocol package is not loaded, such as when you are developing a new protocol or simply using a protocol that is not yet an option on the Layer Setup screen, OSI structures, variables, and routines in C become essential tools also.

63.1 Structures

The programmer may access the information in primitive data units conveniently by using a C structure as a multibyte pointer that is superimposed on data in the PDU's. Before using a structure-pointer, it is necessary to understand the contents of IL buffers and primitive data units. All structures referenced may be found in Table 63-1.

(A) Interlayer Message Buffers

There are a maximum sixteen IL buffers in use at a given time. These buffers may be one of two kinds: data-character or pointer-list. In buffers being passed up the layers, data-character buffers (Figure 63-2) are always used. In buffers going down the layers, pointer-list buffers (Figure 63-1) are primarily used. The difference is that pointer-list buffers contain list-nodes which provide information about the location of data (or "lists") inserted or referenced in the buffer, while data-character buffers do not.

1. *Header.* Each IL buffer contains a header that stores useful information such as the status of the maintain bits that prevent the buffer from being returned to the general pool; the position of the buffered data in the INTERVIEW's display buffer; and the tick count (time) when the data was buffered from the line. (See *il_buffer* structure.)

2. *Service Data Unit.* The IL buffer also contains the data itself. This data component, the service data unit (or "SDU"), is added to as the buffer is passed down the layers, and subtracted from as a buffer travels up the layers. *A data-character IL buffer includes all the data that was present when the data was first buffered,* and the contents of this buffer do not change as the buffer is passed up the layers. What changes is the service data unit, derived from the data-start offset in the PDU.

The first part of the SDU in a pointer-list buffer is a *list-header node* (structure *il_list_header*) which contains information about the location of the first and last text nodes. As a buffer is passed down from Layer 3 to Layer 2 in X.25 (see Figure 63-1), a new text node containing a Layer 3 protocol header is inserted in buffer. Since the Layer 3 data will precede user data, the list node for the protocol information is referenced ahead of any other list nodes, changing the first-node reference in the list header. (If text is appended to the end of existing data, the list node referenced as last will change.)

The SDU in a pointer-list buffer also includes *list nodes* (structure *il_list_node*) which give a pointer to data, the length of the data pointed to, and the offset from the start of the buffer to the *next* list node.

Finally, the service data unit in all buffers includes *data*, whether copied into the buffer (usually protocol information) or located in memory outside of the buffer (usually user data).

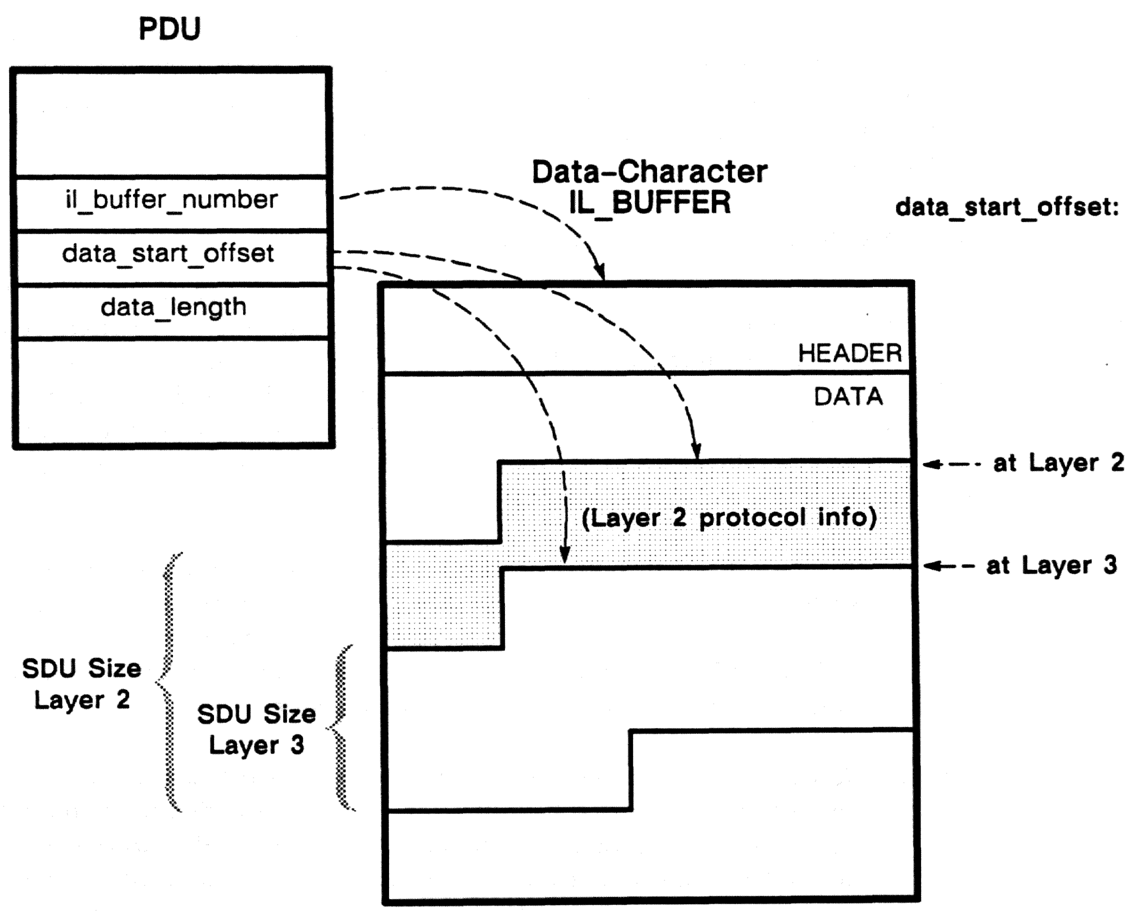


Figure 63-2 Primitive Data Unit and sample Data-Character Buffer being passed *up* the layers.

(B) Primitive Data Units

Like interlayer message buffers, PDU's have a format that is dependent on which direction the primitive is being passed. Refer again to Figure 63-1 and Figure 63-2.

1. *IL buffer number*. The buffer number to be passed with the primitive is always stored in the primitive. This buffer number is actually an 80286-processor segment number.

2. *Data-start offset*. The offset to the beginning of the service data unit for a given layer is different for the two types of buffers. In a pointer-list buffer going down the layers, the data-start offset will indicate the offset from the beginning of the buffer to the list-header node. This offset will vary if different linked lists have been started at different layers. Each list will have its own list header. In a data-character buffer going up the layers, the data-start offset will change from layer to layer. For example, a buffer containing X.25 data that is being passed from Layer 2 to Layer 3 will have an offset at Layer 3 two bytes beyond the offset at Layer 2.

3. *Data length*. The size of the SDU in a data-character buffer also varies from layer to layer. In the example just given, the SDU will be smaller by two bytes at Layer 3 than it was at Layer 2. In pointer-list buffers, the length of all data is unknown at any given layer.

(C) Accessing Information in Structures

There are two stages that are preliminary to accessing the information in these structures. The first step is to convert the 80286-processor segment number into a 32-bit address. The second stage is to place a pointer, in the shape of an IL buffer structure, at that address. Let's use an IL buffer as an example.

1. *Converting a segment number*. The IL-buffer segment number is returned any time you access one of the external, protocol-independent *il_buffer* variables listed in Table 63-1. These variables have names like *m_lo_dl_il_buff* and *up_n_il_buff*.

   To make a pointer to an IL buffer, (1) shift the 80286 segment number to the left sixteen bits, since a full address in the 80286 is 32 bits long; (2) cast it as a *long*, so that the segment number is in the high 16 bits and the offset to a buffer for that segment is zero (the low 16 bits); and (3) cast it as a pointer. The following expression will take care of all three requirements:

   ```
   (void *) ((long) m_lo_dl_il_buff <<16);
   ```

   Now you have a pointer to the first memory location of the most recent monitor-mode IL buffer passed up from Layer 2 to Layer 3. An upward-moving IL buffer was illustrated in Figure 63-2. The precise structure of both the IL buffer is given in the following declaration.

```
{
 struct il_buffer
   {
     unsigned short lock;
     unsigned short maintain_bits;
     unsigned short buffer_size;
     unsigned short transmit_tag;
     unsigned short receive_tag;
     unsigned long char_buff_frame_start;
     unsigned long char_buff_frame_end;
     unsigned short tick_count_high;
     unsigned short tick_count_mid;
     unsigned short tick_count_low;
     unsigned short available_space_offset;
     unsigned short bytes_remaining;
     unsigned long bcc_indicator;
     unsigned char data [4064];
   };
}
```

2. *Create a structure-pointer at a given address.* First, declare the structure of *il_buffer*, as indicated above. Then declare *il_buffer_pointer* as a structure-pointer, as follows:

   ```
   struct il_buffer * il_buffer_pointer;
   ```

   Converting the segment number and assigning it to *il_buffer_pointer* may be accomplished with this one statement:

   ```
   il_buffer_pointer = (void *) ((long) m_lo_dl_il_buff <<16);
   ```

   Now a structure has been created around the most recent upward-moving IL buffer at Layer 3. This means that rather than moving a pointer around in the IL buffer, you can access elements in the buffer directly. The *tick_count_low* variable, for example, would be called *il_buffer_pointer->tick_count_low*. (The -> operator is used in place of the dot operator in structure-pointers.)

   The first element of the *data* string would be called *il_buffer_pointer->data[0]*. Here is a program that displays on the prompt line the fifth data element, the packet-type byte, in every IL buffer that is monitored at Layer 3.

```
{
  extern event m_lo_dl_prmtv;
  extern volatile unsigned short m_lo_dl_il_buff;
  struct il_buffer
    {
      unsigned short lock;
      unsigned short maintain_bits;
      unsigned short buffer_size;
      unsigned short transmit_tag;
      unsigned short receive_tag;
      unsigned long char_buff_frame_start;
      unsigned long char_buff_frame_end;
      unsigned short tick_count_high;
      unsigned short tick_count_mid;
      unsigned short tick_count_low;
      unsigned short available_space_offset;
      unsigned short bytes_remaining;
      unsigned long bcc_indicator;
      unsigned char data [4064];
    };
  struct il_buffer * il_buffer_pointer;
}
LAYER: 3
     STATE: monitor_il_buffers
        CONDITIONS:
        {
         m_lo_dl_prmtv
        }
        ACTIONS:
        {
          il_buffer_pointer = (void *) ((long) m_lo_dl_il_buff <<16);
          pos_cursor (0,0);
         displayf ("%02x ", il_buffer_pointer->data[4]);
        }
```

If you run this program, be sure to load in the Layer 2 and Layer 3 personality packages for X.25. These packages will take care of delivery of the monitor primitives to Layer 3.

Table 63-1
OSI Structures

Type	Variable	Value (hex/decimal)	Meaning
Structure Name: pdu			Structure of an OSI primitive data unit (PDU). Declared as type *struct.* Use this structure as follows. Declare the entire structure. Make a pointer to a PDU by shifting *m*_lo_*dl*_pdu_seg (or up_*n*_pdu_seg) 16 bits to the left. Then convert this pointer to a pointer to a PDU structure: struct pdu * pdu_pointer pdu_pointer =(void *)((long)m_lo_dl_pdu_seg << 16). Reference a structure-pointer variable as follows: pdu_pointer->primitive_code.
unsigned char	primitive_code		Codes for OSI variables are listed in Table 63-2 through Table 63-8. For Layer 3 primitive codes, for example, refer to Table 63-4. The value of this variable is also stored in external variable *m*_lo_*dl*_prmtv_code (or up_*n*_prmtv_code).
unsigned char	path	*0-8*	Path number, both directions. The value of this variable is also stored in external variable *m*_lo_*dl*_prmtv_path (or up_*n*_prmtv_path).
unsigned long	parameter		For future use. At present, under user control.
unsigned short	relay_baton		Maintain bit passed with an interlayer-message buffer, both directions. Zero in this variable identifies maintain bit.
unsigned short	il_buffer_number		Segment number of the interlayer-message buffer, both directions. The value of this variable is also stored in external variable *m*_lo_*dl*_il_buff (or up_*n*_il_buff).
unsigned char	buffer_contents	0	Contains data-character buffer type. Must be used for buffer being passed up.
		1	Contains pointer-list buffer type. May be used for buffers being passed up, but is currently used primarily for buffers being passed down.
unsigned short	data_start_offset		Offset from the beginning of the buffer to the header node in the SDU of an interlayer-message buffer in an OSI primitive being sent down from a layer above. In a primitive being sent up from a layer below, it is the offset to the SDU. Varies according to the layer at which the buffer is located. For example, in a buffer passed up to Layer 3 from Layer 2, the offset would be to the beginning of the Layer 3 header, bypassing Layer 2 header information. The value of this variable is also stored in external variable *m*_lo_*dl*_sdu_*offset* (or up_*n*_sdu).
unsigned short	data_length		Length of the service data unit, including headers and user data. Only for primitives sent up from layer below. Varies with the layer where the buffer is located. For example, at Layer 3, length would exclude Layer 2 header (or trailer) information. The value of this variable is also stored in external variable m_lo_*dl*_sdu_size.

Table 63-1 (continued)

Type	Variable	Value (hex/decimal)	Meaning
Structure Name:	**il_buffer**		Structure of an interlayer-message buffer, both directions. Declared as type *struct.* Use this structure as follows. Declare the entire structure. Make a pointer to an il_buffer by shifting *m*_lo_*dl*_il_buff (or up_*n*_il_buff) 16 bits to the left: il_buffer_pointer = (void *)((long)(lo_dl_il_buff << 16). Then convert this pointer to a pointer to an il_buffer structure: struct il_buffer * il_buffer_pointer. Reference a structure-pointer variable as follows: il_buffer_pointer->tick_count_low.
unsigned short	lock	0	Internal variable which prevents structure from being updated by more than one program at the same time.
unsigned short	maintain_bits		Two-byte variable which provides the status of the maintain bits. A bit with a value of 1 is in use.
unsigned short	buffer_size	1000/4096	Currently, the only value.
unsigned short	transmit_tag		Bits 1-3 define bcc indication:
		0	no bcc
		1	good bcc
		2	bad bcc
		3	abort
		4	half bad bcc (DDCMP)
			Bits 4-8 for future use.
unsigned short	receive_tag		Bits 1-3 define bcc indication:
		0	no bcc
		1	good bcc
		2	bad bcc
		3	abort
		4	half bad bcc (DDCMP)
			Bit 4 identifies side of the line:
		0	td
		1	rd
			Bit 5—message buffer overflow:
		0	frame fits in buffer
		1	frame too large for the buffer
			Bits 6-8 for future use.
unsigned long	char_buff_frame_start		Location in the character buffer of the start of the buffered data.
unsigned long	char_buff_frame_end		Location in the character buffer of the end of the buffered data.

(il_buffer structure continued on next page)

Table 63-1 (continued)

Type	Variable	Value (hex/decimal)	Meaning
il_buffer (continued)			
unsigned short	tick_count_high		Value of internal variable that counts the number of times *l1_tick_count* has reached its maximum value. Together, the three *il_buffer* tick-count variables preserve at each layer the original time when the end of the data (BCC) was clocked into the buffer.
unsigned short	tick_count_mid		16 high-order bits of 32-bit *l1_tick_count*.
unsigned short	tick_count_low		16 low-order bits of 32-bit *l1_tick_count*.
unsigned short	available_space_offset		Offset to the next available space in the interlayer-message buffer.
unsigned short	bytes_remaining		Available number of bytes remaining in the buffer.
unsigned long	bcc_indicator	0	reserved
unsigned char	data [4064]		Contains all data including each layer's header information, as well as the first of two block check characters. Does not vary from layer to layer.
Structure Name: il_list_header			Structure of the header node in an interlayer-message buffer. Only for primitives sent down from the layer above. Declared as type *struct*. Use this structure as follows. Declare the entire structure. Make a pointer to an il_list_header by shifting up_*n*_il_buff (or *m*_lo_*dl*_il_buff) 16 bits to the left and adding the data_start_offset from the PDU structure (also stored as external variable up_*n*_sdu or *m*_lo_*dl*_sdu_*offset*): il_list_header_pointer = (void *)(((long)up_n_il_buff) << 16) + up_n_sdu). Then convert this pointer into a pointer to an il_list_header structure: struct il_list_header * il_list_header_pointer. Reference a structure-pointer variable as follows: il_list_header_pointer->last_node_offset.
unsigned short	first_node_offset		Offset from the beginning of the buffer to the first text node in the buffer. Varies according to the layer at which the buffer is located. At Layer 2, the offset would be to different starting node than at Layer 3.
unsigned short	last_node_offset		Offset to the location of the last text node in the buffer, from the beginning of the buffer.
unsigned long	reserved		reserved

Table 63-1 (continued)

Type	Variable	Value (hex/decimal)	Meaning
Structure Name:	**il_list_node**		Structure of text nodes in an interlayer-message buffer. Only for primitives sent down from the layer above. Declared as type *struct*. Use this structure as follows. Declare the entire structure. Make a pointer to an il_list_node by shifting up_*n*_il_buff (or *m*_lo_*dl*_il_buff) 16 bits to the left and adding the first_node_offset (or last_node_offset) from the il_list_header structure: il_list_node_pointer = (void *)(((long)up_n_il_buff << 16) + il_list_header_pointer->first_node_offset). Point to the next node as follows: next_node_pointer = (il_list_node_pointer + il_list_node_pointer->next_node_offset).
unsigned char *	data_pointer		Pointer to the data in a text node.
unsigned short	data_length		Length of the data in a text node.
unsigned short	next_node_offset		Offset to the location of the next text node in the buffer, from the beginning of the buffer. Generally, there is a text node for each layer's header information and one for the user data. A buffer that started at Layer 3 would have two text nodes, one for Layer 3 header information and one for user data (if any). At Layer 2, the buffer would acquire an additional text node.

63.2 Variables

OSI variables are layer-specific. The information stored in the OSI variables may be obtained by using the structure-pointer to IL buffers and primitives. But rather than requiring the user to repeat this process at each layer as a buffer moves through the layers, monitor and emulate variables have been made available at Layers 2–7 to store layer-specific, as well as general, information: the interlayer-buffer number, the offset to the service data unit, the path number, the size of the SDU, the segment number of the PDU, etc. There are also event variables which indicate that a primitive has been received at a given layer. Table 63-2 through Table 63-8 give the current OSI variables and their meanings.

The exchange of connect primitives shown primarily in Figure 30-4 is demonstrated in Figure 63-3 using C variables and routines. The SEND actions insert data in a buffer and send the buffer in a DATA REQ primitive. See Section 63.3 for an explanation of the *_insert_il_buff_list_cnt* and *send* primitive routines. The conditions use event variables to detect primitives and non-event variables to identify specific primitive types.

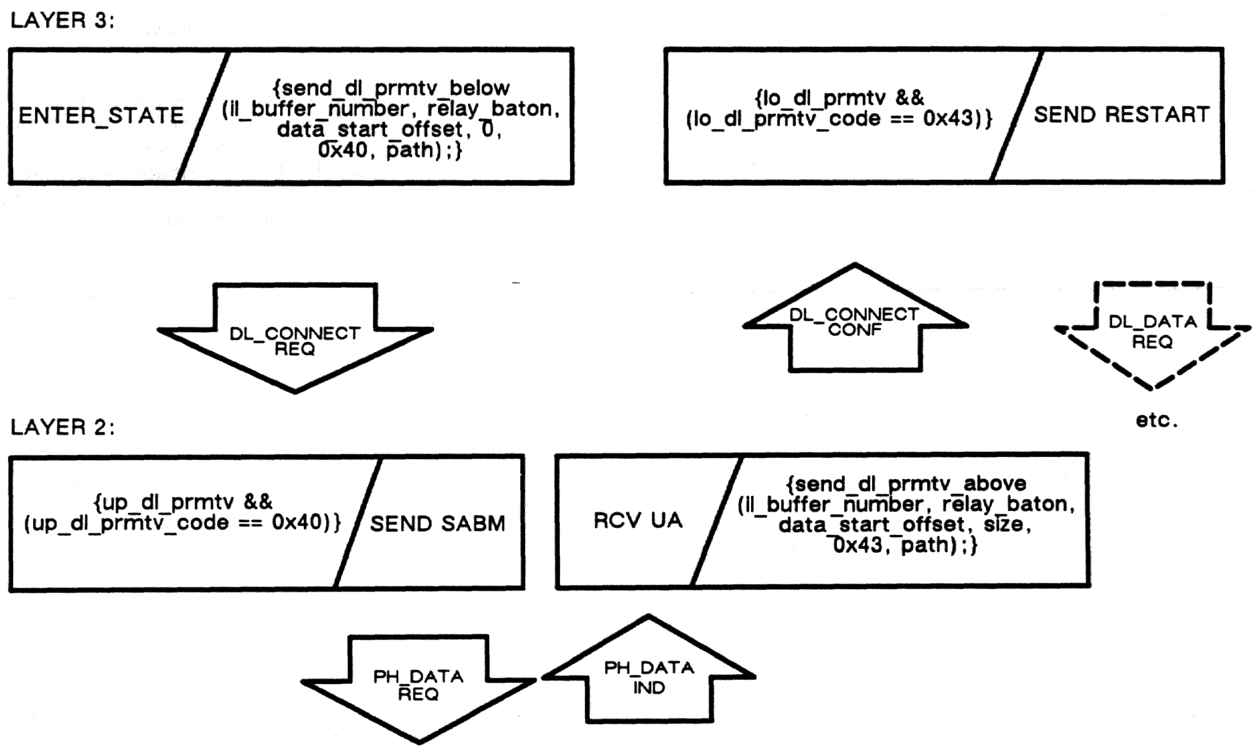


Figure 63-3 Layer 3 uses connect primitives to be sure that the Layer 2 entity below has established a link.

Table 63-2
Layer 1 OSI Variables

Type	Variable	Value (hex/decimal)	Meaning
extern volatile unsigned char	ph_prmtv_type	20/32	ph activate req
		21/33	ph activate ind
		22/34	ph activate resp
		23/35	ph activate conf
		24/36	ph data req
		25/37	ph data ind
		2a/42	ph reset req
		2b/43	ph reset ind
		2c/44	ph reset resp
		2d/45	ph reset conf
		2e/46	ph deactivate req
		2f/47	ph deactivate ind
		30/48	ph debug req
		31/49	ph debug ind
		33/51	ph error report ind
		34/52	ph xmit req
		35/53	ph set idle req
		38/56	ph mgt facility req
		39/57	ph mgt facility ind
			OSI primitive code for primitives moving between Layers 1 and 2. Line Setup configured for emulate mode only.

Table 63-3
Layer 2 OSI Variables

Type	Variable	Value (hex/decimal)	Meaning
extern event	lo_ph_prmtv		True when an OSI primitive is received at Layer 2 from Layer 1. Line Setup configured for emulate mode only.
extern event	m_lo_ph_prmtv		True when an OSI primitive is received at Layer 2 from Layer 1. Line Setup configured for emulate or monitor mode.
extern event	up_dl_prmtv		True when an OSI primitive is received at Layer 2 from Layer 3. Line Setup configured for emulate mode only.
extern volatile unsigned short	lo_ph_pdu_seg		OSI primitive data unit (PDU) IAPX-286 segment number received at Layer 2 from Layer 1. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate mode only.
extern volatile unsigned short	m_lo_ph_pdu_seg		OSI primitive data unit (PDU) IAPX-286 segment number received at Layer 2 from Layer 1. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate or monitor mode.
extern volatile const unsigned char	lo_ph_prmtv_code	21/33 23/35 25/37 2b/43 2d/45 2f/47 31/49 33/51 39/57	ph activate ind ph activate conf ph data ind ph reset ind ph reset conf ph deactivate ind ph debug ind ph error report ind ph mgt facility ind OSI primitive code received at Layer 2 in a PDU from Layer 1. Line Setup configured for emulate mode only.
extern volatile const unsigned char	m_lo_ph_prmtv_code	24/36 25/37	td ph data ind rd ph data ind OSI primitive code received at Layer 2 in a PDU from Layer 1. Line Setup configured for emulate or monitor mode.

Table 63-3 (continued)

Type	Variable	Value (hex/decimal)	Meaning
extern volatile const unsigned char	lo_ph_prmtv_path	*0-8*	Path number received at Layer 2 in a PDU from Layer 1. Line Setup configured for emulate mode only.
extern volatile const unsigned char	m_lo_ph_prmtv_path	*0-8*	Path number received at Layer 2 in a PDU from Layer 1. Line Setup configured for emulate or monitor mode.
extern volatile unsigned short	lo_ph_il_buff		Interlayer-buffer number (an iAPX-286 segment number) received at Layer 2 in a PDU from Layer 1. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate mode only.
extern volatile unsigned short	m_lo_ph_il_buff		Interlayer-buffer number (an iAPX-286 segment number) received at Layer 2 in a PDU from Layer 1. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate or monitor mode.
extern volatile unsigned short	lo_ph_sdu		In OSI primitive received at Layer 2 from Layer 1, the offset to where the service data unit begins. Line Setup configured for emulate mode only.
extern volatile unsigned short	m_lo_ph_sdu_offset		In OSI primitive received at Layer 2 from Layer 1, the offset to where the service data unit begins. Line Setup configured for emulate or monitor mode.
extern volatile unsigned short	m_lo_ph_sdu_size		Size of the service data unit in an interlayer-message buffer, displayed as SIZE on the Layer 2 trace screen. Received at Layer 2 from Layer 1. Same as *data_length* in a PDU. Line Setup configured for emulate or monitor mode.
extern volatile unsigned short	up_dl_pdu_seg		OSI primitive data unit (PDU) iAPX-286 segment number received at Layer 2 from Layer 3. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate mode only.

Table 63-3 (continued)

Type	Variable	Value (hex/decimal)	Meaning
extern volatile const unsigned char	up_dl_prmtv_code	40/64 42/66 44/68 48/72 4a/74 4c/76 4e/78 50/80 52/82 58/88	dl conn req dl conn resp dl data req dl expd data req dl reset req dl reset resp dl disconn req dl debug req dl unit data req dl mgt facility req OSI primitive code received at Layer 2 in a PDU from Layer 3. Line Setup configured for emulate mode only.
extern volatile const unsigned char	up_dl_prmtv_path	*0–8*	Path number received at Layer 2 in a PDU from Layer 3. Line Setup configured for emulate mode only.
extern volatile unsigned short	up_dl_il_buff		Interlayer-buffer number (an iAPX-286 segment number) received at Layer 2 in a PDU from Layer 3. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate mode only.
extern volatile unsigned short	up_dl_sdu		Offset to the start (header node) of the service data unit in an interlayer-message buffer. Received at Layer 2 from Layer 3. Same as *data_start_offset* in a PDU. Line Setup configured for emulate mode only.
extern unsigned long	l2_tick_count		32-bit *l1_tick_count* stored in header of most recent IL buffer passed up to Layer 2. Preserves at each layer the original time when the end of the data (BCC) was clocked into the buffer. Line Setup configured for emulate or monitor mode.

Table 63-4
Layer 3 OSI Variables

Type	Variable	Value (hex/decimal)	Meaning
extern event	lo_dl_prmtv		True when an OSI primitive is received at Layer 3 from Layer 2. Line Setup configured for emulate mode only.
extern event	m_lo_dl_prmtv		True when an OSI primitive is received at Layer 3 from Layer 2. Line Setup configured for emulate or monitor mode.
extern event	up_n_prmtv		True when an OSI primitive is received at Layer 3 from Layer 4. Line Setup configured for emulate mode only.
extern volatile unsigned short	lo_dl_pdu_seg		OSI primitive data unit (PDU) iAPX-286 segment number received at Layer 3 from Layer 2. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate mode only.
extern volatile unsigned short	m_lo_dl_pdu_seg		OSI primitive data unit (PDU) iAPX-286 segment number received at Layer 3 from Layer 2. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate or monitor mode.
extern volatile const unsigned char	lo_dl_prmtv_code	41/65 43/67 45/69 49/73 4b/75 4d/77 4f/79 51/81 53/83 55/85 59/89	dl conn ind dl conn conf dl data ind dl expd data ind dl reset ind dl reset conf dl disconn ind dl debug ind dl unit data ind dl error report ind dl mgt facility ind OSI primitive code received at Layer 3 in a PDU from Layer 2. Line Setup configured for emulate mode only.

Table 63-4 (continued)

Type	Variable	Value (hex/decimal)	Meaning
extern volatile const unsigned char	m_lo_dl_prmtv_code	44/68 45/69 48/72 49/73 54/84 55/85	td dl data ind rd dl data ind td dl expd data ind rd dl expd data ind td dl unit data ind rd dl unit data ind OSI primitive code received at Layer 3 in a PDU from Layer 2. Line Setup configured for emulate or monitor mode.
extern volatile const unsigned char	lo_dl_prmtv_path	*0–8*	Path number received at Layer 3 in a PDU from Layer 2. Line Setup configured for emulate mode only.
extern volatile const unsigned char	m_lo_dl_prmtv_path	*0–8*	Path number received at Layer 3 in a PDU from Layer 2. Line Setup configured for emulate or monitor mode.
extern volatile unsigned short	lo_dl_il_buff		Interlayer-buffer number (an iAPX-286 segment number) received at Layer 3 in a PDU from Layer 2. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate mode only.
extern volatile unsigned short	m_lo_dl_il_buff		Interlayer-buffer number (an iAPX-286 segment number) received at Layer-3 in a PDU from Layer 2. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate or monitor mode.
extern volatile unsigned short	lo_dl_sdu		In OSI primitive received at Layer 3 from Layer 2, the offset to where the service data unit begins. Line Setup configured for emulate mode only.
extern volatile unsigned short	m_lo_dl_sdu_offset		In OSI primitive received at Layer 3 from Layer 2, the offset to where the service data unit begins. Line Setup configured for emulate or monitor mode.
extern volatile unsigned short	m_lo_dl_sdu_size		Size of the service data unit in an interlayer-message buffer, displayed as SIZE on the Layer 3 trace screen. Received at Layer 3 from Layer 2. Same as *data_length* in a PDU. Line Setup configured for emulate or monitor mode.

Table 63-4 (continued)

Type	Variable	Value (hex/decimal)	Meaning
extern volatile unsigned short	up_n_pdu_seg		OSI primitive data unit (PDU) iAPX-286 segment number received at Layer 3 from Layer 4. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate mode only.
extern volatile const unsigned char	up_n_prmtv_code	60/96 62/98 64/100 66/102 68/104 6a/106 6c/108 6e/110 70/112 72/114 74/116 76/118 78/120	n conn req n conn resp n data req n data ack req n expd data req n reset req n reset resp n disconn req n debug req n unit data req n qual data req n qual data ack req n mgt facility req OSI primitive code received at Layer 3 in a PDU from Layer 4. Line Setup configured for emulate mode only.
extern volatile const unsigned char	up_n_prmtv_path	*0-8*	Path number received at Layer 3 in a PDU from Layer 4. Line Setup configured for emulate mode only.
extern volatile unsigned short	up_n_il_buff		Interlayer-buffer number (an iAPX-286 segment number) received at Layer 3 in a PDU from Layer 4. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate mode only.
extern volatile unsigned short	up_n_sdu		Offset to the start (header node) of the service data unit in an interlayer-message buffer. Received at Layer 3 from Layer 4. Same as *data_start_offset* in a PDU. Line Setup configured for emulate mode only.
extern unsigned long	l3_tick_count		32-bit *l1_tick_count* stored in header of most recent IL buffer passed up to Layer 3. Preserves at each layer the original time when the end of the data (BCC) was clocked into the buffer. Line Setup configured for emulate or monitor mode.

Table 63-5
Layer 4 OSI Variables

Type	Variable	Value (hex/decimal)	Meaning
extern event	lo_n_prmtv		True when an OSI primitive is received at Layer 4 from Layer 3. Line Setup configured for emulate mode only.
extern event	m_lo_n_prmtv		True when an OSI primitive is received at Layer 4 from Layer 3. Line Setup configured for emulate or monitor mode.
extern event	up_t_prmtv		True when an OSI primitive is received at Layer 4 from Layer 5. Line Setup configured for emulate mode only.
extern volatile unsigned short	lo_n_pdu_seg		OSI primitive data unit (PDU) iAPX-286 segment number received at Layer 4 from Layer 3. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate mode only.
extern volatile unsigned short	m_lo_n_pdu_seg		OSI primitive data unit (PDU) iAPX-286 segment number received at Layer 4 from Layer 3. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate or monitor mode.
extern volatile const unsigned char	lo_n_prmtv_code	61/97	n conn ind
		63/99	n conn conf
		65/101	n data ind
		67/103	n data ack ind
		69/105	n expd data ind
		6b/107	n reset ind
		6d/109	n reset conf
		6f/111	n disconn ind
		71/113	n debug ind
		73/115	n unit data ind
		75/117	n qual data ind
		77/119	n qual data ack ind
		79/121	n mgt facility ind
		7a/122	n error report ind
			OSI primitive code received at Layer 4 in a PDU from Layer 3. Line Setup configured for emulate mode only.

Table 63-5 (continued)

Type	Variable	Value (hex/decimal)	Meaning
extern volatile const unsigned char	m_lo_n_prmtv_code	64/100 65/101 68/102 69/103 74/116 75/117	td n data ind rd n data ind td n expd data ind rd n expd data ind td n unit data ind rd n unit data ind OSI primitive code received at Layer 4 in a PDU from Layer 3. Line Setup configured for emulate or monitor mode.
extern volatile const unsigned char	lo_n_prmtv_path	*0-8*	Path number received at Layer 4 in a PDU from Layer 3. Line Setup configured for emulate mode only.
extern volatile const unsigned char	m_lo_n_prmtv_path	*0-8*	Path number received at Layer 4 in a PDU from Layer 3. Line Setup configured for emulate or monitor mode.
extern volatile unsigned short	lo_n_il_buff		Interlayer-buffer number (an iAPX-286 segment number) received at Layer 4 in a PDU from Layer 3. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate mode only.
extern volatile unsigned short	m_lo_n_il_buff		Interlayer-buffer number (an iAPX-286 segment number) received at Layer 4 in a PDU from Layer 3. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate or monitor mode.
extern volatile unsigned short	lo_n_sdu		In OSI primitive received at Layer 4 from Layer 3, the offset to where the service data unit begins. Line Setup configured for emulate mode only.
extern volatile unsigned short	m_lo_n_sdu_offset		In OSI primitive received at Layer 4 from Layer 3, the offset to where the service data unit begins. Line Setup configured for emulate or monitor mode.
extern volatile unsigned short	m_lo_n_sdu_size		Size of the service data unit in an interlayer-message buffer. Received at Layer 4 from Layer 3. Same as *data_length* in a PDU. Line Setup configured for emulate or monitor mode.

Table 63-5 (continued)

Type	Variable	Value (hex/decimal)	Meaning
extern volatile unsigned short	up_t_pdu_seg		OSI primitive data unit (PDU) IAPX-286 segment number received at Layer 4 from Layer 5. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate mode only.
extern volatile const unsigned char	up_t_prmtv_code	80/128 82/130 84/132 88/136 8e/142 90/144 92/146 98/152	t conn req t conn resp t data req t expd data req t disconn req t debug req t unit data req t mgt facility req OSI primitive code received at Layer 4 in a PDU from Layer 5. Line Setup configured for emulate mode only.
extern volatile const unsigned char	up_t_prmtv_path	*0-8*	Path number received at Layer 4 in a PDU from Layer 5. Line Setup configured for emulate mode only.
extern volatile unsigned short	up_t_il_buff		Interlayer-buffer number (an IAPX-286 segment number) received at Layer 4 in a PDU from Layer 5. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate mode only.
extern volatile unsigned short	up_t_sdu		Offset to the start (header node) of the service data unit in an interlayer-message buffer. Received at Layer 4 from Layer 5. Same as *data_start_offset* in a PDU. Line Setup configured for emulate mode only.
extern unsigned long	l4_tick_count		32-bit *l1_tick_count* stored in header of most recent IL buffer passed up to Layer 4. Preserves at each layer the original time when the end of the data (BCC) was clocked into the buffer. Line Setup configured for emulate or monitor mode.

Table 63-6
Layer 5 OSI Variables

Type	Variable	Value (hex/decimal)	Meaning
extern event	lo_t_prmtv		True when an OSI primitive is received at Layer 5 from Layer 4. Line Setup configured for emulate mode only.
extern event	m_lo_t_prmtv		True when an OSI primitive is received at Layer 5 from Layer 4. Line Setup configured for emulate or monitor mode.
extern event	up_s_prmtv		True when an OSI primitive is received at Layer 5 from Layer 6. Line Setup configured for emulate mode only.
extern volatile unsigned short	lo_t_pdu_seg		OSI primitive data unit (PDU) iAPX-286 segment number received at Layer 5 from Layer 4. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate mode only.
extern volatile unsigned short	m_lo_t_pdu_seg		OSI primitive data unit (PDU) iAPX-286 segment number received at Layer 5 from Layer 4. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate or monitor mode.
extern volatile const unsigned char	lo_t_prmtv_code	81/129 83/131 85/133 89/137 8f/143 91/145 93/147 95/149 99/153	t conn ind t conn conf t data ind t expd data ind t disconn ind t debug ind t unit data ind t error report ind t mgt facility ind OSI primitive code received at Layer 5 in a PDU from Layer 4. Line Setup configured for emulate mode only.
extern volatile const unsigned char	m_lo_t_prmtv_code	84/132 85/133 88/136 89/137 94/148 95/149	td t data ind rd t data ind td t expd data ind rd t expd data ind td t unit data ind rd t unit data ind OSI primitive code received at Layer 5 in a PDU from Layer 4. Line Setup configured for emulate or monitor mode.

Table 63-6 (continued)

Type	Variable	Value (hex/decimal)	Meaning
extern volatile const unsigned char	lo_t_prmtv_path	*0-8*	Path number received at Layer 5 in a PDU from Layer 4. Line Setup configured for emulate mode only.
extern volatile const unsigned char	m_lo_t_prmtv_path	*0-8*	Path number received at Layer 5 in a PDU from Layer 4. Line Setup configured for emulate or monitor mode.
extern volatile unsigned short	lo_t_il_buff		Interlayer-buffer number (an IAPX-286 segment number) received at Layer 5 in a PDU from Layer 4. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate mode only.
extern volatile unsigned short	m_lo_t_il_buff		Interlayer-buffer number (an IAPX-286 segment number) received at Layer 5 in a PDU from Layer 4. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate or monitor mode.
extern volatile unsigned short	lo_t_sdu		In OSI primitive received at Layer 5 from Layer 4, the offset to where the service data unit begins. Line Setup configured for emulate mode only.
extern volatile unsigned short	m_lo_t_sdu_offset		In OSI primitive received at Layer 5 from Layer 4, the offset to where the service data unit begins. Line Setup configured for emulate or monitor mode.
extern volatile unsigned short	m_lo_t_sdu_size		Size of the service data unit in an interlayer-message buffer. Received at Layer 5 from Layer 4. Same as *data_length* in a PDU. Line Setup configured for emulate or monitor mode.
extern volatile unsigned short	up_s_pdu_seg		OSI primitive data unit (PDU) IAPX-286 segment number received at Layer 5 from Layer 6. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate mode only.

Table 63-6 (continued)

Type	Variable	Value (hex/decimal)	Meaning
extern volatile const unsigned char	up_s_prmtv_code	a0/160 a2/162 a4/164 a8/168 ac/172 ae/174 b0/176 b2/178 b8/184	s conn req s conn resp s data req s expd data req s release req s release resp s debug req s unit data req s mgt facility req OSI primitive code received at Layer 5 in a PDU from Layer 6. Line Setup configured for emulate mode only.
extern volatile const unsigned char	up_s_prmtv_path	*0-8*	Path number received at Layer 5 in a PDU from Layer 6. Line Setup configured for emulate mode only.
extern volatile unsigned short	up_s_il_buff		Interlayer-buffer number (an IAPX-286 segment number) received at Layer 5 in a PDU from Layer 6. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate mode only.
extern volatile unsigned short	up_s_sdu		Offset to the start (header node) of the service data unit in an interlayer-message buffer. Received at Layer 5 from Layer 6. Same as *data_start_offset* in a PDU. Line Setup configured for emulate mode only.
extern unsigned long	l5_tick_count		32-bit *l1_tick_count* stored in header of most recent IL buffer passed up to Layer 5. Preserves at each layer the original time when the end of the data (BCC) was clocked into the buffer. Line Setup configured for emulate or monitor mode.

Table 63-7
Layer 6 OSI Variables

Type	Variable	Value (hex/decimal)	Meaning
extern event	lo_s_prmtv		True when an OSI primitive is received at Layer 6 from Layer 5. Line Setup configured for emulate mode only.
extern event	m_lo_s_prmtv		True when an OSI primitive is received at Layer 6 from Layer 5. Line Setup configured for emulate or monitor mode.
extern event	up_p_prmtv		True when an OSI primitive is received at Layer 6 from Layer 7. Line Setup configured for emulate mode only.
extern volatile unsigned short	lo_s_pdu_seg		OSI primitive data unit (PDU) iAPX-286 segment number received at Layer 6 from Layer 5. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate mode only.
extern volatile unsigned short	m_lo_s_pdu_seg		OSI primitive data unit (PDU) iAPX-286 segment number received at Layer 6 from Layer 5. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate or monitor mode.
extern volatile const unsigned char	lo_s_prmtv_code	a1/161 a3/163 a5/165 a9/169 ad/173 af/175 b1/177 b3/179 b5/181 b9/185	s conn ind s conn conf s data ind s expd data ind s release ind s release conf s debug ind s unit data ind s error report ind s mgt facility ind OSI primitive code received at Layer 6 in a PDU from Layer 5. Line Setup configured for emulate mode only.
extern volatile const unsigned char	m_lo_s_prmtv_code	a4/164 a5/165 a8/168 a9/169 b4/180 b5/181	td s data ind rd s data ind td s expd data ind rd s expd data ind td s unit data ind rd s unit data ind OSI primitive code received at Layer 6 in a PDU from Layer 5. Line Setup configured for emulate or monitor mode.

Table 63-7 (continued)

Type	Variable	Value (hex/decimal)	Meaning
extern volatile const unsigned char	lo_s_prmtv_path	*0-8*	Path number received at Layer 6 in a PDU from Layer 5. Line Setup configured for emulate mode only.
extern volatile const unsigned char	m_lo_s_prmtv_path	*0-8*	Path number received at Layer 6 in a PDU from Layer 5. Line Setup configured for emulate or monitor mode.
extern volatile unsigned short	lo_s_il_buff		Interlayer-buffer number (an iAPX-286 segment number) received at Layer 6 in a PDU from Layer 5. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate mode only.
extern volatile unsigned short	m_lo_s_il_buff		Interlayer-buffer number (an iAPX-286 segment number) received at Layer 6 in a PDU from Layer 5. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate or monitor mode.
extern volatile unsigned short	lo_s_sdu		In OSI primitive received at Layer 6 from Layer 5, the offset to where the service data unit begins. Line Setup configured for emulate mode only.
extern volatile unsigned short	m_lo_s_sdu_offset		In OSI primitive received at Layer 6 from Layer 5, the offset to where the service data unit begins. Line Setup configured for emulate or monitor mode.
extern volatile unsigned short	m_lo_s_sdu_size		Size of the service data unit in an interlayer-message buffer. Received at Layer 6 from Layer 5. Same as *data_length* in a PDU. Line Setup configured for emulate or monitor mode.
extern volatile unsigned short	up_p_pdu_seg		OSI primitive data unit (PDU) iAPX-286 segment number received at Layer 6 from Layer 7. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate mode only.

Table 63-7 (continued)

Type	Variable	Value (hex/decimal)	Meaning
extern volatile const unsigned char	up_p_prmtv_code	c0/192 c2/194 c4/196 c8/200 cc/204 ce/206 d0/208 d2/210 d8/216	p conn req p conn resp p data req p expd data req p release req p release resp p debug req p unit data req p mgt facility req OSI primitive code received at Layer 6 from Layer 7 in a PDU. Line Setup configured for emulate mode only.
extern volatile const unsigned char	up_p_prmtv_path	*0-8*	Path number received at Layer 6 from Layer 7 in a PDU. Line Setup configured for emulate mode only.
extern volatile unsigned short	up_p_il_buff		Interlayer-buffer number (an iAPX-286 segment number) received at Layer 6 from Layer 7 in a PDU. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate mode only.
extern volatile unsigned short	up_p_sdu		Offset to the start (header node) of the service data unit in an interlayer-message buffer. Received at Layer 6 from Layer 7. Same as *data_start_offset* in a PDU. Line Setup configured for emulate mode only.
extern unsigned long	l6_tick_count		32-bit *l1_tick_count* stored in header of most recent IL buffer passed up to Layer 6. Preserves at each layer the original time when the end of the data (BCC) was clocked into the buffer. Line Setup configured for emulate or monitor mode.

Table 63-8
Layer 7 OSI Variables

Type	Variable	Value (hex/decimal)	Meaning
extern event	lo_p_prmtv		True when an OSI primitive is received at Layer 7 from Layer 6. Line Setup configured for emulate mode only.
extern event	m_lo_p_prmtv		True when an OSI primitive is received at Layer 7 from Layer 6. Line Setup configured for emulate or monitor mode.
extern volatile unsigned short	lo_p_pdu_seg		OSI primitive data unit (PDU) IAPX-286 segment number received at Layer 7 from Layer 6. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate mode only.
extern volatile unsigned short	m_lo_p_pdu_seg		OSI primitive data unit (PDU) IAPX-286 segment number received at Layer 7 from Layer 6. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate or monitor mode.
extern volatile const unsigned char	lo_p_prmtv_code	c1/193 c3/195 c5/197 c9/201 cd/205 cf/207 d1/209 d3/211 d5/213 d9/217	p conn ind p conn conf p data ind p expd data ind p release ind p release conf p debug ind p unit data ind p error report ind p mgt facility ind OSI primitive code received at Layer 7 in a PDU from Layer 6. Line Setup configured for emulate mode only.
extern volatile const unsigned char	m_lo_p_prmtv_code	c4/196 c5/197 c8/200 c9/201 d4/212 d5/213	td p data ind rd p data ind td p expd data ind rd p expd data ind td p unit data ind rd p unit data ind OSI primitive code received at Layer 7 in a PDU from Layer 6. Line Setup configured for emulate or monitor mode.
extern volatile const unsigned char	lo_p_prmtv_path	*0-8*	Path number received at Layer 7 in a PDU from Layer 6. Line Setup configured for emulate mode only.

Table 63-8 (continued)

Type	Variable	Value (hex/decimal)	Meaning
extern volatile const unsigned char	m_lo_p_prmtv_path	*0-8*	Path number received at Layer 7 in a PDU from Layer 6. Line Setup configured for emulate or monitor mode.
extern volatile unsigned short	lo_p_il_buff		Interlayer-buffer number (an IAPX-286 segment number) received at Layer 7 in a PDU from Layer 6. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate mode only.
extern volatile unsigned short	m_lo_p_il_buff		Interlayer-buffer number (an IAPX-286 segment number) received at Layer 7 in a PDU from Layer 6. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate or monitor mode.
extern volatile unsigned short	lo_p_sdu		In OSI primitive received at Layer 7 from Layer 6, the offset to where the service data unit begins. Line Setup configured for emulate mode only.
extern volatile unsigned short	m_lo_p_sdu_offset		In OSI primitive received at Layer 7 from Layer 6, the offset to where the service data unit begins. Line Setup configured for emulate or monitor mode.
extern volatile unsigned short	m_lo_p_sdu_size		Size of the service data unit in an interlayer-message buffer. Received at Layer 7 from Layer 6. Same as *data_length* in a PDU. Line Setup configured for emulate or monitor mode.
extern unsigned long	l7_tick_count		32-bit *l1_tick_count* stored in header of most recent IL buffer passed up to Layer 7. Preserves at each layer the original time when the end of the data (BCC) was clocked into the buffer. Line Setup configured for emulate or monitor mode.

63.3 Routines

OSI routines available at each layer make sending primitives to a layer above or below possible (see Figure 63-3). The routine name and its arguments provide the same information as the softkey selections on the Protocol Spreadsheet. (In the early phases of compiling the program, the C translator uses the routines to convert the spreadsheet softkey-token primitives into C.) All routines are protocol-independent.

(A) Layer-Independent OSI Routines

The following interlayer buffer service routines operate at any layer, regardless of protocol (or in the absence of a protocol package).

_get_il_msg_buff

Synopsis

```
extern void _get_il_msg_buff(buffer_number_ptr, maintain_bit_ptr);
unsigned short * buffer_number_ptr;
unsigned short * maintain_bit_ptr;
```

Description

The *_get_il_msg_buff* routine gets a free interlayer message buffer from the pool and returns the buffer number to the caller for use in subsequent calls to other interlayer buffer services. It also returns a maintain bit for use in the freeing operation.

Inputs

The first parameter is a pointer to the location where the buffer number is to be stored. The buffer number that is returned is actually an iAPX-286 segment number which can be converted to a pointer by shifting it 16 bits to the left. If there is no free buffer available, the routine will wait for one to become available.

The second parameter is a pointer to the location where the maintain bit will be stored. Since it must be used in the freeing operation, the maintain bit value should not be modified. The zero bit in this variable indicates your maintain bit.

Example

The variables in which the returned buffer number and maintain bit will be stored must be declared. When calling the routine, reference the addresses of these variables.

```
{
 unsigned short il_buffer_number;
 unsigned short relay_baton;
}
LAYER: 4
     STATE: get_a_buffer
        CONDITIONS: KEYBOARD " "
        ACTIONS:
        {
         _get_il_msg_buff(&il_buffer_number, &relay_baton);
        }
```

The routine will get a buffer number and store it in variable *il_buffer_number*. It will also return a maintain bit and store it in variable *relay_baton*.

_start_il_buff_list

Synopsis

```
extern void _start_il_buff_list(il_buffer_number, start_offset_ptr);
unsigned short il_buffer_number;
unsigned short * start_offset_ptr;
```

Description

The *_start_il_buff_list* routine starts a linked list of text inside an interlayer message buffer. The list is made up of a header node and text nodes. The header node contains offsets to the first and last text nodes. Each text node contains a pointer to the actual text, the length of the text, and the offset to the next text node. This routine actually creates the header node inside the interlayer message buffer and initializes the first and last text node offsets to zero, indicating an empty list. It will return the offset to the list header node for use in subsequent list service calls.

Inputs

The first parameter is the interlayer message buffer number that will contain the list.

The second parameter is a pointer to the location where the offset to the list header will be stored. The returned offset will be zero if there is insufficient room in the buffer for the header node and one text node. Otherwise, it is the offset from the beginning of the message buffer to the start of the header node.

To convert the offset into a pointer, shift the buffer number 16 bits to the left and add the offset:

```
(void *)(((long)il_buffer_number << 16) + data_start_offset);
```

Example

Get a buffer and start a linked list. The variable in which the returned offset will be stored must be declared. When calling the routine, reference the address of this variable.

```
{
 unsigned short il_buffer_number;
 unsigned short relay_baton;
 unsigned short data_start_offset;
}
```

```
STATE: start_a_list
   CONDITIONS: KEYBOARD " "
   ACTIONS:
   {
    _get_il_msg_buff(&il_buffer_number, &relay_baton);
    _start_il_buff_list(il_buffer_number, &data_start_offset);
/* See _insert_il_buff_list_cnt routine on how information is inserted in the buffer. */
   }
```

The routine will get the offset to the header node and store it in variable *data_start_offset*.

_dup_il_buff_list_start

Synopsis

```
extern unsigned short _dup_il_buff_list_start(il_buffer_number, start_offset,
   new_start_offset_ptr);
unsigned short il_buffer_number;
unsigned short start_offset;
unsigned short * new_start_offset_ptr;
```

Description

This routine duplicates the header node of a pointer list. In order for a layer to retain the ability to resend a buffer—that is, to reference again the same list header with the same first-node offset—it must keep its own linked list safe from data inserted at a layer below. The *_dup_il_buff_list_start* routine allows the *lower* layer to start its own list.

If the lower layer will *insert* data into the buffer, it need duplicate only the list header ("*list_start*"), not the entire list. If the layer will append data to the end of the buffer, it must duplicate the complete linked list via the *_dup_il_buff_list* routine.

Inputs

The first parameter is the interlayer message buffer number in which the header node will be duplicated.

The second parameter is the offset to the header node to be duplicated.

The third parameter is a pointer to the location where the offset to the new header node will be stored.

Returns

This routine returns zero if there is not enough room in the buffer for the duplicated header node and at least one list node.

Example

Duplicate the header node of a buffer passed down from Layer 3.

```
{
 extern volatile unsigned short up_dl_il_buff;
 extern volatile unsigned short up_dl_sdu;
 unsigned short l2_data_start_offset;
}
LAYER: 3
     STATE: message
        CONDITIONS: KEYBOARD " "
        ACTIONS: DL_DATA REQ "␐␃␀((FOX))"
LAYER: 2
     STATE: duplicate_header
        CONDITIONS: DL_DATA REQ
        ACTIONS:
        {
         _dup_il_buff_list_start(up_dl_il_buff, up_dl_sdu, &l2_data_start_offset);
/* See _insert_il_buff_list_cnt routine on how information is inserted in the buffer. */
        }
```

_dup_il_buff_list

Synopsis

```
extern unsigned short _dup_il_buff_list(il_buffer_number, start_offset, new_start_offset_ptr);
unsigned short il_buffer_number;
unsigned short start_offset;
unsigned short * new_start_offset_ptr;
```

Description

This routine duplicates an entire pointer list. In order for a layer to be able to retain the ability to resend a buffer—that is, to reference again the same list header with the same first- and last-node offsets—it must keep its own linked list safe from data inserted and appended at a layer below. The *_dup_il_buff_list* routine allows the *lower* layer to have its own list.

If the lower layer will append data to the buffer, it should duplicate the entire linked list. If the layer will only insert data into the buffer, it need only duplicate the header node via the *_dup_il_buff_list_start* routine.

Inputs

The first parameter is the interlayer message buffer number in which the list will be duplicated.

The second parameter is the offset to the header node of the list to be duplicated.

The third parameter is a pointer to the location where the offset to the header node for the new list will be stored.

Returns

This routine returns zero if the duplication is successful. If there is not enough room in the buffer to duplicate the list, one is returned.

Example

Duplicate the entire pointer list of a buffer passed down from Layer 3.

```
{
 extern volatile unsigned short up_dl_il_buff;
 extern volatile unsigned short up_dl_sdu;
 unsigned short l2_data_start_offset;
}
LAYER: 3
     STATE: message
        CONDITIONS: KEYBOARD “ ”
        ACTIONS: DL_DATA REQ “␐␃␀((FOX))”
LAYER: 2
     STATE: duplicate_list
        CONDITIONS: DL_DATA REQ
        ACTIONS:
        {
         _dup_il_buff_list(up_dl_il_buff, up_dl_sdu, &l2_data_start_offset);
/* See _append_il_buff_list_cnt routine on how information is appended to the buffer. */
        }
```

_open_space_in_il_buff

Synopsis

```
extern void _open_space_in_il_buff(il_buffer_number, length, space_offset_ptr);
unsigned short il_buffer_number;
unsigned short length;
unsigned short * space_offset_ptr;
```

Description

The *_open_space_in_il_buff* routine opens up the requested amount of space in the specified interlayer message buffer. It returns an offset from the beginning of the buffer to the start of the open space.

Inputs

The first parameter is the interlayer message buffer number in which space is to be made.

The second parameter is the amount of space (number of bytes) requested.

The third parameter is a pointer to the location where the returned offset will be stored. The returned offset will be zero if there is insufficient room in the buffer.

To convert the offset into a pointer, shift the buffer number 16 bits to the left and add the offset:

```
(void *)(((long)il_buffer_number << 16) + available_space_offset);
```

Example

Always open space in the buffer if you are going to copy data (usually header information) into the buffer. If you are not going to copy data into the buffer, but reference its location in memory outside the buffer (usually user data), you do not need to open space.

The variable in which the returned offset will be stored must be declared. When calling the routine, reference the address of this variable. The length may be entered as a numeric value, in which case a length variable need not be declared.

For example, a buffer at Layer 3 will have three X.25-header bytes inserted. The call for space to hold the header would look like this:

```
{
 unsigned short il_buffer_number;
 unsigned short relay_baton;
 unsigned short data_start_offset;
 unsigned short available_space_offset;
}
      STATE: get_space
         CONDITIONS: KEYBOARD " "
         ACTIONS:
         {
          _get_il_msg_buff(&il_buffer_number, &relay_baton);
          _start_il_buff_list(il_buffer_number, &data_start_offset);
          _open_space_in_il_buff(il_buffer_number, 3, &available_space_offset);
/* See _insert_il_buff_list_cnt routine on how information is inserted in the buffer. */
         }
```

The routine will get the offset to the next available space in the buffer and store it in variable *available_space_offset*.

Once space has been opened, the buffer-number and available-space variables can be converted into an open-space pointer. With this pointer, data can be copied into the space. The pointer can then be referenced in an *_insert_il_buff_list_cnt* routine, so that the opened space becomes threaded onto the linked list in the IL buffer. See the programming example under *_insert_il_buff_list_cnt*.

_free_il_msg_buff

Synopsis

extern void _free_il_msg_buff(il_buffer_number, relay_baton);
unsigned short il_buffer_number;
unsigned short relay_baton;

Description

The *_free_il_msg_buff* routine returns an interlayer message buffer to the pool of free buffers. Before actually returning the buffer to the pool, this routine verifies that all maintain bits have been reset, assuring that all users have freed this buffer.

Inputs

The first parameter is the interlayer-buffer number to be freed.

The second parameter is the maintain bit associated with the buffer user to be freed.

Example

See *_set_maint_buff_bit* routine.

_set_maint_buff_bit

Synopsis

extern void _set_maint_buff_bit(il_buffer_number, new_bit_ptr);
unsigned short il_buffer_number;
*unsigned short * new_bit_ptr;*

Description

The *_set_maint_buff_bit* routine sets a new maintain bit for a given interlayer message buffer. It returns that bit to the caller to be used in the freeing operation.

The maintain bit allocated in the *_get_il_msg_buff* routine should be considered valid only for the layer at which it was obtained. Once you pass a buffer, the maintain bit will hold the buffer at the next layer only until action on it has been processed. (In Spreadsheet terms, the buffer will be held until the ACTIONS block has been processed in response to the first CONDITIONS block identifying the buffer. In any other CONDITIONS block referring to the buffer, the buffer will not be found unless an additional maintain bit was set.) The maintain bit eventually will be freed automatically whether or not any action is taken on it at the next layer. To hold a buffer at a particular layer, or to continue passing the buffer (in either direction), a new maintain bit must be set. The same maintain bit cannot be used continuously, since it will be freed after the *first* process on it (an ACTION to send, for example).

If you wish to keep a buffer available for your use while also sending it to another layer, set two maintain bits. One will be used to pass the buffer; the other will "maintain" the buffer for other processes. The latter will have to be freed via the *_free_il_msg_buff* routine.

Inputs

The first parameter is the interlayer-buffer number in which the new bit will be set.

The second parameter is a pointer to the location where the returned maintain bit will be stored. There are sixteen maintain bits reserved for each interlayer buffer. Each bit is identified by a two-byte variable with a single zero. The first maintain bit allocated is the least significant, so the value returned is hexadecimal FFFE (binary 11111111 11111110). The last maintain bit allocated is 7FFF (01111111 11111111). If all the maintain bits are already in use, FFFF will be returned.

The maintain bit value should not be modified. It must be used in the freeing operation to make sure the buffer is returned to the free buffer pool.

Example

The variable in which the returned maintain bit will be stored must be declared. When calling the routine, reference the address of this variable. For example, you receive a buffer at Layer 2 from Layer 3 (*up_dl_il_buff*) and insert information into it. Before passing the buffer to Layer 1, set two maintain bits. The one stored in variable *maintain_bit* will hold the buffer for the purpose of repeated resends of the frame, if necessary, and will have to be freed via the *_free_il_msg_buff* routine. When you pass the buffer down, use the bit in variable *l2_relay_baton*. When you resend the frame, set a new *resend_baton* bit and pass that down, still holding *maintain_bit* in reserve for subsequent resends.

```
{
unsigned short l2_relay_baton;
unsigned short resend_baton;
unsigned short maintain_bit;
extern volatile unsigned short up_dl_il_buff;
extern volatile unsigned short up_dl_sdu;
unsigned short l2_data_start_offset;
unsigned short available_space_offset;
static unsigned char l2_data[2] = {0x01, 0x00};
int i;
unsigned char * ptr_l2;
```

```
#define make_ptr(number,offset) ((void *)(((long)number << 16) + offset))
}
LAYER: 3
     STATE: send_fox_message
        CONDITIONS: KEYBOARD " "
        ACTIONS: DL_DATA REQ "DL ET NU((FOX))"
LAYER: 2
     STATE: send_a_buffer
        CONDITIONS: DL_DATA REQ
        ACTIONS:
        {
/* See _insert_il_buff_list_cnt routine for an explanation of how information is inserted in the
buffer. */
         _dup_il_buff_list_start (up_dl_il_buff, up_dl_sdu, &l2_data_start_offset);
         _open_space_in_il_buff(up_dl_il_buff, 2, &available_space_offset);
         ptr_l2 = make_ptr(up_dl_il_buff, available_space_offset);
         for(i = 0; i < 2; i++)
            {
             *ptr_l2 = data_l2[i];
             ptr_l2++;
            }
         ptr_l2 -=2;
         _insert_il_buff_list_cnt(up_dl_il_buff, l2_data_start_offset, ptr_l2, 2);
         _set_maint_buff_bit(up_dl_il_buff, &maintain_bit);
         _set_maint_buff_bit(up_dl_il_buff, &l2_relay_baton);
         send_ph_prmtv_below(up_dl_il_buff, l2_relay_baton, l2_data_start_offset, 0, 0x24, 0);
        }
LAYER: 1
     STATE: resend_buffer
        CONDITIONS:  RECEIVE STRING "[F]03((XXXX1001))"
        ACTIONS:
        {
         _set_maint_buff_bit(up_dl_il_buff, &resend_baton);
         l1_il_transmit(up_dl_il_buff, resend_baton, l2_data_start_offset, 1);
/* See Section 59, Monitor/Transmit Line Data, for an explanation of the l1_il_transmit
routine. */
        }
        CONDITIONS:  RECEIVE STRING "[F][F]((XXXX0001))"
        ACTIONS:
        {
         _free_il_msg_buff(up_dl_il_buff,  maintain_bit);
/* See _free_il_msg_buff for an explanation of this routine. */
        }
```

_insert_il_buff_list_cnt

Synopsis

```
extern unsigned short _insert_il_buff_list_cnt(il_buffer_number, data_start_offset, text_ptr,
   text_length);
unsigned short il_buffer_number;
unsigned short data_start_offset;
unsigned char * text_ptr;
unsigned short text_length;
```

Description

The *_insert_il_buff_list_cnt* routine inserts a text node at the beginning of a linked list of text inside of an interlayer message buffer. It will set the text pointer and byte-count in the text node to the values specified.

Inputs

The first parameter is the interlayer-buffer number in which the linked list will be inserted.

The second parameter is the offset to the header node for the linked list, from the beginning of the buffer.

The third parameter is a pointer to a text.

The fourth parameter is the length of the text.

Returns

If the insert is successful, a value of 0 is returned; if it is not successful, a value of 1 is returned. If you want to check the returned value, do so at the time the routine is called, as in the following example at Layers 2 and 3.

Example

If text is to be copied into the buffer, a pointer to the text must be declared. If not, when calling the *_insert_il_buff_list_cnt* routine, reference the address of the text. The length of the text may be entered as an integer, in which case a *length* variable need not be declared.

Always open space in the buffer if you are going to copy data (usually header information) into the buffer. If you are not going to copy data into the buffer, but reference its location in memory outside the buffer (usually user data), you do not need to open space.

In the following spreadsheet example, an interlayer-buffer number is obtained at Layer 5, a header node is created in the buffer, and the address of a fox message text (located in memory outside of the buffer) is inserted into a text node in the buffer.

```
{
unsigned short il_buffer_number;
unsigned short relay_baton;
unsigned short l4_relay_baton
unsigned short l3_relay_baton;
unsigned short l2_relay_baton;
unsigned short data_start_offset;
unsigned short l2_data_start_offset;
```

```
unsigned short available_space_offset;
static unsigned char data[] = "((FOX))";
static unsigned char l3_data[3] = {0x10, 0x04, 0x00};
static unsigned char l2_data[2] = {0x01, 0x00};
int i;
int length;
extern volatile unsigned short up_t_il_buff;
extern volatile unsigned short up_n_il_buff;
extern volatile unsigned short up_dl_il_buff;
extern volatile unsigned short up_n_sdu;
extern volatile unsigned short up_dl_sdu;
extern volatile unsigned short up_t_sdu;
unsigned char * ptr_l3, * ptr_l2;

/* Whenever make_ptr is encountered, the first parameter will be shifted 16 bits to the left. .
The second parameter will be added, and the result cast into a pointer. */

#define make_ptr(number,offset) ((void *)(((long)number << 16) + offset))
}
LAYER: 5
     STATE: begin_message
        CONDITIONS: KEYBOARD " "
        ACTIONS:
        {
         _get_il_msg_buff(&il_buffer_number, &relay_baton);
         _start_il_buff_list(il_buffer_number, &data_start_offset);

/* Do not include the terminating null character in the length determination of a string. */

         length = sizeof(data) - 1;

/* The address of data outside of the buffer is given for insertion. The data itself is not copied
into the buffer. The buffer is then passed down to Layer 4 (see send_t_prmtv_below for an
explanation of this routine). */
         _insert_il_buff_list_cnt(il_buffer_number, data_start_offset, &data[0], length);
         send_t_prmtv_below(il_buffer_number, relay_baton, data_start_offset, 0, 0x84, 0);
        }
```

At Layer 4 a new maintain bit is set to use in passing the buffer to Layer 3. Since no data is inserted, the same *data_start_offset* is used (in the form of the variable *up_t_sdu*). The buffer is then passed down to Layer 3 (see *send_n_prmtv_below* for an explanation of this routine).

```
LAYER: 4
     STATE: pass
        CONDITIONS:  T_DATA_REQ
        ACTIONS:
        {
         _set_maint_buff_bit(up_t_il_buff, &l4_relay_baton);
         send_n_prmtv_below(up_t_il_buff, l4_relay_baton, up_t_sdu, 0, 0x64, 0);
        }
```

At Layer 3, space is opened for an X.25 packet header. A pointer to the opened space is created and the data is inserted into the linked list passed down from Layer 4.

```
LAYER: 3
    STATE: insert_and_send
        CONDITIONS: N_DATA_REQ
        ACTIONS:
        {
         _open_space_in_il_buff(up_n_il_buff, 3, &available_space_offset);
         ptr_l3 = make_ptr(up_n_il_buff, available_space_offset);
         for(i = 0; i < 3; i++)
            {
             *ptr_l3 = l3_data[i];
             ptr_l3++;
            }
/* The location of the data in the buffer is referenced in the insert routine, so the pointer must
be moved back to the beginning of the opened space. The offset to the Layer 3 header node is
given in the insert routine. If the insertion is not successful, an alarm will sound and a message
will be displayed on the prompt line of the screen. */
         ptr_l3 -=3;
         if(_insert_il_buff_list_cnt(up_n_il_buff, up_n_sdu, ptr_l3, 3) != 0)
            {
             sound_alarm();
             display_prompt("Insert failed at Layer 3.");
            }
/* A new maintain bit is set for passing the buffer. The buffer is then passed down to Layer 2
(see send_dl_prmtv_below for an explanation of this routine). */
         _set_maint_buff_bit(up_n_il_buff, &l3_relay_baton);
         send_dl_prmtv_below(up_n_il_buff, l3_relay_baton, up_n_sdu, 0, 0x44, 0);
        }
```

At Layer 2, a new linked list is started. The Layer 2 header could be inserted into the linked list passed down from Layer 3; but if Layer 3 wants to retain the ability to resend a buffer—that is, to reference again the same list header with the same first-node offset—it must keep its own linked list safe from data inserted at Layer 2.

```
LAYER: 2
    STATE: insert_more
        CONDITIONS: DL_DATA_REQ
        ACTIONS:
        {
/* The _dup_il_buff_list_start routine allows Layer 2 to start its own list. Part of this routine
copies the Layer 3 header into the Layer 2 header node. */
         _dup_il_buff_list_start(up_dl_il_buff, up_dl_sdu, &l2_data_start_offset);
/* Space is opened in the buffer. A pointer to this location is created and the data is copied
into the buffer. */
         _open_space_in_il_buff(up_dl_il_buff, 2, &available_space_offset);
         ptr_l2 = make_ptr(up_dl_il_buff, available_space_offset);
         for(i = 0; i < 2; i++)
            {
             *ptr_l2 = l2_data[i];
             ptr_l2++;
            }
/* The location of the data in the buffer is referenced in the insert routine, so the pointer must
be moved back to the beginning of the opened space. The offset to the Layer 2 header node is
given in the insert routine. If the insertion is not successful, an alarm will sound and a message
will be displayed on the prompt line of the screen. */
```

```
ptr_l2 -=2;
if(_insert_il_buff_list_cnt(up_dl_il_buff, l2_data_start_offset, ptr_l2, 2) != 0)
   {
    sound_alarm();
    pos_cursor(0,30);
    displays("Insert failed at Layer 2.");
   }
/* A new maintain bit is set for passing the buffer. The buffer is then passed down to Layer 1
(see send_ph_prmtv_below for an explanation of this routine). */
 _set_maint_buff_bit(up_dl_il_buff, &l2_relay_baton);
 send_ph_prmtv_below(up_dl_il_buff, l2_relay_baton, l2_data_start_offset, 0, 0x24, 0);
}
```

The following text will be sent out onto the line and displayed as line data:

␁␀␐␃␀THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG 0123456789N🄶

_append_il_buff_list_cnt

Synopsis

```
extern unsigned short _append_il_buff_list_cnt(il_buffer_number, data_start_offset, text_ptr,
   text_length);
unsigned short il_buffer_number;
unsigned short data_start_offset;
unsigned char * text_ptr;
unsigned short text_length;
```

Description

The *_append_il_buff_list_cnt* routine appends a text node at the end of a linked list of text inside of an interlayer message buffer. It will set the text pointer and count in the text node to the information provided.

Inputs

See *_insert_il_buff_list_cnt* routine.

Returns

See *_insert_il_buff_list_cnt* routine.

Example

Two modifications to the program shown for the *_insert_il_buff_list_cnt* routine are all that is required to make the program work for appending data. The changes primarily involve Layer 2 in the example, so we will replicate only that portion of the program below. Substitute *_append_il_buff_list_cnt* for every occurrence *_insert_il_buff_list_cnt*. When data is to be appended in a buffer, you should duplicate the entire linked list received from the layer above, not just the header node. So also substitute *_dup_il_buff_list* for *_dup_il_buff_list_start*.

```
LAYER: 2
    STATE: insert_more
        CONDITIONS: DL_DATA_REQ
        ACTIONS:
        {
        _dup_il_buff_list(up_dl_il_buff, up_dl_sdu, &l2_data_start_offset);
        _open_space_in_il_buff(up_dl_il_buff, 2, &available_space_offset);
        ptr_l2 = make_ptr(up_dl_il_buff, available_space_offset);
        for(i = 0; i < 2; i++)
           {
            *ptr_l2 = l2_data[i];
            ptr_l2++;
           }
        ptr_l2 -=2;
        if(_append_il_buff_list_cnt(up_dl_il_buff, l2_data_start_offset, ptr_l2, 2) != 0)
           {
            sound_alarm();
            pos_cursor(0,30);
            displays("Insert failed at Layer 2.");
           }
        _set_maint_buff_bit(up_dl_il_buff, &l2_relay_baton);
        send_ph_prmtv_below(up_dl_il_buff, l2_relay_baton, l2_data_start_offset, 0, 0x24, 0);
        }
```

The following text will be sent out onto the line and displayed as line data:

THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG 0123456789␐␄␀␁␀▨🄶

(B) Layer 1 OSI Routines

OSI data primitives are handled automatically between Layers 1 and 2. In the "up" direction, line data is placed in an IL buffer and the associated data primitive is given automatically to Layer 2. In the "down" direction, data primitives are received at Layer 1 and put out automatically onto the line.

In the absence of line data, if you want to originate a buffer at Layer 1 and send it upward, use the following routine. In primitives being sent down the layers, Layer 1 will automatically send the primitive out onto the line.

send_ph_to_above

Synopsis

```
extern void send_ph_to_above(il_buffer_number, relay_baton, data_start_offset, size, code,
   path);
unsigned short il_buffer_number;
unsigned short relay_baton;
unsigned short data_start_offset;
unsigned short size;
unsigned char code;
unsigned char path;
```

Description

The *send_ph_to_above* emulate routine passes a specified interlayer message buffer from Layer 1 to Layer 2 in an OSI primitive. Received line data is placed in an IL buffer and passed automatically to Layer 2. If you wish to get a buffer "manually" at Layer 1 and then pass it up, use this routine.

Inputs

The first parameter is the interlayer buffer number returned by the *_get_il_msg_buff* routine.

The second parameter is the returned maintain bit from the *_get_il_msg_buff* routine. As soon as Layer 2 processing on the buffer is completed, the bit is automatically freed.

The third parameter is the returned offset (from the call to *_start_il_buff_list*) to the Layer 1 service data unit in a buffer.

The fourth parameter is the length of the data in the buffer.

The fifth parameter is the code specifying the type of primitive in which the buffer will be sent. Refer to variable *lo_ph_prmtv_code* in Table 63-3 for the appropriate primitive code.

The sixth parameter is the path number along which the buffer will be sent.

Example

Get a buffer at Layer 1. Assuming X.25 protocol, insert data into the buffer and pass it up to Layer 2.

```
{
 unsigned short il_buffer_number;
 unsigned short relay_baton;
 unsigned short data_start_offset;
 unsigned short available_space_offset;
 int length;
 int i;
 static unsigned char data[] = {0x01, 0x00, 0x10, 0x04, 0x00, 0x02, 0x01, 0x01};
 unsigned char * ptr;
}
LAYER: 1
     STATE: get_buffer
         CONDITIONS: KEYBOARD " "
         ACTIONS:
         {
          _get_il_msg_buff(&il_buffer_number, &relay_baton);
          _start_il_buff_list(il_buffer_number, &data_start_offset);
          length = sizeof(data);
          _open_space_in_il_buff(il_buffer_number, length, &available_space_offset);
          ptr = (void *)(((long)il_buffer_number << 16) + available_space_offset);
```

```
for(i = 0; i < length; i++)
   {
    *ptr = data[i];
    ptr++;
   }
ptr-=length;
_insert_il_buff_list_cnt(il_buffer_number, data_start_offset, ptr, length);
send_ph_to_above(il_buffer_number, relay_baton, data_start_offset, length, 0x25, 0);
}
```

(C) Layer 2 OSI Routines

The following routines pass OSI primitives from Layer 2 to either Layer 3 or Layer 1.

send_dl_prmtv_above

Synopsis

```
extern void send_dl_prmtv_above(il_buffer_number, l2_relay_baton, l2_data_start_offset, size,
   l2_code, path);
unsigned short il_buffer_number;
unsigned short l2_relay_baton;
unsigned short l2_data_start_offset;
unsigned short size;
unsigned char l2_code;
unsigned char path;
```

Description

The *send_dl_prmtv_above* emulate routine passes a specified interlayer message buffer from Layer 2 to Layer 3 in an OSI primitive.

Inputs

The first parameter is the interlayer buffer number to be sent. For a buffer which has been received at Layer 2 from Layer 1, the variable *lo_ph_il_buff* may be used to identify the buffer number.

The second parameter is the returned maintain bit from a call to *_set_maint_buff_bit*. It is used only to pass a received buffer from Layer 2 to Layer 3. As soon as Layer 3 processing on the buffer is completed, the bit is automatically freed.

The third parameter is the offset to the Layer 2 service data unit in a received buffer. The variable *lo_ph_sdu* contains the offset to the service data unit when the buffer reached Layer 2. The offset must be incremented by the length of the Layer 2 header.

> **NOTE:** In general, do not modify *extern* variables, such as *lo_ph_sdu*, which may be updated by other processes. Name another variable, assign it the same value, and then increment that variable. Or, after *lo_ph_sdu* has been named in the argument of the *send* routine, add the length of the Layer 2 header, as in the example below.

The fourth parameter is the length of the data in the buffer. Use the length indicated in the *pdu* structure–*pdu.data_length*. Then subtract the length of the Layer 2 header.

The fifth parameter is the code specifying the type of primitive in which the buffer will be sent. Refer to variable *lo_dl_prmtv_code* in Table 63-4 for the appropriate primitive code.

The sixth parameter is the path number along which the buffer will be sent. For a buffer which has been received at Layer 2 from Layer 1, the variable *lo_ph_prmtv_path* may be used to specify the path number.

Example

A buffer is received at Layer 2 from Layer 1. Assuming X.25 protocol, the data specific to Layer 2 (the frame header) begins at the SDU offset (*lo_ph_sdu*) and consists of two bytes. Before the buffer is passed up to Layer 3, the offset to the SDU and the size of the SDU will be adjusted by two bytes and a new maintain bit will be set.

```
{
 struct pdu
  {
   unsigned char primitive_code;
   unsigned char path;
   unsigned long parameter;
   unsigned short relay_baton;
   unsigned short il_buffer_number;
   unsigned char buffer_contents;
   unsigned short data_start_offset;
   unsigned short data_length;
  };
 struct pdu * pdu_ptr;
 extern volatile unsigned short lo_ph_pdu_seg;
 extern volatile const unsigned char lo_ph_prmtv_path;
 extern volatile unsigned short lo_ph_il_buff;
 extern volatile unsigned short lo_ph_sdu;
 unsigned short l2_relay_baton;
}
```

```
LAYER: 2
     STATE: send_buffer_up
        CONDITIONS: PH_DATA IND
        ACTIONS:
        {
         pdu_ptr = (void *)((long)lo_ph_pdu_seg << 16);
         _set_maint_buff_bit(lo_ph_il_buff, &l2_relay_baton);
         send_dl_prmtv_above(lo_ph_il_buff, l2_relay_baton, lo_ph_sdu + 2,
              pdu_ptr->data_length - 2, 0x45, lo_ph_prmtv_path);
        }
```

send_m_dl_prmtv_above

Synopsis

```
extern void send_m_dl_prmtv_above(il_buffer_number, l2_relay_baton, l2_data_start_offset,
   size, l2_code, path);
unsigned short il_buffer_number;
unsigned short l2_relay_baton;
unsigned short l2_data_start_offset;
unsigned short size;
unsigned char l2_code;
unsigned char path;
```

Description

The *send_m_dl_prmtv_above* monitor routine passes a specified interlayer message buffer from Layer 2 to Layer 3 in an OSI monitor primitive.

Inputs

See *send_dl_prmtv_above*. Use the monitor variables *m_lo_ph_il_buff, m_lo_ph_sdu_offset*, and *m_lo_ph_sdu_size* as input. Refer to variable *m_lo_dl_prmtv_code* in Table 63-4 for the appropriate primitive code.

Example

Make the appropriate variable declarations. For a condition monitoring RD data primitives, the Layer 2 programming block should look like this:

```
LAYER: 2
     STATE: send_buffer_up
        CONDITIONS: PH_RD_DATA IND
        ACTIONS:
        {
         _set_maint_buff_bit(m_lo_ph_il_buff, &l2_relay_baton);
         send_m_dl_prmtv_above(m_lo_ph_il_buff, l2_relay_baton,m_ lo_ph_sdu_offset + 2,
              m_lo_ph_sdu_size - 2, 0x45, m_lo_ph_prmtv_path);
        }
```

send_ph_prmtv_below

Synopsis

extern void send_ph_prmtv_below(il_buffer_number, l2_relay_baton, l2_data_start_offset, size, l2_code, path);
unsigned short il_buffer_number;
unsigned short l2_relay_baton;
unsigned short l2_data_start_offset;
unsigned short size;
unsigned char l2_code;
unsigned char path;

Description

The *send_ph_prmtv_below* emulate routine passes a specified interlayer message buffer from Layer 2 to Layer 1 in an OSI primitive.

Inputs

The first parameter is the interlayer buffer number to be sent. For a buffer which has been received at Layer 2 from Layer 3, the variable *up_dl_il_buff* may be used to identify the buffer number. If the buffer originated at Layer 2, use the buffer-number variable named in the *_get_il_msg_buff* routine. (See *_insert_il_buff_list_cnt* routine example at Layer 5.)

The second parameter is the returned maintain bit from a call to *_set_maint_buff_bit*. It is used only to pass a received buffer from Layer 2 to Layer 1. As soon as Layer 1 processing on the buffer is completed, the bit is automatically freed. If the buffer originated at Layer 2, use the maintain bit variable named in the *_get_il_msg_buff* routine. (See *_insert_il_buff_list_cnt* routine example at Layer 5.)

The third parameter is the offset to the Layer 2 list header node in the buffer. For a buffer which has been received at Layer 2 from Layer 3, the variable *up_dl_sdu* may be used to indicate the offset.

The fourth parameter is the size of the data in the buffer. It will always be set to zero since the data length is unknown in a primitive being passed down the layers.

The fifth parameter is the code specifying the type of primitive in which the buffer will be sent. Refer to variable *ph_prmtv_type* in Table 63-2 for the appropriate primitive code.

The sixth parameter is the path number along which the buffer will be sent. For a buffer which has been received at Layer 2 from Layer 3, the variable *up_dl_prmtv_path* may be used to specify the path number.

Example

A buffer is received at Layer 2 from Layer 3. No text will be inserted at Layer 2. (For information on inserting text, see *_insert_il_buff_list_cnt* routine.) The buffer will be passed to Layer 1, requiring a new maintain bit to be set. If values are entered for the code and path, variables for code and path need not be declared.

```
{
 extern volatile unsigned short up_dl_il_buff;
 extern volatile unsigned short up_dl_sdu;
 unsigned short l2_relay_baton;
}
LAYER: 2
     STATE: pass_buffer_down
        CONDITIONS: DL_DATA REQ
        ACTIONS:
        {
         _set_maint_buff_bit(up_dl_il_buff, &l2_relay_baton);
         send_ph_prmtv_below(up_dl_il_buff, l2_relay_baton, up_dl_sdu, 0, 0x24, 0);
        }
```

(D) Layer 3 OSI Routines

The following routines pass OSI primitives from Layer 3 to either Layer 4 or Layer 2.

send_n_prmtv_above

Synopsis

```
extern void send_n_prmtv_above(il_buffer_number, l3_relay_baton, l3_data_start_offset, size,
   l3_code, path);
unsigned short il_buffer_number;
unsigned short l3_relay_baton;
unsigned short l3_data_start_offset;
unsigned short size;
unsigned char l3_code;
unsigned char path;
```

Description

The *send_n_prmtv_above* emulate routine passes a specified interlayer message buffer from Layer 3 to Layer 4 in an OSI primitive.

Inputs

The first parameter is the interlayer buffer number to be sent. For a buffer which has been received at Layer 3 from Layer 2, the variable *lo_dl_il_buff* may be used to identify the buffer number.

The second parameter is the returned maintain bit from a call to *_set_maint_buff_bit*. It is used only to pass a received buffer from Layer 3 to Layer 4. As soon as Layer 4 processing on the buffer is completed, the bit is automatically freed.

The third parameter is the offset to the Layer 3 service data unit in a received buffer. The variable *lo_dl_sdu* contains the offset to the service data unit when the buffer reached Layer 3. The offset must be incremented by the length of the Layer 3 header.

> **NOTE:** In general, do not modify *extern* variables, such as *lo_dl_sdu*, which may be updated by other processes. Name another variable, assign it the same value, and then increment that variable. Or, after *lo_dl_sdu* has been named in the argument of the *send* routine, add the length of the Layer 3 header, as in the example below.

The fourth parameter is the length of the data in the buffer. Use the length indicated in the *pdu* structure—*pdu.data_length*. Then subtract the length of the Layer 3 header.

The fifth parameter is the code specifying the type of primitive in which the buffer will be sent. Refer to variable *lo_n_prmtv_code* in Table 63-5 for the appropriate primitive code.

The sixth parameter is the path number along which the buffer will be sent. For a buffer which has been received at Layer 3 from Layer 2, the variable *lo_dl_prmtv_path* may be used to specify the path number.

<u>Example</u>

A buffer is received at Layer 3 from Layer 2. Assuming X.25 protocol, the header consists of three bytes. The offset to and size of the service data unit will be adjusted by three bytes, a new maintain bit will be set, and the buffer will be passed up to Layer 4.

```
{
 struct pdu
  {
   unsigned char primitive_code;
   unsigned char path;
   unsigned long parameter;
   unsigned short relay_baton;
   unsigned short il_buffer_number;
   unsigned char buffer_contents;
   unsigned short data_start_offset;
   unsigned short data_length;
  };
```

```
 struct pdu * pdu_ptr;
 extern volatile unsigned short lo_dl_pdu_seg;
 extern volatile const unsigned char lo_dl_prmtv_path;
 extern volatile unsigned short lo_dl_il_buff;
 extern volatile unsigned short lo_dl_sdu;
 unsigned short l3_relay_baton;
}
LAYER: 3
     STATE: send_buffer_up
         CONDITIONS: DL_DATA IND
         ACTIONS:
         {
          pdu_ptr = (void *)((long)lo_dl_pdu_seg << 16);
          _set_maint_buff_bit(lo_dl_il_buff, &l3_relay_baton);
          send_n_prmtv_above(lo_dl_il_buff, l3_relay_baton, lo_dl_sdu + 3,
                pdu_ptr->data_length - 3, 0x65, lo_dl_prmtv_path);
         }
```

send_m_n_prmtv_above

Synopsis

```
extern void send_m_n_prmtv_above(il_buffer_number, l3_relay_baton, l3_data_start_offset,
   size, l3_code, path);
unsigned short il_buffer_number;
unsigned short l3_relay_baton;
unsigned short l3_data_start_offset;
unsigned short size;
unsigned char l3_code;
unsigned char path;
```

Description

The *send_m_n_prmtv_above* monitor routine passes a specified interlayer message buffer from Layer 3 to Layer 4 in an OSI monitor primitive.

Inputs

See *send_n_prmtv_above*. Use the monitor variables *m_lo_dl_il_buff*, *m_lo_dl_sdu_offset*, and *m_lo_dl_sdu_size* as input. Refer to variable *m_lo_n_prmtv_code* in Table 63-5 for the appropriate primitive code.

Example

Make the appropriate variable declarations. For a condition monitoring RD data primitives, the Layer 3 programming block should look like this:

```
LAYER: 3
     STATE: send_buffer_up
         CONDITIONS: DL_RD_DATA IND
         ACTIONS:
         {
          _set_maint_buff_bit(m_lo_dl_il_buff, &l3_relay_baton);
          send_m_n_prmtv_above(m_lo_dl_il_buff, l3_relay_baton, m_lo_dl_sdu_offset + 3,
                m_lo_dl_sdu_size - 3, 0x65, m_lo_dl_prmtv_path);
         }
```

send_dl_prmtv_below

Synopsis

```
extern void send_dl_prmtv_below(il_buffer_number, l3_relay_baton, l3_data_start_offset, size,
   l3_code, path);
unsigned short il_buffer_number;
unsigned short l3_relay_baton;
unsigned short l3_data_start_offset;
unsigned short size;
unsigned char l3_code;
unsigned char path;
```

Description

The *send_dl_prmtv_below* emulate routine passes a specified interlayer message buffer from Layer 3 to Layer 2 in an OSI primitive.

Inputs

The first parameter is the interlayer buffer number to be sent. For a buffer which has been received at Layer 3 from Layer 4, the variable *up_n_il_buff* may be used to identify the buffer number. If the buffer originated at Layer 3, use the buffer-number variable named in the *_get_il_msg_buff* routine. (See *_insert_il_buff_list_cnt* routine example at Layer 5.)

The second parameter is the returned maintain bit from a call to *_set_maint_buff_bit*. It is used only to pass a received buffer from Layer 3 to Layer 2. As soon as Layer 2 processing on the buffer is completed, the bit is automatically freed. If the buffer originated at Layer 3, use the maintain bit variable named in the *_get_il_msg_buff* routine. (See *_insert_il_buff_list_cnt* routine example at Layer 5.)

The third parameter is the offset to the Layer 3 list header node in the buffer. For a buffer which has been received at Layer 3 from Layer 4, the variable *up_n_sdu* may be used to indicate the offset.

The fourth parameter is the size of the data in the buffer. It will always be set to zero since the data length is unknown in a primitive being passed down the layers.

The fifth parameter is the code specifying the type of primitive in which the buffer will be sent. Refer to variable *up_dl_prmtv_code* in Table 63-3 for the appropriate primitive code.

The sixth parameter is the path number along which the buffer will be sent. For a buffer which has been received at Layer 3 from Layer 4, the variable *up_n_prmtv_path* may be used to specify the path number.

Example

A buffer is received at Layer 3 from Layer 4. No text will be inserted at Layer 3. (For information on inserting text, see *_insert_il_buff_list_cnt* routine.) The buffer will be passed to Layer 2, requiring a new maintain bit to be set. If values are entered for the code and path, these variables need not be declared.

```
{
 extern volatile unsigned short up_n_il_buff;
 extern volatile unsigned short up_n_sdu;
 unsigned short l3_relay_baton;
}
LAYER: 3
     STATE: pass_buffer_down
        CONDITIONS: N_DATA REQ
        ACTIONS:
        {
         _set_maint_buff_bit(up_n_il_buff, &l3_relay_baton);
         send_dl_prmtv_below(up_n_il_buff, l3_relay_baton, up_n_sdu, 0, 0x44, 0);
        }
```

(E) Layer 4 OSI Routines

The following routines pass OSI primitives from Layer 4 to either Layer 5 or Layer 3.

send_t_prmtv_above

Synopsis

```
extern void send_t_prmtv_above(il_buffer_number, l4_relay_baton, l4_data_start_offset, size,
   l4_code, path);
unsigned short il_buffer_number;
unsigned short l4_relay_baton;
unsigned short l4_data_start_offset;
unsigned short size;
unsigned char l4_code;
unsigned char path;
```

Description

The *send_t_prmtv_above* emulate routine passes a specified interlayer message buffer from Layer 4 to Layer 5 in an OSI primitive.

Inputs

The first parameter is the interlayer buffer number to be sent. For a buffer which has been received at Layer 4 from Layer 3, the variable *lo_n_il_buff* may be used to identify the buffer number.

The second parameter is the returned maintain bit from a call to *_set_maint_buff_bit*. It is used only to pass a received buffer from Layer 4 to Layer 5. As soon as Layer 5 processing on the buffer is completed, the bit is automatically freed.

The third parameter is the offset to the Layer 4 service data unit in a received buffer. The variable *lo_n_sdu* contains the offset to the service data unit when the buffer reached Layer 4. The offset must be incremented by the length of the Layer 4 header, if any.

> **NOTE:** In general, do not modify *extern* variables, such as *lo_n_sdu*, which may be updated by other processes. Name another variable, assign it the same value, and then increment that variable. Or, after *lo_n_sdu* has been named in the argument of the *send* routine, add the length of the Layer 4 header, if any.

The fourth parameter is the length of the data in the buffer. Use the length indicated in the *pdu* structure—*pdu.data_length*. Then subtract the length of the Layer 4 header, if any.

The fifth parameter is the code specifying the type of primitive in which the buffer will be sent. Refer to variable *lo_t_prmtv_code* in Table 63-6 for the appropriate primitive code.

The sixth parameter is the path number along which the buffer will be sent. For a buffer which has been received at Layer 4 from Layer 3, the variable *lo_n_prmtv_path* may be used to specify the path number.

Example

A buffer is received at Layer 4 from Layer 3. The offset to and size of the service data unit will be adjusted if needed, a new maintain bit will be set, and the buffer will be passed up to Layer 5.

```
{
 struct pdu
  {
   unsigned char primitive_code;
   unsigned char path;
   unsigned long parameter;
   unsigned short relay_baton;
   unsigned short il_buffer_number;
   unsigned char buffer_contents;
   unsigned short data_start_offset;
   unsigned short data_length;
  };
 struct pdu * pdu_ptr;
 extern volatile unsigned short lo_n_pdu_seg;
 extern volatile const unsigned char lo_n_prmtv_path;
```

```
 extern volatile unsigned short lo_n_il_buff;
 extern volatile unsigned short lo_n_sdu;
 unsigned short l4_relay_baton;
}
LAYER: 4
     STATE: send_buffer_up
        CONDITIONS: N_DATA IND
        ACTIONS:
        {
         pdu_ptr = (void *)((long)lo_n_pdu_seg << 16);
         _set_maint_buff_bit(lo_n_il_buff, &l4_relay_baton);
         send_t_prmtv_above(lo_n_il_buff, l4_relay_baton, lo_n_sdu,pdu_ptr->data_length,
              0x85, lo_n_prmtv_path);
        }
```

send_m_t_prmtv_above

Synopsis

```
extern void send_m_t_prmtv_above(il_buffer_number, l4_relay_baton, l4_data_start_offset,
   size, l4_code, path);
unsigned short il_buffer_number;
unsigned short l4_relay_baton;
unsigned short l4_data_start_offset;
unsigned short size;
unsigned char l4_code;
unsigned char path;
```

Description

The *send_m_t_prmtv_above* monitor routine passes a specified interlayer message buffer from Layer 4 to Layer 5 in an OSI monitor primitive.

Inputs

See *send_t_prmtv_above*. Use the monitor variables *m_lo_n_il_buff, m_lo_n_sdu_offset,* and *m_lo_n_sdu_size* as input. Refer to variable *m_lo_t_prmtv_code* in Table 63-6 for the appropriate primitive code.

Example

Make the appropriate variable declarations. For a condition monitoring RD data primitives, the Layer 4 programming block should look like this:

```
LAYER: 4
     STATE: send_buffer_up
        CONDITIONS: N_RD_DATA IND
        ACTIONS:
        {
         _set_maint_buff_bit(m_lo_n_il_buff, &l4_relay_baton);
         send_m_t_prmtv_above(m_lo_n_il_buff, l4_relay_baton,m_ lo_n_sdu_offset ,
              m_lo_n_sdu_size, 0x85, m_lo_n_prmtv_path);
        }
```

send_n_prmtv_below

Synopsis

```
extern void send_n_prmtv_below(il_buffer_number, l4_relay_baton, l4_data_start_offset, size,
  l4_code, path);
unsigned short il_buffer_number;
unsigned short l4_relay_baton;
unsigned short l4_data_start_offset;
unsigned short size;
unsigned char l4_code;
unsigned char path;
```

Description

The *send_n_prmtv_below* emulate routine passes a specified interlayer message buffer from Layer 4 to Layer 3 in an OSI primitive.

Inputs

The first parameter is the interlayer buffer number to be sent. For a buffer which has been received at Layer 4 from Layer 5, the variable *up_t_il_buff* may be used to identify the buffer number. If the buffer originated at Layer 4, use the buffer-number variable named in the *_get_il_msg_buff* routine. (See *_insert_il_buff_list_cnt* routine example at Layer 5.)

The second parameter is the returned maintain bit from a call to *_set_maint_buff_bit*. It is used only to pass a received buffer from Layer 4 to Layer 3. As soon as Layer 3 processing on the buffer is completed, the bit is automatically freed. If the buffer originated at Layer 4, use the maintain bit variable named in the *_get_il_msg_buff* routine. (See *_insert_il_buff_list_cnt* routine example at Layer 5.)

The third parameter is the offset to the Layer 4 list header node in the buffer. For a buffer which has been received at Layer 4 from Layer 5, the variable *up_t_sdu* may be used to indicate the offset.

The fourth parameter is the size of the data in the buffer. It will always be set to zero since the data length is unknown in a primitive being passed down the layers.

The fifth parameter is the code specifying the type of primitive in which the buffer will be sent. Refer to variable *up_n_prmtv_code* in Table 63-4 for the appropriate primitive code.

The sixth parameter is the path number along which the buffer will be sent. For a buffer which has been received at Layer 4 from Layer 5, the variable *up_t_prmtv_path* may be used to specify the path number.

Example

A buffer is received at Layer 4 from Layer 5. No text will be inserted at Layer 4. (For information on inserting text, see *_insert_il_buff_list_cnt* routine.) The buffer will be passed to Layer 3, requiring a new maintain bit to be set. If values are entered for the code and path, variables for code and path need not be declared.

```
{
 extern volatile unsigned short up_t_il_buff;
 extern volatile unsigned short up_t_sdu;
 unsigned short l4_relay_baton;
}
LAYER: 4
     STATE: pass_buffer_down
        CONDITIONS: T_DATA REQ
        ACTIONS:
        {
         _set_maint_buff_bit(up_t_il_buff, &l4_relay_baton);
         send_n_prmtv_below(up_t_il_buff, l4_relay_baton, up_t_sdu, 0, 0x64, 0);
        }
```

(F) Layer 5 OSI Routines

The following routines pass OSI primitives from Layer 5 to either Layer 6 or Layer 4.

send_s_prmtv_above

Synopsis

```
extern void send_s_prmtv_above(il_buffer_number, l5_relay_baton, l5_data_start_offset, size,
   l5_code, path);
unsigned short il_buffer_number;
unsigned short l5_relay_baton;
unsigned short l5_data_start_offset;
unsigned short size;
unsigned char l5_code;
unsigned char path;
```

Description

The *send_s_prmtv_above* emulate routine passes a specified inter-layer message buffer from Layer 5 to Layer 6 in an OSI primitive.

Inputs

The first parameter is the inter-layer buffer number to be sent. For a buffer which has been received at Layer 5 from Layer 4, the variable *lo_t_il_buff* may be used to identify the buffer number.

The second parameter is the returned maintain bit from a call to *_set_maint_buff_bit*. It is used only to pass a received buffer from Layer 5 to Layer 6. As soon as Layer 6 processing on the buffer is completed, the bit is automatically freed.

The third parameter is the offset to the Layer 5 service data unit in a received buffer. The variable *lo_t_sdu* contains the offset to the service data unit when the buffer reached Layer 5. The offset must be incremented by the length of the Layer 5 header, if any.

> **NOTE:** In general, do not modify *extern* variables, such as *lo_t_sdu*, which may be updated by other processes. Name another variable, assign it the same value, and then increment that variable. Or, after *lo_t_sdu* has been named in the argument of the *send* routine, add the length of the Layer 5 header, if any.

The fourth parameter is the length of the data in the buffer. Use the length indicated in the *pdu* structure—*pdu.data_length*. Then subtract the length of the Layer 5 header, if any.

The fifth parameter is the code specifying the type of primitive in which the buffer will be sent. Refer to variable *lo_s_prmtv_code* in Table 63-7 for the appropriate primitive code.

The sixth parameter is the path number along which the buffer will be sent. For a buffer which has been received at Layer 5 from Layer 4, the variable *lo_t_prmtv_path* may be used to specify the path number.

Example

A buffer is received at Layer 5 from Layer 4. The offset to and size of the service data unit will be adjusted if needed, a new maintain bit will be set, and the buffer will be passed up to Layer 6.

```
{
 struct pdu
  {
   unsigned char primitive_code;
   unsigned char path;
   unsigned long parameter;
   unsigned short relay_baton;
   unsigned short il_buffer_number;
   unsigned char buffer_contents;
   unsigned short data_start_offset;
   unsigned short data_length;
  };
 struct pdu * pdu_ptr;
 extern volatile unsigned short lo_t_pdu_seg;
 extern volatile const unsigned char lo_t_prmtv_path;
 extern volatile unsigned short lo_t_il_buff;
 extern volatile unsigned short lo_t_sdu;
 unsigned short l5_relay_baton;
}
```

```
LAYER: 5
    STATE: send_buffer_up
        CONDITIONS: T_DATA IND
        ACTIONS:
        {
         pdu_ptr = (void *)((long)lo_t_pdu_seg << 16);
         _set_maint_buff_bit(lo_t_il_buff, &l5_relay_baton);
         send_s_prmtv_above(lo_t_il_buff, l5_relay_baton, lo_t_sdu, pdu_ptr->data_length,
              Oxa5, lo_t_prmtv_path);
        }
```

send_m_s_prmtv_above

Synopsis

```
extern void send_m_s_prmtv_above(il_buffer_number, l5_relay_baton, l5_data_start_offset,
   size, l5_code, path);
unsigned short il_buffer_number;
unsigned short l5_relay_baton;
unsigned short l5_data_start_offset;
unsigned short size;
unsigned char l5_code;
unsigned char path;
```

Description

The *send_m_s_prmtv_above* monitor routine passes a specified inter-layer message buffer from Layer 5 to Layer 6 in an OSI monitor primitive.

Inputs

See *send_s_prmtv_above*. Use the monitor *m_lo_t_il_buff*, *m_lo_t_sdu_offset*, and *m_lo_t_sdu_size* variables as input. Refer to variable *m_lo_s_prmtv_code* in Table 63-7 for the appropriate primitive code.

Example

Make the appropriate variable declarations. For a condition monitoring RD data primitives, the Layer 5 programming block should look like this:

```
LAYER: 5
    STATE: send_buffer_up
        CONDITIONS: T_RD_DATA IND
        ACTIONS:
        {
         _set_maint_buff_bit(m_lo_t_il_buff, &l5_relay_baton);
         send_m_s_prmtv_above(m_lo_t_il_buff, l5_relay_baton,m_ lo_t_sdu_offset,
              m_lo_t_sdu_size, Oxa5, m_lo_t_prmtv_path);
        }
```

send_t_prmtv_below

Synopsis

```
extern void send_t_prmtv_below(il_buffer_number, l5_relay_baton, l5_data_start_offset, size,
   l5_code, path);
unsigned short il_buffer_number;
unsigned short l5_relay_baton;
unsigned short l5_data_start_offset;
unsigned short size;
unsigned char l5_code;
unsigned char path;
```

Description

The *send_t_prmtv_below* emulate routine passes a specified inter-layer message buffer from Layer 5 to Layer 4 in an OSI primitive.

Inputs

The first parameter is the inter-layer buffer number to be sent. For a buffer which has been received at Layer 5 from Layer 6, the variable *up_s_il_buff* may be used to identify the buffer number. If the buffer originated at Layer 5, use the buffer-number variable named in the *_get_il_msg_buff* routine. (See *_insert_il_buff_list_cnt* routine example at Layer 5.)

The second parameter is the returned maintain bit from a call to *_set_maint_buff_bit*. It is used only to pass a received buffer from Layer 5 to Layer 4. As soon as Layer 4 processing on the buffer is completed, the bit is automatically freed. If the buffer originated at Layer 5, use the maintain bit variable named in the *_get_il_msg_buff* routine. (See *_insert_il_buff_list_cnt* routine example at Layer 5.)

The third parameter is the offset to the Layer 5 list header node in the buffer. For a buffer which has been received at Layer 5 from Layer 6, the variable *up_s_sdu* may be used to indicate the offset.

The fourth parameter is the size of the data in the buffer. It will always be set to zero since the data length is unknown in a primitive being passed down the layers.

The fifth parameter is the code specifying the type of primitive in which the buffer will be sent. Refer to variable *up_t_prmtv_code* in Table 63-5 for the appropriate primitive code.

The sixth parameter is the path number along which the buffer will be sent. For a buffer which has been received at Layer 5 from Layer 6, the variable *up_s_prmtv_path* may be used to specify the path number.

Example

A buffer is received at Layer 5 from Layer 6. No text will be inserted at Layer 5. (For information on inserting text, see *_insert_il_buff_list_cnt* routine.) The buffer will be passed to Layer 4, requiring a new maintain bit to be set. If values are entered for the code and path, variables for code and path need not be declared.

```
{
 extern volatile unsigned short up_s_il_buff;
 extern volatile unsigned short up_s_sdu;
 unsigned short l5_relay_baton;
}
LAYER: 5
     STATE: pass_buffer_down
        CONDITIONS: S_DATA REQ
        ACTIONS:
        {
         _set_maint_buff_bit(up_s_il_buff, &l5_relay_baton);
         send_t_prmtv_below(up_s_il_buff, l5_relay_baton, up_s_sdu, 0, 0x84, 0);
        }
```

(G) Layer 6 OSI Routines

The following routines pass OSI primitives from Layer 6 to either Layer 7 or Layer 5.

send_p_prmtv_above

Synopsis

```
extern void send_p_prmtv_above(il_buffer_number, l6_relay_baton, l6_data_start_offset, size,
   l6_code, path);
unsigned short il_buffer_number;
unsigned short l6_relay_baton;
unsigned short l6_data_start_offset;
unsigned short size;
unsigned char l6_code;
unsigned char path;
```

Description

The *send_p_prmtv_above* emulate routine passes a specified interlayer message buffer from Layer 6 to Layer 7 in an OSI primitive.

Inputs

The first parameter is the interlayer buffer number to be sent. For a buffer which has been received at Layer 6 from Layer 5, the variable *lo_s_il_buff* may be used to identify the buffer number.

The second parameter is the returned maintain bit from a call to *_set_maint_buff_bit*. It is used only to pass a received buffer from Layer 6 to Layer 7. As soon as Layer 7 processing on the buffer is completed, the bit is automatically freed.

The third parameter is the offset to the Layer 6 service data unit in a received buffer. The variable *lo_s_sdu* contains the offset to the service data unit when the buffer reached Layer 6. The offset must be incremented by the length of the Layer 6 header, if any.

> **NOTE:** In general, do not modify *extern* variables, such as *lo_s_sdu*, which may be updated by other processes. Name another variable, assign it the same value, and then increment that variable. Or, after *lo_s_sdu* has been named in the argument of the *send* routine, add the length of the Layer 6 header, if any.

The fourth parameter is the length of the data in the buffer. Use the length indicated in the *pdu* structure—*pdu.data_length*. Then subtract the length of the Layer 6 header, if any.

The fifth parameter is the code specifying the type of primitive in which the buffer will be sent. Refer to variable *lo_p_prmtv_code* in Table 63-8 for the appropriate primitive code.

The sixth parameter is the path number along which the buffer will be sent. For a buffer which has been received at Layer 6 from Layer 5, the variable *lo_s_prmtv_path* may be used to specify the path number.

Example

A buffer is received at Layer 6 from Layer 5. The offset to and size of the service data unit will be adjusted if needed, a new maintain bit will be set, and the buffer will be passed up to Layer 7.

```
{
 struct pdu
  {
   unsigned char primitive_code;
   unsigned char path;
   unsigned long parameter;
   unsigned short relay_baton;
   unsigned short il_buffer_number;
   unsigned char buffer_contents;
   unsigned short data_start_offset;
   unsigned short data_length;
  };
 struct pdu * pdu_ptr;
 extern volatile unsigned short lo_s_pdu_seg;
 extern volatile const unsigned char lo_s_prmtv_path;
```

```
 extern volatile unsigned short lo_s_il_buff;
 extern volatile unsigned short lo_s_sdu;
 unsigned short l6_relay_baton;
}
LAYER: 6
     STATE: send_buffer_up
        CONDITIONS: S_DATA IND
        ACTIONS:
        {
         pdu_ptr = (void *)((long)lo_s_pdu_seg << 16);
         _set_maint_buff_bit(lo_s_il_buff, &l6_relay_baton);
         send_p_prmtv_above(lo_s_il_buff, l6_relay_baton, lo_s_sdu, pdu_ptr->data_length,
              0xc5, lo_s_prmtv_path);
        }
```

send_m_p_prmtv_above

Synopsis

```
extern void send_m_p_prmtv_above(il_buffer_number, l6_relay_baton, l6_data_start_offset,
   size, l6_code, path);
unsigned short il_buffer_number;
unsigned short l6_relay_baton;
unsigned short l6_data_start_offset;
unsigned short size;
unsigned char l6_code;
unsigned char path;
```

Description

The *send_m_p_prmtv_above* monitor routine passes a specified interlayer message buffer from Layer 6 to Layer 7 in an OSI monitor primitive.

Inputs

See *send_p_prmtv_above*. Use the monitor variables *m_lo_s_il_buff*, *m_lo_s_sdu_offset*, and *m_lo_s_sdu_size* as input. Refer to variable *m_lo_p_prmtv_code* in Table 63-8 for the appropriate primitive code.

Example

Make the appropriate variable declarations. For a condition monitoring RD data primitives, the Layer 6 programming block should look like this:

```
LAYER: 6
     STATE: send_buffer_up
        CONDITIONS: S_RD_DATA IND
        ACTIONS:
        {
         _set_maint_buff_bit(m_lo_s_il_buff, &l6_relay_baton);
         send_m_p_prmtv_above(m_lo_s_il_buff, l6_relay_baton,m_ lo_s_sdu_offset,
              m_lo_s_sdu_size, 0xc5, m_lo_s_prmtv_path);
        }
```

send_s_prmtv_below

Synopsis

extern void send_s_prmtv_below(il_buffer_number, l6_relay_baton, l6_data_start_offset, size, l6_code, path);
unsigned short il_buffer_number;
unsigned short l6_relay_baton;
unsigned short l6_data_start_offset;
unsigned short size;
unsigned char l6_code;
unsigned char path;

Description

The *send_s_prmtv_below* emulate routine passes a specified interlayer message buffer from Layer 6 to Layer 5 in an OSI primitive.

Inputs

The first parameter is the interlayer buffer number to be sent. For a buffer which has been received at Layer 6 from Layer 7, the variable *up_p_il_buff* may be used to identify the buffer number. If the buffer originated at Layer 6, use the buffer-number variable named in the *_get_il_msg_buff* routine. (See *_insert_il_buff_list_cnt* routine example at Layer 5.)

The second parameter is the returned maintain bit from a call to *_set_maint_buff_bit*. It is used only to pass a received buffer from Layer 6 to Layer 5. As soon as Layer 5 processing on the buffer is completed, the bit is automatically freed. If the buffer originated at Layer 6, use the maintain bit variable named in the *_get_il_msg_buff* routine. (See *_insert_il_buff_list_cnt* routine example at Layer 5.)

The third parameter is the offset to the Layer 6 list header node in the buffer. For a buffer which has been received at Layer 6 from Layer 7, the variable *up_p_sdu* may be used to indicate the offset.

The fourth parameter is the size of the data in the buffer. It will always be set to zero since the data length is unknown in a primitive being passed down the layers.

The fifth parameter is the code specifying the type of primitive in which the buffer will be sent. Refer to variable *up_s_prmtv_code* in Table 63-6 for the appropriate primitive code.

The sixth parameter is the path number along which the buffer will be sent. For a buffer which has been received at Layer 6 from Layer 7, the variable *up_p_prmtv_path* may be used to specify the path number.

Example

A buffer is received at Layer 6 from Layer 7. No text will be inserted at Layer 6. (For information on inserting text, see *_insert_il_buff_list_cnt* routine.) The buffer will be passed to Layer 5, requiring a new maintain bit to be set. If values are entered for the code and path, variables for code and path need not be declared.

```
{
 extern volatile unsigned short up_p_il_buff;
 extern volatile unsigned short up_p_sdu;
 unsigned short l6_relay_baton;
}
LAYER: 6
     STATE: pass_buffer_down
        CONDITIONS: P_DATA REQ
        ACTIONS:
        {
         _set_maint_buff_bit(up_p_il_buff, &l6_relay_baton);
         send_s_prmtv_below(up_p_il_buff, l6_relay_baton, up_p_sdu, 0, 0xa4, 0);
        }
```

(H) Layer 7 OSI Routines

send_p_prmtv_below

Synopsis

```
extern void send_p_prmtv_below(il_buffer_number, relay_baton, data_start_offset, size, code,
   path);
 unsigned short il_buffer_number;
unsigned short relay_baton;
unsigned short data_start_offset;
unsigned short size;
unsigned char code;
unsigned char path;
```

Description

The *send_p_prmtv_below* emulate routine passes a specified interlayer message buffer from Layer 7 to Layer 6 in an OSI primitive.

Inputs

The first parameter is the interlayer buffer number to be sent. Use the buffer-number variable named in the *_get_il_msg_buff* routine. (See *_insert_il_buff_list_cnt* routine example at Layer 5.)

The second parameter is the returned maintain bit from the call to *_get_il_msg_buff*.

The third parameter is the returned offset (from a call to *_start il_buff_list*) to the Layer 7 list header node in the buffer.

The fourth parameter is the size of the data in the buffer. It will always be set to zero since the data length is unknown in a primitive being passed down the layers.

The fifth parameter is the code specifying the type of primitive in which the buffer will be sent. Refer to variable *up_p_prmtv_code* in Table 63-7 for the appropriate code.

The sixth parameter is the path number along which the buffer will be sent.

Example

A buffer is obtained at Layer 7. The buffer will be passed to Layer 6, without any data inserted. (For information on inserting text, see *_insert_il_buff_list_cnt* routine.) If values are entered for the code and path, variables for code and path need not be declared.

```
{
 unsigned short il_buffer_number;
 unsigned short data_start_offset;
 unsigned short relay_baton;
}
LAYER: 7
     STATE: pass_buffer_down
        CONDITIONS: KEYBOARD " "
        ACTIONS:
        {
         _get_il_msg_buff(&il_buffer_number, &relay_baton);
         _start_il_buff_list(il_buffer_number, &data_start_offset);
         send_p_prmtv_below(il_buffer_number, relay_baton, data_start_offset, 0, 0xc4, 0);
        }
```

64 Print

The PRINTER port is a serial interface through which the programmer may direct output from the INTERVIEW to a printer. The printer port is located at the rear of the INTERVIEW between the REMOTE RS-232 and AUXILIARY ports.

> NOTE: Before directing output to the printer port, configure the Printer Setup menu as explained in Section 14.2.

Each spreadsheet PRINT action or call to one of the C print routines causes output to be added to a queue of unprinted text in the print buffer. If not doing so already, the print server also begins to poll the print buffer for text to print. As long as there is unprinted text in the buffer, the print server polls the buffer, removes text, and sends it to the printer port of the INTERVIEW. Use the *_print_buffer* structure to monitor the flow of text in and out of the print buffer.

Use any of the four C print routines explained in this section to add text to the print buffer. Three of them—*printc*, *printf*, and *prints*—are similar to the *displayc*, *displayf*, and *displays* routines which direct output to the Display Window. See Section 61.3(C). With the *set_print_header* routine, you determine the heading which will appear at the top of each printed page. One other routine, *sprintf*, writes output to a string. The string can then be referenced in subsequent calls to *printf*. (You may also use the string named in *sprintf* in calls to *displayf*, *tracef*, or *fprintf*.)

64.1 Structures

Refer to Table 64-1 for the structure of the print buffer. Compare *_print_buffer.in* with *_print_buffer.out* to determine whether or not the print buffer has emptied. When the values of these two variables are equal, the buffer is empty.

> NOTE: Consider the variables in the *_print_buffer* structure read-only variables. In general, do not modify *extern* structures or variables which may be updated by other processes.

At times, processes may add transactions to the print buffer more quickly than the print server takes them out. If a process cannot add to the buffer without overwriting unprinted text, a buffer overrun occurs. When your INTERVIEW is configured for data playback, you can minimize print-buffer overruns by periodically suspending playback and allowing the print server to empty the buffer. In judging how often to suspend playback, keep in mind the following points: 1) In general, the more conditions a program has that trigger print actions, the more frequently playback should be suspended. 2) When planning to print Run-mode buffers, remember that the faster the playback speed, the quicker the print buffer fills.

Table 64-1
Print Structures

Type	Variable	Value (hex/decimal)	Meaning
Structure Name: print_buffer			Structure of the print buffer. Declared as type *struct*.
unsigned short	in	*a-207/10-8199*	offset into the print buffer (from the physical beginning of the buffer) to the location where next transaction text will be added. Advances with each spreadsheet PRINT action or call to a C print routine. When *in* equals *out*, the print buffer is empty.
unsigned short	out	*a-207/10-8199*	offset into the print buffer (from the physical beginning of the buffer) to the last transaction text printed from the buffer. Advances each time text is actually sent out the printer port of the INTERVIEW. When *out* equals *in*, the print buffer is empty.
unsigned short	buffer_end	209/8201	offset to the physical end of the print buffer—i.e., to the end of the array named *buffer* (see below)
unsigned short	lock		when process is printing, locks out other processes from accessing the print buffer
char	polling	0 *non-zero*	print server is not polling print server is polling print buffer for text to print
char	overrun	0 *non-zero*	print buffer is not in overrun state print buffer is in overrun state—i.e., a process attempting to add text to the print buffer can't because unprinted text in the buffer would be overwritten. Following message will appear on printout: "*print buffer overrun has occurred.*"
char	buffer [8192]		array of text transactions
Structure Name: _print_buffer			An instance of the *print_buffer* structure, declared as type *extern struct print_buffer*. Use the variables contained in this structure to monitor flow of text in and out of the print buffer. Reference structure variables as follows: *_print_buffer.in*.

The following example shows how you might use a TIMEOUT condition to check the print buffer periodically. Each time the timeout expires, the program determines whether or not the buffer is half full. If so, playback is suspended. If the buffer is only one-quarter full, playback is resumed. (Other conditions in the program, not illustrated here, would cause print actions to send output to the print buffer.)

```
{
 #define PRINT_BUFFER_SZ 8192
 #define STOP_POINT (PRINT_BUFFER_SZ/2)
 #define START_POINT (PRINT_BUFFER_SZ/4)
}
LAYER: 1
     {
      struct print_buffer
       {
        unsigned short in;
        unsigned short out;
        unsigned short buffer_end;
        unsigned short lock;
        char polling;
        char overrun;
       };
      extern struct print_buffer _print_buffer;
      int crnt_buffer_sz;
     }
     STATE: check_print_buffer
             CONDITIONS: ENTER_STATE
             ACTIONS: TIMEOUT RESTART ck_buffer 0.01
             CONDITIONS: TIMEOUT ck_buffer
             ACTIONS:
             {
              crnt_buffer_sz = ((_print_buffer.in + PRINT_BUFFER_SZ) - _print_buffer.out) %
                   PRINT_BUFFER_SZ;
              if(crnt_buffer_sz > STOP_POINT)
                suspend_rcrd_play();
              else if(crnt_buffer_sz < START_POINT)
                start_rcrd_play();
             }
             TIMEOUT RESTART ck_buffer 0.01
```

64.2 Variables

There are no variables associated exclusively with print functions.

64.3 Routines

printc

Synopsis

extern void printc(character);
const char character;

Description

The *printc* routine outputs a single ASCII character to the print buffer for printing, converting the value provided as the argument into its ASCII equivalent. Decimal and octal values are converted to hexadecimal format before the ASCII equivalent is sought.

Inputs

The only parameter is a numerical value. The value may be given as a hexadecimal, octal, or decimal constant; as an alphanumeric constant inside of single quotes; or as a variable. A hexadecimal value must be preceded by the prefix 0x or 0X; an octal value must be preceded by the prefix 0. If no prefix appears before the input, the number is assumed to be decimal. Valid numeric entries are 00 to 127, decimal. An alphanumeric character placed between single quotes will be output as is to the printer.

Example

The *printc* entries on the left output the printed character given on the right:

printc('a');	a
printc(65);	A
printc(0x65);	e
printc(065);	5

printf

Synopsis

extern int printf(format_ptr, . . .);
*const char * format_ptr;*

Description

The *printf* routine writes output to the print buffer for printing, under control of the string pointed to by *format_ptr* that specifies how subsequent arguments are converted for output. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but otherwise ignored. The *printf* routine returns when the end of the format string is encountered.

Inputs

The format is composed of zero or more directives: ordinary characters (not %), which are copied unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments. Each conversion specification is introduced by the character %. After the %, the following appear in sequence:

- Zero or more *flags* that modify the meaning of the conversion specification. The flag characters and their meanings are:

  - The result of the conversion will be left-justified within the field.

  + The result of a signed conversion will always begin with a plus or minus sign.

  space If the first character of a signed conversion is not a sign, a space will be prepended to the result. If the *space* and + flags both appear, the *space* flag will be ignored.

  # The result is to be converted to an "alternate form." For d, i, u, c, and s conversions, the flag has no effect. For o conversion, it increases the precision to force the first digit of the result to be a zero. For x (or X) conversion, a nonzero result will have 0x (or 0X) prepended to it.

- An optional decimal integer specifying a minimum *field width*. If the converted value has fewer characters than the field width, it will be padded on the left (or right, if the left adjustment flag, described above, has been given) to the field width. The padding is with spaces unless the field width integer starts with a zero, in which case the padding is with zeros.

- An optional *precision* that gives the minimum number of digits to appear for the d, i, o, u, x, and X conversions, or the maximum number of characters to be written from an array in an s conversion. The precision takes the form of a period (.) followed by an optional decimal integer; if the integer is omitted, it is treated as zero. The amount of padding specified by the precision overrides that specified by the field width.

- An optional h specifying that a following d, i, o, u, x, or X conversion specifier applies to a *short int* or *unsigned short int* argument (the argument will have been promoted according to the integral promotions, and its value shall be converted to *short int* or *unsigned short int* before printing); or an optional l specifying that a following d, i, o, u, x, or X conversion specifier applies to a *long int* or *unsigned long int* argument. If an h or l appears with any other conversion specifier, it is ignored.

- A character that specifies the type of *conversion* to be applied. (Special AR extensions have been added.) The conversion specifiers and their meanings are:

  d, i, o, u, x, X

  The *int* argument is converted to signed decimal (d or i), unsigned octal (o), unsigned decimal (u), or unsigned hexadecimal notation (x or X); the letters abcdef are used for x conversion and the letters ABCDEF for X conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no characters.

  c — The *int* argument is converted to an *unsigned char*, and the resulting character is written.

  s — The argument shall be a pointer to a null-terminated array of 8-bit *chars*. Characters from the string are printed up to (but not including) the terminating null character: if the precision is specified, no more than that many characters are printed. The string may be an array into which output was written via the *sprintf* routine.

  p — The argument shall be a pointer to void. The value of the pointer is converted to a sequence of printable characters, in this format: *0000:0000*. There are always exactly 4 digits to the right of the colon. The number of digits to the left of the colon is determined by the pointer's value and the precision specified. Use this conversion to print 80286 memory addresses. The segment number will appear to the left of the colon and the offset to the right.

  % — A % is written. No argument is converted.

  \n — Writes hexadecimal 0D 0A, the ASCII carriage-return and linefeed characters. No argument is converted.

If a conversion specification is invalid, the behavior is undefined.

If any argument is or points to an aggregate (except for an array of characters using *%s* conversion or any pointer using *%p* conversion), the behavior is undefined.

In no case does a nonexistent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is expanded to contain the conversion result.

Returns

The *printf* routine returns the number of characters output.

Example

To print a date and time in the form "Sunday, July 3, 10:02," where weekday and month are pointers to strings:

```
LAYER: 1
{
 unsigned char date_time [100];
 unsigned char weekday [10];
 unsigned char month [10];
 unsigned short day;
 unsigned char hour;
 unsigned char min;
}
     STATE: output_to_printer
        CONDITIONS: KEYBOARD " "
        ACTIONS:
        {
         printf( "%s, %s %d, %.2d:%.2d\n", weekday, month, day, hour, min);
        }
```

sprintf

Synopsis

```
extern int sprintf(string_ptr, format_ptr);
unsigned char string [128];
const char * format_ptr;
```

Description

The *sprintf* routine is similar to the *printf* routine, except that *sprintf* writes output to a string, while *printf* writes output directly to the print buffer for printing. The *sprintf* routine is useful for writing formatted output to a display, printer, or file.

The output is under control of the string pointed to by *format_ptr* that specifies how subsequent arguments are converted for output. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but otherwise ignored. The *sprintf* routine returns when the end of the format string is encountered.

Inputs

The first parameter is a pointer to the array to which output will be written.

For the second parameter, see *printf* routine.

Returns

This routine returns the number of characters written into the array, not counting the added null terminating character.

Example

Refer again to the sample program for the *displayf* routine in Section 61.3(C). This time you also want to send the output to a printer. By using the *sprintf* routine, you only have to enter the format string once.

```
LAYER: 1
{
 unsigned char date_time [100];
 unsigned char weekday [10];
 unsigned char month [10];
 unsigned short day;
 unsigned char hour;
 unsigned char min;
}
     STATE: output_to_display_window_and_printer
        CONDITIONS: KEYBOARD " "
        ACTIONS:
        {
         sprintf(date_time, "%s, %s %d, %.2d: %.2d\n", weekday, month, day, hour,
               min);
         displayf("%s", date_time);
         printf("%s", date_time);
        }
```

set_print_header

Synopsis

extern int set_print_header(format_ptr);
*const char * format_ptr;*

Description

This routine writes output to the print buffer, to be printed after each form feed, under control of the string pointed to by *format_ptr*. Paging is done automatically by the INTERVIEW. The *set_print_header* routine returns when the end of the format string is encountered.

Inputs

The format is composed of zero or more ordinary characters. Octal or hexadecimal values also may be input, with octal preceded by \ and hex by \x. Pad each value to three integers with leading zeroes.

The status information shown above the prompt line on the display screens of the INTERVIEW can be sent to a printer with the following inputs:

#d	date	(mm/dd/yy)
#t	time	(hh:mm)
#p	page (not shown on the display screens)	
#b	block number	
##	#	

Returns

The *set_print_header* routine returns the length of the header (0-255), or a -1 if the header exceeds the buffer size.

Example

If you want the date, time, and page number to appear in the heading on each page sent to a printer, enter the following:

```
LAYER: 2
    STATE: header
       CONDITIONS: ENTER_STATE
       ACTIONS:
       {
        set_print_header("#### #d  #t                                   #p ####\n");
       }
```

The printer output will look like this:

```
## 09/01/89  09:30                                   Page :  1 ##
.
.
.
## 09/01/89  09:31                                   Page :  2 ##
.
.
.
```

prints

Synopsis

extern void prints(string_ptr);
*const char * string_ptr;*

Description

The *prints* routine is similar to the *displays* routines, except that *prints* writes output to the print buffer for printing while *displays* writes output to the Display Window. The output is under control of the string pointed to by the argument. The *prints* routine returns when the end of the string is encountered. The softkey equivalent of this routine is the PRINT PROMPT action on the Protocol Spreadsheet. A PRINT PROMPT action automatically time–stamps the output. Although *prints* does not, you can create your own time or date stamp with *set_print_header*.

Inputs

The input is a pointer to a string composed of zero or more ordinary characters. The newline nonliteral sequence "\n" writes hex 0D 0A (ASCII ␍␊) to the output string. Octal or hexadecimal values also may be included in the string, with octal preceded by \ and hex by \x. Pad each value to three integers with leading zeroes.

Example

The following entry

```
prints("End of test.");
```

produces the following output to a printer:

End of test.

65 Disk I/O

The disk I/O routines explained in this section allow disk files to be read from and written to during Run mode. "Streams" describes how most of the routines operate on a data stream rather than the actual file. Under "Routines," all the disk I/O routines are explained. These routines perform read and write functions as well as other file maintenance tasks in Run mode, such as creating directories, renaming files, and deleting files.

65.1 Streams

Most disk I/O routines are not executed on the actual disk file, but on a stream which includes a copy of the file's data. Opening a disk file for reading or writing associates a stream with the file. A stream may be input or output. Input streams are read-only. Output streams are write-only. In either case, the stream remains associated with a disk file until the file is closed.

You may have more than one stream associated with a given file. (A maximum of ten streams may be open at one time.) For example, to read from and write to an existing file, you must open the file twice, once to create an input stream and once to create an output stream.

(A) Stream Components

A stream contains everything needed to perform disk I/O functions on a file.

1. *Buffer*. A buffer containing a copy of the data in a disk file is part of the stream. When a disk file is opened for reading, sectors of the disk containing the file are copied to this buffer.

   Sometimes a file's size may exceed the maximum size (512 bytes) of the buffer. In this instance, as much data from the file as will fit in the buffer is copied. As each character is read from the input stream, it is removed. (The *ungetc* routine may temporarily return a removed character to an input stream.) Each call to *fread*, *fgetc*, or *fgets* further empties the buffer, while leaving the contents of the disk file unchanged. When the buffer is empty, the next sector (or sectors) of the disk file is (are) automatically copied into the buffer.

   Similarly, when a file is opened for writing, the empty buffer is filled as *fwrite* or other output routines are invoked. Characters written to the output stream are not transferred to the disk file until there is a call to *fflush*. *Fflush* is automatic in *fclose* or when the stream buffer is full.

2. *File-position indicator*. The file-position indicator keeps track of progression through the disk file. For files opened in read mode, the indicator is initially located at the first character (character zero) in the file. As characters are read from the input stream, the indicator advances through the file.

   For existing files opened in append mode, the indicator is positioned after the last character in the file. For newly created files or files opened in overwrite mode, it is located at the beginning of the file. Every time an output routine is executed, the file-position indicator is advanced by the number of characters successfully written to the stream.

3. *Buffer pointer*. The stream also contains a pointer into the associated buffer of a file. In input streams, it points to the next character to be read. In output streams, it points to the next empty byte.

4. *EOF indicator*. If the end-of-file (EOF) indicator is set in a input stream, it means that a read operation encountered the end of the file. The EOF indicator is cleared via calls to *fopen*, *fseek*, *rewind*, *clearerr*, or *ungetc*.

5. *Error indicator*. In input streams, this indicator gets set when an *fread*, *fgetc*, or *fgets* routine does not successfully execute. Attempting to execute these input routines (or *ungetc)* on an output stream sets the error indicator. In output streams, the error indicator gets set when the *fflush*, *fwrite*, *fputc*, *fputs*, or *fprintf* routine does not successfully execute, or when output routines try to execute on an input stream. A call to *fopen, clearerr*, or *fseek*, clears the error indicator in either input or output streams. A *rewind* operation on an input stream also clears the indicator.

(B) Stream Pointer

The *fopen* routine returns a pointer to the stream. Disk I/O routines which perform operations on a stream require the stream pointer as an argument. It has been named *stream_ptr* in the routines discussed below.

(C) Locking Streams

Each file stream is locked internally during operations on it. If the user program is executing different conditions on multiple processors and both actions require writing to the *same file stream*, internally the *stdio* library will allow the first task that requests to write to execute until completion and the second task will be locked out. All processes that are locked out are temporarily put to sleep and removed from the tasking queues for that CPU. When the first process completes its operations on the stream, the locked-out processes are woken up and may try to claim the lock. Deadlock or deadly embrace situations can never arise internally to the *stdio* library.

If two or more file streams are associated with a single *file*, processes on each stream may try to operate on the file concurrently. Internal locking does not apply in this situation, so use the locking routines.

65.2 Routines

Disk I/O routines fall into four categories. The first category includes routines valid for both input and output streams, including the two locking routines (not exclusive to disk I/O). The remaining groups are routines valid for input streams only, routines applicable to output streams only, and routines which handle other file maintenance functions.

The routines and their descriptions closely conform to the ANSI specification for the Programming Language C, as defined in the draft document published July 9, 1986. Discrepancies with the ANSI standard are noted. The document number is X3J11-86-098. Refer to pages 107-129.

Use the *#include <stdio.h>* pre-processor directive with all disk I/O routines. The *stdio.h* file contains type definitions and function prototypes, making declarations of the routines unnecessary.

When a filename is required as an argument, give the absolute pathname of the file, prefixed by the device name. Valid device names are FD1, FD2, or HRD. See Section 13.2(B) for a discussion of absolute pathnames. The disk filename is required as an argument for the *fopen* routine, which opens a file for reading or writing. From that point on, disk I/O routines relating to that file use the stream pointer, explained above, as input. File maintenance routines, such as *rename* or *remove*, use the filename as input.

> **NOTE:** A single program can perform disk I/O functions as well as data playback or recording. Disk I/O, however, must be suspended while disk recording (or playback) proceeds, and vice versa. RAM recording, on the other hand, may occur simultaneously with disk I/O operations. Refer to the *start_rcrd_play* and *suspend_rcrd_play* routines in Section 69 for more information on the interaction between disk I/O and recording/playback.

(A) Input/Output-Stream Routines

Several disk I/O routines may be executed on either input or output streams. *fopen* opens an existing disk file for reading or writing, or creates a new file. In each case, a stream is associated with the file until there is a call to *fclose*. *fclose* or a specific call to *fflush* delivers any output written to a stream to the host environment where it will be written to the disk file.

> **NOTE:** Always include a call to *fclose* in your program to make sure output is written to the file.

Test the end-of-file and error indicators with the *feof* and *ferror* routines, respectively. These same indicators may be cleared via the *clearerr* routine.

The *fseek* and *rewind* routines manipulate the file-position indicator and erase any memory of a character put into the stream via *ungetc*.

The *lock* and *unlock* routines prevent deadlock from occurring when processes on multiple streams try to operate concurrently on a single file.

fopen

Synopsis

```
#include <stdio.h>
extern FILE * fopen(filename_ptr, mode_ptr);
const char * filename_ptr;
const char * mode_ptr;
```

Description

The *fopen* routine opens a file for access. Depending on the open mode, a file can be opened for reading (via an input stream) or for writing (via an output stream). For existing files, this routine also clears the end-of-file and error indicators.

Inputs

The first parameter is a pointer to the file to be opened, represented as the name of the file, placed inside double quotation marks. The filename must be the absolute pathname, prefixed by the device name (HRD, FD1, or FD2).

The second parameter is a pointer to a string (represented as a character inside double quotation marks) which identifies the type of open to be performed. Of the ANSI standard open modes, the following are supported:

r Open an existing file for reading only. The file-position indicator is located at the start (character zero) of the file.

w Create a file, or open an existing file, for writing only. For an existing file, truncate its length to zero and discard the contents.

a Create a file, or open an existing file, for writing only. For an existing file, retain the contents and locate the file-position indicator at the end of the file. Append new data to the end of existing data, unless a call to *fseek* or *rewind* has repositioned the file-position indicator. In this instance, overwrite existing data. (This implementation is different from the ANSI specification which appends new data to the end of existing data regardless of any previous calls to *fseek*.)

rb Currently implemented the same as "r." Use "rb" for the *fseek* routine.

wb Currently implemented the same as "w." Use "wb" for the *fseek* routine.

ab Currently implemented the same as "a." Use "ab" for the *fseek* routine.

Returns

This routine returns a pointer to the stream, with a type definition FILE (defined in the *stdio.h* file).

If the open fails (for example, the file does not exist), zero is returned.

Example

Open a file called "*buff01*" in the */usr* directory on a disk in floppy drive 2. Store the pointer to the stream in *stream_ptr*. Indicate whether or not the open is successful on the prompt line.

```
{
 #include <stdio.h>
 FILE * stream_ptr;
}
LAYER: 1
     STATE: open_a_file
         CONDITIONS: ENTER_STATE
         ACTIONS: PROMPT "Press O to open file.                                    "
         CONDITIONS: KEYBOARD "oO"
         ACTIONS:
         {
          if((stream_ptr = fopen("FD2/usr/buff01", "r")) == 0)
             display_prompt("Cannot open file.                                    ");
          else
             display_prompt("File opened.                                         ");
         }
```

fclose

Synopsis

```
#include <stdio.h>
extern int fclose(stream_ptr);
FILE * stream_ptr;
```

Description

All opened files must be closed. If the disk file to be closed is an input file, then any data remaining in the stream buffer is discarded. If the file is an output file, any data written to the stream is written to the file. (In other words, *fclose* automatically calls *fflush*.) The stream is freed from its association with the disk file, and the disk file is closed.

Inputs

The only parameter is the stream pointer.

Returns

If the stream is successfully closed, zero is returned. If errors are detected, or if the stream is already closed, a non-zero value is returned.

Example

Close the file that was opened in the *fopen* example. Indicate whether or not the close is successful on the prompt line.

```
{
#include <stdio.h>
FILE * stream_ptr;
}
LAYER: 1
    STATE: open_and_close_a_file
       CONDITIONS: ENTER_STATE
       ACTIONS: PROMPT "Press O to open file.                                    "
       CONDITIONS: KEYBOARD "oO"
       ACTIONS:
       {
        if((stream_ptr = fopen("FD2/usr/buff01", "r")) == 0)
           display_prompt("Cannot open file.                                     ");
        else
           display_prompt("File opened.                                          ");
       }
       CONDITIONS: KEYBOARD "cC"
       ACTIONS:
       {
        if(fclose(stream_ptr) != 0)
           display_prompt("Either file is already closed, or close cannot be executed.   ");
        else
           display_prompt("File closed.                                          ");
       }
```

fflush

Synopsis

```
#include <stdio.h>
extern int fflush(stream_ptr)
FILE * stream_ptr;
```

Description

If *stream_ptr* points to an output stream, the *fflush* routine causes any unwritten data for that stream to be delivered to the host environment where it will be written to the file. If *stream_ptr* points to an input stream, the *fflush* routine undoes the effect of any preceding *ungetc* operation on the stream.

Inputs

The only parameter is the stream pointer.

Returns

If a write error occurs, non-zero is returned and the error indicator is set.

Example

Assume the X.25 personality package has been loaded in at Layer 2. Whenever you receive a frame type "unknown," write the actual value of the control byte to an output file stream *and* to the disk file.

```
{
 #include <stdio.h>
 FILE * stream_ptr;
 extern volatile const unsigned char rcvd_frame_cntrl_byte_1;
}
LAYER: 2
     STATE: write_then_fflush
        CONDITIONS: ENTER_STATE
        ACTIONS:
        {
         if((stream_ptr = fopen("FD2/usr/frame_unkwn", "a")) == 0)
            display_prompt("Cannot open file.                                  ");
         else
            display_prompt("File opened.                                       ");
         pos_cursor(1,0);
        }
        CONDITIONS: RCV UNKNOWN
        ACTIONS:
        {
         if(fprintf(stream_ptr, "%02x\n ", rcvd_frame_cntrl_byte_1) < 0)
            displayf("Error in printing to stream.                       \n");
         else
            displayf("Print to stream completed.                         \n");
         if(fflush(stream_ptr) != 0)
            display_prompt("Write error.                                       ");
         else
            display_prompt("Write to file completed.  Press C to close file.   ");
        }
        CONDITIONS: KEYBOARD "cC"
        ACTIONS:
        {
         if(fclose(stream_ptr) != 0)
            display_prompt("Either file is already closed, or close cannot be executed.   ");
         else
            display_prompt("File closed.                                       ");
        }
```

feof

Synopsis

```
#include <stdio.h>
extern int feof(stream_ptr);
FILE * stream_ptr;
```

Description

This routine tests the end-of-file indicator for an associated stream.

Inputs

The only parameter is the stream pointer.

Returns

The *feof* routine returns a non-zero value if the end-of-file indicator is set for the stream.

Example

Get a character from a file. If it is not at the end of the file, display it; otherwise prompt with "End of file."

```
{
 #include <stdio.h>
 FILE * stream_ptr;
 int character;
}
LAYER: 1
     STATE: test_for_eof
        CONDITIONS: ENTER_STATE
        ACTIONS: PROMPT "Press O to open file.                                    "
        CONDITIONS: KEYBOARD "oO"
        ACTIONS:
        {
         if((stream_ptr = fopen("FD2/usr/buff01", "rb")) == 0)
            display_prompt("Cannot open file.                                    ");
         else
            display_prompt("File opened.. Press G to get character.              ");
         pos_cursor(1,0);
        }
        CONDITIONS: KEYBOARD "gG"
        ACTIONS:
        {
         character = fgetc(stream_ptr);
         if(feof(stream_ptr) != 0)
            display_prompt("End of file.  Press C to close file.                 ");
         else
            displayf("%c", character);
        }
        CONDITIONS: KEYBOARD "cC"
        ACTIONS:
        {
         if(fclose(stream_ptr) != 0)
            display_prompt("Either file is already closed, or close cannot be executed.   ");
         else
            display_prompt("File closed.                                         ");
        }
```

ferror

Synopsis

```
#include <stdio.h>
extern int ferror(stream_ptr);
FILE * stream_ptr;
```

Description

This routine tests the error indicator for a stream.

Inputs

The only parameter is the stream pointer.

Returns

The *ferror* routine returns a non-zero value if the error indicator is set for the stream.

Example

Read a file called *"buff01"* from the */usr* directory on the disk in drive 2. If the number of elements read is less than the number designated to be read, determine whether an end-of-file was encountered or a read error occurred.

```
{
 #include <stdio.h>
 FILE * stream_ptr;
 char data [6091];
 size_t n;
}
LAYER: 1
      STATE: read_a_file
         CONDITIONS: ENTER_STATE
         ACTIONS: PROMPT "Press O to open file.                                   "
         CONDITIONS: KEYBOARD "oO"
         ACTIONS:
         {
          if((stream_ptr = fopen("FD2/usr/buff01", "r")) == 0)
             display_prompt("Cannot open file.                                    ");
          else
             display_prompt("File opened.  Press R to read the file.              ");
         }
         CONDITIONS: KEYBOARD "rR"
         ACTIONS:
         {
          n = fread(data, 1, 6091, stream_ptr);
          if(n != 6091)
             {
              if(ferror(stream_ptr) != 0)
                display_prompt("Read error.                                       ");
              else if(feof(stream_ptr) != 0)
                display_prompt("End-of-file encountered.                          ");
             }
          else
             displayf("\n%.6091s", data);
          display_prompt("Press C to close the file.                            ");
         }
```

```
CONDITIONS: KEYBOARD "cC"
ACTIONS:
{
 if(fclose(stream_ptr) != 0)
    display_prompt("Either file is already closed, or close cannot be executed.      ");
 else
    display_prompt("File closed.                                                     ");
}
```

clearerr

Synopsis

```
#include <stdio.h>
extern void clearerr(stream_ptr);
FILE * stream_ptr;
```

Description

This routine clears the end-of-file and error indicators for a stream. When an error occurs, no further operations are allowed until the error indicators are explicitly cleared. (These indicators are also cleared by a *fopen* or *rewind* operation.)

Inputs

The only parameter is the stream pointer.

Example

If a write error occurs, clear the indicators.

```
{
 #include <stdio.h>
 FILE * stream_ptr;
 int character;
}
LAYER: 1
     STATE: clear_indicators
        CONDITIONS: ENTER_STATE
        ACTIONS: PROMPT "Press O to open file.                                       "
        CONDITIONS: KEYBOARD "oO"
        ACTIONS:
        {
         if((stream_ptr = fopen("FD2/usr/buff01", "wb")) == 0)
            display_prompt("Cannot open file.                                       ");
         else
            display_prompt("File opened. Press P to write character.                ");
        }
```

```
CONDITIONS: KEYBOARD "pP"
ACTIONS:
{
 character = fputc('h', stream_ptr);
 if(character == EOF)
   {
   display_prompt("Write error.  All indicators will be cleared.                    ");
   clearerr(stream_ptr);
   }
 else
   display_prompt("Write completed.  Press C to close the file.                     ");
}
CONDITIONS: KEYBOARD "cC"
ACTIONS:
{
 if(fclose(stream_ptr) != 0)
   display_prompt("Either file is already closed, or close cannot be executed.      ");
 else
   display_prompt("File closed.                                                     ");
}
```

fseek

Synopsis

```
#include <stdio.h>
extern int fseek(stream_ptr, bytes, reference);
FILE * stream_ptr;
long int bytes;
int reference;
```

Description

This routine manipulates the file-position indicator, according to the ANSI specification for binary files. Future read operations will be referenced from that point. *fseek* clears the end-of-file indicator and resets the *ungetc* variable.

> **NOTE:** The ANSI specification for text files is not currently implemented. To ensure proper execution of *fseek* if future releases include the ANSI specification for text files, open files for *fseek* as binary ("rb," "wb," or "ab").

Inputs

The first parameter is the stream pointer.

The second parameter is the number of characters the file-position indicator should be moved from a specified position. A positive number advances the file-position indicator forward in the file; a negative number moves it backward.

The third parameter specifies the location of the file-position indicator. SEEK_SET will move the file-position indicator from the beginning of the file; SEEK_END will move the file-position indicator from the end-of-file; and SEEK_CUR will move the file-position indicator from its current position.

Returns

This routine returns non-zero for an improper request; otherwise it returns zero.

Example

Open a file and move the file-position indicator 4 characters from the beginning of the file. Each time the [S] key is pressed, move the indicator one character backward from its current position. After 4 executions, the indicator will be back at the beginning of the file.

```
{
 #include <stdio.h>
 FILE * stream_ptr;
 int character;
}
LAYER: 1
     STATE: move_indicator
        CONDITIONS: ENTER_STATE
        ACTIONS: PROMPT "Press O to open file.                              "
        CONDITIONS: KEYBOARD "oO"
        ACTIONS:
        {
         if((stream_ptr = fopen("FD2/usr/buff01", "rb")) == 0)
            display_prompt("Cannot open file.                               ");
         else
           {
            display_prompt("File opened.  ");
            pos_cursor(0,14);
            if(fseek(stream_ptr, 4, SEEK_SET) != 0)
              displays("Improper fseek request.                             ");
            else
              displays("Fseek completed.  Press S to seek new position.     ");
           }
        }
        CONDITIONS: KEYBOARD "sS"
        ACTIONS:
        {
         if(fseek(stream_ptr, -1, SEEK_CUR) != 0)
            display_prompt("Improper fseek request.  Press C to close file.  ");
         else
            display_prompt("Fseek completed.  Press C to close file.         ");
        }
```

```
CONDITIONS: KEYBOARD "cC"
ACTIONS:
{
 if(fclose(stream_ptr) != 0)
    display_prompt("Either file is already closed, or close cannot be executed.      ");
 else
    display_prompt("File closed.                                                      ");
}
```

rewind

Synopsis

```
#include <stdio.h>
extern void rewind(stream_ptr);
FILE * stream_ptr;
```

Description

This routine returns the file-position indicator to the beginning of the file (i.e., it is equivalent to an *fseek* with the number of characters to move set as zero and the specified position SEEK_SET). The rewind operation also clears the end-of-file and error indicators and erases any memory of the character in a previous *ungetc* operation.

Inputs

The only parameter is the stream pointer.

Example

In this example, the first call to *fgetc* following the *rewind* operation will read the first character in the file.

```
{
 #include <stdio.h>
 FILE * stream_ptr;
 int character;
}
LAYER: 1
    STATE: move_indicator
      CONDITIONS: ENTER_STATE
      ACTIONS: PROMPT "Press O to open file.                                           "
      CONDITIONS: KEYBOARD "oO"
      ACTIONS:
      {
       if((stream_ptr = fopen("FD2/usr/buff01", "rb")) == 0)
          display_prompt("Cannot open file.                                            ");
       else
          display_prompt("File opened.  Press S to fseek.                              ");
      }
```

```
CONDITIONS: KEYBOARD "sS"
ACTIONS:
{
 if(fseek(stream_ptr, 4, SEEK_SET) != 0)
    display_prompt("Improper fseek request.                                  ");
 else
    display_prompt("Fseek completed.  Press spacebar to rewind.              ");
}
CONDITIONS: KEYBOARD " "
ACTIONS:
{
 rewind(stream_ptr);
 display_prompt("Press G to get a character.                                 ");
}
CONDITIONS: KEYBOARD "gG"
ACTIONS:
{
 character =fgetc(stream_ptr);
 display_prompt("Press C to close file.                                      ");
}
CONDITIONS: KEYBOARD "cC"
ACTIONS:
{
 if(fclose(stream_ptr) != 0)
    display_prompt("Either file is already closed, or close cannot be executed.  ");
 else
    display_prompt("File closed.                                              ");
}
```

lock

Synopsis

```
#include <stdio.h>
extern void lock(lock_variable_ptr);
int * lock_variable_ptr;
```

Description

The *lock* routine implements a lock using the integer variable pointed to by the routine parameter. If the lock variable is currently locked, the task goes to sleep. When an unlock on the same variable occurs (within an independent task), the task invoking the lock function will attempt to claim the lock. If successful, the task is executed; otherwise, it goes back to sleep until the next unlock.

> **NOTE:** If locking is used at any place in the program, all related or possibly concurrent routines must also use the locking functions.

> **NOTE:** The lock variable should always be defined as a global integer, never as local to a function. The lock variable should never be altered by the user program or deadlock can occur. Deadlock also results if the lock is invoked twice within the same task without an intervening unlock.

Inputs

The only parameter is a pointer to the lock variable.

Example

Two tasks concurrently write to their own file streams. The file streams are local to the routine *write_fox*, making them independent of each other even though both are referenced by *stream_ptr*. During the *fclose* operation (which automatically calls *fflush*), however, both tasks need to write to the same file. The locking routines ensure that the writes to the file occur sequentially, not concurrently.

```
{
 #include <stdio.h>
 const char data [] = "((FOX)) \n";
 int key;
 void write_fox()
  {
   FILE * stream_ptr;
   size_t n;
   lock(&key);
   if((stream_ptr = fopen("FD2/usr/buff01", "a")) == 0)
     display_prompt("Cannot open file.                                            ");
   else
     display_prompt("File opened.                                                 ");
   n = fwrite(data, 1, sizeof(data)-1, stream_ptr);
   pos_cursor(1,0);
   if(n != (sizeof(data)-1))
     displayf("Write error.                                                 \n");
   else
     displayf("Write completed.                                             \n");
   if(fclose(stream_ptr) != 0)
     displayf("Either file is already closed, or close cannot be executed.   ");
   else
     displayf("File closed.                                                 ");
   unlock(&key);
  }
}
LAYER: 1
  TEST: a
    STATE: write_and_signal
       CONDITIONS: RECEIVE STRING "THE QUICK BROWN FOX"
       ACTIONS: SIGNAL xyz
       {
        write_fox();
       }
```

```
TEST: b
   STATE: write_only
      CONDITIONS: ON_SIGNAL xyz
      ACTIONS:
      {
       write_fox();
      }
```

unlock

Synopsis

```
#include <stdio.h>
extern void unlock(lock_variable_ptr);
int * lock_variable_ptr;
```

Description

The *unlock* routine implements the inverse of the *lock* routine using the same integer variable. Sleeping tasks will be woken up to retry their attempt to claim the lock. One will succeed, and the rest will go back to sleep. See also *lock* routine.

Inputs

The only parameter is a pointer to the lock variable.

Example

See *lock* routine.

(B) Input-Stream Routines

The following routines are valid for input streams only. An attempt to apply them to output streams results in a read error. The error indicator for the input stream will be set.

Three routines read characters from the input stream. The *fread* and *fgets* routines transfer a specified number of characters from the stream buffer into a user-defined array. *fgetc* reads the next character from the input stream. The *ungetc* routine temporarily forces a designated character back into the input stream.

fread

Synopsis

```
#include <stdio.h>
extern size_t fread(data_ptr, size, number, stream_ptr);
void * data_ptr;
size_t size;
size_t number;
FILE * stream_ptr;
```

Description

This routine reads elements from the input-stream buffer and puts them into a user-defined buffer. The file-position indicator is advanced by the number of characters successfully read. The *fread* routine can read a file whose elements are more than eight bits each, 16-bit *shorts* or 32-bit *longs*, for example. The *fgets* routine is similar to *fread*. *fgets*, however, reads only 8-bit characters. The primary use of *fread* is to read the entire contents of a file, whereas the primary purpose of *fgets* is to read from a file one line at a time.

Inputs

The first parameter is a pointer to an array in which the incoming data should be placed.

The second parameter is the number of bytes in each element to be read. If the value of this parameter is zero, the contents of the array and the stream remain unchanged.

The third parameter is the number of elements to be read. If the value of this parameter is zero, the contents of the array and the stream remain unchanged.

The fourth parameter is the stream pointer.

Returns

The *fread* routine returns the total number of elements read. If the number of elements read is less than the number of elements designated to be read, an end-of-file has been encountered or a read error has occurred. Use the *feof* and *ferror* routines to distinguish an end-of-file from a read error. If an error occurs, the location of the file-position indicator is indeterminate.

Example

Read in a file called "*buff01*" from the */usr* directory on the disk in drive 2 and display it on the Program Trace screen. (See Section 61.4 for information on using trace buffers in C.) Determine the size of the array *data* from the file size indicated on the File Maintenance screen.

```
{
#include <trace_buf.h>
#include <stdio.h>
FILE * stream_ptr;
char data [6091];
size_t n;
extern struct trace_buf prog_trbuf;
}
```

```
LAYER: 1
     STATE: read_a_file
        CONDITIONS: ENTER_STATE
        ACTIONS: PROMPT "Press O to open file.                                              "
        CONDITIONS: KEYBOARD "oO"
        ACTIONS:
        {
          if((stream_ptr = fopen("FD2/usr/buff01", "r")) == 0)
             display_prompt("Cannot open file.                                              ");
         else
             display_prompt("File opened.  Press R to read the file.                        ");
        }
        CONDITIONS: KEYBOARD "rR"
        ACTIONS:
        {
         n = fread(data, 1, 6091, stream_ptr);
         if(n != 6091)
             display_prompt("Either a read error has occurred, or an EOF has been
                  encountered.    ");
         else
           {
            tracef(&prog_trbuf, "%.6091s", data);
            display_prompt("Press C to close the file.                                     ");
           }
        }
        CONDITIONS: KEYBOARD "cC"
        ACTIONS:
        {
         if(fclose(stream_ptr) != 0)
             display_prompt("Either file is already closed, or close cannot be executed.     ");
         else
             display_prompt("File closed.                                                   ");
        }
```

fgets

Synopsis

```
#include <stdio.h>
extern char * fgets(string_ptr, max_number, stream_ptr);
char * string_ptr;
int max_number;
FILE * stream_ptr;
```

Description

This routine gets at the most one less than the specified number of characters from an input stream and puts them in an array. If an EOF, newline, or null is encountered in the stream, no more characters will be read, even if the specified number of characters has not yet been read. The newline will be retained. A terminating null character is written after the last character read into the array. The file-position indicator is advanced by the number of characters successfully read. The *fgets* routine is similar to *fread*. The *fread* routine can read a file whose elements are more than eight bits each, 16-bit *shorts* or 32-bit *longs*, for example. *fgets*, however, reads only 8-bit characters. The primary use of *fgets* is to read from a file one line at a time.

Inputs

The first parameter is a pointer to the array into which the characters will be put.

The second parameter is the maximum number of characters (minus one) to be read.

The third parameter is the stream pointer.

Returns

If the routine is successful, a pointer to the array is returned. If end-of-file is encountered before any characters have been read into the array or if a read error occurs, a null pointer is returned. The contents of the array are indeterminate when a read error occurs.

Example

Five characters, at the most, from a disk file will be put into an array called *data* and displayed on the screen.

```
{
 #include <stdio.h>
 FILE * stream_ptr;
 char data [10];
}
LAYER: 1
     STATE: read_characters
        CONDITIONS: ENTER_STATE
        ACTIONS: PROMPT "Press O to open file.                                      "
        CONDITIONS: KEYBOARD "oO"
        ACTIONS:
        {
         if((stream_ptr = fopen("FD2/usr/buff01", "r")) == 0)
            display_prompt("Cannot open file.                                        ");
         else
            display_prompt("File opened.  Press G to get string.                     ");
        }
        CONDITIONS: KEYBOARD "gG"
        ACTIONS:
        {
         fgets(data, 6, stream_ptr);
         displayf("\n%.6s", data);
         display_prompt("Press C to close the file.                                  ");
        }
```

```
CONDITIONS: KEYBOARD "cC"
ACTIONS:
{
 if(fclose(stream_ptr) != 0)
    display_prompt("Either file is already closed, or close cannot be executed.     ");
 else
    display_prompt("File closed.                                                     ");
}
```

fgetc

Synopsis

```
#include <stdio.h>
extern int fgetc(stream_ptr);
FILE * stream_ptr;
```

Description

The *fgetc* routine gets the next character (if present) from the input stream. The character is an *unsigned char* cast to an *int* (stored in the least-significant byte of the *int*). The file-position indicator advances by one character.

Inputs

The only parameter is the stream pointer.

Returns

This routine returns the next character in the input stream. EOF is returned if an end-of-file is encountered or if a read error occurs. The *stdio.h* file defines the macro EOF as −1. Use the *feof* and *ferror* routines to determine the reason for a returned EOF.

Example

In the following example, open an input file for reading. Each time the G key is pressed, display the next character in the file.

```
{
 #include <stdio.h>
 FILE * stream_ptr;
 int character, end;
}
LAYER: 1
    STATE: get_next_character
       CONDITIONS: ENTER_STATE
       ACTIONS: PROMPT "Press O to open file.                                      "
       CONDITIONS: KEYBOARD "oO"
       ACTIONS:
       {
        if((stream_ptr = fopen("FD2/usr/buff01", "r")) == 0)
           display_prompt("Cannot open file.                                        ");
        else
           display_prompt("File opened.  Press G to get a character.                ");
        displayf("\n");
       }
```

```
CONDITIONS: KEYBOARD "gG"
ACTIONS:
{
 character = fgetc(stream_ptr);
 if(character == EOF)
    {
    end = feof(stream_ptr);
    if(end != 0)
      display_prompt("EOF encountered.                                              ");
    else
      display_prompt("Read error.                                                   ");
    }
 else
    displayf("%c", character);
}
CONDITIONS: KEYBOARD "cC"
ACTIONS:
{
 if(fclose(stream_ptr) != 0)
    display_prompt("Either file is already closed, or close cannot be executed.    ");
 else
    display_prompt("File closed.                                                   ");
}
```

ungetc

Synopsis

```
#include <stdio.h>
extern int ungetc(character, stream_ptr);
int character;
FILE * stream_ptr;
```

Description

This routine temporarily forces a specified character into a variable associated with the input stream, overwriting the previous *ungetc* variable. The routine does not affect the location of the file-position indicator. The next *fgetc* will read the *ungetc* variable, not the stream. An intervening *fflush*, *fseek*, or *rewind* erases memory of the character. If the *ungetc* function is called too many times on the same stream without an intervening read, *fflush*, *fseek*, or *rewind* operation on that stream, the operation may fail. *Ungetc* also clears the end-of-file indicator.

Inputs

The first parameter is the character to be put into the input stream.

The second parameter is the stream pointer.

Returns

This routine returns the specified character. If the operation fails, EOF is returned and the input stream remains unchanged. It will fail if the values of the specified character and the macro EOF are equal.

Example

Read a character from the stream. Press the U key when you want to return the last character read to the stream. The next call to *fgetc* will read the returned character.

```
{
 #include <stdio.h>
 FILE * stream_ptr;
 int character;
}
LAYER: 1
    STATE: get_next_character
      CONDITIONS: ENTER_STATE
      ACTIONS: PROMPT "Press O to open file.                              "
      CONDITIONS: KEYBOARD "oO"
      ACTIONS:
      {
       if((stream_ptr = fopen("FD2/usr/buff01", "r")) == 0)
          display_prompt("Cannot open file.                              ");
       else
          display_prompt("File opened.  Press G to get a character.       ");
      }
      CONDITIONS: KEYBOARD "gG"
      ACTIONS:
      {
       character = fgetc(stream_ptr);
       if(character == EOF)
          display_prompt("End of file or read error.                     ");
       else
          {
          pos_cursor(0,0);
          displayf("character = %c  Press U to return character to stream.", character);
          }
      }
      CONDITIONS: KEYBOARD "uU"
      ACTIONS:
      {
       if((ungetc(character, stream_ptr)) == EOF)
          display_prompt("Character not returned.                        ");
       else
          display_prompt("Character returned.                            ");
      }
      CONDITIONS: KEYBOARD "cC"
      ACTIONS:
      {
       if(fclose(stream_ptr) != 0)
          display_prompt("Either file is already closed, or close cannot be executed.   ");
       else
          display_prompt("File closed.                                   ");
      }
```

(C) Output-Stream Routines

The following routines are valid for output streams only. An attempt to apply them to input streams will result in a write error. The error indicator for the output stream will be set.

Four routines write to output streams. The *fwrite* and *fputs* routines transfer a specified number of characters from a user-defined array into the stream buffer. *fputc* writes a character to the next empty byte in an output-stream buffer. *fprintf* writes formatted output to an output stream similar to the way *displayf* writes output to the Display Window.

fwrite

Synopsis

```
#include <stdio.h>
extern size_t fwrite(output_ptr, size, number, stream_ptr);
const void * output_ptr;
size_t size;
size_t number;
FILE * stream_ptr;
```

Description

This routine writes elements from a user-defined array to the output-stream buffer. The file-position indicator is advanced by the number of characters successfully written.

Inputs

The first parameter is a pointer to an array from which the data should be taken. Declare it as *const* if it is read-only. In cases where the array will be written to, as in the example below, do not include *const* as part of the declaration.

The second parameter is the number of bytes in each element to be written.

The third parameter is the number of elements to be written.

The fourth parameter is the stream pointer.

Returns

The *fwrite* routine returns the total number of elements written. If the number of elements written is less than the number of elements designated to be written, a write error has occurred. If an error occurs, the location of the file-position indicator is indeterminate.

Example

Read the contents of a file, and write them to a new file.

```
{
 #include <stdio.h>
 FILE * read_stream;
 FILE * write_stream;
 char output [6091];
 size_t n;
}
LAYER: 1
     STATE: write_to_a_file
        CONDITIONS: ENTER_STATE
        ACTIONS: PROMPT "Press O to open files.                                      "
        CONDITIONS: KEYBOARD "oO"
        ACTIONS:
        {
         if((read_stream = fopen("FD2/usr/buff01", "r")) == 0)
            {
            display_prompt("Cannot open buff01.  ");
            pos_cursor(0,21);
            }
         else
            {
            display_prompt("Buff01 opened.  ");
            pos_cursor(0,16);
            }
         if((write_stream = fopen("FD2/usr/new_file", "w")) == 0)
            displays("Cannot open new_file.                                        ");
         else
            displays("New_file opened.  Press R to read buff01.                     ");
        }
        CONDITIONS: KEYBOARD "rR"
        ACTIONS:
        {
         n = fread(output, 1, 6091, read_stream);
         if(n != 6091)
            display_prompt("Either a read error has occurred, or an EOF has been
                 encountered.     ");
         else
            display_prompt("Press W to write to new_file.                              ");
        }
        CONDITIONS: KEYBOARD "wW"
        ACTIONS:
        {
         n = fwrite(output, 1, 6091, write_stream);
         if(n != 6091)
            display_prompt("Write error.  Press C to close files.                     ");
         else
            display_prompt("Write completed.  Press C to close files.                 ");
        }
```

```
CONDITIONS: KEYBOARD "cC"
ACTIONS:
{
 if(fclose(read_stream) != 0)
   {
   display_prompt("Either buff01 is already closed, or close cannot be executed.   ");
   pos_cursor(0,0);
   }
 else
   {
   display_prompt("Buff01 closed.   ");
   pos_cursor(0,16);
   }
 if(fclose(write_stream) != 0)
   displays("Either new_file is already closed, or close cannot be executed.      ");
 else
   displays("New_file closed.                                                     ");
}
```

fputs

Synopsis

```
#include <stdio.h>
extern int fputs(string_ptr, stream_ptr);
const char * string_ptr;
FILE * stream_ptr;
```

Description

This routine writes a string of characters from an array, excluding the terminating null character, to the output stream. The file-position indicator is advanced by the number of characters successfully written.

Inputs

The first parameter is a pointer to the string to be written.

The second parameter is the stream pointer.

Returns

This routine returns zero if it is successful; it returns a non-zero value if a write error occurs.

Example

Write a fox message at the end of existing data in a file.

```
{
 #include <stdio.h>
 FILE * stream_ptr;
 char data [] = "((FOX))\n";
}
```

```
LAYER: 1
    STATE: write_a_string
      CONDITIONS: ENTER_STATE
      ACTIONS: PROMPT "Press O to open file.                                          "
      CONDITIONS: KEYBOARD "oO"
      ACTIONS:
      {
       if((stream_ptr = fopen("FD2/usr/buff01", "a")) == 0)
          display_prompt("Cannot open file.                                          ");
       else
          display_prompt("File opened.  Press P to write string.                     ");
      }
      CONDITIONS: KEYBOARD "pP"
      ACTIONS:
      {
       if(fputs(data, stream_ptr) != 0)
          display_prompt("Write error.  Press C to close file.                       ");
       else
          display_prompt("Write completed.  Press C to close file.                   ");
      }
      CONDITIONS: KEYBOARD "cC"
      ACTIONS:
      {
       if(fclose(stream_ptr) != 0)
          display_prompt("Either file is already closed, or close cannot be executed.  ");
       else
          display_prompt("File closed.                                               ");
      }
```

fputc

Synopsis

```
#include <stdio.h>
extern int fputc(character, stream_ptr);
int character;
FILE * stream_ptr;
```

Description

This routine writes a given character (cast to an *unsigned char*) to an output stream. The file-position indicator advances one character.

Inputs

The first parameter is the character to be written to the output stream. It may be given as a hexadecimal, octal, or decimal constant; as an alphanumeric constant inside single quotes; or as a variable. A hexadecimal value must be preceded by the prefix 0x or 0X; an octal value must be preceded by the prefix 0. If no prefix appears before the input, the number is assumed to be decimal.

The second parameter is the stream pointer.

Returns

If the character is successfully written to the output stream, the routine returns that character. If a write error occurs, EOF is returned and the error indicator is set.

Example

Open the named file. If the file does not already exist, create it. If it does exist, truncate its length to zero, thereby deleting its contents. Put the character read from the input stream pointed to by *read_stream* into the output stream pointed to by *write_stream*. This example is similar to the one given for *fwrite*, except that in this case, each time the P key is pressed, only one character is copied, rather than the entire file.

```
{
 #include <stdio.h>
 FILE * read_stream;
 FILE * write_stream;
 int character;
}
LAYER: 1
     STATE: copy_a_character
        CONDITIONS: ENTER_STATE
        ACTIONS: PROMPT "Press O to open files.                                        "
        CONDITIONS: KEYBOARD "oO"
        ACTIONS:
        {
         if((read_stream = fopen("FD2/usr/buff01", "r")) == 0)
            {
            display_prompt("Cannot open buff01.  ");
            pos_cursor(0,21);
            }
         else
            {
            display_prompt("Buff01 opened.  ");
            pos_cursor(0,16);
            }
         if((write_stream = fopen("FD2/usr/buff01_copy", "w")) == 0)
            displays("Cannot open buff01_copy.                                      ");
         else
            displays("Buff01_copy opened.  Press P to copy a character.             ");
        }
        CONDITIONS: KEYBOARD "pP"
        ACTIONS:
        {
         character = fgetc(read_stream);
         if(character == EOF)
            {
            if(feof(read_stream) != 0)
              display_prompt("EOF encountered.  Press C to close files.                 ");
            else
              display_prompt("Read error.  Press C to close files.                      ");
            }
```

```
else
   fputc(character, write_stream);
}
CONDITIONS: KEYBOARD "cC"
ACTIONS:
{
if(fclose(read_stream) != 0)
   {
   display_prompt("Either buff01 is already closed, or close cannot be executed. ");
   pos_cursor(0,0);
   }
else
   {
   display_prompt("Buff01 closed. ");
   pos_cursor(0,16);
   }
if(fclose(write_stream) != 0)
   displayf("Either buff01_copy is already closed, or close cannot be executed.    ");
else
   display("Buff01_copy closed.                                                    ");
}
```

fprintf

Synopsis

```
#include <stdio.h>
extern int fprintf(stream_ptr, format_ptr, ...);
FILE * stream_ptr;
char * format_ptr;
```

Description

The *fprintf* routine is similar to the *sprintf* routine, except that *fprintf* writes output to an output *stream*, while *sprintf* writes output to an *array*. The output is under control of the string pointed to by *format_ptr* that specifies how subsequent arguments are converted for output. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but otherwise ignored. The *fprintf* routine returns when the end of the format string is encountered. (*Sprintf* is documented in Section 64.3.)

Inputs

The first parameter is the stream pointer.

The second parameter points to the format string composed of zero or more directives: ordinary characters (not %), which are copied unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments. Each conversion specification is introduced by the character %. After the %, the following appear in sequence:

- Zero or more *flags* that modify the meaning of the conversion specification. The flag characters and their meanings are:

  - The result of the conversion will be left-justified within the field.

  + The result of a signed conversion will always begin with a plus or minus sign.

  space If the first character of a signed conversion is not a sign, a space will be prepended to the result. If the *space* and + flags both appear, the *space* flag will be ignored.

  # The result is to be converted to an "alternate form." For d, i, c, and s conversions, the flag has no effect. For o conversion, it increases the precision to force the first digit of the result to be a zero. For x (or X) conversion, a nonzero result will have 0x (or 0X) prepended to it.

- An optional decimal integer specifying a minimum *field width*. If the converted value has fewer characters than the field width, it will be padded on the left (or right, if the left adjustment flag, described above, has been given) to the field width. The padding is with spaces unless the field width integer starts with a zero, in which case the padding is with zeros.

- An optional *precision* that gives the minimum number of digits to appear for the d, i, o, u, x, and X conversions or the maximum number of characters to be written from an array in an s conversion. The precision takes the form of a period (.) followed by an optional decimal integer; if the integer is omitted, it is treated as zero. The amount of padding specified by the precision overrides that specified by the field width.

- An optional h specifying that a following d, i, o, u, x, or X conversion specifier applies to a *short int* or *unsigned short int* argument (the argument will have been promoted according to the integral promotions, and its value shall be converted to *short int* or *unsigned short int* before printing); or an optional l specifying that a following d, i, o, u, x, or X conversion specifier applies to a *long int* or *unsigned long int* argument. If an h or l appears with any other conversion specifier, it is ignored.

- A character that specifies the type of *conversion* to be applied. (Special AR extensions have been added.) The conversion specifiers and their meanings are:

d, i, o, u, x, X

The *int* argument is converted to signed decimal (d or i), unsigned octal (o), unsigned decimal (u), or unsigned hexadecimal notation (x or X); the letters abcdef are used for x conversion and the letters ABCDEF for X conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no characters.

c The *int* argument is converted to an *unsigned char*, and the resulting character is written.

s The argument shall be a pointer to a null-terminated array of 8-bit *chars*. Characters from the string are written up to (but not including) the terminating null character: if the precision is specified, no more than that many characters are written. The string may be an array into which output was written via the *sprintf* routine.

p The argument shall be a pointer to void. The value of the pointer is converted to a sequence of printable characters, in this format: *0000:0000*. There are always exactly 4 digits to the right of the colon. The number of digits to the left of the colon is determined by the pointer's value and the precision specified. Use this conversion to display 80286 memory addresses. The 16-bit segment number will appear to the left of the colon and the 16-bit offset to the right.

% A % is written. No argument is converted.

\n Writes hexadecimal 0A, the ASCII linefeed character. No argument is converted.

If a conversion specification is invalid, the behavior is undefined.

If any argument is or points to an aggregate (except for an array of characters using *%s* conversion or any pointer using *%p* conversion), the behavior is undefined.

In no case does a nonexistent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is expanded to contain the conversion result.

Returns

This routine returns the number of characters written, or a negative value if an output error occurred.

Example

Assume the X.25 personality package has been loaded in at Layer 2. When an unknown frame is received, copy the actual value of the control byte to an output stream.

```
{
#include <stdio.h>
FILE * stream_ptr;
extern volatile const unsigned char rcvd_frame_cntrl_byte_1;
}
LAYER: 2
    STATE: save_unknowns
      CONDITIONS: ENTER_STATE
      ACTIONS: PROMPT "Press O to open file.                                  "
      CONDITIONS: ENTER_STATE
      ACTIONS:
      {
       if((stream_ptr = fopen("FD2/usr/frame_unkwn", "w")) == 0)
          display_prompt("Cannot open file.                                      ");
       else
          display_prompt("File opened.                                           ");
      }
      CONDITIONS: RCV UNKNOWN
      ACTIONS:
      {
       if(fprintf(stream_ptr, "%02x\n", rcvd_frame_cntrl_byte_1) < 0)
          display_prompt("Error in printing to stream.                           ");
       else
          display_prompt("Print to stream completed.  Press C to close file.     ");
      }
      CONDITIONS: KEYBOARD "cC"
      ACTIONS:
      {
       if(fclose(stream_ptr) != 0)
          display_prompt("Either file is already closed, or close cannot be executed.   ");
       else
          display_prompt("File closed.                                           ");
      }
```

(D) File Maintenance Routines

rename

Synopsis

```
#include <stdio.h>
extern int rename(oldfile_ptr, newfile_ptr);
const char * oldfile_ptr;
const char * newfile_ptr;
```

Description

This routine renames a specified file. A file can only be renamed if it resides on the active disk, indicated on the Current Directory line of the File Maintenance screen. Renaming an open file does not affect subsequent disk I/O operations on the stream. The stream is still associated with the same file, even though the filename has changed.

Inputs

The first parameter is a pointer to a string, the current name of the file. Give the absolute pathname of the file, prefixed by the device name (HRD, FD1, or FD2).

The second parameter is a pointer to a string, the new name to be given to the file. Give the absolute pathname of the file, prefixed by the device name.

Returns

If the rename operation succeeds, zero is returned. If it fails, a non-zero value is returned. If the renaming fails, the file will still be known by its original name.

Example

Change the name of a file from *old* to *backup*. Prompt whether or not the rename operation was successful.

```
{
 #include <stdio.h>
}
LAYER: 1
     STATE: rename
        CONDITIONS: ENTER_STATE
        ACTIONS: PROMPT "Press spacebar to rename file.                    "
        CONDITIONS: KEYBOARD " "
        ACTIONS:
        {
         if(rename("FD1/usr/old", "FD1/usr/backup") != 0)
            display_prompt("Rename failed.                          ");
         else
            display_prompt("File has been renamed.                  ");
        }
```

remove

Synopsis

```
#include <stdio.h>
extern int remove(file_ptr);
const char * file_ptr;
```

Description

This routine removes the named file from the disk. The file must be closed in order for the remove operation to succeed. Subsequent attempts to open the file will fail. Empty directories may also be removed with this routine.

Inputs

The only input is a pointer to a string, i.e., the filename. It must be the absolute pathname, prefixed by the device name (HRD, FD1, or FD2).

Returns

Zero is returned if the file is removed; non-zero if it is not (for example, the file does not exist in the specified location).

Example

Remove file *oldfile* from the */usr* directory on the disk in floppy drive 1. Prompt whether or not the remove operation was successful.

```
{
 #include <stdio.h>
}
LAYER: 1
     STATE: delete_a_file
        CONDITIONS: ENTER_STATE
        ACTIONS: PROMPT "Press D to delete file.                              "
        CONDITIONS: KEYBOARD "dD"
        ACTIONS:
        {
         if(remove("FD1/usr/oldfile") != 0)
            display_prompt("File has not been deleted.                    ");
         else
            display_prompt("File deleted.                                 ");
        }
```

mkdir

Synopsis

```
#include <stdio.h>
extern int mkdir(directory_ptr);
char * directory_ptr;
```

Description

This routine creates a directory.

Inputs

The only parameter is a pointer to a string, i.e., the name of the directory to be created. The absolute pathname must be used, prefixed by the device name (FD1, FD2, or HRD).

Returns

If the directory is created, zero is returned; otherwise, a non-zero value is returned.

Example

Create a sub-directory called *disk_i_o* in the */usr* directory on the disk in drive 2.

```
{
 #include <stdio.h>
}
LAYER: 1
     STATE: make_directory
        CONDITIONS: ENTER_STATE
        ACTIONS: PROMPT "Press M to make a directory.                    "
        CONDITIONS: KEYBOARD "mM"
        ACTIONS:
        {
         if(mkdir("FD2/usr/disk_i_o") != 0)
            display_prompt("Directory not created.                       ");
         else
            display_prompt("Directory created.                           ");
        }
```

_set_file_type

Synopsis

```
#include <stdio.h>
extern int _set_file_type(pathname_ptr, type_buff_ptr);
char * pathanme_ptr;
char * type_buff_ptr;
```

Description

This routine determines the type identification of a specified file on the File Maintenance screen. If a file is created by a "w" or "a" open mode and a file type is not specified with the *_set_file_type* routine, it will be designated as an ASCII file.

Inputs

The first parameter is a pointer to a string, the name of the file. The filename must be the absolute pathname, prefixed by the device name (HRD, FD1, or FD2).

The second parameter is a pointer to a string, the file type. The type may be any of the following (upper or lower case is acceptable):

SYS	System
DIR	Directory
PRGM	Program
SETUP	Setup
OBJ	Object code
LOBJ	Linkable-object
ASCII	ASCII
BITIM	Bit-image data
CHDAT	Character data

Returns

If the operation succeeds, the routine returns zero; otherwise, it returns a non-zero value.

Example

The following example is almost the same one used for *fwrite*: read the contents of a program file and write them to a new file. The difference is that *new_file* is set to be a program file. In the *fwrite* example, the type designation in the file directory would default to "ASCII." It would still load and run as a program file, however, since the file's contents, not its type label, determine which operations are valid.

```
{
 #include <stdio.h>
 FILE * read_stream;
 FILE * write_stream;
 char output [6091];
 size_t n;
}
LAYER: 1
     STATE: write_to_a_file
        CONDITIONS: ENTER_STATE
        ACTIONS: PROMPT "Press O to open files.                              "
        CONDITIONS: KEYBOARD "oO"
        ACTIONS:
        {
         if((read_stream = fopen("FD2/usr/buff01", "r")) == 0)
            {
            display_prompt("Cannot open buff01.  ");
            pos_cursor(0,21);
            }
         else
            {
            display_prompt("Buff01 opened.  ");
            pos_cursor(0,16);
            }
         if((write_stream = fopen("FD2/usr/new_file", "w")) == 0)
            displays("Cannot open new_file.                              ");
         else
            displays("New_file opened.  Press "sS" to set the file type.   ");
        }
        CONDITIONS: KEYBOARD "sS"
        ACTIONS:
        {
         if(_set_file_type("FD2/usr/new_file", "PRGM") != 0)
            display_prompt("File type not set.  Press R to read buff01.      ");
         else
            display_prompt("File type set.  Press R to read buff01.          ");
        }
```

```
CONDITIONS: KEYBOARD "rR"
ACTIONS:
{
 n = fread(output, 1, 6091, read_stream);
 if(n != 6091)
    display_prompt("Either a read error has occurred, or an EOF has been
         encountered.    ");
 else
    display_prompt("Press W to write to new_file.                          ");
}
CONDITIONS: KEYBOARD "wW"
ACTIONS:
{
 n = fwrite(output, 1, 6091, write_stream);
 if(n != 6091)
    display_prompt("Write error.  Press C to close files.                  ");
 else
    display_prompt("Write completed.  Press C to close files.              ");
}
CONDITIONS: KEYBOARD "cC"
ACTIONS:
{
 if(fclose(read_stream) != 0)
    {
    display_prompt("Either buff01 is already closed, or close cannot be executed.  ");
    pos_cursor(0,0);
    }
 else
    {
    display_prompt("Buff01 closed.   ");
    pos_cursor(0,16);
    }
 if(fclose(write_stream) != 0)
    displays("Either new_file is already closed, or close cannot be executed.    ");
 else
    displays("New file closed.                                                   ");
}
```

_get_file_type

Synopsis

```
#include <stdio.h>
extern int _get_file_type(pathname_ptr, type_buff_ptr);
char * pathname_ptr;
char * type_buff_ptr;
```

Description

This routine determines the type of a specified file.

Inputs

The first parameter is a pointer to a string, the name of the file. The filename must be the absolute pathname, prefixed by the device name (HRD, FD1, or FD2).

The second parameter is a pointer to an array in which the file type should be written. See *_set_file_type* for the different file types.

Returns

If the operation succeeds, the routine returns zero; otherwise, it returns a non-zero value.

Example

```
{
 #include <stdio.h>
 FILE * stream_ptr;
 char type [8];
}
LAYER: 1
     STATE: find_type
        CONDITIONS: ENTER_STATE
        ACTIONS: PROMPT "Press G to get file type.                              "
        CONDITIONS: KEYBOARD "gG"
        ACTIONS:
        {
         if(_get_file_type("FD2/usr/new_file", &type[0]) != 0)
            display_prompt("File type not found.                                   ");
         else
            displayf("File type=%s.                                            ", type);
        }
```

66 Status

The structures and variables referenced in this section provide information about the current status of the programmer's INTERVIEW. This information must be accessed via C coding on the Protocol Spreadsheet since these structures and variables have no softkey equivalents.

66.1 Unit Configuration

Two structures presented in Table 66-1 may be accessed by the user to identify current features of the INTERVIEW. *unit_setup* variables reflect current Line Setup menu and FEB tick-rate selections. *unit_config* variables contain information about the user's INTERVIEW hardware and software.

66.2 Current Display Mode

The variables *display_screen_changed*, *crnt_display_screen*, and *prev_display_screen* track movement via softkey from one display screen to another. These variables also indicate transitions between Run mode and Freeze mode. They are documented in Section 61.1.

Table 66-1
Status Structures

Type	Variable	Value (hex/decimal)	Meaning
Structure Name: unit_setup			Structure containing Line Setup and FEB tick-rate selections. Declared as type *extern struct*. Reference member variables of the structure as follows: *unit_setup.speed_dce.*
unsigned long	speed_dce		If *Clock Source* selection is *Internal*, this variable has *Speed* value entered on Line Setup. If *Clock Source* is *External*, this variable has *DCE* speed indicated under *Clock Source: Internal Split.*
unsigned long	speed_dte		If *Clock Source* selection is *Internal*, this variable has *Speed* value entered on Line Setup. If *Clock Source* is *External*, this variable has *DTE* speed indicated under *Clock Source: Internal Split.*
unsigned long	usec_per_tick		tick rate selected on FEB Setup
		a/10	10 usec
		64/100	100 usec
		3e8/1000	1 msec
		2710/10000	10 msec
		186a0/100000	1000 msec
		f4240/1000000	1 sec
unsigned char	bit_order_polarity	0	normal
		1	normal-inverse
		2	reverse-normal
		3	reverse-inverse
unsigned char	bits_per_byte	*5-8*	
unsigned char	clocking_type	0	internal
		1	external
		2	internal-split
unsigned char	data_source	0	disk
		1	line
unsigned char	format	0	sync
		1	bop
		2	async
		3	isoc
unsigned char	mode	0	automonitor
		1	monitor
		2	emulate dce
		3	emulate dte
unsigned char	parity	0	none
		1	even
		2	odd
		3	mark
		4	space
unsigned char	code_name [13]		ASCII, EBCDIC, etc.

Table 66-1 (continued)

Type	Variable	Value (hex/decimal)	Meaning
Structure Name: unit_config			Structure containing unit configuration. Declared as type *extern struct*. Reference member variables of the structure as follows: *unit_config.floppy_exists_mask.*
unsigned char	floppy_exists_mask	1 2	floppy1 floppy2†
unsigned char	hard_disk	0 1	not present present
unsigned char	test_board	0 1	not present present
unsigned char	mux	0 1	not present present
unsigned char	modem	0 1	not present present
unsigned char	num_mpms	*0–4*	number of MPM boards present
struct mpm_info	mpm [4]		array of structures. Each element in the array is an instance of the structure *mpm_info* and corresponds to one of four MPM boards which may be present. Reference member variables of the structure elements in the array as follows: *unit_config.mpm[0].present.*
unsigned char	cpm_rev		reserved
unsigned char	gbm_rev		reserved
unsigned char	pcm_rev		reserved
unsigned char	modem_rev		reserved
unsigned char	mux_rev		reserved
unsigned char	tim_type	f0/240 f1/241 f2/242 f3/243 f4/244 f5/245 *f6–fb/246–251* fc/252 fd/253 fe/254 ff/255	RS-232 X.21 V.35 RS-449 expansion adaptor RC-8245 reserved ISDN G.703 T1 none
unsigned long	last_ram_cpm		the value of this variable plus one yields the CPM memory size (in bytes)

(unit_config continued on next page)

† If *(unit_config.floppy_exists_mask & value) == value*, the drive is present.
For example, if *(unit_config.floppy_exists_mask & 2) == 2*, floppy drive 2 is present.

Table 66-1 (continued)

Type	Variable	Value (hex/decimal)	Meaning
unsigned long	self_test_errors	*(mask)*	self-test errors encountered during power-up††
		1	CPM DRAM error
		2	CPM 32-bit counter
		4	CPM System Timing Controller (9513a)
		8	CPM DMAC
		10/16	MPM0 DRAM (tested from CPM-global bus)
		20/32	MPM0 DRAM (tested from MPM0)
		40/64	MPM0 interrupt latch
		80/128	unused
		100/256	MPM1 DRAM (tested from CPM-global bus)
		200/512	MPM1 DRAM (tested from MPM1)
		400/1024	MPM1 interrupt latch
		800/2048	unused
		1000/4096	MPM3 DRAM (tested from CPM-global bus)
		2000/8192	MPM3 DRAM (tested from MPM3)
		4000/16384	MPM3 interrupt latch
		8000/32768	unused
		10000/65536	unused
		20000/131072	unused
		40000/262144	unused
		80000/524288	unused
		100000/1048576	unused
		200000/2097152	unused
		400000/4194304	unused
		800000/8388608	unused
		1000000/16777216	unused
		2000000/33554432	unused
		4000000/67108864	unused
		8000000/134217728	unused
		10000000/268435456	unused
		20000000/536870912	unused
		40000000/1073741824	unused
		80000000/2147483648	unused
unsigned long	version	6	current value for this version of *unit_config* structure
unsigned long	model_number	19c8/6600	INTERVIEW 6600
		1a90/6800	INTERVIEW 6800 *TURBO*
		1b58/7000	INTERVIEW 7000
		1c20/7200	INTERVIEW 7200 *TURBO*
		1d4c/7500	INTERVIEW 7500
		1e14/7700	INTERVIEW 7700 *TURBO*

(unit_config continued on next page)

†† If *(unit_config.self_test_errors & mask) == mask*, the error is present.
If *(unit_config.self_test_errors & 0xffffffff) == 0*, no errors encountered during power-up.

Table 66-1 (continued)

Type	Variable	Value (hex/decimal)	Meaning
unsigned char	feb_type	0	original version
		1	version with increased speed of software and faster access to ticks from FEB
		2	version which supports high-speed RAM recording, specifically aggregate T1 or G.703 data capture
		3	version which also supports INTERVIEW 7200 and 7700 *TURBO*s
unsigned char	is_turbo	0	unit is not *TURBO*
		1	unit it *TURBO*
unsigned char	xdram_rev_num		XDRAM revision number
unsigned char	xdram_present	0	XDRAM board is not present
		1	XDRAM board is present
unsigned long	xdram_lo_addr		low end of memory range
unsigned long	xdram_hi_addr		high end of memory range
unsigned long	spare1		reserved/undefined
unsigned long	spare2		reserved/undefined
unsigned long	spare3		reserved/undefined
unsigned long	spare4		reserved/undefined
unsigned long	spare5		reserved/undefined
unsigned long	spare6		reserved/undefined
unsigned long	spare7		reserved/undefined
unsigned long	spare8		reserved/undefined
unsigned long	spare9		reserved/undefined
unsigned long	sw_version		software version†††
unsigned long	fw_version		firmware version†††
<u>**Structure Name:**</u> **mpm_info**			Structure containing information on specific MPM board. Instance of this structure for each MPM board is contained in array named *unit_config.mpm*. Declared as type *extern struct*.
unsigned char	rev_num		MPM revision number
unsigned char	present	0	specific MPM board (of four) not present
		1	specific MPM board (of four) present
unsigned long	lo_addr		low end of memory range
unsigned long	hi_addr		high end of memory range

†††To display the software version in the same format presented on the main menu screen, 5.00 for example, use the following format in a call to *displayf* (or *tracef*):

displayf(" %lu.%02lu%c", ((unit_config.sw_version >> 8)/100), ((unit_config.sw_version >> 8)%100), (char)(unit_config.sw_version & 0xff));

The same format may be used for presentation of the firmware version.

67 Remote Port I/O

The REMOTE RS-232 port is a "spare" serial interface through which the programmer may communicate with other equipment. The remote port is located at the rear of the INTERVIEW next to the printer port. (The REMOTE LED on the front panel of the INTERVIEW is related to remote control of the unit, unimplemented at this time.)

Remote-port functions must be coded in C regions on the Protocol Spreadsheet. There are no spreadsheet-token equivalents of the C variables and routines described in this section. Use these variables and routines in either emulate or monitor mode to transmit and receive data through the remote port.

The remote-communications process on the CPM controls the flow of data between the user's program and the remote port. When data is received through the remote port, this process temporarily buffers it in a 2048-byte input queue. The user's program makes requests for data from the input queue via the *rmt_getc*, *rmt_getl*, and *rmt_gets* input routines discussed below. When the remote-communications process receives a request, it removes data from the queue and passes it to the task. If there are no outstanding requests at the time data is received, it is discarded from the input queue—i.e., data received between requests cannot be retrieved. This is the default condition of the input queue.

To "lock" all received characters in the input queue, call *rmt_lock*. When the input queue is locked, the remote-communications process removes data only when 1) a user task has requested data via the *rmt_getc*, *rmt_getl*, or *rmt_gets* routine, 2) the input queue is full and some data must be discarded in order for incoming data to be buffered, or 3) *rmt_flushi* is executed. "Unlock" the input queue with *rmt_unlock*. *rmt_unlock*, *rmt_flushi*, and *rmt_flusho* are automatically executed whenever the INTERVIEW returns to Program mode.

> **NOTE:** Although requests to receive (or transmit) data from more than one task will be queued by the remote-communications process, a single task can have only one such request outstanding at a time.

Similarly, when the programmer wants to send data out the remote port, he calls *rmt_putc*, *rmt_puts*, or *rmt_putb*. The remote-communications process temporarily places these requests in an output queue before transmitting them through the remote port.

67.1 Structures

There are no structures associated exclusively with remote functions.

67.2 Variables

Table 67-1 lists the event variables specific to remote port I/O operations. Use most of these variables to detect changes in the status of the input and output queues.

As data is received through the remote port, the remote-communications process temporarily stores it in the input queue. Use *rmt_input_not_empty*, *rmt_input_almost_full*, and *rmt_input_overflow* to monitor how full the input queue is. When the input queue is "almost full," incoming data must be stopped in order to prevent the queue from overflowing.

rmt_input_almost_empty and *rmt_input_empty* are significant events as the remote. communications process takes data out of the input queue. These events indicate that that the input queue is ready to accept more data.

Table 67-1
Remote Port I/O Variables

Type	Variable	Value (hex/decimal)	Meaning
extern event	rmt_break		True when a break (NULL with a framing error) is received through the remote port. Line Setup configured for emulate or monitor mode.
extern event	rmt_input_not_empty		True when remote input-queue transitions from empty to not empty. Beginning to receive characters. Line Setup configured for emulate or monitor mode.
extern event	rmt_input_almost_full		True when the remote input-queue transitions from less than 3/4 full to 3/4 full as data is being put into the queue. Line Setup configured for emulate or monitor mode.
extern event	rmt_input_overflow		True when remote input-queue transitions from not full to full. At this point, the oldest existing data in the queue is discarded to make room for new data coming in the remote port. Line Setup configured for emulate or monitor mode.
extern event	rmt_input_almost_empty		True when the remote input-queue transitions from more than 1/4 full to 1/4 full as data is being taken out of the queue. Line Setup configured for emulate or monitor mode.
extern event	rmt_input_empty		True when remote input-queue transitions from not empty to empty. All characters have been read or discarded. Line Setup configured for emulate or monitor mode.
extern event	rmt_output_empty		True when remote output-queue transitions from not empty to empty. All data output to the remote port has been transmitted. Line Setup configured for emulate or monitor mode.

67.3 Routines

Remote routines fall into three categories. *Input* routines are used to read data received from the remote port. Use *output* routines to transmit data through the remote port. The last category of routines reads or sets *parameters* for the remote port.

(A) Input Routines

Use *rmt_getc*, *rmt_getl*, and *rmt_gets* to read data received through the remote port. Use *rmt_lock* and *rmt_unlock* to control the flow of data from the input queue.

rmt_getc

Synopsis

```
extern int rmt_getc(wait);
int wait;
```

Description

The *rmt_getc* routine reads the next character (if present) from the remote port.

Inputs

If no character is available from the input queue when *rmt_getc* is called, this parameter determines when the routine will return. A non-zero value for this parameter means wait for a character to become available before returning. If another task has already requested data from the queue, this request will be queued.

When the value is zero, the routine will return without a character if none is available. If there is already an outstanding request from another task, a zero value also causes the remote-communications process to return from the routine without checking the input queue.

> **NOTE:** More than one test (task) may request data from the input queue. The remote-communications processes queues these requests as they are made. To ensure that requests are processed in turn, use this "wait" parameter consistently across tests. If you set the parameter in a call to *rmt_getc* (or *rmt_gets*) in one test, do the same in all tests.

Returns

If a character is present in the input queue, this routine returns the character (as an *int*) read. If no character is present and the routine's "wait" parameter is zero, a -1 will be returned. When the parameter is zero, a -1 also will be returned if there is already an outstanding request from another task.

Example

In the following example, the routine will not wait for a character to become available in the remote port before returning. Each time the C key is pressed, the next character, if present, will be displayed. If a -1 is returned instead of a character, a message to that effect will be displayed on the prompt line.

```
LAYER: 1
    STATE: get_next_character
        CONDITIONS: ENTER_STATE
        ACTIONS:
        {
         display_prompt("Press C to get next character.                              ");
         rmt_lock();
        }
        CONDITIONS: KEYBOARD "cC"
        ACTIONS:
        {
         int character;
         character = rmt_getc(0);
         if(character == -1)
            display_prompt("No character available.                                  ");
         else
            displayf("%c", character);
        }
```

rmt_getl

Synopsis

extern int rmt_getl(string_ptr, max_length);
*char * string_ptr;*
int max_length;

Description

rmt_getl reads from the remote port one line at a time. This routine gets at the most the specified number of characters from the remote port and puts them in an array. Unless a carriage return or linefeed is encountered, the routine will not return until the specified number of characters has been read. A carriage return or linefeed causes the routine to return, even if the specified number of characters has not yet been read. The carriage return or linefeed will be replaced by a terminating NULL character in the array.

Inputs

The first parameter is a pointer to the array into which the characters will be put.

The second parameter is the maximum number of characters to be read.

Returns

This routine returns the number of characters (preceding the terminating NULL) read into the array.

Example

Each time the L key is pressed, twenty characters, at the most, will be read from the remote port, put into an array called *data*, and displayed on the screen.

```
LAYER: 1
    STATE: read_line
       CONDITIONS: ENTER_STATE
       ACTIONS:
       {
        display_prompt("Press L to get next line.                                   ");
        rmt_lock();
       }
       CONDITIONS: KEYBOARD "IL"
       ACTIONS:
       {
        int number;
        unsigned char data [25];
        number = rmt_getl(data, 20);
        displayf("\n%u characters read:\n%.20s\n", number, data);
       }
```

rmt_gets

Synopsis

```
extern int rmt_gets(string_ptr, length, wait);
char * string_ptr;
int length;
int wait;
```

Description

Similar to *rmt_getl*, this routine gets a specified number of characters from the remote port and puts them in an array. Unlike *rmt_getl*, characters continue to be read even if a carriage return or linefeed is encountered. The array is not NULL-terminated.

Inputs

The first parameter is a pointer to the array into which the characters will be put.

The second parameter is the number of characters to be read.

If the specified number of characters is not available from the input queue when *rmt_getl* is called, the third parameter determines when the routine will return. A non-zero value for this parameter means wait for the specified number of characters to become available before returning. If another task has already requested data from the queue, this request will be queued.

When the value is zero, the routine will return with less than the specified number of characters if all are not available. If there is already an outstanding request from another task, a zero value also causes the remote-communications process to return from the routine without checking the input queue.

> **NOTE:** More than one test (task) may request data from the input queue. The remote-communications processes queues these requests as they are made. To ensure that requests are processed in turn, use this "wait" parameter consistently across tests. If you set the parameter in a call to *rmt_gets* (or *rmt_getc*) in one test, do the same in all tests.

Returns

This routine returns the number of characters read from the remote port.

Example

When the Ⓢ key is pressed, 4000 characters will be read from the remote port, put into an array called *data*, displayed on the screen (until a NULL is encountered—see *%s* in *tracef* routine, Section 61), and written to a file named *echo_time*. This is the program that might be run to receive the file transmitted in the *rmt_putb* example.

```
{
#define FILE_LENGTH 4000
#define FILENAME "FD1/usr/echo_time"
#include <stdio.h>
#include <trace_buf.h>
extern struct trace_buf l1_trbuf;
FILE * stream_ptr;
size_t n;
unsigned char data [FILE_LENGTH];
int count;
}
```

```
LAYER: 1
    STATE: get_string
      CONDITIONS: ENTER_STATE
      ACTIONS:
      {
       rmt_lock();
       if((stream_ptr = fopen(FILENAME, "w")) == 0)
          display_prompt("Cannot open file.");
       else
         {
          display_prompt("Press S to read string.");
          pos_cursor(1,0);
         }
      }
      CONDITIONS: KEYBOARD "sS"
      ACTIONS:
      {
       count = rmt_gets(data, FILE_LENGTH, 1);
       if(count != FILE_LENGTH)
          displayf("Could not read entire string.\n");
       tracef(&l1_trbuf, "%d characters read: \n%s\n\n", count, data);
       n = fwrite(data, 1, FILE_LENGTH, stream_ptr);
       if(n != FILE_LENGTH)
          displayf("A write error has occurred.\n");
       else
          displayf("File written. \n");
       if(fclose(stream_ptr) != 0)
          displayf("Either file is already closed, or close cannot be executed. \n");
       else
          displayf("File closed.\n");
      }
```

rmt_flushi

Synopsis

extern int rmt_flushi();

Description

If characters have been received in the input queue, but have not been read yet, this routine causes them to be discarded. Whenever the INTERVIEW enters or leaves Run mode, *rmt_flushi* is automatically executed. This ensures that the input queue is empty.

> NOTE: A call to any of the routines which *set* the parameters of the remote port also causes *rmt_flushi* to be executed automatically. The routines which only *get* the current parameters of the remote port have no effect on the input queue.

When the programmer calls *rmt_flushi*, requests for data from the input queue will be processed before the input queue is flushed. When a call to *rmt_flushi* is made from another test, however, input routines waiting for characters from the input queue will be returned.

Returns

rmt_flushi returns a zero when the input queue is flushed successfully. Otherwise, it returns a non-zero value.

Example

This example is the same as that for *rmt_getc*. Notice that as the program enters the first state, the input queue is flushed.

```
LAYER: 1
    STATE: get_next_character
      CONDITIONS: ENTER_STATE
      ACTIONS:
      {
       display_prompt("Press C to get next character.                    ");
       rmt_lock();
       rmt_flushi();
      }
      CONDITIONS: KEYBOARD "cC"
      ACTIONS:
      {
       int character;
       character = rmt_getc(0);
       if(character ==-1)
          display_prompt("No character available.                        ");
       else
          displayf("%c", character);
      }
```

rmt_lock

Synopsis

extern void rmt_lock();

Description

Recall that in its default state, the input queue does not retain characters received through the remote port between requests from user tasks. Data in the queue must either be passed to a user task or be discarded. The *rmt_lock* routine "locks" all received characters in the input queue until they are requested. (Refer again to the beginning of this section.)

Example

The following example is the same as the one for the *rmt_getl* routine. Notice that a call to *rmt_lock* is made as the program begins. The operator makes a request for data from the input queue by pressing [L]. The next line of data in the input queue will be removed and put in the array named *data*.

```
LAYER: 1
    STATE: read_line
       CONDITIONS: ENTER_STATE
       ACTIONS:
       {
        display_prompt("Press L to get next line.                                  ");
        rmt_lock();
       }
       CONDITIONS: KEYBOARD "IL"
       ACTIONS:
       {
        int number;
        unsigned char data [25];
        number = rmt_getl(data, 20);
        displayf("\n%u characters read:\n%.20s\n", number, data);
       }
```

rmt_unlock

Synopsis

extern void rmt_unlock();

Description

The *rmt_unlock* routine implements the inverse of the *rmt_lock* routine. If characters are received in the remote port and there are no outstanding requests for data, the remote-communications process will discard the characters. (Refer also to *rmt_lock* and to the beginning of this section.)

rmt_unlock is automatically executed when the INTERVIEW returns to Program mode.

Example

In the following example, the input queue is locked as soon as the program begins. It will remain locked until the operator press U (or PROGRAM).

```
LAYER: 1
    STATE: read_line
       CONDITIONS: ENTER_STATE
       ACTIONS:
       {
        display_prompt("Press L to get next line.                                  ");
        rmt_lock();
       }
```

```
CONDITIONS: KEYBOARD "lL"
ACTIONS:
{
 int number;
 unsigned char data [25];
 number = rmt_getl(data, 20);
 displayf("\n%u characters read:\n%.20s\n", number, data);
}
CONDITIONS: KEYBOARD "uU"
ACTIONS:
{
 rmt_unlock();
}
```

(B) Output Routines

Use the following routines to transmit data through the remote port.

rmt_putc

Synopsis

```
extern int rmt_putc(character, wait);
unsigned char character;
int wait;
```

Description

This routine sends a specified character to the output queue of the remote port for transmission.

Inputs

The first parameter is the character to be transmitted. It may be given as a hexadecimal, octal, or decimal constant; as an alphanumeric constant inside single quotes; or as a variable. A hexadecimal value must be preceded by the prefix 0x or 0X; an octal value must be preceded by the prefix 0. If no prefix appears before the input, the number is assumed to be decimal.

If space in the output queue is not available for the character when *rmt_putc* is called, the second parameter determines when the routine will return. A non-zero value for this parameter means wait for space in the output queue to become available and return zero when the character is in the queue. If there is already a request from another task, this request will be queued.

When the value is zero and space in the output queue is not available, the routine will return -1. The character will not be in the queue. If another task is already waiting for access to the output queue, a zero value also causes the remote-communications process to return from the routine without checking for available space in the output queue.

NOTE: More than one test (task) may request to send data to the output queue. The remote-communications processes queues these requests as they are made. To ensure that requests to output data are processed in turn, use this "wait" parameter consistently across tests. If you set the parameter in a call to *rmt_putc* (*rmt_puts* or *rmt_putb*) in one test, do the same in all tests.

Returns

If the character is successfully written to the output queue, the routine returns zero. If no space is available in the output queue and the routine's "wait" parameter is zero, a −1 will be returned. When the parameter is zero, a −1 also will be returned if another task is already waiting for access to the output queue.

Example

In the following example, the next character in a fox message will be sent to the output queue of the remote port each time the operator presses Ⓒ. As a character is successfully queued, it will be displayed in the Display Window. If no space is available in the output queue for the character, −1 will be returned and a message to that effect will be displayed on the prompt line. No more characters will be sent.

```
{
 unsigned char data [] = "((FOX)) CR";
 unsigned char character;
 int i, length, error;
}
LAYER: 1
     STATE: transmit_characters
        CONDITIONS: ENTER_STATE
        ACTIONS:
        {
         display_prompt("Press C to transmit character.                              ");
         length = sizeof(data) - 1;
        }
        CONDITIONS: KEYBOARD "cC"
        ACTIONS:
        {
         for(i = 0; i < length; i++)
            {
             character = data[i];
             error = rmt_putc(character, 0);
             if(error == -1)
                  display_prompt("No space available in output queue.           ");
             else
                  displayf("%c", character);
            }
        }
```

rmt_puts

Synopsis

extern int rmt_puts(string_ptr, wait);
*const char * string_ptr;*
int wait;

Description

This routine outputs a NULL-terminated string to the output queue of the remote port.

Inputs

The first parameter is a pointer to the string to be transmitted.

If space in the output queue is not available for the string when *rmt_puts* is called, the second parameter determines when the routine will return. A non-zero value for this parameter means wait for space in the output queue to become available and return when the string is in the queue. If there is already a request from another task, this request will be queued.

When the value is zero and space is not available in the output queue, the routine returns the number of characters, if any, put into the queue. If another task is already waiting for access to the output queue, a zero value also causes the remote-communications process to return from the routine without checking for available space in the output queue.

> **NOTE:** More than one test (task) may request to send data to the output queue. The remote-communications processes queues these requests as they are made. To ensure that requests to output data are processed in turn, use this "wait" parameter consistently across tests. If you set the parameter in a call to *rmt_puts* (*rmt_putc* or *rmt_putb*) in one test, do the same in all tests.

Returns

This routine returns the number of characters put into the output queue.

Example

The following example is similar to the one given for *rmt_putc*. When the [S] key is pressed, the fox message will be sent to the remote port. The difference is that the message will be output to the remote port as a string (rather than character by character). If the output queue is full, the routine will not wait for space to become available before returning. The number of characters successfully queued will be displayed in the Display Window. If the number of characters queued is less than the length of the string, a message to that effect will be displayed on the prompt line.

```
{
 unsigned char data [] = "((FOX))␍";
 int count, length;
}
LAYER: 1
     STATE: transmit_string
        CONDITIONS: ENTER_STATE
        ACTIONS:
        {
         display_prompt("Press S to transmit string.                                   ");
         length = sizeof(data) - 1;
        }
        CONDITIONS: KEYBOARD "sS"
        ACTIONS:
        {
         count = rmt_puts(data, 0);
         if(count !=length)
            display_prompt("Could not output entire string.                            ");
         pos_cursor(1,0);
         displayf("%d characters transmitted.", count);
        }
```

rmt_putb

Synopsis

```
extern int rmt_putb(string_ptr, length, wait);
const char * string_ptr;
int length;
int wait;
```

Description

This routine sends a string of specified length to the output queue of the remote port.

Inputs

The first parameter indicates the string to be output.

The second parameter is the length of the string to be output.

If space in the output queue is not available for the string when *rmt_putb* is called, the third parameter determines when the routine will return. A non-zero value for this parameter means wait for space in the output queue to become available and return when all characters in the string have been queued. If another task is already waiting for access to the output queue, this request will be queued.

When the value is zero and space is not available in the output queue, the routine returns the number of characters, if any, put into the queue. If there is already an outstanding request from another task, a zero value also causes the remote-communications process to return from the routine without checking for available space in the output queue.

> **NOTE:** More than one test (task) may request to send data to the output queue. The remote-communications processes queues these requests as they are made. To ensure that requests to output data are processed in turn, use this "wait" parameter consistently across tests. If you set the parameter in a call to *rmt_putb* (*rmt_putc* or *rmt_puts*) in one test, do the same in all tests.

Returns

This routine returns the number of characters put into the output queue.

Example

This is the program that might be run to transmit the file received in the *rmt_gets* example. The user specifies the filename and its size (shown in the directory listing on the File Maintenance screen) in the two *#define* preprocessor directives at the beginning of the program. When the program begins, the contents of the file named *echo_time* will be read into an array called *data*. When the operator presses the [T] key, the contents of the array will be transmitted and displayed.

```
{
#define FILE_LENGTH 4000
#define FILENAME "FD1/usr/echo_time"
#include <stdio.h>
#include <trace_buf.h>
extern struct trace_buf l1_trbuf;
FILE * stream_ptr;
size_t n;
unsigned char data [FILE_LENGTH];
unsigned char size [FILE_LENGTH+100];
int count;
}
LAYER: 1
    STATE: transmit_string
        CONDITIONS: ENTER_STATE
        ACTIONS:
        {
         if((stream_ptr = fopen(FILENAME, "r")) == 0)
            display_prompt("Cannot open file.");
```

```
else
  {
   pos_cursor(1,0);
   n = fread(data, 1, FILE_LENGTH, stream_ptr);
   if(n != FILE_LENGTH)
        displayf("Either a read error has occurred, or an EOF has been
             encountered.\n");
   if(fclose(stream_ptr) != 0)
        displayf("Either file is already closed, or close cannot be executed. \n");
   else
        displayf("File closed.\n");
   if(n == FILE_LENGTH)
        display_prompt("Press T to transmit characters.");
  }
}
CONDITIONS: KEYBOARD "tT"
ACTIONS:
{
 count = rmt_putb(data, FILE_LENGTH, 1);
 if(count != FILE_LENGTH)
    displayf("Could not output entire string.\n");
 sprintf(size, "%d characters transmitted: %%.%dH", count, count);
 tracef(&l1_trbuf, size, data);
 tracef(&l1_trbuf, "\n\n");
}
```

rmt_flusho

Synopsis

extern int rmt_flusho();

Description

If characters are queued to be output from the remote port, but have not been transmitted yet, this routine causes them to be discarded. This ensures that anything previously in the output queue port will be deleted.

rmt_flusho is automatically executed when the INTERVIEW returns to Program mode.

> **NOTE:** A call to any of the routines which *set* the parameters of the remote port causes *rmt_flusho* to be executed automatically. The routines which only *get* the current parameters of the remote port have no effect on the output queue.

Returns

rmt_flusho returns a zero when the output queue is flushed successfully. Otherwise, it returns a non-zero value.

Example

This example is the same as that for *rmt_putc*. Notice that as the program enters the first state, the output queue is flushed.

```
{
 unsigned char data [] = "《FOX》";
 unsigned char character'
 int i, length, error;
}
LAYER: 1
     STATE: transmit_a_character
        CONDITIONS: ENTER_STATE
        ACTIONS:
        {
         rmt_flusho();
         display_prompt("Press C to transmit character.                                ");
         length = sizeof(data);
        }
        CONDITIONS: KEYBOARD "cC"
        ACTIONS:
        {
         for(i = 0; i < length; i++)
            {
             character = data[i];
             error = rmt_putc(character, 0);
             if(error ==-1)
                {
                 display_prompt("No space available in output queue.                      ");
                 break;
                }
             else
                  displayf("%c", character);
            }
        }
```

rmt_suspendo

Synopsis

extern int rmt_suspendo();

Description

If characters are queued to be output from the remote port, but have not been transmitted yet, this routine causes transmitting to be suspended. The output queue will *not* be flushed. Use this routine only when the remote port handshaking mode is full-duplex without flow control.

Returns

rmt_suspendo returns a zero when transmitting is successfully suspended. Otherwise, it returns a non-zero value.

Example

When the INTERVIEW receives an X-OFF as a signal to stop sending data, it will suspend transmissions from the remote port.

```
{
 extern event rmt_input_not_empty;
 int character;
}
LAYER: 1
     STATE: suspend_output
        CONDITIONS: ENTER_STATE
        ACTIONS:
        {
         rmt_lock();
        }
        CONDITIONS:
        {
         rmt_input_not_empty
        }
        ACTIONS:
        {
         character = rmt_getc(1);
         if(character == 0x13)
            rmt_suspendo();
        }
        TIMEOUT ck_input RESTART 0.001
        CONDITIONS: TIMEOUT ck_input
        ACTIONS:
        {
         character = rmt_getc(1);
         if(character == 0x13)
            rmt_suspendo();
        }
        TIMEOUT ck_input RESTART 0.001
```

rmt_resumeo

Synopsis

extern int rmt_resumeo();

Description

This routine resumes transmission of characters from the remote port. Use this routine only when the remote port handshaking mode is full-duplex without flow control.

Returns

rmt_resumeo returns a zero when transmitting is successfully resumed. Otherwise, it returns a non-zero value.

Example

When the INTERVIEW receives an X-ON as a signal to send data, it will resume transmissions from the remote port.

```
{
 int character;
}
LAYER: 1
     STATE: resume_output
        CONDITIONS: ENTER_STATE
        ACTIONS:
        {
         rmt_lock();
        }
        TIMEOUT RESTART ck_input 0.001
        CONDITIONS: TIMEOUT check_input
        ACTIONS:
        {
         character = rmt_getc(1);
         if(character == 0x11)
             rmt_resumeo();
        }
        TIMEOUT ck_input RESTART 0.001
```

rmt_send_break

Synopsis

extern int rmt_send_break(wait);
int wait;

Description

This routine causes a break, queued as other transmits, to be transmitted.

Inputs

If space in the output queue is not available for the break when *rmt_send_break* is called, the only parameter determines when the routine will return. A non-zero value for this parameter means wait for space in the output queue to become available and return zero when the break is in the queue. If there is already a request from another task, this request will be queued.

When the value is zero and space in the output queue is not available, the routine will return -1. The break will not be in the queue. If another task is already waiting for access to the output queue, a zero value also causes the remote-communications process to return from the routine without checking for available space in the output queue.

> **NOTE:** More than one test (task) may request to send data to the output queue. The remote-communications processes queues these requests as they are made. To ensure that requests to output data are processed in turn, use this "wait" parameter consistently across tests. If you set the parameter in a call to *rmt_send_break* (*rmt_putc, rmt_puts* or *rmt_putb*) in one test, do the same in all tests.

Returns

If the break is successfully written to the output queue, the routine returns zero. If no space is available in the output queue and the routine's "wait" parameter is zero, a -1 will be returned. When the parameter is zero, a -1 also will be returned if another task is already waiting for access to the output queue.

Example

In this example, a break will be transmitted each time the operator presses the space bar.

```
LAYER: 1
    STATE: transmit_break
        CONDITIONS: KEYBOARD " "
        ACTIONS:
        {
         rmt_send_break(1);
        }
```

(C) Configuration Routines

The default configuration for the remote port at boot-up is the following:

Baud rate = 1200
Bits/character = 8
Parity = None
Mode = Full-duplex

Use the first four routines discussed below to change these settings. The programmer's reconfiguration of the remote port is not affected when the INTERVIEW exits or re-enters Run mode.

A call to any of these *set* routines causes *rmt_flushi* and *rmt_flusho* to be executed automatically before the parameter is set.

Use the remaining four routines to read the current parameter-settings for the remote port. These *get* routines have no effect on the input and output queues.

rmt_set_baud_rate

Synopsis

extern int rmt_set_baud_rate(speed);
int speed;

Description

This routine sets the baud rate for the remote port. The default value at boot-up is 1200.

> NOTE: A call to *rmt_set_baud_rate* causes *rmt_flushi* and *rmt_flusho* to be executed automatically before the baud rate is set.

Inputs

The only parameter is the desired baud rate. Values that are multiples of 300 in the range 300 through 19200 are valid.

Returns

If the specified baud rate is valid and successfully set, zero is returned. If the baud rate is valid, but not successfully set, -1 is returned. For an invalid baud rate, the routine returns -2.

Example

In order for two devices to communicate with each other, they must be using the same baud rate. When they are not the same, some devices send a break as a signal for the other to adjust its baud rate. If the following example, the INTERVIEW will change the baud rate for the remote port whenever a break is received.

```
{
 extern event rmt_break;
 int error;
 int speed = 300;
}
LAYER: 1
     STATE: adjust_baud_rate
        CONDITIONS:
        {
         rmt_break
        }
```

```
ACTIONS:
{
 error = rmt_set_baud_rate(speed);
 if(error != -1)
   {
    speed *= 2;
    if(speed > 19200)
         speed = 300;
   }
 else
    displayf("Unable to set the baud rate to %d.", speed);
}
```

rmt_set_bits

Synopsis

```
extern int rmt_set_bits(value);
int value;
```

Description

This routine sets the number of bits per character for the remote port. The default setting at boot-up is 8 bits/character.

> **NOTE:** A call to *rmt_set_bits* causes *rmt_flushi* and *rmt_flusho* to be executed automatically before the number of bits/character is set.

Inputs

The only parameter is the number of bits/character. Valid values are five through eight.

Returns

If the specified number of bits/character is valid and successfully set, zero is returned. If the number is valid, but not successfully set, -1 is returned. For an invalid value, the routine returns -2.

Example

In this example, the number of bits/character for the remote port is set to 7 and displayed on the Display Window screen.

```
LAYER: 1
    STATE: set_parameters
      CONDITIONS: ENTER_STATE
      ACTIONS:
      {
       displayf("Bits = %d ", rmt_set_bits(7));
      }
```

rmt_set_parity

Synopsis

extern int rmt_set_parity(parity);
int parity;

Description

This routine sets the parity for the remote port. The default setting at boot-up is no parity.

> **NOTE:** A call to *rmt_set_parity* causes *rmt_flushi* and *rmt_flusho* to be executed automatically before the parity for the remote port is set.

Inputs

The only parameter is a value designating the desired parity. Valid values are the following: none (0), odd (1), even (2), mark (3), or space (4).

Returns

If the specified parity value is valid and successfully set, zero is returned. If the value is valid, but not successfully set, -1 is returned. For an invalid parity value, the routine returns -2.

Example

In this example, the number of bits/character for the remote port is set to 7 and parity is even. Both settings are displayed on the Display Window screen.

```
LAYER: 1
    STATE: set_parameters
        CONDITIONS: ENTER_STATE
        ACTIONS:
        {
         displayf("Bits = %d  Parity = %d ", rmt_set_bits(7), rmt_set_parity(2));
        }
```

rmt_set_mode

Synopsis

extern int rmt_set_mode(mode);
int mode;

Description

This routine sets the handshaking mode for the remote port. The default setting at boot-up is FDX with no flow control.

> **NOTE:** A call to *rmt_set_mode* causes *rmt_flushi* and *rmt_flusho* to be executed automatically before the mode for the remote port is set.

Inputs

The only parameter is a value designating the mode. Valid values are the following:

0 = Full-duplex with no flow control (FDX)
1 = Half-duplex (HDX)
2 = Full-duplex with X-ON/X-OFF characters for flow control
3 = Full-duplex with DTR and CTS EIA leads for flow control. Use a special null-modem cable for direct connections. See Figure 67-1.

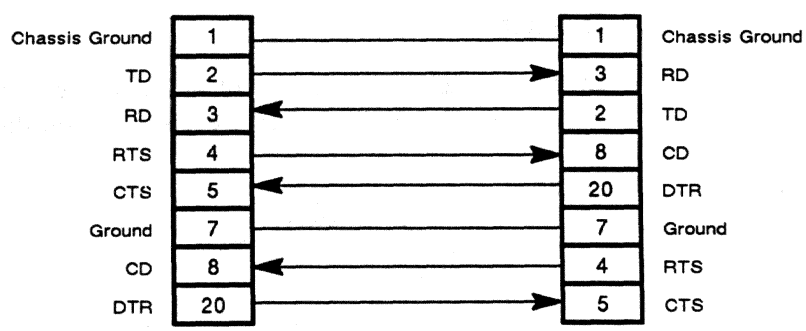


Figure 67-1 Null-modem cable connections.

Returns

If the specified mode value is valid and successfully set, zero is returned. If the value is valid, but not successfully set, −1 is returned. For an invalid mode value, the routine returns −2.

Example

In this example, the number of bits/character for the remote port is set to 7, parity is even, and the mode is set for FDX with X-ON/X-OFF. All three settings are displayed on the Display Window screen.

```
LAYER: 1
    STATE: set_parameters
      CONDITIONS: ENTER_STATE
      ACTIONS:
      {
       displayf("Bits = %d  Parity = %d  Mode = % d ", rmt_set_bits(7),
          rmt_set_parity(2), rmt_set_mode(2));
      }
```

rmt_get_baud_rate

Synopsis

extern int rmt_get_baud_rate();

Description

This routine gets the current baud-rate setting for the remote port.

Returns

The baud rate for the remote port is returned.

Example

As the program begins, the current baud-rate setting for the remote port is displayed on the Display Window screen.

```
LAYER: 1
    STATE: baud_rate
      CONDITIONS: ENTER_STATE
      ACTIONS:
      {
       displayf("Baud = %d ", rmt_get_baud_rate());
      }
```

rmt_get_bits

Synopsis

extern int rmt_get_bits();

Description

This routine tells how many bits there are per character. Possible values are five through eight.

Returns

The current number of bits per character for the remote port is returned.

Example

In this example, the current baud-rate setting and the number of bits/character for the remote port are displayed on the Display Window screen.

```
LAYER: 1
     STATE: current_parameters
        CONDITIONS: ENTER_STATE
        ACTIONS:
        {
         displayf("Baud = %d  Bits = %d ", rmt_get_baud_rate(), rmt_get_bits());
        }
```

rmt_get_parity

Synopsis

extern int rmt_get_parity();

Description

This routine gets the current parity setting for the remote port.

Returns

The current number of bits per character for the remote port is returned.

Example

In this example, the current baud-rate setting, number of bits/character, and the parity for the remote port are displayed on the Display Window screen.

```
LAYER: 1
     STATE: current_parameters
        CONDITIONS: ENTER_STATE
        ACTIONS:
        {
         displayf("Baud = %d  Bits = %d  Parity = %d ", rmt_get_baud_rate(),
             rmt_get_bits(), rmt_get_parity());
        }
```

rmt_get_mode

Synopsis

extern int rmt_get_mode();

Description

This routine gets the current handshaking mode for the remote port.

Returns

The current handshaking mode for the remote port is returned:

0 = Full–duplex with no flow control (FDX)
1 = Half–duplex (HDX)
2 = Full–duplex with X–ON/X–OFF characters for flow control
3 = Full–duplex with DTR and CTS EIA leads for flow control Requires a special null–modem cable for INTERVIEW–to–INTERVIEW direct connections. Refer to Figure 67-1.

Example

In this example, the current baud–rate setting, number of bits/character, parity, and handshaking mode for the remote port are displayed on the Display Window screen.

```
LAYER: 1
    STATE: current_parameters
      CONDITIONS: ENTER_STATE
      ACTIONS:
      {
       displayf("Baud = %d  Bits = %d  Parity = %d  Mode = %d ", rmt_get_baud_rate(),
          rmt_get_bits(), rmt_get_parity(), rmt_get_mode());
      }
```

68 AUX Port I/O

Bit Number	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Transmitter's AUX Port Lead Configuration	O	I	I	I	I	I	I	I/C	O	O	O	O	O	O	O	O
Pin Number	16	14	12	10	8	6	4	2	15	13	11	9	7	5	3	1
Bit will be used for	C	U	U	U	U	U	U	C	D	D	D	D	D	D	D	D

Bit Number	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Receiver's AUX Port Lead Configuration	I/C	I	I	I	I	I	I	O	I	I	I	I	I	I	I	I
Pin Number	16	14	12	10	8	6	4	2	15	13	11	9	7	5	3	1

O	Output/Non-control
I	Input/Non-control
I/C	Input/Control
C	Control
D	Data
U	Unassigned

Figure 68-1 Sample AUX port lead configurations for two INTERVIEWs connected by their AUX interfaces. Assume one-way data transmission (i.e., one device is controlling the other).

68 AUX Port I/O

The Auxiliary (AUX) port is a "spare" interface through which the programmer may communicate with other lab equipment. The AUX port is located at the rear of the INTERVIEW, between the printer and RGB connectors. It is controlled by a Zilog CIO (*C*ounter/Timer, Parallel *I*nput/*O*utput Unit) chip. The AUX port may be used as a serial or parallel interface. When it is operated as a parallel port, up to sixteen bits (one bit on each of sixteen leads) may be transmitted simultaneously.

AUX-port control must be coded in C regions on the Protocol Spreadsheet. There are no spreadsheet-token equivalents of the C variables and routines described in this section.

A normal configuration of equipment using the AUX port will involve two INTERVIEWs with AUX port setups that mirror each other to some extent, as in Figure 68-1. The transmitting INTERVIEW will use one of its output leads as a "strobe" to signal to the receiving INTERVIEW that an AUX word is available to be read. The receiver will detect this strobe as an *aux_change* event.

The receiving INTERVIEW will use one of its output leads to acknowledge each AUX word received. The transmitting INTERVIEW will detect this acknowledgment as an *aux_change* event.

NOTE: The AUX port is not controlled by the same CPU that handles the user program. The need for interprocessor communication without data buffering makes rapid, successive transmissions difficult to handle. It is recommended, therefore, that control bits be set aside for flow control—a bit set by the transmitter as input/control is set by the receiver as output/non-control, and vice versa—and that every output word be acknowledged before a succeeding word is output.

68.1 Variables

Table 68-1 lists the variables specific to AUX I/O operations. The fast-event variable, *aux_change,* detects a change in a lead that has been configured as a control lead. Any or all of the sixteen leads in the interface may be designated control leads. Section 68.2 explains how to configure control leads.

aux_change does not establish which control lead(s) has changed. Two associated variables, *curr_aux_value* and *prev_aux_value*, indicate the status of all sixteen leads. These are two-byte (*short*) variables. Each lead is represented by a different bit in the *short*. If the bit-value of a given lead is zero, the lead is on. If the bit-value is one, the lead is off.

Whenever a control lead changes, the value in *curr_aux_value* is written to *prev_aux_value*. Then *curr_aux_value* is updated.

Table 68-1
AUX Port I/O Variables

Type	Variable	Meaning
extern fast_event	aux_change	True when the status of a lead designated as control (and input) changes. Is automatically made to come true by the CIO chip as soon as leads have been configured via *set_aux_direction* and *set_aux_ctl_leads* routines. Therefore, condition must be tested again in a different state. Line Setup configured for emulate or monitor mode.
extern volatile const unsigned short	curr_aux_value	Each bit designates a different lead. A bit-value of one indicates a given lead is on. When value of *curr_aux_value* is exclusive *ored* (^) with *prev_aux_value*, result indicates those leads whose status has changed. Updated when *aux_change* comes true. Line Setup configured for emulate or monitor mode.
extern volatile const unsigned short	prev_aux_value	Value of previous *curr_aux_value*. Updated when control leads change, but only after logic has had a chance to compare current and previous leads. Line Setup configured for emulate or monitor mode.

68.2 Routines

In the examples for the following routines, assume that two INTERVIEW's are connected and that data flows in one direction.

> *CAUTION: You may damage the AUX interface if the same lead is designated as output on both units. We suggest that you set the leads on each unit as input/output and control/non-control* before *you connect the AUX interfaces. See Figure 68-1.*

set_aux_direction

Synopsis

```
extern void set_aux_direction(input_or_output);
unsigned short input_or_output;
```

Description

This routine designates leads on the AUX port as input or output. Designated output leads for the transmitter are set as input leads by the receiver.

Inputs

The only input is a sixteen-bit variable. Each bit in the variable designates one lead and may be set to zero (output) or one (input).

Example

Both sides of the connection may be transmitter or receiver. But for simplification in examples, let's designate only one side as the transmitter and the other as the receiver. In this example, the transmitter sets all 8 bits of the low-order byte as output bits for data, the low-order bit of the high byte as input (for handshaking), the next 6 bits of the high byte as input (unused), and the high-order bit as output (the receiver will designate this bit as input for handshaking).

```
LAYER: 1
    STATE: set_input_leads
        CONDITIONS: ENTER_STATE
        ACTIONS:
        {
         set_aux_direction(0x7f00);
        }
```

The other (receiver) INTERVIEW sets a bit as input (for handshaking). It must be one that was designated as output by the transmitter, the highest-order bit of the high byte. The data bits set as output by the transmitter must be set as input by the receiver. The receiver's *set_aux_direction* routine would look like this:

```
LAYER: 1
     STATE: set_input_leads
        CONDITIONS: ENTER_STATE
        ACTIONS:
        {
         set_aux_direction(0xfeff);
        }
```

set_aux_ctl_leads

Synopsis

extern void set_aux_ctl_leads(ctl_or_not);
unsigned short ctl_or_not;

Description

This routine determines whether or not leads will be control leads. Control leads must also be input leads, but input leads do not necessarily have to be control leads. Output leads can never be control leads.

Inputs

The only input is a sixteen-bit variable. Each bit in the variable designates one lead and may be set to zero (non-control) or one (control).

Example

Assuming the input/output bits set in the previous example, the transmitter sets all 8 data bits (output) as non-control, the low-order input bit of the high byte as control (for handshaking), the next 6 input bits of the high byte as non-control (unused), and the high-order output bit as non-control (the receiver will designate this bit as control for handshaking).

```
LAYER: 1
     STATE: set_control_leads
        CONDITIONS: ENTER_STATE
        ACTIONS:
        {
         set_aux_ctl_leads(0x0100);
        }
```

The "receiver" INTERVIEW sets one input bit as control for handshaking purposes. It must be one that was designated as output by the transmitter, the highest-order bit of the high byte. The receiver's *set_aux_ctl_leads* routine would look like this:

```
LAYER: 1
    STATE: set_control_leads
       CONDITIONS: ENTER_STATE
       ACTIONS:
       {
        set_aux_ctl_leads(0x8000);
       }
```

write_aux

Synopsis

extern void write_aux(output_word);
unsigned short output_word;

Description

This routine sends a combination of data, control, and (perhaps) unused bits as output. Input bits are not transmitted by the CIO.

Inputs

The only input is a sixteen-bit variable. Each bit designates one lead and may represent data or control information, or be unused. If a given lead was designated as a control lead, it is an input lead and the CIO will not transmit the status of the bit in any case, so its setting of 1 or 0 does not matter. If the lead was designated as a non-control lead, it might contain data, be unused, or contain an alternating value to indicate acknowledgment (if the other side designated it as a control lead).

Example

The transmitting INTERVIEW is going to send data to the receiving INTERVIEW. Before the next transmission can be sent, an acknowledgment must be received. The acknowledgment is detected by the fast-event variable *aux_change*.

> **NOTE:** The CIO chip automatically generates a true *aux_change* condition when the *set_aux_ctl_leads* routine has been executed. The *aux_change* condition, therefore, should be placed in a separate programming state from the *set_aux_ctl_leads* routine.

The transmitter's program might look like this:

```
LAYER: 1
 {
  extern fast_event aux_change;
  extern volatile const unsigned short curr_aux_value;
  volatile unsigned short curr;
  unsigned short mask;
  unsigned char data;
 }
    STATE: configure_leads
       CONDITIONS: ENTER_STATE
       ACTIONS:
       {
        set_aux_direction(0x7f00);
        set_aux_ctl_leads(0x0100);
        curr = curr_aux_value;
        data = 0x01;
        mask = curr ^ 0x8000;
        display_prompt("Connect cable.  Press spacebar to transmit.                    ");
        pos_cursor(1,0);
       }
       NEXT_STATE: send_data
    STATE: send_data
       CONDITIONS: KEYBOARD " "
       ACTIONS:
       {
        if(data <= 10)
          {
           write_aux(mask | data);
           displayf("Transmission %d waiting for ACK.                    \n", data);
          }
       }
       NEXT_STATE: waiting
    STATE: waiting
       CONDITIONS: {aux_change}
       ACTIONS:
       {
        data++;
        mask = (mask ^ 0x8000);
        displayf("ACK received:  %04x  Press spacebar to transmit.          \n", curr);
       }
       NEXT_STATE: send_data
       CONDITIONS: {data > 10}
       ACTIONS:
       {
        display_prompt("End of test.                                              ");
       }
```

The receiver's program would look like this:

```
LAYER: 1
 {
  extern fast_event aux_change;
  extern volatile const unsigned short curr_aux_value;
  volatile unsigned short curr;
  unsigned short mask;
  int count;
 }
     STATE: configure_leads
        CONDITIONS: ENTER_STATE
        ACTIONS:
        {
         set_aux_direction(0xfeff);
         set_aux_ctl_leads(0x8000);
        }
        CONDITIONS: {aux_change}
        ACTIONS:
        {
         curr = curr_aux_value;
         count = 1;
         mask = curr ^ 0x0100;
         display_prompt("Connect cable.  Ready to receive.                              ");
         pos_cursor(1,0);
        }
        NEXT_STATE: receive_data
     STATE: receive_data
        CONDITIONS: {aux_change}
        ACTIONS:
        {
          displayf("Transmission %d received:  %04x    Press spacebar to send ACK.              \n",
                count, curr);
        }
        NEXT_STATE: send_ack
        CONDITIONS: {count > 10}
        ACTIONS:
        {
         display_prompt("End of test.                                                   ");
        }
     STATE: send_ack
        CONDITIONS: KEYBOARD " "
        ACTIONS:
        {
         if(count <= 10)
            {
             write_aux(mask);
             count++;
             mask = (mask ^ 0x0100);
            }
        }
        NEXT_STATE: receive_data
```

> **NOTE:** If you designate more than one lead as control, you might need to compare *prev_aux_value* with *curr_aux_value* to determine if the lead you are interested in is the one that changed. Here, since there is only one input-control lead on each side, the event *aux_change* is sufficient to signal and to acknowledge transmission. The value of *prev_aux_value* does not have to be checked.

set_aux_reg

Synopsis

```
extern void set_aux_reg(reg_value_word);
unsigned short reg_value_word;
```

Description

The CIO chip may be reconfigured by the user via the *set_aux_reg* routine.

> **NOTE:** At present, the initial configuration of the Master Interrupt Control Register is (0x0082). The initial configuration of the Master Configuration Control Register is (0x0194).

Inputs

The only input is a sixteen-bit variable. The high byte is the CIO register number; the low byte is the value to store in the register number. For register numbers and their values, consult Appendix B in Zilog's *Z8036 Z-CIO/Z8536 CIO Counter/Timer and Parallel I/O Unit Technical Manual*, March 1982.

Example

The Master Configuration Control Register allows for selective enabling/disabling of the CIO ports. Port A's input/output is reflected in the least-significant byte of *reg_value_word*. Port B's input/output is reflected in the most-significant byte of *reg_value_word*.

> **NOTE:** Port C of the CIO chip is used internally and is not available to the user of the INTERVIEW.

Suppose you want to disable port B input, output, and interrupts (ports A and C enabled) in one state, and in another state restore the original configuration (ports A, B, and C enabled):

```
LAYER: 1
    STATE: reconfigure_chip
      CONDITIONS: ENTER_STATE
      ACTIONS:
      {
       set_aux_reg(0x0114);
      }
    STATE: restore_original_config
      CONDITIONS: ENTER_STATE
      ACTIONS:
      {
       set_aux_reg(0x0194);
      }
```

69 Other Library Tools

The C structures, variables, and routines in this section provide additional programming tools not specific to any particular protocol. Most of these tools approximate layer-independent conditions or actions. Refer to Section 27 for more detailed explanations of the purposes of specific conditions and actions. Sometimes the name of the variable or routine is sufficient for identifying its related spreadsheet token. When this is not the case, the information is provided below.

69.1 Structures

Use the structures *tm*, *crnt_tm*, and *prev_tm* listed in Table 69-1 to monitor the current and previous date and time. Each minute the values in *crnt_tm* are copied to *prev_tm*. Then *crnt_tm* is updated. These structures are used to produce the date/time displays at the top of Run-mode screens and the Date/Time Setup screen.

The variables *flag_struct.prev*, *flag_struct.current*, and *flag_struct.old* (in the *flag_struct* structure) are used each time a flag is incremented, decremented, or set to a particular value. The current, previous, and old values these variables represent work the same way as their counterparts in the counter structure, discussed fully in Section 62.1(A).

> **NOTE:** The purpose of flags is to make it easy for the user to isolate selected bits in a variable. The translator does most of the work of flags by taking the user's flag masks and coding them in C. Flags constructed entirely in C bypass the translator and require the programmer to create the flag-mask code normally generated by the translator.

Before using the timeout routines included in this section, declare an instance of the *timeout* structure shown in Table 69-1. Refer to the *timeout_restart_action* and *timeout_stop_action* routines for examples of how to use this structure.

The *keyboard* structure stores the value of the most recent ASCII key used. The structure variable *keyboard.value* is updated only by the fast-event variable *keyboard_new_key*.

Table 69-1
Structure Fields—Other Library Tools

Type	Variable	Value (hex/decimal)	Meaning
Structure Name: keyboard			Declared as type *extern struct*. Declared automatically if program KEYBOARD condition is used. Updated by *keyboard_new_key* event variable. Reference the structure variable as follows: *keyboard.value*.
char	value		ASCII value of key just executed.
Structure Name: tm			Structure of time of day. Declared as type *extern struct*. Reference a structure variable as follows: *tm.tm_sec*.
int	tm_sec	*0-3b/0-59*	Seconds after the minute. Not currently updated; always set to -1.
int	tm_min	*0-3b/0-59*	Minutes after the hour.
int	tm_hour	*0-17/0-23*	Hours since midnight.
int	tm_mday	*1-1f/1-31*	Day of month.
int	tm_mon	*0-b/0-11*	Months since January.
int	tm_year		Years since 1900.
int	tm_wday	*0-6*	Days since Sunday. Not currently updated; always set to -1.
int	tm_yday	*0-16d/0-365*	Days since January 1. Not currently updated; always set to -1.
int	tm_isdst		Daylight Savings Time flag. Not currently updated; always set to -1.
Structure Name: crnt_tm			Structure of current time of day. Updated every minute. Declared as type *extern struct tm*.
Structure Name: prev_tm			Structure of previous time of day, one minute ago. Declared as type *extern struct tm*.
Structure Name: flag_struct			Structure of a flag. Declared as type *struct*. Declared automatically if a program flag is used. Program flags assigned to structure as follows: *struct flag_struct flag_*name. Reference a structure variable as follows: *flag_*name*.current*.
unsigned short	prev		When converting a flag action to C, the translator compares *prev* with *current* to determine whether flag has changed. Then *prev* is updated to *current* and *flag_*name*_change* is signaled.
unsigned short	current		This value of flag is acted on directly by program actions.
unsigned short	old		When converting a flag condition to C, the translator compares *old* with *current* to determine whether true condition is new (transitional). After program logic has examined flag, *old* is updated to *prev*.

Table 69-1 (continued)

Type	Variable	Value (hex/decimal)	Meaning
Structure Name:	**timeout**		Structure of a timeout. Declared as type *struct*. Declared automatically if a program timeout is used. Program timeouts assigned to structure as follows: *struct timeout* name. Reference a structure variable as follows: *timeout*_name.*event_id*.
unsigned long	event_id		Four bytes of a 6-byte timeout, containing the segment number and offset. *Timeout*_name_*stop* routines set this event id to zero.
unsigned short	event_id_uid		Two bytes of a 6-byte timeout which uniquely identify (uid) the timeout. Do not try to assign a value to this variable.

69.2 Variables

All of the variables in Table 69-2 are valid in either emulate or monitor mode.

(A) Monitoring Events

The event variables in Table 69-2 are *fevar_time_of_day*, *flag_name_change*, *timeout_name_expired*, *signal_name*, *keyboard_new_key*, and *keyboard_new_any_key*.

Event variable *fevar_time_of_day* comes true once a minute. An example of how to use this variable is provided in Section 54.1. This event variable is part of the spreadsheet TIME condition.

The event variable *keyboard_new_key* is used by the translator in a spreadsheet KEYBOARD condition. It comes true when any ASCII key is pressed. The event *keyboard_new_any_key*, on the other hand, comes true when an ASCII or other keyboard key is pressed. The only keys which will not trigger this event are [EDIT], [CAPS LOCK], [F1]-[F8] softkeys, and [DONE].

(B) Status Variables

Status variables are those in Table 69-2 that do not include *event* in the Type column. Their associated event variables guarantee that they are updated and tested.

Time and date variables are updated by *fevar_time_of_day*. Variables *crnt_time_of_day*, *prev_time_of_day*, *crnt_date_of_day*, and *prev_date_of_day* are older versions of variables that belong to the *crnt_tm* and *prev_tm* structures. The C translator uses these older versions when it construct time-of-day conditions (e.g., CONDITIONS: TIME 1614).

The status variable *keyboard_any_key* is updated by the fast-event variable *keyboard_new_any_key*.

Table 69-2
Other Library Variables

Type	Variable	Value (hex/decimal)	Meaning
extern fast_event	fevar_time_of_day		True once per minute. Line Setup configured for emulate or monitor mode.
extern event	flag_*name*_change		This event must be signaled by the program itself; it is not "external" to the program. The translator signals this event as part of the FLAG increment, decrement, or set action. Line Setup configured for emulate or monitor mode.
extern event	timeout_*name*_expired		This event must be signaled by the program itself. It is not "external" to the C program. The translator signals this event as part of the *timeout_restart_action* routine. Line Setup configured for emulate or monitor mode.
extern event	signal_*name*		True when the named signal is the argument in a *signal* routine. Spreadsheet-token equivalent is ON_SIGNAL name. Line Setup configured for emulate or monitor mode.
extern volatile unsigned short	crnt_time_of_day	*0-937/0-2359*	Current time is stored in this variable. Updated as soon as time changes. Line Setup configured for emulate or monitor mode.
extern volatile unsigned short	prev_time_of_day	*0-937/0-2359*	Current time is stored in this variable. Updated when time changes, but only after logic has had a chance to compare current and previous time. Line Setup configured for emulate or monitor mode.
extern volatile const unsigned char	crnt_date_of_day	*1-1f/1-31*	Current date is stored in this variable. Updated as soon as date changes. Line Setup configured for emulate or monitor mode.
extern volatile const unsigned char	prev_date_of_day	*1-1f/1-31*	Current date is stored in this variable. Updated when date changes, but only after logic has had a chance to compare current and previous date. Line Setup configured for emulate or monitor mode.
extern fast_event	keyboard_new_key		True when any ASCII key is pressed. Line Setup configured for emulate or monitor mode.

Table 69-2 (continued)

Type	Variable	Value (hex/decimal)	Meaning
extern fast_event	keyboard_new_any_key		True when any key is pressed. The only exceptions are [EDIT], [CAPS LOCK], [F1]-[F8] softkeys, and [DONE]. Line Setup configured for emulate or monitor mode.
extern volatile unsigned short	keyboard_any_key		Identifies last key or key-combination executed. Line Setup configured for emulate or monitor mode.
		0-7f/0-127	ASCII keys
		80-17f/ 128-383	not used
			Field entry keys:
		180/384	[HEX]
		181/385	[NOT EQUAL]
		182/386	[BIT MASK]
		183/387	[FLAG]
		184/388	[DON'T CARE]
		185/389	[CLEAR]
		186/390	[SHIFT]-[CLEAR]
		187/391	[CTRL]-[CLEAR]
		18a/394	[TAB]
		18b/395	[SHIFT]-[TAB]
		18c/396	[SHIFT]-[FLAG]
		18d/397	[CTRL]-[FLAG]
		18e/398	[SHIFT]-[BIT MASK]
		18f/399	[CTRL]-[BIT MASK]
		190/400	[SHIFT]-[DON'T CARE]
		191/401	[CTRL]-[DON'T CARE]
		192/402	[CTRL]-[TAB]
		193/403	[SHIFT]-[NOT EQUAL]
		194/404	[CTRL]-[NOT EQUAL]
		195/405	[SHIFT]-[HEX]
		196/406	[CTRL]-[HEX]
		10d/269	[RETURN]
		1a0/416	[CTRL]-[RETURN]
		1a1/417	[SHIFT]-[RETURN]
			Editing Keypad Keys:
		1a2/418	[INSERT LINE]
		1a3/419	[INSERT CHAR]
		1a4/420	[DELETE LINE]
		1a5/421	[DELETE CHAR]
		1a6/422	[SHIFT]-[INSERT LINE]
		1a7/423	[SHIFT]-[INSERT CHAR]
		1a8/424	[SHIFT]-[DELETE LINE]

(keyboard_any_key variable continued on next page)

Table 69-2 (continued)

Type	Variable	Value (hex/decimal)	Meaning
	(keyboard_any_key continued)		*Editing Keypad Keys (cont):*
		1a9/425	[SHIFT]-[DELETE CHAR]
		1aa/426	[CTRL]-[INSERT LINE]
		1ab/427	[CTRL]-[INSERT CHAR]
		1ac/428	[CTRL]-[DELETE LINE]
		1ad/429	[CTRL]-[DELETE CHAR]
			Utility Keys:
		1b0/432	[PROGRAM]
		1b1/433	[RUN]
		1b2/434	[LOAD]
		1b3/435	[SAVE]
		1b4/436	[FREEZE]
		1b5/437	[RECORD]
		1b7/439	[PRINT]
		1b8/440	[XEQ]
		1ba/442	[ ]
		1bb/443	[SHIFT]-[ ]
		1bc/444	[CTRL]-[ ]
		1bd/445	[SHIFT]-[PROGRAM]
		1be/446	[CTRL]-[PROGRAM]
		1bf/447	[SHIFT]-[RUN]
		1c0/448	[CTRL]-[RUN]
		1c1/449	[SHIFT]-[LOAD]
		1c2/450	[CTRL]-[LOAD]
		1c3/451	[SHIFT]-[SAVE]
		1c4/452	[CTRL]-[SAVE]
		1c5/453	[SHIFT]-[FREEZE]
		1c6/454	[CTRL]-[FREEZE]
		1c7/455	[SHIFT]-[RECORD]
		1c8/456	[CTRL]-[RECORD]
		1c9/457	[SHIFT]-[EDIT]
		1ca/458	[CTRL]-[EDIT]
		1cb/459	[SHIFT]-[PRINT]
		1cc/460	[CTRL]-[PRINT]
		1cd/461	[SHIFT]-[XEQ]
		1ce/462	[CTRL]-[XEQ]
			Pure Cursor Keys:
		1d0/464	[↑]
		1d1/465	[SHIFT]-[↑]
		1d2/466	[CTRL]-[↑]
		1d3/467	[←]
		1d4/468	[SHIFT]-[←]
		1d5/469	[CTRL]-[←]
		1d6/470	[HOME]

(keyboard_any_key variable continued on next page)

Table 69-2 (continued)

Type	Variable	Value (hex/decimal)	Meaning
	(keyboard_any_key continued)		*Pure Cursor Keys (cont):*
		1d7/471	[SHIFT]-[HOME]
		1d8/472	[CTRL]-[HOME]
		1d9/473	[→]
		1da/474	[SHIFT]-[→]
		1db/475	[CTRL]-[→]
		1dc/476	[↓]
		1dd/477	[SHIFT]-[↓]
		1de/478	[CTRL]-[↓]
			Cursor Keypad Keys:
		1e0/480	[RUB OUT]
		1e1/481	[MARK]
		1e2/482	[SHIFT]-[MARK]
		1e3/483	[BRK]
		1e4/484	[SHIFT]-[BRK]
		1e5/485	[ROLL BACK]
		1e6/486	[PREV PAGE]
		1e7/487	[ROLL FWD]
		1e8/488	[NEXT PAGE]
		1e9/489	[CTRL]-[MARK]
		1ea/490	[CTRL]-[BRK]
		1eb/491	[SHIFT]-[ROLL BACK]
		1ec/492	[CTRL]-[ROLL BACK]
		1ed/493	[SHIFT]-[ROLL FWD]
		1ee/494	[CTRL]-[ROLL FWD]
		1ef/495	[SHIFT]-[PREV PAGE]
		1f0/496	[CTRL]-[PREV PAGE]
		1f1/497	[SHIFT]-[NEXT PAGE]
		1f2/498	[CTRL]-[NEXT PAGE]
		1f3/499	[SHIFT]-[RUB OUT]
		1f4/500	[CTRL]-[RUB OUT]
			Other Keys:
		1f5/501	[CTRL]-[1]
		1f6/502	[CTRL]-[2]
		1f7/503	[CTRL]-[3]
		1f8/504	[CTRL]-[4]
		1f9/505	[CTRL]-[5]
		1fa/506	[CTRL]-[6]
		1fb/507	[CTRL]-[7]
		1fc/508	[CTRL]-[8]
		188/392	[CTRL]-[9]
		189/393	[CTRL]-[0]
		1fd/509	[CTRL]-[-]

69.3 Routines

timeout_restart_action

Synopsis

```
extern void timeout_restart_action(timeout_name_ptr, value, function);
struct * timeout_name_ptr
  {
   unsigned long event_id;
   unsigned short event_id_uid;
  };
unsigned short value;
void function ();
```

Description

This routine starts a named timeout timer running down, starting at a specified value. When the timer reaches zero, a named function is called. The *timeout_restart_action* routine, preceded by a call to the *timeout_stop_action* routine, is the equivalent of the softkey TIMEOUT name RESTART action on the Protocol Spreadsheet.

Inputs

The first parameter is a pointer to the timeout structure. See Table 69-1 for further explanation of the *timeout* structure.

The second parameter is the starting value of the timeout timer in milliseconds.

The third parameter is the name of a routine to be called when the timeout expires. The routine may include the following statement: *timeout_name.event_id = 0;*. Timeout-stop actions set this event ID to zero. This action is not strictly necessary here, since the timeout has already expired; but the action may make the processing of subsequent stop actions slightly more efficient.

The body of the routine to be called may also include this statement: *signal(timeout_*name*_expired);*. In a softkey-entered TIMEOUT RESTART action, both statements are included in a routine called *timeout_*name*_isp*.

> **NOTE:** The routine named in the third parameter is an interrupt service process (isp). A long definition for this routine makes the processing of *timeout_restart_action* unpredictable.

Example

When a frame is sent, start a timeout timer at 2 seconds. When it expires, sound the alarm. If another frame is sent before the 2 seconds expires, stop the current timer and restart the timeout.

```
{
 struct timeout
  {
   unsigned long event_id;
   unsigned short event_id_uid;
  };
 struct timeout timeout_example;
 extern event timeout_example_expired;
 void timeout_example_isp ()
  {
   timeout_example.event_id = 0;
   signal(timeout_example_expired);
  }
}
LAYER: 2
    STATE: example_of_timeout
        CONDITIONS: FRAME_SENT
        ACTIONS:
        {
         timeout_stop_action(&timeout_example);
         timeout_restart_action(&timeout_example, 2000, timeout_example_isp);
        }
        CONDITIONS:
        {
         timeout_example_expired
        }
        ACTIONS: ALARM
```

Here is a version of the program that accomplishes the same result without an action to signal the timeout event:

```
{
 struct timeout
  {
   unsigned long event_id;
   unsigned short event_id_uid;
  };
 struct timeout timeout_example;
 extern void sound_alarm();
}
LAYER: 2
    STATE: example_of_timeout
        CONDITIONS: FRAME_SENT
        ACTIONS:
        {
         timeout_stop_action(&timeout_example);
         timeout_restart_action(&timeout_example, 2000, sound_alarm);
        }
```

timeout_stop_action

Synopsis

```
extern void timeout_stop_action(timeout_name_ptr);
struct * timeout_name_ptr
  {
   unsigned long event_id;
   unsigned short event_id_uid;
  };
```

Description

This routine stops a named timeout timer, preventing it from expiring. The softkey equivalent of this routine is the **TIMEOUT name STOP** action on the Protocol Spreadsheet. *timeout_stop_action* also precedes the call to the *timeout_restart_action* in the spreadsheet **TIMEOUT name RESTART** action.

Inputs

The only parameter is a pointer to the *timeout* structure. See Table 69-1 for further explanation of the timeout structure.

Example

In this example, if the user presses the [S] key, the timeout timer will not expire and the alarm will not sound (until another frame is sent and the timeout is restarted).

```
{
 struct timeout
  {
   unsigned long event_id;
   unsigned short event_id_uid;
  };
 struct timeout timeout_example;
 extern void sound_alarm();
}
LAYER: 2
     STATE: stop_a_timeout
        CONDITIONS: FRAME_SENT
        ACTIONS:
        {
         timeout_stop_action(&timeout_example);
         timeout_restart_action(&timeout_example, 2000, sound_alarm);
        }
        CONDITIONS:  KEYBOARD "Ss"
        ACTIONS:
        {
          timeout_stop_action(&timeout_example);
        }
```

index

Synopsis

```
extern char * index(string, character);
char * string;
char character;
```

Description

This routine searches for an instance of a character starting at the beginning of a specified list. The routine is used by the C translator to convert CONDITIONS: KEYBOARD softkey entries into C. This routine must be declared.

Inputs

The first parameter is a list of characters to be searched.

The second parameter is the character to be searched for in the list.

Returns

This routine returns a pointer to the first instance of the specified character, or zero if it does not occur.

Example

In the example below, the following test is established: when a key is pressed on the keyboard, search for a match to the keyboard character in the string " abc ". If it is found, sound the alarm.

```
{
 extern char * index();
 extern fast_event keyboard_new_key;
 extern struct keyboard
  {
   char value;
  };
 extern struct keyboard keyboard;
}
LAYER: 1
     STATE: index_example
        CONDITIONS:
        {
         (keyboard_new_key && index(" abc ", keyboard.value))
        }
        ACTIONS: ALARM
```

Let's suppose that the user presses the space bar. In this case, the returned pointer will be pointing to the blank preceding the "a." If *rindex* had been used, the returned pointer would be pointing to the blank following the "c." As long as any non-null character is returned, the condition is *true*.

rindex

Synopsis

```
extern char * rindex(string, character);
char * string;
char character;
```

Description

This routine searches for an instance of a character starting at the end of a specified list. This routine must be declared.

Inputs

See *index*.

Returns

See *index*.

Example

See *index*.

load_program

Synopsis

```
extern void load_program(filename_ptr)
const char * filename_ptr;
```

Description

The *load_program* routines allows you to link programs together while the unit is in Run mode. When a call to *load_program* is encountered in a spreadsheet program, the current program is exited. The program named as the argument in the routine is loaded and run. When you return to Program mode, the program displayed on the Protocol Spreadsheet will be the one just loaded. If *load_program* fails, you are returned to the main menu screen in Program mode.

Inputs

The only input is the absolute pathname, prefixed by the device name, of the file to be loaded. Valid device names are "HRD," "FD1," and "FD2."

Example

In the example below, at the successful conclusion of the last of a series of tests in module 18, a program for module 19 will be loaded and run.

```
LAYER: 3
    STATE: test_26
        CONDITIONS: ENTER_STATE
        ACTIONS: SEND DIAG
        CONDITIONS: RCV CLEAR_CONF
        ACTIONS: TRACE "Test_26 passed"
        {
         load_program("FD1/usr/module_19");
        }
```

lock

Synopsis

#include <stdio.h>
extern void lock(lock_variable_ptr);
*int * lock_variable_ptr;*

Description

The *lock* routine implements a lock using the integer variable pointed to by the routine parameter. If the lock variable is currently locked, the task goes to sleep. When an unlock on the same variable occurs (within an independent task), the task invoking the lock function will attempt to claim the lock. If successful, the task is executed; otherwise, it goes back to sleep until the next unlock.

> **NOTE:** If locking is used at any place in the program, all related or possibly concurrent routines must also use the locking functions.

> **NOTE:** The lock variable should always be defined as a global integer, never as local to a function. The lock variable should never be altered by the user program or deadlock can occur. Deadlock also results if the lock is invoked twice within the same task without an intervening unlock.

Inputs

The only parameter is a pointer to the lock variable.

Example

Two tasks concurrently write to their own file streams. (The file streams are local to the routine *write_fox*, making them independent of each other even though they have the same name.) However, during the *fclose* operation (which automatically calls *fflush*), both tasks need to write to the same file. The locking routines ensure that the writes to the file occur sequentially, not concurrently.

```
{
#include <stdio.h>
const char data [] = "((FOX))\n";
int key;
void write_fox()
 {
  FILE * stream_ptr;
  size_t n;
  lock(&key);
  if((stream_ptr = fopen("FD2/usr/buff01", "a")) == 0)
     display_prompt("Cannot open file.                                             ");
  else
     display_prompt("File opened.                                                  ");
  n = fwrite(data, 1, sizeof(data)-1, stream_ptr);
  pos_cursor(1,0);
  if(n != (sizeof(data)-1))
     displayf("Write error.                                                  \n");
  else
     displayf("Write completed.                                              \n");
  if(fclose(stream_ptr) != 0)
     displayf("Either file is already closed, or close cannot be executed.      ");
  else
     displayf("File closed.                                                    ");
  unlock(&key);
 }
}
LAYER: 1
         TEST: a
      STATE: write_and_signal
          CONDITIONS: RECEIVE STRING "THE QUICK BROWN FOX"
          ACTIONS: SIGNAL xyz
          {
           write_fox();
          }
         TEST: b
      STATE: write_only
          CONDITIONS: ON_SIGNAL xyz
          ACTIONS:
          {
           write_fox();
          }
```

unlock

Synopsis

```
#include <stdio.h>
extern void unlock(lock_variable_ptr);
int * lock_variable_ptr;
```

Description

The *unlock* routine implements the inverse of the *lock* routine using the same integer variable. Sleeping tasks will be woken up to retry their attempt to claim the lock. One will succeed, and the rest will go back to sleep. See also *lock* routine.

Inputs

The only parameter is a pointer to the lock variable.

Example

See *lock* routine.

signal

Synopsis

extern void signal(signal_name);

Description

This routine conveys instructions to other tests and layers where conditions are monitoring the signal by name. The softkey equivalent of this routine is the SIGNAL action on the Protocol Spreadsheet.

Inputs

The only parameter is a name descriptive of the event being signaled.

Example

```
LAYER: 2
     STATE: signal_routine
        CONDITIONS: RCV FRMR
        ACTIONS:
        {
         signal(signal_link_down);
        }
        CONDITIONS: ON_SIGNAL link_down
        ACTIONS: ALARM
```

Here is a related example, this time with the signal detection also given in C. Note that a signal automatically generates an "event" that can be detected alone in a *waitfor* clause.

```
{
 extern event link_down;
}
LAYER: 2
     STATE: signal_event
        CONDITIONS: RCV FRMR
        ACTIONS:
        {
         signal(link_down);
        }
        CONDITIONS:
        {
         link_down
        }
        ACTIONS: ALARM
```

sound_alarm

Synopsis

extern void sound_alarm();

Description

This routine will sound the alarm. The softkey equivalent of this routine is the ALARM action on the Protocol Spreadsheet.

Example

When a bad BCC is detected on the DTE side of the link, sound the alarm.

```
LAYER: 1
    STATE: example
        CONDITIONS: DTE BAD_BCC
        ACTIONS:
        {
         sound_alarm();
        }
```

start_rcrd_play

Synopsis

extern void start_rcrd_play();

Description

Depending on the Line Setup configuration, this routine activates data recording or playback. If the Line Setup menu shows **Mode:** MONITOR, **Source:** DISK, the routine controls playback. In all other cases, it initiates recording.

Unless your recording source is RAM, make a call to *fclose* in programs containing disk I/O routines (Section 65) before you start to record (or resume playback). If you don't, the file will be closed automatically as soon as recording (or playback) begins, even if processes on the file have not been completed. (Using the RECORD key to activate recording or resume playback will have the same effect.)

Example

```
LAYER: 1
    STATE: example
        CONDITIONS: KEYBOARD " "
        ACTIONS:
        {
         start_rcrd_play();
        }
```

suspend_rcrd_play

Synopsis

extern void suspend_rcrd_play();

Description

Depending on the Line Setup configuration, this routine suspends data recording or playback. If the Line Setup menu shows **Mode:** MONITOR, **Source:** DISK, the routine controls playback. In all other cases, it suspends recording. Once recording or playback is suspended, resume it with a call to *start_rcrd_play*.

Unless your recording source is RAM, do not call disk I/O routines (Section 65) until you suspend recording (or playback). If you do, the disk I/O operation will fail.

> NOTE: Although playback is immediately suspended when *suspend_rcrd_play* is executed, the screen display continues until the character buffer's contents are fully displayed. (For bit-image data, the FIFO must empty.) At slower playback speeds, you may notice a slight delay before the display actually freezes.

Example

```
LAYER: 2
    STATE: example
        CONDITIONS: KEYBOARD " "
        ACTIONS:
        {
         suspend_rcrd_play();
        }
```

send_key

Synopsis

extern void send_key(number_of_keys, keys_ptr);
unsigned char number_of_keys;
*unsigned short * keys_ptr;*

Description

This routine sends a specific keystroke (or sequence of keys) during Run mode, as though the operator pressed the key. It also may be used to change the Run-mode display.

Inputs

The first parameter specifies the number of keys to be sent.

The second parameter is a pointer to an array of *shorts*. This array lists the keys to be sent. To send keyboard keys, use the values listed in Table 69-2 for the *keyboard_any_key* variable. To change the Run-mode display, send two keys. The first "key" always has a value of 0xff75. The second "key" identifies the desired display-screen. Use the values listed in Table 61-1 for the *crnt_display_screen* variable.

Example

For this example, assume you are playing back data from a disk and that the initial Run-mode screen is the dual-line data display. After a five-second pause, playback is slowed as though you pressed ⇩. As soon as a bad BCC is detected on the DTE side, the data display will change to the Layer 2 Protocol Trace screen.

```
{
 unsigned short keys [] = {0xff75, 0x42};
 unsigned short slow_down [] = {0x1dc};
}
LAYER: 2
    STATE: change_displays
       CONDITIONS: ENTER_STATE
       ACTIONS: TIMEOUT pause RESTART 5
       CONDITIONS: TIMEOUT pause
       {
        send_key(1, slow_down);
       }
       CONDITIONS: DTE BDBCC
       ACTIONS:
       {
        send_key(2, keys);
       }
```

70 X.21 Library

The Test Interface Module (TIM) located in the rear of the INTERVIEW determines the leads available for monitoring and control (Section 10). The variables and routines in this section apply to the X.21 interface module. RS-232, V.35, and RS-449 modules are treated in Section 60.

To use the C variables and routines explained in this section, you must select **Buffer Control Leads: YES** on the FEB Setup menu. See Section 7.1(B). If no other source for clock is provided, use internal clock (Line Setup menu). Finally, load in the X.21 package via the Layer Setup screen.

The variables and routines approximate X.21 Layer 1 spreadsheet-generated conditions and actions. Their use on the Protocol Spreadsheet is not limited to any particular layer, though normally they belong at Layer 1. Refer to Section 32 for more detailed explanations of the purposes of specific conditions and actions. Sometimes the name of the variable or routine is sufficient for identifying its related spreadsheet token. When this is not the case, the information is provided below.

70.1 Structures

Use the structure *xmit_list*, shown in Table 70-1, when transmitting line data via the *x21_transmit_call* routine. Refer to *x21_transmit_call* in Section 70.3(A) for an example of how to use this structure.

Table 70-1
X.21 Structures

Type	Variable	Value (hex/decimal)	Meaning
Structure Name:	xmit_list		Structure of a transmit list for *x21_transmit_call* routine. Declared as type *struct*. Declared automatically if a softkey-entered CALL_SETUP_SEND action is taken. Reference member variables of the structure as follows: *xmit_list.string_length*
unsigned char *	string		pointer to the location of the transmit string—the transmit string is declared separately
unsigned short	string_length	*0-ffff/0-65535*	length of the transmit string

70.2 Variables

With an X.21 TIM installed, you may monitor the T and R data leads, the C and I control leads, and UA. See Table 70-2.

The fast-event variable *fevar_eia_changed* detects a change in leads. It does not establish which lead(s) has changed, nor the validity of the lead's status. Two associated status variables, *current_eia_leads* and *previous_eia_leads*, indicate the condition of the leads. These are two-byte (*short*) variables. Each lead is represented by a different bit in the *short*. Table 70-2 provides the mask that can be used to isolate each lead.

Other bits in these variables monitor the validity of lead status. For the status of a lead to be considered valid in X.21, the lead must be stable for a minimum of 16 bit-times. Each lead's valid status is indicated by a separate bit in *current_eia_leads* and *previous_eia_leads*. Again, refer to Table 70-2.

Whenever a lead changes, the value in *current_eia_leads* is written to *previous_eia_leads*. Then *current_eia_leads* is updated.

(A) Masking To Detect a Change in a Given Lead

To test whether or not a given lead changed, I for example, while disregarding its status, enter the following condition on the Protocol Spreadsheet:

```
CONDITIONS:
{
fevar_eia_changed && (((current_eia_leads ^ previous_eia_leads) & 0x40) == 0x40)
}
```

Select a mask value from the list in Table 70-2 to indicate which lead you care about. Specify multiple leads with a mask derived via hexadecimal addition.

The mask for I is 0x40. In the example, the event *fevar_eia_changed* updated *current_eia_leads*. The new *current_eia_leads* was *bitwise-exclusive-OR*ed with *previous_eia_leads* to identify all the leads that changed. Then the result was *bitwise AND*ed with the I mask to determine if I was among the leads that changed. If this result was equal to the mask, the lead changed.

Following the evaluation of the condition, *previous_eia_leads* was updated to match *current_eia_leads*.

Table 70-2
X.21 Variables

Type	Variable	Value (hex/decimal)	Meaning
extern fast_event	fevar_eia_changed		True when the status changes for an EIA lead. Line Setup configured for emulate or monitor mode.
extern const volatile unsigned short	current_eia_leads	1 2 4 8 10/16 40/64 80/128 100/256 200/512 400/1024	C-valid B (RS-232 equivalent is SQ) I-valid (RI) R-valid (DSR) T-valid (DTR) I (CTS) C (RTS) R (RD) UA T (TD) A value in this list indicates which lead(s) you care about. When *anded* (&) with *current_eia_leads*, the result equals zero if the lead is *on* (or the mask if the lead is *off*). For validity checks, the result of *anding* with *current_eia_leads* equals the mask for *valid* (or zero for *invalid*). <u>Examples</u>: STATE: c_on_and_valid { *if ((current_eia_leads & 0x81) == 1) sound_alarm();* } STATE: c_off_and_valid { *if ((current_eia_leads & 0x81) == 0x81) sound_alarm();* } *Note:* This variable will store EIA status if (1) internal or external clock is supplied, (2) EIA leads are enabled on FEB Setup, and (3) *fevar_eia_changed* has updated the leads. Line Setup configured for emulate or monitor mode.
extern const volatile unsigned short	previous_eia_leads		Same values as *current_eia_leads*. Updated when leads change, but only after logic has had a chance to compare current and previous leads. Line Setup configured for emulate or monitor mode.

(B) Masking For the Status or Validity of a Lead

You may also test the current status or validity of a lead, independent of any change. If a mask testing for status is *and*ed with *current_eia_leads*, zero will mean that the lead in on. If the result equals the mask, the lead is off. If a mask testing for validity is *and*ed with *current_eia_leads*, the lead status is valid when the result equals the mask. If the result is zero, the status is invalid.

To test for both status and validity, derive a mask via hexadecimal addition. *And* the mask with *current_eia_leads*, as in this *if* statement testing for I "on" and valid:

```
STATE: test_for_i_on_and_valid
  {
  if((current_eia_leads & 0x44) == 4)  sound_alarm();
  }
```

(C) Detect Change and Current Status

The two examples shown above could be combined to test for I changing from off to valid on:

```
CONDITIONS:
{
  (fevar_eia_changed && (((current_eia_leads ^ previous_eia_leads) & 0x40) == 0x40) &&
       ((current_eia_leads & 0x44) == 4))
}
```

This example approximates the translator's version of the spreadsheet-token condition LEADS I V-ON when it appears alone in a conditions block. When a LEADS condition is combined with another condition, in most cases the other condition will supply the event variable and only the lead status test will be used.

70.3 Routines

(A) Control and Transmit

Use the following routines in emulate mode only. If you try to call one of these routines in monitor mode, you may be returned to the main program menu. When you go to the Protocol Spreadsheet and search for errors, a message like the following may be displayed: *"Error 140: Unresolved reference ctl_eia."*

ctl_eia

Synopsis

```
extern void ctl_eia(on_mask, off_mask);
unsigned short on_mask;
unsigned short off_mask;
```

Description

The *ctl_eia* routine allows you to control the status of the two X.21 control–leads. Which lead you control depends on your emulation mode. When the Line Setup menu shows **Mode:** EMULATE DCE, you control I. An EMULATE DTE selection gives you control over C. The softkey equivalent of this routine is the LEADS action on the Protocol Spreadsheet.

Inputs

The first parameter indicates which lead you want to turn on. One bit in the parameter controls a given lead: I (01) and C (04). Wherever there is a *zero* in the first parameter, the corresponding lead will be turned on. A one in this parameter will *not* cause any lead to be turned off. A value of 0xff will mean *don't care* (no action).

The second parameter indicates which lead you want in the "off" condition. One bit in the parameter controls a given lead: I (01) and C (04). Wherever there is a *one* in the second parameter, the corresponding lead will be turned off. Zeroes in this parameter do *not* turn leads on. A value of 0 will mean *don't care* (no action).

> **NOTE:** If both bytes are attempting to control the same lead, the off parameter will override the on parameter.

Example

Suppose your emulate mode is EMULATE DCE. As a DCE, you control the I lead. (An attempt to control the status of C will fail, since the DTE controls this lead.) When C is raised, you want to turn I on; when C drops, turn I off.

```
LAYER: 1
    STATE: control_I
       CONDITIONS: LEADS C ON
       ACTIONS:
       {
        ctl_eia(0xfe, 0x00);
       }
       CONDITIONS: LEADS C OFF
       ACTIONS:
       {
        ctl_eia(0xff, 0x01);
       }
```

x21_idle_action

Synopsis

extern void x21_idle_action(character);
unsigned char character;

Description

Only for format SYNC, the *x21_idle_action* routine allows you to change the idle-line condition applied by the INTERVIEW. A LEADS R BELLS action, for example, requires the *x21_transmit_call* routine in addition to *x21_idle_action*.

Inputs

The only parameter is a character or numeric value representing the idle character.

Example

To signal an incoming call, you would use the *x21_transmit_call* routine to send the sync pattern. Then you would use the *x21_idle_action* routine to send an idle string of *bells*:

```
LAYER: 1
{
 unsigned char syncs [] = {0x16,0x16};
 struct xmit_list
  {
   unsigned char * string;
   unsigned short string_length;
  };
 struct xmit_list send_string [] = {&syncs[0], 2};
}
    STATE: signal_incoming_call
       CONDITIONS: KEYBOARD " "
       ACTIONS:
       {
        x21_transmit_call(1, &send_string[0], 0);
        x21_idle_action('BL');
       }
```

x21_transmit_call

Synopsis

```
extern void x21_transmit_call(count, struct_send_string_ptr, xmit_tag);
unsigned short count;
struct xmit_list
  {
   char * string_ptr;
   unsigned short string_length;
  };
struct xmit_list * struct_send_string_ptr;
unsigned short xmit_tag;
```

Description

The *x21_transmit_call* routine sends a specified data string in call-setup mode. The softkey equivalent of this routine is the CALL_SETUP_SEND action.

Inputs

The first parameter is the number of strings to be sent.

The second parameter is a pointer to a structure which in turn identifies the location and length of each string.

The third parameter is a transmit tag. In other contexts it identifies the type of BCC to be sent. In X.21, however, no BCC is sent from Layer 1. The value of this parameter should be zero.

Example

Assume you are emulating a DTE. To send a call request in call-setup mode, enter the following spreadsheet program:

```
LAYER: 1
{
 unsigned char syncs [] = {0x16,0x16};
 unsigned char number [] = "1234567";
 unsigned char end [] = "+";
 struct xmit_list
  {
   unsigned char * string;
   unsigned short string_length;
  };
 struct xmit_list send_string [] = {&syncs[0], 2, &number[0], sizeof(number) - 1, &end[0], 1};
}
```

```
STATE: send
   CONDITIONS: RECEIVE STRING "[S]++"
   ACTIONS:
   {
    x21_transmit_call(3, &send_string[0], 0);
   }
```

Notice in the preceding example that sync characters were sent in the same call to *x21_transmit_call* that sent the called number. The equivalent softkey-generated action is LEADS T DATA CALL_SETUP_SEND "ⓢⓢ1234567+".

set_tcr_b

Synopsis

```
extern void set_tcr_b (tcr_register_mask, tcr_register_value);
unsigned char tcr_register_mask;
unsigned char tcr_register_value;
```

Description

This routine clamps the transmit line to 0 (space) or 1 (mark), or unclamps it so that transmit routines may be executed. In X.21, steady zero will signal a clear request/indication or a clear confirm, while steady 1 will indicate one of the call-ready or call-setup states.

The X.21 softkey actions that are built on this routine are LEADS R *(T)* ONE, LEADS R *(T)* ZERO, and LEADS R *(T)* DATA. In other contexts, the routine simply initiates and terminates a *break*.

Inputs

The first parameter is the mask that is *and*ed with the current TCR register to turn the current values of bits 3 and 4 (counting 1-8 from the right) to zero. This mask is always 0xf3.

The second parameter contains the new values of bits 3 and 4 that will be written to the register. The three available parameters are 0x08 to clamp the line to zero, 0x0c to clamp the line to 1, and 0x04 to unclamp the line and permit data transmissions.

Example

Assume you are emulating a DTE. To indicate a clear confirmation, enter the following spreadsheet program:

```
LAYER: 1
    STATE:
        CONDITIONS: KEYBOARD " "
        ACTIONS:
        {
         set_tcr_b (0xf3, 0x08);
         ctl_eia(0xff, 0x04);
        }
```

The equivalent softkey-generated action is LEADS T ZERO C OFF.

(B) Phase

The following routines are valid in either emulate or monitor mode.

enter_call_phase

Synopsis

extern void enter_call_phase();

Description

During the call-establishment phase, this routine overrides existing selections on the Line Setup menu with ASCII code, 7-bit odd parity, and SYNC format.

Example

When a lead changes, look for these conditions: T and R on (space), C and I off, and all leads valid. If conditions are true, enter call phase.

```
{
 extern fast_event fevar_eia_changed;
 extern const volatile unsigned short current_eia_leads;
}
LAYER: 1
    STATE: look_for_change_to_call_phase
        CONDITIONS:
        {
         fevar_eia_changed && ((current_eia_leads & 0x5dd) == 0xdd)
        }
        ACTIONS:
        {
         enter_call_phase();
        }
```

enter_data_phase

Synopsis

extern void enter_data_phase();

Description

During the data-transfer phase, this routine implements existing selections on the Line Setup menu.

Example

When a lead changes, look for these conditions: T and R off (mark), C and I on, and all leads valid. If conditions are true, enter data phase.

```
{
 extern fast_event fevar_eia_changed;
 extern const volatile unsigned short current_eia_leads;
}
LAYER: 1
     STATE: look_for_change_to_data_phase
        CONDITIONS:
        {
         fevar_eia_changed && ((current_eia_leads & 0x5dd) == 0x51d)
        }
        ACTIONS:
        {
         enter_data_phase();
        }
```

71 X.25 Layer 2 Library

When the X.25 Layer 2 package is loaded in via the Layer Setup screen, the following external routines and variables become available for use by the programmer. Their use on the Protocol Spreadsheet is not limited to any particular layer, though normally they belong at Layer 2.

The variables and routines approximate X.25 Layer 2 spreadsheet-generated conditions and actions. Refer to Section 33 for more detailed explanations of the purposes of specific conditions and actions. Sometimes the name of the variable or routine is sufficient for identifying its related spreadsheet token. When this is not the case, the information is provided below.

71.1 Structures

The structure *send_frame_structure* defines the format of transmitted X.25 frames. See Table 71-1. Use this structure to send frames via the *send_frame* routine in emulate mode. See Section 71.3(B). Each variable in the structure relates to some softkey selection or user entry in the SEND action.

71.2 Variables

(A) Monitoring Events

1. *Emulate or monitor mode.* X.25 Layer 2 events include frames detected, good or bad BCC's, and aborts. All event variables in Table 71-2 containing a *dte_* or *dce_* prefix are valid in either emulate or monitor mode. These event variables are *dte_frame, dce_frame, dte_good_bcc, dce_good_bcc, dte_bad_bcc, dce_bad_bcc, dte_abort, dce_abort.* The variable *dce_good_bcc*, for example, equates to DCE GDBCC.

   You can use both *dte* and *dce* variables relating to the same event in one conditions block. Suppose you want to count all bad BCC's from either side of the line. Enter the following CONDITIONS/ACTIONS block:

```
CONDITIONS:
{
  dte_bad_bcc || dce_bad_bcc
}
ACTIONS: COUNTER bad_bcc INC
```

Table 71-1
X.25 Layer 2 Structures

Type	Variable	Value (hex/decimal)	Meaning
Structure Name: send_frame_structure			Structure of a frame in X.25. Declared as type struct. Declared automatically if a softkey-entered SEND action is taken. Program frames assigned to structure as follows: struct send_frame_structure *name*. Reference a structure variable as follows: *name*.bcc_type. If values in the frame structure are not initialized by the user, they default to 0. You may initialize the values when the structure is declared: struct send_frame_structure *name* = {1, 1, 1, 0, 1, 1, 3, 0x71, 3, 0};
unsigned char	addr_type	0 1 2	command response other
unsigned char	frame_type	*(The codes for frame_type are the same as for the X.25-variable rcvd_frame_type.)*	
unsigned char	nr_type	0 1 2 3	auto value received ns plus 1 last nr sent
unsigned char	ns_type	0 1 2 3	auto skip last nr received value
unsigned char	p_f_type	0 1 2	0 1 loopback
unsigned char	bcc_type	0 1 2 3	default (bad bcc) good bcc bad bcc abort
unsigned char	addr_value	1 3	to DCE to DTE
unsigned char	cntrl_byte	*(actual value of the control byte)*	
unsigned char	nr_value	*0-7 (MOD 8)*	if nr_type = 1
unsigned char	ns_value	*0-7 (MOD 8)*	if ns_type = 3

Table 71-2
X.25 Layer 2 Variables

Type	Variable	Value (hex/decimal)	Meaning
extern event	dte_frame		True when a DTE frame is detected. Line Setup configured for emulate or monitor mode.
extern event	dce_frame		True when a DCE frame is detected. Line Setup configured for emulate or monitor mode.
extern event	dte_good_bcc		True when a good BCC is calculated for a DTE frame. Line Setup configured for emulate or monitor mode.
extern event	dce_good_bcc		True when a good BCC is calculated for a DCE frame. Line Setup configured for emulate or monitor mode.
extern event	dte_bad_bcc		True when a bad BCC is calculated for a DTE frame. Line Setup configured for emulate or monitor mode.
extern event	dce_bad_bcc		True when a bad BCC is calculated for a DCE frame. Line Setup configured for emulate or monitor mode.
extern event	dte_abort		True when an abort is detected for a DTE frame. Line Setup configured for emulate or monitor mode.
extern event	dce_abort		True when an abort is detected for a DCE frame. Line Setup configured for emulate or monitor mode.
extern event	rcvd_frame		True when a frame is received. Line Setup configured for emulate mode only.
extern event	invalid_frame		True when an invalid frame is detected. Line Setup configured for emulate mode only.
extern event	l2_T1		True when the T1 timeout-timer has expired. Line Setup configured for emulate mode only.
extern event	bcc_error		True when a BCC error is detected. Line Setup configured for emulate mode only.
extern event	nr_error		True when an N(R) error is detected in a received INFO or supervisory frame. Line Setup configured for emulate mode only.
extern event	ns_error		True when an N(S) error is detected in a received INFO frame. Line Setup configured for emulate mode only.

Table 71-2 (continued)

Type	Variable	Value (hex/decimal)	Meaning
extern event	frame_sent		True when frame is passed down to Layer 1. Line Setup configured for emulate mode only.
extern volatile const unsigned char	m_frame_addr	1 3	to DCE to DTE Line Setup configured for emulate or monitor mode.
extern volatile const unsigned char	m_frame_type	*(same as rcvd_frame_type—Line Setup configured for emulate or monitor mode)*	
extern volatile const unsigned char	m_frame_cntrl_byte_1	*(actual value of control byte—Line Setup configured for emulate or monitor mode)*	
extern volatile const unsigned char	m_frame_pf	0 10/16	pf=0 pf=1 Line Setup configured for emulate or monitor mode.
extern volatile const unsigned char	m_frame_bcc_type	1 2 3	good bad abort Line Setup configured for emulate or monitor mode.
extern volatile const unsigned char	rcvd_frame_addr	1 3	to DCE to DTE Line Setup configured for emulate mode only.
extern volatile const unsigned char	rcvd_frame_type	0 1 5 9 d/13 2f/47 6f/111 43/67 f/15 f/15 63/99 87/135 ff/255 ff/255	info rr rnr rej srej sabm sabme disc dm sarm ua frmr other unknown Line Setup configured for emulate mode only.
extern volatile const unsigned char	rcvd_frame_cntrl_byte_1	*(actual value of control byte—Line Setup configured for emulate mode only)*	
extern volatile const unsigned char	rcvd_frame_pf	0 10/16	pf=0 pf=1 Line Setup configured for emulate mode only.
extern volatile const unsigned char	rcvd_frame_bcc_type	1 2 3	good bad abort Line Setup configured for emulate mode only.
extern volatile const unsigned char	rcvd_frame_nr	*0–7 (MOD 8)*	Line Setup configured for emulate mode only.

Table 71-2 (continued)

Type	Variable	Value (hex/decimal)	Meaning
extern volatile const unsigned char	rcvd_frame_ns	*0-7 (MOD 8)*	Line Setup configured for emulate mode only.
extern volatile unsigned short	rcvd_frame_buff_seg		Inter-layer message buffer number (actually, an IAPX-286 segment number) in a received frame. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate mode only.
extern volatile unsigned short	rcvd_frame_sdu_offset		Offset to where the service data unit begins in an inter-layer message buffer in a received frame. Add to buffer segment number (converted to pointer) to point to first byte in frame. Line Setup configured for emulate mode only.
extern volatile unsigned short	rcvd_frame_sdu_size		Size of service data unit in a received frame. Line Setup configured for emulate mode only.
extern volatile unsigned short	l2_current_window_edge		When equal to upper edge, window is full; when equal to lower edge, window is empty; when not equal to upper edge, window is not full; and when not equal to lower edge, window is not empty. Line Setup configured for emulate mode only.
extern volatile unsigned short	l2_lower_window_edge		see l2_current_window_edge
extern volatile unsigned short	l2_upper_window_edge		see l2_current_window_edge
extern volatile unsigned short	l2_resend_edge		When resend edge is not equal to lower window edge, there is more to resend; when resend edge is equal to lower window edge, there is no more to resend. Line Setup configured for emulate mode only.
extern unsigned char	l2_enhance	0 1 4 5 8 9 12/18	normal reverse low reverse low blink reverse blink blink low Line Setup configured for emulate or monitor mode.
extern unsigned char	l2_suppress	0 1	off on Line Setup configured for emulate or monitor mode.

Using spreadsheet tokens, the same test needs two CONDITIONS/ACTIONS blocks:

```
CONDITIONS: DTE BDBCC
ACTIONS: COUNTER bad_bcc INC
CONDITIONS: DCE BDBCC
ACTIONS: COUNTER bad_bcc INC
```

When the user selects DTE or DCE on the first rack of softkeys for Layer 2 conditions, a second rack appears from which he must select a particular frame type. A DTE INFO condition, for example, when translated, includes two C variables, one event variable and one status variable:

```
{
 dte_frame && (m_frame_type == 0)
}
```

As a C programmer, you do not need to specify a frame type. To include all frames in a condition, use the event variable only:

```
CONDITIONS:
{
 dte_frame
}
```

2. *Emulate mode only*. Some events may be detected in emulate mode only. The event variables are *rcvd_frame*, *invalid_frame*, *l2_T1*, *bcc_error*, *nr_error*, *ns_error*, and *frame_sent*.

   If you try to use one of these variables in monitor mode, you may be returned to the main program menu. When you go to the Protocol Spreadsheet and search for errors, a message like the following may be displayed: "*Error 140: Unresolved reference rcvd_frame.*"

   When the user selects RCV on the first rack of softkeys for Layer 2 conditions, a second rack appears from which he must select a particular frame type. When the translator converts a RCV INFO condition into C, it will include two C variables, one event variable and one status variable:

```
{
 rcvd_frame && (rcvd_frame_type == 0)
}
```

   The C programmer does not have to specify a frame type. To include all received frames in a condition, use the event variable only:

```
CONDITIONS:
{
 rcvd_frame
}
```

   Error detecting may be accomplished via *bcc_error*, *nr_error*, *ns_error*, and *invalid_frame*. These variables equate to the softkey tokens bearing similar names.

One of the emulate-mode variables monitors an emulate action. The event variable *frame_sent* will come true as soon as the frame has been passed to the layer below. Note that if Layer 1 is an X.21 protocol in call-setup phase, a frame that is "sent" at Layer 2 will stop at Layer 1 and will not be transmitted out onto the line.

(B) Status Variables

Status variables are those in Table 71-2 that do not include *event* in the Type column. Their associated event variables guarantee that they are updated and tested.

The softkey-generated condition for received Info frames is RCV INFO. The C version of the same condition should look like this:

```
CONDITIONS:
{
 rcvd_frame && (rcvd_frame_type == 0)
}
```

1. *Frame characteristics*. All status variables in Table 71-2 containing an *m_* prefix are valid in either emulate or monitor mode: *m_frame_addr*, *m_frame_type*, *m_frame_cntrl_byte_1*, *m_frame_pf*, and *m_frame_bcc_type*. Use these variables to monitor a particular address, frame type, control byte, P/F value, or BCC.

   All status variables in Table 71-2 containing a *rcvd_* prefix are valid in emulate mode only: *rcvd_frame_addr*, *rcvd_frame_type*, *rcvd_frame_cntrl_byte_1*, *rcvd_frame_bcc_type*, *rcvd_frame_pf*, *rcvd_frame_nr*, and *rcvd_frame_ns*. Use these variables to monitor a particular address, frame type, control byte, BCC, or P/F, N(R), or N(S) value.

   If you try to use an emulate-mode variable in monitor mode, you may be returned to the main program menu. When you go to the Protocol Spreadsheet and search for errors, a message like the following may be displayed: "*Error 140: Unresolved reference rcvd_frame_type.*"

2. *Frame buffers*. As BOP frames are received, they are automatically placed in IL message buffers to be passed up the layers. Three emulate-mode variables provide the user with access to the information in the frame that is located beyond the control byte. These variables are *rcvd_frame_buff_seg*, *rcvd_frame_sdu_offset*, and *rcvd_frame_sdu_size*. See Section 63.1 for a more detailed discussion of the buffer components to which these variables refer.

Make a pointer to an IL buffer by casting *rcvd_frame_buff_seg* as a *long*, shifting it left sixteen bits, adding *rcvd_frame_sdu_offset*, and casting the result to a pointer. Increment the pointer twice (thereby adding two to the offset).

```
{
 unsigned char * ptr;
 ptr = (void *)(((long)rcvd_frame_buff_seg << 16) + rcvd_frame_sdu_offset);
 ptr+=2;
}
```

It is now pointing at the first byte in the X.25 Layer 3 header. You may continue to move through the frame for its entire length, indicated in *rcvd_frame_sdu_size*.

3. *Transmit window*. Four related variables test the status of the Layer 2 window. The particular values of these variables at any given time is not significant. What is significant is how they compare to each other. The softkey status condition on the left makes the variable comparison on the right:

WINDOW FULL	*l2_current_window_edge == l2_upper_window_edge*
WINDOW EMPTY	*l2_current_window_edge == l2_lower_window_edge*
WINDOW NOT_FULL	*l2_current_window_edge != l2_upper_window_edge*
WINDOW NOT_EMPTY	*l2_current_window_edge != l2_lower_window_edge*
MORE_TO_RESEND	*l2_resend_edge != l2_lower_window_edge*
NO_MORE_TO_RESEND	*l2_resend_edge == l2_lower_window_edge*

(C) Controlling Protocol Trace Display

To enhance or suppress particular frames on the Layer 2 Protocol Trace screen in emulate or monitor mode, assign a coded value to *l2_enhance* or *l2_suppress*. The values are listed in Table 71-2. To assign a value to either of these variables, place the statement in an ACTIONS block. For example, display RNR frames in reverse-video and suppress display of invalid frames:

```
CONDITIONS: RCV RNR
ACTIONS:
{
 l2_enhance = 1;
}
CONDITIONS: RCV INVALID
ACTIONS:
{
 l2_suppress = 1;
}
```

Check the value of these display–control variables in a CONDITIONS block

```
CONDITIONS: RCV INFO
{
 l2_enhance == 1
}
ACTIONS:
{
 l2_enhance = 0;
}
```

or an ACTIONS block:

```
CONDITIONS: RCV INFO
ACTIONS:
{
 if(l2_enhance == 1)
   l2_enhance = 0;
}
```

71.3 Routines

Use the following routines in emulate mode only. If you try to call one of these routines in monitor mode, you may be returned to the main program menu. When you go to the Protocol Spreadsheet and search for errors, a message like the following may be displayed: "*Error 140: Unresolved reference l2_give_data.*"

(A) Receive

l2_give_data

Synopsis

```
extern void l2_give_data();
```

Description

The *l2_give_data* routine takes an interlayer message buffer associated with a received INFO frame, changes the SDU offset to point to higher–level data, and sends a DL_DATA IND primitive up to Layer 3 along with a reference to this buffer. The softkey equivalent of this routine is the GV_DATA action on the Protocol Spreadsheet.

Example

Layer 3 wants access to the line in order to receive and send data. Assuming X.25 personality packages are loaded at Layers 2 and 3, enter the following program:

```
LAYER: 2
    STATE: datalink
      CONDITIONS: DL_CONNECT REQ
      ACTIONS: DL_CONNECT CONF
      CONDITIONS: DL_DATA REQ
      ACTIONS: SEND INFO "((DL_DATA))"
      CONDITIONS: RCV INFO
      ACTIONS:
      {
       l2_give_data();
      }
```

(B) Transmit

resend_frame

Synopsis

extern void resend_frame(pf, first_or_next);
unsigned char pf;
unsigned char first_or_next;

Description

The *resend_frame* routine will set the P/F bit to a specified value and resend either the first or next frame in the window. The softkey equivalent of this routine is the (PROTOCL) RESEND action on the Protocol Spreadsheet.

Inputs

The first parameter is the value of the P/F bit in the frame. It may be set to either 0, 1, or 2 (for loopback).

The second parameter indicates whether the first frame in the window will be sent, or whether the next frame in the window will be sent. The first resend action will send the first frame in the window regardless of whether first or next has been selected. Legal entries are 0 (first) or 1 (next).

Example

Suppose you want to resend the entire transmit window if you receive a REJ frame.

```
LAYER: 2
    STATE: xfer
      /* Whatever conditions and actions send data precede the following condition. */

      CONDITIONS: RCV REJ RESP
      ACTIONS:
      {
       resend_frame(1, 0);
      }
      NEXT_STATE: recover
```

```
STATE: recover
   CONDITIONS: FRAME_SENT
        MORE_TO_RESEND
   ACTIONS:
   {
    resend_frame(1,1);
   }
   CONDITIONS: FRAME_SENT
        NO_MORE_TO_RESEND
   NEXT_STATE: xfer
```

reset_nr

Synopsis

extern void reset_nr();

Description

This routine resets the N(R) field in information and supervisory frames to zero. The softkey equivalent of this routine is the (PROTOCL) RSET_NR action on the Protocol Spreadsheet.

Example

When a link is established, reset N(R).

```
LAYER: 2
   STATE: reset
      CONDITIONS: ENTER_STATE
      ACTIONS: SEND SABM
      CONDITIONS: RCV UA
      ACTIONS:
      {
       reset_nr();
      }
```

reset_ns

Synopsis

extern void reset_ns();

Description

The N(S) field in information and supervisory frames is reset to zero and the transmit window is cleared. The softkey equivalent of this routine is the (PROTOCL) RSET_NS action on the Protocol Spreadsheet.

Example

When a link is established, reset N(S).

```
LAYER: 2
    STATE: reset
        CONDITIONS: ENTER_STATE
        ACTIONS: SEND SABM
        CONDITIONS: RCV UA
        ACTIONS:
        {
         reset_ns();
        }
```

send_frame

Synopsis

```
extern void send_frame(il_buffer_number, relay_baton, data_start_offset, transmit_frame_ptr);
unsigned short il_buffer_number;
unsigned short relay_baton;
unsigned short data_start_offset;
struct send_frame_structure
{
 unsigned char addr_type;
 unsigned char frame_type;
 unsigned char nr_type;
 unsigned char ns_type;
 unsigned char p_f_type;
 unsigned char bcc_type;
 unsigned char addr_value;
 unsigned char cntrl_byte;
 unsigned char nr_value;
 unsigned char ns_value;
};
struct send_frame_structure * transmit_frame_ptr;
```

Description

The *send_frame* routine adds a frame-level header to an interlayer message buffer and passes the buffer to Layer 1. The softkey equivalent of this routine is the SEND action on the Protocol Spreadsheet.

Inputs

The first parameter is the interlayer message buffer number. See Section 63.3(A), Layer-Independent OSI routines.

The second parameter is the maintain bit used to hold the buffer while the send operation is being performed. See Section 63.3(A).

The third parameter is the offset from the beginning of the buffer to the start of the service data unit. See Section 63.3(A).

The fourth parameter is a pointer to the frame structure to be sent. For a description of *send_frame_structure*, see Table 71-1.

Example

Send an Info frame containing a canned fox message and a good BCC onto the line.

```
{
 static unsigned short il_buffer_number;
 static unsigned short relay_baton;
 static unsigned short data_start_offset;
 struct send_frame_structure
  {
   unsigned char addr_type;
   unsigned char frame_type;
   unsigned char nr_type;
   unsigned char ns_type;
   unsigned char p_f_type;
   unsigned char bcc_type;
   unsigned char addr_value;
   unsigned char cntrl_byte;
   unsigned char nr_value;
   unsigned char ns_value;
  };
 struct send_frame_structure transmit_frame;
 static char transmit_string [] = "《FOX》";
}
LAYER: 2
      STATE: send_a_frame
         CONDITIONS: KEYBOARD " "
         ACTIONS:
         {
          _get_il_msg_buff(&il_buffer_number, &relay_baton);
          _start_il_buff_list(il_buffer_number,&data_start_offset);
          transmit_frame.bcc_type  = 1;
          _insert_il_buff_list_cnt(il_buffer_number, data_start_offset, &transmit_string[0],
                (sizeof(transmit_string) - 1));
          send_frame(il_buffer_number, relay_baton, data_start_offset, &transmit_frame);
         }
```

72 X.25 Layer 3 Library

When the X.25 Layer 3 package is loaded in via the Layer Setup screen, the following external routines and variables become available for use by the programmer. Their use on the Protocol Spreadsheet is not limited to any particular layer, though normally they belong at Layer 3.

The variables and routines approximate X.25 Layer 3 spreadsheet-generated conditions and actions. Refer to Section 34 for more detailed explanations of the purposes of specific conditions and actions. Sometimes the name of the variable or routine is sufficient for identifying its related spreadsheet token. When this is not the case, the information is provided below.

72.1 Structures

The *send_packet_structure* defines the format of transmitted X.25 packets. See Table 72-1. Use this structure to send packets via the *send_packet* routine in emulate mode. See Section 72.3(B). Each variable in the structure relates to some softkey selection or user entry in the SEND action.

72.2 Variables

(A) Monitoring Events

1. *Emulate or monitor mode.* Two X.25 Layer 3 event variables are valid in either emulate or monitor mode. These event variables are *dte_packet* and *dce_packet*.

   When the user selects DTE or DCE on the first rack of softkeys for Layer 3 conditions, a second rack appears from which he must select a particular packet type. A DTE DATA condition, for example, when translated, includes two C variables, one event variable and one status variable:

   ```
   {
    dte_packet && (m_packet_type == 0)
   }
   ```

Table 72-1
X.25 Layer 3 Structures

Type	Variable	Value (hex/decimal)	Meaning
Structure Name:	**send_packet_structure**		Structure of a packet in X.25. Declared as type struct. Declared automatically if a softkey-entered SEND action is taken. Program packets assigned to structure as follows: struct send_packet_structure *name*. Reference a structure variable as follows: *name*.q_bit. If values in the frame structure are not initialized by the user, they default to 0. You may initialize the values when the structure is declared: struct send_packet_structure *name* = {2, 0x13, 0x13, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 2, &*facilities_string*[0], 0, 0};
unsigned char	path_num	*0-8* fe/254	path number use path number of last received packet
unsigned char	packet_type	*(The codes for packet_type are the same as for the X.25-variable m_packet_type.)*	
unsigned char	packet_type_byte	*(actual value of the packet type byte)*	
unsigned char	m_bit	0 1	m = 0 m = 1
unsigned char	d_bit	0 40/64	d = 0 d = 1
unsigned char	q_bit	0 80/128	q = 0 q = 1
unsigned char	pr_type	0 1 2 3	auto value received ps plus 1 last pr sent
unsigned char	ps_type	0 1 2 3	auto skip received pr value
unsigned char	pr_value	*0-7 (MOD 8)*	if pr_type = 1
unsigned char	ps_value	*0-7 (MOD 8)*	if ps_type = 3
unsigned char	cause	*(value of cause byte—see Figure 36-15)*	
unsigned char	diag_flag	0 1	diagnostic field not present diagnostic field is present
unsigned char	diag	*(value of diagnostic byte—consult CCITT Recommendation X.25, pp. 237-8)*	
unsigned char	spare	0	reserved space
unsigned char	facilities_len	*0-ff/0-255*	length of the facilities field
char *	facilities		pointer to the location of the facilities field—the facilities field is declared separately
unsigned short	data_len		reserved for future use
char *	data		reserved for future use

Table 72-2
X.25 Layer 3 Variables

Type	Variable	Value (hex/decimal)	Meaning
extern event	dte_packet		True when a DTE packet is detected. Line Setup configured for emulate or monitor mode.
extern event	dce_packet		True when a DCE packet is detected. Line Setup configured for emulate or monitor mode.
extern event	rcvd_packet		True when a packet is received from Layer 2. Line Setup configured for emulate mode only.
extern event	invalid_packet		True when an invalid packet is detected. Line Setup configured for emulate mode only.
extern event	pr_error		True when an P(R) error is detected in a data or supervisory packet. Line Setup configured for emulate mode only.
extern event	ps_error		True when an P(S) error is detected in a data packet. Line Setup configured for emulate mode only.
extern event	packet_sent		True when a packet has been passed down to Layer 2. Line Setup configured for emulate mode only.
extern volatile unsigned short	m_packet_lcn	*0-fff/0-4095*	Logical channel number. Line Setup configured for emulate or monitor mode.
extern volatile unsigned char	m_packet_lcn_grp	*0-f/0-15*	Logical channel group number. Line Setup configured for emulate or monitor mode.
extern volatile const unsigned char	m_packet_q	0 80/128	q=0 q=1 Line Setup configured for emulate or monitor mode.
extern volatile const unsigned char	m_packet_d	0 40/64	d=0 d=1 Line Setup configured for emulate or monitor mode.
extern volatile const unsigned char	m_packet_m	0 10/16	m=0 m=1 Line Setup configured for emulate or monitor mode.
extern volatile const unsigned char	m_packet_pr	*0-7 (MOD 8)*	Line Setup configured for emulate or monitor mode.
extern volatile const unsigned char	m_packet_ps	*0-7 (MOD 8)*	Line Setup configured for emulate or monitor mode.

Table 72-2 (continued)

Type	Variable	Value (hex/decimal)	Meaning
extern volatile const unsigned char	m_packet_cause	*(same as rcvd_pkt_cause—Line Setup configured for emulate or monitor mode)*	
extern volatile const unsigned char	m_packet_diag_code	*(same as rcvd_pkt_diagn—Line Setup configured for emulate or monitor mode)*	
extern volatile const unsigned char	m_packet_type_byte	*(actual value of packet type byte—Line Setup configured for emulate or monitor mode)*	
extern volatile const unsigned char	m_packet_type	b/11	call
		f/15	call acc
		13/19	clear
		17/23	clear conf
		0	data
		23/35	int
		27/39	int conf
		1	rr
		5	rnr
		9	rej
		1b/27	reset
		1f/31	reset conf
		fb/251	restart
		ff/255	restart conf
		f1/241	diag
		f3/243	reg
		f7/247	reg conf
		11/17	other pkt
		11/17	unknown pkt
			Line Setup configured for emulate or monitor mode.
extern volatile unsigned short	rcvd_pkt_lcn	*0-fff/0-4095*	Logical channel number in a received packet. Line Setup configured for emulate mode only.
extern volatile const unsigned char	rcvd_pkt_q	0	q=0
		80/128	q=1
			Line Setup configured for emulate mode only.
extern volatile const unsigned char	rcvd_pkt_d	0	d=0
		40/64	d=1
			Line Setup configured for emulate mode only.
extern volatile const unsigned char	rcvd_pkt_m	0	m=0
		10/16	m=1
			Line Setup configured for emulate mode only.
extern volatile const unsigned char	rcvd_pkt_pr	*0-7 (MOD 8)*	Line Setup configured for emulate mode only.
extern volatile const unsigned char	rcvd_pkt_ps	*0-7 (MOD 8)*	Line Setup configured for emulate mode only.
extern volatile const unsigned char	rcvd_pkt_cause	*(see Figure 36-15—Line Setup configured for emulate mode only)*	
extern volatile const unsigned char	rcvd_pkt_diagn	*(consult CCITT Recommendation X.25, pp.237-8—Line Setup configured for emulate mode only)*	

Table 72-2 (continued)

Type	Variable	Value (hex/decimal)	Meaning
extern volatile const unsigned char	rcvd_pkt_type_byte	*(actual value of packet type byte—Line Setup configured for emulate mode only)*	
extern volatile const unsigned char	rcvd_packet_type	b/11	call
		f/15	call acc
		13/19	clear
		17/23	clear conf
		0	data
		23/35	int
		27/39	int conf
		1	rr
		5	rnr
		9	rej
		1b/27	reset
		1f/31	reset conf
		fb/251	restart
		ff/255	restart conf
		f1/241	diag
		f3/243	reg
		f7/247	reg conf
		11/17	other pkt
		11/17	unknown pkt
			Line Setup configured for emulate mode only.
extern volatile unsigned short	m_packet_buff_seg		Inter-layer message buffer number (actually, an IAPX-286 segment number). This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate or monitor mode.
extern volatile unsigned short	m_packet_info_seg		Same as *m_packet_buff_seg*.
extern volatile unsigned short	m_packet_sdu_offset		Offset to where the service data unit begins in an inter-layer message buffer. Add to *m_pkt_buff_seg* (converted to pointer) to point to first packet-header byte. Line Setup configured for emulate or monitor mode.
extern volatile unsigned short	m_packet_info_offset		Offset to where the packet information begins, excluding the header. Add to *m_pkt_buff_seg* (converted to pointer) to point to packet data. Line Setup configured for emulate or monitor mode.
extern volatile unsigned short	m_packet_length		Length of the packet, including header. Line Setup configured for emulate or monitor mode.
extern volatile unsigned short	m_packet_info_length		Length of the packet information, excluding the header. Line Setup configured for emulate or monitor mode.

Table 72-2 (continued)

Type	Variable	Value (hex/decimal)	Meaning
extern volatile unsigned short	rcvd_pkt_buff_seg		Inter-layer message buffer number (actually, an IAPX-286 segment number) in a received packet. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate mode only.
extern volatile unsigned short	rcvd_pkt_info_seg		Same as *rcvd_pkt_buff_seg*.
extern unsigned short	rcvd_pkt_sdu_offset		Offset to where the service data unit begins in an inter-layer message buffer in a packet received. Add to *rcvd_pkt_buff_seg* (converted to pointer) to point to first packet-header byte. Line Setup configured for emulate mode only.
extern volatile unsigned short	rcvd_pkt_info_offset		Offset to where the packet information begins, excluding the header. Add to *rcvd_pkt_buff_seg* (converted to pointer) to point to packet data. Line Setup configured for emulate mode only.
extern unsigned short	rcvd_pkt_length		Length of a received packet, including header information. Line Setup configured for emulate mode only.
extern volatile unsigned short	rcvd_pkt_info_length		Length of the information in a received packet, excluding the header. Line Setup configured for emulate mode only.
extern volatile unsigned char *	m_packet_ptr		Pointer to the packet, beginning at the first byte in the header. Line Setup configured for emulate or monitor mode.
extern volatile unsigned char *	m_packet_info_ptr		Pointer to the information in a packet. Initially points to the byte immediately following the packet-type byte. Line Setup configured for emulate or monitor mode.
extern volatile unsigned char *	rcvd_packet_ptr		Pointer to the packet, beginning at the first byte in the header. Line Setup configured for emulate mode only.
extern volatile unsigned char *	rcvd_pkt_info_ptr		Pointer to the packet information, initially located at the byte immediately following the packet header. Line Setup configured for emulate mode only.

Table 72-2 (continued)

Type	Variable	Value (hex/decimal)	Meaning
extern volatile const unsigned char	rcvd_device_path		Path number connecting received packet to particular LCN and particular set of call parameters on the X.25 Packet Level Setup screen. Line Setup configured for emulate mode only.
extern unsigned char	l3_enhance	0 1 4 5 8 9 12/18	normal reverse low reverse low blink reverse blink blink low Line Setup configured for emulate or monitor mode.
extern unsigned char	l3_suppress	0 1	off on Line Setup configured for emulate or monitor mode.

A C programmer does not have to specify a packet type. To include all packets in a condition, use the event variable only:

```
CONDITIONS:
{
 dte_packet
}
```

2. *Emulate mode only*. Some events may be detected in emulate mode only. These are *rcvd_packet*, *invalid_packet*, *pr_error*, *ps_error*, and *packet_sent*.

   If you try to use one of these variables in monitor mode, you may be returned to the main program menu. When you go to the Protocol Spreadsheet and search for errors, a message like the following may be displayed: "*Error 140: Unresolved reference rcvd_packet.*"

   When the user selects RCV on the first rack of softkeys for Layer 3 conditions, a second rack appears from which he must select a particular packet type. When the translator converts a RCV DATA condition into C, it will include two C variables, one event variable and one status variable:

```
{
 rcvd_packet && (rcvd_packet_type == 0)
}
```

   As a C programmer, you do not have to specify a packet type. To include all received packets in a condition, use the event variable only:

```
CONDITIONS:
{
 rcvd_packet
}
```

Error detecting may be accomplished via *pr_error*, *ps_error*, and *invalid_packet*. These variables equate to the softkey tokens bearing similar names.

One of the emulate-mode variables monitors an emulate action. "SEND"ing a packet means queuing a packet to be passed down to Layer 2. If the Layer 2 link is not established, for example, the packet will be held at Layer 3 pending link establishment. The event variable *packet_sent* will not come true until the packet actually has been passed to the layer below. Use this condition to start accurate response-time measurements at the packet level rather than at the line level.

(B) Status Variables

Status variables are those in Table 72-2 that do not include *event* in the Type column. Their associated event variables guarantee that they are updated and tested.

The softkey-generated condition for received Data packets is **RCV DATA**. The C version of the same condition should look like this:

```
CONDITIONS:
{
rcvd_packet && (rcvd_packet_type == 0)
}
```

1. *Packet characteristics*. All status variables in Table 72-2 containing an *m_* prefix are valid in either emulate or monitor mode: *m_packet_lcn, m_packet_lcn_grp, m_packet_q, m_packet_d, m_packet_m, m_packet_pr, m_packet_ps, m_packet_cause, m_packet_diag_code, m_packet_type*, and *m_packet_type_byte*.

   All status variables in Table 72-2 containing a *rcvd_* prefix are valid in emulate mode only: *rcvd_pkt_lcn, rcvd_pkt_q, rcvd_pkt_d, rcvd_pkt_m, rcvd_pkt_pr, rcvd_pkt_ps, rcvd_pkt_cause, rcvd_pkt_diagn, rcvd_pkt_type*, and *rcvd_pkt_type_byte*.

   If you try to use an emulate-mode variable in monitor mode, you may be returned to the main program menu. When you go to the Protocol Spreadsheet and search for errors, a message like the following may be displayed: "*Error 140: Unresolved reference rcvd_packet_type.*"

2. *Packet buffers*. Packets are passed up to Layer 3 from Layer 2 in IL message buffers. Several variables provide the user with access to the information in the packet that is located beyond the packet-type byte. These variables are *rcvd_pkt_buff_seg, m_packet_buff_seg, rcvd_pkt_sdu_offset, m_packet_sdu_offset, rcvd_pkt_length*, and *m_packet_length*. See Section 63.1 for a more detailed discussion of the buffer components to which these variables refer.

3. *Pointers*. Two variables, *rcvd_pkt_info_ptr* and *m_packet_info_ptr*, point to the first byte beyond the packet header. You may move these pointers to

access data throughout the length of the packet. The length is indicated by *rcvd_pkt_info_length* (or *m_packet_info_length*).

4. *Path.* An IL buffer that is sent down the layers or received up the layers is provided with a "path" number that ties it, at X.25 Layer 3, to a particular LCN as well as to a particular set of Call Request parameters on the X.25 Packet Level Setup screen.

   When a call request is sent or received by the INTERVIEW, the call parameters are correlated to the Packet Level Setup screen. If the INTERVIEW sends a call request that specifies a path number, or if the INTERVIEW receives a call request that matches one of the path entries on the setup screen, the LCN of the call request is tied to the path number (path #3, for example), and any subsequent packets with the same LCN will satisfy *rcvd_device_path == 3* conditions.

(C) Controlling Protocol Trace Display

To enhance or suppress particular packets on the Layer 3 Protocol Trace screen in emulate or monitor mode, assign a coded value to *l3_enhance* or *l3_suppress*. The values are listed in Table 72-2. To assign a value to either of these variables, place the statement in an ACTIONS block. For example, display RNR packets in reverse-video and suppress display of invalid packets:

```
CONDITIONS: RCV RNR
ACTIONS:
{
 l3_enhance = 1;
}
CONDITIONS: RCV INVALID
ACTIONS:
{
 l3_suppress = 1;
}
```

Check the value of these display-control variables in a CONDITIONS block

```
CONDITIONS: RCV DATA
{
 l3_enhance == 1
}
ACTIONS:
{
 l3_enhance = 0;
}
```

or an ACTIONS block:

```
CONDITIONS: RCV DATA
ACTIONS:
{
 if(l3_enhance == 1)
   l3_enhance = 0;
}
```

72.3 Routines

Use the following routines in emulate mode only. If you try to call one of these routines in monitor mode, you may be returned to the main program menu. When you go to the Protocol Spreadsheet and search for errors, a message like the following may be displayed: *"Error 140: Unresolved reference l3_give_data."*

(A) Receive

l3_give_data

Synopsis

extern void l3_give_data();

Description

The *l3_give_data* routine takes an interlayer message buffer associated with a received data packet, changes the SDU offset to point to higher-level data, and sends an N_DATA IND primitive up to Layer 4 along with a reference to this buffer. The softkey equivalent of this routine is the GV_DATA action on the Protocol Spreadsheet.

Example

Layer 4 wants access to the line in order to receive and send data. Assuming X.25 personality packages are loaded at Layers 2 and 3, enter the following program:

```
LAYER: 2
    STATE: datalink
       CONDITIONS: DL_CONNECT REQ
       ACTIONS: DL_CONNECT CONF
       CONDITIONS: DL_DATA REQ
       ACTIONS: SEND INFO "((DL_DATA))"
       CONDITIONS: RCV INFO
       ACTIONS: GIVE_DATA
LAYER: 3
    STATE: pass_data_up
       CONDITIONS: N_CONNECT REQ
       ACTIONS: SEND CALL
       CONDITIONS: RCV CALL_CONF
       ACTIONS: N_CONNECT IND
       CONDITIONS: N_DATA REQ
       ACTIONS: SEND DATA "((N_DATA))"
       CONDITIONS: RCV DATA
       ACTIONS:
       {
        l3_give_data();
       }
LAYER: 4
    STATE: establish_link
       CONDITIONS: ENTER_STATE
       ACTIONS: N_CONNECT REQ
```

(B) Transmit

l3_clear_path

Synopsis

extern void l3_clear_path(path_number);
unsigned char path_number;

Description

The *l3_clear_path* routine resets P(R)- and P(S)-related variables, clears the transmit window, and resets the LCN and address fields to void (unless permanently assigned on the Layer 3 X.25 Packet Level Setup screen) on a designated path.

Inputs

The only parameter is the path number which is to be cleared. The value may be 0 - 8, or 0xfe if you want the path number to be that of the last received packet.

Example

When a Clear packet is received, clear the path.

```
LAYER: 3
     STATE: clearing
        CONDITIONS: RCV CLEAR
        ACTIONS: SEND CLEAR_CONF
        {
         l3_clear_path(0xfe);
        }
```

l3_more_to_resend

Synopsis

extern unsigned char l3_more_to_resend(path_number);
unsigned char path_number;

Description

The *l3_more_to_resend* routine determines whether or not there are any more packets in the transmit window to resend. It is used in combination with a transitional condition such as *packet_sent* as a condition on the Protocol Spreadsheet. The softkey equivalent is PACKET_SENT MORE_TO_RESEND or PACKET_SENT NO_MORE_TO_RESEND.

Inputs

The only parameter is the path number associated with the transmit window. The value may be 0 – 8, or 0xfe if you want the path number to be that of the last received packet.

Returns

If there is more to resend, the returned value is non-zero. If there is no more to resend, or if the given path is invalid, the returned value is 0.

Example

In this example, the entire transmit window will be resent.

```
{
 extern event packet_sent;
}
LAYER: 3
     STATE: xfer
        /* Whatever conditions and actions send data precede the following condition. */

        CONDITIONS: RCV REJ
        ACTIONS: RESEND_FIRST
        NEXT_STATE: recover
     STATE: recover
        CONDITIONS: ENTER_STATE
        {
         packet_sent &&(l3_more_to_resend(0xfe) != 0)
        }
        ACTIONS: RESEND_NEXT
        CONDITIONS:
        {
         packet_sent &&(l3_more_to_resend(0xfe) == 0)
        }
        NEXT_STATE: xfer
```

l3_window_full

Synopsis

extern unsigned char l3_window_full(path_number);
unsigned char path_number;

Description

This routine determines whether the Layer 3 window for a specified path is full or not full. When the window is full, no additional packets will be buffered until some acknowledgment is received. It is used in combination with a transitional condition such as *receive_packet* as a condition on the Protocol Spreadsheet. The softkey equivalent is RCV RR (PROTOCL) WINDOW NOT_FULL or RCV RR (PROTOCL) WINDOW FULL.

Inputs

The only parameter is the path number whose window is to be checked. The value may be 0 – 8, or 0xfe if you want the path number to be that of the last received packet.

Returns

If the window is full, or if the given path is invalid, the returned value is non-zero. If the window is not full, the returned value is 0.

Example

Transmit data packets until the transmit window is full.

```
{
 extern event packet_sent;
}
LAYER: 3
     STATE: check_window
        CONDITIONS:
        {
         packet_sent && (l3_window_full(0xfe) != 0)
        }
        ACTIONS: SEND DATA "((FOX))"
```

l3_window_empty

Synopsis

extern unsigned char l3_window_empty(path_number);
unsigned char path_number;

Description

This routine determines whether the Layer 3 window for a specified path is empty or not empty. It is used in combination with a transitional condition such as *receive_packet* as a condition on the Protocol Spreadsheet. The softkey equivalent is RCV RR (PROTOCL) WINDOW NOT_EMPTY or RCV RR (PROTOCL) WINDOW EMPTY.

Inputs

The only parameter is the path number whose window is to be checked. The value may be 0 – 8, or 0xfe if you want the path number to be that of the last received packet.

Returns

If the window is empty, or if the given path is invalid, the returned value is non-zero. If the window is not empty, the returned value is 0.

Example

If a timeout expires and the transmit window is not empty, resend the first packet in the window.

```
{
 extern event timeout_ack_expired;
 extern event rcvd_packet;
}
LAYER: 3
     STATE: check_window
        CONDITIONS: PACKET_SENT
        ACTIONS: TIMEOUT ack RESTART
        CONDITIONS:
        {
         rcvd_packet
        }
        ACTIONS: TIMEOUT ack STOP
        CONDITIONS:
        {
         timeout_ack_expired && (l3_window_empty(0xfe) != 0)
        }
        ACTIONS: RESEND FIRST
```

resend_packet

Synopsis

```
extern void resend_packet(path_number, first_or_next);
unsigned char path_number;
unsigned char first_or_next;
```

Description

The *resend_packet* routine will resend either the first or next packet in the window along a specified path. The softkey equivalent of this routine is the RESEND action on the Protocol Spreadsheet.

Inputs

The first parameter is the value of the path on which to resend the packet. It may be 0 - 8, or 0xfe for the path of the last received packet.

The second parameter indicates whether the first packet in the window will be sent, or whether the next packet in the window will be sent. The first resend action will send the first packet in the window regardless of whether first or next has been selected. Legal entries are 0 (first) or 1 (next).

Example

Suppose you want to resend the entire transmit window if you receive a REJ packet. In this example, it's being sent along the path of the last received packet.

```
LAYER: 3
    STATE: xfer
      /* Whatever conditions and actions send data precede the following condition. */

      CONDITIONS: RCV REJ
      ACTIONS:
      {
       resend_packet(0xfe, 0);
      }
      NEXT_STATE: recover
    STATE: recover
      CONDITIONS: PACKET_SENT
          MORE_TO_RESEND
      ACTIONS:
      {
       resend_packet(0xfe,1);
      }
      CONDITIONS: PACKET_SENT
          NO_MORE_TO_RESEND
      NEXT_STATE: xfer
```

reset_pr_ps

Synopsis

extern void reset_pr_ps(path_number);
unsigned char path_number;

Description

The P(R) and P(S) fields in data and supervisory packets are reset to zero. The transmit window is also cleared. The softkey equivalent of this routine is the (PROTOCL) RSTPRPS action on the Protocol Spreadsheet.

Inputs

The only parameter is the path number on which P(R) and P(S) are to be reset. The value may be 0 – 8, or 0xfe if you want the path number to be that of the last received packet.

Example

In this example, P(R) and P(S) are reset on path 2 whenever a Reset packet is received.

```
LAYER: 3
    STATE: reset
      CONDITIONS: RCV RESET
      ACTIONS:
      {
       reset_pr_ps(2);
      }
```

send_packet

Synopsis

```
extern void send_packet(il_buffer_number, relay_baton, data_start_offset,
     transmit_packet_ptr);
unsigned short il_buffer_number;
unsigned short relay_baton;
unsigned short data_start_offset;

struct send_packet_structure
{
 unsigned char path_num;
 unsigned char packet_type;
 unsigned char packet_type_byte;
 unsigned char m_bit;
 unsigned char d_bit;
 unsigned char q_bit;
 unsigned char pr_type;
 unsigned char ps_type;
 unsigned char pr_value;
 unsigned char ps_value;
 unsigned char cause;
 unsigned char diag_flag
 unsigned char diag;
 unsigned char cntrl_byte;
 unsigned char facilities_len;
 char * facilities;
 unsigned short data_len;
 char * data;
};
struct send_packet_structure * transmit_packet_ptr;
```

Description

The *send_packet* routine adds a packet-level header to an interlayer message buffer and passes the buffer to Layer 2. The softkey equivalent of this routine is the SEND action on the Protocol Spreadsheet.

Inputs

The first parameter is the interlayer message buffer number. See Section 63.3(A), Layer-Independent OSI routines.

The second parameter is the maintain bit used to hold the buffer while the send operation is being performed. See See Section 63.3(A).

The third parameter is the offset from the beginning of the buffer to the start of the service data unit. See See Section 63.3(A).

The fourth parameter is a pointer to the packet structure to be sent. For a description of *send_packet_structure* see Table 72-1.

Example

To successfully send a packet out to the line, you must include the Layer 2 section of the program below. In this example, you are sending a Call Request packet with a facilities field present.

```
{
 static unsigned short il_buffer_number;
 static unsigned short relay_baton;
 static unsigned short data_start_offset;
 struct send_packet_structure
  {
   unsigned char path_num;
   unsigned char packet_type;
   unsigned char packet_type_byte;
   unsigned char m_bit;
   unsigned char d_bit;
   unsigned char q_bit;
   unsigned char pr_type;
   unsigned char ps_type;
   unsigned char pr_value;
   unsigned char ps_value;
   unsigned char cause;
   unsigned char diag_flag
   unsigned char diag;
   unsigned char cntrl_byte;
   unsigned char facilities_len;
   char * facilities;
   unsigned short data_len;
   char * data;
  };
 static char transmit_string [] = "((FOX))";
 static char facilities_string [] = "01010401414503430707";
 struct send_packet_structure transmit_packet = {0, 0x13, 0x13, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0,
       (sizeof(facilities_string)-1), &facilities_string[0], 0, 0};
}
LAYER: 2
     STATE: datalink
         CONDITIONS: DL_CONNECT REQ
         ACTIONS: DL_CONNECT CONF
         CONDITIONS: DL_DATA REQ
         ACTIONS: SEND INFO "((DL_DATA))"
         CONDITIONS: RCV INFO
         ACTIONS: GIVE_DATA
LAYER: 3
     STATE: send_a_packet
         CONDITIONS: KEYBOARD " "
         ACTIONS:
         {
          _get_il_msg_buff(&il_buffer_number, &relay_baton);
          _start_il_buff_list(il_buffer_number,&data_start_offset);
          _insert_il_buff_list_cnt(il_buffer_number, data_start_offset, &transmit_string[0],
                (sizeof(transmit_string) - 1));
          send_packet(il_buffer_number, relay_baton, data_start_offset, &transmit_packet);
         }
```

NOTE: A null is appended to the end of an array initialized as a string inside quotation marks; it is not appended to the end of an array entered inside curly braces. So, if *facilities_string* was initialized as a list of values, like this—

```
static char facilities_string [] = {1, 1, 4, 1, 0x41, 0x45, 0x03, 0x43, 7, 7};
```

—then *transmit_packet* would look like this—

```
struct send_packet_structure transmit_packet = {0, 0x13, 0x13, 0, 0, 0, 0, 0, 0, 0, 1,
      1, 0, 0, sizeof(facilities_string), &facilities_string[0], 0, 0};
```

73 SDLC Library

When the SDLC package is loaded in via the Layer Setup screen, the following external routines and variables become available for use by the programmer. Their use on the Protocol Spreadsheet is not limited to any particular layer, though normally they belong at Layer 2.

The variables and routines approximate SDLC Layer 2 spreadsheet-generated conditions and actions. Refer to Section 35 for more detailed explanations of the purposes of specific conditions and actions. Sometimes the name of the variable or routine is sufficient for identifying its related spreadsheet token. When this is not the case, the information is provided below.

73.1 Structures

The structure *send_frame_structure* defines the format of transmitted SDLC frames. See Table 73-1. Use this structure to send frames via the *send_frame* routine in emulate mode. See Section 73.3(B). Each variable in the structure relates to some softkey selection or user entry in the SEND action.

73.2 Variables

(A) Monitoring Events

1. *Emulate or monitor mode.* SDLC events include frames detected, good or bad BCC's, and aborts. All event variables in Table 73-2 containing a *dte_* or *dce_* prefix are valid in either emulate or monitor mode. These event variables are *dte_frame, dce_frame, dte_good_bcc, dce_good_bcc, dte_bad_bcc, dce_bad_bcc, dte_abort, dce_abort*. The variable *dce_good_bcc*, for example, equates to DCE GDBCC.

   You can use both *dte* and *dce* variables relating to the same event in one conditions block. Suppose you want to count all bad BCC's from either side of the line. Enter the following CONDITIONS/ACTIONS block:

   ```
   CONDITIONS:
   {
     dte_bad_bcc || dce_bad_bcc
   }
   ACTIONS: COUNTER bad_bcc INC
   ```

Table 73-1
SDLC Structures

Type	Variable	Value (hex/decimal)	Meaning
Structure Name: send_frame_structure			Structure of a frame in SDLC. Declared as type struct. Declared automatically if a softkey-entered SEND action is taken. Program frames assigned to structure as follows: struct send_frame_structure *name*. Reference a structure variable as follows: *name*.bcc_type. If values in the frame structure are not initialized by the user, they default to 0. You may initialize the values when the structure is declared: struct send_frame_structure *name* = {2, 1, 1, 0, 1, 1, 3, 0x71, 3, 0};
unsigned char	addr_type	2	
unsigned char	frame_type	*(The codes for frame_type are the same as for the SDLC-variable rcvd_frame_type.)*	
unsigned char	nr_type	0 1 2 3	auto value received ns plus 1 last nr sent
unsigned char	ns_type	0 1 2 3	auto skip last nr received value
unsigned char	p_f_type	0 1 2	0 1 loopback
unsigned char	bcc_type	0 1 2 3	default (bad bcc) good bcc bad bcc abort
unsigned char	addr_value	*00-ff/0-255*	
unsigned char	cntrl_byte	*(actual value of the control byte)*	
unsigned char	nr_value	*0-7 (MOD 8)*	if nr_type = 1
unsigned char	ns_value	*0-7 (MOD 8)*	if ns_type = 3

Table 73-2
SDLC Variables

Type	Variable	Value (hex/decimal)	Meaning
extern event	dte_frame		True when a DTE frame is detected. Line Setup configured for emulate or monitor mode.
extern event	dce_frame		True when a DCE frame is detected. Line Setup configured for emulate or monitor mode.
extern event	dte_good_bcc		True when a good BCC is calculated for a DTE frame. Line Setup configured for emulate or monitor mode.
extern event	dce_good_bcc		True when a good BCC is calculated for a DCE frame. Line Setup configured for emulate or monitor mode.
extern event	dte_bad_bcc		True when a bad BCC is calculated for a DTE frame. Line Setup configured for emulate or monitor mode.
extern event	dce_bad_bcc		True when a bad BCC is calculated for a DCE frame. Line Setup configured for emulate or monitor mode.
extern event	dte_abort		True when an abort is detected for a DTE frame. Line Setup configured for emulate or monitor mode.
extern event	dce_abort		True when an abort is detected for a DCE frame. Line Setup configured for emulate or monitor mode.
extern event	rcvd_frame		True when a frame is received. Line Setup configured for emulate mode only.
extern event	invalid_frame		True when an invalid frame is detected. Line Setup configured for emulate mode only.
extern event	l2_T1		True when the T1 timeout-timer has expired. Line Setup configured for emulate mode only.
extern event	bcc_error		True when a BCC error is detected. Line Setup configured for emulate mode only.
extern event	nr_error		True when an N(R) error is detected in a received INFO or supervisory frame. Line Setup configured for emulate mode only.
extern event	ns_error		True when an N(S) error is detected in a received INFO frame. Line Setup configured for emulate mode only.

Table 73-2 (continued)

Type	Variable	Value (hex/decimal)	Meaning
extern event	frame_sent		True when frame is passed down to Layer 1. Line Setup configured for emulate mode only.
extern volatile const unsigned char	m_frame_addr	*00-ff/0-255*	Line Setup configured for emulate or monitor mode.
extern volatile const unsigned char	m_frame_type	*(same as rcvd_frame_type—Line Setup configured for emulate or monitor mode)*	
extern volatile const unsigned char	m_frame_cntrl_byte_1	*(actual value of control byte—Line Setup configured for emulate or monitor mode)*	
extern volatile const unsigned char	m_frame_pf	0 10/16	pf=0 pf=1 Line Setup configured for emulate or monitor mode.
extern volatile const unsigned char	m_frame_bcc_type	1 2 3	good bad abort Line Setup configured for emulate or monitor mode.
extern volatile const unsigned char	rcvd_frame_addr	*00-ff/0-255*	Line Setup configured for emulate mode only.
extern volatile const unsigned char	rcvd_frame_type	0 1 5 9 d/13 3 7 7 f/15 23/35 43/67 43/67 63/99 83/131 87/135 af/175 c7/199 cf/207 e3/227 ef/239 b/11 ff/240 ff/240	info rr rnr rej srej ui rim sim dm up disc rd ua snrm frmr xid cfgr snrme test bcn lpda other unknown Line Setup configured for emulate mode only.
extern volatile const unsigned char	rcvd_frame_cntrl_byte_1	*(actual value of control byte—Line Setup configured for emulate mode only)*	
extern volatile const unsigned char	rcvd_frame_pf	0 10/16	pf=0 pf=1 Line Setup configured for emulate mode only.

Table 73-2 (continued)

Type	Variable	Value (hex/decimal)	Meaning
extern volatile const unsigned char	rcvd_frame_bcc_type	1 2 3	good bad abort Line Setup configured for emulate mode only.
extern volatile const unsigned char	rcvd_frame_nr	*0–7 (MOD 8)*	Line Setup configured for emulate mode only.
extern volatile const unsigned char	rcvd_frame_ns	*0–7 (MOD 8)*	Line Setup configured for emulate mode only.
extern volatile unsigned short	rcvd_frame_buff_seg		Inter-layer message buffer number (actually, an IAPX-286 segment number) in a received frame. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate mode only.
extern volatile unsigned short	rcvd_frame_sdu_offset		Offset to where the service data unit begins in an inter-layer message buffer in a received frame. Add to buffer segment number (converted to pointer) to point to first byte in frame. Line Setup configured for emulate mode only.
extern volatile unsigned short	rcvd_frame_sdu_size		Size of service data unit in a received frame. Line Setup configured for emulate mode only.
extern volatile unsigned short	l2_current_window_edge		When equal to upper edge, window is full; when equal to lower edge, window is empty; when not equal to upper edge, window is not full; and when not equal to lower edge, window is not empty. Line Setup configured for emulate mode only.
extern volatile unsigned short	l2_lower_window_edge		see l2_current_window_edge
extern volatile unsigned short	l2_upper_window_edge		see l2_current_window_edge
extern volatile unsigned short	l2_resend_edge		When resend edge is not equal to lower window edge, there is more to resend; when resend edge is equal to lower window edge, there is no more to resend. Line Setup configured for emulate mode only.

Table 73-2 (continued)

Type	Variable	Value (hex/decimal)	Meaning
extern unsigned char	l2_enhance	0	normal
		1	reverse
		4	low
		5	reverse low
		8	blink
		9	reverse blink
		12/18	blink low
			Line Setup configured for emulate or monitor mode.
extern unsigned char	l2_suppress	0	off
		1	on
			Line Setup configured for emulate or monitor mode.

Using spreadsheet tokens, the same test needs two CONDITIONS/ACTIONS blocks:

```
CONDITIONS: DTE BDBCC
ACTIONS: COUNTER bad_bcc INC
CONDITIONS: DCE BDBCC
ACTIONS: COUNTER bad_bcc INC
```

When the user selects DTE or DCE on the first rack of softkeys for Layer 2 conditions, a second rack appears from which he must select a particular frame type. A DTE INFO condition, for example, when translated, includes two C variables, one event variable and one status variable:

```
{
dte_frame && (m_frame_type == 0)
}
```

As a C programmer, you do not have to specify a frame type. To include all frames in a condition, use the event variable only:

```
CONDITIONS:
{
dte_frame
}
```

2. *Emulate mode only.* Some events may be detected in emulate mode only. The event variables are *rcvd_frame, invalid_frame, l2_T1, bcc_error, nr_error, ns_error,* and *frame_sent.*

   If you try to use one of these variables in monitor mode, you may be returned to the main program menu. When you go to the Protocol Spreadsheet and search for errors, a message like the following may be displayed: *"Error 140: Unresolved reference rcvd_frame."*

   When the user selects RCV on the first rack of softkeys for Layer 2 conditions, a second rack appears from which he must select a particular frame type. When the translator converts a RCV INFO condition into C, it will include two C variables, one event variable and one status variable:

```
{
rcvd_frame && (rcvd_frame_type == 0)
}
```

In a C condition, a frame type does not have to be specified. To include all received frames in a condition, use the event variable only:

```
CONDITIONS:
{
rcvd_frame
}
```

Error detecting may be accomplished via *bcc_error*, *nr_error*, *ns_error*, and *invalid_frame*. These variables equate to the softkey tokens bearing similar names.

One of the emulate-mode variables monitors an emulate action. The event variable *frame_sent* will come true as soon as the frame has been passed to the layer below.

(B) Status Variables

Status variables are those in Table 73-2 that do not include *event* in the Type column. Their associated event variables guarantee that they are updated and tested.

The softkey-generated condition for received Info frames is RCV INFO. The C version of the same condition should look like this:

```
CONDITIONS:
{
rcvd_frame && (rcvd_frame_type == 0)
}
```

1. *Frame characteristics*. All status variables in Table 73-2 containing an *m_* prefix are valid in either emulate or monitor mode: *m_frame_addr*, *m_frame_type*, *m_frame_cntrl_byte_1*, *m_frame_pf*, and *m_frame_bcc_type*. Use these variables to monitor a particular address, frame type, control byte, P/F value, or BCC.

   All status variables in Table 73-2 containing a *rcvd_* prefix are valid in emulate mode only: *rcvd_frame_addr*, *rcvd_frame_type*, *rcvd_frame_cntrl_byte_1*, *rcvd_frame_bcc_type*, *rcvd_frame_pf*, *rcvd_frame_nr*, and *rcvd_frame_ns*. Use these variables to monitor a particular address, frame type, control byte, BCC, or P/F, N(R), or N(S) value.

If you try to use an emulate-mode variable in monitor mode, you may be returned to the main program menu. When you go to the Protocol Spreadsheet and search for errors, a message like the following may be displayed: "*Error 140: Unresolved reference rcvd_frame_type.*"

2. *Frame buffers*. As BOP frames are received, they are automatically placed in IL message buffers to be passed up the layers. Three emulate-mode variables provide the user with access to the information in the frame that is located beyond the control byte. These variables are *rcvd_frame_buff_seg*, *rcvd_frame_sdu_offset*, and *rcvd_frame_sdu_size*. See Section 63.1 for a more detailed discussion of the buffer components to which these variables refer.

   Make a pointer to an IL buffer by casting *rcvd_frame_buff_seg* as a *long*, shifting it left sixteen bits, adding *rcvd_frame_sdu_offset*, and casting the result to a pointer. Increment the pointer twice (thereby adding two to the offset).

```
{
unsigned char * ptr;
ptr = (void *)(((long)rcvd_frame_buff_seg << 16) + rcvd_frame_sdu_offset);
ptr+=2;
}
```

   It is now pointing at the first byte in the information field. You may continue to move through the frame for its entire length, indicated in *rcvd_frame_sdu_size*.

3. *Transmit window*. Four related variables test the status of the Layer 2 window. The particular values of these variables at any given time is not significant. What is significant is how they compare to each other. The softkey status condition on the left makes the variable comparison on the right:

WINDOW FULL	*l2_current_window_edge == l2_upper_window_edge*
WINDOW EMPTY	*l2_current_window_edge == l2_lower_window_edge*
WINDOW NOT_FULL	*l2_current_window_edge != l2_upper_window_edge*
WINDOW NOT_EMPTY	*l2_current_window_edge != l2_lower_window_edge*
MORE_TO_RESEND	*l2_resend_edge != l2_lower_window_edge*
NO_MORE_TO_RESEND	*l2_resend_edge == l2_lower_window_edge*

(C) Controlling Protocol Trace Display

To enhance or suppress particular frames on the Layer 2 Protocol Trace screen in emulate or monitor mode, assign a coded value to *l2_enhance* or *l2_suppress*. The values are listed in Table 73-2. To assign a value to either of these variables, place the statement in an ACTIONS block. For example, display RNR frames in reverse-video and suppress display of invalid frames:

```
CONDITIONS: RCV RNR
ACTIONS:
{
 l2_enhance = 1;
}
CONDITIONS: RCV INVALID
ACTIONS:
{
 l2_suppress = 1;
}
```

Check the value of these display-control variables in a CONDITIONS block

```
CONDITIONS: RCV INFO
{
 l2_enhance == 1
}
ACTIONS:
{
 l2_enhance = 0;
}
```

or an ACTIONS block:

```
CONDITIONS: RCV INFO
ACTIONS:
{
 if(l2_enhance == 1)
   l2_enhance = 0;
}
```

73.3 Routines

Use the following routines in emulate mode only. If you try to call one of these routines in monitor mode, you will be returned to the main program menu. When you go to the Protocol Spreadsheet and search for errors, a message like the following will be displayed: "*Error 140: Unresolved reference l2_give_data.*"

(A) Receive

l2_give_data

Synopsis

```
extern void l2_give_data();
```

Description

The *l2_give_data* routine takes takes an interlayer message buffer associated with a received INFO frame, changes the SDU offset to point to higher-level data, and sends a DL_DATA IND primitive up to Layer 3 along with a reference to this buffer. The softkey equivalent of this routine is the GIVE_DATA action on the Protocol Spreadsheet.

Example

Layer 3 wants access to the line in order to receive and send data. Assuming the SDLC personality package is loaded at Layer 2, enter the following program:

```
LAYER: 2
     STATE: datalink
        CONDITIONS: DL_CONNECT REQ
        ACTIONS: DL_CONNECT CONF
        CONDITIONS: DL_DATA REQ
        ACTIONS: SEND INFO "((DL_DATA))"
        CONDITIONS: RCV INFO
        ACTIONS:
        {
         l2_give_data();
        }
```

(B) Transmit

resend_frame

Synopsis

extern void resend_frame(pf, first_or_next);
unsigned char pf;
unsigned char first_or_next;

Description

The *resend_frame* routine will resend either the first or next frame in the window with the P/F bit set to a specified value. The softkey equivalent of this routine is the (PROTOCL) RESEND action on the Protocol Spreadsheet.

Inputs

The first parameter is the value of the P/F bit in the frame. It may be set to either 0, 1, or 2 (for loopback).

The second parameter indicates whether the first frame in the window will be sent, or whether the next frame in the window will be sent. The first resend action will send the first frame in the window regardless of whether first or next has been selected. Legal entries are 0 (first) or 1 (next).

Example

Suppose you want to resend the entire transmit window if you receive a REJ frame.

```
LAYER: 2
    STATE: xfer
        /* Whatever conditions and actions send data precede the following condition. */
        CONDITIONS: RCV REJ RESP
        ACTIONS:
        {
         resend_frame(1, 0);
        }
        NEXT_STATE: recover
    STATE: recover
        CONDITIONS: FRAME_SENT
            MORE_TO_RESEND
        ACTIONS:
        {
         resend_frame(1,1);
        }
        CONDITIONS: FRAME_SENT
            NO_MORE_TO_RESEND
        NEXT_STATE: xfer
```

reset_nr

Synopsis

extern void reset_nr();

Description

This routine resets the N(R) field in information and supervisory frames to zero. The softkey equivalent of this routine is the (PROTOCL) RSET_NR action on the Protocol Spreadsheet.

Example

When a link is established, reset N(R).

```
LAYER: 2
    STATE: reset
        CONDITIONS: ENTER_STATE
        ACTIONS: SEND SABM
        CONDITIONS: RCV UA
        ACTIONS:
        {
         reset_nr();
        }
```

reset_ns

Synopsis

```
extern void reset_ns();
```

Description

The N(S) field in information and supervisory frames is reset to zero and the transmit window is cleared. The softkey equivalent of this routine is the (PROTOCL) RSET_NS action on the Protocol Spreadsheet.

Example

When a link is established, reset N(S).

```
LAYER: 2
    STATE: reset
      CONDITIONS: ENTER_STATE
      ACTIONS: SEND SABM
      CONDITIONS: RCV UA
      ACTIONS:
      {
       reset_ns();
      }
```

send_frame

Synopsis

```
extern void send_frame(il_buffer_number, relay_baton, data_start_offset, transmit_frame_ptr);
unsigned short il_buffer_number;
unsigned short relay_baton;
unsigned short data_start_offset;
struct send_frame_structure
{
 unsigned char addr_type;
 unsigned char frame_type;
 unsigned char nr_type;
 unsigned char ns_type;
 unsigned char p_f_type;
 unsigned char bcc_type;
 unsigned char addr_value;
 unsigned char cntrl_byte;
 unsigned char nr_value;
 unsigned char ns_value;
};
struct send_frame_structure * transmit_frame_ptr;
```

Description

The *send_frame* routine adds a frame-level header to an interlayer message buffer and passes the buffer to Layer 1. The softkey equivalent of this routine is the SEND action on the Protocol Spreadsheet.

Inputs

The first parameter is the interlayer message buffer number. See Section 63.3(A), Layer–Independent OSI routines.

The second parameter is the maintain bit used to hold the buffer while the send operation is being performed. See Section 63.3(A).

The third parameter is the offset from the beginning of the buffer to the start of the service data unit. See Section 63.3(A).

The fourth parameter is a pointer to the frame structure to be sent. For a description of *send_frame_structure* see Table 73-1.

Example

Send an Info frame containing a canned fox message and a good BCC onto the line.

```
{
 static unsigned short il_buffer_number;
 static unsigned short relay_baton;
 static unsigned short data_start_offset;
 struct send_frame_structure
  {
   unsigned char addr_type;
   unsigned char frame_type;
   unsigned char nr_type;
   unsigned char ns_type;
   unsigned char p_f_type;
   unsigned char bcc_type;
   unsigned char addr_value;
   unsigned char cntrl_byte;
   unsigned char nr_value;
   unsigned char ns_value;
  };
 struct send_frame_structure transmit_frame;
 static char transmit_string [] = "《FOX》";
}
LAYER: 2
     STATE: send_a_frame
        CONDITIONS: KEYBOARD " "
        ACTIONS:
        {
         _get_il_msg_buff(&il_buffer_number, &relay_baton);
         _start_il_buff_list(il_buffer_number,&data_start_offset);
         transmit_frame.bcc_type = 1;
         _insert_il_buff_list_cnt(il_buffer_number, data_start_offset, &transmit_string[0],
               (sizeof(transmit_string) - 1));
         send_frame(il_buffer_number, relay_baton, data_start_offset, &transmit_frame);
        }
```

74 SNA Library

When the SNA package is loaded in via the Layer Setup screen, the following external variables become available for use by the programmer. Their use on the Protocol Spreadsheet is not limited to any particular layer, though normally they belong at Layer 2.

SDLC variables and routines, while they are included in the SNA layer-personality package, are not documented here. They are documented fully in Section 73.

Those variables that are specific to the SNA package are documented here. They pertain to fields in SNA transmission headers, request/response headers, and request/response units. These variables have no spreadsheet-token equivalents.

74.1 Structures

Use the SDLC *send_frame_structure* shown in Table 73-1.

74.2 Variables

The variables discussed below apply when the Line Setup menu shows either emulate or monitor mode. Emulate mode, however, is not supported by emulate-only conditions and actions on the Protocol Spreadsheet.

(A) Monitoring Events

Use the SDLC event variables discussed in Section 73.2(A).

(B) Status Variables

All SNA variables in Table 74-1 are status variables. Also refer to the SDLC status variables listed in Table 73-2.

There are no softkey tokens on the spreadsheet that are equivalent to the SNA variables listed in Table 74-1. To search for Info frames with a FID2 transmission header, for example, use C variables. The condition should look like this:

```
CONDITIONS:
{
dte_frame && (m_frame_type == 0) && (m_packet_fid_type == 2)
}
```

Table 74-1
SNA Variables†

Type	Variable	Value (hex/decimal)	Meaning
extern volatile unsigned short	m_packet_length		Length of the packet, including the transmission and request/response headers. Line Setup configured for emulate or monitor mode.
	<u>Transmission Header:</u>		
extern volatile const unsigned char	m_packet_fid_type		<u>Format Identification Type:</u>
		0	FID0; TH 10 bytes
		1	FID1; TH 10 bytes
		2	FID2; TH 6 bytes
		3	FID3; TH 2 bytes
		4	FID4; TH 26 bytes
		f/15	FIDF; TH 26 bytes
			Line Setup configured for emulate or monitor mode.
extern volatile const unsigned short	m_packet_daf	*0-ffff/0-65535*	Destination address field—2 bytes in FID0 and FID1; 1 byte in FID2. Line Setup configured for emulate or monitor mode.
extern volatile const unsigned short	m_packet_def	*0-ffff/0-65535*	Destination element field—2 bytes; FID4 only. Line Setup configured for emulate or monitor mode.
extern volatile const unsigned long	m_packet_dsaf	*0-ffffffff/ 0-4294967295*	Destination subarea address field—4 bytes; FID4 only. Line Setup configured for emulate or monitor mode.
extern volatile const unsigned char	m_packet_lsid		<u>Local Session Identification: FID3 only</u>
		(actual value of byte)	SSCP-PU session
			SSCP-LU session
			Reserved
			LU-LU session
			Line Setup configured for emulate or monitor mode.
extern volatile const unsigned short	m_packet_oaf	*00-ff/0-255*	Origin address field—2 bytes in FID0 and FID1; 1 byte in FID2. Line Setup configured for emulate or monitor mode.
extern volatile const unsigned short	m_packet_oef	*00-ff/0-255*	Origin element field—2 bytes; FID4 only. Line Setup configured for emulate or monitor mode.
extern volatile const unsigned long	m_packet_osaf	*0-ffffffff/ 0-4294967295*	Origin subarea address field—4 bytes; FID4 only. Line Setup configured for emulate or monitor mode.

† Refer to Table 73-2 for SDLC variables.

Table 74-1 (continued)

Type	Variable	Value (hex/decimal)	Meaning
	Transmission Header (contiued):		
extern volatile unsigned char *	th_ptr		Pointer for the transmission header; begins at the byte containing FID type. Line Setup configured for emulate or monitor mode.
	Request/Response Header:		
extern volatile const unsigned char	m_packet_ru_category		Request/Response Unit:
		0	Function Management Data (FMD)
		20/32	Network Control (NC)
		40/64	Data Flow Control (DFC)
		60/96	Session Control (SC)
			Line Setup configured for emulate or monitor mode.
extern volatile const unsigned char	m_packet_fi		Format Indicator:
		0	User data without header in RU
		8	In LU-LU frame, indicates header follows the RH. In SC, NC, or DFC RU, indicates a formatted RU beginning with a request code.
			Line Setup configured for emulate or monitor mode.
extern volatile const unsigned char	m_packet_rri		Request/Response Indicator:
		0	request
		80/128	response
			Line Setup configured for emulate or monitor mode.
extern volatile unsigned char	m_packet_rti		Response Type Indicator:
		0	positive response
		10/16	negative response
			Line Setup configured for emulate or monitor mode.
extern volatile unsigned char	m_packet_sdi		Sense Data Indicator:
		0	sense data not included
		4	sense data included
			Line Setup configured for emulate or monitor mode.
extern volatile unsigned char *	rh_ptr		Pointer for the request/response header; begins at the byte containing the request/response indicator. Line Setup configured for emulate or monitor mode.
	Request/Response Unit:		
extern volatile unsigned char *	ru_ptr		Pointer for the request/response unit; begins at the first byte in the unit. Line Setup configured for emulate or monitor mode.

1. *Info frame characteristics.* Most status variables in Table 74-1 contain an *m_* prefix, indicating that they are valid in emulate or monitor mode. Some variables are associated with the transmission header: *m_packet_fid_type, m_packet_daf, m_packet_def, m_packet_dsaf, m_packet_lsid, m_packet_oaf, m_packet_oef,* and *m_packet_osaf.* Other variables are associated with the request/response header: *m_packet_ru_category, m_packet_fi, m_packet_rri, m_packet_rti,* and *m_packet_sdi.*

2. *Pointers.* There are three pointers to SNA fields. *th_ptr* points to first byte of the transmission header, *rh_ptr* points to the first byte of request/response header, and *ru_ptr* points to the start of the request/response unit.

(C) Controlling Protocol Trace Display

To enhance or suppress particular packets on the Layer 2 Protocol Trace screen in emulate or monitor mode, assign a coded value to *l2_enhance* or *l2_suppress.* The values are listed in Table 73-2. To assign a value to either of these variables, place the statement in an ACTIONS block. For example, display only Info frames with FID2 transmission headers. Of these, display frames with sense data in reverse-video.

```
CONDITIONS:
{
dte_frame && (m_frame_type == 0) && (m_packet_fid_type != 2)
}
ACTIONS:
{
l2_suppress = 1;
}
CONDITIONS:
{
dte_frame && (m_frame_type == 0) && (m_packet_fid_type == 2) && (m_packet_sdi == 4)
}
ACTIONS:
{
l2_enhance = 1;
}
```

Check the value of these display-control variables in a CONDITIONS block

```
CONDITIONS:
{
dte_frame && (m_frame_type == 0) && (m_packet_fid_type != 2) && (l2_suppress == 0)
}
ACTIONS:
{
l2_suppress = 1;
}
```

or an ACTIONS block:

```
CONDITIONS:
{
 dte_frame && (m_frame_type == 0) && (m_packet_fid_type != 2)
}
ACTIONS:
{
 if(l2_suppress == 0)
   l2_suppress = 1;
}
```

74.3 Routines

There are no routines associated exclusively with SNA. Use the SDLC routines discussed in Section 73.3. To send a frame including SNA protocol, for example, include a *transmit_string* of SNA data in the *send_frame* routine.

75 DDCMP Library

When the DDCMP package is loaded in via the Layer Setup screen, the following external variables become available for use by the programmer. Their use on the Protocol Spreadsheet is not limited to any particular layer, though normally they belong at Layer 1.

75.1 Structures

There are no *extern* structures associated exclusively with DDCMP.

75.2 Variables

The only variables exclusive to DDCMP relate to block checking. When the DDCMP package is loaded in, the results of both header and data block checks are displayed on the data screen. If you want your program to detect good or bad BCC's, you may use the BCC selections on the trigger menus and at Layer 1 of the Protocol Spreadsheet to interrogate the header block check only.

If you want to detect a good or bad data block check, you must use one of the C event variables listed in Table 75-1.

Here is a program that counts bad DTE BCC's for both header and data:

```
{
  extern fast_event fevar_bd_bcc2_td;
}
LAYER: 1
  STATE: count_all_bad_dte_bccs
    CONDITIONS: DTE BAD_BCC
    ACTIONS: COUNTER t_bdbcc INC
    CONDITIONS:
    {
      fevar_bd_bcc2_td
    }
    ACTIONS: COUNTER t_bdbcc INC
```

75.3 Routines

There are no routines associated exclusively with DDCMP.

Table 75-1
DDCMP Variables

Type	Variable	Value (hex/decimal)	Meaning
extern fast_event	fevar_gd_bcc_rd		True when a good *header* BCC is received on RD. Line Setup configured for emulate or monitor mode.
extern fast_event	fevar_gd_bcc_td		True when a good *header* BCC is received on TD. Line Setup configured for emulate or monitor mode.
extern fast_event	fevar_bd_bcc_rd		True when a bad *header* BCC is received on RD. Line Setup configured for emulate or monitor mode.
extern fast_event	fevar_bd_bcc_td		True when a bad *header* BCC is received on TD. Line Setup configured for emulate or monitor mode.
extern fast_event	fevar_gd_bcc2_td		True when a good *data* BCC is received on TD. Line Setup configured for emulate or monitor mode.
extern fast_event	fevar_gd_bcc2_rd		True when a good *data* BCC is received on RD. Line Setup configured for emulate or monitor mode.
extern fast_event	fevar_bd_bcc2_td		True when a bad *data* BCC is received on TD. Line Setup configured for emulate or monitor mode.
extern fast_event	fevar_bd_bcc2_rd		True when a bad *data* BCC is received on RD. Line Setup configured for emulate or monitor mode.

76 ISDN D Channel Library

To use the C structures, variables, and routines explained in this section, your INTERVIEW must be equipped with Option 15. Install the ISDN Test Interface Module (TIM) in the rear of the INTERVIEW, as explained in Section 10. Also install the ISDN mux board according to the directions in Appendix J4. Load in the ISDN_D Layer 1 package via the Layer Setup screen. The ISDN_D package contains the variables and most of the routines documented below. Finally, select one of the B channels in the **Channel** field on the ISDN Interface Setup screen. See Section 48.5.

The configuration of the INTERVIEW described above supports dual-channel monitoring. Dual-channel monitoring means tracking one of the B channels and the D channel. All menu selections (with the possible exception of **Speaker** on the ISDN Interface Setup menu), triggers, and spreadsheet conditions and actions apply to the B channel selected. Use the C structures, variables, and routines in this section to monitor the D channel.

> NOTE: When the ISDN Interface Setup screen shows **Channel:** D, your unit is configured for single-channel monitoring. Menu selections, triggers, and the Protocol Spreadsheet apply to the D channel. Do not load in the ISDN_D Layer 1 package.

You may develop your own program to monitor the D channel, or simply load and run the program contained in the ISDN trace application package (available as OPT-951-35).

76.1 Structures

Use the structure *xmit_list*, shown in Table 76-1, when transmitting on the D channel via the *send_d_frame* routine. Refer to *send_d_frame* in Section 76.3 for an example of how to use this structure.

Table 76-1
ISDN Structures

Type	Variable	Value (hex/decimal)	Meaning
Structure Name:	**xmit_list**		Structure of a transmit list for *send_d_frame* routine. Declared as type *struct*. Reference member variables of the structure as follows: *xmit_list.string_length*.
unsigned char *	string		pointer to the location of the transmit string—the transmit string is declared separately
unsigned short	string_length	*0-ffff/0-65535*	length of the transmit string

76.2 Variables

There are three event variables associated with the ISDN_D personality package. They are *d_dte_frame*, *d_dce_frame*, and *d_rcv_frame*. See Table 76-2.

(A) Monitoring Events

1. *In monitor mode*. When a frame is detected on the D channel, one monitor event, *d_dce_frame* or *d_dte_frame*, is signaled. Use both event variables to construct an ISDN trace.
2. *In emulate mode*. In emulate mode, the receive event *d_rcv_frame* and one of the monitor events are signaled when a frame is received on the D channel. The INTERVIEW's transmissions on the D channel may not be monitored when the unit is in dual-channel mode. The implication of this difference is that ISDN trace programs written in monitor mode may not be placed intact in an emulation program.

Table 76-2
ISDN Variables

Type	Variable	Value (hex/decimal)	Meaning
extern event	d_dte_frame		True when a DTE frame is detected on the D channel. Line Setup configured for emulate or monitor mode.
extern event	d_dce_frame		True when a DCE frame is detected on the D channel. Line Setup configured for emulate or monitor mode.
extern event	d_rcv_frame		True when a frame is received on the D channel. Line Setup configured for emulate mode only.

76.3 Routines

There are two routines associated with the ISDN_D package: *send_d_frame* and *send_d_frame_il*. Another ISDN routine, *set_isdn_speaker_chan*, controls the speaker for either of the B channels. This routine is supplied by the ISDN Test Interface Module.

(A) Transmit

Use the following routines in emulate mode only. If you try to call one of these routines in monitor mode, you may be returned to the main program menu. When you go to the Protocol Spreadsheet and search for errors, a message like the following may be displayed: *"Error 140: Unresolved reference send_d_frame_il."*

send_d_frame

Synopsis

```
extern void send_d_frame(count, struct_send_string_ptr, xmit_tag);
unsigned short count;
struct xmit_list
  {
   unsigned char * string_ptr;
   unsigned short string_length;
  };
struct xmit_list * struct_send_string_ptr;
unsigned short xmit_tag;
```

Description

The *send_d_frame* routine sends a specified string on the D channel with a user–determined BCC.

Inputs

The first parameter is the number of strings to be sent.

The second parameter is a pointer to a structure which in turn identifies the location and length of each string.

The third parameter is a transmit tag which includes a BCC in bits 0–2: good (001), bad (010), or abort (011). Bits 3–7 are reserved for future use. Integers may be used to indicate the value of the transmit tag: good (1), bad (2), and abort (3).

Example

Assume you want to send on channel D a fox message inside of an X.25 data packet with a good block check. You might have 2 strings, one with the Layers 2 and 3 header information, and one with the fox message. You would send these strings as follows:

```
{
 unsigned char headers [] = {0x01, 0x00, 0x10, 0x04, 0x00};
 unsigned char message [] = "((FOX))";
 struct xmit_list
  {
   unsigned char * string;
   unsigned short string_length;
  };
 struct xmit_list send_string [] = {&headers[0], 5, &message[0], sizeof(message) - 1};
}
```

```
LAYER: 1
    STATE: send_message
       CONDITIONS: KEYBOARD " "
       ACTIONS:
       {
        send_d_frame(2, &send_string[0], 1);
       }
```

send_d_frame_il

Synopsis

```
extern void send_d_frame_il(il_buffer_number, relay_baton, data_start_offset, transmit_tag);
unsigned short il_buffer_number;
unsigned short relay_baton;
unsigned short data_start_offset;
unsigned short transmit_tag;
```

Description

This routine sends a designated interlayer message buffer out on the D channel.

Inputs

The first parameter is the interlayer message buffer number.

The second parameter is the maintain bit used to hold the buffer while the send operation is performed at Layer 1.

The third parameter is the offset from the beginning of the buffer to the service data unit (SDU).

The fourth parameter is a transmit tag which includes a BCC in bits 0-2: good (001), bad (010), or abort (011). Bits 3-7 are reserved for future use. Integers may be used to indicate the value of the transmit tag: good (1), bad (2), and abort (3).

Example

Send the same text as in the example for *send_d_frame*. Refer to Section 63.3(A) for a description of the *_get_il_msg_buff*, *_start_il_buff_list*, and *_insert_il_buff_list_cnt* routines.

```
{
 unsigned short il_buffer_number;
 unsigned short relay_baton;
 unsigned short data_start_offset;
 unsigned char message [] = "01\x00010 04\x000((FOX))";
}
```

```
LAYER: 1
     STATE: send_message
       CONDITIONS: KEYBOARD " "
       ACTIONS:
       {
        _get_il_msg_buff(&il_buffer_number, &relay_baton);
        _start_il_buff_list(il_buffer_number, &data_start_offset);
        _insert_il_buff_list_cnt(il_buffer_number, data_start_offset, &message[0],
             (sizeof(message) - 1));
        send_d_frame_il(il_buffer_number, relay_baton, data_start_offset, 1);
       }
```

(B) Speaker Control

set_isdn_speaker_chan

Synopsis

extern void set_isdn_speaker_chan(selection);
unsigned short selection;

Description

The *set_isdn_speaker_chan* routine allows the programmer to control the speaker located on the ISDN mux board, Option 15. The programmer may enable the speaker for one of the B channels. This selection is independent of the channel selected for monitor or emulation on the ISDN Interface Setup screen.

Inputs

The only parameter is the channel selection. A value of one means turn the speaker on for channel B1. Enable the speaker for channel B2 with two. Turn the speaker off by setting the value to zero.

Example

Suppose you want to know whether data or voice is being transmitted over channel B1. Use the *set_isdn_speaker_chan* routine to enable the speaker for B1. Even if you are otherwise using the INTERVIEW to monitor B2, you will hear the B1 transmissions.

```
LAYER: 1
     STATE: enable_b1
       CONDITIONS: KEYBOARD "sS"
       ACTIONS:
       {
        set_isdn_speaker_chan(1);
       }
```

77 LAPD Library

When the LAPD package is loaded in via the Layer Setup screen, the following external routines and variables become available for use by the programmer. Their use on the Protocol Spreadsheet is not limited to any particular layer, though normally they belong at Layer 2.

The variables and routines approximate LAPD Layer 2 spreadsheet-generated conditions and actions. Refer to Section 39 for more detailed explanations of the purposes of specific conditions and actions. Sometimes the name of the variable or routine is sufficient for identifying its related spreadsheet token. When this is not the case, the information is provided below.

77.1 Structures

The structure *send_frame_structure* defines the format of transmitted LAPD frames. See Table 77-1. Use this structure to send frames via the *send_frame* routine in emulate mode. See Section 77.3(B). Each variable in the structure relates to some softkey selection or user entry in the SEND action.

77.2 Variables

(A) Monitoring Events

1. *Emulate or monitor mode*. LAPD events include frames detected, good or bad BCC's, and aborts. All event variables in Table 77-2 containing a *dte_* or *dce_* prefix are valid in either emulate or monitor mode. These event variables are *dte_frame, dce_frame, dte_good_bcc, dce_good_bcc, dte_bad_bcc, dce_bad_bcc, dte_abort, dce_abort*. The variable *dce_good_bcc*, for example, equates to DCE GDBCC.

   You can use both *dte* and *dce* variables relating to the same event in one conditions block. Suppose you want to count all bad BCC's from either side of the line. Enter the following CONDITIONS/ACTIONS block:

```
CONDITIONS:
{
  dte_bad_bcc || dce_bad_bcc
}
ACTIONS: COUNTER bad_bcc INC
```

Table 77-1
LAPD Structures

Type	Variable	Value (hex/decimal)	Meaning
Structure Name: send_frame_structure			Structure of a frame in LAPD. Declared as type struct. Declared automatically if a softkey-entered SEND action is taken. Program frames assigned to structure as follows: struct send_frame_structure *name*. Reference a structure variable as follows: *name*.bcc_type. If values in the frame structure are not initialized by the user, they default to 0. You may initialize the values when the structure is declared: struct send_frame_structure *name* = {1, 1, 2, 0, 0, 0, 1, 1, 1, 0, 0, 0};
unsigned char	sapi_type	1	no other value valid—indicates a value is given
unsigned char	tei_type	1	no other value valid—indicates a value is given
unsigned char	cr_type	0 1 2	0 1 loopback
unsigned char	frame_type	*(The codes for frame_type are the same as for the LAPD-variable rcvd_frame_type.)*	
unsigned char	nr_type	0 1 2 3	auto value received ns plus 1 last nr sent
unsigned char	ns_type	0 1 2 3	auto skip last nr received value
unsigned char	p_f_type	0 1 2	0 1 loopback
unsigned char	bcc_type	0 1 2 3	default (bad bcc) good bcc bad bcc abort
unsigned char	sapi_value	*00-3f/0-63*	
unsigned char	tei_value	*00-7f/0-127*	
unsigned char	cntrl_byte	*(actual value of the control byte)*	
unsigned char	nr_value	*0-7 (MOD 8)*	if nr_type = 1
unsigned char	ns_value	*0-7 (MOD 8)*	if ns_type = 3

Table 77-2
LAPD Variables

Type	Variable	Value (hex/decimal)	Meaning
extern event	dte_frame		True when a DTE frame is detected. Line Setup configured for emulate or monitor mode.
extern event	dce_frame		True when a DCE frame is detected. Line Setup configured for emulate or monitor mode.
extern event	dte_good_bcc		True when a good BCC is calculated for a DTE frame. Line Setup configured for emulate or monitor mode.
extern event	dce_good_bcc		True when a good BCC is calculated for a DCE frame. Line Setup configured for emulate or monitor mode.
extern event	dte_bad_bcc		True when a bad BCC is calculated for a DTE frame. Line Setup configured for emulate or monitor mode.
extern event	dce_bad_bcc		True when a bad BCC is calculated for a DCE frame. Line Setup configured for emulate or monitor mode.
extern event	dte_abort		True when an abort is detected for a DTE frame. Line Setup configured for emulate or monitor mode.
extern event	dce_abort		True when an abort is detected for a DCE frame. Line Setup configured for emulate or monitor mode.
extern event	rcvd_frame		True when a frame is received. Line Setup configured for emulate mode only.
extern event	invalid_frame		True when an invalid frame is detected. Line Setup configured for emulate mode only.
extern event	l2_T1		True when the T1 timeout-timer has expired. Line Setup configured for emulate mode only.
extern event	bcc_error		True when a BCC error is detected. Line Setup configured for emulate mode only.
extern event	nr_error		True when an N(R) error is detected in a received INFO or supervisory frame. Line Setup configured for emulate mode only.
extern event	ns_error		True when an N(S) error is detected in a received INFO frame. Line Setup configured for emulate mode only.

Table 77-2 (continued)

Type	Variable	Value (hex/decimal)	Meaning
extern event	frame_sent		True when frame is passed down to Layer 1. Line Setup configured for emulate mode only.
extern volatile const unsigned char	m_frame_addr_sapi	*00-3f/0-63*	Line Setup configured for emulate or monitor mode.
extern volatile const unsigned char	m_frame_addr_tei	*00-7f/0-127*	Line Setup configured for emulate or monitor mode.
extern volatile const unsigned char	m_frame_addr_cr	0 1	0 1 Line Setup configured for emulate or monitor mode.
extern volatile const unsigned char	m_frame_type	*(same as rcvd_frame_type—Line Setup configured for emulate or monitor mode)*	
extern volatile const unsigned char	m_frame_cntrl_byte_1	*(actual value of control byte—Line Setup configured for emulate or monitor mode)*	
extern volatile const unsigned char	m_frame_pf	0 10/16	pf=0 pf=1 Line Setup configured for emulate or monitor mode.
extern volatile const unsigned char	m_frame_bcc_type	1 2 3	good. bad abort Line Setup configured for emulate or monitor mode.
extern volatile const unsigned char	m_frame_nr	*0-7 (MOD 8)*	Line Setup configured for emulate or monitor mode.
extern volatile const unsigned char	m_frame_ns	*0-7 (MOD 8)*	Line Setup configured for emulate or monitor mode.
extern volatile const unsigned char	rcvd_frame_addr_sapi	*00-3f/0-63*	Line Setup configured for emulate mode only.
extern volatile const unsigned char	rcvd_frame_addr_tei	*00-7f/0-127*	Line Setup configured for emulate mode only.
extern volatile const unsigned char	rcvd_frame_addr_cr	0 1 2	0 1 loopback Line Setup configured for emulate mode only.
extern volatile const unsigned char	rcvd_frame_type	0 1 3 5 9 2f/37 6f/111 43/67 f/15 f/15 63/99 67/103 87/135 e7/224 ff/255 ff/255	info rr ui rnr rej sabm sabme disc dm sarm ua slo frmr sl1 other unknown Line Setup configured for emulate mode only.

Table 77-2 (continued)

Type	Variable	Value (hex/decimal)	Meaning
extern volatile const unsigned char	rcvd_frame_cntrl_byte_1	*(actual value of control byte—Line Setup configured for emulate mode only)*	
extern volatile const unsigned char	rcvd_frame_pf	0 10/16	pf=0 pf=1 Line Setup configured for emulate mode only.
extern volatile const unsigned char	rcvd_frame_bcc_type	1 2 3	good bad abort Line Setup configured for emulate mode only.
extern volatile const unsigned char	rcvd_frame_nr	*0-7 (MOD 8)*	Line Setup configured for emulate mode only.
extern volatile const unsigned char	rcvd_frame_ns	*0-7 (MOD 8)*	Line Setup configured for emulate mode only.
extern volatile unsigned short	rcvd_frame_buff_seg		Inter-layer message buffer number (actually, an IAPX-286 segment number) in a received frame. This segment number can be converted to a pointer by shifting it left 16 bits. Line Setup configured for emulate mode only.
extern volatile unsigned short	rcvd_frame_sdu_offset		Offset to where the service data unit begins in an inter-layer message buffer in a received frame. Add to buffer segment number (converted to pointer) to point to first byte in frame. Line Setup configured for emulate mode only.
extern volatile unsigned short	rcvd_frame_sdu_size		Size of service data unit in a received frame. Line Setup configured for emulate mode only.
extern volatile unsigned short	l2_current_window_edge		When equal to upper edge, window is full; when equal to lower edge, window is empty; when not equal to upper edge, window is not full; and when not equal to lower edge, window is not empty. Line Setup configured for emulate mode only.
extern volatile unsigned short	l2_lower_window_edge		see l2_current_window_edge
extern volatile unsigned short	l2_upper_window_edge		see l2_current_window_edge
extern volatile unsigned short	l2_resend_edge		When resend edge is not equal to lower window edge, there is more to resend; when resend edge is equal to lower window edge, there is no more to resend. Line Setup configured for emulate mode only.

Table 77-2 (continued)

Type	Variable	Value (hex/decimal)	Meaning
extern unsigned char	l2_enhance	0	normal
		1	reverse
		4	low
		5	reverse low
		8	blink
		9	reverse blink
		12/18	blink low
			Line Setup configured for emulate or monitor mode.
extern unsigned char	l2_suppress	0	off
		1	on
			Line Setup configured for emulate or monitor mode.

Using spreadsheet tokens, the same test needs two CONDITIONS/ACTIONS blocks:

```
CONDITIONS: DTE BDBCC
ACTIONS: COUNTER bad_bcc INC
CONDITIONS: DCE BDBCC
ACTIONS: COUNTER bad_bcc INC
```

When the user selects DTE or DCE on the first rack of softkeys for Layer 2 conditions, a second rack appears from which he must select a particular frame type. A DTE INFO condition, for example, when translated, includes two C variables, one event variable and one status variable:

```
{
 dte_frame && (m_frame_type == 0)
}
```

In C, the programmer does not need to specify a frame type. To include all frames in a condition, use the event variable only:

```
CONDITIONS:
{
 dte_frame
}
```

2. *Emulate mode only.* Some events may be detected in emulate mode only. The event variables are *rcvd_frame*, *invalid_frame*, *l2_T1*, *bcc_error*, *nr_error*, *ns_error*, and *frame_sent*.

   If you try to use one of these variables in monitor mode, you may be returned to the main program menu. When you go to the Protocol Spreadsheet and search for errors, a message like the following may be displayed: *"Error 140: Unresolved reference rcvd_frame."*

   When the user selects RCV on the first rack of softkeys for Layer 2 conditions, a second rack appears from which he must select a particular frame type. When the translator converts a RCV INFO condition into C, it will include two C variables, one event variable and one status variable:

```
{
rcvd_frame && (rcvd_frame_type == 0)
}
```

The C programmer does not have to specify a frame type. To include all received frames in a condition, use the event variable only:

```
CONDITIONS:
{
rcvd_frame
}
```

Error detecting may be accomplished via *bcc_error*, *nr_error*, *ns_error*, and *invalid_frame*. These variables equate to the softkey tokens bearing similar names.

One of the emulate-mode variables monitors an emulate action. The event variable *frame_sent* will not come true until the frame actually has been passed to the layer below.

(B) Status Variables

Status variables are those in Table 77-2 that do not include *event* in the Type column. Their associated event variables guarantee that they are updated and tested.

The softkey-generated condition for received Info frames is RCV INFO. The C version of the same condition should look like this:

```
CONDITIONS:
{
rcvd_frame && (rcvd_frame_type == 0)
}
```

1. *Frame characteristics.* All status variables in Table 77-2 containing an *m_* prefix are valid in either emulate or monitor mode: *m_frame_addr_sapi*, *m_frame_addr_tei*, *m_frame_addr_cr*, *m_frame_type*, *m_frame_cntrl_byte_1*, *m_frame_pf*, *m_frame_bcc_type*, *m_frame_nr*, and *m_frame_ns*.

   All status variables in Table 77-2 containing a *rcvd_* prefix are valid in emulate mode only: *rcvd_frame_addr_sapi*, *rcvd_frame_addr_tei*, *rcvd_frame_addr_cr*, *rcvd_frame_type*, *rcvd_frame_cntrl_byte_1*, *rcvd_frame_pf*, *rcvd_frame_bcc_type*, *rcvd_frame_nr*, and *rcvd_frame_ns*.

   If you try to use an emulate-mode variable in monitor mode, you may be returned to the main program menu. When you go to the Protocol Spreadsheet and search for errors, a message like the following may be displayed: "*Error 140: Unresolved reference rcvd_frame_type.*"

2. *Frame buffers*. As BOP frames are received, they are automatically placed in IL message buffers to be passed up the layers. Three emulate-mode variables provide the user with access to the information in the frame that is located beyond the control byte. These variables are *rcvd_frame_buff_seg*, *rcvd_frame_sdu_offset*, and *rcvd_frame_sdu_size*. See Section 63.1 for a more detailed discussion of the buffer components to which these variables refer.

   Make a pointer to an IL buffer by casting *rcvd_frame_buff_seg* as a *long*, shifting it left sixteen bits, adding *rcvd_frame_sdu_offset*, and casting the result to a pointer. Increment the pointer twice (thereby adding two to the offset).

```
{
 unsigned char * ptr;
 ptr = (void *)(((long)rcvd_frame_buff_seg << 16) + rcvd_frame_sdu_offset);
 ptr+=2;
}
```

   It is now pointing at the first byte in the information field. You may continue to move through the frame for its entire length, indicated in *rcvd_frame_sdu_size*.

3. *Transmit window*. Four related variables test the status of the Layer 2 window. The particular values of these variables at any given time is not significant. What is significant is how they compare to each other. The softkey status condition on the left makes the variable comparison on the right:

WINDOW FULL	*l2_current_window_edge == l2_upper_window_edge*
WINDOW EMPTY	*l2_current_window_edge == l2_lower_window_edge*
WINDOW NOT_FULL	*l2_current_window_edge != l2_upper_window_edge*
WINDOW NOT_EMPTY	*l2_current_window_edge != l2_lower_window_edge*
MORE_TO_RESEND	*l2_resend_edge != l2_lower_window_edge*
NO_MORE_TO_RESEND	*l2_resend_edge == l2_lower_window_edge*

(C) Controlling Protocol Trace Display

To enhance or suppress particular frames on the Layer 2 Protocol Trace screen in emulate or monitor mode, assign a coded value to *l2_enhance* or *l2_suppress*. The possible values are listed in Table 77-2. To assign a value to either of these variables, place the statement in an ACTIONS block. For example, display RNR frames in reverse-video and suppress display of invalid frames:

```
CONDITIONS: RCV RNR
ACTIONS:
{
 l2_enhance = 1;
}
CONDITIONS: RCV INVALID
ACTIONS:
{
 l2_suppress = 1;
}
```

Check the value of these display-control variables in a CONDITIONS block

```
CONDITIONS: RCV INFO
{
 l2_enhance == 1
}
ACTIONS:
{
 l2_enhance = 0;
}
```

or an ACTIONS block:

```
CONDITIONS: RCV INFO
ACTIONS:
{
 if(l2_enhance == 1)
   l2_enhance = 0;
}
```

77.3 Routines

Use the following routines in emulate mode only. If you try to call one of these routines in monitor mode, you will be returned to the main program menu. When you go to the Protocol Spreadsheet and search for errors, a message like the following will be displayed: "*Error 140: Unresolved reference l2_give_data.*"

(A) Receive

l2_give_data

Synopsis

extern void l2_give_data();

Description

The *l2_give_data* routine takes an interlayer message buffer associated with a received INFO frame, changes the SDU offset to point to higher-level data, and sends a DL_DATA IND primitive up to Layer 3 along with a reference to this buffer. The softkey equivalent of this routine is the GIVE_DATA action on the Protocol Spreadsheet.

Example

Layer 3 wants access to the line in order to receive and send data. Assuming the LAPD personality package is loaded at Layer 2, enter the following program:

```
LAYER: 2
     STATE: datalink
        CONDITIONS: DL_CONNECT REQ
        ACTIONS: DL_CONNECT CONF
        CONDITIONS: DL_DATA REQ
        ACTIONS: SEND INFO "((DL_DATA))"
        CONDITIONS: RCV INFO
        ACTIONS:
        {
         l2_give_data();
        }
```

(B) Transmit

resend_frame

Synopsis

extern void resend_frame(pf, first_or_next);
unsigned char pf;
unsigned char first_or_next;

Description

The *resend_frame* routine will resend either the first or next frame in the window with the P/F bit set to a specified value. The softkey equivalent of this routine is the (PROTOCL) RESEND action on the Protocol Spreadsheet.

Inputs

The first parameter is the value of the P/F bit in the frame. It may be set to either 0, 1, or 2 (for loopback).

The second parameter indicates whether the first frame in the window will be sent, or whether the next frame in the window will be sent. The first resend action will send the first frame in the window regardless of whether first or next has been selected. Legal entries are 0 (first) or 1 (next).

Example

Suppose you want to resend the entire transmit window if you receive a REJ frame.

```
LAYER: 2
    STATE: xfer
        /* Whatever conditions and actions send data precede the following condition. */

        CONDITIONS: RCV REJ RESP
        ACTIONS:
        {
         resend_frame(1, 0);
        }
        NEXT_STATE: recover
    STATE: recover
        CONDITIONS: FRAME_SENT
            MORE_TO_RESEND
        ACTIONS:
        {
         resend_frame(1,1);
        }
        CONDITIONS: FRAME_SENT
            NO_MORE_TO_RESEND
        NEXT_STATE: xfer
```

reset_nr

Synopsis

extern void reset_nr();

Description

This routine resets the N(R) field in information and supervisory frames to zero. The softkey equivalent of this routine is the (PROTOCL) RSET_NR action on the Protocol Spreadsheet.

Example

When a link is established, reset N(R).

```
LAYER: 2
    STATE: reset
        CONDITIONS: ENTER_STATE
        ACTIONS: SEND SABM
        CONDITIONS: RCV UA
        ACTIONS:
        {
         reset_nr();
        }
```

reset_ns

Synopsis

```
extern void reset_ns();
```

Description

The N(S) field in information and supervisory frames is reset to zero and the transmit window is cleared. The softkey equivalent of this routine is the (PROTOCL) RSET_NS action on the Protocol Spreadsheet.

Example

When a link is established, reset N(S).

```
LAYER: 2
     STATE: reset
        CONDITIONS: ENTER_STATE
        ACTIONS: SEND SABM
        CONDITIONS: RCV UA
        ACTIONS:
        {
         reset_ns();
        }
```

send_frame

Synopsis

```
extern void send_frame(il_buffer_number, relay_baton, data_start_offset, transmit_frame_ptr);
unsigned short il_buffer_number;
unsigned short relay_baton;
unsigned short data_start_offset;
struct send_frame_structure
{
 unsigned char sapi_type;
 unsigned char tei_type;
 unsigned char cr_type;
 unsigned char frame_type;
 unsigned char nr_type;
 unsigned char ns_type;
 unsigned char p_f_type;
 unsigned char bcc_type;
 unsigned char sapi_value;
 unsigned char tei_value;
 unsigned char cntrl_byte;
 unsigned char nr_value;
 unsigned char ns_value;
};
struct send_frame_structure * transmit_frame_ptr;
```

Description

The *send_frame* routine adds a frame-level header to an interlayer message buffer and passes the buffer to Layer 1. The softkey equivalent of this routine is the SEND action on the Protocol Spreadsheet.

Inputs

The first parameter is the interlayer message buffer number. See Section 63.3(A), Layer-Independent OSI routines.

The second parameter is the maintain bit used to hold the buffer while the send operation is being performed. See Section 63.3(A).

The third parameter is the offset from the beginning of the buffer to the start of the service data unit. See Section 63.3(A).

The fourth parameter is a pointer to the frame structure to be sent. For a description of *send_frame_structure* see Table 77-1.

Example

Send an Info frame containing a canned fox message and a good BCC onto the line.

```
{
 static unsigned short il_buffer_number;
 static unsigned short relay_baton;
 static unsigned short data_start_offset;
 struct send_frame_structure
  {
   unsigned char sapi_type;
   unsigned char tei_type;
   unsigned char cr_type;
   unsigned char frame_type;
   unsigned char nr_type;
   unsigned char ns_type;
   unsigned char p_f_type;
   unsigned char bcc_type;
   unsigned char sapi_value;
   unsigned char tei_value;
   unsigned char cntrl_byte;
   unsigned char nr_value;
   unsigned char ns_value;
  };
 struct send_frame_structure transmit_frame;
 static char transmit_string [] = "((FOX))";
}
LAYER: 2
     STATE: send_a_frame
         CONDITIONS: KEYBOARD " "
         ACTIONS:
         {
          _get_il_msg_buff(&il_buffer_number, &relay_baton);
          _start_il_buff_list(il_buffer_number,&data_start_offset);
          transmit_frame.bcc_type = 1;
          _insert_il_buff_list_cnt(il_buffer_number, data_start_offset, &transmit_string[0],
                (sizeof(transmit_string) - 1));
          send_frame(il_buffer_number, relay_baton, data_start_offset, &transmit_frame);
         }
```

78 Q.931 Library

When the Q.931 package is loaded in via the Layer Setup screen, the following external variables become available for use by the programmer. Their use on the Protocol Spreadsheet is not limited to any particular layer, though normally they belong at Layer 3.

The variables approximate Q.931 Layer 3 spreadsheet-generated conditions and actions. Refer to Section 38 for more detailed explanations of the purposes of specific conditions and actions. Sometimes the name of the variable is sufficient for identifying its related spreadsheet token. When this is not the case, the information is provided below.

78.1 Structures

There are no *extern* structures associated exclusively with Q.931.

78.2 Variables

The variables discussed below apply when the Line Setup menu shows either emulate or monitor mode. Emulate mode, however, is not supported by emulate-only conditions and actions on the Protocol Spreadsheet.

(A) Monitoring Events

Q.931 Layer 3 event variables detect packets on either side of the line. See Table 78-1. They are valid in either emulate or monitor mode. The event variables are *dte_packet* and *dce_packet*.

When the user selects DTE or DCE on the first rack of softkeys for Layer 3 conditions, a second rack appears from which he must select a particular message type. A DTE INFO condition, for example, when translated, includes two C variables, one event variable and one status variable:

```
{
 dte_packet && (m_message_type == 0x7b)
}
```

As a C programmer, you do not need to specify a message type. To include all DTE messages in a condition, use the event variable only:

```
CONDITIONS:
{
 dte_packet
}
```

Table 78-1
Q.931 Variables

Type	Variable	Value (hex/decimal)	Meaning
extern event	dte_packet		True when a DTE packet is detected. Line Setup configured for emulate or monitor mode.
extern event	dce_packet		True when a DCE packet is detected. Line Setup configured for emulate or monitor mode.
extern volatile const unsigned char	m_packet_bcc_type	1	good
		2	bad
		3	abort
			Line Setup configured for emulate or monitor mode.
extern volatile const unsigned char	m_prot_disc	*00-ff/0-255*	Actual value of protocol discriminator—should be 8. Line Setup configured for emulate or monitor mode.
extern volatile const unsigned char	m_call_ref_flag	0	origination side
		1	destination side
			Line Setup configured for emulate or monitor mode.
extern volatile const unsigned char	m_message_type_defined	0	Actual value received is not a defined value for a LAPD message type.
		1	Actual value received is one of the following valid values for a LAPD message type: 1 alerting 2 call proceeding 5 setup 7 connect d/13 setup ack f/15 connect ack 20/32 user info 21/33 suspend rej 22/34 resume rej 25/37 suspend 26/38 resume 2d/45 suspend ack 2e/46 resume ack 40/64 detach 45/69 disconnect 48/72 detach ack 4d/77 release 5a/90 release complete 60/96 cancel 62/98 facility 64/100 register 68/104 cancel ack 6a/106 facility ack 6c/108 register ack

(m_message_type_defined continued on next page)

Table 78-1 (continued)

Type	Variable	Value (hex/decimal)	Meaning
	(m_message_type_defined continued)		70/112 cancel rej 72/114 facility rej 74/116 register rej 79/121 congestion control 7b/123 info 7d/125 status Line Setup configured for emulate or monitor mode.
extern volatile const unsigned char	m_message_type	*00-ff/0-255*	Actual value of the message-type byte. Line Setup configured for emulate or monitor mode.
extern volatile const unsigned char	m_call_ref_len	*0-15*	Length of the call-reference value field. Line Setup configured for emulate or monitor mode.
extern volatile const unsigned char	m_info_element_len		Length of information element field. The total includes all information elements. Line Setup configured for emulate or monitor mode.
extern volatile const unsigned char *	m_ptr_to_call_ref		Pointer to the call-reference value field. Begins at the first byte, containing the call reference length. Line Setup configured for emulate or monitor mode.
extern volatile const unsigned char *	m_ptr_to_info_element		Pointer to the information element field. Begins at the first byte after the message-type byte. Line Setup configured for emulate or monitor mode.
extern unsigned char	l3_enhance	0 1 4 5 8 9 12/18	normal reverse low reverse low blink reverse blink blink low Line Setup configured for emulate or monitor mode.
extern unsigned char	l3_suppress	0 1	off on Line Setup configured for emulate or monitor mode.

(B) Status Variables

Status variables are those in Table 78-1 that do not include *event* in the Type column. Their associated event variables guarantee that they are updated and tested.

The softkey-generated condition for DTE Info frames is DTE INFO. The C version of the same condition should look like this:

```
CONDITIONS:
{
 dte_packet && (m_message_type == 0x7b)
}
```

1. *Packet characteristics.* All status variables in Table 78-1 containing an *m_* prefix are valid in either emulate or monitor mode: *m_packet_bcc_type, m_prot_disc, m_call_ref_len, m_call_ref_flag, m_message_type, m_message_type_defined,* and *m_info_element_len.*

2. *Pointers.* Two pointers provide access to variable-length fields. *m_ptr_to_call_ref* is the pointer to the call-reference field. *m_ptr_to_info_element* is the pointer to the information-element field.

(C) Controlling Protocol Trace Display

To enhance or suppress particular packets on the Layer 3 Protocol Trace screen in emulate or monitor mode, assign a coded value to *l3_enhance* or *l3_suppress*. The values are listed in Table 78-1. To assign a value to either of these variables, place the statement in an ACTIONS block. For example, display Suspend messages in reverse-video and suppress display of Status messages:

```
CONDITIONS: DTE SUSPEND
ACTIONS:
{
 l3_enhance = 1;
}
CONDITIONS: DTE STATUS
ACTIONS:
{
 l3_suppress = 1;
}
```

Check the value of these display-control variables in a CONDITIONS block

```
CONDITIONS: DTE INFO
{
 l3_enhance == 1
}
ACTIONS:
{
 l3_enhance = 0;
}
```

or an ACTIONS block:

```
CONDITIONS: DTE INFO
ACTIONS:
{
 if(l3_enhance == 1)
   l3_enhance = 0;
}
```

78.3 Routines

There are no routines associated exclusively with Q.931.

79 SS#7 Layer 2 Library

When the SS#7 Layer 2 package is loaded in via the Layer Setup screen, most of the following external variables become available for use by the programmer. Their use on the Protocol Spreadsheet is not limited to any particular layer, though normally they belong at Layer 2.

The SS#7 Layer 1 variables shown in Table 79-2 are accessible only when the Layer 1 SS7_CMPRESN package is loaded in via the Layer Setup screen. They do not have related spreadsheet tokens. These Layer 1 variables are included in this section since they are associated with the Layer 2 event variables in Table 79-1.

The Layer 2 variables approximate SS#7 Layer 2 spreadsheet-generated conditions and actions. Refer to Section 42 for more detailed explanations of the purposes of specific conditions and actions. Sometimes the name of the variable is sufficient for identifying its related spreadsheet token. When this is not the case, the information is provided below.

79.1 Structures

There are no *extern* structures associated exclusively with SS#7.

79.2 Variables

The variables discussed below apply when the Line Setup menu shows either emulate or monitor mode. Emulate mode, however, is not supported by emulate-only conditions and actions on the Protocol Spreadsheet.

(A) Monitoring Events

SS#7 Layer 2 events include frames detected, good or bad BCC's, and aborts. All event variables in Table 79-1 containing a *dte_* or *dce_* prefix are valid in either emulate or monitor mode. These event variables are *dte_frame*, *dce_frame*, *dte_good_bcc*, *dce_good_bcc*, *dte_bad_bcc*, *dce_bad_bcc*, *dte_abort*, *dce_abort*.

You can use both *dte* and *dce* variables relating to the same event in one conditions block. Suppose you want to count all bad BCC's from either side of the line. Enter the following CONDITIONS/ACTIONS block:

```
CONDITIONS:
{
  dte_bad_bcc || dce_bad_bcc
}
ACTIONS: COUNTER bad_bcc INC
```

Table 79-1
SS#7 Layer 2 Variables

Type	Variable	Value (hex/decimal)	Meaning
extern event	dte_frame		True when a non-suppressed DTE frame is detected. Line Setup configured for emulate or monitor mode.
extern event	dce_frame		True when a non-suppressed DCE frame is detected. Line Setup configured for emulate or monitor mode.
extern event	dte_good_bcc		True when a non-suppressed good BCC is calculated for a DTE frame. Line Setup configured for emulate or monitor mode.
extern event	dce_good_bcc		True when a non-suppressed good BCC is calculated for a DCE frame. Line Setup configured for emulate or monitor mode.
extern event	dte_bad_bcc		True when a bad BCC is calculated for a DTE frame. Line Setup configured for emulate or monitor mode.
extern event	dce_bad_bcc		True when a bad BCC is calculated for a DCE frame. Line Setup configured for emulate or monitor mode.
extern event	dte_abort		True when an abort is detected for a DTE frame. Line Setup configured for emulate or monitor mode.
extern event	dce_abort		True when an abort is detected for a DCE frame. Line Setup configured for emulate or monitor mode.
extern volatile const unsigned char	m_unit_type	1 2 3	Fill-in Signal Unit (FI) Link Status Signal Unit (LSU) Message Signal Unit (MSU) Line Setup configured for emulate or monitor mode.
extern volatile const unsigned char	m_bib	0 *non-zero*	0 1 Line Setup configured for emulate or monitor mode.
extern volatile const unsigned char	m_fib	0 1	0 1 Line Setup configured for emulate or monitor mode.

Table 79-1 (continued)

Type	Variable	Value (hex/decimal)	Meaning
extern volatile const unsigned char	m_li	0	FI
		1-2	LSU
		3-3f/63	MSU
			Line Setup configured for emulate or monitor mode.
extern volatile const unsigned char	m_so0	0	out of alignment
		1	normal
		2	emergency
		3	out of service
		4	processor out
		5	busy
			Line Setup configured for emulate or monitor mode.
extern volatile const unsigned char	m_frame_bcc_type	1	good bcc
		2	bad bcc
		3	abort
			Line Setup configured for emulate or monitor mode.
extern unsigned char	l2_enhance	0	normal
		1	reverse
		4	low
		5	reverse low
		8	blink
		9	reverse blink
		12/18	blink low
			Line Setup configured for emulate or monitor mode.
extern unsigned char	l2_suppress	0	off
		1	on
			Line Setup configured for emulate or monitor mode.

When the user selects DTE or DCE on the first rack of softkeys for Layer 2 conditions, a second rack appears from which he must select a particular frame type. A DTE FILL_IN condition, for example, when translated, includes two C variables, one event variable and one status variable:

```
{
 dte_frame && (m_unit_type == 1)
}
```

The C programmer does not need to specify a frame type. To include all frames in a condition, use the event variable only:

```
CONDITIONS:
{
 dte_frame
}
```

(B) Status Variables

Status variables are those in Table 79-1 that do not include *event* in the Type column. Their associated event variables guarantee that they are updated and tested.

The softkey-generated condition for DTE Busy Link Status Signal Unit is DTE STATUS= B. The C version of the same condition should look like this:

```
CONDITIONS:
{
dte_frame && (m_unit_type == 2) && (m_so0 == 5)
}
```

Status variables in Table 79-1 containing an *m_* prefix are valid in either emulate or monitor mode: *m_unit_type, m_bib, m_fib, m_li, m_so0,* and *m_frame_bcc_type*.

The Layer 1 variables listed in Table 79-2 are also status variables, valid in either emulate or monitor mode. Any of the Layer 2 event variables in Table 79-1 guarantee that they are updated and tested.

> **NOTE:** The SS#7 Layer 1 variables are updated frequently. If you want to track these variables for statistical purposes, we recommend that you copy their values into temporary variables.

(C) Controlling Protocol Trace Display

To enhance or suppress particular frames on the Layer 2 Protocol Trace screen in emulate or monitor mode, assign a coded value to *l2_enhance* or *l2_suppress*. The values are listed in Table 79-1. To assign a value to either of these variables, place the statement in an ACTIONS block. For example, display only Link Signal Units. Of these, display Emergency LSU's in reverse-video.

```
CONDITIONS:
{
dte_frame && (m_unit_type != 2)
}
ACTIONS:
{
l2_suppress = 1;
}
CONDITIONS:
{
dte_frame && (m_unit_type == 2) && (m_so0 == 2)
}
ACTIONS:
{
l2_enhance = 1;
}
```

Table 79-2
SS#7 Layer 1 Variables

Type	Variable	Value (hex/decimal)	Meaning
extern unsigned short	dte_frames_suppressed		Number of DTE Fill-In or Link Status Signal Units suppressed since the last non-suppressed frame. Line Setup configured for emulate or monitor mode.
extern unsigned short	dce_frames_suppressed		Number of DCE Fill-In or Link Status Signal Units suppressed since the last non-suppressed frame. Line Setup configured for emulate or monitor mode.
extern unsigned short	dte_flags		Number of DTE flags received since the last non-suppressed frame. Line Setup configured for emulate or monitor mode.
extern unsigned short	dce_flags		Number of DCE flags received since the last non-suppressed frame. Line Setup configured for emulate or monitor mode.

Check the value of these display-control variables in a CONDITIONS block

```
CONDITIONS:
{
 dte_frame && (m_unit_type == 2) && (m_so0 == 2) && (l2_enhance == 0)
}
ACTIONS:
{
 l2_enhance = 1;
}
```

or an ACTIONS block:

```
CONDITIONS:
{
 dte_frame && (m_unit_type == 2) && (m_so0 == 2)
}
ACTIONS:
{
 if(l2_enhance == 0)
   l2_enhance = 1;
}
```

79.3 Routines

There are no routines associated exclusively with SS#7.

80 SS#7 Layer 3 Library

When the SS#7 Layer 3 package is loaded in via the Layer Setup screen, the following external variables become available for use by the programmer. Their use on the Protocol Spreadsheet is not limited to any particular layer, though normally they belong at Layer 3.

The variables approximate SS#7 Layer 3 spreadsheet-generated conditions and actions. Refer to Section 43 for more detailed explanations of the purposes of specific conditions and actions. Sometimes the name of the variable is sufficient for identifying its related spreadsheet token. When this is not the case, the information is provided below.

80.1 Structures

There are no *extern* structures associated exclusively with SS#7.

80.2 Variables

The variables discussed below apply when the Line Setup menu shows either emulate or monitor mode. Emulate mode, however, is not supported by emulate-only conditions and actions on the Protocol Spreadsheet.

(A) Monitoring Events

SS#7 Layer 3 event variables detect Message Signal Units on either side of the line. See Table 80-1. They are valid in either emulate or monitor mode. The event variables are *dte_packet* and *dce_packet*.

When the user selects DTE or DCE on the first rack of softkeys for Layer 3 conditions, a second rack appears from which he must select a particular MSU type. A DTE NETM condition, for example, when translated, includes two C variables, one event variable and one status variable:

```
{
 dte_packet && (m_sio_si == 0)
}
```

As a C programmer, you do not have to specify an MSU type. To include all DTE Message Signal Units in a condition, use the event variable only:

```
CONDITIONS:
{
 dte_packet
}
```

Table 80-1
SS#7 Layer 3 Variables

Type	Variable	Value (hex/decimal)	Meaning
extern event	dce_packet		True when a DCE packet is detected. Line Setup configured for emulate or monitor mode.
extern event	dte_packet		True when a DTE packet is detected. Line Setup configured for emulate or monitor mode.
extern volatile const unsigned char	m_sio_ni	0 40/64 80/128 c0/192	international 0 international 1 national 0 national 1 Line Setup configured for emulate or monitor mode.
extern volatile const unsigned char	m_sio_priority	0 10/16 20/32 30/48	priority=0 priority=1 priority=2 priority=3 Line Setup configured for emulate or monitor mode.
extern volatile const unsigned char	m_sio_si	*0-7* 0 1 2 3 4 5 6 7 *8-f/8-15*	User Part: netm ntr nts sccp tup isdn dup0 dup1 spare Line Setup configured for emulate or monitor mode.
extern volatile const unsigned char	m_code_type	 1 2	Test headers:† (high 4 bits not defined) ltm lta

(m_code_type continued on next page)

† The high four bits in test headers are not defined. To check the value of *m_code_type* for test headers, *and* *m_code_type* with 0x0f:

header = m_code_type & 0x0f;

For LTM's, *header* equals 1; for LTA's, *header* equals 2.

Table 80-1 (continued)

Type	Variable	Value (hex/decimal)	Meaning
(m_code_type continued)			SCCP headers:
		1	cr
		2	cc
		3	cref
		4	rlsd
		5	rlc
		6	dt1
		7	dt2
		8	ak
		9	udt
		a/10	udts
		b/11	ed
		c/12	ea
		d/13	rsr
		e/14	rsc
		f/15	err
		10/16	it
			NETM headers:
		11/17	coo
		12/18	eco
		13/19	rct
		14/20	tfp
		15/21	rsp (US format only)
		15/21	rst (CCITT format only)
		16/22	lin
		18/24	dlc
		21/33	coa
		22/34	eca
		23/35	tfc
		24/36	tcp (US format only)
		25/37	rsr (US format only)
		25/37	rst (CCITT format only, national option)
		26/38	lun
		28/40	css
		34/52	tfr
		35/53	rcp (US format only)
		36/54	lia
		38/56	cns
		44/68	tcr (US format only)
		45/69	rcr (US format only)
		46/70	lua
		48/72	cnp
		51/81	cbd
		54/84	tfa
		56/86	lid
		61/96	cba
		64/100	tca (US format only)
		66/102	lfu
		76/118	lll
		86/134	lrl

(m_code_type continued on next page)

Table 80-1 (continued)

Type	Variable	Value (hex/decimal)	Meaning
(m_code_type continued)			TUP headers:
		6	anu
		10	reserved
		11/17	iam
		12/18	gsm
		13/19	grq
		14/20	acm
		15/21	sec
		16/22	anc
		17/23	rlg
		18/24	mgb
		19/25	cfm
		21/33	iai
		24/36	chg
		25/37	cgc
		26/38	ann
		27/39	blo
		28/40	mba
		29/41	cpm
		31/49	sam
		32/50	cot
		35/53	nnc
		36/54	cbk
		37/55	bla
		38/56	mgu
		39/57	cpa
		41/65	sao
		42/66	ccf
		45/69	adi
		46/70	clf
		47/71	ubl
		48/72	mua
		49/73	csv
		55/85	cfl
		56/86	ran
		57/87	uba
		58/88	hgb
		59/89	cvm
		65/101	ssb
		66/102	fot
		67/103	ccr
		68/104	hba
		69/105	crm
		75/117	unn
		76/118	ccl
		77/119	rsc
		78/120	hgu
		79/121	cli
		85/133	los
		88/136	hua

(m_code_type continued on next page)

Table 80-1 (continued)

Type	Variable	Value (hex/decimal)	Meaning
(m_code_type continued)			*(TUP headers continued)*
		95/149	sst
		98/152	grs
		a5/165	acb
		a8/168	gra
		b5/181	dpn
		b8/184	sgb
		c5/197	mpr
		c8/200	sba
		d8/216	sgu
		e8/232	sua
		f5/245	eum
		f6/246	eam
			ISDN headers:
		1	iam
		2	sam
		3	inr
		4	inf
		5	cot
		6	acm
		8	fot
		9	anm
		a/10	ubm
		b/11	rel
		d/12	pau
		e/14	res
		f/15	rlsd
		10/16	rlc
		11/17	ccr
		12/18	rsc
		13/19	blo
		14/20	ubl
		15/21	bla
		16/22	uba
		17/23	grs
		18/24	cgb
		19/25	cgu
		1a/26	cgba
		1b/27	cgua
		1c/28	cmr
		1d/29	cmc
		1e/30	rcm
		1f/31	far
		20/32	faa
		21/33	frj
		22/34	fad
		23/35	fai
		25/37	csvr
		26/38	csvs
		27/39	drs
		28/40	pam
		29/41	gra
			Line Setup configured for emulate or monitor mode.

Table 80-1 (continued)

Type	Variable	Value (hex/decimal)	Meaning
extern volatile unsigned long	m_label_dpc	*0-3fff/ 0-16383* *0-ffffff/ 0-16777215*	CCITT format (2 bytes) ANSI format (3 bytes) Line Setup configured for emulate or monitor mode.
extern volatile unsigned long	m_label_opc	*0-3fff/ 0-16383* *0-ffffff/ 0-16777215*	CCITT format (2 bytes) ANSI format (3 bytes) Line Setup configured for emulate or monitor mode.
extern volatile const unsigned char	m_label_sls	*0-f/0-15* *0-1f/0-31*	CCITT format ANSI format Line Setup configured for emulate or monitor mode.
extern volatile unsigned short	m_cic	*0-fff/0-4095* *0-ffff/0-65535*	TUP MSUs ISDN MSUs Line Setup configured for emulate or monitor mode.
extern unsigned char	l3_enhance	0 1 4 5 8 9 12/18	normal reverse low reverse low blink reverse blink blink low Line Setup configured for emulate or monitor mode.
extern unsigned char	l3_suppress	0 1	off on Line Setup configured for emulate or monitor mode.

(B) Status Variables

Status variables are those in Table 80-1 that do not include *event* in the Type column. Their associated event variables guarantee that they are updated and tested.

The softkey-generated condition for NETM Message Signal Units on the DTE side of the line is DTE NETM. The C version of the same condition should look like this:

```
CONDITIONS:
{
 dte_packet && (m_sio_si == 0)
}
```

Most status variables in Table 80-1 contain an *m_* prefix: *m_sio_ni, m_sio_priority, m_sio_si, m_code_type, m_label_dpc, m_label_opc, m_label_sls,* and *m_cic*.

(C) Controlling Protocol Trace Display

To enhance or suppress particular packets on the Layer 3 Protocol Trace screen in emulate or monitor mode, assign a coded value to *l3_enhance* or *l3_suppress*. The values are listed in Table 80-1. To assign a value to either of these variables, place the statement in an ACTIONS block. For example, display only messages with NETM headers. Of these, display Transfer Restricted headers in reverse-video.

```
CONDITIONS:
{
 dte_packet && (m_sio_si != 0)
}
ACTIONS:
{
 l3_suppress = 1;
}
CONDITIONS:
{
 dte_packet && (m_sio_si == 0) && (m_code_type == 0x34)
}
ACTIONS:
{
 l3_enhance = 1;
}
```

Check the value of these display-control variables in a CONDITIONS block

```
CONDITIONS:
{
 dte_packet && (m_sio_si != 0) && (l2_suppress == 0)
}
ACTIONS:
{
 l2_suppress = 1;
}
```

or an ACTIONS block:

```
CONDITIONS:
{
 dte_packet && (m_sio_si != 0)
}
ACTIONS:
{
 if(l2_suppress == 0)
   l2_suppress = 1;
}
```

80.3 Routines

There are no routines associated exclusively with SS#7.

Appendix A: Operator Messages

Appendix A1: Interactive Messages

The following messages are displayed on the second line of the screen, normally during execution of menu–screen functions.

MESSAGE	MEANING
Attempt to transfer source to itself	Source selections in the **From** and **To** fields on the Data Transfer screen are the same. Change one selection. To use one drive to perform data transfer involving two disks, change **To** selection to NEW.
Attempted to mount uninitialized disk	Check disk. It may require formatting.
Attempting to initialize link	Physical link being established prior to transfer.
BNDX message request failed	Message should not normally appear. If it recurs, contact Customer Service.
Backup complete, no errors	Duplication process is successful.
Bad object file format	Data is not recognized in format of object file. Try again to save the source file as an object file.
Can't load object file – Incompatible FEB installed	Current hardware is different from hardware of unit on which object file was saved. Save the source file as an object file on the unit on which it will be loaded and run.
Can't load object file – Incompatible MPM addressing	Current hardware is different from hardware of unit on which object file was saved. Save the source file as an object file on the unit on which it will be loaded and run.

Can't load object file - Incompatible mux installed	Current hardware is different from hardware of unit on which object file was saved. Save the source file as an object file on the unit on which it will be loaded and run.
Can't load object file - Insufficient MPMs	Current hardware is different from hardware of unit on which object file was saved. Save the source file as an object file on the unit on which it will be loaded and run.
Can't load object file - No mux installed	Current hardware is different from hardware of unit on which object file was saved. Save the source file as an object file on the unit on which it will be loaded and run.
Can't read disk	Make sure disk is correctly inserted. If message recurs, disk may be bad.
Cannot append to a wrapped DAT	Record Setup menu shows **Stop at:** ENDLESS LOOP. End of data acquisition tracks was reached, so wrapping occurred. Then, Data Transfer command attempted to append data from source to the end of DAT on destination disk. Select **Start At:** BEGIN on Data Transfer screen.
Cannot copy a file to itself	Be certain that name of destination file on File Maintenance screen differs from name of source file.
Cannot copy directory tree into itself	Attempt to copy a directory into one of its subdirectories. For example, a command to copy */usr* into */usr/programs* will fail.
Cannot delete a non-empty directory	File named for deletion is directory containing files. Before deleting directory, delete or move files.
Cannot open file	In attempt to load or save a file, the file could not be opened. Check the write-protect window. It should be closed to write to the disk.
Cannot open redirect file	Printer Setup menu shows that output will be redirected to a file. Check to make sure that the disk is properly inserted in the correct drive and is not write-protected.

Cannot move file across disk boundaries	Attempt to rename a file from one disk to another. Make sure only one drive is specified on File Maintenance screen.
Cannot remove an open file	Attempt was made to copy a directory into itself. Files being copied also need to be deleted, but cannot be since they are open. Copy the directory to another source. In general, close files before attempting to delete them.
Cannot unmount disk, files open	Attempt was made to remove disk before operation was completed.
Cannot write to redirect file	Printer Setup menu shows that output will be redirected to a file. Error in trying to write to the file. Check disk.
Change floppy disk 1	During multi–disk recording operation, disk in drive 1 has been filled. Remove old disk and insert new one.
Change floppy disk 2	During multi–disk recording operation, disk in drive 2 has been filled. Remove old disk and insert new one.
Character buffer not yet allocated	**From** field on the Data Transfer screen shows CHARBUFFER, but unless Run has been executed, there is no character buffer. Press [RUN], and then try the transfer operation again.
Compilation aborted	User has pressed ABORT softkey or [PROGRAM] to arrest the Compile operation (from the File Maintenance menu). Destination file may have been partially overwritten if compile was to an existing file.
Compilation completed	The Compile command (from the File Maintenance menu) has been executed.
Compilation failed – Errors detected	The Compile command (from the File Maintenance menu) has been aborted because of errors. Go to the Protocol Spreadsheet and press [EDIT], [F8] to display the first error message.
Compilation is in progress	The Compile command (from the File Maintenance menu) is being executed to compile and save a file of standard C code as object code.

Copy completed	Selected file(s) has (have) been copied successfully.
Copy is in progress	Selected file(s) is (are) being copied.
Could not load a layer personality package	Attempt to load a protocol package from the Layer Setup screen has failed. Make sure that correct disk is installed in drive indicated on menu. If attempt still fails, package may have been corrupted.
Current test invalidated	Changes have been made to the menu screens or Protocol Spreadsheet which invalidate a loaded object file.
Data transfer source and destination are the same	Source selections in the **From** and **To** fields on the Data Transfer screen are the same. Change one selection. To use one drive to perform data transfer involving two disks, change **To** selection to NEW.
Destination disk does not contain user file system	Check file contents on File Maintenance screen. If disk is intended for user files, you may need to allocate disk space to the filing system. Use Disk Utilities screen to check disk allocation.
Destination file is a directory	Copy or save operation not complete because file named to receive copy is a directory. Change destination filename and re-execute.
Destination file is write protected	Change destination filename or write-enable file. Then repeat save or copy operation.
Directory file expected	Check directory named in Change Directory command. Use only names labeled DIR in file listings.
Directory is empty	Attempted to copy the contents of an empty directory.
Directory is not empty	Directory cannot be deleted until all of the files it contains have been deleted.
Disk corrupted	Disk is worn out or damaged and should not be used for future operations.
Disk duplication aborted	Operator has aborted disk duplication. Data on destination disk may have been partially overwritten.

Disk duplication in progress	Disk is being duplicated. Do not remove disks from active drives.
Disk formatted	Formatting operation is complete.
Disk full	No space left on disk to perform operation. Use a new disk, or remove unneeded data from disk.
Disk not mounted	Re-insert disk and attempt operation again. Also try to power-up again. If message recurs, the disk may be bad.
Disk record error (controller error) -- Aborted	Disk may be write protected or recording too fast. Also may be an internal error or bad disk. Try again with a new disk. Contact Customer Service if it recurs.
Disk record error (timeout) -- Aborted	May be an internal error or bad disk. Try again with a new disk. Disk may be write protected or recording too fast.
Display screen command failed	Message should not normally appear. Contact Customer Service if it recurs.
Entering testprep	First status message entering Run mode. Test preparation mode precedes compilation of program.
Error during load of code file	Check code files. Message indicates code file is not found, or code file has been modified.
Error trying to print file	Try printing again. If attempt fails, disk file is probably corrupted.
Error trying to print screen	Try printing again. If problem recurs, contact Customer Service.
Errors occurred during load	Try loading again. If attempt fails, disk file is probably corrupted.
Errors occurred during save	Try saving again. If attempt fails, disk is probably corrupted.
Errors occurred during testprep	Attempt to perform Save command as a object file before the program had ever been compiled. An error was detected as compilation was attempted. Go to the Protocol Spreadsheet and search for errors. The Line Setup menu, for example, may show **Mode:** MONITOR and **Source:** DISK, but no disk is present in the selected drive.

Fatal Hardware Error	Invalid hardware setup. Contact Customer Service.
Fatal Software Error	Message should not normally appear. If it recurs, contact Customer Service.
FE buffer overflowed - Incoming data halted	Data is coming in faster than it can be received.
File access error	File named cannot be accessed. Check disk.
File copy aborted	User has pressed ABORT softkey to arrest copy operation. Destination file may have been partially overwritten if copy was to an existing file.
File is a directory	Operation, View for example, could not be performed on a directory. Select or enter the name of a file. See listings on File Maintenance screen for entry NOT labeled DIR.
File is write-protected	File named cannot be deleted or saved. Check name. To perform operation on named file, write-enable it from the File Maintenance screen.
File loaded	File has been loaded successfully as a Program, Setup, or Object file.
File name not found	Filename (or directory) as entered does not appear in listings. Check spelling of entry. Make sure you are operating in correct directory.
File name already exists	Attempt to use the Make Directory command, naming a directory that already exists.
File saved	File has been saved successfully as a Program, Setup, or Object file.
File size can't be increased, index block full	File is larger than the file system can handle.
Formatting disk - max floppy disk DAT allowed = 1422K bytes	Too much space specified for data acquisition tracks. Maximum space has been allocated.
Formatting disk - max hard disk DAT allowed = 20774K bytes	Too much space specified for data acquisition tracks. Maximum space has been allocated.

Formatting disk	Formatting in progress. Do not remove disk from active drive.
Formatting will destroy data – Depress F1 key to continue.	Message appears when a formatted disk has been inserted for reformatting. Press ABORT to avoid overwriting data. Press F1 to format the disk.
Function failed -- Check media	Attempt operation again. If operation still fails, disk may be bad. Try new disk.
Function(s) not yet implemented	Operation attempted is not available with the software version installed.
Il buffer services error	Error in using OSI variables or routines. May occur, for example, if operation is attempted on buffer which no longer exists. Set maintain bits at each layer that needs to reserve the buffer for subsequent operations.
Illegal device name	Message should not normally appear. If it recurs, contact Customer Service.
Illegal expansion unit, not 1–255	Message should not normally appear. If it recurs, contact Customer Service.
Illegal file number passed to open	Message should not normally appear. If it recurs, contact Customer Service.
Illegal major device number given	Message should not normally appear. If it recurs, contact Customer Service.
Illegal parameter to volume init. function	Message should not normally appear. If it recurs, contact Customer Service.
Illegal pathname	Pathname provided is incomplete or invalid. Check entry.
Illegal position parameter, not 0–2	Message should not normally appear. If it recurs, contact Customer Service.
Illegal synchronization mode, not 0–2	Message should not normally appear. If it recurs, contact Customer Service.
Indirect stat update msg received	Message should not normally appear. If it recurs, contact Customer Service.

Insert next disk -- Depress F1 key to continue	More than one disk required to perform duplication.
Insert destination disk, depress F1 key to continue	Operation involving more than one disk being performed using one drive.
Insert source disk, depress F1 key to continue	Operation involving more than one disk being performed using one drive.
Internal disk sub-system error	Message should not normally appear. If it recurs, contact Customer Service.
Inter-processor communication overrun	Communication from MPM to CPM occurring too fast for CPM. Available buffer space exceeded.
Invalid contents in field	Entry made in menu field is illegal.
Invalid DAT block version number	Each block in DAT has a version number. If it is wrong, the disk may be corrupted.
Invalid DAT version number	Header in DAT has a version number. If it is wrong, the disk may be corrupted.
Invalid data type	During attempted playback, INTERVIEW did not recognize type of data. Be certain recorded data rather than program data is being accessed.
Invalid destination	Destination file in a Copy command is a relative pathname on a drive which is not the current drive.
Invalid disk sub-system function number	Message should not normally appear. If it recurs, contact Customer Service.
Invalid file identifier, no such open file	Message should not normally appear. If it recurs, contact Customer Service.
Invalid file identifier, out of range	Message should not normally appear. If it recurs, contact Customer Service.
Invalid filetype	Command cannot be used on file type indicated. A Load command, for example, is not valid for SYS files.

Invalid filetype for viewing	View command cannot be used for data in file indicated. Files with type SYS, for example, cannot be viewed.
Invalid layer number	User has entered layer number out of valid range.
Invalid object code version	Object file was saved under a different version than current software. Save the source file as an object file using the same software with which is will be loaded and run.
Invalid section name	Message should not normally appear. If it recurs, contact Customer Service.
Invalid stat update msg received	Message should not normally appear. If it recurs, contact Customer Service.
Load aborted	Operator has aborted Load operation. Program already residing in INTERVIEW may have been altered.
Load is in progress	Selected Program, Setup, or Object file is being loaded.
Loaded package and configuration screen don't match	Message should not normally appear. If it recurs, contact Customer Service.
Marked entry not copied	Too many items have been marked for single operation. Not all files marked have been copied. Check listings on the File Maintenance screen. Files still marked are not yet copied. Repeat copy operation on remaining files.
Marked entry not deleted	Too many items have been marked for single operation. Not all files marked have been deleted. Check listings on the File Maintenance screen. Files still marked are not yet deleted. Repeat delete operation on remaining files.
Marked entry not printed	Too many items have been marked for single operation. Not all files marked have been printed. Check listings on the File Maintenance screen. Files still marked are not yet printed. Repeat print operation on remaining files.

Maximum number of entries exceeded	Error on Tabular Statistics screen. Maximum number of entries is 100.
Maximum disks already mounted	Message should not normally appear. If it recurs, contact Customer Service.
Memory has not been unlocked yet	Message should not normally appear. If it recurs, contact Customer Service.
Message exchange full	Message should not normally appear. If it recurs, contact Customer Service.
Message ID too big	Message should not normally appear. If it recurs, contact Customer Service.
Move would destroy directory tree structure	Attempt to rename a directory the same as one of its subdirectories.
MPM -- Bus error	May indicate a hardware problem, but check program for logic errors relating to storage allocation. May have attempted to access something that doesn't exist.
MPM -- Divide fault	Program may include an attempt to divide by zero. Check for other logic errors in program.
MPM -- Processor fault	May indicate a hardware problem, but check program for logic errors relating to storage allocation. May have attempted to access something that doesn't exist.
MPM -- Memory fault	Logic error in program, typically relating to storage allocation. May have attempted to access something that doesn't exist, accessing an array outside of its range, for example. May also indicate type mismatches.
MPM -- Stack fault	Logic error in program, typically relating to storage allocation. May have attempted to access something that doesn't exist, accessing an array outside of its range, for example. May also indicate type mismatches.
NEWDISK illegal with source of hard disk	When **To** field for a Copy command is NEW, it means that the same drive will be used to perform a copy involving two disks. Change **From** field to FD1 or FD2.

NEWDISK illegal with source of RAM or hard disk	When **To** field for a Data Transfer command is NEW, it means that the same drive will be used to perform a transfer involving two disks. Change **From** field to FD1 or FD2.
No DAT RAM currently allocated	Attempt to transfer data from RAM without it having been recorded previously to RAM.
No default directory set	Message should not normally appear. If it recurs, contact Customer Service.
No file name specified	Enter or indicate file on which operation attempted is to be performed.
No package loaded for this layer	Selection has been made on the Layer Setup screen, but no protocol package has been loaded. Return to Layer Setup, check selection, and press XEQ.
No packages loaded	Selections have been made on the Layer Setup screen, but no protocol packages have been loaded. Return to Layer Setup, check selections, and press XEQ.
No RAM recording memory available	Program is too large to be recorded into available RAM.
No start of section indicator	Operation on file cannot be performed because (1) file is not a program or setup, (2) format of the file is invalid, or (3) file has been corrupted.
No message entered in message buffer	Check BERT screen. Configured menu indicated a message would be sent, but none was entered.
Obsolete object program – Source must be recompiled	Object file is incompatible with current software.
Out of memory	Insufficient memory to perform operation. (Program is too large to run.)
Operation not allowed on specified file	Selected command cannot be used on file indicated.
Parent directory of file is write–protected	Parent directory must be write–enabled before you can modify or delete this file.
Parent directory of target file does not exist	Check spelling of directory.

Play underrun	Data could not be output at speed requested.
Premature end of section	Operation on file cannot be performed because (1) file is not a program or setup, (2) format of the file is invalid, or (3) file has been corrupted.
Previous lock user has died	Message should not normally appear. If it recurs, contact Customer Service.
Print queue is full	Maximum number of print jobs has been requested. Wait for some requests to be completed. Then repeat print operation.
Printing is done	Print jobs requested are completed.
Record overrun	Data being received too rapidly for capture to RAM.
Remove screen command failed	Message should not normally appear. If it recurs, contact Customer Service.
Replace screen command failed	Message should not normally appear. If it recurs, contact Customer Service.
Resetting compiled test	Program being run again without recompiling. Menus can be viewed and selected changes can be made to menus without forcing a recompile.
Routine calling save_prog_setup is unknown	Message should not normally appear. If it recurs, contact Customer Service.
Save aborted	Operator has aborted save operation. If save was to an existing file, the file may have been partially overwritten.
Save is in progress	Selected Program, Setup, or Object file is being saved.
Seek attempted before beginning of DAT	Data Transfer screen shows transfer from disk with **Start At Block** entry that precedes the block number at which data actually begins. DAT may begin at block 20, for example. If you enter **Start At Block: 2,** this error message will be displayed. To guarantee that data transfer starts from the beginning of DAT, enter zero in the **Start At Block** field. Zero is a special entry. It references the beginning of DAT, regardless of what the actual block number may be. Any other entry is interpreted as a literal block number.

Seek attempted past end of DAT	Data Transfer screen shows transfer from disk with **Start At Block** entry that exceeds the block number at which data actually ends. DAT may end at block 100, for example. If you enter **Start At Block: 101**, this error message will be displayed. To guarantee that data transfer starts from the beginning of DAT, enter zero in the **Start At Block** field. Zero is a special entry. It references the beginning of DAT, regardless of what the actual block number may be. Any other entry is interpreted as a literal block number.
Seek attempted past end of file	Message should not normally appear. If it recurs, contact Customer Service.
Source & destination can't be the same disk	Error in entries made for disk duplication. Check disks selected.
Source disk does not contain user file system	Check disk contents on the Disk Utility screen. If disk is intended for user files, you may need to allocate space to the filing system.
Source file not found	In Interview 10/15/20 file transfer, the source file which was specified does not exist.
Stopped at end of DAT	During playback, stopped at end of recorded data. During recording, stopped at end of data acquisition tracks.
TEST PREPARATION Phase 1	Program is being compiled.
TEST PREPARATION Phase 2	Program is being compiled.
TEST PREPARATION Phase 3	Program is being compiled.
TEST PREPARATION Phase 4	Program is being compiled.
TEST PREPARATION Phase 5	Program is being compiled.
TEST PREPARATION Phase 6	Program is being compiled.
TEST PREPARATION Phase 7	Program is being compiled.
There are no free locks left	Message should not normally appear. If it recurs, contact Customer Service.

Token in load file is invalid	Operation on file cannot be performed because (1) file is not a program, setup, or object, (2) format of the file is invalid, or (3) file has been corrupted.
Token is incomplete	Operation on file cannot be performed because (1) file is not a program, setup, or object, (2) format of the file is invalid, or (3) file has been corrupted.
Too many files in directory	Maximum number of files that can be displayed is 200. If the current directory contains more than 200 files, this message is displayed.
Too many files on disk, FLIST full	Directory area on disk is full, although there may be more space available for recording. Delete unnecessary filenames to gain access to free space remaining on disk.
Too many open files for process	Each process is limited to a maximum of ten files open at one time.
Too many open files for system	There is a system-wide limit of 20 files open at one time.
Too many processes using disk sub-system	There can be no more than twelve processes using file I/O simultaneously.
Too many source files selected	Operator has used MARK to select multiple source files for executing the Compile command (from the File Maintenance menu). Select only one source file to compile and save as a linkable-object (LOBJ) file.
Transfer aborted	Data transfer operation has been aborted. Partial transfer of data may have occurred, overwriting storage medium at destination.
Transfer complete	Data transfer has been completed successfully.
Transfer in progress	Data transfer being executed.
Transmit overrun	Attempt to transmit data faster than unit can transmit.
Unable to access disk	No file named in a Data Transfer to a file, disk not in drive, disk is write-protected, or disk is unformatted.

Unable to access disk in selected drive.	During multi-disk recording, the next drive in sequence does not contain a disk, contains a write-protected disk, or contains an unformatted disk. During file maintenance operations or disk duplication, source disk is not present in selected drive, is write-protected, or is unformatted.
Unable to access m_list	Disk error. Check disk and try operation again. If message recurs, disk may need reformatting.
Unable to execute XEQ key	Attempt to execute a File Maintenance command before the current directory is displayed.
Unable to open DAT	There are no data acquisition tracks on the disk being accessed.
Unable to open file	Disk error. Check disk and try operation again. If message recurs, disk may need reformatting.
Unable to open next disk	Recording using more than one disk. Next disk may not be installed.
Unable to read DAT info block	DAT block not where indicated. Disk may be corrupted.
Unable to read file	Check disk.
Unable to read from DAT	Check disk.
Unable to write to DAT	1) May have attempted to transfer more data to destination disk than the space allocated for data during disk formatting. A Summary of the disk may show no free space remaining for data acquisition. 2) Disk may be write-protected.
Unhandled CPM interrupt	Message should not normally appear. If it recurs, contact Customer Service.
Unhandled MPM interrupt	Press [PROGRAM] to recover from this error. Message should not normally appear, however. If it recurs, contact Customer Service.
Unknown DAT type	Data acquisition tracks may have been recorded on a unit with more recent software than is installed in the unit being used to playback the data.

Unknown filetype	Check entry in file listing. Check disk. Try operation again. If message recurs, delete and recreate file (if possible). Some file types may not be known for certain operations. A Print command on a SYS file, for example, generates this error message.
Unprintable screen	Screen requested cannot be printed. Refer to section on printing for printable screens.
Unrecoverable error during format	Format operation failed. Make sure disk is inserted properly in selected drive. Check disk type. One Mbyte disks are not supported. If disk type is correct, re-attempt formatting. If second attempt fails, disk may be bad.

Appendix A2: Error Messages Issued by C Translator

If a spreadsheet program contains any of the following errors, the compilation will be interrupted and you will be returned to the Protocol Spreadsheet. A diagnostic message about the first error will be displayed at the top (second line) of the screen. To search for additional error messages, press F8.

MESSAGE	MEANING
AR "C" conditions text too long	A C region in a CONDITIONS block is more than 300 characters.
Bad format in object file	Unsuccessful attempt to access linkable-object file via the OBJECT block identifier. Use the Compile command on the File Maintenance screen to recreate the LOBJ file, and try again.
BIB value out of range	An SS#7 condition at Layer 2 specifies a BIB= value that is not zero or one.
Bit mask exceeds maximum length	A FLAG condition or FLAG name SET action includes a bit mask that exceeds 16 bits. In other uses, bit mask is typically eight bits.
Cannot find object file	Attempt to access a linkable-object file via the OBJECT block identifier. Either the file does not exist, or it resides in a directory not included in the search path.
Constant reference stack overflow	1) Attempt to nest constants more than eight deep, or 2) constants are defined circularly.
Constant value too long	Context in which constant is used determines what value is too long. A constant in a Layer 1 receive string condition, for example, when expanded, cannot exceed 32 characters.
Duplicate state name	Attempt to use state name twice in the same test.

Duplicate test name	Attempt to use test name twice in the same task (layer).
Edit buffer full	Spreadsheet program is too large. Use include files.
Empty conditions section	There is no entry for a CONDITIONS block.
FIB value out of range	An SS#7 condition at Layer 2 specifies an FIB= value that is not zero or one.
Identifier exceeds maximum length	Message should not normally appear. It means, however, that an identifier is too long for the context in which it is being used.
Idle string must contain exactly one character	Layer 1 IDLE action includes a string with more than one character.
Illegal bit value	Bit has been assigned a value other than zero or one. In X.25 protocol, for example, the user supplies a value for the Q, D, or M bit at Layer 3.
Illegal cause value	An X.25 condition at Layer 3 specifies a numeric value for the CAUSE= selection which is outside the valid range. Select a value between hexadecimal 0 and FF.
Illegal CIC type for ISDN	An SS#7 condition at Layer 3 specifies a CIC= value for an ISDN header which is outside the valid range. Select a value between hexadecimal 0 and FFFF.
Illegal CIC type for TUP	An SS#7 condition at Layer 3 specifies a CIC= value for a TUP header which is outside the valid range. Select a value between hexadecimal 0 and FFF.
Illegal control byte	An X.25, LAPD, SDLC, or SNA condition at Layer 2 (as in the example which follows) specifies a value for the frame type which is outside the valid range: CONDITIONS: DTE OTHER 1FF. Select a value between hexadecimal 0 and FF.
Illegal diag value	An X.25 condition at Layer 3 specifies a value for the DIAG= selection which is outside the valid range. Select a value between hexadecimal 0 and FF.

Illegal DPC type	An SS#7 condition at Layer 3 specifies a value for the DPC= selection which is outside the valid range. For CCITT format, select a value between hexadecimal 0 and 3FFF. For ANSI format, select a value between hexadecimal 0 and FFFFFF.
Illegal frame address	An X.25, SDLC, or SNA condition at Layer 2 (as in the example which follows) specifies a value for the frame address which is outside the valid range: CONDITIONS: DTE INFO ADDR= 1FF. Select a value between hexadecimal 0 and FF.
Illegal LCN value	An X.25 condition at Layer 3 specifies a value for the LCN= selection which is outside the valid range. Select a value between hexadecimal 0 and FFF.
Illegal OPC type	An SS#7 condition at Layer 3 specifies a value for the OPC= selection which is outside the valid range. For CCITT format, select a value between hexadecimal 0 and 3FFF. For ANSI format, select a value between hexadecimal 0 and FFFFFF.
Illegal path number	An X.25 condition or action at Layer 3 specifies a value for the PATH= selection which is outside the valid range. Select a value between zero and eight.
Illegal P/F bit	An X.25, SDLC, or SNA condition or SEND action at Layer 2 specifies a value for the P/F= selection that is not zero or one.
Illegal PR value	An X.25 SEND action at Layer 3 specifies a value for the PR= selection which is outside the valid range. Select a value between zero and 127.
Illegal PS value	An X.25 SEND action at Layer 3 specifies a value for the PS= selection which is outside the valid range. Select a value between zero and 127.
Illegal receive count	In an X.25, SDLC, LAPD, or SNA Layer 2 SEND action, the value specified for N(R) is out of range. Select a value between zero and 127.
Illegal SAPI value	A LAPD condition or SEND action at Layer 2 specifies a value for the SAPI= selection which is outside the valid range. Select a value between hexadecimal 0 and 3F.

Illegal send count	In an X.25, SDLC, LAPD, or SNA Layer 2 SEND action, the value specified for N(S) is out of range. Select a value between zero and 127.
Illegal SI type	In an SS#7 Layer 3 OTHER condition, the value specified for Service Information is out of range. Select a value between hexadecimal 0 and FF.
Illegal SLS type	An SS#7 condition at Layer 3 specifies a value for the SLS= selection which is outside the valid range. For CCITT format, select a value between hexadecimal 0 and F. For ANSI format, select a value between hexadecimal 0 and 1F.
Illegal TEI value	A LAPD condition or SEND action at Layer 2 specifies a value for the TEI= selection which is outside the valid range. Select a value between hexadecimal 0 and 7F.
Incomplete EIA action	Required number of softkey selections have not been made for a Layer 1 EIA action.
Invalid constant reference	Valid characters for a constant name include 0–9, upper- and lower-case letters, and underscores. Name cannot begin with a number. The message also may indicate that a special "constant" of the form ((*name[45]*)) has been used, but the string using it is missing the enclosing quotation marks.
Invalid counter arguments	In a SEND action, the string to be sent contains a reference of the form ((*counter[n]*)). The value of *n* is out of range. Select a value between zero and three.
Invalid counter value	COUNTER condition or a COUNTER name SET action specifies a value for the counter which is outside the valid range. Select a value between zero and 4,294,967,295.
Invalid day of month	TIME condition specifies a day of the month which is outside the valid range. Select a value between one and thirty-one.
Invalid flag arguments	In a SEND action, the string to be sent contains a reference of the form ((*flag[n]*)). The value of *n* is out of range. Select either zero or one.

Invalid time value	TIME condition specifies a time which is outside the valid range for the 24–hour format.
Invalid time of day	TIME condition specifies a time which is outside the valid range for the 24–hour format.
Invalid timeout value	TIMEOUT name RESTART action specifies a value which is outside the valid range. Select a value between 0.001 and 65.535. Do not begin entry with decimal point.
Invalid trigger (lacks transitional)	Condition does not contain an event. At Layer 3 in X.25 for example, the status–only condition MORE_TO_RESEND is not combined with an event. Add an event such as PACKET_SENT to the condition.
Invalid trigger (multiple transitional–only)	Condition contains more than one event. Since no two events can come true at the same time, move one of the events to a separate CONDITIONS block.
Invalid character in constant name	Valid characters include 0–9, upper– and lower–case letters, and underscores. Name cannot begin with a number.
No closing))	Double parentheses delimit constants.
No closing]	In a SEND action, the string contains a reference to a flag or counter which includes additional information inside brackets. The closing bracket is missing.
No more errors	There are no more errors to be displayed via GO_ERR. The next time you press F8, the last error message will be displayed.
Not an object file	Attempt to access a file via the OBJECT block identifier that has a type other than LOBJ (linkable–object). Use the Compile command on the File Maintenance screen to create a linkable–object file from a source file containing standard C code.
Obsolete object file version	Attempt to access a linkable–object file via the OBJECT block identifier. Use the Compile command on the File Maintenance screen to recompile the source file, and try again.

Obsolete package loaded	A layer personality package is loaded which came from an older version of the software. The package is attempting to use facilities which are not provided in the current version of the software.
Out of buffer space	The translator has run out of memory.
Out of memory	The translator has run out of memory.
Premature end of file	1) Required softkey selections for a condition have not been made. To send a string at Layer 1, for example, you must make a BCC selection, or 2) string does not contain closing quotation mark.
Receive string cannot be longer than 32 characters	RECEIVE STRING condition at Layer 1 contains more than 32 characters. Note that constants are expanded before they are counted.
Reference to undefined constant	Attempt to use a constant that has not been defined.
Reference to undefined state	State name referenced in NEXT_STATE does not exist.
Syntax error	Program may contain punctuation errors or incomplete softkey selections. This message often accompanies other errors messages.
There is no next state	Included NEXT_STATE: NEXT, but no state follows.
Unclosed AR"C" region	1) Unequal number of opening and closing curly braces, 2) unclosed quotation marks, or 3) unclosed parentheses.
Unclosed quoted string	Insert closing quotation marks at end of string.
Undefined name	The string in a SEND action contains a constant reference which does not refer to any defined flag, counter, constant, or special inter-layer data constant.
Unexpected character in constant name	Valid characters include 0-9, upper- and lower-case letters, and underscores. Name cannot begin with a number.
Unknown object file version	Should not normally appear. May be attempting to use version of a linkable-object file which is incompatible with older versions of software.
WAIT_EOF with ENTER_STATE	A trigger condition has both ENTER_STATE and a line condition including the WAIT_EOF option. This error is a special case of the "Invalid trigger (multiple transition-only)" error.

Appendix A3: Error Messages Issued by C Compiler

Most of the following messages report errors that interrupt the compilation of a spreadsheet program and return you to the Protocol Spreadsheet. A diagnostic message about the first error will be displayed at the top (second line) of the screen. Some messages also serve as warnings. Warnings do not cause compilation to be aborted, but they are displayed on the Protocol Spreadsheet with error messages. Suppress warning messages using the following *#pragma*:

```
#pragma nowarn
```

Table A3-1
Numbered Error Messages Returned for C Coding†

001	Only integral values may be added to pointers.
002	Within constant expressions, the operand of the unary '&' operator must be an object of static storage class.
003	Only integers and pointers may be converted to pointers.
004	Attempt to create more than one instance of a task which waits for fast_event variable—task instance '(identifier)'.
005	Only numeric values may be converted to float.
006	Illegal operation on relocatable value in constant expression.
007	Illegal conversion from a structure or union type.
008	Operands of binary operator have incompatible types.
009	Illegal indirection through a non-pointer value.
010	An integral constant expression is required.
011	A scalar expression is required.
012	Bitfield values are not allowed in constant expressions.
013	Operands of '*', '/', and '%' must be numeric.

† Errors 001 - 699 are returned by the compiler. Errors 700 and higher are returned by the pre-processor.

Table A3-1 (Continued)

014	Operands of logical operators must be integers.
015	Pointer values being compared or subtracted in constant expressions must point to the same aggregate.
016	Assignment operators are invalid in constant expressions.
017	Operands of % operator may not be floating point.
018	The ++ and -- operators are invalid in constant expressions.
019	A non-relocatable constant expression is expected.
020	Relocatable quantities cannot be converted to float in constant expressions.
021	Void expressions are not permitted in constant expressions.
022	A structure or union is required for membership operators.
023	Attempt to apply a subscript to something other than an array or pointer.
024	Only integral values and pointers may be subtracted from pointers.
025	Pointers may only be subtracted from pointers of the same type.
026	Undeclared variable '(identifier)'.
027	Constant expressions may not have type 'void'.
028	Illegal implicit pointer-to-floating conversion.
029	Illegal implicit pointer-to-integer conversion.
030	Illegal implicit pointer-to-pointer conversion.
031	Illegal implicit integer-to-pointer conversion.
032	Illegal implicit floating-to-pointer conversion.
033	Illegal conversion.
034	Attempt to use an event variable in an arithmetic expression.

Table A3-1 (Continued)

035	Parameter declarations are invalid with function prototypes.
036	Functions may not be initialized.
037	Task instances may not be initialized.
038	Typedefs may not be initialized.
039	Invalid initializer on function or task.
040	Array or structure initializers must be a list of constant expressions.
041	Attempt to initialize a bitfield with a relocatable value.
042	String is too long to fit into array.
043	Too many levels of braces in initializer.
044	Too many initializers.
045	Union (identifier) undefined.
046	Struct (identifier) undefined.
047	Task has more than one entrypoint.
048	File has more than one entrypoint.
049	A function exceeds 64K bytes in size.
050	Integral type expected.
051	Incompatible types.
052	Pointers must be of the same type.
053	Integral expression expected.
054	Illegal operands of minus.
055	Arithmetic types required.

Table A3-1 (Continued)

056	Division by zero.
057	Division by zero prohibited.
058	Illegal types.
059	Arithmetic types expected.
060	Integral types expected.
061	The operands of the (symbol) operator have incompatible types.
062	Operands of incompatible type to '(symbol)' operator.
063	Branch condition must have scalar type.
064	Value of void function used.
065	Value of task invocation used.
066	Attempt to invoke an object which is not a function or task.
067	Argument must not be void.
068	Not enough arguments supplied in function call.
069	Too many arguments supplied in function call.
070	Unknown size.
071	Attempt to call bad function.
072	Extensive use of fast_event variables has caused a code segment to overflow it's 64K byte limit.
073	(Identifier) undeclared.
074	The left operand of the DOT operator must be of structure or union type.
075	The left operand of the -> operator must be either a pointer to a structure, or a pointer to a union.

Table A3-1 (Continued)

076	(Identifier) is an unknown member.
077	Illegal indirection or illegal subscript.
078	Illegal L–value.
079	Operand of prefix (symbol ++ or ––) operator must be scalar.
080	Operand of postfix (symbol ++ or ––) operator must be scalar.
081	Unary PLUS operator requires scalar operand.
082	Unary minus operator requires arithmetic operand.
083	Bitwise NOT operator requires integral operand.
084	DEFAULT not inside SWITCH.
085	Multiple DEFAULT's in switch.
086	Label '(identifier)' multiply defined.
087	BREAK outside of loop or switch.
088	CONTINUE outside of loop.
089	RETURN is invalid inside WAITFOR.
090	Void functions must not return a value.
095	Conflicting tag: (struct, union, enum, or task) (identifier) and (struct, union, enum, or task) (identifier).
091	Expression must have type 'label'.
092	Controlling expression must be integral.
093	CASE expression must be integral.
094	Duplicate CASE, value = (number).

Table A3-1 (Continued)

096	WAITFOR is invalid within another WAITFOR.
097	Attempt to wait for non-event variable.
098	Invalid L-value.
099	Attempt to modify CONST L-value.
100	Attempt to use an event value in an expression.
101	Attempt to use a label value in an expression.
102	Attempt to use a void value.
103	Arrays of functions or tasks are invalid.
104	Illegal storage class for function.
105	Illegal storage class for task instance.
106	Invalid storage class.
107	Extern variables may not be initialized within a function.
108	Function (identifier) redeclared.
109	Function (identifier) redefined.
110	(Identifier) redeclared.
111	(Identifier) redefined.
112	Typedef redefined.
113	Label '(identifier)' is undefined.
114	(1) Unknown size for (identifier).
115	(2) Unknown size for (identifier).
116	Enum (tag identifier) redeclared.

Table A3-1 (Continued)

117	Newline in character constant.
118	Newline in string constant.
119	Unknown character.
120	Unexpected character.
121	Token type missing.
122	Wrong type of declarator for function definition.
123	Parameter # (number) has no identifier.
124	(Named item) is declared in the parameter declarations, but is not listed in the parameter list of the function.
125	Extra ; in function definition.
126	Syntax error in attribute.
127	Syntax error in declarator or initializer.
128	Syntax error in initializer.
129	Syntax error in parameter definition.
130	Attempt to initialize a formal parameter.
131	Syntax error in parameter.
132	Syntax error in struct/union declaration.
133	Syntax error in type specifier.
134	Syntax error in structure member.
135	Syntax error in enumerator list.
136	Syntax error in enumerator.

Table A3-1 (Continued)

137	Syntax error in task specifier.
138	Syntax error in array subscript.
139	Syntax error in parameter list.
140	Unresolved reference (identifier).
141	Syntax error in statement.
142	Syntax error in conditional expression.
143	Syntax error in DO statement.
144	Syntax error in condition.
145	Syntax error in BREAK statement.
146	Syntax error in CONTINUE statement.
147	Syntax error in RETURN statement.
148	Syntax error in GOTO expression.
149	Syntax error in TASK list.
150	Syntax error in FOR statement.
151	Syntax error in FOR initialization.
152	Syntax error in FOR condition.
153	Syntax error in FOR increment.
154	Syntax error in SWITCH condition.
155	Syntax error in CASE expression.
156	Syntax error in compound statement.

Table A3-1 (Continued)

157	Syntax error in sizeof type name.
158	Syntax error in subscript.
159	Syntax error in function call.
160	Syntax error in expression.
161	Type clash.
162	More than one storage class.
163	Array of unknown size.
164	Cannot take size of function.
165	Structure or union of unknown size.
166	Circular definition of enumerated type.
167	Cannot take size of task.
168	Struct (identifier) redeclared.
169	Functions and tasks may not be structure members.
170	Zero-width bitfields may not be named.
171	Structure member (identifier) multiply defined.
172	Invalid negative bit-field width.
173	(Struct, union, enum, or task) (identifier) multiply defined.
174	Task (identifier) redeclared.
175	Union (identifier) redeclared.
176	Functions and tasks may not be union members.

Table A3-1 (Continued)

177	Invalid zero-bit member.
178	Union member (identifier) multiply defined.
179	No main routine supplied.
180	Arrow operator given structure, not a pointer.
181	DOT operator given a pointer to a structure, not a structure.
182	Address of array.
183	Address of function.
184	Address of register variable.
185	Address of bit-field.
186	Address of non-L-value.
187	Attempt to use a LABEL value in an expression.
188	Attempt to use an EVENT value in an expression.
189	Invalid zero or negative array dimension.
190	Only const or volatile allowed.
191	Maximum bit-field width is 16.
192	Illegal storage class for formal parameter.
193	Function parameters may not be functions.
194	Function parameters may not be tasks.
195	Bad parameter storage class.
196	Event expression required in waitfor clause.

Table A3-1 (Continued)

197	Scalars must be initialized with a single expression, optionally in braces.
198	Label undeclared.
199	Syntax error in declarator or initializer.
200	Pointers to different objects shouldn't be subtracted.
201	Duplicate formal parameters of a function.
202	External variables may not be initialized inside of a function.
203	Formal parameters of functions may not be initialized.
204	Attempt to use labels outside a function.
205	Variable (identifier) undeclared.
206	Attempt to take the value of a typedef.
207	Function's stack frame is too large.
208	Floating point has not yet been implemented.
209	Invalid conversion of relocatable quantity in constant expression.
210	Attempt to redefine the reserved name '(identifier)'.
211	CASE outside of switch statement.
212	Returned values of this size are not implemented.
213	Unrecoverable syntax error.
214	Parsing stack overflow.
215	Too many errors have been encountered during compilation.
216	Compiler aborted.

Table A3-1 (Continued)

217	Register variable '(identifier)' declared with non-scalar type.
218	Implicit declaration of function '(identifier)'.
219	Out of memory during compilation—program too big.
220	Internal software error in compiler (error number). Compilation aborted.
221	No T1 Mux Installed.
222	A waitfor statement has one or more condition clauses, none of which names an event variable. This is often caused by either misspelling an event variable, or by failing to declare an event variable.
223	The variable '(identifier)' has been declared inside of a task with the "extern" attribute, but has never been defined within that task. In this context, the keyword "extern" may only be used to forward-declare an identifier which is fully defined later in the task body.
226	Invalid or Incompatible Data Acquisition Tracks on selected playback device.
227	No ISDN Mux Installed.
230	Object file (name) is in obsolete format. Fix by recompiling it.
231	Symbol (identifier) multiply defined by object file.
232	The symbol (identifier) has been used as an event variable in one module and as a function or variable in another.
233	The symbol (identifier) has type event in one module, but has type fast_event in another.
234	Different modules have used the symbol (identifier) inconsistently as code, data, or read-only data.
235	Bad format in object file (name).
236	Cannot find object file (name).

Table A3-1 (Continued)

700	#else inside of #else clause.
701	#elif inside of #else clause.
702	Too many nested #if's.
703	Extra tokens at end of line.
704	Unexpected end of file.
705	Identifier missing from #ifdef directive.
706	Identifier missing from #ifndef directive.
707	#elif without matching #if.
708	#else without matching #if.
709	#endif without matching #if.
710	Syntax error.
711	Syntax error in constant expression.
712	String constant in constant expression.
713	Invalid character in constant expression.
714	Error in hex number.
715	End of file in char constant.
716	Newline in char constant.
717	End of file in string.
718	Newline inside string.
719	Attempt to divide by zero.
720	Unknown preprocessor command.

Table A3-1 (Continued)

721	Syntax error in formal parameters of macro.
722	Duplication of formal parameter (identifier) in macro definition.
723	No macro name given.
724	Macro redefined.
725	Syntax error in #line directive.
726	Unterminated string literal.
727	Cannot open include file (identifier).
728	Cannot find include file (identifier).
729	Identifier does not exist.
730	Syntax error in #include directive.
731	Include identifier is not defined.
732	Unterminated character constant.
733	End of file inside char constant.
734	End of file inside string.
735	End of file inside comment.
736	Argument list required.
737	Attempt to close bracket [or { with).
738	Attempt to close arg list with }.
739	Attempt to close bracket [or (with }.
740	Attempt to close arg list with].

Table A3-1 (Continued)

741	Attempt to close bracket (or { with].
742	Incomplete argument list.
743	No parameter after a # char.
744	(User-generated error message.)
745	File ends with \\.
746	Number of arguments does not match number of parameters.
749	Identifier missing from #undef.

Appendix B: Glossary of Acronyms, Abbreviations and Mnemonics

ACK	Acknowledgment
ACTLU	Activate Logical Unit (SNA)
ADR	Address
AK	ACK: Acknowledgment
ANSI	American National Standards Institute
ASCII	American Standard Code for Information Interchange, standard code for digital communications
ASYNC	Asynchronous format (indicates START and STOP bits)
AUX	Auxiliary
BAUDOT	Five bit code for data transmission using one start and one stop element; used in some teletypewriter machines
BBI	Begin Bracket Indicator (SNA)
BCC	Block Check Calculation
BCN	Beacon (SDLC)
BDLC	Burroughs Data Link Control
BEL	Bell
BERT	Bit Error Rate Testing
BIB	Backward Indicator Bit (SS#7 Layer 2)
BISYNC	Binary Synchronous Communications Protocol (IBM); also BSC
BITIM	Bit image
BL	BEL: Bell
BLI	Blink (CRT enhancement)
BM	Bit Mask
BNC	A highly reliable twist-lock connector used to carry wide-band video/digital signals: used with coaxial cable (G.703)
BOP	Bit-Oriented Protocol, e.g., SDLC
bps	Bits per second
BS	Backspace
BSC	BISYNC

C	Control (X.21 signal)
CAN	Cancel
CAS	Channel Associated Signaling (G.703)
CCITT	Consultative Committee, International Telephone and Telegraph
CCS	Clear Channel Signaling (G.703)
CCSS#7	Common Channel Signaling System #7
CD	Carrier Detect (RS-232/V.24 and V.35 signal); same as RLSD
CDI	Change Direction Indicator (SNA)
CEDI	Conditional End Bracket Indicator (SNA)
CF	Command Format (SNA)
CHAR	Character
CHDAT	Character data
CIC	Circuit Identifier Code (SS#7 Layer 3)
CLR	Clear
CN	CAN: Cancel
CONF	Confirm
const	Constant, modifier to a declaration in C language
CPM	Central Processing Module
CR	Carriage Return
CRC	Cyclic Redundancy Check
CSI	Code Selection Indicator (SNA)
CSN	Command Sequence Number (SNA)
CTS	Clear To Send (RS-232/V.24 and V.35 signal)
D	D bit (Bit 7 in first octet of packet-level X.25)
DAF	Destination Address Field (SNA)
DAT	Data Acquisition Tracks
DB-25	25-Pin D connector (standard for RS-232/V.24)
DC1	Device Control 1
DC3	Device Control 3
DCE	Data Circuit-terminating Equipment (or Data Communications Equipment), typically a modem
DCF	Data Count Field (SNA)
DDCMP	Digital Data Communications Message Protocol
DEC	Decrement
DEF	Destination Element Field (SNA)
#define	Preprocessor directive, C language
DEL	Delete
DFC	Data Flow Control (SNA)
DIAG	Diagnostic (X.25 Layer 3)
DIR	Directory
DISC	Disconnect (SDLC, LAPD, X.25 Layer 2)
DL	DLE: Data Link Escape; also Data Link layer (OSI primitive)
DLC	Data Link Control
DLE	Data Link Escape (used principally in transparent BISYNC)

DM	Disconnected Mode (SDLC, LAPD, X.25 Layer 2)
DMA	Direct Memory Access
DPC	Destination Point Code (SS#7 Layer 2)
DRAM	Dynamic Random Access Memory; one Mbyte of memory space of each MPM, dedicated to storage or receive data
DRI	Direct Response Indicator (SNA)
DSAF	Destination Subarea Address Field (SNA)
DSK	Disk
DSR	Data Set Ready (RS–232/V.24 and V.35 control lead)
DTE	Data Terminal Equipment
DTR	Data Terminal Ready (RS–232/V.24 and V.35 control lead)
DUP	Duplicate
EB	ETB, EOB: End of Transmission Block
EBCD	Extended Binary Coded Decimal
EBCDIC	Extended Binary Coded Decimal Interchange Code
EBI	End Bracket Indicator (SNA)
EC	ESC: Escape
EIA	Electronic Industries Association
#elif	Else if, preprocessor directive, C language
EM	EOM: End of Message
#endif	Preprocessor directive, C language
ENQ	Enquiry
enum	Enumeration, set of integer constants, C language
EOB	End of Transmission Block
EOF	End of Frame
EOM	End of Message
EOT	End of Transmission
EPROM	PROM containing power–up software and initialization routines
EQ	Equal
EQ	ENQ: Enquiry
ERI	Exception Response Indicator
ERN	Explicit Route Number
ESC	Escape
ET	EOT: End of Transmission
ETB	End of Transmission Block
ETX	End of Text
evar	Event variable, pre–declared identifier, ARD extension to C language
EX	ETX: End of Text
extern	External, storage class specifier, C language
FAC	Facilities (X.25 Layer 3)
FAS	Frame Alignment Signal (G.703)
FCS	Frame Check Sequence (used in BOP)
FD1	Floppy–disk Drive 1

FD2	Floppy-disk Drive 2
FDX	Full duplex (permits simultaneous data in both directions)
FEB	Front End Buffer
fevar	Fast event variable, pre-declared identifier, ARD extension to C language
FF	Form Feed
FI	Format Indicator (SNA)
FIB	Forward Indicator Bit (SS#7 Layer 2)
FID	Format Identifier (SNA)
fifo	First in, first out; memory queue on boards
FMD	Function Management Data (SNA)
FRMR	Frame Reject (SDLC, LAPD, X.25 Layer 2)
FS	Field Separator
FSN	Forward Sequence Number (SS#7 Layer 2)
GBM	Global Bus Module
GE	Greater than or equal to
GFI	Group Format Indicator (X.25 Layer 3)
goto	Jump statement, C language
GS	Group Separator
GT	Greater than
HDLC	High Level Data Link Control procedure
HDX	Half duplex (data cannot be transmitted in both directions simultaneously)
HEX	Hexadecimal number; also the hex key
HRD	Hard disk
Hz	Hertz
I	Indication (X.21 signal)
I	Information (SDLC, LAPD, X.25 Layer 2)
iAPX 286	Part number for Intel 80286 processor
IERN	Initial Explicit Route Number (SNA)
#if	Preprocessor directive, C language
#ifdef	If defined..., preprocessor directive, C language
#ifndef	If not defined..., preprocessor directive, C language
IL	Interlayer (message buffer)
INC	Increment
#include	Preprocessor directive, C language
IND	Indication
INFO	Information (SDLC, LAPD, X.25 Layer 2)
init	Initialize
int	Integer data type, C language
INT	Interrupt (X.25 Layer 3)
I/O	Input/Output
IPARS	International Passenger Airlines Reservation System

ISDN	Integrated Services Digital Network
ISO	International Standards Organization
ISOC	Isochronous
JIS	Japanese Industrial Standard
kana	Japanese syllabic alphabet
Kbits	Kilobits
Kbps	Kilobits per second
Kbyte	Kilobyte
LAF	Local Address Field (SNA)
LAPD	Link Access Procedure on the D-channel
LCG	Logical Channel Group (X.25 Layer 3)
LCN	Logical Channel Number (X.25 Layer 3)
LE	Less than or equal to
LED	Light Emitting Diode
LF	Line Feed
LI	Length Indicator (SS#7 Layer 2)
LOBJ	Linkable-object
LRC	Longitudinal Redundancy Check
LSU	Link Status Unit (SS#7 Layer 2)
LT	Less than
LTA	Link Test Acknowledge
LTM	Link Test Message
LU	Logical Unit (SNA)
M	M bit (X.25 Layer 3, Bit 4 of first octet)
macro	Macro replacement of text initiated by define pre-processor directive, C language
Mbyte	Megabyte
MOD	Modulus; maximum window size for frames or packets, 8 or 128
MPM	Main Processing Module
msec	Millisecond
MSU	Message Signal Unit (SS#7 Layer 2)
mux	Multiplexer
N	Network layer (OSI primitive)
NAK	Negative Acknowledgment
NC	Network Control (SNA)
NE	Not equal to
NETM	Network Management (SS#7 Layer 3)
NI	Network Indicator
NK	NAK: Negative Acknowledgment
NP	Network Priority (SNA)

Nr	Number (next) receive frame (SDLC, LAPD, X.25 Layer 2); also NR and N(R)
NRZI	Non-Return to Zero Inverted (used with SDLC and ASYNC modems—sometimes with clocked modems)
Ns	Number (frame) sent (SDLC, LAPD, X.25 Layer 2); also NS and N(S)
NT	Network Termination (ISDN)
NTR	Network Test Regular (SS#7 Layer 3)
NTS	Network Test Special (SS#7 Layer 3)
NU	NUL: Null
NUL	Null
OBJ	Object code
OEF	Origin Element Field (SNA)
OPC	Originating Point Code (SS#7 Layer 3)
OSAF	Origin Subarea Address Field (SNA)
OSI	Open Systems Interconnection
OUTSYNC	Out of synchronization
pad	DEL or idle line character
pal	Programmable array logic
parens	Parentheses
PCM	Peripheral Control Module
PDU	Primitive Data Unit
PERC	Percentage
P/F	Poll/Final bit used in control byte at frame level (SDLC, X.25)
PH	Physical layer (OSI primitive)
PI	Pacing Indicator (SNA)
PIU	Path Information Unit (SNA)
PLU	Primary Logical Unit (SNA)
pos	Position
Pr	Packet (next) receive sequence number (X.25 Layer 3); also PR and P(R)
#pragma	Preprocessor directive, C language
PRGM	Program
PROG TR	Program Trace Run-mode screen
PROM	Programmable Read-Only Memory
Ps	Packet send sequence number (X.25 Layer 3); also PS and P(S)
PU	Physical Unit (SNA)
Q	Q bit (Bit 8 of first octet in packet-level X.25)
QRI	Queued Response Indicator (SNA)
R	Receive (X.21 signal)
RAM	Random Access Memory
RD	Received Data (RS-232/V.24 and V.35 signal)
REG	Registration (X.25 Layer 3)

REJ	Reject (SDLC, LAPD, X.25 Layer 2)
REQ	Request
RESP	Response
Rev	Reverse
RGB	Red Green Blue (connector for color monitor)
RH	Request/Response Header (SNA)
RJ-11C	Standard for common telephone jack
RLSD	Received Line Signal Detect (RS-232/V.24 signal); same as CD: Carrier Detect
RNR	Receive Not-Ready (SDLC, LAPD, X.25 Layer 2, X.25 Layer 3)
ROM	Read-Only Memory (firmware/software storage)
RR	Receive Ready (SDLC, LAPD, X.25 Layer 2, X.25 Layer 3)
RS	Record Separator
RS-232/V.24	List of definitions for interchange circuit between data terminal equipment and data circuit termination equipment established by EIA
RS-449	EIA standard for 37-pin and 9-pin DTE-DCE interface
RTI	Response Type Indicator (SNA)
RTS	Request To Send (RS-232/V.24 and V.35 signal)
RU	Request/Response Unit (SNA)
SABM	Set Asynchronous Balanced Mode (LAPD, X.25 Layer 2)
SABME	Set Asynchronous Balanced Mode Extended (LAPD, X.25 Layer 2)
SAPI	Service Access Point Identifier (LAPD)
SB	SUB: Substitute
SC	Session Control (SNA)
SCCP	Signalling Connection Control Part (SS#7 Layer 3)
SCR	Signal Clock Receive (RS-232/V.24 and V.35 signal), used when DCE clock drives DTE
SCT	Signal Clock Transmit (RS-232/V.24 and V.35 signal), used when DCE clock drives DTE
SCTE	Signal Clock Transmit External (RS-232/V.24 and V.35 signal), used when DTE clock drives DCE
SDI	Sense Data Indicator (SNA)
SDLC	Synchronous Data Link Control (IBM)
SDU	Service Data Unit
SELECTRIC	IBM typewriter/printer code
SFO	Status Field Octet (SS#7 Layer 2)
SH	SOH: Start of Header
SI	Shift In
SI0	Sequenced Information Frame 0 (LAPD)
SI1	Sequenced Information Frame 1 (LAPD)
SIG	Signal
SIO	Service Information Octet (SS#7 Layer 3)
SLS	Signalling Link Selection (SS#7 Layer 3)
SLU	Secondary Logical Unit (SNA)

SMP	Sample
SNA	System Network Architecture (IBM)
SNAI	SNA Indicator (SNA)
SNF	Sequence Number Field (SNA)
SNRM	Set Normal Response Mode (SDLC)
SO	Shift Out
SOH	Start of Header
SRC	Source
SREJ	Selective Reject (SDLC)
SSCP	System Services Control Point (SNA)
SS#7	CCSS#7: Common Channel Signaling System #7
struct	Structure, data type which consists of a group of variables referenced under the same name, C language
STR	String
STX	Start of Text
SUB	Substitute
SX	STX: Start of Text
SY	SYN: Synchronization character
SYN	Synchronization character
SYS	System file
sys	System directory
T	Transmit (X.21 signal)
T	Transport layer (OSI primitive)
TD	Transmitted Data (RS-232/V.24 and V.35 signal)
TE	Terminal Equipment (ISDN)
TEI	Terminal Endpoint Identifier (LAPD)
TGNFI	Transmission Group Non-Fifo Indicator (SNA)
TGSI	Transmission Group Sweep Indicator (SNA)
TGSNF	Transmission Group Sequence Number Field (SNA)
TH	Transmission Header
TIM	Test Interface Module
TPF	Transmission Priority Field (SNA)
TS	Transmission Services (SNA)
TTL	Transistor-to-Transistor Logic
TUP	Telephone User Part (SS#7 Layer 3)
typedef	Type definition, data type which creates new name for existing data type, C language
UA	Unnumbered Acknowledgment (SDLC, LAPD, X.25 Layer 2)
UI	Unnumbered Information (SDLC)
UL	Underwriters' Laboratory
#undef	Undefine, preprocessor directive, C language
US	Unit Separator
usec	Microsecond

USER TR	Run-mode User Trace screen
usr	User directory
VRCWI	Virtual Route Change Window Indicator (SNA)
VRCWRI	Virtual Route Change Window Reply Indicator (SNA)
VRID	Virtual Route Identifier (SNA)
VRN	Virtual Route Number (SNA)
VRPCI	Virtual Route Pacing Count Indicator (SNA)
VRPRQ	Virtual Route Pacing Request (SNA)
VRPRS	Virtual Route Pacing Response (SNA)
VRRWI	Virtual Route Reset Window Indicator (SNA)
VRSI	Virtual Route Support Indicator (SNA)
VRSSN	Virtual Route Send Sequence Number (SNA)
VRSTI	Virtual Route Sequence and Type Indicator (SNA)
VT	Vertical Tab
X.21	CCITT recommendation governing synchronous DTE–DCE operation on public data networks
X.25	CCITT recommendation governing the packet mode link connecting the user site with a public data network
XDRAM	Extended Dynamic Random Access Memory
XEQ	Execute
XFER	Transfer
XID	Exchange Identification (SDLC)
XMIT	Transmit, transmission

Appendix C: Selectable Data Speeds

There are four clock crystals installed in the INTERVIEW. These clocks provide the bits-per-second rates listed on the following pages. (An optional crystal is also available that may be factory-configured for speeds not listed here.)

These baud rates apply to all clock and data format selections, with one exception. If you are operating in Emulate DCE mode using internal clock and the data format is anything other than Async, you may enter clock speeds 16 times higher than those listed. The following formula allows you to determine whether a higher data speed is selectable in this special case.

The frequency of each standard clock crystal is first divided by 2 to derive four values of X:

X1 = 3686400/2

X2 = 4096000/2

X3 = 4608000/2

X4 = 5376000/2

Divide the desired bps rate into each of the values of X to produce result Y.

Y1 = X1/bps

Y2 = X2/bps

Y3 = X3/bps

Y4 = X4/bps

Round each of the Y values to the nearest whole number.

Next, divide each Y value into the corresponding X value, to produce four possible speeds:

X1/Y1 = SPEED1

X2/Y2 = SPEED2

X3/Y3 = SPEED3

X4/Y4 = SPEED4

The values resulting from this calculation are the data speeds which may be selected for the desired bits-per-second rate. Select the closest speed and use this as your entry on the Line Setup Screen as the Internal Clock speed.

Table C-1
Line Setup Clock Speeds

168000.0	12923.08	6582.9	4421.05	3339.1	2679.07	2215.38
144000.0	12800.0	6545.45	4413.79	3294.12	2666.67	2210.53
128000.0	12126.3	6461.54	4363.64	3291.43	2648.3	2206.810
115200.0	12000.0	6400.0	4347.2	3282.05	2625.0	2194.3
84000.0	11636.36	6260.87	4307.69	3272.73	2618.18	2181.82
76800.0	11520.0	6227.0	4266.67	3245.1	2612.24	2173.58
72000.0	11200.0	6222.22	4235.29	3230.77	2588.8	2169.49
64000.0	11076.92	6095.24	4200.0	3200.0	2584.62	2153.85
57600.0	10971.4	6063.16	4189.1	3169.81	2571.43	2149.25
56000.0	10666.67	6000.0	4129.03	3156.2	2560.0	2133.33
48000.0	10500.0	5907.7	4114.29	3130.43	2545.45	2126.58
46080.0	10472.73	5818.18	4097.56	3121.95	2526.32	2117.65
42666.67	10285.71	5793.10	4042.1	3113.51	2509.80	2113.8
42000.0	10017.4	5760.0	4000.0	3111.11	2507.46	2100.0
38400.0	9882.35	5619.5	3972.41	3072.0	2504.35	2098.36
36000.0	9846.15	5600.0	3906.98	3063.83	2482.76	2094.55
33600.0	9600.0	5565.22	3905.1	3054.55	2477.4	2086.96
32914.3	9333.33	5538.46	3891.89	3047.62	2470.59	2075.7
32000.0	9216.0	5485.71	3878.79	3031.58	2461.54	2074.07
28800.0	9142.86	5419.35	3840.0	3000.0	2451.06	2064.52
28000.0	9000.0	5358.1	3818.18	2992.2	2440.68	2057.14
25600.0	8861.54	5333.33	3789.47	2976.74	2434.78	2048.78
24000.0	8842.11	5250.0	3777.0	2953.85	2425.3	2038.9
23040.0	8533.33	5236.36	3764.71	2947.37	2415.09	2031.75
21333.33	8470.59	5142.86	3733.33	2938.78	2400.0	2028.17
21000.0	8400.0	5120.0	3716.13	2916.5	2375.3	2024.010
20945.5	8228.57	5090.91	3692.31	2909.09	2370.37	2021.05
20571.43	8000.0	5008.610	3657.14	2896.55	2366.110	2003.5
19200.0	7944.8	4965.52	3652.17	2880.0	2360.66	2000.0
18666.67	7680.0	4941.18	3600.0	2847.46	2351.02	1986.21
18285.71	7636.36	4923.08	3574.47	2844.44	2333.33	1976.47
18000.0	7578.95	4902.1	3555.56	2823.53	2327.27	1972.60
17723.1	7529.41	4800.0	3544.6	2809.76	2322.58	1969.23
16800.0	7432.3	4740.74	3512.110	2800.0	2304.0	1953.49
16457.14	7304.35	4702.0	3500.0	2782.61	2301.37	1952.54
16000.0	7200.0	4666.67	3490.91	2775.9	2285.71	1945.95
15360.0	7111.11	4645.16	3459.46	2769.23	2281.2	1939.39
15272.73	7000.0	4608.0	3438.8	2754.010	2270.27	1931.03
14400.0	6981.8	4571.43	3428.57	2742.86	2258.82	1920.0
14222.22	6857.14	4540.54	3388.24	2723.40	2250.0	1910.45
14000.0	6776.47	4517.6	3368.42	2716.98	2245.61	1909.09
13552.9	6736.84	4500.0	3360.0	2710.6	2240.0	1904.1
13090.91	6720.0	4430.77	3348.84	2709.68	2236.9	1894.74

Table C-1 (continued)

1888.52	1655.17	1473.68	1331.8	1210.08	1112.58	1028.57
1887.64	1647.06	1471.26	1324.14	1208.63	1107.69	1024.39
1882.35	1645.71	1469.39	1322.83	1207.55	1105.26	1024.0
1873.2	1641.03	1467.5	1321.10	1206.3	1103.45	1021.28
1870.13	1636.36	1460.87	1319.59	1200.0	1102.4	1019.47
1866.67	1634.0	1458.23	1316.6	1196.26	1099.24	1018.18
1858.06	1631.07	1454.55	1312.50	1193.8	1098.04	1015.87
1855.07	1622.54	1449.1	1309.09	1191.49	1097.14	1015.0
1846.15	1620.25	1448.28	1306.12	1190.08	1094.02	1014.08
1843.2	1617.98	1440.0	1302.33	1187.63	1091.9	1012.05
1828.57	1615.38	1438.20	1301.7	1185.19	1090.91	1010.53
1826.09	1611.2	1435.810	1297.210	1183.010	1086.79	1007.87
1822.78	1600.0	1431.1	1294.38	1181.5	1084.75	1006.99
1814.2	1589.0	1425.74	1292.93	1180.33	1083.87	1006.1
1806.45	1584.91	1423.73	1292.31	1175.51	1082.71	1005.99
1802.82	1582.42	1422.22	1287.2	1174.83	1081.7	1001.74
1800.0	1580.25	1413.5	1285.71	1174.31	1076.92	1000.0
1787.23	1578.08	1411.76	1282.44	1170.73	1076.64	997.4
1786.0	1570.09	1406.59	1280.0	1169.5	1075.63	994.08
1777.78	1567.3	1404.88	1274.34	1166.67	1074.63	993.10
1772.31	1565.22	1400.0	1272.9	1163.64	1071.6	992.25
1768.42	1560.98	1398.06	1272.73	1161.29	1070.06	988.8
1758.8	1556.76	1396.0	1267.33	1158.62	1066.67	988.24
1756.010	1555.56	1391.30	1265.93	1157.8	1063.29	986.30
1753.42	1548.39	1388.43	1263.16	1153.15	1061.8	984.62
1750.0	1546.3	1388.0	1259.0	1152.0	1058.82	982.46
1745.45	1542.17	1387.95	1254.90	1150.68	1057.85	980.4
1734.94	1541.28	1384.62	1253.73	1146.3	1056.88	979.59
1732.3	1536.0	1379.6	1252.17	1142.86	1056.60	977.010
1731.96	1531.91	1377.05	1245.4	1140.59	1052.1	976.74
1729.73	1527.27	1376.34	1244.44	1135.14	1051.09	976.27
1719.40	1525.8	1371.43	1242.72	1135.0	1050.0	972.97
1714.29	1523.81	1365.85	1241.38	1133.86	1049.18	972.2
1706.67	1515.79	1363.3	1238.71	1132.74	1047.27	971.010
1696.97	1513.51	1361.70	1235.29	1129.41	1043.48	969.610
1694.12	1505.88	1358.49	1232.1	1127.52	1042.5	968.07
1684.21	1500.0	1355.29	1230.77	1125.0	1040.65	966.44
1681.8	1496.10	1354.84	1226.28	1123.9	1037.84	965.52
1680.0	1488.37	1347.37	1225.53	1122.81	1037.04	964.0
1674.42	1486.73	1345.79	1220.34	1120.0	1035.97	962.41
1669.57	1486.5	1344.0	1219.05	1118.45	1033.2	960.0
1663.37	1484.54	1339.53	1217.39	1116.28	1032.26	956.0
1662.34	1476.92	1333.33	1212.63	1113.04	1030.67	955.22

Table C-1 (continued)

954.55	893.02	835.82	786.89	743.23	702.93	669.32
953.64	889.6	834.78	786.3	742.27	702.44	667.8
952.07	888.89	832.37	785.28	740.8	700.3	666.67
949.15	886.15	831.8	785.05	740.09	700.0	665.9
948.15	884.21	831.68	783.67	739.88	699.45	665.810
947.37	883.44	831.17	782.61	738.46	699.03	664.03
944.26	882.76	828.78	781.310	736.84	698.18	663.59
943.82	879.58	827.59	781.0	736.1	697.010	663.21
941.18	879.39	825.81	780.49	735.63	696.1	662.07
940.4	878.05	823.53	778.38	734.69	695.65	661.42
938.55	876.71	822.86	777.78	733.76	694.21	660.55
936.59	876.0	820.51	775.76	733.62	694.0	660.2
935.06	875.0	819.9	774.19	731.43	693.98	659.79
934.31	872.73	819.51	773.15	730.96	692.31	658.82
933.33	870.75	818.18	771.08	730.43	691.89	658.29
932.8	870.47	817.02	770.64	729.11	691.36	657.53
929.03	869.4	815.53	770.05	727.27	689.82	656.41
928.18	867.47	815.29	768.0	726.8	688.910	656.25
927.54	866.17	814.1	767.12	724.53	688.52	654.55
925.3	865.98	813.56	766.47	724.14	688.17	653.610
923.08	864.86	811.59	765.96	723.62	687.8	653.06
921.60	862.9	811.27	765.4	723.16	685.71	652.7
920.86	862.28	810.13	763.64	722.3	684.49	651.58
918.03	861.54	808.99	762.91	721.03	683.7	651.16
917.9	859.70	808.4	761.90	720.0	682.93	650.85
917.110	859.06	807.69	760.4	719.10	682.46	649.75
914.29	857.14	805.59	760.18	717.95	681.66	649.0
913.04	856.5	805.03	757.89	717.8	680.85	648.65
911.39	853.33	804.47	757.310	716.42	680.16	647.19
910.7	852.79	803.83	756.76	715.53	679.6	646.46
908.11	852.07	802.8	755.4	715.08	679.25	646.15
907.80	850.2	800.0	753.93	714.89	677.65	645.74
907.09	848.48	797.2	753.36	713.3	677.42	645.4
905.66	847.68	796.21	752.94	712.87	677.25	643.68
903.5	847.06	795.58	750.5	711.86	676.06	643.58
903.23	844.22	795.03	750.0	711.11	675.7	643.22
901.41	844.0	794.48	748.54	709.36	674.610	642.86
900.0	842.11	792.45	748.05	708.86	673.68	641.8
898.310	840.88	791.8	746.67	707.18	672.810	641.22
896.5	840.0	791.21	746.11	706.75	672.0	640.0
895.10	837.8	790.12	745.6	705.88	671.7	638.78
894.41	837.21	789.04	744.19	704.6	670.16	638.2
893.62	836.60	788.73	743.36	703.210	669.77	637.17

Table C-1 (continued)

636.82	608.610	581.82	559.22	538.32	517.8	498.52
636.46	607.9	581.31	558.95	537.82	516.92	498.27
636.36	607.59	580.65	558.14	537.31	516.59	498.05
634.7	606.64	580.4	557.9	537.1	516.13	497.6
634.36	606.410	579.31	556.52	536.74	515.4	497.04
633.96	606.32	579.19	556.29	535.81	515.34	496.55
633.66	605.04	578.89	555.98	535.56	514.29	496.6
633.0	604.7	578.31	555.2	535.32	514.06	496.12
632.97	604.32	577.4	554.46	535.03	513.76	495.58
631.58	603.77	577.32	554.11	534.6	513.1	495.5
631.2	603.14	576.58	553.85	533.33	512.46	494.85
630.54	602.51	576.0	552.63	532.1	512.110	494.42
629.51	602.15	575.34	552.5	531.65	512.0	494.21
629.21	601.6	574.6	551.72	531.37	510.9	494.12
628.82	600.94	573.99	551.2	531.12	510.64	493.4
627.8	600.0	573.71	551.110	530.88	509.96	493.15
627.45	598.4	573.38	550.82	529.97	509.73	492.67
626.87	598.13	573.13	549.9	529.7	509.09	492.31
626.09	597.86	571.7	549.62	529.41	508.83	491.47
624.54	597.51	571.43	549.36	528.93	508.6	491.3
624.39	596.89	570.3	549.02	528.44	507.94	491.23
623.38	595.74	570.210	548.57	528.30	507.55	490.42
622.70	595.35	569.49	547.53	527.47	507.49	490.21
622.22	595.04	569.17	547.3	527.2	507.04	489.710
621.36	593.81	568.89	547.23	526.75	506.4	489.2
621.0	593.64	567.57	547.01	526.65	506.02	488.55
620.69	592.59	567.49	546.0	526.03	505.93	488.37
619.93	592.3	566.93	545.97	525.55	505.26	488.14
619.35	591.55	566.37	545.45	525.0	504.50	486.96
617.7	590.77	566.1	544.68	524.8	504.2	486.69
618.36	590.16	565.66	543.69	524.59	503.94	486.49
618.03	589.86	564.71	543.4	523.64	503.410	486.08
617.65	589.47	563.88	543.310	523.36	503.1	485.55
616.04	589.3	563.76	542.37	522.45	503.06	484.85
615.38	587.76	563.3	542.1	521.74	502.99	484.15
614.4	587.41	562.50	541.94	521.27	502.0	484.03
613.14	587.16	562.0	541.35	520.33	501.96	483.22
612.77	586.3	561.95	540.85	520.12	501.74	483.02
612.44	585.37	561.87	540.19	519.86	501.49	482.76
611.1	584.77	561.40	540.08	518.92	500.87	482.01
610.91	584.47	560.31	539.6	518.52	500.0	481.61
610.17	583.33	560.6	539.33	518.22	499.8	481.38
609.52	582.910	560.0	538.46	517.99	498.70	481.20

Table C-1 (continued)

481.0	464.09	449.12	435.23	422.29	407.93	393.85
480.0	463.77	449.110	435.05	422.11	407.77	393.44
479.40	463.6	448.510	434.72	422.0	407.64	393.17
479.0	463.02	448.25	434.11	421.98	407.07	392.64
478.63	462.81	448.2	433.9	421.2	406.78	392.52
478.41	462.7	448.0	433.810	421.05	406.35	392.37
478.01	462.65	447.55	433.73	420.44	405.710	391.84
477.61	462.09	447.4	433.08	420.0	405.63	391.61
477.27	461.7	447.20	432.99	419.83	405.06	391.44
477.0	461.54	446.81	432.43	419.67	404.82	391.30
476.82	460.80	446.51	432.3	418.95	404.49	390.610
476.03	460.43	445.99	431.88	418.91	404.21	390.51
475.92	460.27	445.82	431.46	418.9	403.85	390.24
475.84	460.06	445.62	431.14	418.60	403.79	389.79
475.25	459.9	444.79	430.98	418.30	403.36	389.19
475.1	459.02	444.44	430.77	418.1	402.88	389.06
474.58	458.96	443.9	430.7	417.91	402.710	388.89
474.07	458.78	443.27	429.9	417.39	402.52	388.14
473.68	458.510	443.08	429.85	416.94	402.23	387.99
473.24	458.1	442.91	429.67	416.87	401.91	387.88
473.1	457.77	442.2	429.53	416.18	401.39	387.010
472.32	457.14	442.11	429.1	415.88	401.25	386.71
472.13	456.52	441.72	428.57	415.84	401.11	386.58
471.91	456.2	441.38	428.3	415.58	400.95	386.21
471.2	455.610	440.94	428.25	414.99	400.0	386.06
470.59	455.52	440.5	428.09	414.81	399.05	385.54
470.20	455.34	440.37	427.48	414.39	398.89	385.32
469.27	455.28	439.86	427.210	414.24	398.75	385.28
469.2	454.4	439.79	426.67	413.79	398.62	385.03
469.06	454.26	439.69	426.310	412.90	398.10	384.44
468.86	454.05	439.02	426.04	412.78	397.79	384.38
468.29	453.90	438.9	425.9	412.61	397.52	384.0
467.97	453.54	438.64	425.32	411.76	397.24	383.56
467.53	452.83	438.36	425.25	411.58	397.16	383.23
467.15	452.7	438.02	425.09	411.43	396.69	382.98
466.67	452.210	438.0	424.78	410.76	396.28	382.72
466.4	451.76	437.69	424.3	410.26	396.23	382.69
466.310	451.61	437.50	424.24	409.96	395.88	382.09
466.02	451.41	437.2	423.84	409.76	395.60	381.96
465.5	450.9	436.86	423.53	409.09	395.29	381.82
465.45	450.70	436.36	423.17	408.95	395.06	381.46
465.37	450.40	435.5	422.8	408.76	394.52	380.95
464.52	450.0	435.37	422.44	408.51	394.37	380.110

Table C-1 (continued)

380.09	367.61	355.93	344.91	334.20	324.32	315.110
379.95	367.35	355.56	344.410	334.11	324.05	315.010
379.82	366.88	355.18	344.26	333.91	323.610	314.75
379.23	366.81	354.68	344.09	333.33	323.510	314.61
378.95	366.76	354.57	343.88	332.95	323.23	314.410
378.610	366.41	354.46	343.68	332.67	323.08	314.02
378.38	366.01	354.43	343.56	332.56	322.87	313.810
377.95	365.71	353.81	343.16	332.47	322.69	313.73
377.70	365.48	353.68	342.86	332.02	322.46	313.43
377.58	365.22	353.59	342.25	331.99	322.42	313.04
377.53	364.67	353.37	342.16	331.710	322.15	312.96
376.96	364.56	352.94	342.04	331.61	321.84	312.85
376.68	364.43	352.62	341.84	331.36	321.79	312.36
376.47	363.64	352.29	341.46	331.03	321.61	312.27
375.98	363.41	352.20	341.33	330.75	321.43	312.110
375.84	362.85	352.08	341.23	330.71	321.22	311.69
375.37	362.72	351.65	340.83	330.28	320.89	311.44
375.24	362.61	351.46	340.77	330.09	320.80	311.35
375.0	362.26	351.22	340.43	330.06	320.71	311.11
374.27	362.07	350.73	340.08	329.810	320.61	311.02
374.16	361.81	350.68	339.82	329.52	320.0	310.68
374.03	361.58	350.36	339.62	329.41	319.39	310.54
373.33	361.29	350.15	339.52	329.14	319.29	310.51
373.18	361.13	350.0	339.39	329.05	319.20	310.34
373.06	360.90	349.73	338.82	328.77	319.11	309.96
372.82	360.56	349.51	338.71	328.21	318.79	309.93
372.51	360.52	349.27	338.62	328.13	318.58	309.68
372.09	360.0	349.09	338.03	328.02	318.41	309.39
371.68	359.74	348.77	337.83	327.49	318.23	309.18
371.61	359.55	348.67	337.73	327.37	318.18	309.01
371.13	359.10	348.55	337.35	327.27	317.88	308.85
371.01	358.97	348.04	337.24	326.85	317.62	308.82
370.86	358.88	347.83	336.84	326.53	317.58	308.43
370.42	358.54	347.11	336.67	326.35	317.36	308.35
370.18	358.21	346.99	336.45	326.21	317.18	308.26
370.04	357.76	346.88	336.0	325.79	316.98	308.02
369.94	357.54	346.39	335.96	325.610	316.83	307.69
369.23	357.45	346.15	335.86	325.58	316.48	307.20
368.88	357.32	345.95	335.66	325.42	316.38	307.13
368.42	356.69	345.68	335.33	325.06	316.05	307.04
368.29	356.66	345.32	335.08	324.95	315.79	306.95
368.05	356.55	345.01	334.88	324.87	315.62	306.57
367.82	356.44	344.97	334.66	324.51	315.27	306.38

Table C-1 (continued)

306.22	303.710	301.18	150.0	100.0	45.0	16.0
306.01	303.32	301.08	144.0	96.0	42.0	15.0
305.73	303.25	300.0	140.0	90.0	40.0	14.0
305.57	303.16	288.0	134.5	84.0	36.0	12.0
305.49	302.70	280.0	128.0	80.0	35.0	10.0
305.45	302.60	256.0	125.0	75.0	32.0	9.0
305.08	302.52	210.0	120.0	72.0	30.0	8.0
304.90	302.36	200.0	250.0	70.0	28.0	7.0
304.76	302.16	192.0	240.0	64.0	25.0	6.0
304.44	301.89	180.0	225.0	60.0	24.0	5.0
304.35	301.62	175.0	224.0	56.0	21.0	4.0
304.04	301.57	168.0	112.0	50.0	20.0	3.0
303.96	301.26	160.0	105.0	48.0	18.0	2.0

Appendix D: Code Charts

Appendix D1: Keyboard-to-Hex Translation

Table D1-1
Keyboard-to-EBCDIC

	UNSHIFTED		SHIFTED		CONTROL	
KEY	CHAR[1]	HEX[2]	CHAR[1]	HEX[2]	CHAR[1]	HEX[2]
A	a	81	A	C1	SOH	01
B	b	82	B	C2	STX	02
C	c	83	C	C3	ETX	03
D	d	84	D	C4	EOT	37
E	e	85	E	C5	ENQ	2D
F	f	86	F	C6	ACK	2E
G	g	87	G	C7	BEL	2F
H	h	88	H	C8	BS	16
I	i	89	I	C9	HT	05
J	j	91	J	D1	LF	25
K	k	92	K	D2	VT	0B
L	l	93	L	D3	FF	0C
M	m	94	M	D4	CR	0D
N	n	95	N	D5	SO	0E
O	o	96	O	D6	SI	0F
P	p	97	P	D7	DLE	10
Q	q	98	Q	D8	DC1	11
R	r	99	R	D9	DC2	12
S	s	A2	S	E2	DC3	13
T	t	A3	T	E3	DC4	3C
U	u	A4	U	E4	NAK	3D
V	v	A5	V	E5	SYN	32
W	w	A6	W	E6	ETB	26
X	x	A7	X	E7	CAN	18
Y	y	A8	Y	E8	EM	19
Z	z	A9	Z	E9	SUB	3F
0	0	F0	)	5D	*Same as Unshifted*	
1	1	F1	!	5A		
2	2	F2	@	7C		
3	3	F3	#	7B		
4	4	F4	$	5B		
5	5	F5	%	6C		
6	6	F6	–	–		
7	7	F7	&	50		
8	8	F8	*	5C		
9	9	F9	(	4D		
dash	dash	60	underline	6D		
=	=	7E	+	4E	~	A1
\[3]	\	E0	¦	6A	'	79
[	–	–	{	C0	DEL	07
]	–	–	}	D0	ESC	27
;	;	5E	:	7A	NUL	00
'	'	7D	"	7F	GS	1D
,	,	6B	<	4C	RS	35
.	.	4B	>	6E	US	1F
/	/	61	?	6F	FS	22
Space	Space	40	Space	40	Space	40

Untranslatable characters ("–" in the above table) that are entered in transmit strings will be replaced by NULL (hex 00) during transmission.

[1]CHAR displayed in Run mode

[2]HEX byte trapped/transmitted

[3]Enter the hex value for the \ character.

Table D1-2
Keyboard-to-ASCII

	UNSHIFTED		SHIFTED		CONTROL	
KEY	CHAR[1]	HEX[2]	CHAR[1]	HEX[2]	CHAR[1]	HEX[2]
A	a	61	A	41	SOH	01
B	b	62	B	42	STX	02
C	c	63	C	43	ETX	03
D	d	64	D	44	EOT	04
E	e	65	E	45	ENQ	05
F	f	66	F	46	ACK	06
G	g	67	G	47	BEL	07
H	h	68	H	48	BS	08
I	i	69	I	49	HT	09
J	j	6A	J	4A	LF	0A
K	k	6B	K	4B	VT	0B
L	l	6C	L	4C	FF	0C
M	m	6D	M	4D	CR	0D
N	n	6E	N	4E	SO	0E
O	o	6F	O	4F	SI	0F
P	p	70	P	50	DLE	10
Q	q	71	Q	51	DC1	11
R	r	72	R	52	DC2	12
S	s	73	S	53	DC3	13
T	t	74	T	54	DC4	14
U	u	75	U	55	NAK	15
V	v	76	V	56	SYN	16
W	w	77	W	57	ETB	17
X	x	78	X	58	CAN	18
Y	y	79	Y	59	EM	19
Z	z	7A	Z	5A	SUB	1A
0	0	30	)	29		
1	1	31	!	21		
2	2	32	@	40	*Same*	
3	3	33	#	23		
4	4	34	$	24		
5	5	35	%	25	*as*	
6	6	36	^	5E		
7	7	37	&	26		
8	8	38	*	2A	*Unshifted*	
9	9	39	(	28		
dash	dash	2D	underline	5F		
=	=	3D	+	2B	~	7E
\[3]	\	5C	¦	7C	`	60
[	[	5B	{	7B	DEL	7F
]	]	5D	}	7D	ESC	1B
;	;	3B	:	3A	NUL	00
'	'	27	"	22	GS	1D
,	,	2C	<	3C	RS	1E
.	.	2E	>	3E	US	1F
/	/	2F	?	3F	FS	1C
Space	Space	20	Space	20	Space	20

[1]CHAR displayed in Run mode
[2]HEX byte trapped/transmitted (space parity)
[3]Enter the hex value for the \ character.

Table D1-3
Keyboard-to-EBCD

	UNSHIFTED			SHIFTED			CONTROL	
KEY	LOWER(\)[1]	UPPER(^)[2]	HEX[3]	LOWER(\)[1]	UPPER(^)[2]	HEX[3]	CHAR[4]	HEX[3]
A	a	A	23	a	A	23	SOH	3E
B	b	B	13	b	B	13	–	–
C	c	C	73	c	C	73	–	–
D	d	D	0B	d	D	0B	EOT	7C
E	e	E	6B	e	E	6B	–	–
F	f	F	5B	f	F	5B	–	–
G	g	G	3B	g	G	3B	–	–
H	h	H	07	h	H	07	BS	5D
I	i	I	67	i	I	67	HT	2F
J	j	J	61	j	J	61	LF	6E
K	k	K	51	k	K	51	–	–
L	l	L	31	l	L	31	–	–
M	m	M	49	m	M	49	CR	6D
N	n	N	29	n	N	29	–	–
O	o	O	19	o	O	19	–	–
P	p	P	79	p	P	79	–	–
Q	q	Q	45	q	Q	45	–	–
R	r	R	25	r	R	25	DC2	4C
S	s	S	52	s	S	52	–	–
T	t	T	32	t	T	32	DC4	4F
U	u	U	4A	u	U	4A	–	–
V	v	V	2A	v	V	2A	SYN	3D
W	w	W	1A	w	W	1A	ETB	5E
X	x	X	7A	x	X	7A	–	–
Y	y	Y	46	y	Y	46	–	–
Z	z	Z	26	z	Z	26	–	–
0	0	)	54	0	>	54		
1	1	=	20	$	!	75		
2	2	<	10	@	02	02	*Same*	
3	3	;	70	#	"	34		
4	4	:	08	$	!	75		
5	5	%	68	5	%	68	*as*	
6	6	'	58	^	^	1C		
7	7	>	38	&	+	43		
8	8	*	04	8	*	04	*Unshifted*	
9	9	(	64	9	(	64		
dāsh	dāsh	dāsh	01	–	–	–		
=	1	=	20	&	+	43	–	–
\[5]	\	\	1F	–	–	–	–	–
[	–	–	–	–	–	–	DEL	7F
]	–	–	–	–	–	–	–	–
;	3	;	70	4	:	08	–	–
'	6	'	58	#	"	34	–	–
,	,	76	76	2	<	10	RS	2C
.	.	37	37	7	>	38	–	–
/	/	?	62	/	?	62	–	–
Space	Space	Space	40	Space	Space	40	–	–

Untranslatable characters ("–" in the above table) that are entered n transmit strings will be replaced by SPACE (hex 40) during transmission.

[1]CHAR displayed in Run mode if latest case-control character was lower

[2]CHAR displayed in Run mode if latest case-control character was upper

[3]HEX byte trapped/transmitted (odd parity)

[4]CHAR displayed in Run mode

[5]Enter the hex value for the \ character.

Table D1-4
Keyboard-to-XS-3 (SYN=35; EOM=55)

KEY	UNSHIFTED CHAR[1]	UNSHIFTED HEX[2]	SHIFTED CHAR[1]	SHIFTED HEX[2]	CONTROL CHAR[1]	CONTROL HEX[2]
A	A	54	A	54	–	–
B	B	15	B	15	–	–
C	C	16	C	16	–	–
D	D	57	D	57	–	–
E	E	58	E	58	–	–
F	F	19	F	19	–	–
G	G	1A	G	1A	–	–
H	H	5B	H	5B	–	–
I	I	1C	I	1C	–	–
J	J	64	J	64	–	–
K	K	25	K	25	–	–
L	L	26	L	26	–	–
M	M	67	M	67	–	–
N	N	68	N	68	–	–
O	O	29	O	29	–	–
P	P	2A	P	2A	–	–
Q	Q	6B	Q	6B	–	–
R	R	2C	R	2C	–	–
S	S	75	S	75	–	–
T	T	76	T	76	–	–
U	U	37	U	37	–	–
V	V	38	V	38	–	–
W	W	79	W	79	–	–
X	X	7A	V	7A	–	–
Y	Y	3B	Y	3B	–	–
Z	Z	7C	Z	7C	–	–
0	0	43	)	3D		
1	1	04	!	23		
2	2	45	@	20	Same	
3	3	46	#	1F		
4	4	07	$	62		
5	5	08	%	6D	as	
6	6	49	^	2F		
7	7	4A	&	73		
8	8	0B	*	61	Unshifted	
9	9	4C	(	31		
dash	dash	02	underline	70		
=	=	5D	+	10	–	–
\[3]	\	0D	–	–	–	–
[	[	4F	–	–	–	–
]	]	01	–	–	–	–
;	;	0E	:	51	–	–
'	'	6E	"	7F	–	–
,	,	32	<	5E	–	–
.	.	52	>	3E	–	–
/	/	34	?	13	–	–
Space	Space	40	Space	40	Space	40

Untranslatable characters ("–" in the above tables) that are entered in transmit strings will be replaced by NULL (hex 00) during transmission.
[1]CHAR displayed in Run mode
[2]HEX byte trapped/transmitted (odd parity)
[3]Enter the hex value for the \ character.

Table D1-5
Keyboard-to-IPARS

KEY	UNSHIFTED CHAR[1]	UNSHIFTED HEX[2]	SHIFTED CHAR[1]	SHIFTED HEX[2]	CONTROL CHAR[1]	CONTROL HEX[2]
A	A	31	A	31	-	-
B	B	32	B	32	-	-
C	C	33	C	33	-	-
D	D	34	D	34	-	-
E	E	35	E	35	-	-
F	F	36	F	36	-	-
G	G	37	G	37	-	-
H	H	38	H	38	-	-
I	I	39	I	39	-	-
J	J	21	J	21	-	-
K	K	22	K	22	-	-
L	L	23	L	23	-	-
M	M	24	M	24	CR	0C
N	N	25	N	25	-	-
O	O	26	O	26	-	-
P	P	27	P	27	-	-
Q	Q	28	Q	28	-	-
R	R	29	R	29	-	-
S	S	12	S	12	-	-
T	T	13	T	13	-	-
U	U	14	U	14	-	-
V	V	15	V	15	-	-
W	W	16	W	16	-	-
X	X	17	X	17	-	-
Y	Y	18	Y	18	-	-
Z	Z	19	Z	19	-	-
0	0	0A	)	2E		
1	1	01	-	-		
2	2	02	@	20	*Same*	
3	3	03	#	1B		
4	4	04	$	30		
5	5	05	%	3C	*as*	
6	6	06	-	-		
7	7	07	-	-		
8	8	08	*	0B	*Unshifted*	
9	9	09	(	2F		
dāsh	dāsh	1A	-	-		
=	=	0E	+	2C	-	-
\	-	-	-	-	-	-
[	[	1E	-	-	-	-
]	-	-	-	-	-	-
;	-	-	:	2A	-	-
'	-	-	"	3E	-	-
,	,	1F	<	2B	-	-
.	.	3B	-	-	-	-
/	-	-	?	3A	-	-
Space	Space	1C	Space	1C	Space	1C

[1]CHAR displayed in Run mode
[2]HEX byte trapped/transmitted

Table D1-6
Keyboard-to-REVERSE EBCD

	UNSHIFTED			SHIFTED			CONTROL		
KEY	LOWER(\)[1]	UPPER(^)[2]	HEX[3]	LOWER(\)[1]	UPPER(^)[2]	HEX[3]	LOWER(\)[1]	UPPER(^)[2]	HEX[3]
A	a	A	31	a	A	31	–	–	–
B	b	B	32	b	B	32	–	–	–
C	c	C	73	c	C	73	–	–	–
D	d	D	34	d	D	34	EOT	EOT	4F
E	e	E	75	e	E	75	–	–	–
F	f	F	76	f	F	76	–	–	–
G	g	G	37	g	G	37	–	–	–
H	h	H	38	h	H	38	BS	BS	6E
I	i	I	79	i	I	79	HT	HT	3D
J	j	J	61	j	J	61	LF	LF	5D
K	k	K	62	k	K	62	–	–	–
L	l	L	23	l	L	23	FF	FF	0D
M	m	M	64	m	M	64	CR	CR	60
N	n	N	25	n	N	25	–	–	–
O	o	O	26	o	O	26	–	–	–
P	p	P	67	p	P	67	–	–	–
Q	q	Q	68	q	Q	68	–	–	–
R	r	R	29	r	R	29	–	–	–
S	s	S	52	s	S	52	–	–	–
T	t	T	13	t	T	13	–	–	–
U	u	U	54	u	U	54	–	–	–
V	v	V	15	v	V	15	–	–	–
W	w	W	16	w	W	16	ETB	ETB	5E
X	x	X	57	x	X	57	–	–	–
Y	y	Y	58	y	Y	58	–	–	–
Z	z	Z	19	z	Z	19	–	–	–
0	0	)	4A	0	)	4A	*Same as Unshifted*		
1	1	=	01	$	!	6B			
2	2	<	02	@	10	10			
3	3	;	43	#	"	0B			
4	4	:	04	$	!	6B			
5	5	%	45	5	%	45			
6	6	'	46	^	^	0E			
7	7	>	07	&	+	70			
8	8	*	08	8	*	08			
9	9	(	49	9	(	49			
dāsh	dāsh	dāsh	20	–	–	–			
=	1	=	01	&	+	70	-	'	7C
\[4]	\	\	3E	–	–	–	~	'	7C
[	[	]	4C	}	{	1C	DEL	DEL	7F
]	[	]	4C	}	{	1C	–	–	–
;	3	;	43	4	:	04	–	–	–
'	6	'	46	#	"	0B	–	–	–
,	,	5B	5B	2	<	02	–	–	–
.	.	3B	3B	7	>	07	–	–	–
/	/	?	51	/	?	51	–	–	–
Space	Space	Space	40	Space	Space	40	Space	Space	40

Untranslatable characters ("–" in the above table) that are entered in transmit strings will be replaced by SPACE (hex 40) during transmission.

[1]CHAR displayed in Run mode if latest case-control character was lower

[2]CHAR displayed in Run mode if latest case-control character was upper

[3]HEX byte trapped/transmitted (odd parity)

[4]Enter the hex value for the \ character.

Table D1-7
Keyboard-to-SELECTRIC

	UNSHIFTED			SHIFTED			CONTROL	
KEY	LOWER(\)[1]	UPPER(^)[2]	HEX[3]	LOWER(\)[1]	UPPER(I^)[2]	HEX[3]	CHAR[4]	HEX[3]
A	a	A	79	a	A	79	–	–
B	b	B	76	b	B	76	–	–
C	c	C	7A	c	C	7A	–	–
D	d	D	2A	d	D	2A	EOT	7C
E	e	E	4A	e	E	4A	–	–
F	f	F	73	f	F	73	–	–
G	g	G	23	g	G	23	–	–
H	h	H	26	h	H	26	BS	5D
I	i	I	19	i	I	19	HT	2F
J	j	J	43	j	J	43	LF	6E
K	k	K	1A	k	K	1A	–	–
L	l	L	46	l	L	46	–	–
M	m	M	61	m	M	61	CR	6D
N	n	N	52	n	N	52	–	–
O	o	O	45	o	O	45	–	–
P	p	P	0B	p	P	0B	–	–
Q	q	Q	5B	q	Q	5B	–	–
R	r	R	29	r	R	29	–	–
S	s	S	25	s	S	25	–	–
T	t	T	02	t	T	02	–	–
U	u	U	32	u	U	32	–	–
V	v	V	31	v	V	31	–	–
W	w	W	75	w	W	75	ETB	5E
X	x	X	62	x	X	62	–	–
Y	y	Y	67	y	Y	67	–	–
Z	z	Z	54	z	Z	54	–	–
0	0	)	64	0	)	64		
1	1	[	20	!	01	01		
2	2	@	10	2	@	10		
3	3	#	70	3	#	70	*Same*	
4	4	$	04	4	$	04		
5	5	%	08	5	%	08	*as*	
6	6	58	58	^	^	1C		
7	7	&	68	7	&	68		
8	8	*	38	8	*	38	*Unshifted*	
9	9	(	34	9	(	34		
dash	dash	underline	37	dash	underline	37		
=	=	+	13	=	+	13	–	–
\[5]	\	\	1F	–	–	–	–	–
[	1	[	20	–	–	–	DEL	7F
]	–	–	–	–	–	–	–	–
;	;	:	6B	;	:	6B	–	–
'	'	"	49	'	"	49	–	–
,	,	,	3B	–	–	–	RS	2C
.	.	.	51	–	–	–	–	–
/	/	?	07	/	?	07	–	–
Space	Space	Space	40	Space	Space	40	Space	40

Untranslatable characters ("–" in the above table) that are entered in transmit strings will be replaced by SPACE (hex 40) during transmission.

[1]CHAR displayed in Run mode if latest case-control character was lower

[2]CHAR displayed in Run mode if latest case-control character was upper

[3]HEX byte trapped/transmitted (odd parity)

[4]CHAR displayed in Run mode

[5]Enter the hex value for the \ character.

Table D1-8
Keyboard-to-BAUDOT

	UNSHIFTED			SHIFTED			CONTROL		
KEY	LOWER(\)[1]	UPPER(^)[2]	HEX[3]	LOWER(\)[1]	UPPER(^)[2]	HEX[3]	LOWER(\)[1]	UPPER(^)[2]	HEX[3]
A	A	dash	03	A	dash	03	-	-	-
B	B	?	19	B	?	19	-	-	-
C	C	:	0E	C	:	0E	-	-	-
D	D	$	09	D	$	09	-	-	-
E	E	3	01	E	3	01	-	-	-
F	F	!	0D	F	!	0D	-	-	-
G	G	&	1A	G	&	1A	S	BEL	05
H	H	#	14	H	#	14	-	-	-
I	I	8	06	I	8	06	-	-	-
J	J	'	0B	J	'	0B	LF	LF	02
K	K	(	0F	K	(	0F	-	-	-
L	L	)	12	L	)	12	-	-	-
M	M	.	1C	M	.	1C	CR	CR	08
N	N	,	0C	N	,	0C	-	-	-
O	O	9	18	O	9	18	-	-	-
P	P	0	16	P	Ø	16	-	-	-
Q	Q	1	17	Q	1	17	-	-	-
R	R	4	0A	R	4	0A	-	-	-
S	S	BEL	05	S	BEL	05	-	-	-
T	T	5	10	T	5	10	-	-	-
U	U	7	07	U	7	07	-	-	-
V	V	;	1E	V	;	1E	-	-	-
W	W	2	13	W	2	13	-	-	-
X	X	/	1D	X	/	1D	-	-	-
Y	Y	6	15	Y	6	15	-	-	-
Z	Z	"	11	Z	"	11	-	-	-
0	P	0	16	L	)	12	-	-	-
1	Q	1	17	F	!	0D	-	-	-
2	W	2	13	-	-	-	-	-	-
3	E	3	01	H	#	14	-	-	-
4	R	4	0A	D	$	09	-	-	-
5	T	5	10	-	-	-	-	-	-
6	Y	6	15	^	^	1B	-	-	-
7	U	7	07	G	&	1A	-	-	-
8	I	8	06	-	-	-	-	-	-
9	O	9	18	K	(	0F	-	-	-
dash	A	dash	03	-	-	-	-	-	-
=	-	-	-	-	-	-	-	-	-
\[4]	\	\	1F	-	-	-	-	-	-
[	-	-	-	-	-	-	-	-	-
]	-	-	-	-	-	-	-	-	-
;	V	;	1E	C	:	0E	NUL	NUL	00
'	J	'	0B	Z	"	11	-	-	-
,	N	,	0C	-	-	-	-	-	-
.	M	.	1C	-	-	-	-	-	-
/	X	/	1D	B	?	19	-	-	-
Space	Space	Space	04	Space	Space	04	-	-	-

Untranslatable characters ("-" in the above table) that are entered in transmit strings will be replaced by NULL (hex 00) during transmission.

[1]CHAR displayed in Run mode if latest case-control character was letter

[2]CHAR displayed in Run mode if latest case-control character was figure

[3]HEX byte searched for/transmitted

[4]Enter the hex value for the \ character.

Table D1-9
Keyboard-to-JIS7

	UNSHIFTED			SHIFTED			CONTROL		
KEY	LOWER(SI)[1]	UPPER(SO)[2]	HEX[3]	LOWER(SI)[1]	UPPER(SO)[2]	HEX[3]	LOWER(SI)[1]	UPPER(SO)[2]	HEX[3]
A	a	61	61	A	チ	41	SOH	SOH	01
B	b	62	62	B	ツ	42	STX	STX	02
C	c	63	63	C	テ	43	ETX	ETX	03
D	d	64	64	D	ト	44	EOT	EOT	04
E	e	65	65	E	ナ	45	ENQ	ENQ	05
F	f	66	66	F	ニ	46	ACK	ACK	06
G	g	67	67	G	ヌ	47	BEL	BEL	07
H	h	68	68	H	ネ	48	BS	BS	08
I	i	69	69	I	ノ	49	HT	HT	09
J	j	6A	6A	J	ハ	4A	LF	LF	0A
K	k	6B	6B	K	ヒ	4B	VT	VT	0B
L	l	6C	6C	L	フ	4C	FF	FF	0C
M	m	6D	6D	M	ヘ	4D	CR	CR	0D
N	n	6E	6E	N	ホ	4E	SO	SO	0E
O	o	6F	6F	O	マ	4F	SI	SI	0F
P	p	70	70	P	ミ	50	DLE	DLE	10
Q	q	71	71	Q	ム	51	DC1	DC1	11
R	r	72	72	R	メ	52	DC2	DC2	12
S	s	73	73	S	モ	53	DC3	DC3	13
T	t	74	74	T	ヤ	54	DC4	DC4	14
U	u	75	75	U	ユ	55	NAK	NAK	15
V	v	76	76	V	ヨ	56	SYN	SYN	16
W	w	77	77	W	ラ	57	ETB	ETB	17
X	x	78	78	X	リ	58	CAN	CAN	18
Y	y	79	79	Y	ル	59	EM	EM	19

[1]CHAR displayed in Run mode if latest case-control character was Shift In (SI).
[2]CHAR displayed in Run mode if latest case-control character was Shift Out (SO).
[3]HEX byte trapped/transmitted (space parity)

Table D1-9 (continued)

	UNSHIFTED			SHIFTED			CONTROL		
KEY	LOWER($^S{}_I$)[1]	UPPER($^S{}_O$)[2]	HEX[3]	LOWER($^S{}_I$)[1]	UPPER($^S{}_O$)[2]	HEX[3]	LOWER($^S{}_I$)[1]	UPPER($^S{}_O$)[2]	HEX[3]
Z	z	7A	7A	Z	レ	5A	SUB	SUB	1A
0	0	ー	30	)	ゥ	29			
1	1	ア	31	!	。	21			
2	2	イ	32	@	タ	40	*Same*		
3	3	ウ	33	#	」	23			
4	4	エ	34	$	、	24			
5	5	オ	35	%	・	25	*as*		
6	6	カ	36	^	゛	5E			
7	7	キ	37	&	ヲ	26			
8	8	ク	38	*	エ	2A	*Unshifted*		
9	9	ケ	39	(	ィ	28			
dash	dash	ュ	2D	underline	゜	5F			
=	=	ス	3D	+	オ	2B	~	7E	7E
\\[4]	¥	ワ	5C	¦	7C	7C	`	60	60
[	[	ロ	5B	{	7B	7B	DEL	DEL	7F
]	]	ン	5D	}	7D	7D	ESC	ESC	1B
;	;	サ	3B	:	コ	3A	NUL	NUL	00
'	'	ァ	27	"	「	22	GS	GS	1D
,	,	ャ	2C	<	シ	3C	RS	RS	1E
.	.	ョ	2E	>	セ	3E	US	US	1F
/	/	ッ	2F	?	ソ	3F	FS	FS	1C
Space	Space	Space	20	Space	Space	20	Space	Space	20

[1]CHAR displayed in Run mode if latest case-control character was Shift In ($^S{}_I$).
[2]CHAR displayed in Run mode if latest case-control character was Shift Out ($^S{}_O$).
[3]HEX byte trapped/transmitted (space parity)
[4]Enter the hex value for the ¥ and ワ characters.

Table D1-10
Keyboard-to-JIS8 (space parity)[1]

	UNSHIFTED		SHIFTED		CONTROL	
KEY	CHAR[2]	HEX[3]	CHAR[2]	HEX[3]	CHAR[2]	HEX[3]
A	a	61	A	41	SOH	01
B	b	62	B	42	STX	02
C	c	63	C	43	ETX	03
D	d	64	D	44	EOT	04
E	e	65	E	45	ENQ	05
F	f	66	F	46	ACK	06
G	g	67	G	47	BEL	07
H	h	68	H	48	BS	08
I	i	69	I	49	HT	09
J	j	6A	J	4A	LF	0A
K	k	6B	K	4B	VT	0B
L	l	6C	L	4C	FF	0C
M	m	6D	M	4D	CR	0D
N	n	6E	N	4E	SO	0E
O	o	6F	O	4F	SI	0F
P	p	70	P	50	DLE	10
Q	q	71	Q	51	DC1	11
R	r	72	R	52	DC2	12
S	s	73	S	53	DC3	13
T	t	74	T	54	DC4	14
U	u	75	U	55	NAK	15
V	v	76	V	56	SYN	16
W	w	77	W	57	ETB	17
X	x	78	X	58	CAN	18
Y	y	79	Y	59	EM	19
Z	z	7A	Z	5A	SUB	1A
0	0	30	)	29		
1	1	31	!	21		
2	2	32	@	40	*Same*	
3	3	33	#	23		
4	4	34	$	24		
5	5	35	%	25	*as*	
6	6	36	^	5E		
7	7	37	&	26		
8	8	38	*	2A	*Unshifted*	
9	9	39	(	28		
dash	dash	2D	underline	5F		
=	=	3D	+	2B	~	7E
\[4]	¥	5C	¦	7C	`	60
[	[	5B	{	7B	DEL	7F
]	]	5D	}	7D	ESC	1B
;	;	3B	:	3A	NUL	00
'	'	27	"	22	GS	1D
,	,	2C	<	3C	RS	1E
.	.	2E	>	3E	US	1F
/	/	2F	?	3F	FS	1C
Space	Space	20	Space	20	Space	20

[1]Hex data-entry will override parity
[2]CHAR displayed in Run mode
[3]HEX byte trapped/transmitted
[4]Enter the hex value for the ¥ character.

Table D1-11
Keyboard-to-JIS8 (mark parity)[1]

KEY	UNSHIFTED CHAR[2]	UNSHIFTED HEX[3]	SHIFTED CHAR[2]	SHIFTED HEX[3]	CONTROL CHAR[2]	CONTROL HEX[3]
A	E1	E1	ﾁ	C1	81	81
B	E2	E2	ﾂ	C2	82	82
C	E3	E3	ﾃ	C3	83	83
D	E4	E4	ﾄ	C4	84	84
E	E5	E5	ﾅ	C5	85	85
F	E6	E6	ﾆ	C6	86	86
G	E7	E7	ﾇ	C7	87	87
H	E8	E8	ﾈ	C8	88	88
I	E9	E9	ﾉ	C9	89	89
J	EA	EA	ﾊ	CA	8A	8A
K	EB	EB	ﾋ	CB	8B	8B
L	EC	EC	ﾌ	CC	8C	8C
M	ED	ED	ﾍ	CD	8D	8D
N	EE	EE	ﾎ	CE	8E	8E
O	EF	EF	ﾏ	CF	8F	8F
P	F0	F0	ﾐ	D0	90	90
Q	F1	F1	ﾑ	D1	91	91
R	F2	F2	ﾒ	D2	92	92
S	F3	F3	ﾓ	D3	93	93
T	F4	F4	ﾔ	D4	94	94
U	F5	F5	ﾕ	D5	95	95
V	F6	F6	ﾖ	D6	96	96
W	F7	F7	ﾗ	D7	97	97
X	F8	F8	ﾘ	D8	98	98
Y	F9	F9	ﾙ	D9	99	99
Z	FA	FA	ﾚ	DA	9A	9A
0	ｰ	B0	ｩ	A9		
1	ｱ	B1	｡	A1		
2	ｲ	B2	ﾀ	C0	*Same*	
3	ｳ	B3	｣	A3		
4	ｴ	B4	､	A4		
5	ｵ	B5	･	A5	*as*	
6	ｶ	B6	ﾞ	DE		
7	ｷ	B7	ｦ	A6		
8	ｸ	B8	ｪ	AA	*Unshifted*	
9	ｹ	B9	ｨ	A8		
dash	ｭ	AD	ﾟ	DF		

[1]Hex data-entry will override parity
[2]CHAR displayed in Run mode
[3]HEX byte trapped/transmitted

Table D1-11 (continued)

KEY	UNSHIFTED CHAR	UNSHIFTED HEX	SHIFTED CHAR	SHIFTED HEX	CONTROL CHAR	CONTROL HEX
=	ス	BD	オ	AB	FE	FE
\[4]	ワ	DC	FC	FC	E0	E0
[	ロ	DB	FB	FB	FF	FF
]	ン	DD	FD	FD	9B	9B
;	サ	BB	コ	BA	80	80
'	ァ	A7	「	A2	9D	9D
,	ャ	AC	シ	BC	9E	9E
.	ョ	AE	セ	BE	9F	9F
/	ッ	AF	ソ	BF	9C	9C
Space	A0	A0	A0	A0	A0	A0

[4]Enter the hex value for the ワ character.

Appendix D2: Hex-to-Display Translation

The left-hand column in the following table (labeled "INPUT HEX") is the hex value presented on the Run-mode data screen when HEX display is turned on.

The remaining columns show the character that is presented for each hex value in each of the available code sets when hex display is turned off. Where no character in the code set corresponds to the hex value received, hex display is always used.

The bit in the "input hex" value that was received first by the INTERVIEW's receivers will vary with the code. In the column heading for each code we have placed a small arrow next to the rightmost or leftmost bit to indicate which was the first bit received. In IPARS, for example, the leftmost bit is the first bit received.

We have tried also to indicate the significance of each bit. In EBCD, the third bit from the left in the hex value is the least significant (=1) bit, while the rightmost bit is the most significant (=32). This means that the first ten hex values in this code set are not really 00 through 09. Rather they are 00, 20, 10, 30, 08, 28, 18, 38, 04 and 24—corresponding to the characters SPACE, 1, 2, 3, 4, 5, 6, 7, 8, and 9, and corresponding also to the following binary series, which increments from left to right:

00000000
00100000
00010000
00110000
00001000
00101000
00011000
00111000
00000100
00100100
etc.

Table D2-1
Hex-to-Display Translation

INPUT HEX	ASCII[1] (P)64 32 16 8421	EBCDIC[1] 128 64 32 16 8421	EBCD[1] (P)1248 16 32 LOWER	UPPER	XS-3[1] (P)32 16 8421	IPARS[2] 32 16 8421	REV EBCD[3] 1248 16 32 (P) LOWER	UPPER	BAUDOT[1] 16 8421 LETTERS	FIGURES	SELECTRIC[1] (P)1248 16 32 LOWER	UPPER	JIS7[1] (P)64 32 16 8421 LOWER	UPPER	JIS8[1] 128 64 32 16 8421
00	NU	NU	*space*	*space*	*space*	*hex*	*space*	*space*	*hex*	*hex*	SP	SP	NU	NU	NU
01	SH	SH	dāsh	dāsh	]	1	1	=	E	3	!	*hex*	SH	SH	SH
02	SX	SX	@	*hex*	dāsh	2	2	<	LF	LF	t	T	SX	SX	SX
03	EX	EX	&	+	0	3	3	;	A	–	j	J	EX	EX	EX
04	ET	*hex*	8	*	1	4	4	:	*space*	*hex*	4	$	ET	ET	ET
05	EQ	HT	q	Q	2	5	5	%	S	*hex*	o	O	EQ	EQ	EQ
06	AK	*hex*	y	Y	3	6	6	'	I	8	l	L	AK	AK	AK
07	BL	*pad*	h	H	4	7	7	>	U	7	/	?	BL	BL	BL
08	BS	*hex*	4	:	5	8	8	*	CR	CR	5	%	BS	BS	BS
09	HT	*hex*	m	M	6	9	9	(	D	$	'	"	HT	HT	HT
0A	LF	*hex*	u	U	7	0	0	)	R	4	e	E	LF	LF	LF
0B	VT	VT	d	D	8	*	#	"	J	'	p	P	VT	VT	VT
0C	FF	FF	D2	D2	9	CR	[	]	N	,	*hex*	*hex*	FF	FF	FF
0D	CR	CR	*hex*	*hex*	\	*hex*	FF	FF	F	!	*hex*	*hex*	CR	CR	CR
0E	SO	SO	*hex*	*hex*	;	=	^	^	C	:	*hex*	*hex*	SO	SO	SO
0F	SI	SI	D4	D4	[	*hex*	ET	ET	K	(	*hex*	*hex*	SI	SI	SI
10	DL	DL	2,	<	+	*hex*	@	*hex*	T	5	2	@	DL	DL	DL
11	D1	D1	k	K	:	*hex*	/	?	Z	*hex*	.		D1	D1	D1
12	D2	D2	s	S	.	S	s	S	L	)	n	N	D2	D2	D2
13	D3	D3	b	B	?	T	t	T	W	2	=	+	D3	D3	D3
14	D4	*hex*	0	)	A	U	u	U	H	#	z	Z	D4	D4	D4
15	NK	*hex*	*hex*	*hex*	B	V	v	V	Y	6	*hex*	*hex*	NK	NK	NK
16	SY	BS	.	*hex*	C	W	w	W	P	0	*hex*	*hex*	SY	SY	SY
17	EB	*hex*	*hex*	*hex*	D	X	x	X	Q	1	*hex*	*hex*	EB	EB	EB
18	CN	CN	6	'	E	Y	y	Y	O	9	6	*hex*	CN	CN	CN
19	EM	EM	o	O	F	Z	z	Z	B	?	l	I	EM	EM	EM
1A	SB	*hex*	w	W	G	dāsh	*hex*	*hex*	G	&	k	K	SB	SB	SB
1B	EC	*hex*	f	F	H	#	,	*hex*	^	^	q	Q	EC	EC	EC
1C	FS	*hex*	^	^	I	*space*	}	{	M	.	^	^	FS	FS	FS
1D	GS	GS	BS	BS	=	*hex*	LF	LF	X	/	BS	BS	GS	GS	GS
1E	RS	*hex*	EB	EB	<	[	EB	EB	V	;	EB	EB	RS	RS	RS
1F	US	US	\	\	#	,	*hex*	*hex*	\	\	\	\	US	US	US

[1]Select Bit Order/Polarity: NORMAL
[2]Select Bit Order/Polarity: REV-INVERT
[3]Select Bit Order/Polarity: REVERSE-NORM

Table D2-1 (continued)

INPUT HEX	ASCII	EBCDIC	EBCD LOWER	EBCD UPPER	XS-3	IPARS	REV EBCD LOWER	REV EBCD UPPER	BAUDOT LETTERS	BAUDOT FIGURES	SELECTRIC LOWER	SELECTRIC UPPER	JIS7 LOWER	JIS7 UPPER	JIS8
20	*space*	*hex*	1	=	@	@	dash	dash	*hex*	*hex*	1	[	*space*	*space*	*space*
21	!	*hex*	j	J	*	J	j	J	*hex*	*hex*	m	M	!	。	!
22	"	FS	/	?	$	K	k	K	*hex*	*hex*	x	X	"	「	"
23	#	*hex*	a	A	!	L	l	L	*hex*	*hex*	g	G	#	」	#
24	$	*hex*	9	G	J	M	m	M	*hex*	*hex*	0	)	$	、	$
25	%	LF	r	R	K	N	n	N	*hex*	*hex*	s	S	%	・	%
26	&	EB	z	Z	L	O	o	O	*hex*	*hex*	h	H	&	ヲ	&
27	'	EC	i	I	M	P	p	P	*hex*	*hex*	y	Y	'	ァ	'
28	(	*hex*	5	%	N	Q	q	Q	*hex*	*hex*	7	&	(	ィ	(
29	)	*hex*	n	N	O	R	r	R	*hex*	*hex*	r	R	)	ゥ	)
2A	*	*hex*	v	V	P	:	*hex*	*hex*	*hex*	*hex*	d	D	*	ェ	*
2B	+	*hex*	e	E	Q	<	$	!	*hex*	*hex*	;	:	+	ォ	+
2C	,	*hex*	RS	RS	R	+	*hex*	*hex*	*hex*	*hex*	RS	RS	,	ャ	,
2D	–	EQ	CR	CR	%	*hex*	CR	CR	*hex*	*hex*	CR	CR	dash	ュ	dash
2E	.	AK	LF	LF	'	)	BS	BS	*hex*	*hex*	LF	LF	.	ョ	.
2F	/	BL	HT	HT	^	(	*hex*	*hex*	*hex*	*hex*	HT	HT	/	ッ	/
30	0	*hex*	3	;	underline	$	&	+	*hex*	*hex*	3	#	0	ー	0
31	1	*hex*	l	L	(	A	a	A	*hex*	*hex*	v	V	1	ア	1
32	2	SY	t	T	,	B	b	B	*hex*	*hex*	u	U	2	イ	2
33	3	*hex*	c	C	&	C	c	C	*hex*	*hex*	f	F	3	ウ	3
34	4	*hex*	#	"	/	D	d	D	*hex*	*hex*	9	(	4	エ	4
35	5	RS	$	!	S*	E	e	E	*hex*	*hex*	w	W	5	オ	5
36	6	*hex*	,	*hex*	T	F	f	F	*hex*	*hex*	b	B	6	カ	6
37	7	ET	.	*hex*	U	G	g	G	*hex*	*hex*	–	_	7	キ	7
38	8	*hex*	7	>	V	H	h	H	*hex*	*hex*	8	*	8	ク	8
39	9	*hex*	p	P	W	I	i	I	*hex*	*hex*	a	A	9	ケ	9
3A	:	*hex*	x	X	X	?	*hex*	*hex*	*hex*	*hex*	c	C	:	コ	:
3B	;	*hex*	g	G	Y	.	.	*hex*	*hex*	*hex*	,	,	;	サ	;
3C	<	D4	ET	ET	Z	%	~	'	*hex*	*hex*	ET	ET	<	シ	<
3D	=	NK	SY	SY	)	*hex*	HT	HT	*hex*	*hex*	*hex*	*hex*	=	ス	=
3E	>	*hex*	SH	SH	>	"	\	\	*hex*	*hex*	*hex*	*hex*	>	セ	>
3F	?	SB	*pad*	*pad*	"	*hex*	*pad*	*pad*	*hex*	*hex*	*pad*	*pad*	?	ソ	?

*SYNC = even parity S (35_{16}).

Table D2-1 (continued)

INPUT HEX	ASCII	EBCDIC	EBCD LOWER	EBCD UPPER	XS-3	IPARS	REV EBCD LOWER	REV EBCD UPPER	BAUDOT LETTERS	BAUDOT FIGURES	SELECTRIC LOWER	SELECTRIC UPPER	JIS7 LOWER	JIS7 UPPER	JIS8
40	@	*space*	*space*	*space*	*space*	*hex*	*space*	*space*	*hex*	*hex*	SP	SP	@	タ	@
41	A	*hex*	_	_	]	*hex*	1	=	*hex*	*hex*	!	*hex*	A	チ	A
42	B	*hex*	@	*hex*	dāsh	*hex*	2	<	*hex*	*hex*	t	T	B	ツ	B
43	C	*hex*	&	+	0	*hex*	3	;	*hex*	*hex*	j	J	C	テ	C
44	D	*hex*	8	*	1	*hex*	4	:	*hex*	*hex*	4	$	D	ト	D
45	E	*hex*	q	Q	2	*hex*	5	%	*hex*	*hex*	o	O	E	ナ	E
46	F	*hex*	y	Y	3	*hex*	6	'	*hex*	*hex*	l	L	F	ニ	F
47	G	*hex*	h	H	4	*hex*	7	>	*hex*	*hex*	/	?	G	ヌ	G
48	H	*hex*	4	:	5	*hex*	8	*	*hex*	*hex*	5	%	H	ネ	H
49	I	*hex*	m	M	6	*hex*	9	(	*hex*	*hex*	'	"	I	ノ	I
4A	J	*hex*	u	U	7	*hex*	0	)	*hex*	*hex*	e	E	J	ハ	J
4B	K	.	d	D	8	*hex*	#	"	*hex*	*hex*	p	P	K	ヒ	K
4C	L	<	D4	D2	9	*hex*	[	]	*hex*	*hex*	*hex*	*hex*	L	フ	L
4D	M	(	*hex*	*hex*	\	*hex*	FF	FF	*hex*	*hex*	*hex*	*hex*	M	ヘ	M
4E	N	+	*hex*	*hex*	;	*hex*	^	^	*hex*	*hex*	*hex*	*hex*	N	ホ	N
4F	O	*hex*	D4	D4	[	*hex*	ET	ET	*hex*	*hex*	*hex*	*hex*	O	マ	O
50	P	&	2	<	+	*hex*	@	*hex*	*hex*	*hex*	2	@	P	ミ	P
51	Q	*hex*	k	K	:	*hex*	/	?	*hex*	*hex*	.	.	Q	ム	Q
52	R	*hex*	s	S	.	*hex*	s	S	*hex*	*hex*	n	N	R	メ	R
53	S	*hex*	b	B	?	*hex*	t	T	*hex*	*hex*	=	+	S	モ	S
54	T	*hex*	0	)	A	*hex*	u	U	*hex*	*hex*	z	Z	T	ヤ	T
55	U	*hex*	*hex*	*hex*	B*	*hex*	v	V	*hex*	*hex*	*hex*	*hex*	U	ユ	U
56	V	*hex*	*hex*	*hex*	C	*hex*	w	W	*hex*	*hex*	*hex*	*hex*	V	ヨ	V
57	W	*hex*	*hex*	*hex*	D	*hex*	x	X	*hex*	*hex*	*hex*	*hex*	W	ラ	W
58	X	*hex*	6	'	E	*hex*	y	Y	*hex*	*hex*	6	*hex*	X	リ	X
59	Y	*hex*	o	O	F	*hex*	z	Z	*hex*	*hex*	l	I	Y	ル	Y
5A	Z	!	w	W	G	*hex*	*hex*	*hex*	*hex*	*hex*	k	K	Z	レ	Z
5B	[	$	f	F	H	*hex*	,	hex	*hex*	*hex*	q	Q	[	ロ	[
5C	\	*	^	^	I	*hex*	}	{	*hex*	*hex*	^	^	¥	ワ	¥
5D	]	)	BS	BS	=	*hex*	LF	LF	*hex*	*hex*	BS	BS	]	ン	]
5E	^	;	EB	EB	<	*hex*	EB	EB	*hex*	*hex*	EB	EB	^	゛	^
5F	_	*hex*	\	\	#	*hex*	*hex*	*hex*	*hex*	*hex*	\	\	underline	•	underline

*EOM = even parity B (55_{16}).

Table D2-1 (continued)

INPUT HEX	ASCII	EBCDIC	EBCD LOWER	EBCD UPPER	XS-3	IPARS	REV EBCD LOWER	REV EBCD UPPER	BAUDOT LETTERS	BAUDOT FIGURES	SELECTRIC LOWER	SELECTRIC UPPER	JIS7 LOWER	JIS7 UPPER	JIS8
60	'	dāsh	1	=	@	*hex*	dāsh	dāsh	*hex*	*hex*	1	[	'	*hex*	'
61	a	/	j	J	*	*hex*	j	J	*hex*	*hex*	m	M	a	*hex*	a
62	b	*hex*	/	?	$	*hex*	k	K	*hex*	*hex*	x	X	b	*hex*	b
63	c	*hex*	a	A	!	*hex*	l	L	*hex*	*hex*	g	G	c	*hex*	c
64	d	*hex*	9	(	J	*hex*	m	M	*hex*	*hex*	0	)	d	*hex*	d
65	e	*hex*	r	R	K	*hex*	n	N	*hex*	*hex*	s	S	e	*hex*	e
66	f	*hex*	z	Z	L	*hex*	o	O	*hex*	*hex*	h	H	f	*hex*	f
67	g	*hex*	l	I	M	*hex*	p	P	*hex*	*hex*	y	Y	g	*hex*	g
68	h	*hex*	5	%	N	*hex*	q	Q	*hex*	*hex*	7	&	h	*hex*	h
69	i	*hex*	n	N	O	*hex*	r	R	*hex*	*hex*	r	RI	i	*hex*	i
6A	j	¦	v	V	P	*hex*	*hex*	*hex*	*hex*	*hex*	d	D	j	*hex*	j
6B	k	,	e	E	Q	*hex*	$	!	*hex*	*hex*	;	:	k	*hex*	k
6C	l	%	RS	RS	R	hex	*hex*	*hex*	*hex*	*hex*	RS	RS	l	*hex*	l
6D	m	_	CR	CR	%	*hex*	CR	CR	*hex*	*hex*	CR	CR	m	*hex*	m
6E	n	>	LF	LF	'	*hex*	BS	BS	*hex*	*hex*	LF	LF	n	*hex*	n
6F	o	?	HT	HT	^	*hex*	*hex*	*hex*	*hex*	*hex*	HT	HT	o	*hex*	o
70	p	*hex*	3	;	_	*hex*	&	+	*hex*	*hex*	3	#	p	*hex*	p
71	q	*hex*	l	L	(	*hex*	a	A	*hex*	*hex*	v	V	q	*hex*	q
72	r	*hex*	t	T	,	*hex*	b	B	*hex*	*hex*	u	U	r	*hex*	r
73	s	*hex*	c	C	&	*hex*	c	C	*hex*	*hex*	f	F	s	*hex*	s
74	t	*hex*	#	"	/	*hex*	d	D	*hex*	*hex*	q	(	t	*hex*	t
75	u	*hex*	$	!	S	*hex*	e	E	*hex*	*hex*	w	W	u	*hex*	u
76	v	*hex*	,	*hex*	T	*hex*	f	F	*hex*	*hex*	b	B	v	*hex*	v
77	w	*hex*	.	*hex*	U	*hex*	g	G	*hex*	*hex*	dāsh	underline	w	*hex*	w
78	x	*hex*	7	>	V	*hex*	h	H	*hex*	*hex*	8	*	x	*hex*	x
79	y	'	p	P	W	*hex*	i	I	*hex*	*hex*	a	A	y	*hex*	y
7A	z	:	x	X	X	*hex*	*hex*	*hex*	*hex*	*hex*	c	C	z	*hex*	z
7B	{	#	g	G	Y	*hex*	.	*hex*	*hex*	*hex*	,	,	{	*hex*	{
7C	¦	@	ET	ET	Z	*hex*	~	'	*hex*	*hex*	ET	ET	¦	*hex*	¦
7D	}	'	SY	SY	)	*hex*	*HT*	*HT*	*hex*	*hex*	*hex*	*hex*	}	*hex*	}
7E	~	=	SH	SH	>	*hex*	\	\	*hex*	*hex*	*hex*	*hex*	~	*hex*	~
7F	*pad*	"	*pad*	*pad*	"	*hex*	*pad*	*pad*	*hex*	*hex*	*pad*	*pad*	*pad*	*hex*	*pad*

Table D2-1 (continued)

INPUT HEX	ASCII	EBCDIC	EBCD LOWER	EBCD UPPER	XS-3	IPARS	REV EBCD LOWER	REV EBCD UPPER	BAUDOT LETTERS	BAUDOT FIGURES	SELECTRIC LOWER	SELECTRIC UPPER	JIS7 LOWER	JIS7 UPPER	JIS8
80	NU	*hex*			*hex*	*hex*							NU	NU	*hex*
81	SH	a			*hex*	*hex*							SH	SH	*hex*
82	SX	b			*hex*	*hex*							SX	SX	*hex*
83	EX	c			*hex*	*hex*							EX	EX	*hex*
84	ET	d			*hex*	*hex*							ET	ET	*hex*
85	EQ	e			*hex*	*hex*							EQ	EQ	*hex*
86	AK	f			*hex*	*hex*							AK	AK	*hex*
87	BL	g			*hex*	*hex*							BL	BL	*hex*
88	BS	h			*hex*	*hex*							BS	BS	*hex*
89	HT	i			*hex*	*hex*							HT	HT	*hex*
8A	LF	*hex*			*hex*	*hex*							LF	LF	*hex*
8B	VT	*hex*			*hex*	*hex*							VT	VT	*hex*
8C	FF	*hex*			*hex*	*hex*							FF	FF	*hex*
8D	CR	*hex*			*hex*	*hex*							CR	CR	*hex*
8E	SO	*hex*			*hex*	*hex*							SO	SO	*hex*
8F	SI	*hex*			*hex*	*hex*							SI	SI	*hex*
90	DL	*hex*			*hex*	*hex*							DL	DL	*hex*
91	D1	j			*hex*	*hex*							D1	D1	*hex*
92	D2	k			*hex*	*hex*							D2	D2	*hex*
93	D3	l			*hex*	*hex*							D3	D3	*hex*
94	D4	m			*hex*	*hex*							D4	D4	*hex*
95	NK	n			*hex*	*hex*							NK	NK	*hex*
96	SY	o			*hex*	*hex*							SY	SY	*hex*
97	EB	p			*hex*	*hex*							EB	EB	*hex*
98	CN	q			*hex*	*hex*							CN	CN	*hex*
99	EM	r			*hex*	*hex*							EM	EM	*hex*
9A	SB	*hex*			*hex*	*hex*							SB	SB	*hex*
9B	EC	*hex*			*hex*	*hex*							EC	EC	*hex*
9C	FS	*hex*			*hex*	*hex*							FS	FS	*hex*
9D	GS	*hex*			*hex*	*hex*							GS	GS	*hex*
9E	RS	*hex*			*hex*	*hex*							RS	RS	*hex*
9F	US	*hex*			*hex*	*hex*							US	US	*hex*

Table D2-1 (continued)

INPUT HEX	ASCII	EBCDIC	EBCD LOWER	EBCD UPPER	XS-3	IPARS	REV EBCD LOWER	REV EBCD UPPER	BAUDOT LETTERS	BAUDOT FIGURES	SELECTRIC LOWER	SELECTRIC UPPER	JIS7 LOWER	JIS7 UPPER	JIS8
A0	*space*	*hex*			*hex*	*hex*							*space*	*space*	*hex*
A1	!	~			*hex*	*hex*							!	｡	｡
A2	"	s			*hex*	*hex*							"	「	「
A3	#	t			*hex*	*hex*							#	」	」
A4	$	u			*hex*	*hex*							$	、	、
A5	%	v			*hex*	*hex*							%	・	・
A6	&	w			*hex*	*hex*							&	ヲ	ヲ
A7	'	x			*hex*	*hex*							'	ァ	ァ
A8	(	y			*hex*	*hex*							(	ィ	ィ
A9	)	z			*hex*	*hex*							)	ゥ	ゥ
AA	*	*hex*			*hex*	*hex*							*	ェ	ェ
AB	+	*hex*			*hex*	*hex*							+	ォ	ォ
AC	,	*hex*			*hex*	*hex*							,	ャ	ャ
AD	dash	*hex*			*hex*	*hex*							dash	ュ	ュ
AE	.	*hex*			*hex*	*hex*							.	ョ	ョ
AF	/	*hex*			*hex*	*hex*							/	ッ	ッ
B0	0	*hex*			*hex*	*hex*							0	ー	ー
B1	1	*hex*			*hex*	*hex*							1	ア	ア
B2	2	*hex*			*hex*	*hex*							2	イ	イ
B3	3	*hex*			*hex*	*hex*							3	ウ	ウ
B4	4	*hex*			*hex*	*hex*							4	エ	エ
B5	5	*hex*			*hex*	*hex*							5	オ	オ
B6	6	*hex*			*hex*	*hex*							6	カ	カ
B7	7	*hex*			*hex*	*hex*							7	キ	キ
B8	8	*hex*			*hex*	*hex*							8	ク	ク
B9	9	*hex*			*hex*	*hex*							9	ケ	ケ
BA	:	*hex*			*hex*	*hex*							:	コ	コ
BB	;	*hex*			*hex*	*hex*							;	サ	サ
BC	<	*hex*			*hex*	*hex*							<	シ	シ
BD	=	*hex*			*hex*	*hex*							=	ス	ス
BE	>	*hex*			*hex*	*hex*							>	セ	セ
BF	?	*hex*			*hex*	*hex*							?	ソ	ソ

Table D2-1 (continued)

INPUT HEX	ASCII	EBCDIC	EBCD LOWER	EBCD UPPER	XS-3	IPARS	REV EBCD LOWER	REV EBCD UPPER	BAUDOT LETTERS	BAUDOT FIGURES	SELECTRIC LOWER	SELECTRIC UPPER	JIS7 LOWER	JIS7 UPPER	JIS8
C0	@	{			*hex*	*hex*							@	タ	タ
C1	A	A			*hex*	*hex*							A	チ	チ
C2	B	B			*hex*	*hex*							B	ツ	ツ
C3	C	C			*hex*	*hex*							C	テ	テ
C4	D	D			*hex*	*hex*							D	ト	ト
C5	E	E			*hex*	*hex*							E	ナ	ナ
C6	F	F			*hex*	*hex*							F	ニ	ニ
C7	G	G			*hex*	*hex*							G	ヌ	ヌ
C8	H	H			*hex*	*hex*							H	ネ	ネ
C9	I	I			*hex*	*hex*							I	ノ	ノ
CA	J	*hex*			*hex*	*hex*							J	ハ	ハ
CB	K	*hex*			*hex*	*hex*							K	ヒ	ヒ
CC	L	*hex*			*hex*	*hex*							L	フ	フ
CD	M	*hex*			*hex*	*hex*							M	ヘ	ヘ
CE	N	*hex*			*hex*	*hex*							N	ホ	ホ
CF	O	*hex*			*hex*	*hex*							O	マ	マ
D0	P	}			*hex*	*hex*							P	ミ	ミ
D1	Q	J			*hex*	*hex*							Q	ム	ム
D2	R	K			*hex*	*hex*							R	メ	メ
D3	S	L			*hex*	*hex*							S	モ	モ
D4	T	M			*hex*	*hex*							T	ヤ	ヤ
D5	U	N			*hex*	*hex*							U	ユ	ユ
D6	V	O			*hex*	*hex*							V	ヨ	ヨ
D7	W	P			*hex*	*hex*							W	ラ	ラ
D8	X	Q			*hex*	*hex*							X	リ	リ
D9	Y	R			*hex*	*hex*							Y	ル	ル
DA	Z	*hex*			*hex*	*hex*							Z	レ	レ
DB	[	*hex*			*hex*	*hex*							[	ロ	ロ
DC	\	*hex*			*hex*	*hex*							¥	ワ	ワ
DD	]	*hex*			*hex*	*hex*							]	ン	ン
DE	^	*hex*			*hex*	*hex*							^	゛	゛
DF	_	*hex*			*hex*	*hex*							underline	゜	゜

Table D2-1 (continued)

INPUT HEX	ASCII	EBCDIC	EBCD LOWER	EBCD UPPER	XS-3	IPARS	REV EBCD LOWER	REV EBCD UPPER	BAUDOT LETTERS	BAUDOT FIGURES	SELECTRIC LOWER	SELECTRIC UPPER	JIS7 LOWER	JIS7 UPPER	JIS8
E0	`	{			*hex*	*hex*							`	*hex*	*hex*
E1	a	*hex*			*hex*	*hex*							a	*hex*	*hex*
E2	b	S			*hex*	*hex*							b	*hex*	*hex*
E3	c	T			*hex*	*hex*							c	*hex*	*hex*
E4	d	U			*hex*	*hex*							d	*hex*	*hex*
E5	e	V			*hex*	*hex*							e	*hex*	*hex*
E6	f	W			*hex*	*hex*							f	*hex*	*hex*
E7	g	X			*hex*	*hex*							g	*hex*	*hex*
E8	h	Y			*hex*	*hex*							h	*hex*	*hex*
E9	i	Z			*hex*	*hex*							i	*hex*	*hex*
EA	j	*hex*			*hex*	*hex*							j	*hex*	*hex*
EB	k	*hex*			*hex*	*hex*							k	*hex*	*hex*
EC	l	*hex*			*hex*	*hex*							l	*hex*	*hex*
ED	m	*hex*			*hex*	*hex*							m	*hex*	*hex*
EE	n	*hex*			*hex*	*hex*							n	*hex*	*hex*
EF	o	*hex*			*hex*	*hex*							o	*hex*	*hex*
F0	p	0			*hex*	*hex*							p	*hex*	*hex*
F1	q	1			*hex*	*hex*							q	*hex*	*hex*
F2	r	2			*hex*	*hex*							r	*hex*	*hex*
F3	s	3			*hex*	*hex*							s	*hex*	*hex*
F4	t	4			*hex*	*hex*							t	*hex*	*hex*
F5	u	5			*hex*	*hex*							u	*hex*	*hex*
F6	v	6			*hex*	*hex*							v	*hex*	*hex*
F7	w	7			*hex*	*hex*							w	*hex*	*hex*
F8	x	8			*hex*	*hex*							x	*hex*	*hex*
F9	y	9			*hex*	*hex*							y	*hex*	*hex*
FA	z	*hex*			*hex*	*hex*							z	*hex*	*hex*
FB	{	*hex*			*hex*	*hex*							{	*hex*	*hex*
FC	¦	*hex*			*hex*	*hex*							¦	*hex*	*hex*
FD	}	*hex*			*hex*	*hex*							}	*hex*	*hex*
FE	~	*hex*			*hex*	*hex*							~	*hex*	*hex*
FF	*pad*	*hex*			*hex*	*hex*							*pad*	*hex*	*hex*

Appendix D3: User-Defined Codes

The character set shown in Table D3-1 can be used to adapt existing code sets or to create customized codes. Follow the steps in the example below to create a new code set.

As an example, we'll change the standard ASCII code set to one which includes the ¥ (yen) symbol.

1. *Determine hex values.* First, we will determine which hexadecimal value or values we want to have generate the ¥ symbol, one for space parity and one for mark parity. In our example, the values will be hexadecimal 5C and DC.

2. *Read existing code file to spreadsheet.* Whether adapting an existing code set or creating a new one, use an existing code file as a template. (Some files include shifted and unshifted coding.) Go to the Protocol Spreadsheet and press [EDIT], BLOCK ([F1]), IN/OUT ([F7]), READ/U ([F3]). Enter the name of the file when prompted. The absolute pathname of the standard ASCII code file is *HRD/sys/codes/ASCII.* Press [XEQ]. Do not use the Load command on the File Maintenance screen to access the file.

   The ASCII code set will be displayed on the Protocol Spreadsheet, as in Figure D3-1. Initially, the file is highlighted on the Protocol Spreadsheet in reverse video. You may clear the highlighting by pressing [DONE], [F3]. Since you will be writing your revised code set back to a file, however, you may want to retain the highlighting. Then you will not have to identify the block again before writing.

3. *Locate position.* Positions in the code proceed sequentially, beginning with hexadecimal 00 and ending with FF. Each row in the code table contains eight elements. The first two rows, for example, correspond to hex 00 through 0F. The next two rows contain elements in positions 10 through 1F, and so on. Move the cursor to position 5C.

```
                    ** Protocol Spreadsheet **
Name of file: HRD/sys/codes/ASCII
/
/   Code for ASCII
/

Version:  3;

Name: "ASCII";

To Graphic:

nu   sh   sx   ex   et   eq   ak   bl
bs   ht   lf   vt   ff   cr   so   si
dl   dl   d2   d3   d4   nk   sy   eb
cn   em   sb   ec   fs   gs   rs   us
' '  '!'  '"'  '#'  '$'  '%'  '&'  '''
'('  ')'  '*'  '+'  ','  '-'  '.'  '/'

  F 1     F 2     F 3     F 4      F 5     F 6     F 7      F 8
 BEGIN    END    CLEAR  DELETE    MOVE    COPY   IN/OUT
```

Figure D3-1 When the standard ASCII code file is written to the Protocol Spreadsheet, a code table appears with 32 rows of eight elements per row, corresponding to 256 possible hex values.

4. *Enter new code.* Replace the entry with a new value. Refer to Table D3-1. All values under "Code–Table Entry" are three–digit hexadecimals. A leading zero identifies an entry as a numerical value and guarantees accurate translation. Notice in Figure D3-1 that there is special notation for ASCII control characters. A *d1* entry, for example, translates as the ASCII control character ᴰ₁. Entered as *0d1* (or *0D1*), the meaning is æ. Values which begin with a digit in the range 0–9, *80* for example, do not strictly require the leading zero. Also notice in Figure D3-1 and Figure D3-2 that alphanumerics may be entered as character constants. A set of single quotation marks surrounds a character constant, an alternative way of entering ASCII keyboard characters.

   In our example, replace '\' with *080*. Figure D3-2 shows the set after the first replacement. Next, locate and edit position DC.

```
                         ** Protocol Spreadsheet **
Name of file: HRD/sys/codes/ASCII
Name: "ASCII";

To Graphic:

nu   sh   sx   ex   et   eq   ak   bl
bs   ht   lf   vt   ff   cr   so   si
dl   dl   d2   d3   d4   nk   sy   eb
cn   em   sb   ec   fs   gs   rs   us
' '  '!'  '"'  '#'  '$'  '%'  '&'  '''
'('  ')'  '*'  '+'  ','  '-'  '.'  '/'
'0'  '1'  '2'  '3'  '4'  '5'  '6'  '7'
'8'  '9'  ':'  ';'  '<'  '='  '>'  '?'
'@'  'A'  'B'  'C'  'D'  'E'  'F'  'G'
'H'  'I'  'J'  'K'  'L'  'M'  'N'  'O'
'P'  'Q'  'R'  'S'  'T'  'U'  'V'  'W'
'X'  'Y'  'Z'  '['  080  ']'  '^'  '_'

  F 1      F 2      F 3      F 4       F 5      F 6      F 7      F 8
 BEGIN     END     CLEAR   DELETE     MOVE     COPY    IN/OUT
```

Figure D3-2 On the bottom line of the spreadsheet, the entry *080* has replaced the previous entry. On Table D3-1, *080* corresponds to the *yen* symbol.

5. *Write file to disk.* If you cleared the highlighting, mark the file via the BLOCK, BEGIN, and END softkey selections. Use the BLOCK, IN/OUT, and WRITE/U commands to write your code to the disk. Give the file a different name to prevent an existing file from being overwritten.

6. *Reboot.* Turn the INTERVIEW off. When you turn the unit back on, it will reboot and automatically load in the new (or edited) code set. The first seven characters in the name of the code file will be displayed as a softkey selection for the **Code** field on the Line Setup screen.

> **NOTE:** If your code contains an error—a hexadecimal value does not begin with a digit, for example—it will not be loaded into the INTERVIEW's memory, even if it appears as a **Code** selection on the Line Setup menu. Usually, the standard ASCII code will be used instead.

Table D3-1
Code-Set Characters

Character	Code-Table Entry	Character	Code-Table Entry
NU	000	EC	01b
SH	001	FS	01c
SX	002	GS	01d
EX	003	RS	01e
ET	004	US	01f
EQ	005	*(space)*	020
AK	006	!	021
BL	007	"	022
BS	008	#	023
HT	009	$	024
LF	00a	%	025
VT	00b	&	026
FF	00c	'	027
CR	00d	(	028
SO	00e	)	029
SI	00f	*	02a
DL	010	+	02b
D1	011	,	02c
D2	012	−	02d
D3	013	.	02e
D4	014	/	02f
NK	015	Ø	030
SY	016	1	031
EB	017	2	032
CN	018	3	033
EM	019		
SB	01a		

Table D3-1 (continued)

Character	Code-Table Entry	Character	Code-Table Entry
4	034	P	050
5	035	Q	051
6	036	R	052
7	037	S	053
8	038	T	054
9	039	U	055
:	03a	V	056
;	03b	W	057
<	03c	X	058
=	03d	Y	059
>	03e	Z	05a
?	03f	[	05b
@	040	\	05c
A	041	]	05d
B	042	^	05e
C	043	_	05f
D	044	`	060
E	045	a	061
F	046	b	062
G	047	c	063
H	048	d	064
I	049	e	065
J	04a	f	066
K	04b	g	067
L	04c	h	068
M	04d	i	069
N	04e		
O	04f		

Table D3-1 (continued)

Character	Code-Table Entry	Character	Code-Table Entry
j	06a	ｦ	086
k	06b	ｧ	087
l	06c	ｨ	088
m	06d	ｩ	089
n	06e	ｪ	08a
o	06f	ｫ	08b
p	070	ｬ	08c
q	071	ｭ	08d
r	072	ｮ	08e
s	073	ｯ	08f
t	074	ｰ	090
u	075	ｱ	091
v	076	ｲ	092
w	077	ｳ	093
x	078	ｴ	094
y	079	ｵ	095
z	07a	ｶ	096
{	07b	ｷ	097
¦	07c	ｸ	098
}	07d	ｹ	099
~	07e	ｺ	09a
▨	07f	ｻ	09b
¥	080	ｼ	09c
｡	081	ｽ	09d
｢	082	ｾ	09e
｣	083	ｿ	09f
､	084		
･	085		

Table D3-1 (continued)

Character	Code-Table Entry	Character	Code-Table Entry
タ	0a0	ロ	0bb
チ	0a1	ワ	0bc
ツ	0a2	ン	0bd
テ	0a3	゛	0be
ト	0a4	゜	0bf
ナ	0a5	Ç	0c0
ニ	0a6	ü	0c1
ヌ	0a7	é	0c2
ネ	0a8	â	0c3
ノ	0a9	ä	0c4
ハ	0aa	à	0c5
ヒ	0ab	å	0c6
フ	0ac	ç	0c7
ヘ	0ad	ê	0c8
ホ	0ae	ë	0c9
マ	0af	è	0ca
ミ	0b0	ï	0cb
ム	0b1	î	0cc
メ	0b2	ì	0cd
モ	0b3	Ä	0ce
ヤ	0b4	Å	0cf
ユ	0b5	É	0d0
ヨ	0b6	æ	0d1
ラ	0b7	Æ	0d2
リ	0b8	ô	0d3
ル	0b9		
レ	0ba		

Table D3-1 (continued)

Character	Code-Table Entry	Character	Code-Table Entry
ö	0d4	ú	0e3
ò	0d5	ñ	0e4
û	0d6	Ñ	0e5
ù	0d7	ª	0e6
ÿ	0d8	º	0e7
Ö	0d9	¿	0e8
Ü	0da	⌐	0e9
¢	0db	¬	0ea
£	0dc	½	0eb
B	0dd	¼	0ec
₧	0de	¡	0ed
ƒ	0df	¨	0ee
á	0e0	§	0ef
í	0e1	•	0f0 †
ó	0e2		

† Values 0f1–0ff are undefined.

Appendix E: Communications with the AR Division Factory

All communications with the factory of the AR Division of Telenex Corporation begin with a call to Customer Service:

Customers outside the Washington D.C. Greater Metropolitan Area and Virginia	1-800-368-3261
In Virginia	1-703-644-9190
Local customers	644-9190

If necessary, Customer Service will direct your call to the appropriate department.

E.1 Returning an INTERVIEW or Subassemblies for Repair

(A) Authorization

1. The first step is always to call AR Division Customer Service in Springfield, Virginia.

2. Customer Service will issue a RETURN AUTHORIZATION (RA) number. This number should be posted on the outside of the package of all equipment returned for repair. The RA number, as well as a description of the problem, should be cited in all documentation, written correspondence, or telephone conversations concerning the specific repair.

WARNING: Special RA numbers are issued for customers who have purchased a Maintenance Agreement plan (or plans) from AR Division. Since these numbers identify equipment under maintenance, you <u>must</u> post this RA number on the outside of the package in order for AR Division to honor the terms of the Maintenance Agreement.

3. Turnaround time for repairs is usually two weeks in addition to transportation time. Customer Service can arrange to furnish a rental unit if it is not practical for you to be without the equipment for that length of time. We can either include the rental fee on the repair bill or bill the rental fee separately.

   NOTE: AR Division offers expedited service Maintenance Agreement plans. Under these plans, the customer chooses between expedited repair (72-hour factory turnaround) or a loaner unit for the duration of the repair. Contact Customer Service for complete details.

(B) Shipping

1. Always include with the shipment a detailed description of the problem to be corrected. Put the assigned RA number on this document.

2. If the item is out of warranty, you should either

   a. provide a purchase order for the repair, or

   b. request an estimate of the amount of the repair.

3. Select suitable packing materials for electronic equipment containing a cathode ray tube, and pack the INTERVIEW with care. If possible, the carton and foam packing material in which you received the equipment should be used for returning it for repairs.

4. Write the return authorization number on the outside of the shipment: "ATTN RA *number*."

5. International customers should address the shipment to

   Telenex Corporation, AR Division
   ATTN RA *number*
   c/o Emery Customs Brokers
   101A Executive Drive
   Sterling, Virginia 22170
   U.S.A.

   NOTE: For customs purposes, international customers *MUST* identify the country of origin (usually the U.S.A.) for returned equipment on the *pro forma* invoice. When returning an individual part, use the country of origin listed on the part.

6. Domestic customers should address the shipment to

   Customer Service
   Telenex Corporation
   AR Division
   ATTN RA *number*
   7401 Boston Boulevard
   Springfield, Virginia 22153
   U.S.A.

7. Ship PREPAID *even if you have a Maintenance Agreement with AR Division*. No collect shipments will be accepted unless previously authorized by Customer Service.

8. Most repairs will be completed within two weeks, not including transportation time.

E.2 Ordering Replacement Parts or Assemblies

To obtain price quotations or to order spare or replacement parts, contact Customer Service. Customer Service will need to know the model designation of the unit, its serial number and software version, and what options are installed.

E.3 PC Board or Subassembly Exchanges

The AR Division's repair replacement policy applies to the exchange of PC Boards or Subassemblies that need repair. Please contact Customer Service.

E.4 For Analysis of Problems

For applications, troubleshooting, or repair problems requiring technical assistance, call Customer Service.

E.5 Warranties

There is a standard warranty on all AR Division equipment. This warranty is for 12 months.

Extended and/or Expedited Service Agreements are available for INTERVIEW 7000 Series equipment. Operating system software maintenance is also offered. Please contact Customer Service.

E.6 Loaner Units

Loaner units are available under some hardware Maintenance Agreement plans. Contact Customer Service for additional information.

Appendix F: Packing and Shipping Instructions

The INTERVIEW is usually shipped either as baggage or as freight. The basic difference, of course, is in quantity and quality of handling to which the unit is subjected. It follows that different packing methods are called for.

When a unit is shipped as baggage, it will probably be subjected to much less severe treatment than when it is shipped by freight. The AR Division of Telenex Corporation offers its INTERVIEW Soft Pack Travel Bag, Option No. OPT-951-99-1, for this purpose. This bag has two inches of high-density foam protecting all surfaces of the INTERVIEW. It is yellow for easy identification among other luggage. An identification card case, FRAGILE markings, and leather appointments are standard features. On the outside is a large pocket for carrying notes, manuals, and so forth.

Before packing the INTERVIEW in the carrying bag, remove any diskettes from the microdiskette drives. To protect the heads during transit, insert the two yellow plastic shipping diskettes that were delivered with the unit, one in each drive. The manual should go in the front (center) pocket of the travel bag. There is an inside pocket for the power cord and other cables.

Put the INTERVIEW in the bag with its handle up (as in Figure F-1). Then close and secure the bag cover with its velcro closing.

> *CAUTION: The bag is considered to be reasonable protection for the INTERVIEW when it is shipped as baggage. However, it should never be used for freight shipment. The AR Division of Telenex Corporation can assume no liability for damage to units shipped this way, owing to circumstances beyond our control.*

For freight shipment, the INTERVIEW should be packed in molded polyurethane foam and a heavy-duty outer cardboard carton, as delivered by AR Division. All manuals and accessories should be packed in a separate box within the carton. This packing system has been designed to give maximum reasonable protection to the INTERVIEW and ensure its safe arrival. However, damages due to mishandling must be the responsibility of the carrier.

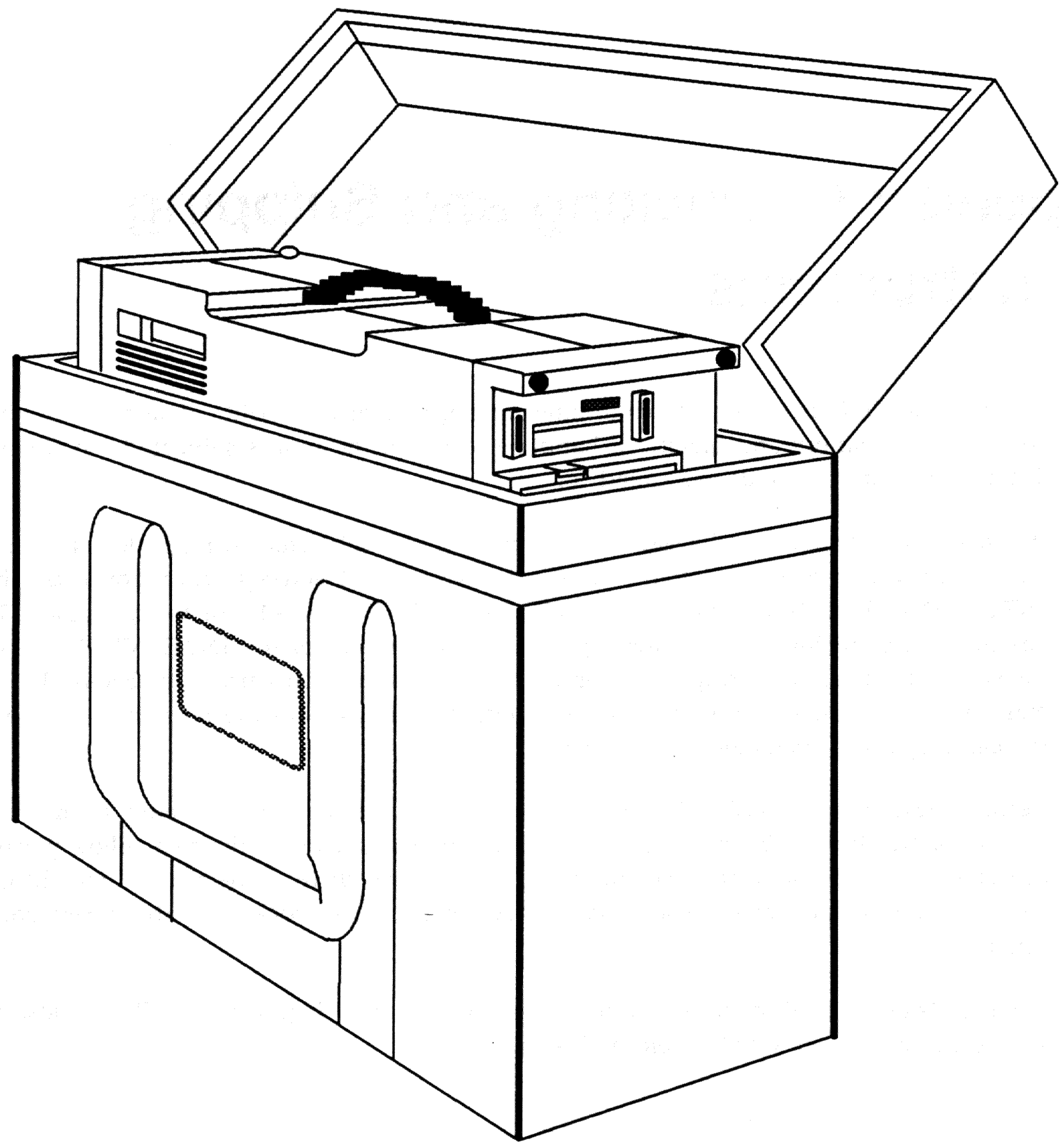


Figure F-1 Soft Pack Travel Bag, Option 99.

Figure F-2 Hard Shell Travel Case, Option 95.

For freight shipment, we also recommend the hard-shell travel case (OPT-951-95-1). See Figure F-2. This is a wheeled suitcase made of high-impact plastic, steel and rubber. It is designed for use with all AR test equipment. Because it has built-in wheels and an extension handle, the hard-shell travel case is especially useful for frequent hand-toting of the instrument.

NOTE: Please do not return any unit to the AR Division without prior authorization (see Appendix E).

Appendix G: Rack Mount (OPT–951–98–1)

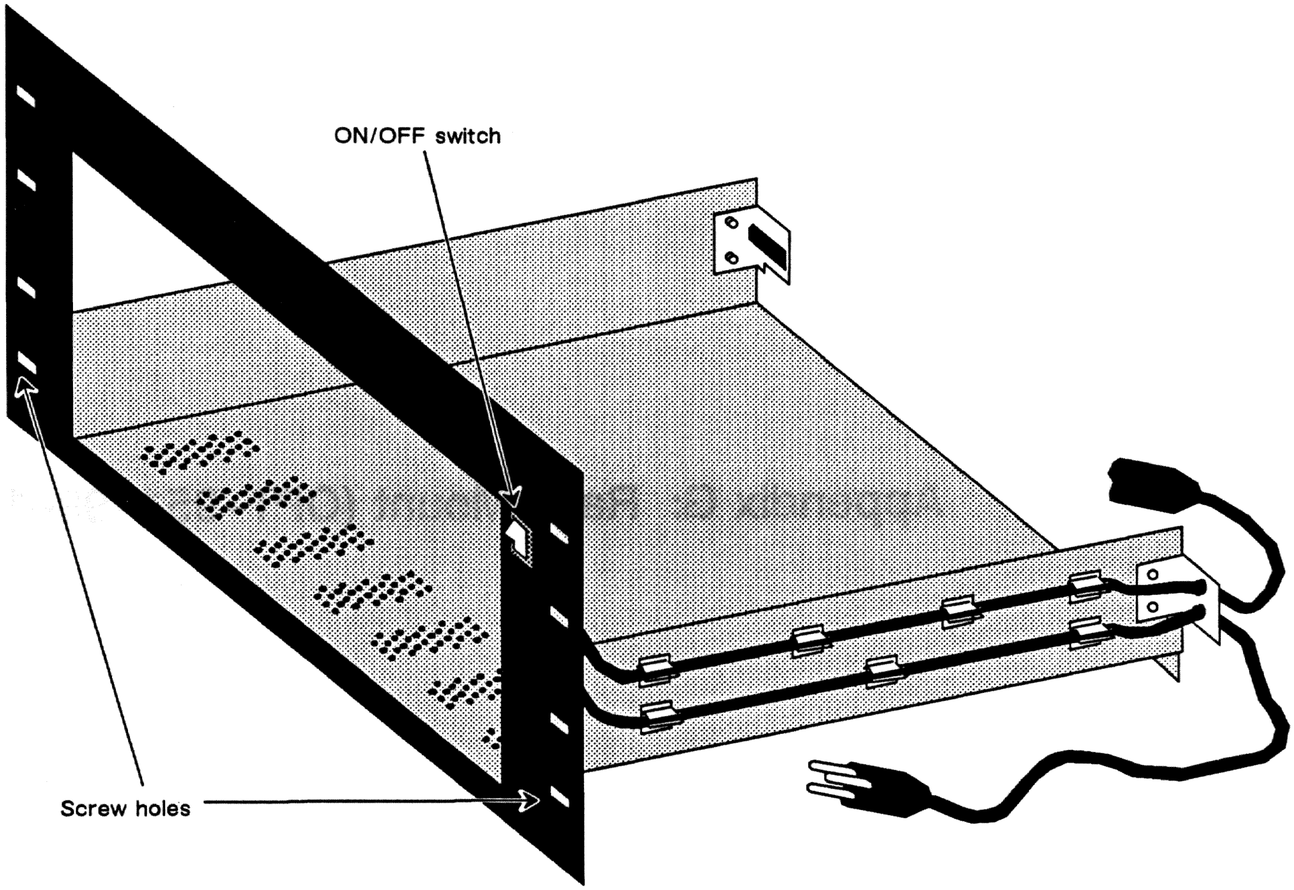


Figure G-1 Rack mount for INTERVIEW 7000/7500.

Appendix G: Rack Mount (OPT–951–98–1)

A Rack Mount Kit (OPT–951–98–1) allows the INTERVIEW to be installed in a standard 19–inch wide equipment rack.

G.1 General Description

The Kit will fit either standard vertical high–boy or sloped front–panel, low–boy racks. Please note that, for proper installation, the rack must be equipped with a horizontal writing shelf.

The Rack Mount Kit offers the user slide–in/out mounting with a sloped keyboard position.

Physical specifications are as follows:

Height: 10.5 inches
Width: 19 inches
Depth: 18 inches
Weight: approximately 5.5 pounds

G.2 Installation

1. Install the rack mount into the front of the cabinet directly above the writing shelf. Secure the rack mount with the eight sets of included black panel screws (ARD #33689) and nut clips (ARD #33686).
2. Slide the INTERVIEW about three–quarters of the way into the opening. *DO NOT SLIDE THE UNIT IN FURTHER AT THIS TIME.*
3. Open the front panel and rest the keyboard on the writing shelf by sliding back the top two blue latches. At this point the hooks of the latches are exposed out the front of the unit. Press down slightly on the recessed circle of these latches and continue to slide the latches inside the unit until they stop. The indented circle should be almost centered in the sliding area and the hooks of the latches are no longer visible from the front of the unit. *These latches must be properly placed or they will lock the keyboard shut if it is accidentally closed.*
4. Carefully slide the unit into the rack mount, with the keyboard lying open, until the front blue rubber bumpers on the right side of the unit are behind the face of the rack mount. You will have approximately one inch of the unit exposed out the front of the cabinet. The INTERVIEW is now in proper position for operation.

5. Notice the rack mount has two electrical wires connecting to a switch mounted on the right front of the rack. Plug the female connector of the top wire into the power connector, located at the bottom left of the rear panel of the unit. It is a standard three-wire grounded male connector.

6. The bottom wire of the rack mount is now the power connector for your unit. Plug this male connector into a standard outlet. Check the voltage selection; see Section 1.5(B). Turn on the power switch, located on the left side of the rear panel of the unit. This permits the ON/OFF switch on the rack mount to become the power switch for your unit.

7. To complete the connections on your unit, refer to Section 1, Hardware.

Appendix H: Optional Codes JIS7/JIS8

JIS7 and JIS8 Katakana character sets are contained in files named *JIS7* and *JIS8* in the */sys/codes* directory of DSK–951–025–1, the floppy diskette that comprises software option OPT–951–22–1. The files should be copied into the */sys/codes* directory on the boot-up disk. When the unit is rebooted, the new codes will be available as **Code** selections on the Line Setup menu.

H.1 Accessing the Directory Containing JIS7 and JIS8 Files

Insert the disk containing the optional codes into Floppy Drive 1 (FD1). With the unit powered on and booted, press [PROGRAM], FMAINT to access the File Maintenance screen. Press CHNGDIR and FLOPPY1, then enter the following pathname in the **Name** field: */sys/codes*. The first two lines of your File Maintenance should look like the screen in Figure H-1.

Press [XEQ] to access the directory containing the *JIS7* and *JIS8* files.

```
                         ** File Maintenance **
Command:  CHANGE DIR                     Push XEQ To Perform Command
Drive: FD1 Name: /sys/codes

Current Directory: HRD/usr
Include               DIR                  32    08/30/88   16:09
layer_pkgs            DIR                 112    12/17/88   09:21








Enter Directory Pathname: /sys/codes
  F 1     F 2     F 3     F 4     F 5     F 6     F 7     F 8
```

Figure H-1 To see the *JIS7* and *JIS8* files in the File Maintenance listings, you must change to the *FD1/sys/codes* directory.

H.2 Copying *JIS7* and *JIS8* Files into */sys/codes* Directory

Press COPY. Leave the source pathname on the **From** line blank: we will make the **From** selections via the MARK key in the body of the current directory listings themselves. Press the ↓ key to move the cursor to the **To** field.

On the **To** line, select the boot disk-drive. This may be the hard (HRD) drive; or you may install the boot-up diskette in Floppy Drive 2 (FD2). If your unit has only a single disk drive, you will use Floppy Drive 1 (FD1) to house first the source disk and then the destination (boot-up) diskette. In that case, select **To:** NEW.

In the **Name** data-entry field, type */sys/codes*. Be sure to type a slash (/) both before and after the *sys* entry.

Now move the cursor into the directory listings. With the blinking cursor positioned over the filename *JIS7*, press MARK. Move the cursor down over *JIS8* and mark this file as well.

Your screen should resemble the screen drawn in Figure H-2. Press XEQ to copy the *JIS7* and *JIS8* files to the */sys/codes* directory on the boot disk.

If you are using a single-drive unit, prompts will "walk" you through the exchange of disks in the single drive.

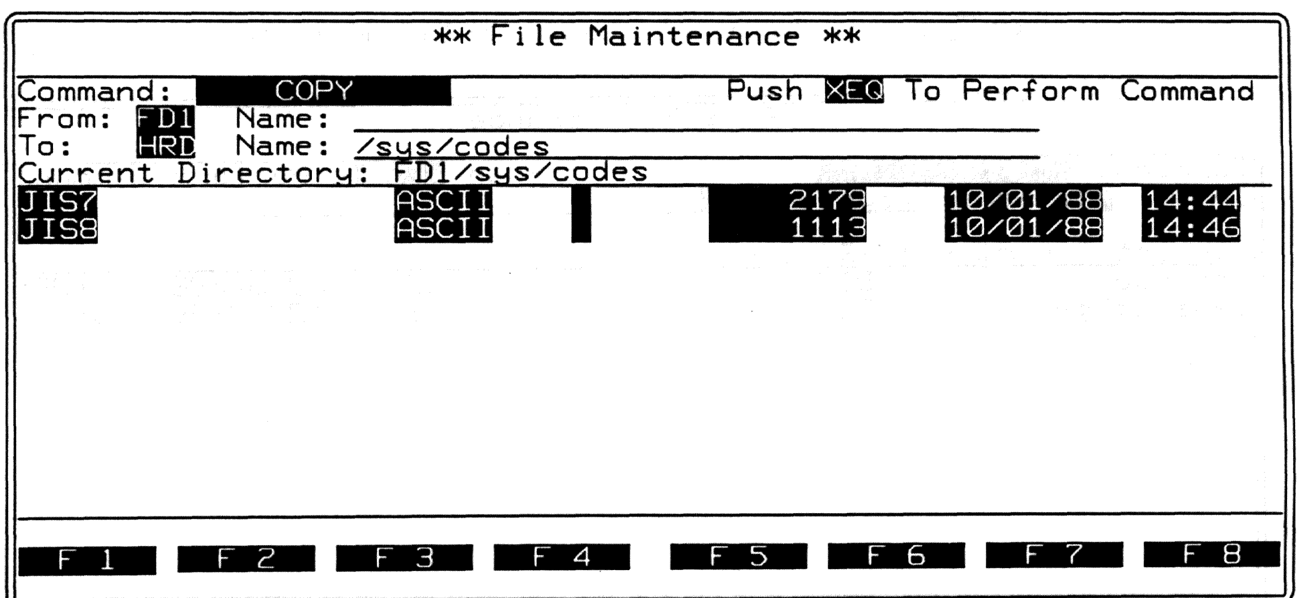


Figure H-2 You may use the *MARK* key to select both JIS files for copying into the */sys/codes* directory on the boot disk.

H.3 Selecting JIS7 or JIS8 Code

Once the JIS files are copied into the */sys/codes* directory, reboot the unit as follows: turn the unit off, wait ten seconds, then turn the power switch on again.

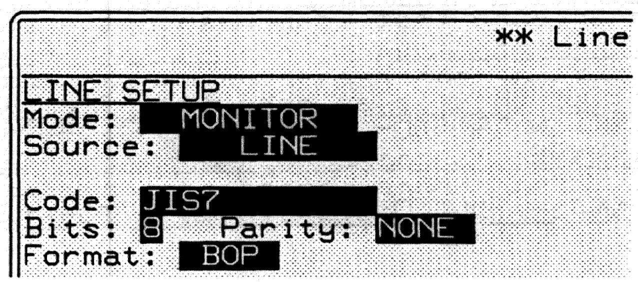


Figure H-3 **Files loaded into the** ***/sys/codes*** **directory are selectable in the** ***Code*** **field on the Line Setup menu.**

After bootup, press PROGRAM, **SETUP**, **LINE**, to access the Line Setup menu. Move the cursor down to the **Code** field. Press ROLL FWD or ROLL BACK to rotate the selections in this field until you have verified that JIS7 and JIS8 are available as new code selections. Figure H-3 shows a line setup with JIS7 selected in the **Code** field.

H.4 Testing with JIS7/JIS8

In your line setup, be sure to change **Mode:** AUTOMON to MONITOR or to one of the emulate modes. The Automonitor sequence will not configure the unit to run with JIS7/JIS8 code, and it will usually change the code selection to ASCII.

Figure H-4 shows a screen display for JIS7, a shifted code. Note that the messages with Katakana text begin with Shift Out (SO, hex 0E).

When you type monitor/receive strings or transmit strings into your program, the characters displayed on the trigger menus or on the Protocol Spreadsheet will always be ASCII. Use the JIS7 and JIS8 charts in Appendix E to correlate your ASCII data-entries with the actual JIS7/JIS8 characters that will be searched for or transmitted.

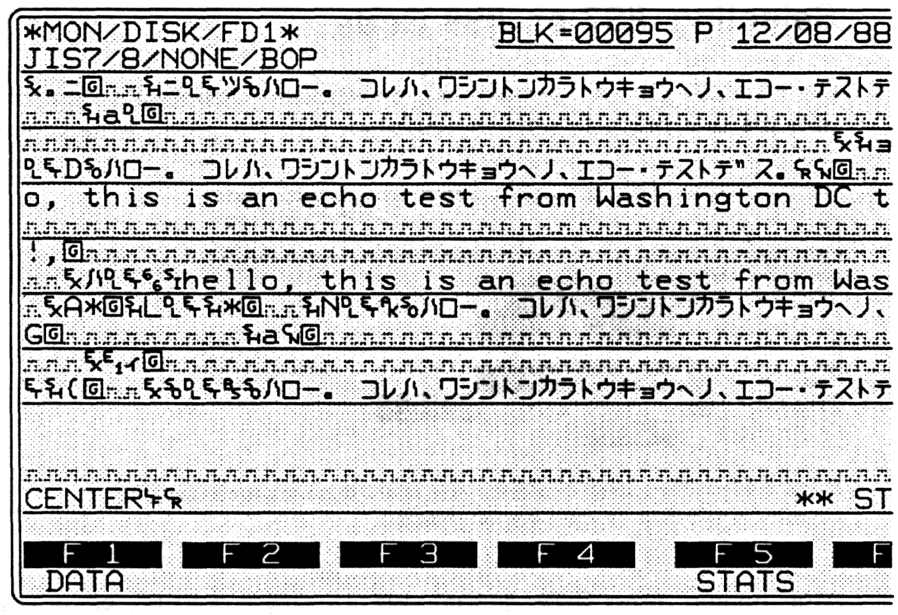
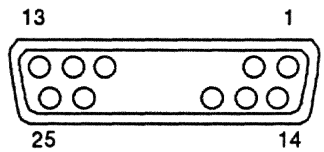


Figure H-4 JIS7 is a shifted code, with an upshift character (*SO*) preceding Katakana conversion and a downshift character (*SI*) preceding ASCII conversion.

Appendix I: Interface Specifications

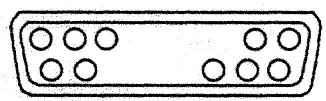
Table I-1
Remote Connector



(DB-25, female)

Pin No.	Pin Name	Signal Description
1	Frame Ground	Ground
2	TD	RS-232/V.24 Output
3	RD	RS-232/V.24 Input
4	RTS	RS-232/V.24 Output
5	CTS	RS-232/V.24 Input
6		
7	Signal Ground	Ground
8	CD	RS-232/V.24 Input
9		
10		
11		
12		
13		
14		
15	SCT	RS-232/V.24 Input
16		
17	SCR	RS-232/V.24 Input
18		
19		
20	DTR	RS-232/V.24 Output
21		
22		
23		
24		
25		

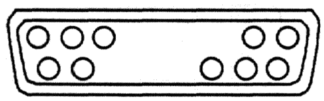
Table I-2
Printer Connector



(DB-25, male)

Pin No.	Pin Name	Signal Description
1	Frame Ground	Ground
2	TD	RS-232/V.24 Input
3	RD	RS-232/V.24 Output
4	RTS	RS-232/V.24 Input
5	CTS	RS-232/V.24 Output
6	DSR	RS-232/V.24 Output
7	Signal Ground	Ground
8	CD	RS-232/V.24 Output
9		
10		
11		
12		
13		
14		
15	SCT	RS-232/V.24 Input
16		
17	SCR	RS-232/V.24 Input
18		
19		
20	DTR	RS-232/V.24 Input
21		
22		
23		
24		
25		

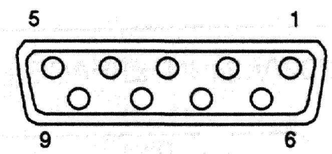
Table I-3
Auxiliary Connector



(16-pin Bi-directional TTL Input/Output, DB-25, female)

Pin No.	Pin Name
1	PA0
2	PB0
3	PA1
4	PB1
5	PA2
6	PB2
7	PA3
8	PB3
9	PA4
10	PB4
11	PA5
12	PB5
13	PA6
14	PB6
15	PA7
16	PB7
17	Signal Ground
18	Reserved
19	Signal Ground
20	Reserved
21	Signal Ground
22	Reserved
23	Signal Ground
24	Reserved
25	Signal Ground

Table I-4
RGB Monitor



(DB-9, female)

Pin No.	Pin Name
1	Signal Ground
2	Signal Ground
3	Red
4	Green
5	Blue
6	Brightness
7	Reserved
8	Horizontal Sync
9	Vertical Sync

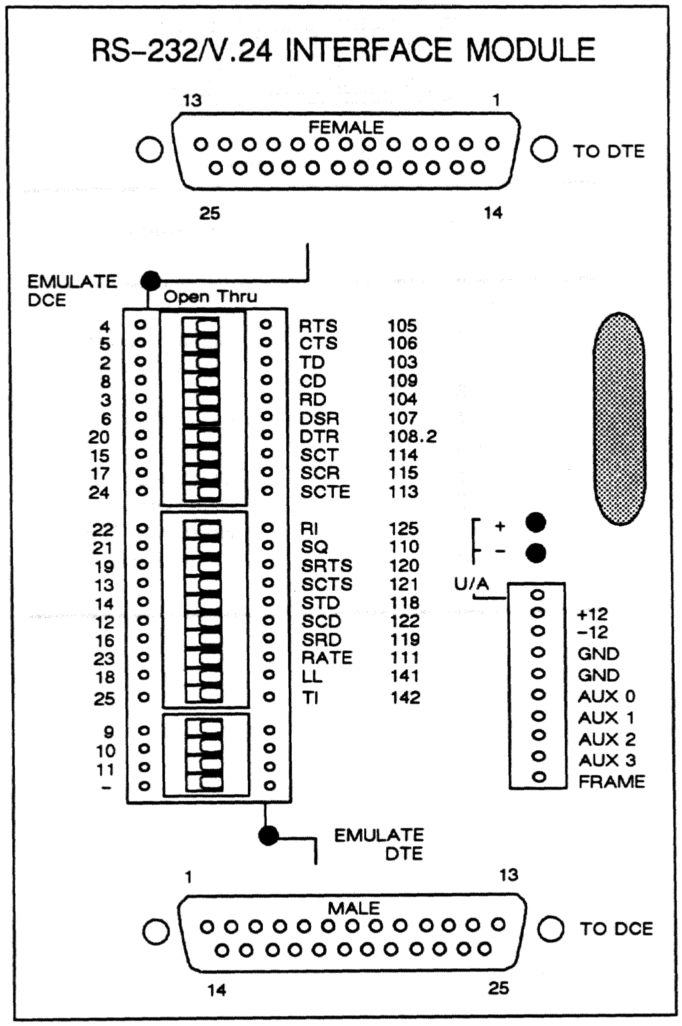


Figure I-1 RS-232/V.24 Interface Module.

Table I-5
RS-232 Test Interface Module

Pin No.	Pin Name	Monitor Mode	Signal Description: To DCE (Em DTE) (DB-25, male)	Signal Description: To DTE (Em DCE) (DB-25, female)
1	Frame Ground	Frame Ground	Frame Ground	Frame Ground
2	TD	High Impedance Input	RS-232/V.24 Output	RS-232/V.24 Input
3	RD	High Impedance Input	RS-232/V.24 Input	RS-232/V.24 Output
4	RTS	High Impedance Input	RS-232/V.24 Output	RS-232/V.24 Input
5	CTS	High Impedance Input	RS-232/V.24 Input	RS-232/V.24 Output
6	DSR	High Impedance Input	RS-232/V.24 Input	RS-232/V.24 Output
7	Signal Ground	Signal Ground	Signal Ground	Signal Ground
8	CD	High Impedance Input	RS-232/V.24 Input	RS-232/V.24 Output
9	—	Test Point	Test Point	Test Point
10	—	Test Point	Test Point	Test Point
11	—	Test Point	Test Point	Test Point
12	SCD	Test Point	Test Point	Test Point
13	SCTS	Test Point	Test Point	Test Point
14	STD	Test Point	Test Point	Test Point
15	SCT	High Impedance Input	RS-232/V.24 Input	RS-232/V.24 Output
16	SRD	Test Point	Test Point	Test Point
17	SCR	High Impedance Input	RS-232/V.24 Input	RS-232/V.24 Output
18	LL	Test Point	Test Point	Test Point
19	SRTS	Test Point	Test Point	Test Point
20	DTR	High Impedance Input	RS-232/V.24 Output	RS-232/V.24 Input
21	SQ	Test Point	Test Point	Test Point
22	RI	Test Point	Test Point	Test Point
23	DSRS	Test Point	Test Point	Test Point
24	SCTE	High IMpedance Input	RS-232/V.24 Output	RS-232/V.24 Input
25	TI	Test Point	Test Point	Test Point

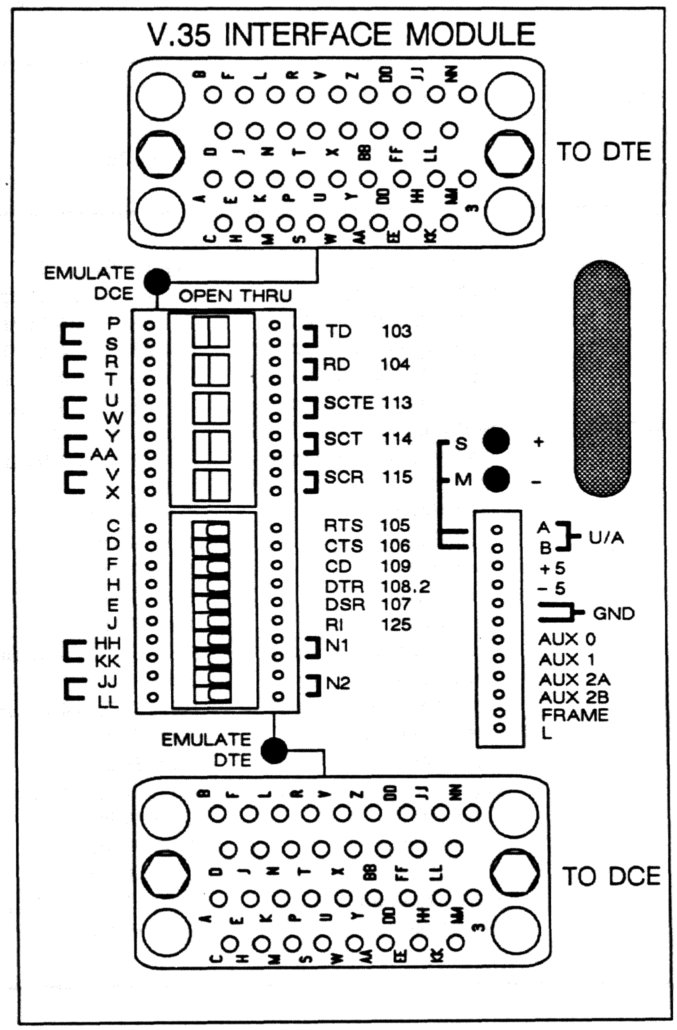


Figure I-2 V.35 Interface Module.

Table I-6
V.35 Test Interface Module

Pin No.	Circuit	Signal	Monitor Mode	To DTE (Em DCE) (34-pin, female)	To DCE (Em DTE) (34-pin, female)
A	101	Frame Ground	Frame Ground	Frame Ground	Frame Ground
B	102	Signal Ground	Signal Ground	Signal Ground	Signal Ground
C	105	RTS	High Impedance Input	V.35 Input	V.35 Output
D	106	CTS	High Impedance Input	V.35 Output	V.35 Input
E	107	DSR	High Impedance Input	V.35 Output	V.35 Input
F	109	CD	High Impedance Input	V.35 Output	V.35 Input
H	108	DTR	High Impedance Input	V.35 Input	V.35 Output
J	125	RI	Test Point	Test Point	Test Point
R	104A	RD	High Impedance Input	V.35 Output	V.35 Input
T	104B				
V	115A	SCR	High Impedance Input	V.35 Output	V.35 Input
X	115B				
Y	114A	SCT	High Impedance Input	V.35 Output	V.35 Input
AA	114B				
P	103A	TD	High Impedance Input	V.35 Input	V.35 Output
S	103B				
U	113A	SCTE	High Impedance Input	V.35 Input	V.35 Output
W	113B				
K	F1	—	—	—	—
M	F1	—	—	—	—
L	F2	Test Point	Test Point	Test Point	Test Point
N	F2	—	—	—	—
Z	F3	—	—	—	—
BB	F3	—	—	—	—
CC	F4	—	—	—	—
EE	F4	—	—	—	—
DD	F5	—	—	—	—
FF	F5	—	—	—	—
HH	N1	{ Test Point	Test Point	Test Point	Test Point
KK	N1	Test Point	Test Point	Test Point	Test Point
JJ	N2	{ Test Point	Test Point	Test Point	Test Point
LL	N2	Test Point	Test Point	Test Point	Test Point
MM	F	—	—	—	—
NN	F	—	—	—	—

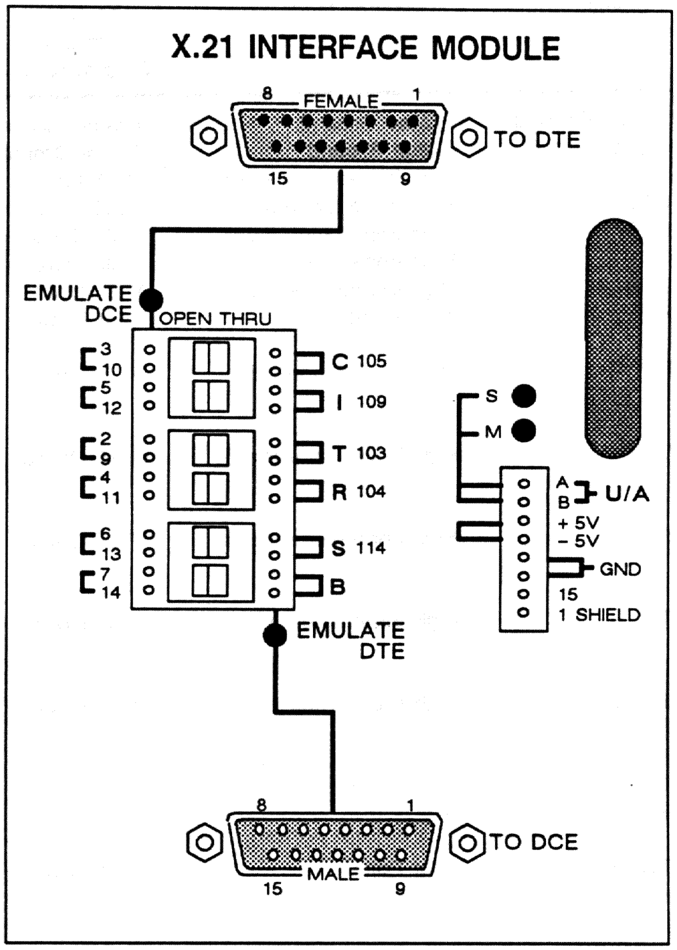


Figure I-3 X.21 Interface Module.

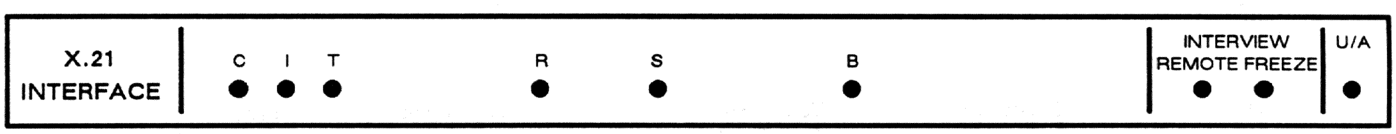


Figure I-4 X.21 LED Overlay.

Table I–7
X.21 Test Interface Module

Pin No.	Circuit	Pin ID	Pin Name	Monitor Mode	Signal Description: To DCE (Em DTE) (15 Pin, male)	Signal Description: To DTE (Em DCE) (15 Pin, female)
1	—	Shield	Shield	Frame Ground	Frame Ground	Frame Ground
2,9	103	T	Transmit Data	High Impedance Input	X.21 Output	X.21 Input
3,10	105	C	Control	High Impedance Input	X.21 Output	X.21 Input
4,11	104	R	Receive Data	High Impedance Input	X.21 Input	X.21 Output
5,12	109	I	Indicator	High Impedance Input	X.21 Input	X.21 Output
6,13	114	S	Signal Timing	High Impedance Input	X.21 Input	X.21 Output
7,14	—	B	Byte Strobe	High Impedance Input	X.21 Input	X.21 Output
15	—	—	—	Test Point	Test Point	Test Point
Patch Panel:		U/A A,B*		High Impedance Differential Input	High Impedance Differential Input	High Impedance Differential Input
		+5V		Output	Output	Output
		-5V		Output	Output	Output
		GND	Ground	Ground	Ground	Ground

* UA A and B can be used for balanced or unbalanced signals. (Do not connect B when you are looking at unbalanced signals.)

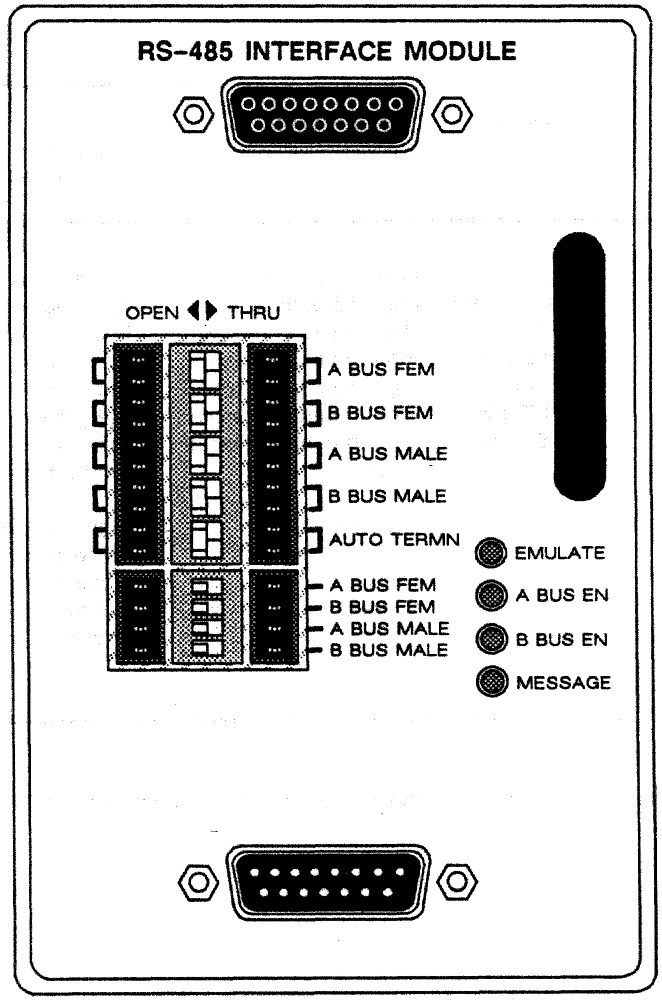


Figure I-5 RS-485 Interface Module.

Table I-8
RS-485 Test Interface Module

				Signal Description	
Pin No.	Circuit	Pin ID	Monitor Mode	15 Pin, Male	15 Pin, Female
1	—	Shield	Frame Ground	Frame Ground	Frame Ground
2	103	A Bus +	High Impedance Input	RS-485 Output	RS-485 Input
9	103	A Bus -	High Impedance Input	RS-485 Output	RS-485 Input
3,10	—	—	—	—	—
4	104	B Bus +	High Impedance Input	RS-485 Input	RS-485 Output
11	104	B Bus -	High Impedance Input	RS-485 Input	RS-485 Output
5,12	—	—	—	—	—
6,13	—	—	—	—	—
7,14	—	—	—	—	—
15	—	—	—	—	—

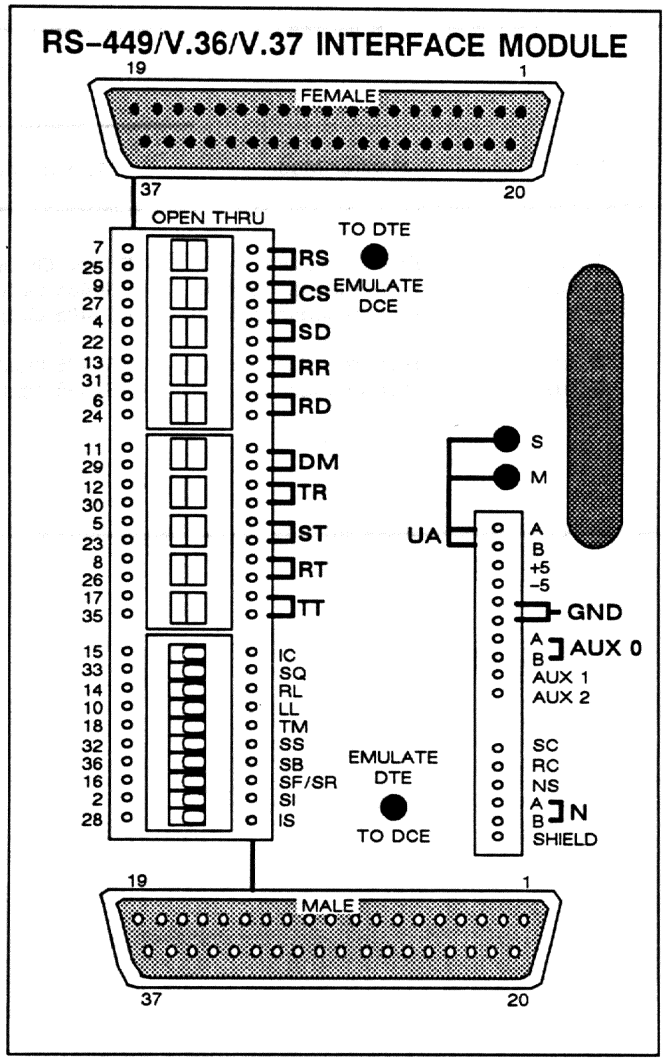


Figure I-6 RS-449/V.36/V.37 Interface Module.

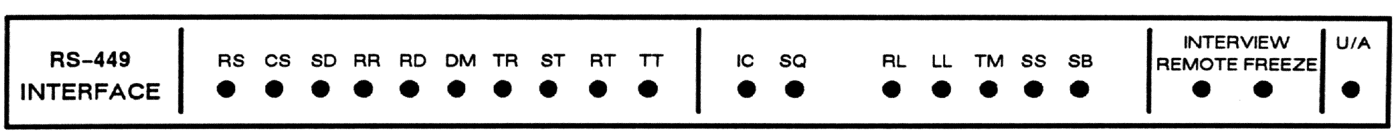


Figure I-7 RS-449/V.36/V.37 LED Overlay.

Table I-9
RS-449/V.36/V.37 Test Interface Module

Pin No.	Pin ID	Pin Name	Monitor Mode	Signal Description: To DCE (Em DTE) (36 Pin, male)	Signal Description: To DTE (Em DCE) (36 Pin, female)
2	SI	Signalling Rate Indicator	Test Point	Test Point	Test Point
4,22	SD	Send Data	High Impedance Input	RS-449 Output	RS-449 Input
5,23	ST	Send Timing	High Impedance Input	RS-449 Input	RS-449 Output
6,24	RD	Receive Data	High Impedance Input	RS-449 Input	RS-449 Output
7,25	RS	Request to Send	High Impedance Input	RS-449 Output	RS-449 Input
8,26	RT	Receive Timing	High Impedance Input	RS-449 Input	RS-449 Output
9,27	CS	Clear to Send	High Impedance Input	RS-449 Input	RS-449 Output
10	LL	Local Loopback	High Impedance Input	Test Point	Test Point
11,29	DM	Data Mode	High Impedance Input	RS-449 Input	RS-449 Output
12,30	TR	Terminal Ready	High Impedance Input	RS-449 Output	RS-449 Input
13,31	RR	Receiver Ready	High Impedance Input	RS-449 Input	RS-449 Output
14	RL	Remote Loopback	High Impedance Input	High Impedance Input	High Impedance Input
15	IC	Incoming Call	High Impedance Input	High Impedance Input	High Impedance Input
16	SF/SR	Select Frequency/ Signaling Rate Selector	Test Point	Test Point	Test Point
17,35	TT	Terminal Timing	High Impedance Input	RS-449 Output	RS-449 Input
18	TM	Test Mode	High Impedance Input	High Impedance Input	High Impedance Input
19	SG	Signal Ground	Signal Ground	Signal Ground	Signal Ground
28	IS	In Service	Test Point	Test Point	Test Point
32	SS	Select Standby	High Impedance Input	High Impedance Input	High Impedance Input
33	SQ	Signal Quality	High Impedance Input	High Impedance Input	High Impedance Input
36	SB	Standby Indicator	High Impedance Input	High Impedance Input	High Impedance Input
Auxiliary Patch Panel:					
	UA A,B	Unassigned Input	High Impedance Input	High Impedance Input	High Impedance Input
	+5	+5 Volts	Output	Output	Output
	-5	-5 Volts	Output	Output	Output
19	GND	Ground	Signal Ground	Signal Ground	Signal Ground
	AUX0 A,B	Auxiliary	Output	Output	Output
	AUX1	Auxiliary	Output	Output	Output
	AUX2	Auxiliary	Output	Output	Output
37	SC	Send Common	Send Common	Send Common	Send Common
20	RC	Receive Common	Receive Common	Receive Common	Receive Common
34	NS	New Signal	Test Point	Test Point	Test Point
3,21	N A,B	National A, B	Reserved	Reserved	Reserved
1	SHIELD	Shield	Frame Ground	Frame Ground	Frame Ground

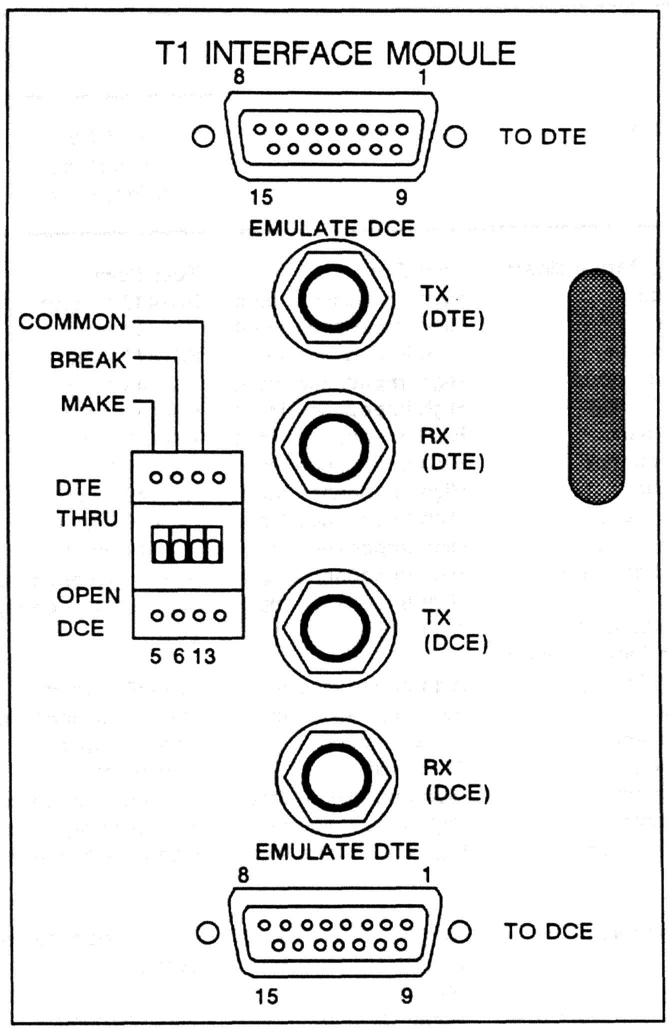


Figure I-8 T1 Interface Module.

Table I-10
T-1 Test Interface Specifications [1]

Pin No.	Pin Name	Signal Description: Monitor	To DCE (Em DTE)	To DTE (Em DCE)
1	Send Data Tip	High Impedance Input	Output	Input
2	Frame Ground	Ground	Ground	Ground
3	Receive Data Tip	High Impedance Input	Input	Output
5	Remote Test Make	High Impedance Input	Test Point	Test Point
6	Remote Test Break	High Impedance Input	Test Point	Test Point
9	Send Data Ring	High Impedance Input	Output	Input
11	Receive Data Ring	High Impedance Input	Input	Output
13	Remote Test Common	High Impedance Input	Test Point	Test Point

(1) Unlisted connectors are wired 1-for-1 through the two connectors. Test points are connected to switches and test points only.

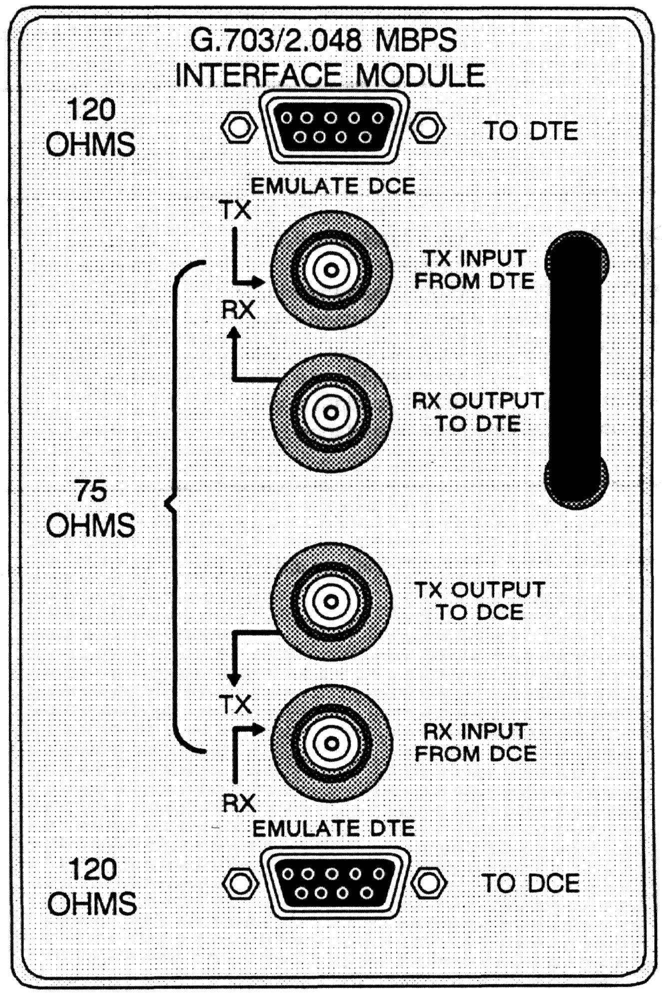


Figure I-9 G.703 Interface Module.

Table I-11
G.703 Test Interface Specifications [1]

		Signal Description		
Pin No.	Pin Name	Monitor	To DCE (Em DTE)	To DTE (Em DCE)
Pin No.	Pin Name	Monitor	To DCE (Em DTE)	To DTE (Em DCE)
1	Receive Data Tip	High Impedence Input	Input	Output
2	Frame Ground	Ground	Ground	Ground
5	Send Data Tip	High Impedence Input	Output	Input
6	Receive Data Ring	High Impedence Input	Input	Output
9	Send Data Ring	High Impedence Input	Output	Input

(1) Unlisted connectors are wired 1-for-1 through the two connectors.

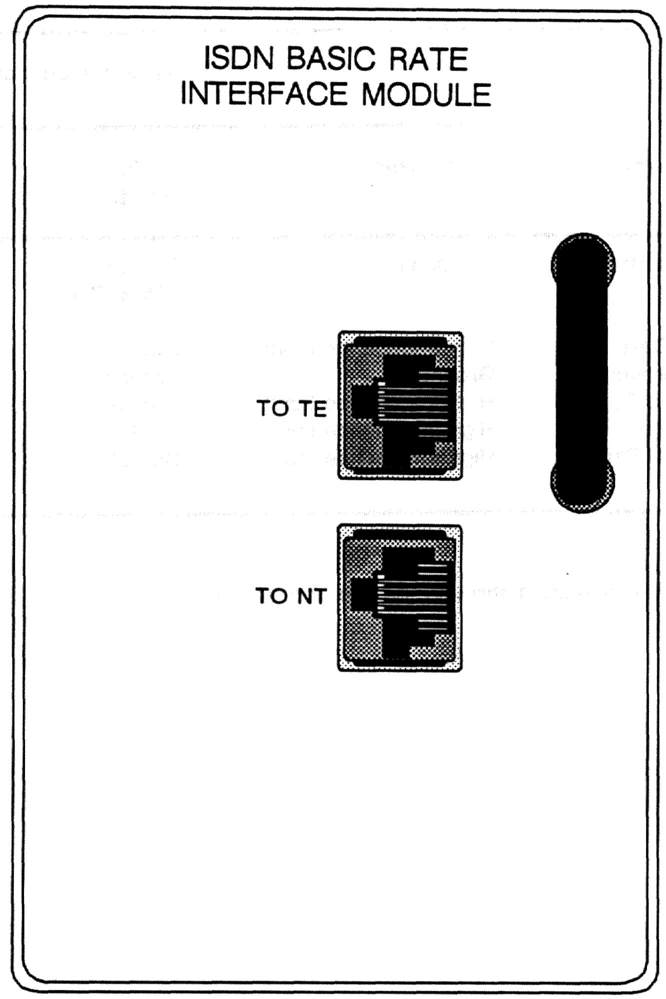


Figure I-10 ISDN Interface Module.

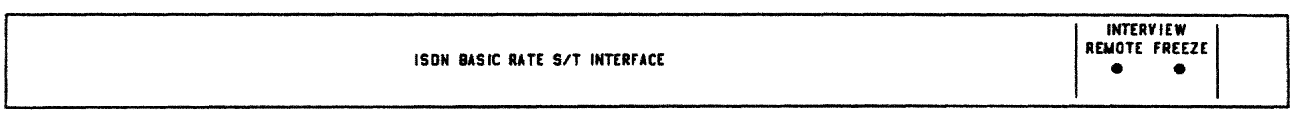


Figure I-11 ISDN LED overlay.

Table I-12
ISDN Test Interface Specifications [1]

Pin No.	Pin Name	Signal Description: Monitor	To DCE (Em DTE)	To DTE (Em DCE)
3	Send Data Tip	High Impedence Input	Output	Input
4	Receive Data Tip	High Impedence Input	Input	Output
5	Receive Data Ring	High Impedence Input	Input	Output
6	Send Data Ring	High Impedence Input	Output	Input
7 [2]	- voltage	Output	Output	Output
8 [2]	+ voltage	Output	Output	Output

(1) Unlisted connectors are wired 1-for-1 through the two connectors.

(2) Pins 7 and 8 have a voltage differential of 40o volts; see ISO 8877 (1987-08-15) and CCITT I.430.

Appendix J: Field Service on the INTERVIEW 7000 Series

This appendix is to guide you in proper removal, handling, and installation of logic boards and other components in the INTERVIEW 7000 Series.

- J1 alerts you to the problem of static electricity.
- J2 covers the removal of logic cards.
- J3 discusses the installation of logic cards.
- J4 covers the installation of the optional multiplexer boards.
- J5 discusses the replacement of firmware on the CPM board.
- J6 covers the installation of a hard disk drive in the INTERVIEW 7000 and 7200 *TURBO*, option OPT-951-01-1.
- J7 covers other components requiring attention.

Appendix J1: Eliminating Static Electricity

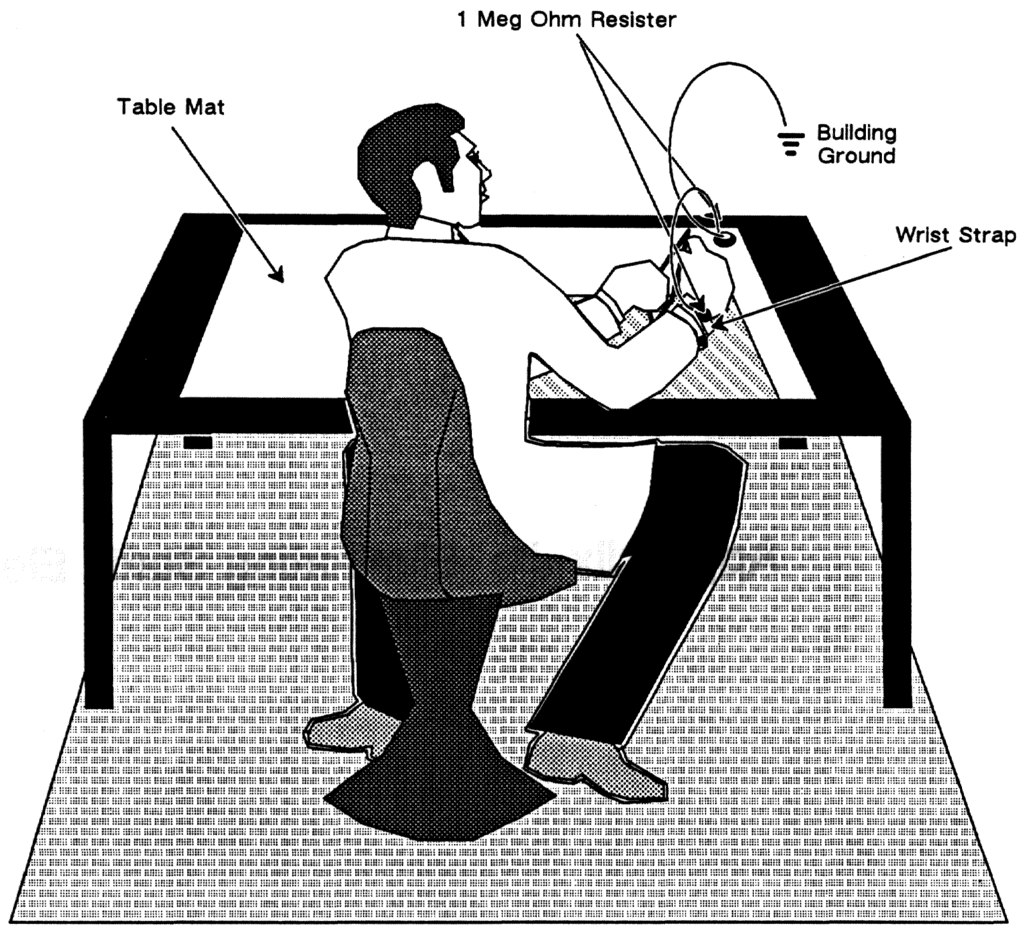


Figure J1-1 Illustration of grounded workstation. Note grounded wrist strap and table mat.

Appendix J1: Eliminating Static Electricity

STATIC ELECTRICITY CAN DAMAGE THE UNIT WHEN THE COVER IS REMOVED. Before you begin to remove the cover, be certain you have taken appropriate anti-static precautions.

J1.1 Take Precautions

As a *minimum* precaution, a grounded wrist strap should be used in conjunction with an anti-static work surface mat. Without these precautions, walking (or just shifting your feet) on a carpet or tile floor, shifting your position in a chair, or simply rolling a chair as little as a foot or two can generate sufficient static electricity to damage circuitry. See Figure J1-1 for an example of a static-free workstation for working with sensitive parts and assemblies.

Place unit on anti-static mat with power OFF. Then put on a grounded wrist strap and proceed.

J1.2 Use of Anti-Static Packing

When returning any boards to the AR Division factory, reuse any bags and anti-static packaging from boards sent by the AR Division previously. Pack the parts tightly to prevent motion which could generate static. If you did not save the packaging, you *must* obtain special anti-static packing before you begin or you will damage the components on the board.

Appendix J2: Removing Logic Boards

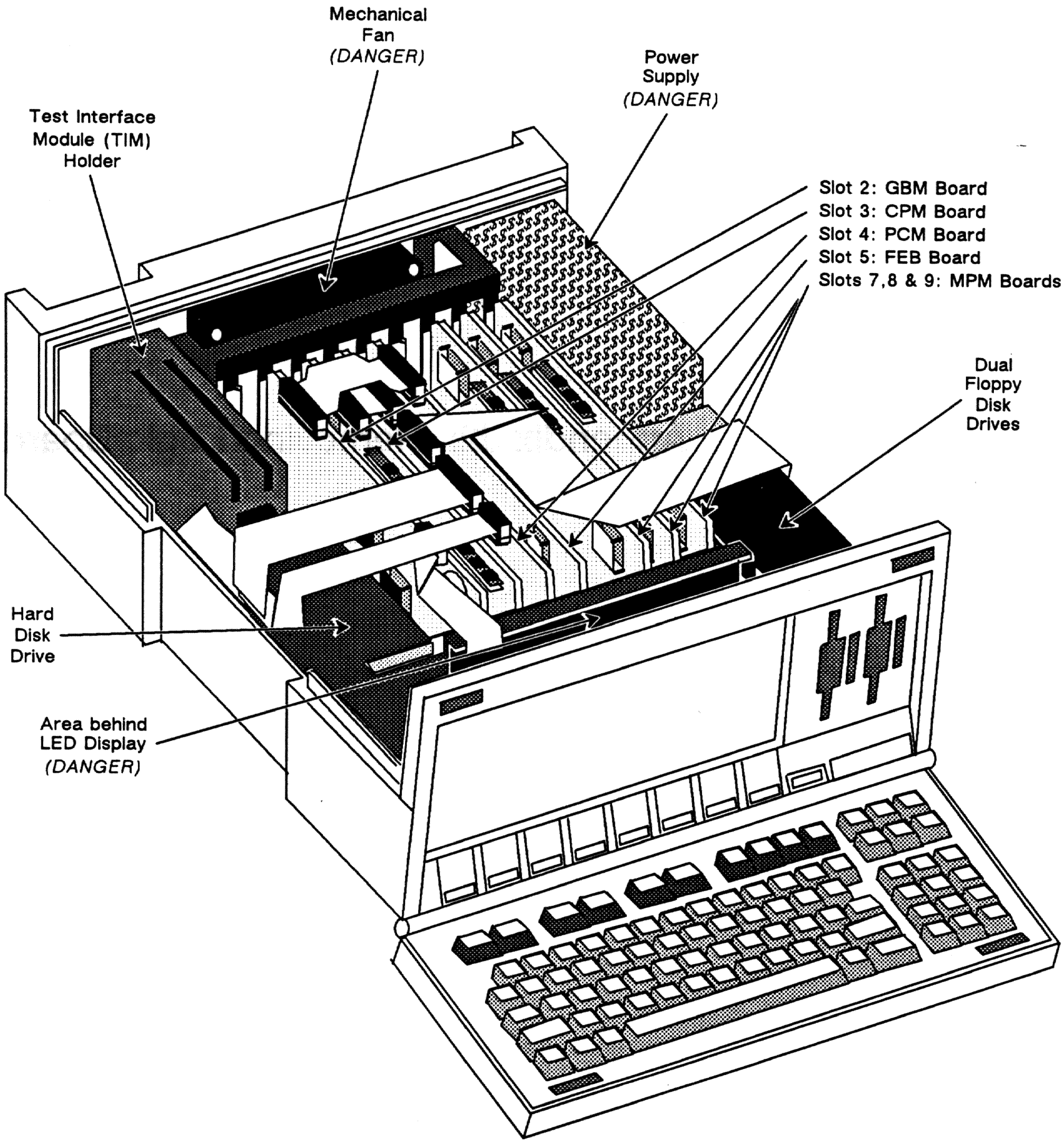


Figure J2-1 Viewing interior components of the INTERVIEW 7000 Series.

Appendix J2: Removing Logic Boards

CAUTION: STATIC ELECTRICITY CAN DAMAGE THE UNIT WHEN THE COVER IS REMOVED. Before you begin this section, you must take the proper anti-static precautions given in Appendix J1.

J2.1 Remove the Cover

To remove the cover of your unit, first unplug all connecting cords on the back of the INTERVIEW. Place the unit on its back and remove the six long screws recessed in the base.

Place the unit in its working position, lying on its base. Open the front panel, sliding the top two blue latches back. At this point the hooks of the latches are exposed out the front of the unit. Press down slightly on the recessed circle of these latches and continue to slide the latches inside the unit until they stop. The indented circle should be almost centered in the sliding area and the hooks of the latches are no longer visible from the front of the unit. *These latches must be properly placed or they will break when you remove the cover.*

Keeping an eye on the position of the latches, grasp the sides of the cover and remove it. The handle on the left side and rubber bumper feet on the right side can help to give you some leverage. You may have to rock the cover slightly back and forth to free it. Replace the handle in its slot on the base.

J2.2 View the Interior Layout

(A) Anticipate Potential Hazards

The inside of your INTERVIEW reveals three potentially hazardous areas. **DO NOT TOUCH THESE AREAS OR INJURY MAY RESULT!** These three danger areas include the mechanical fan at the center back of the unit (physical hazard), the power supply at the back right side of the unit (230V shock hazard), and the area directly behind the LED display (190V shock hazard). See Figure J2-1 for their locations.

Take care as you remove the boards not to make contact with any of these three areas.

(B) View Interior Components of the INTERVIEW 7000 and 7200 *TURBO*

In addition to these three potentially hazardous components, on the left near the front of your unit is a bracket for a Winchester hard disk drive. The standard INTERVIEW 7000 does not have a hard drive, but the bracket to hold it is in position for the option of adding a hard drive. If your INTERVIEW 7000 has been upgraded with this option, you will have two cables running up the side of your hard drive and over it, connecting it to the PCM board.

Immediately behind the hard disk drive is the TIM (Test Interface Module) holder. The dual floppy disk drives are on the right near the front, with a cable connecting them to the PCM board. In the center of the unit are the nine slots which house your boards.

Reading from left to right, the slots contain the following boards:

- slot 1—empty (unless upgraded with optional multiplexer board)
- slot 2—GBM, Global Bus Module Board
- slot 3—CPM, 68K Processor Board
- slot 4—PCM, 68K Peripheral Board
- slot 5—FEB, Front End Buffer Board
- slot 6—reserved for XDRAM board (OPT-951-23-1)
- slot 7—empty (unless upgraded with optional MPM board)
- slot 8—empty (unless upgraded with optional MPM board)
- slot 9—MPM, 286 Processor Board

There is one cable connecting the LED display to the GBM board. Additionally, there are two jumper cables; one connecting the FEB and GBM boards and a smaller one connecting the CPM and PCM boards.

(C) View Interior Components of the INTERVIEW 7500 and 7700 *TURBO*

In addition to these three potentially hazardous components, on the left near the front of your unit is the Winchester hard disk drive. It has two cables running up its side and over it, connecting it to the PCM board.

Immediately behind the hard disk drive is the TIM (Test Interface Module) holder. The dual floppy disk drives are on the right near the front, with a cable connecting them to the PCM board. In the center of the unit are the nine slots which house your boards.

Reading from left to right, the slots contain the following boards:

- slot 1—empty (unless upgraded with optional multiplexer board)
- slot 2—GBM, Global Bus Module Board
- slot 3—CPM, 68K Processor Board
- slot 4—PCM, 68K Peripheral Board
- slot 5—FEB, Front End Buffer Board
- slot 6—reserved for XDRAM board (OPT-951-23-1)
- slot 7—MPM, 286 Processor Board
- slot 8—MPM, 286 Processor Board
- slot 9—MPM, 286 Processor Board

There is one cable connecting the LED display to the GBM board. Additionally, there are two jumper cables; one connecting the FEB and GBM boards and a smaller one connecting the CPM and PCM boards.

J2.3 Remove the Boards

As stated previously in Section J1.2, if you have bags and packaging from boards sent by the AR Division previously, reuse those anti-static materials for packing. If you did not save the packaging, you *must* obtain a special anti-static packing before you begin or you will damage the components on the board.

(A) Disconnect the Cables

Make certain the unit is on an anti-static mat and you are wearing your grounded wrist strap.

You may wish to record where each cable was attached for reference when your replacement boards arrive. Refer back to Figure J2-1. Your upgraded replacement boards will be sent as soon as possible.

Notice that several of the connectors are "keyed" for easy, correct alignment. Those that are not "keyed" need to have special care taken to line up the pins with the proper connecting holes when they are reconnected.

Disconnect the cables *carefully*. They may be easily removed with an IC clip holder. Lift the connector straight up, holding onto its edges. DO NOT PULL ON THE CABLE; you could break it or damage the connectors. If you do not have an IC clip holder, a blade screwdriver will also work.

> *CAUTION: If you use a screwdriver, use the top of the black mounting rack (which secures the ends of the boards) as your leverage point. Avoid using the edges of the boards or other components to pry the connector loose.*

(B) Remove the Boards

CAUTION: Do not hold any board by its gold edge connector. Hold it by the sides, NEVER touching the components.

It is probably easiest to remove the boards from left to right. To do so, grasp the board by its top corner edges, and *gently* pull it straight up and out. Then, holding the board by its side edges, place it **immediately** into an anti-static bag and close the bag. Repeat the process with each board to be removed.

J2.4 Replace the Cover

If you are not going to replace any boards at this time, continue with this section to replace the cover to protect the remaining components. If you are ready to replace boards, proceed down to Section J2.5, Package the Boards, and then on to Section J3, Installing Logic Boards.

Grasp the cover with the latch area to the front facing you. Make certain the latches are recessed as far as they can go into the cover and their hooks are not beyond the edge of the cover. Slide the cover down. For the INTERVIEW 7500 and 7700 *TURBO*—and those INTERVIEW 7000s and 7200 *TURBO*s having the optional hard drive—take care that the two cables to the Winchester hard disk drive (on your left, directly behind the handle) are not being pinched or damaged by the cover. (Some models may contain a small, removable, protective plastic sheet to help shield the cables from the cover.)

Place the unit on its back once again and replace the six screws to hold the cover secure.

J2.5 Package the Boards

In preparation for storage or shipping, package each board removed from the INTERVIEW in its own anti-static bag. If shipping, wrap it further in anti-static packing material and place it securely in a shipping box. You may package several boards in a single box, as long as *each board is wrapped individually in an anti-static bag*. When returning a board, please reference the *return authorization number* from your original packing list. For shipping information, see Appendix E, Communications with the AR Division Factory.

Appendix J3: Installing Logic Boards

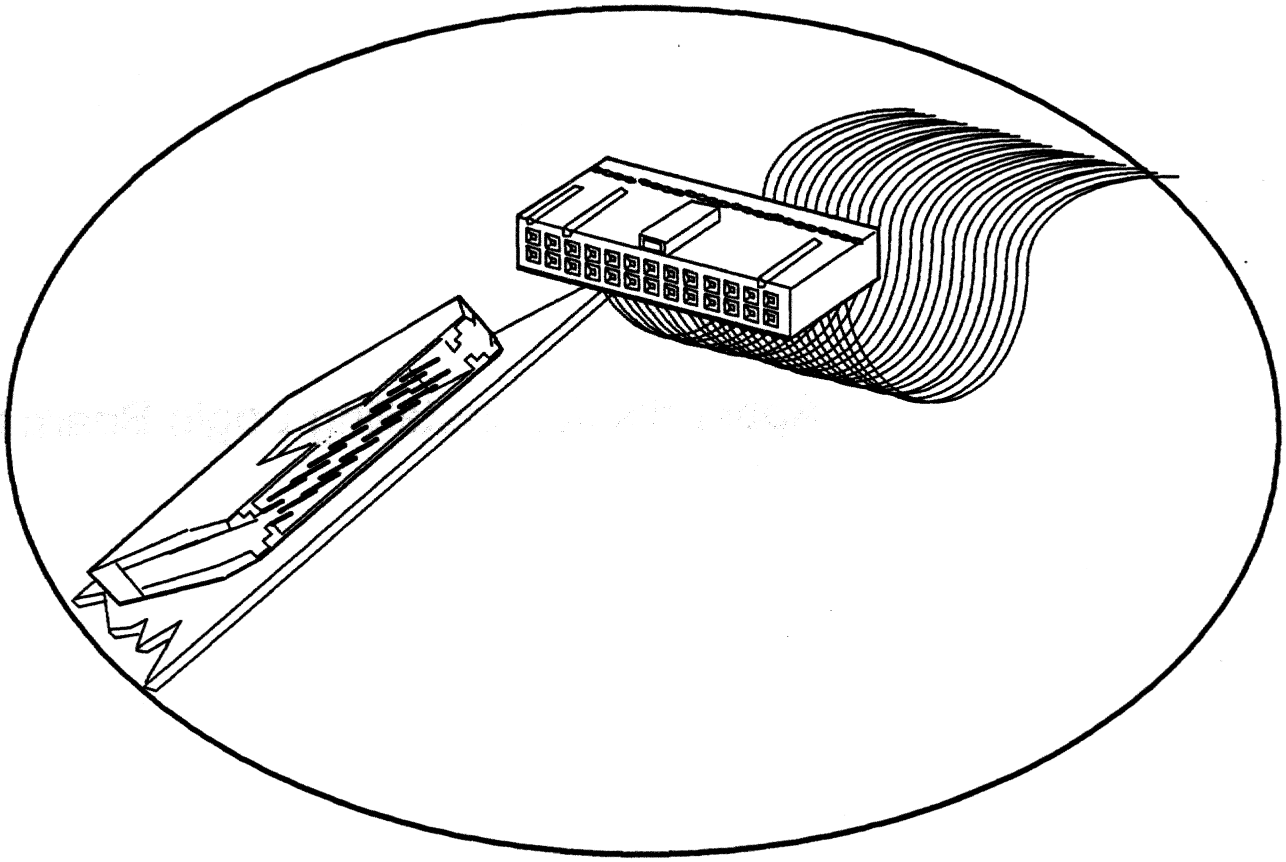


Figure J3-1 Some connectors are "keyed" for correct alignment.

Appendix J3: Installing Logic Boards

Observe the proper handling of the boards: refer to Appendix J1 and J2 for using proper anti–static precautions and removing the boards.

Figure J2-1 gave you visual locations of the components in the INTERVIEW 7000 Series. For a detailed listing of board placement in the slots, see Section J2.2. Each board is labeled in its upper left corner. The motherboard (on the floor of the unit) is labeled with the corresponding name of the board by the slot into which that particular board is set.

J3.1 INTERVIEW 7000 Series Hardware Architecture

In troubleshooting any malfunction, it is important to know which board contains the component that controls the particular function.

The INTERVIEW contains six types of board which are connected to the motherboard and which can be easily removed and replaced. (See Appendix J2 and J4 for board removal information.) These boards are:

- MUX Multiplexer Board (if upgraded with option)
- GBM Global Bus Module
- CPM Control Processor Module
- PCM Peripheral Control Module
- FEB Front–End Buffer
- MPM Main Processor Module

A seventh type of board is installed in the Test Interface Module.

Figure J3-2 is a block diagram showing the components on all of these boards. The figure also shows how the components are interconnected in the unit. Following the figure is a descriptive listing of the components.

> NOTE: The symbol ▧ in the diagram represents a bi–directional buffer.

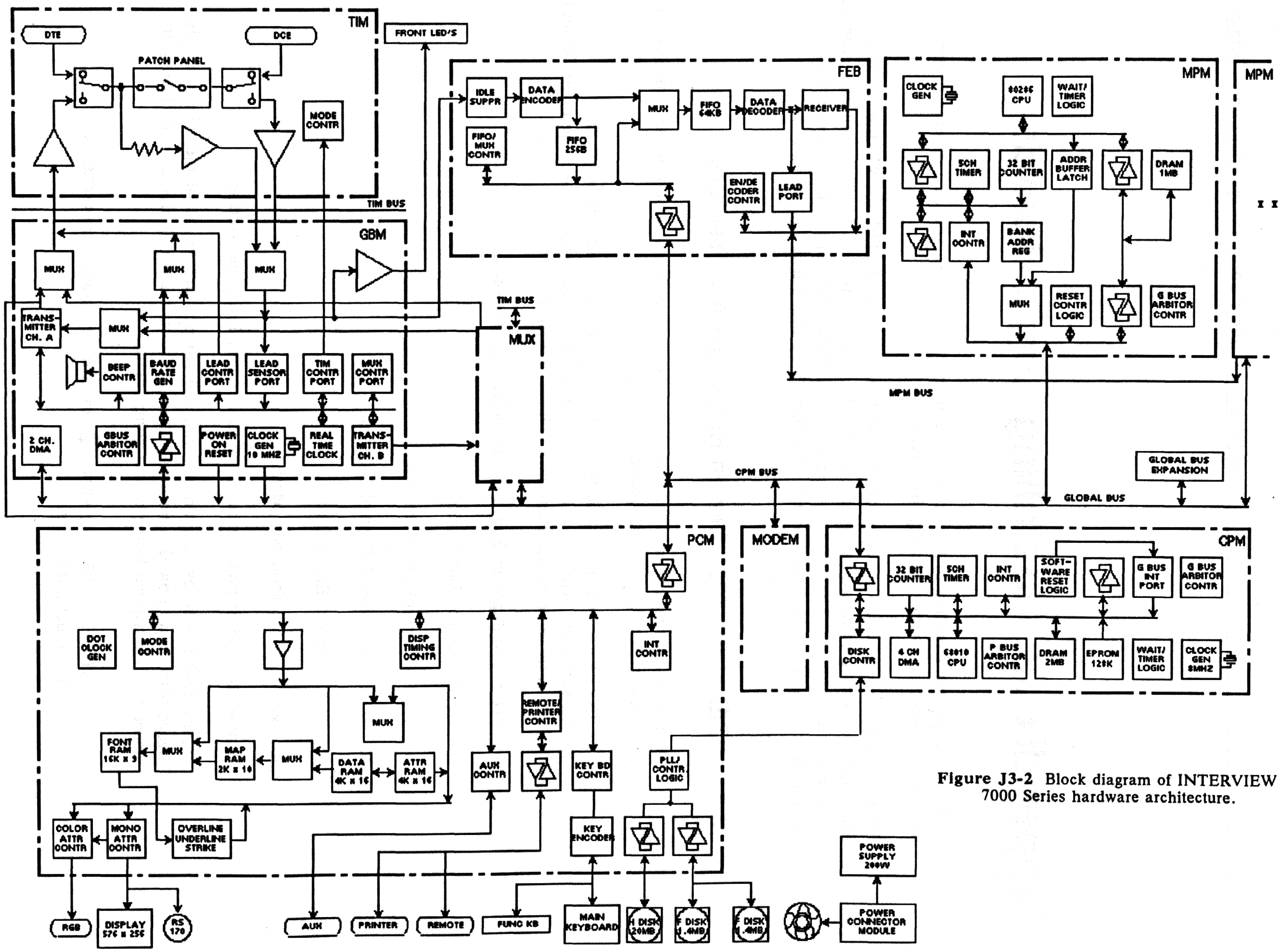


Figure J3-2 Block diagram of INTERVIEW 7000 Series hardware architecture.

(A) MUX (Multiplexer Board)

This board provides whatever additional processing is needed for special data formats such as T1, G.703, ISDN, etc.

The mux board clocks–in data from transmitter channel A on the GBM. Once at the mux board, the data is processed as required and then sent back to the GBM to be transmitted over the line.

(B) GBM (Global Bus Module)

1. *Mux*. The muxes multiplex various signals to be transmitted, received, or monitored depending upon which mode of operation is being used.
2. *Transmitter channel A*. The transmitter generates the data stream to be sent out when the INTERVIEW is in emulate DTE mode. The data can either be sent out directly to the TIM or sent to the mux board for further processing.
3. *Beep control*. The beep control does just that—it controls the beep. There are two types of beeps defined by their length—one second and one–half second. Only the half–second beep is currently in use.
4. *Baud rate generator*. The baud rate generator generates the clock for the transmit data when in emulate DTE mode.
5. *Gbus arbitor control*. This arbitrates requests by the GBM to obtain the global bus. When the bus is idle and the GBM has top priority over all other requests, this logic will enable the GBM to access the global bus.
6. *Power on reset*. This circuitry generates the proper timing for reset signals for the INTERVIEW unit from a power–up situation.
7. *10 MHz clock generator*. This generates the global bus clock.
8. *Lead control port*. The INTERVIEW writes to this port in order to send out the proper control signals for the data interface. These control signals are dependent upon the emulate mode for which the unit is configured.
9. *Lead sensor port*. The INTERVIEW reads the lead sensor port in order to monitor incoming control signals from the data interface.
10. *TIM control port*. The INTERVIEW writes to this port in order to setup the TIM to the desired mode of operation.
11. *Real time clock*. This circuitry keeps track of current time and date. This information can be displayed on the screen for the user and can also be used to timestamp the incoming data.
12. *Transmitter channel B*. This is a second transmit channel used to send data out into the data stream through the mux board.

13. *Mux control port.* The INTERVIEW software writes to this port to set up the muxes for the proper data flow through the GBM, depending upon the mode of operation.

14. *2 channel DMA.* In order for the INTERVIEW to transmit data in emulate mode, the DMA controller requests a byte of data from memory on an MPM. When this byte is obtained, the DMA controller then sends it out through one of the transmitter channels.

(C) CPM (Control Processor Module)

The CPM board controls most of the operations of the INTERVIEW unit.

1. *Disk control circuitry.* The disk controller is the 9580B. In record mode, the disk controller sends a block of data through the PCM to one of the disk drives. When in playback, the disk controller reads one of the drives through the PCM so that the CPM can send recorded receive data to the FEB. The 9580 controls up to three disk drives—one Winchester hard drive and two floppy drives.

2. *8 MHz clock generator.* This circuitry generates the clock that is used by the 68010 CPU and the peripherals on the CPM.

3. *Global bus interrupt port.* This port can be written to by any board that has access to the global bus. There are four global interrupts that can be set, each of which will cause an interrupt of the 68010 processor. Thus, any board that is connected to the global bus has the ability to interrupt the 68010.

4. *Global bus arbitor.* This circuitry queues requests by the CPM for access to the global bus and waits until priority is obtained before enabling the CPM onto the global bus. Control bits are set to determine the mode of operation of the arbitor. One mode causes the CPM to always release the bus after an access. Another mode causes the CPM to hold onto the global bus until it is requested by another board.

5. *4 channel DMA.* The CPM currently uses only two of the available four DMA channels. Channel zero is used for recording data and channel one is used for playback of data. When recording data, the FEB interrupts the CPM to signify that a byte of receive data is available. The DMA then reads the byte from the FEB into memory on the CPM. When enough bytes have been accumulated to form a block, the disk controller will send the block to the specified disk drive. When in playback mode, the DMA sends out bytes of receive data to the FEB. This is data that has been moved into CPM memory by the disk controller. In addition, the DMA circuitry allows for any memory-to-memory transfers necessary between the CPM and other boards.

6. *Software reset logic.* The CPM board contains logic which enables the 68010 to generate a software reset to the rest of the unit. The reset signal is

generated by executing a RESET instruction in the 68010. This will cause the software reset to be generated without disturbing any of the registers within the 68010.

7. *Interrupt controllers*. The three interrupt controllers monitor interrupts from the disk controller, time-out logic, and global bus. These interrupt controllers generate the appropriate vectors for the CPU's interrupt vector table.

8. *Channel timer*. This function is carried out by the 9513A, a chip used to generate output signals that are used as CPU interrupts. The time intervals for these interrupts are user-programmable.

9. *32 bit counter*. This circuitry counts a 1 MHz clock. The CPU can latch this value at any time and then read the lower 16 bits of the count, as well as the upper 16 bits if they are needed. The 32 (or less) -bit count is used by the software to determine specific time intervals necessary for execution of certain routines.

10. *EPROM*. The EPROM contains the power-up software and initialization routines. When the power is turned on, the 68010 processor begins execution by fetching from the EPROM.

11. *Wait/timer logic*. This logic generates an error interrupt to the CPU if an instruction causes the board to hang up. If the processor does not receive an acknowledgment within approximately .33 milliseconds after it has begun a cycle, an interrupt will be generated.

12. *P bus arbitor*. The P bus (CPM bus on block diagram) connects the CPM directly to the PCM and the FEB. Thus, the CPM can directly read and write to ports on any of these boards without accessing the global bus. The arbitor is simply a pair of pals which decode the address from the CPU. If the address is one of the ports on the PCM or FEB, and an acknowledgment has been received from the appropriate board, the data buffers are enabled.

13. *DRAM*. The DRAM is 2 Mbyte of memory space that contains the operating software for the INTERVIEW unit. At power up, this software is loaded into the DRAM from either the hard or floppy disks. In addition, any programs that are compiled for run mode are stored in the DRAM.

14. *68010 processor*. This processor is the brain of the INTERVIEW unit. It controls virtually every operation of the box. The 68010 processor operates using the software stored in the DRAM on the CPM.

(D) PCM (Peripheral Control Module)

The PCM board provides the interface to all the major peripherals in the INTERVIEW unit, as well as external peripherals attached to the unit.

1. *Dotclock generator.* The dotclock is generated by buffering a 23.9114 MHz oscillator and then dividing it by two. This gives a dotclock frequency of 11.9557 MHz.
2. *Aux controller.* The aux control function is performed by the 8536 chip on the PCM. (This chip is also known as a CIO.) This part enables the unit to read or write to other nodes through a DB-25 connector located on the back of the unit. The 8536 has two 8-bit ports, and each line may be configured as an input or an output. When configured as an input, the 8536 can be programmed to look for a specific condition on one bit or search for an entire word, and then generate an interrupt.
3. *Remote/printer controller.* The control functions for the remote port and the printer port are performed by the 8530A chip on the PCM. (This chip is also known as the SIO.) The SIO provides a serial interface between the INTERVIEW unit and other nodes. It receives serial data from external sources, strips off any flags or block-check characters, and causes an interrupt when it has an 8-bit word of data to be read out of it. On the transmit side, the SIO takes a word of data from the processor on the CPM, adds the necessary flags and block-check characters, and sends out the data over the serial interface.

   The SIO has two available ports on the PCM—one is used for the printer interface and the other is used for the remote interface. For the printer interface, the SIO acts as a DCE. It accepts TD as an input and generates RD as an output. It also generates the appropriate RS-232 handshake control signals. For the remote interface, the SIO acts as a DTE, sending out TD data to the remote port and receiving RD data from the remote port. The remote port is used to control the INTERVIEW unit from a remote terminal instead of from the keyboard.
4. *Display timing controller.* The display timing control functions are performed by the Signetics 2674 chip. During initialization, the CPM programs this chip with information about the display device, including the size of each character in dots, the number of characters per line, the number of lines per screen, and the size of the horizontal and vertical blanking intervals and sync pulses. Also programmed into the chip is the initial location and block size of the character data block in RAM. When the 2674 runs, it automatically generates the proper timing of character data, blanking intervals, and sync pulses. In addition, it also generates control signals which allow for overline, underline, and strikethrough.
5. *Attribute RAM and mux.* The attribute RAM takes a 12 bit address and generates 16 bits of output which define all attributes of a particular

character. The 12 bit address can come from two possible sources. When the display is scanning, data is being read from the attribute RAM and the address comes from a buffer attached to the output of the display controller. When writing to the display, data is being written into the attribute RAM and the address comes from a latch which stores the desired write location as output from the processor on the CPM. Attributes of a character include underline, overline, strikethrough, reverse image, blinking, blanking, low intensity (monochrome only), background color, foreground color, and data type (hex or ASCII).

6. *Data RAM.* The data RAM takes the same 12 bit address as the attribute RAM and generates 12 bits of output which provide character mapping information. The address is multiplexed from either the display controller (read) or the address latch (write) as described for the attribute RAM. One bit of each output word is dedicated as a flag to designate that a particular character may not be displayed in hex format. (These are usually flags, block-check, or control characters.) Three bits of the output represent 8 possible character formats (i.e. ASCII, EBCDIC, hex, etc.) The other eight bits represent 256 possible characters within each character format.

7. *Mapping RAM and mux.* The mapping RAM takes a 10 bit address and generates 10 bits of output which provide the location of a specific character dot pattern in the font RAM. The 10 bit address is multiplexed from two sources. When the display is scanning (read), the address comes from the output of the data RAM. When writing mapping information to the mapping RAM, the address comes from a latch which stores the desired write address from the processor on the CPM. The address input defines a specific character and a specific data format. The mapping RAM takes this information and generates an output which points to the location in font RAM where the actual dot pattern is stored.

8. *Font RAM and mux.* The font RAM takes a 14 bit address and generates 9 bits of actual display dot information. The 14 bit address is multiplexed from two sources. When the display is scanning (read), the lower 4 address bits come from a latch connected to the display controller chip and the upper 10 bits come from the mapping RAM. When writing font information to the font RAM, the address comes from a latch which stores the desired write address from the processor on the CPM. Each character on the screen is 9 dots wide by 12 dots high. The lower 4 address bits decode the specific row within the character and the upper 10 bits decode to a possible 1024 actual character dot patterns. The output is a 9–bit word which gives the actual on–off pattern of dots in a particular row of a given character.

9. *Overline, strikethrough and latch.* This circuitry actually generates the strikethrough and overline attributes by overriding dot information from the font RAM in specific rows of a character. Finally, the resulting dot data is

latched and processed by the video attribute controllers. (Underline is done internally by the attribute controller.)

10. *Monochrome attributes controller.* This function is performed by a 2675 that has been programmed to operate in monochrome mode. The 2675 creates a character clock from the dot clock, such that one character clock is generated for every nine dot clocks. (Each character is nine dots wide.) For each character to be output to the video monitor, the 2675 takes the nine bits of processed dot data, along with control signals for blanking, blinking, underlining, cursor position, and reverse imaging, and creates the serial stream of dot data called monochrome video. This data stream is then sent on through a buffer to the plasma display or through driving transistors to the RS-170 port that can be connected to a CRT.

11. *Color attributes controller.* This function is performed by a 2675 that has been programmed to operate in color mode. For each character to be output to a color monitor, the 2675 takes the nine bits of processed dot data—along with control signals for blanking, blinking, underlining, cursor position, foreground color, and background color—and creates four serial streams of data which will generate the color video. The four different data streams represent red, blue, green, and luminescence. This data passes through a buffer before being sent out to the color monitor through a port in the rear of the unit.

12. *Keyboard controller.* The keyboard controller is an 8051 microcontroller programmed to function like an INTEL 8278. (The 8278 is a keyboard controller that was used until it was obsoleted by the manufacturer.) The controller continually writes out a sequence of addresses that are decoded to represent rows and columns of the main keyboard and function keyboard. The scanning process continues until the controller finds a key depress or key release condition. The controller then interrupts the 68010 processor on the CPM which reads the key value out of the controller.

13. *Key encoder.* The key encoder is simply a combination of buffers and decoders which take the scan address from the keyboard controller and convert it into row and column select signals that go directly to the main keyboard and function keyboard. Each scan address decodes to a unique row and column combination on one of the keyboards.

14. *Interrupt control.* The interrupt control logic combines interrupts from several sources into one interrupt signal that is sent to the CPM board. The CPM then reads a 3-bit code generated by the interrupt logic to determine the source of the interrupt on the PCM. Interrupts from the PCM are caused by key press, key release, break detect on receive data, or interrupts from the 8536 chip (aux controller).

15. *Phase locked loop/disk control logic.* The disk control logic simply takes the disk control signals from the CPM and sends them out to the selected drive,

either floppy or hard. It also buffers control signals from the disk drives and sends them back to the disk controller on the CPM. The 9582 data disk separator recovers clock from a stream of data coming from a disk drive and sends it back to the CPM. The 9582 separator has a built-in phase-locked loop that is used for this clock recovery.

16. *Mode control.* This is simply a latch that is loaded from the CPM. The various bits of this latch control certain aspects of the display. By programming certain bits of this latch, one can force the display to hex format, control the cursor, choose half or full duplex, or program the display controller for operation from a 50 Hz power supply.

(E) FEB (Front-End Buffer)

The Front End Buffer provides the necessary pre-processing for data in the receive path. There are three paths that the data may take through the FEB. In run mode, raw data passes through the FEB to the receiver and gets read by an MPM. The data can also be stored in encoded form on one of the disk drives by the disk controller on the CPM. In playback mode, data does not come from an external device, but from a selected disk drive. In this mode, data from the disk controller is multiplexed into the receive path.

1. *Idle suppress.* When selected, this circuitry removes the idle characters from the receive data stream.

2. *Data encoder.* This circuitry takes the receive data and encodes it in a format that enables easy storage. The encoded format includes data and control lead information along with time ticks.

3. *1024B fifo (small).* The small fifo ("*first in, first out*") stores the encoded data until it can be read by the DMA controller on the CPM board. This path is used when the INTERVIEW is in record mode.

4. *Fifo/mux control.* The CPM writes to this port to set the appropriate control bits for the fifos and to configure the mux for the proper mode of operation.

5. *Mux.* The mux multiplexes between real-time data and playback data in the receive path.

6. *64K fifo (big).* The big fifo acts as a "rubber band" between the FEB and the MPM—it "stretches or shrinks" with the amount of data received and sends it out at the proper rate. The FEB puts data into the fifo as it receives it and waits for the receiver on the FEB to request a byte.

7. *Data decoder.* The data decoder puts the data back into raw form as it appeared before entering the data encoder. The raw data is then passed on to the receiver to be sent out to the MPM. Control lead information is also output from the decoder and sent to the lead port.

8. *Lead port.* The MPM reads this port on the GBM to obtain information on the status of the control leads.

9. *Encoder/decoder control.* The MPM writes to this port on the GBM to set the control bits for the data encoder and data decoder on the board.

10. *Receiver.* When a byte of data has been processed through the decoder, it passes on to the receiver circuitry. The MPM continuously polls the receiver to check for the presence of a byte of data, and if there is data present, the MPM reads it from the receiver into its memory.

(F) MPM (Main Processor Module)

The MPM does all the higher level processing of the receive data. This board also generates the transmit data to be sent out when in emulate mode. If the XDRAM board is present, see Table J3-1 for proper switch settings for switch S1 on the MPM's for specific allocations of memory space.

1. *Clock generator.* This circuitry generates the clock that runs the 80286 CPU and peripherals on this board.

2. *Global bus arbitor control.* This circuitry queues requests for the global bus. It then grants access to the global bus when the global bus is idle and this board has the highest priority of all requesting boards. This function is performed by the bus arbitor chip.

3. *Bank address register.* The CPU writes to this register to set the upper address bits for the 80286 when it is operating in real mode. In real mode, the CPU does not drive address bits 19 through 23, so these bits must be set in this register if the CPU is running in real mode.

4. *Address buffer latch.* This circuitry simply latches the address bits from the processor in order to guarantee that the address is stable throughout the entire CPU cycle.

5. *Mux.* This multiplexer simply chooses whether to take the upper address bits directly from the processor, or from the bank address register described above. The register is used when the CPU is in real mode, because the CPU will only drive these bits in protected mode.

6. *80286 CPU.* This processor controls the operation of the MPM. The processor operates on software located in the DRAM on the MPM. This software is compiled by the CPM and loaded into the MPM. The software will tell the MPM how to process the data, and what trigger conditions to look for in the data stream. The CPM continually polls the MPM to see if data is available to be sent to any of the user interfaces: printer, plasma display, and remote port.

7. *DRAM.* The DRAM on each MPM is dedicated to storage of receive data. In addition, the DRAM contains the operating software for the MPM. This

memory is dual-ported; this means that the DRAM can be accessed by either the local MPM bus from the 80286 processor or by the global bus. Operation of the DRAM and arbitration of requests from the two ports is controlled by the DRAM controller chip. This chip also generates refreshes for the DRAM at the proper intervals.

8. *Wait/timer logic*. This circuitry generates the proper amount of wait states for some local cycles. It also monitors the operation of the MPM to check for a condition where the CPU gets hung up. The timer logic will generate a non-maskable interrupt to the 80286 processor if a cycle begins and no acknowledge is received within approximately 7 milliseconds.

9. *Reset control logic*. The MPM can be reset in several ways. A global software reset will cause the MPM to go into a reset state. The CPM may also cause a reset on the MPM by writing to a specific global port. Once the MPM has been reset, it will remain in the reset state until the reset is released by the CPM writing to another dedicated global port. Thus, when the unit is first powered up, the CPM can execute its initialization routines, and load the MPM software into the MPM DRAM while the MPM is still in the reset state.

10. *32 bit counter*. The 32 bit counter is made up of four 74LS590's in series which count a 1 MHz clock. These counters are used to determine elapsed time between certain events in the data stream. The count is latched by writing to a dedicated local port. The count is read 16 bits at a time, by reading one of two local ports. The count may be cleared to zero by writing to another dedicated local port.

11. *5 channel timer*. The 5 channel timer is the 9513A, a chip that uses six clock sources as inputs. The user programs the chip to count one of the clock sources and generate an output when the terminal count is reached. The output can be a high or low pulse, or simply a toggle of the output line. Each condition may be executed once or repeatedly. There are five separate outputs, one for each channel. Each channel is independent and may be programmed differently. The 9513A is used on the MPM to generate "timeout" interrupts at specific intervals as required by the user program.

12. *Interrupt control*. The interrupt control circuitry on the MPM consists of three 8259A chips cascaded to allow for a maximum of 22 interrupts to the 80286 processor. Five interrupts come from the 9513A timer logic. Ten interrupts come directly from the global bus. These global interrupts are initiated by peripherals on the global bus. Four other interrupts are software driven interrupts that are caused by another processor writing to dedicated memory-mapped I/O ports on the MPM. Two interrupts come from the FEB, but only one is used.

(G) TIM (Test Interface Module)

The test interface module (TIM) is the interface between the external data path and the INTERVIEW unit. Data enters the unit at the place marked DTE and exits the unit at the place marked DCE.

1. *Mode control.* The mode control circuitry determines the operating mode of the unit. There are three operating modes: monitor, emulate DTE, and emulate DCE. When in monitor mode, the mode control circuitry configures the relays on the TIM so that data may pass through from the DTE to the DCE while being monitored by the INTERVIEW unit. When in emulate DTE mode, the mode control circuitry configures the relays so that the connection to the DTE is broken and data generated by the INTERVIEW unit is transmitted out to the DCE. When in emulate DCE mode, the mode control circuitry configures the relays so that data generated by the DTE is received by the INTERVIEW unit and the connection to the DCE is broken. In addition, when in either emulate mode, the mode control circuitry sets the relays to enable the INTERVIEW to send or receive the appropriate control signals depending on the emulate mode selected.

2. *Patch panel.* The patch panel is simply a row of switches and headers which enable the user to selectively connect and break individual signals on the back of the test interface module.

J3.2 Install a Board

Remove the new board from its static-proof bag. Remember to handle the board by its edges, not touching its components or the gold edge-connector. Grasping the board by the two top corners, slide it *gently* into the correct slot, seating it properly in the connector.

Check the settings for switch S1 on each MPM board to make certain that they are correct. Check that the proper MPM board will be in its correct slot. For the INTERVIEW 7500 and 7700 *TURBO* they should read as shown in Table J3-1.

For the INTERVIEW 7000 and 7200 *TURBO*, place your MPM board in slot 9, with the settings the same as those referenced for that board in Table J3-1 also.

Table J3-1
S1-Switch Settings for MPM Boards
in the INTERVIEW 7000 Series

MPM	Slot 7			Slot 8			Slot 9		
Pin	1	2	3	1	2	3	1	2	3
Units with 3 MPMs:									
Normal†	ON	OFF	ON	OFF	ON	ON	ON	ON	ON
High-Speed††	OFF	OFF	ON	ON	OFF	ON	ON	ON	ON
Units with 2 MPMs:									
Normal†				OFF	ON	ON	ON	ON	ON
High-Speed††				OFF	OFF	ON	ON	ON	ON
Units with 1 MPM:									
Normal†							ON	ON	ON
High-Speed††							ON	ON	ON

† **These are the normal MPM switch settings. Changing these settings will affect object program compatibility with other units.**

†† **These MPM switch settings maximize the size of high-speed record RAM when the file** ***/sys/xdram_rcrd*** **resides on the boot-up disk during power-up.**

J3.3 Reconnect the Cables

Reconnect the cables in the reverse order that you removed them. Remember, the smallest jumper cable connects the CPM and PCM boards and the larger jumper cable connects the FEB and GBM boards. Of the other cables, one connects the LED display to the GBM board and another connects the dual floppy disk drives to the PCM board. Additionally, there are two cables connecting the Winchester hard drive to the PCM board in the INTERVIEW 7500 and 7700 *TURBO* (and those INTERVIEW 7000s and 7200 *TURBO*s containing the optional hard drive).

Recall that several of the connectors are "keyed" for easy, correct alignment. Those that are not "keyed" need to have special care taken to line up the pins with the proper connecting holes. See Figure J3-1.

J3.4 Test the Unit

Before you completely secure the cover, it is suggested that you test your unit to make certain that it functions after this board exchange. First, replace the cover.

Grasp the cover with the latch area to the front facing you. Make certain the latches are recessed as far as they can go into the cover and their hooks are not beyond the edge of the cover. Slide the cover down. For the INTERVIEW 7500 and 7700 *TURBO*—and those INTERVIEW 7000s and 7200 *TURBOs* with the optional hard drive—take care that the two cables to the Winchester hard disk drive (on your left, directly behind the handle) are not being pinched or damaged by the cover. (Some models may contain a small, removable, protective plastic sheet to help shield the cables from the cover.)

Now you can safely test the unit. For this test you must power the unit. Reconnect your power cable, turn on the unit, and check for the system self-tests.

The words "System RAM test now accessing all RAM :CPM0M0M1M2" appear on your screen, indicating that the first self test for the system is being performed. Following this first test are the tests for the rest of the system. The INTERVIEW prompts the user as each test is passed: RAM, timer, DMA, all MPMs, and 32-bit timer.

The statement "The unit has passed ALL system tests" should appear after these tests. The screen repaints as illustrated in Figure J3-3 for the INTERVIEW 7500 and Figure J3-4 for the INTERVIEW 7000.

If the self test produces an error, try reinstalling the new boards. Go to the beginning of this section and carefully follow the same precautions and instructions given. Replace the cover and power up the unit again so it can perform the self test once more. If the self test still gives errors, call Customer Service.

If the start-up screen is blank or the front-panel LEDs are not red or green (except for the REMOTE and FREEZE LEDs), the connectors may not be attached properly. Try reconnecting them again.

```
              ** INTERVIEW  7500 **

     DISKS: FLOPPY 1   FLOPPY 2   HARD DISK

     PROCESSORS: 4

     SELF TEST ERRORS: NONE

Press:
[PROGRAM] to enter the menu page
[RUN]     to run the default program

Software  Version:  7.00
Firmware  Version:  5.00

OPTIONS:

TIM: RS-232/V.24

Copyright (c) 1987, 1989
Telenex Corporation
```

Figure J3-3 INTERVIEW 7500 screen after self test.

```
              ** INTERVIEW  7000 **

     DISKS: FLOPPY 1   FLOPPY 2

     PROCESSORS: 2

     SELF TEST ERRORS: NONE

Please insert system disk and press any key
```

Figure J3-4 INTERVIEW 7000 screen after self test.

J3.5 Secure the Cover

Place the unit on its back once again and replace the six screws to hold the cover secure.

Appendix J4: Installing Multiplexer Board

CAUTION: STATIC ELECTRICITY CAN DAMAGE THE UNIT WHEN THE COVER IS REMOVED. Before you begin this section, you must take the proper anti-static precautions given in Appendix J1.

Observe the proper handling of the boards: refer to Sections J1 and J2 on proper anti-static precautions.

J4.1 Prepare the Slot or Remove Board

(A) Locate the Slot

Read Appendix J2 for instructions on removing the cover and for information on the interior layout of the INTERVIEW 7000 Series. Then refer to Figure J4-1. The optional multiplexer (mux) board will be installed in the first slot, which is likely to be empty.

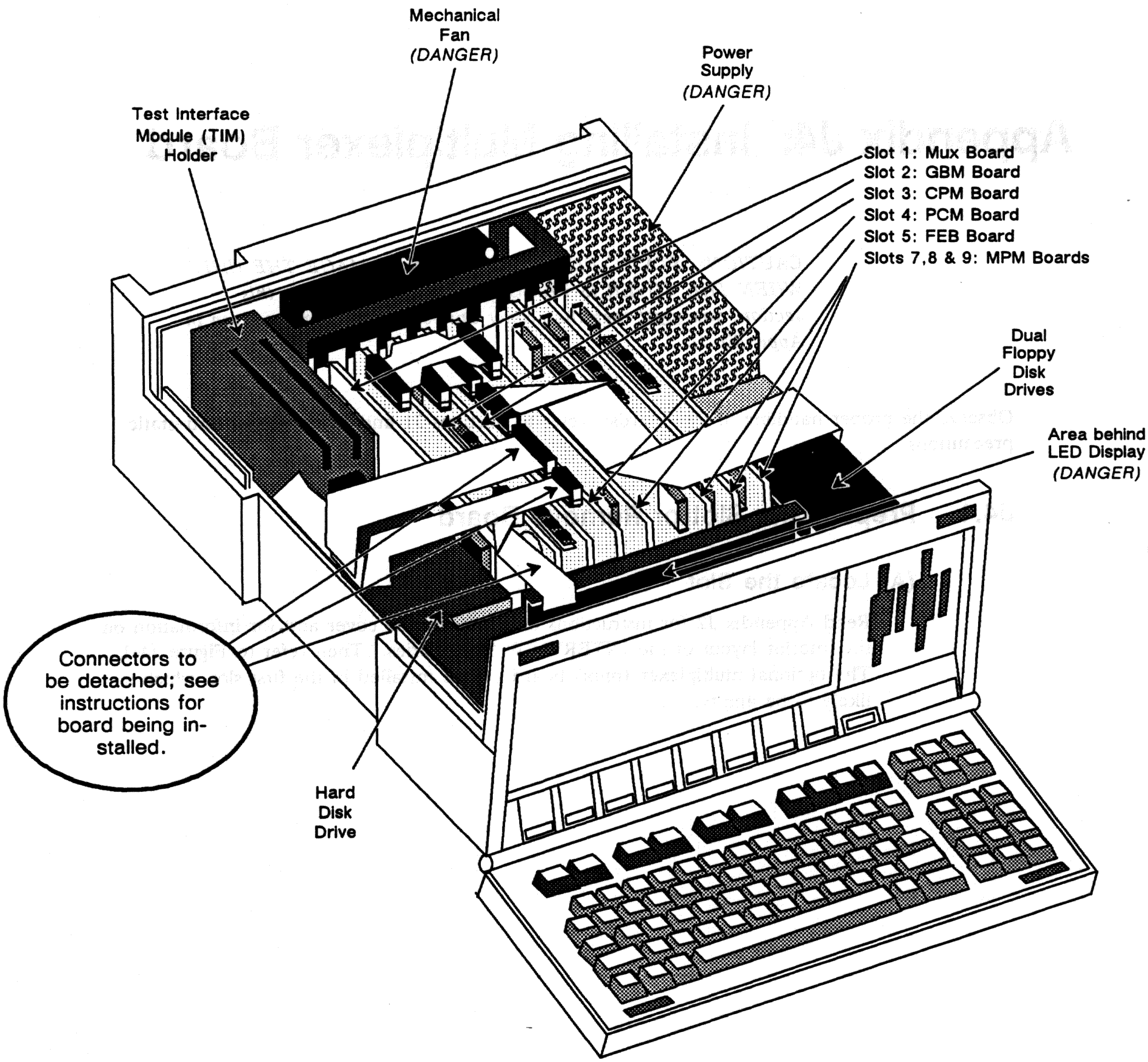


Figure J4-1 Three cable connections are detached to install the mux board in slot 1.

(B) Detach Surrounding Cables

Make certain the unit is on an anti-static mat and you are wearing a grounded wrist strap.

Disconnect the cable connections *carefully*. They may be easily removed with an IC clip holder. Lift the connector straight up, holding onto its edges. DO NOT PULL ON THE CABLE; you could break it or damage the connectors. If you do not have an IC clip holder, a blade screwdriver will also work.

> *CAUTION: If you use a screwdriver, use the top of the black mounting rack (which secures the ends of the boards) as your leverage point. Avoid using the edges of the boards or other components to pry the connectors loose.*

Refer to Figure J4-1 to locate the three cable connectors to be detached—two from cables extending from the hard disk drive and one from a cable attached to the back of the LED display. The two from the hard disk drive are attached to the PCM board in slot 4. The other cable is attached to the GBM board in slot 2. Remove these three connectors and fold the cables back out of the way.

(C) Remove Old Board (if Present)

> *CAUTION: Do not hold any board by its gold edge connector. Hold it by the sides, NEVER touching the components.*

Refer to Figure J4-1. If you are replacing the mux board, locate it in slot 1. Remove the board by grasping the top corner edges. *Gently* pull it straight up and out, and holding the board by its side edges, place it on the anti-static mat.

If requested to do so, package and return the old mux board to the AR Division. Instructions for proper packing and shipping may be found in Appendix Section J2.5.

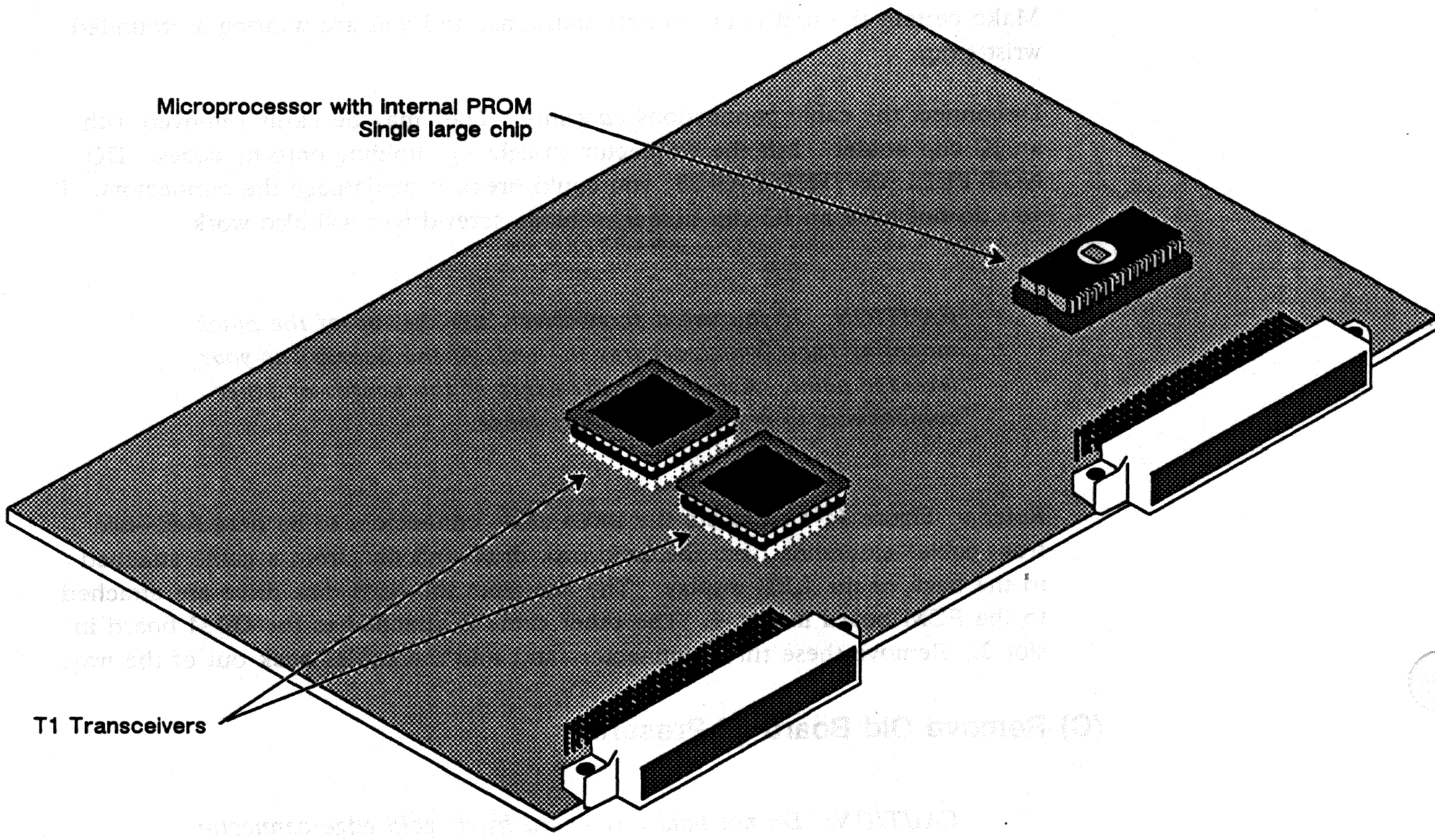


Figure J4-2 All PROMs are factory installed on the T1 mux board, represented above by the most prominent components.

J4.2 Replace the Board

(A) Install the Board in the Appropriate Slot

Install the mux board (T1 mux board represented in Figure J4-2) in slot 1, in the same manner in which you removed it. Take care to hold it only by the edges. *Gently* slide it into its connector.

(B) Reconnect the Cables

The cables should be reconnected in the reverse order in which you disconnected them.

First, replace the jumper cable running from the back of the LED display to the GBM board (in slot 2). Then reconnect the two cables from the hard disk drive to the PCM board (in slot 4).

J4.3 Test the Unit

Before replacing the cover, review your work to be certain you have followed all the instructions correctly. Follow procedures in Section J3.4 to replace the cover (as a safety measure before you test the unit) as well as to test the new firmware in a power–up before you secure the cover.

The mux board option should be listed on the power–up screen in the OPTIONS: field. If there are any errors listed on the SELF TEST ERRORS: line, repeat the procedures from the beginning of this section and try reseating the mux board. Replace the cover and power up the unit again so it can perform the self test once more. If self test errors persist, contact Customer Service.

J4.4 Secure the Cover

When the SELF TEST ERRORS: line shows NONE, place the unit on its back and replace the six screws to hold the cover secure.

Appendix J5: Replacing Firmware on the CPM Board

CAUTION: STATIC ELECTRICITY CAN DAMAGE THE UNIT WHEN THE COVER IS REMOVED. Before you begin this section, you must take the proper anti-static precautions given in Appendix J1.

From time to time there may be a PROM change to upgrade the boards in the INTERVIEW. Correct procedures for changing a PROM are documented in this section.

J5.1 Avoid Inherent Difficulties

When removing or installing PROMs, take care to avoid generating static electricity, bending or breaking pins, and making improper connections.

(A) Static Electricity

Preventing static electricity is essential and is covered in Appendix J1. Taking the proper precautions given in that section will protect the sensitive components.

(B) Bent or Broken Pins

Individual pins on any connector or PROM can easily bend or break. Attaching or detaching components as vertically as possible and aligning them properly will help prevent damage to the pins.

(C) Improper Connections

Improper connections can cause system self-test errors as well as damage to the components. You can avoid improper connections with proper alignment of the pins. Be certain each individual pin is secure in its corresponding socket.

Always align the PROM with its notched end above the notched end of the socket. See Figure J5-2.

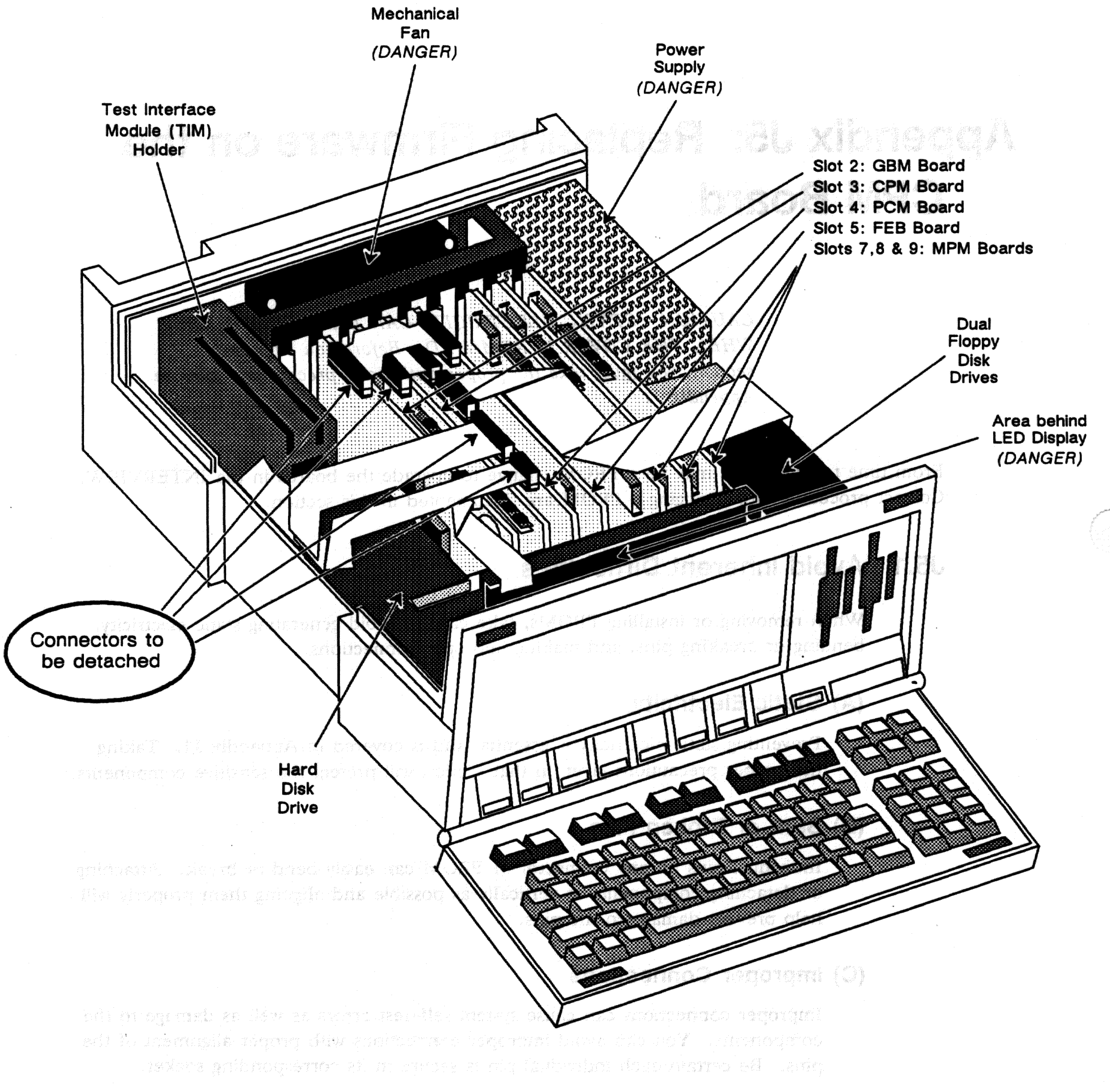


Figure J5-1 Four cable connections are detached to remove the CPM board from slot 3.

J5.2 Remove the CPM Board

(A) Locate the Board

Read Appendix J2 for instructions on removing the cover and for information on the interior layout of the INTERVIEW. Refer to Figure J5-1. The CPM board is in the third slot from the left.

(B) Detach Surrounding Cables

Make certain the unit is on an anti-static mat and you are wearing a grounded wrist strap.

To remove the CPM board, first remove the cables surrounding it and attached to it. Disconnect the cable connections *carefully*. They may be easily removed with an IC clip holder. Lift the connector straight up, holding onto its edges. DO NOT PULL ON THE CABLE; you could break it or damage the connectors. If you do not have an IC clip holder, a blade screwdriver will also work.

> *CAUTION: If you use a screwdriver, use the top of the black mounting rack (which secures the ends of the boards) as your leverage point. Avoid using the edges of the boards or other components to pry the connectors loose.*

Refer to Figure J5-1 to locate the four connectors to be detached—two from cables extending from the hard disk drive and one each from two small jumper cables. The two from the hard disk drive are attached to the PCM board in slot 4. One jumper cable is attached to the GBM board in slot 2 and the other is attached to the CPM board in slot 3. Remove these four connectors and fold the cables back out of the way.

(C) Remove the Board

> *CAUTION: Do not hold any board by its gold edge connector. Hold it by the sides, NEVER touching the components.*

Referring to Figure J5-1, locate the CPM board in slot 3 and remove it by grasping the top corner edges. *Gently* pull it straight up and out, and holding the board by its side edges, place it on the anti-static mat.

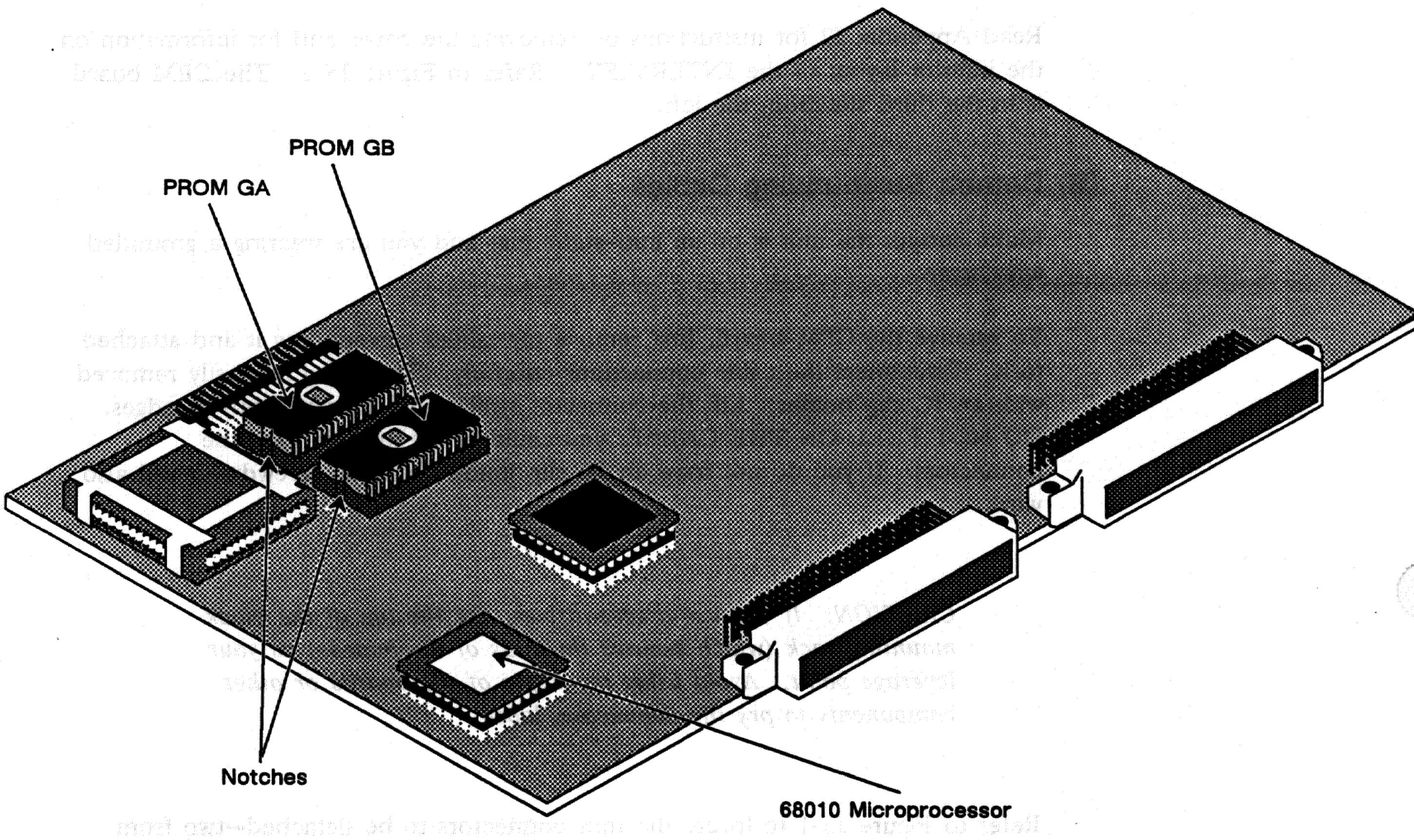


Figure J5-2 PROM GA and PROM GB are located on the upper left side of the CPM board.

J5.3 Exchange the PROMs on the CPM Board

(A) Locate the PROMs

The PROMs to be changed are located in the upper left-hand corner of the CPM board, one above the other. Refer to Figure J5-2. In the INTERVIEW 7000 and 7200 *TURBO*, they are identified as *951GA-101X* and *951GB-102X*; in the INTERVIEW 7500 and 7700 *TURBO*, they are identified as *951GA-111X* and *951GB-112X*. (The *X* refers to the present release level and this last letter will change for each new PROM.) Note that the PROM with GA in its identification number is always in the upper socket and the PROM with GB in its number is always in the lower one.

(B) Remove the PROMs

The PROMs may be removed with an IC clip holder, grasping the ends of the PROM and lifting it as straight up as possible. If an IC clip is unavailable, a blade screwdriver may be used. *Do not use any component on the board to pry up the PROM with the screwdriver.* Place the screwdriver under the smaller edge of the PROM and *gently* maneuver the blade in a rocking, semi-circular motion to loosen the PROM pins as evenly as possible.

Place the old PROM in the packing material in which the replacement PROM was sent for its return to the AR Division. (See Appendix F, Packing and Shipping Instructions.)

(C) Install the New PROMs

To install each replacement PROM, refer to Figure J5-2. After examining the PROMs to make sure the two rows of PROM pins are parallel, notice that one end of each PROM has a notch in it. Each notch has a corresponding one in that PROM's socket on the board. Align the notches as well as the pins and their corresponding holes. Again, the PROM with GA in its identification number goes into the upper socket and the PROM with GB in its number goes into the lower one.

J5.4 Replace the Board

(A) Install the Board in the Appropriate Slot

Replace the board in slot 3 in the same manner in which you removed it. Take care to hold it only by the edges. *Gently* slide it into its connector.

(B) Reconnect the Cables

The cables should be reconnected in the reverse order in which you disconnected them. First, replace the smallest jumper cable from the PCM board to the CPM board. Next, replace the larger jumper cable from the FEB board to the GBM board (in slot 2). Finally, reconnect the two cables from the hard disk drive to the PCM board (in slot 4).

J5.5 Test the Unit

Before replacing the cover, review your work to be certain you have followed all the instructions correctly. Follow procedures in Section J3.4 to replace the cover (as a safety measure before you test the unit) as well as to test the new firmware in a power-up before you secure the cover.

If the SELF TEST ERRORS: line shows anything other than NONE and you are installing the CPM board, try reinstalling the PROMs. Go to the beginning of this section (Appendix J5) and carefully follow the same precautions and instructions given. Replace the cover and power up the unit again so it can perform the self test once more. If the self test still gives errors, call Customer Service.

J5.6 Secure the Cover

When the SELF TEST ERRORS: line shows NONE, place the unit on its back and replace the six screws to hold the cover secure.

Appendix J6: Installing Hard Disk Drive (OPT-951-01-1)

CAUTION: STATIC ELECTRICITY CAN DAMAGE THE UNIT WHEN THE COVER IS REMOVED. Before you begin this section, you must take the proper anti-static precautions given in Appendix J1.

Refer to Appendix Section J2.1 for cover removal information and to Appendix Section J2.2 for a description of the interior layout.

NOTE: Your unit will not have the cables across the top of the hard drive bracket—we will install these later. Nor will it have MPM boards in slots 7 and 8, as Figure J2-1 indicates, unless you have the optional MPM boards installed.

OPT-951-01-1, Winchester 20 Megabyte hard disk drive, is for INTERVIEW 7000 and 7200 *TURBO* units. Installation consists of four steps: removing the empty hard disk bracket, securing the hard drive in the bracket, replacing the bracket and hard drive in the unit, and connecting cables to the hard drive.

J6.1 Remove the Hard Disk Drive Bracket

Locate the hard disk drive bracket at the front of the left side of the unit, as shown in Figure J2-1.

There is an unattached power cable with a four pin female connector behind this bracket. This cable will give power to the hard drive once it is installed. For now, move it towards the back and outside of your unit, clear of the bracket.

There are five screws holding this bracket in the unit—two small Phillips screws toward the front outside of the bracket and three standard-head captive screws holding the bracket to the bottom of the unit, as shown in Figure J6-1. Remove the two small screws and loosen the three captive screws on the base of the bracket—one is in the front and the other two are in the back. Then lift the bracket straight up and out of the unit.

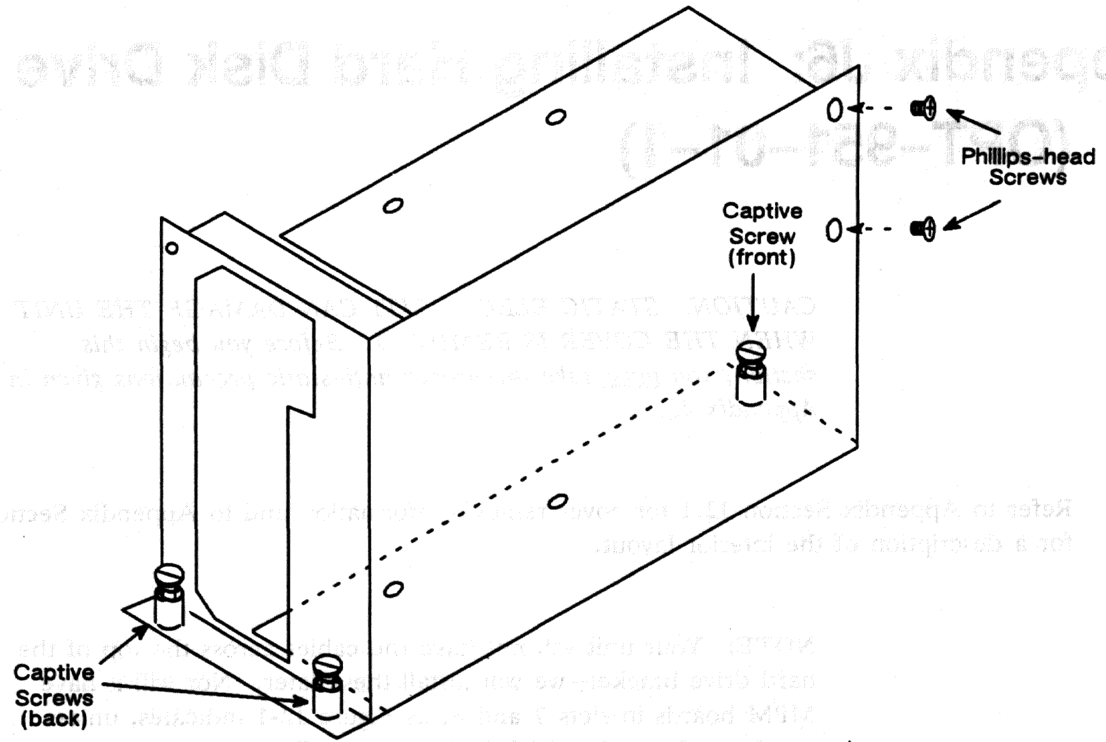


Figure J6-1 Hard disk drive bracket. Note placement of screws.

J6.2 Secure the Hard Disk Drive in the Bracket

Remove the Winchester 20 Megabyte Hard Disk Drive from its packing material. Notice one side of the drive has exposed components. This side will remain exposed when you place it in the bracket. Slide the drive into the bracket with the component side of the drive facing out on the open side of the bracket, as shown in Figure J6-2.

Use the four Phillips-head screws included in your installation kit to mount your hard drive into the bracket. Insert and secure two of these into the top of the bracket and the other two into the bottom of the bracket.

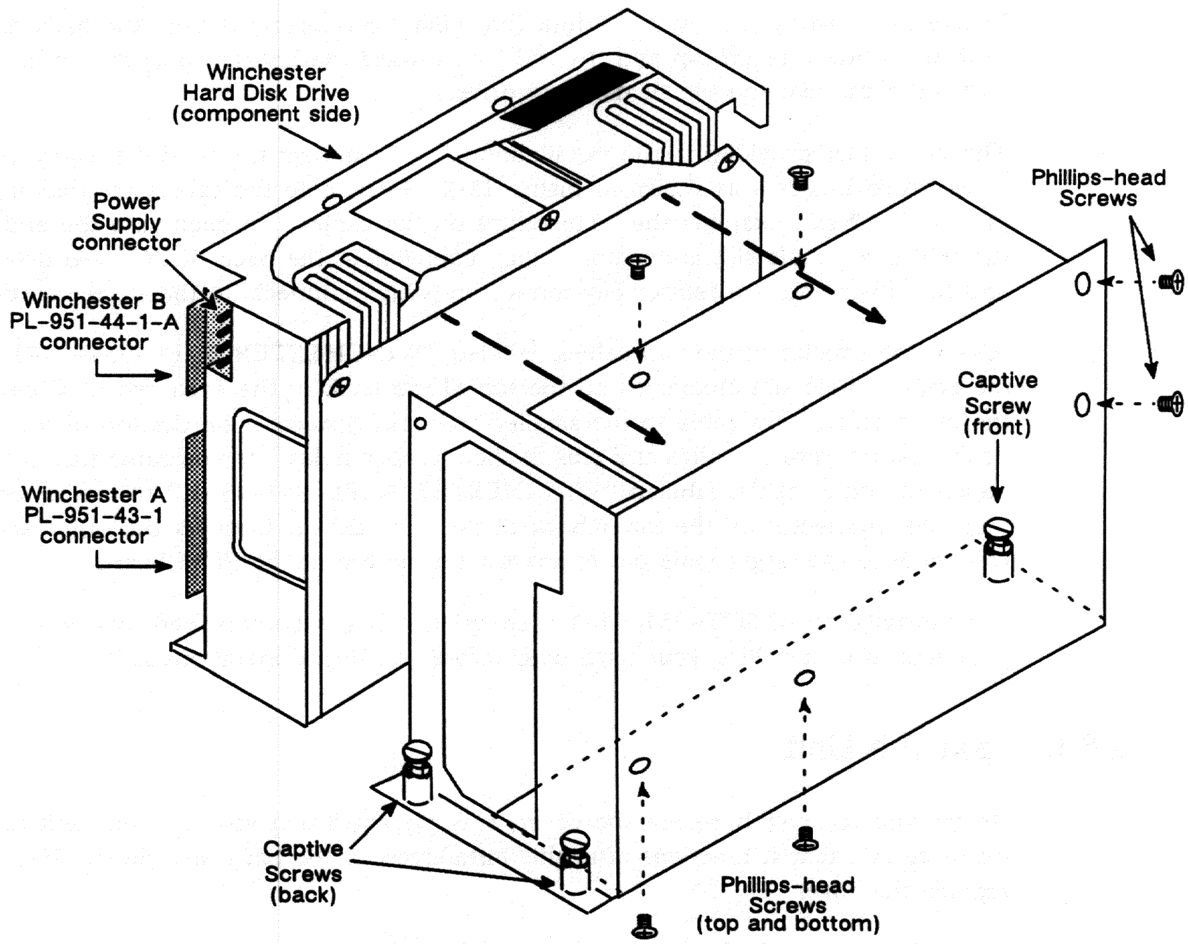


Figure J6-2 **Slide the Winchester hard disk drive into the bracket with the component side exposed and secure the drive in the bracket with the four Phillips-head screws provided.**

J6.3 Replace the Bracket with the Hard Drive into the Unit

Place the hard drive and its bracket in the INTERVIEW 7000 or 7200 *TURBO* by sliding it down vertically in the same position that the empty bracket had occupied. The exposed side of the bracket and hard drive should face into the unit. Secure the same five screws you released in removing the bracket: three captive screws to the base of the unit and two Phillips screws on the outside of the bracket.

J6.4 Connect the Cables

There are three cables to be connected: the power cable already in your unit and the two cables provided in the installation kit. See Figure J6-2 for the locations of the male connectors on the back of the hard drive.

Locate the power cable (with the four pin female connector) behind the hard drive. With the white wire on top and the red wire toward the bottom, plug the connector into the back near the top of the hard drive.

The two cables provided in the installation kit will connect the hard drive with the PCM board in slot 4 as shown in Figure J2-1. First, note the cables are folded for a proper fit. Next, examine the connections on the cables. In each case the end near the fold is a female slot connector (which connects to the back of the hard drive) and the other end is a female pin connector (which connects to the PCM board).

Attach the smaller of the two cables, labeled "WINCHESTER B, PL-951-44-1," to the smaller male slot connector on the top of the back of the hard drive. Connect the other end of this cable to the smaller male pin connector on the top of the PCM board, taking care the align the pins in their proper holes. In the same manner, attach the other cable, labeled "WINCHESTER A, PL-951-43-1," to the larger male slot connector on the lower back of the hard drive. Connect the other end of this cable to the larger male pin connector on the top of the PCM board.

Your installation of OPT-951-01-1 is complete. It is recommended that you test your unit after installing your hard disk before you begin operations again.

J6.5 Test the Unit

Before you completely secure the cover, it is suggested that you test your unit to make certain that it functions after this installation of the hard disk drive. First, replace the cover.

Grasp the cover with the latch area to the front facing you. Make certain the latches are recessed as far as they can go into the cover and their hooks are not beyond the edge of the cover. Slide the cover down. Take care that the two cables to the hard disk drive are not being pinched or damaged by the cover. (Some models may contain a small, removable, protective plastic sheet to help shield the cables from the cover.)

Now you can safely test the unit. For this test you must power the unit. Reconnect your power cable, turn on the unit, and check for the system self-tests.

The words "System RAM test now accessing all RAM :CPMOMOM1M2" appear on your screen, indicating that the first self test for the system is being performed. Following this first test are the tests for the rest of the system. The INTERVIEW prompts the user as each test is passed: RAM, timer, DMA, all MPMs, and 32-bit timer.

The statement "The unit has passed ALL system tests" should appear after these tests. The screen repaints as illustrated in Figure J6-3.

The SELF TEST ERRORS: line should show NONE and the OPTIONS: line should display 01-1 as shown in the figure. If the self test produces an error, try reinstalling the hard drive. Go to the beginning of this section and carefully follow the same precautions and instructions given. Replace the cover and power up the unit again so it can perform the self test once more. If the self test still gives errors, call Customer Service.

If the start-up screen is blank or the front-panel LEDs are not red or green (except for the REMOTE and FREEZE LEDs), the connectors may not be attached properly. Try reconnecting them again.

```
                  ** INTERVIEW  7000 **

     DISKS: FLOPPY 1  FLOPPY 2

     PROCESSORS: 2

     SELF TEST ERRORS: NONE

Press:
[PROGRAM] to enter the menu page
[RUN]     to run the default program

Software  Version:  7.00
Firmware  Version:  5.00

OPTIONS: 01-1

TIM: RS-232/V.24

Copyright (c) 1987, 1989
Telenex Corporation
```

Figure J6-3 INTERVIEW 7000 (with OPT-951-01-1) screen after self test.

J6.6 Secure the Cover

Place the unit on its back once again and replace the six screws to hold the cover secure.

Appendix J7: Servicing Other Components

Other components of the INTERVIEW 7000 Series may require changing or modifying. Those changes and modifications will be documented in this section.

J7.1 Changing the Dual Floppy Disk Drive Bracket

If either ejection button of your floppy disk drives tends to stick when you operate the drive, you can eliminate the problem by changing the bracket for the dual floppy disk drives. While this is a simple operation, there are certain steps you must follow to insure proper installation.

First you must prepare the drives for the exchange. Then, you will disconnect the power source from the drives and lift out the present bracket holding the drives. Next you will remove the cables, exchange brackets, and replace the cables. Finally, you will replace the disk drives into the unit and reconnect the power source cables.

(A) Prepare the Drives

To prepare the dual floppy disk drives for removal, remove any disks they contain. This secures the eject button inside the drive so it won't dislodge upon removal of the unit. Locate the cable which attaches the dual floppy disk drives to the PCM board. Detach the connector from the board.

(B) Disconnect the Power Source from the Drives

Locate the small round cables which connect each drive to the power supply. They are found at the lower back of the floppy disk drives. Disconnect them from the drives.

(C) Lift out the Present Bracket

To prepare the present bracket for removal, locate the four captive screws which secure the bracket to the chassis. Using a common blade screwdriver, loosen these four screws from their bosses. These screws will not come out; loosen them sufficiently to release the bracket from the chassis.

Grasp the sides of the bracket and slide it towards the back of the chassis, freeing the eject buttons and the drives from the front panel slots. Then lift the drive assembly up and out of the unit.

(D) Remove the Cables

The dual floppy disk drives have a common cable which you previously detached from the PCM board. Remove the cable from the back of the disk drives, taking care not to damage the cable. Lift the connectors straight off the connecting pins. For reference, this cable is labeled "PL-951-42-1-A."

(E) Exchange the Brackets

In this next step, mark the disk drives so that you are certain to keep them in the same positions in the new bracket as they presently sit in the old bracket.

Using a common blade screwdriver, unscrew the four small screws on the top of the bracket and the four small screws on the bottom of the bracket. Place both brackets upright next to each other. Slide out one disk drive and place it the same position in the new bracket. Similarly, place the second disk drive in the new bracket.

At this point, you will replace the four top screws, turning them just enough to catch the drive in the bracket. One or two turns will suffice on each screw. Turn the assembly over and attach the four bottom screws in the same manner.

Place the assembly upright once again, aligning it with the eject buttons to your right. The drives should still be moveable in the bracket. Grasp the drive closest to you and move it as far to the left and away from you as it will go. Tighten the top two mounting screws for that drive. Follow the same procedure for the drive farthest from you.

Turn the assembly over, keeping the eject buttons on your right. Grasp the drive closest to you and pull it towards you. Tighten the bottom two mounting screws for this drive. Follow the same procedure for the drive farthest from you.

(F) Replace the Cables

Attach the cable to the back of the disk drives. Notice that these cables are "keyed." (Use Figure J3-1 as a reference.) With the eject buttons facing you, wrap the cable around the drive so the cable fold is on the right side of the drive.

(G) Replace the Dual Floppy Disk Drives into the Unit

Check the four bosses which hold the captive screws securing the bracket. If the threaded insert is not flush with or lower than the top of the boss, push it down inside the boss with the tip of a hot soldering iron.

Making sure the power cables you first disconnected are pushed out of the way, place the floppy drive assembly into the unit. Slide it forward, positioning the eject buttons and drive openings into the front panel slots. Tighten down the four captive screws and test the mechanical operation of each disk drive by inserting a floppy disk and then ejecting it.

(H) Reconnect the Power Source Cables

Locate the connecting pins for the power supply cables on the bottom of the rear of each floppy disk drive. Attach these cables to their connectors.

J7.2 Secure the Cover

Grasp the cover with the latch area to the front facing you. Make certain the latches are recessed as far as they can go into the cover and their hooks are not beyond the edge of the cover. Slide the cover down. For the INTERVIEW 7500 and 7700 *TURBO*—and those INTERVIEW 7000s and 7200 *TURBO*s with the optional hard drive—take care that the two cables to the Winchester hard disk drive (on your left, directly behind the handle) are not being pinched or damaged by the cover. (Some models may contain a small, removable, protective plastic sheet to help shield the cables from the cover.)

Place the unit on its back once again and replace the six screws to hold the cover secure.

J7.3 Return Parts to AR Division Factory

Any parts which need to be returned to the AR Division should be properly packaged. Contact Customer Service for a RETURN AUTHORIZATION (RA) number.

Customers outside the Washington D.C. Greater Metropolitan Area	1-800-368-3261
Local customers	644-9190

The RA number should be posted on the outside of the package of all equipment returned for repair. The RA number, as well as a description of the problem, should be cited in all documentation, written correspondence, or telephone conversations concerning the specific repair.

International customers should address the shipment to

Telenex Corporation, AR Division
ATTN RA *number*
c/o Emery Customs Brokers
101A Executive Drive
Sterling, Virginia 22170
U.S.A.

> **NOTE:** For customs purposes, international customers *MUST* identify the country of origin for returned equipment on the *pro forma* invoice. When returning an individual part, use the country of origin listed on the part.

Domestic customers should address the shipment to

Customer Service
Telenex Corporation
AR Division ATTN RA *number*
7401 Boston Boulevard
Springfield, Virginia 22153
U.S.A.

Consult Appendix E for additional information on returning parts to the AR Division Factory.

Appendix K: C Language Summary

[The following material is adapted from the appendixes of the Proposed C Standard (ANSI document X3J11/86-098) with certain additions to describe the INTERVIEW implementation of C. The appendixes are not a part of the American National Standard. Information presented is collected from the Standard but is not necessarily complete.]

[For more information on C language and syntax, consult Section C of the Proposed Standard.]

K.1 LANGUAGE SYNTAX

[Editorial comments which appear in this section but are not part of the proposed standard are enclosed in non-italicized square brackets.]

In the syntax notation used in this section, syntactic categories (non-terminals) are indicated by *italic* type, and literal words and characters (terminals) by **bold** type. [These items have no sub-categories.] A colon following a non-terminal introduces its definition. Alternative definitions are listed on separate lines, except when prefaced by the words "one of." An optional symbol is indicated by the subscript *opt* [here, represented as *[opt]*], so that

{ *expression [opt]* }

indicates an optional expression enclosed in braces.

K.1.1 Lexical Grammar

K.1.1.1 Tokens

A token is a minimal lexical element of the language. [Categories of tokens are given below. Each of these categories is further described in a separate sub-section.]

token:

keyword
identifier
constant
string-literal

operator
punctuator

K.1.1.2 Keywords

[These words (entirely in lower-case) are reserved due to their special meanings.]

keyword: one of

auto	**double**	**int**	**struct**
break	**else**	**long**	**switch**
case	**enum**	**register**	**typedef**
char	**extern**	**return**	**union**
const	**float ***	**short**	**unsigned**
continue	**for**	**signed**	**void**
default	**goto**	**sizeof**	**volatile**
do	**if**	**static**	**while**

[* The reserved word **float** is not used in the INTERVIEW implementation of C.]

[The following two words are reserved in the INTERVIEW implementation.]

task **waitfor**

K.1.1.3 Identifiers

An identifier is a sequence of nondigit characters (including the underscore and upper-case letters) and digits. An identifier may not consist of the same sequence of characters as a keyword. [An identifier should also be distinct from user functions or library functions.]

[Essentially, identifiers refer to variables, functions, labels, and various user-defined objects.]

identifier:

nondigit
identifier nondigit
identifier digit

nondigit: one of

_ a b c d e f g h i j k l m
n o p q r s t u v w x y z
A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z

digit: one of

0 1 2 3 4 5 6 7 8 9

K.1.1.4 Constants

[Constants may be any of the basic allowable data types. Floating point constants are not supported.]

constant:

integer–constant
enumeration–constant
character–constant

fractional–constant:

digit–sequence [opt] . *digit–sequence*
digit–sequence .

exponent–part:

e *sign [opt] digit–sequence*
E *sign [opt] digit–sequence*

sign: one of

+ -

digit–sequence:

digit
digit–sequence digit

integer–constant:

decimal–constant integer–suffix [opt]
octal–constant integer–suffix [opt]
hexadecimal–constant integer–suffix [opt]

decimal–constant:

nonzero–digit
decimal–constant digit

octal–constant:

0
octal–constant octal–digit

hexadecimal–constant:

0x *hexadecimal–digit*
0X *hexadecimal–digit*
hexadecimal–constant hexadecimal–digit

nonzero–digit: one of

1 2 3 4 5 6 7 8 9

octal–digit: one of

0 1 2 3 4 5 6 7

hexadecimal-digit: one of

0 1 2 3 4 5 6 7 8 9
a b c d e f
A B C D E F

integer-suffix:

unsigned-suffix long-suffix [opt]
long-suffix unsigned-suffix [opt]

unsigned-suffix: one of

u U

long-suffix: one of

1 L

enumeration-constant:

identifier

character-constant:

'c-char-sequence'

c-char-sequence:

c-char
c-char-sequence c-char

c-char:

any character in the source character set except
the single-quote ', backslash \, or new-line character
escape-sequence

escape-sequence: one of

\' \" \? \\
\o \oo \ooo
x*h* \x*hh* \x*hhh*
\a \b \f \n \r \t \v

K.1.1.5 String literals

A string literal is a sequence of zero or more characters enclosed in double quotes, as in "**xyz**".

A double quote within a string literal is represented by the escape sequence \".

string-literal:

"s-char-sequence [opt]"

s-char-sequence:

s-char
s-char-sequence s-char

s-char:

any character in the source character set except
the double-quote ", backslash \, or new-line character
escape-sequence

K.1.1.6 Operators

An operator specifies an operation to be performed that yields a value.

[An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations.]

operator: one of

[ ] ( ) . ->
++ -- & * + - ~ ! sizeof
/ % << >> < > <= >= == != ^ | && ||
? :
= *= /= %= += -= <<= >>= &= ^= |=
, # ##

K.1.1.7 Punctuators

A punctuator is a symbol that has independent syntactic and semantic significance but does not specify an operation...that yields a value.

Depending on context, the same symbol may also represent an operator or part of an operator.

punctuator: one of

[ ] ( ) { } * , : = ; ... #

K.1.2 Phrase structure grammar

K.1.2.1 Expressions

An expression is a sequence of operators and operands [variables and constants] that specifies how to compute a value or (in the case of a void expression) how to generate side effects.

primary-expression:

identifier
constant
string-literal
(expression)

postfix-expression:

primary-expression
postfix-expression [expression]
postfix-expression (argument-expression-list [opt])

postfix-expression . identifier
postfix-expression -> identifier
postfix-expression ++
postfix-expression --

argument-expression-list:

assignment-expression
argument-expression-list , assignment-expression

unary-expression:

postfix-expression
++ *unary-expression*
-- *unary-expression*
unary-operator cast-expression
sizeof *unary-expression*
sizeof (*type-name*)

unary-operator: one of

& * + – - !

cast-expression:

unary-expression
(*type-name*) *cast-expression*

multiplicative-expression:

cast-expression
*multiplicative-expression * cast-expression*
multiplicative-expression / cast-expression
multiplicative-expression % cast-expression

additive-expression:

multiplicative-expression
additive-expression + multiplicative-expression
additive-expression – multiplicative-expression

shift-expression:

additive-expression
shift-expression << additive-expression
shift-expression >> additive-expression

relational-expression:

shift-expression
relational-expression < shift-expression
relational-expression > shift-expression
relational-expression <= shift-expression
relational-expression >= shift-expression

equality-expression:

relational-expression
equality-expression == *relational-expression*
equality-expression != *relational-expression*

AND-expression:

equality-expression
AND-expression & *equality-expression*

exclusive-OR-expression:

AND-expression
exclusive-OR-expression ^ *AND-expression*

inclusive-OR-expression:

exclusive-OR-expression
inclusive-OR-expression | *exclusive-OR-expression*

logical-AND-expression:

inclusive-OR-expression
logical-AND-expression && *inclusive-OR-expression*

logical-OR-expression:

logical-AND-expression
logical-OR-expression || *logical-AND-expression*

conditional expression:

logical-OR-expression
logical-OR-expression ? *expression* : *conditional-expression*

assignment-expression:

conditional-expression
unary-expression assignment-operator assignment-expression

assignment-operator: one of

= *= /= %= += -= <<= >>= &= ^= |=

expression:

assignment-expression
expression , *assignment-expression*

constant-expression:

conditional-expression

K.1.2.2 Declarations

A declaration specifies the interpretation and attributes of a set of identifiers. A declaration that also causes storage to be reserved for an object or function named by an identifier is a definition.

declaration:

declaration-specifiers init-declarator-list [opt];

declaration-specifiers:

storage-class-specifier declaration-specifiers [opt]
type-specifier declaration-specifiers [opt]

init-declarator-list:

init-declarator
init-declarator-list , init-declarator

init-declarator:

declarator
declarator = initializer

storage-class-specifier:

typedef
extern
static
auto
register

type-specifier:

char
short
int
long
signed
[float not supported**]**
unsigned
double
const
volatile
void
struct-or-union-specifier
enum-specifier
typedef-name
task-specifier

[The last *type-specifier* in the list above, *task-specifier*, is specific to the INTERVIEW implementation and is not standard C.]

struct-or-union-specifier:

struct-or-union identifier [opt] { struct-declaration-list }
struct-or-union identifier

struct-or-union:

struct
union

struct-declaration-list:

struct-declaration
struct-declaration-list struct-declaration

struct-declaration:

type-specifier-list struct-declarator-list ;

struct-declarator-list:

struct-declarator
struct-declarator-list , struct-declarator

struct-declarator:

declarator
declarator [opt] : constant-expression

enum-specifier:

enum *identifier [opt] { enumerator-list }*
enum *identifier*

enumerator-list:

enumerator
enumerator-list , enumerator

enumerator:

enumeration-constant
enumeration-constant = constant-expression

declarator:

pointer [opt] declarator2

declarator2:

identifier
(declarator)
declarator2 [constant-expression [opt]]
declarator2 (parameter-type-list)
declarator2 (identifier-list [opt])

pointer:

* *type-specifier-list [opt]*
* *type-specifier-list [opt] pointer*

type-specifier-list:
type-specifier
type-specifier-list type-specifier

parameter-type-list:
parameter-list
parameter-list , ...

parameter-list:
parameter-declaration
parameter-list , parameter-declaration

parameter-declaration:
declaration-specifiers declarator
type-name

identifier-list:
identifier
identifier-list , identifier

type-name:
type-specifier-list abstract-declarator [opt]

abstract-declarator:
pointer
pointer [opt] abstract-declarator2

abstract-declarator2:
(abstract-declarator)
abstract-declarator2 [opt] [constant-expression [opt]]
abstract-declarator2 [opt] (parameter-type-list [opt])

typedef-name:
identifier

initializer:
assignment-expression
{ initializer-list }
{ initializer-list , }

initializer-list:
initializer
initializer-list , initializer

task-specifier:
task-identifier { task-body }
task { task-body }
task-identifier

task–body:

external–definition
layer–declaration
task–body external–definition
task–body layer–declaration

layer–declaration:

#pragma layer integer–constant

K.1.2.3 Statements

A statement specifies an action to be performed.

statement:

labeled–statement
compound–statement
expression–statement
selection–statement
iteration–statement
jump–statement
waitfor–statement

[The last *statement* in the list above, *waitfor–statement*, is specific to the INTERVIEW implementation and is not standard C.]

labeled–statement:

identifier : statement
case *constant–expression : statement*
default *: statement*

compound–statement:

{ *declaration–list [opt] statement–list [opt]* }

declaration–list:

declaration
declaration–list declaration

statement–list:

statement
statement–list statement

expression–statement:

expression [opt] ;

selection–statement:

if *(expression) statement*
if *(expression) statement* **else** *statement*
switch *(expression) statement*

iteration-statement:

while (*expression*) *statement*
do *statement* **while** (*expression*) ;
for (*expression [opt]* ; *expression [opt]* ; *expression [opt])* *statement*

jump-statement:

goto *identifier* ;
continue ;
break ;
return *expression [opt]* ;

waitfor-statement:

waitfor { }
waitfor { waitfor_list }

waitfor-list:

expression : statement
waitfor_list expression : statement

K.1.2.4 External definitions

file:

external-definition
file external-definition

external-definition:

function-definition
declaration

function-definition:

declaration-specifiers [opt] declarator function-body

function-body:

declaration-list [opt] compound-statement

K.1.3 Preprocessing directives

[These are instructions to the compiler included in source code.]

The implementation can include sections of program text, conditionally include other source files, and replace macro's.

preprocessing-file:

group
preprocessing-file group

group:

group-part
group group-part

group-part:

tokens [opt] new-line
if-section
control-line

tokens:

token
tokens token

if-section:

if-group elif-groups [opt] else-group [opt] endif-line

if-group:

 # if *constant-expression new-line group [opt]*
 # ifdef *identifier new-line group [opt]*
 # ifndef *identifier new-line group [opt]*

elif-groups:

elif-group
elif-groups elif-group

elif-group:

 # elif *constant-expression new-line group [opt]*

else-group:

 # else *new-line group [opt]*

endif-line:

 # endif *new-line*

control-line:

 # include *<x-char-sequence> new-line*
 # include *"x-char-sequence" new-line*
 # define *identifier tokens [opt] new-line*
 # define *identifier lparen identifier-list [opt]) tokens [opt] new-line*
 # undef *identifier new-line*
 # line *digit-sequence string-literal [opt] new-line*
 # pragma *x-char-sequence new-line*
*new-line*

x-char-sequence:

x-char
x-char-sequence x-char

x-char:

any character in the source character set except the new-line character *lparen:*

the left-parenthesis character without preceding white space

new-line:

the new-line character

K.1.3.1 AR pragmas

#pragma hook *hook_type hook_text*

hook_type: decimal-constant

hook_text: string-constant

#pragma layer *integer-constant*

#pragma nowarn

#pragma object *object_file_name*

object_file_name: string constant

#pragma tracebuf *trace_buffer_declaration_list [opt]*

trace_buffer_declaration_list:

trace_buffer_declaration
trace_buffer_declaration_list trace_buffer_declaration

trace_buffer_declaration:

decimal-constant decimal-constant
* *decimal-constant*

Index To ADD-951-10

A

B

C

D

E

F

G

H

I

K

L

M

O

P

R

S

T

U

V

W

ADDENDUM

MAY 1990

Index A
Technical Manual Part I and Part II

Symbols

A

B

C

D

E

F

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

Index B
C Structures, Variables, and Routines

A

B

C

D

E

F

G

I

K

L

M

N

O

P

Q

R

S

T

U

W

X