WANG

VS

Multi-Station User's Reference

VS
Multi-Station User's Reference

1st Edition — December 1983

800-1149-01

WANG LABORATORIES, INC., ONE INDUSTRIAL AVENUE, LOWELL, MA 01851 • TEL. (617) 459-5000, TWX 710-343-6769, Telex 94-7421

Disclaimer of Warranties and Limitation of Liabilities

The staff of Wang Laboratories, Inc., has taken due care in preparing this manual; however, nothing contained herein modifies or alters in any way the standard terms and conditions of the Wang purchase, lease, or license agreement by which this software package was acquired, nor increases in any way Wang's liability to the customer. In no event shall Wang Laboratories, Inc., or its subsidiaries be liable for incidental or consequential damages in connection with or arising from the use of the software package, the accompanying manual, or any related materials.

NOTICE:

All Wang Program Products are licensed to customers in accordance with the terms and conditions of the Wang Laboratories, Inc. Standard Program Products License; no ownership of Wang Software is transferred and any use beyond the terms of the aforesaid License, without the written authorization of Wang Laboratories, Inc., is prohibited.

PREFACE

This manual provides reference information for the VS Multi-Station. The VS Multi-Station package allows you to customize the keyboard and to simultaneously display up to four interactive tasks on the workstation screen. The Multi-Station also includes a Glossary language, which allows you to write programs that you can assign to Glossary key combinations.

Chapter 1 provides an overview of the Multi-Station facilities. It also includes a sample application to demonstrate the combined use of the features.

Chapter 2 describes how to use a Multi-Station. It describes the processes of loading Multi-Station microcode, managing windows and glossaries, and the differences between Multi-Station operation and standard workstation operation.

Chapter 3 describes the PERSON utility, which allows you to define a customized workstation personality. Chapter 4 describes the Glossary language.

Appendix A describes system administration procedures. It describes the steps that a system administrator must take to convert a workstation to a Multi-Station.

Appendix B includes the GETPARM requests for the utilities included in the Multi-Station package. Appendices C and D provide the hexadecimal codes for the ASCII and WISCII character sets, respectively.

Appendix E provides a sample Glossary language source file. Appendix F provides a list of the words reserved by the Glossary language. Appendix G provides the hexadecimal values for the AID characters, used by one of the Glossary language subroutines.

This manual provides information for programmers, operators, and system administrators. It assumes familiarity with the VS environment, described in the VS Programmer's Introduction (800-1101), and the VS Procedure Language, described in the VS Procedure Language Reference (800-1205). System administrators should already be familiar with the discussion of the GENEDIT utility in the VS System Administrator's Reference (800-1144). Readers wishing to write Glossary language programs should be familiar with the VS programming environment, described in the VS Program Development Tools Reference (800-1307), and one of the VS languages, described in the following manuals:

VS Assembly Language Reference	(800-1200)
VS BASIC Language Reference	(800-1202)
VS COBOL Reference	(800-1201)
VS FORTRAN Reference	(800-1208)
VS PL/I Language Reference	(800-1209)

CONTENTS

CONTENTS (continued)

FIGURES

TABLES

CHAPTER 1
INTRODUCTION TO THE VS MULTI-STATION

1.1 OVERVIEW

The VS Multi-Station package transforms a standard VS workstation into a personalized, multiwindowed workstation. The VS Multi-Station enhances productivity by allowing you to customize the keyboard and to process multiple interactive tasks simultaneously.

The VS Multi-Station supports the following features, which combine to create an efficient, productive, and personal workstation environment:

- Windowing -- Allows you to run and display up to four interactive programs concurrently.

- Personal keyboard definition -- Enables you to define the meaning and the function of each key.

- An expanded set of key functions -- Provides you with added feature options when assigning functions to workstation keys.

- A Glossary language -- Allows you to define new, expanded key functions that operate under program control. Glossary programs can automate repetitive keyboard operations and can perform conditional and text-manipulating operations. After you define a Glossary program, any user can invoke the program.

- Enhanced control of workstation features -- Allows you to control the alarm, blinking fields, and tab operation and to select type-ahead capability.

- International options -- Allow you to alter the character set, define key combinations for accented characters, and modify the default capitalization rules.

The number of windows a particular Multi-Station has is determined when the system is configured. You define all other Multi-Station features through an interactive and easy-to-use utility known as the Personality Editor. The collection of features you select in the Personality Editor is called your workstation personality. Your workstation personality can be loaded whenever you log on to any Multi-Station and can be changed at any time. Thus, the Multi-Station features are part of your workstation personality and are not a fixed part of the workstation.

1.2 WINDOWING

A VS Multi-Station can have up to four windows. Each window is an independent VS task, from which you can run any VS data processing program. For example, you can simultaneously run the EDITOR, the DISPLAY utility, a compiler, and the Symbolic Debugger from separate windows of a Multi-Station.

You can dynamically adjust the number and the size of the displayed windows. Each window can occupy an entire screen, or portions of several windows can be displayed simultaneously. Portions of a program screen outside the boundaries of a window scroll into view as the cursor moves into them. You can set the default window configuration through the Personality Editor and can change it at any time by pressing special window function keys.

VS Word Processing or VS Alliance® can also run from window one of a Multi-Station. However, you cannot access the other windows while VS Word Processing or VS Alliance is in use. The tasks in the other windows continue processing until they must modify the screen.

1.3 PERSONAL KEYBOARD DEFINITION

The Multi-Station Personality Editor offers you total control of the meaning and the function of each key on the keyboard. You can relocate standard workstation keys. For example, you can reverse the locations of the Enter and New Line keys to be consistent with the corresponding word processing key locations or can modify the QWERTY key sequence to AZERTY.

You can also incorporate new key functions supported only by the Multi-Station. Some of these functions support other Multi-Station features; others are improved versions of existing key functions. A summary of the new key functions follows:

- Window operation keys -- You can define keys that display the next or the previous workstation window and move a window up, down, left, or right.

- Data transfer keys -- You can define keys that can copy information from one location to another on the same or a different window.

- Glossary keys -- You can dedicate a workstation key to a specific Glossary function, rather than having to press the Glossary key and another key.

- Enhanced key operations -- The Multi-Station supports new key functions such as an Insert Mode key that operates in a manner similar to the word processing insert mode and a Back Line key that is the inverse of the standard New Line key.

Alliance is a registered trademark of Wang Laboratories, Inc.

1.4 THE VS GLOSSARY LANGUAGE

The VS Multi-Station Glossary language combines the decision processing and the text manipulation of a programming language with the capability to issue and accept keystrokes. The Glossary language allows users to define single key functions that perform complex or repetitive operations easily. In many ways, Multi-Station glossaries are similar to the familiar word processing glossaries; however, Multi-Station glossaries are substantially more powerful.

The VS Multi-Station also incorporates a glossary-by-example facility, which allows you to record and reissue a series of keystrokes. You can also append a glossary-by-example to a Glossary program for subsequent editing through the Personality Editor. PERSON automatically translates the glossary-by-example keystrokes into the appropriate syntax for the Glossary language. Thus, you can specify keystrokes in a Glossary program by typing them on the workstation.

The structure and form of the Glossary language resemble a simplified PL/I. The Glossary language, however, is much easier to learn because Glossary programs have only the character and integer data types and do not use file I/O. The Glossary language includes standard PL/I built-in functions for text string manipulation. It also includes a set of functions and procedures (subroutines) that is designed specifically for workstation applications. A program can control the keyboard, check the current window, manipulate the cursor, and highlight screen locations through Glossary language features.

A Glossary program is composed of a collection of procedures. Each Glossary key function is defined as a single procedure. The Glossary program is edited and compiled through the VS EDITOR or the Personality Editor and attached to your workstation personality. You invoke Glossary programs by pressing the Glossary key and the defined key (just as in VS Word Processing) or by pressing a single key dedicated to the Glossary key combination through the Personality Editor.

Because a Glossary program is part of a workstation personality, it can be invoked from any window. The Multi-Station also supports global glossaries, which run in all windows at once, and auto-start glossaries, which automatically run when the personality is loaded. Global glossaries can perform such operations as logging off all windows. An auto-start glossary, for example, can automatically log you on to all the windows on the Multi-Station.

1.5 ENHANCED CONTROL OF WORKSTATION FEATURES

The VS Multi-Station controls such workstation features as the alarm, the keyboard click, and the blinking of error fields through your workstation personality. The Multi-Station also allows you to control the operation of the tab keys and to specify whether or not the cursor wraps at the screen edges. The Multi-Station supports two new workstation features: type-ahead and window status. The type-ahead option stores all keystrokes in a buffer, allowing you to continue typing while waiting for system response. The window status feature is designed for the Multi-Station and optionally displays the status and the number of the active window.

Because these features are part of your workstation personality, they can be automatically set whenever you log on to any Multi-Station or return from VS Word Processing or VS Alliance. The Multi-Station places traditionally hardware-oriented features under user-modifiable software control.

1.6 INTERNATIONAL OPTIONS

The VS Multi-Station allows you to tailor the workstation to your native language. By modifying the character set, you can customize the keyboard for your language. The Multi-Station also allows you to define accented characters as combinations of an accent key and the key to be accented. Thus, the Multi-Station does not have to dedicate a key to each accented character. The VS supports uppercase-only fields; the Multi-Station allows International users to control the automatic conversion of characters entered into such fields.

The International features are interactively selected through the Personality Editor and are part of your workstation personality. The Multi-Station reduces the effect of native language on a system. Each user can define the Multi-Station to operate within the desired language. Multilingual users can define a personality for each language to run on the same workstation.

1.7 A SAMPLE MULTI-STATION APPLICATION

The Multi-Station offers a large set of features, which are best understood when seen in practice. To demonstrate the coordinated use of the Multi-Station package, this section considers the case where you are a user who programs in VS BASIC and uses VS Word Processing. This section demonstrates how you can change the keyboard layout to accommodate personal typing habits and how you can write and install glossaries that log on and off all the windows and that run the EDITOR with BASIC supplied as the default language.

Users of VS Word Processing have certain typing habits when they return to the VS data processing environment. The data processing equivalent of the Execute key is the Enter key, but the Enter key is located in the position of the word processing Return key. The Return key in word processing performs a function similar to the New Line key. VS utilities and the Command Processor tend to use PF4 to display a previous screen and PF5 to display the next screen of information. To provide consistency with the word processing environment, you can change the keyboard in the following way:

- Reverse the positions of the Enter and New Line keys
- Convert the Back Tab key to PF4 and the Erase key to PF5
- Convert the infrequently-used + and - keys on the numeric keypad to Back Tab and Erase, respectively

The Multi-Station has three windows, so you also need keys to display the next and the previous window. To form a parallel to the word processing keys, you can place the Next Window key on the shifted Erase key (next screen) and the Previous Window key on the shifted Back Tab key. You also need a few keys to change the size of the windows, and place these on the shifted values of the numeric keypad. Thus, you also want to make the following keyboard changes:

- Convert the shifted Back Tab key to the Previous Window key
- Convert the shifted Erase key to the Next Window key
- Convert the 5, 6, 2, and 3 keys on the numeric keypad to window moving keys.

You change the keyboard through the PERSON utility, described in Chapter 3. However, an overview of the process of keyboard definition is provided here.

When you run the PERSON utility, you create a new personality that is based on the standard 2256C personality. You then edit the keyboard layout and change each key to the desired new value. Figure 1-1 demonstrates the process of changing the meaning of a key.

Locate the cursor at the key to be changed and press PF9.

```
                    Multi-Station Personality Editor
                       keypads section (unshifted)

                         insrt bktab              +     -   tab

                      nline delet erase           7     8     9

                            up                    4     5     6

                      left  home  right           1     2     3

                            down            enter 0     3     .


■ insert mode    _ mov wnd down  _ new line      _ reset
_ invisible wnd  _ MOV wnd down  _ next window   _ right
_ invoke gl _    _ mov wnd left  _ pf __         _ space
_ left           _ MOV wnd left  _ pick up       _ tab
_ look down      _ mov wnd right _ prev window   _ up
_ look left      _ MOV wnd right _ put down
_ look right     _ mov wnd up    _ recall
_ look up        _ MOV wnd up    _ recall wnd

  (4) Previous page     (5) Next page      (13) Help  (16) Return
```

Locate the cursor at the new key function and press ENTER.

```
                    Multi-Station Personality Editor
                       keypads section (unshifted)
keyboard section updated
                         insmd bktab              +     -   tab

                      nline delet erase           7     8     9

                            up                    4     5     6

                      left  home  right           1     2     3

                            down            enter 0     3     .









(4/5) PREVIOUS / NEXT keyboard section
  (9) SEARCH for a key function
 (10) REPLACE into WS  (11) External COPY  (13) Help  (16) Return
```

Figure 1-1. Changing the Meaning of Workstation Keys

When you have redefined all the keys, you can load the new personality into the workstation. Then, you select the Glossaries entry from the PERSON main menu and edit a new Glossary source file. PERSON links to the VS EDITOR, and you enter the following source text. The Glossary language is described in Chapter 4.

```
/*  The following procedure automatically logs on all workstations  */
/*  whenever the personality is loaded.  The procedure first        */
/*  checks to make sure that the window is displaying the logon     */
/*  screen.  It then issues a HELP to ensure that the cursor is     */
/*  located at the User ID field and enters MGL on window 1 and     */
/*  ML concatenated with the window number on the other windows.    */
/*  The procedure then enters the password and logs on.             */

auto_logon: procedure options (main);
    if substr(screen(1), 41, 5) = "Logon" then do;
        call playout("(-help!-)");
        if window = 1 then do;
           call playout("MGL");
        end; else do;
            call playout("ML"!!char(window));
        end;
        call playout("(-tab-)PASSWORD(-enter-)");
    end;
end auto_logon;

/*  The following procedure logs off the current window or all the  */
/*  windows when invoked globally.  It first exits any program by   */
/*  issuing a series of PF16 keys until the Command Processor is    */
/*  displayed (identified by the value 'Workstation' on Row 4).     */
/*  The procedure then issues a PF16 and an Enter, logging off the  */
/*  window.                                                         */

auto_logoff: procedure options ('k');
    do while (substr(screen(4), 2, 11) ↑= 'Workstation');
        call playout ("(-pf-16-)");
        call waitforunlock;
    end;
    call playout ("(-pf-16-)(-enter-)");
end auto_logoff;

/*  The following procedure runs the EDITOR from the Command        */
/*  Processor.  It then enters BASIC as the language and erases     */
/*  any text remaining in the field from a default value.           */

edit_in_BASIC: procedure options('B');
    call playout("(-pf-1-)EDITOR(-enter-)
                  BASIC(-erase-)");
end edit_in_BASIC;
```

After entering the source file, you can compile the source and load it into the workstation by pressing PF9 from the EDITOR special menu. When you exit the EDITOR, you return to the PERSON utility. When exiting the PERSON utility, you save the changes made to the personality. You can then modify your Logon procedure to load the new personality each time you log on.

You now have a customized workstation personality that is loaded automatically at each logon. If you wish to make further keyboard modifications or add new Glossary programs, you can edit the existing personality at any time through the PERSON utility.

The above application illustrates one of many possible ways of incorporating a set of Multi-Station features into a personality. You can incorporate all Multi-Station features into the personality through the PERSON utility, but you can also incorporate a glossary and window sizes in other ways. Figure 1-2 provides an overview of all the ways you can incorporate options in a personality.

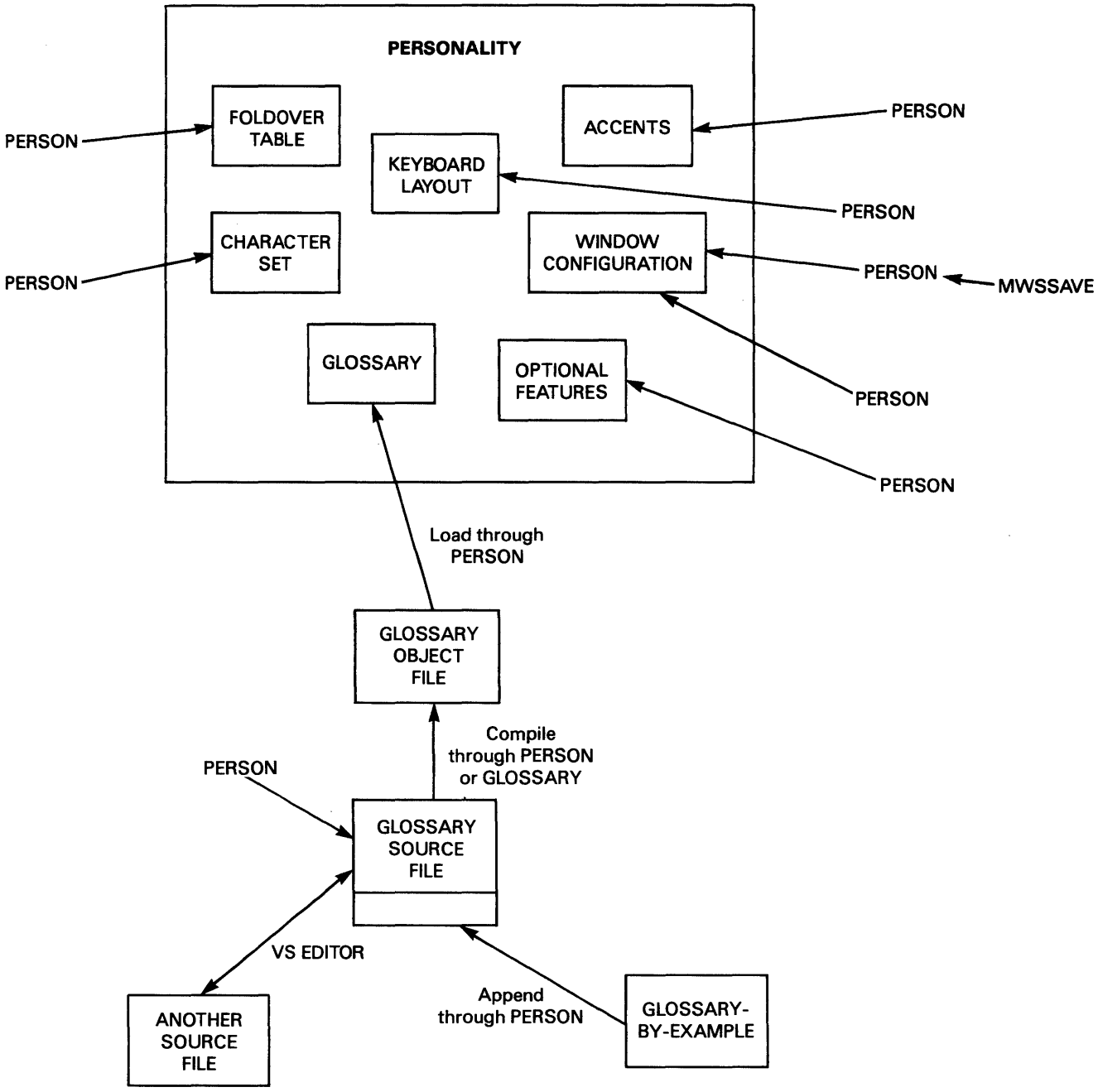


Figure 1-2. Incorporating Personality Features

CHAPTER 2
USING A VS MULTI-STATION

2.1 INTRODUCTION

While a personalized Multi-Station is more convenient to use than a standard workstation, its use involves several different procedures. The VS Multi-Station package consists of workstation microcode, the PERSON utility, a Glossary language compiler, and several support utilities. This chapter describes the coordinated use of the Multi-Station facilities to produce an effective workstation environment.

Assuming that your workstation is configured as a Multi-Station with a set number of windows (system administrators should refer to Appendix A for installation instructions), you need the following items to use a Multi-Station:

- A separate User ID for each window
- A workstation personality

Your system administrator can assign additional User IDs through the SECURITY utility; refer to the VS System Administrator's Reference for details on the SECURITY utility. You define a workstation personality through the PERSON utility, described in Chapter 3. This chapter describes the use of the remainder of the Multi-Station features.

2.2 LOADING WORKSTATION PERSONALITIES

When a Multi-Station is initialized at IPL-time, it is given a default personality that is identical to that of a 2256C workstation except that the pseudoblank character (■) displays as a Dec Tab character (␣). The Multi-Station does not acquire the characteristics defined in your workstation personality until you load the personality file into the workstation. A personality file remains loaded until you replace it with another personality or power off the workstation, or until the system is initialized. Thus, an unused Multi-Station may or may not have the default personality loaded when you log on.

The Multi-Station package includes three utilities that allow you to load or save your personality file under different conditions. While you can always load your personality by running the PERSON utility (refer to Chapter 3), you can also load your personality through the MWSRSTR utility or through the MWSLOAD utility. The MWSSAVE utility allows you to save the personality loaded in the workstation (including the current window configuration) in a separate file.

The MWSRSTR utility allows you to specify and load a workstation personality. When MWSRSTR processing begins, it requests you to enter the file, library, and volume names of the personality file you want to load. When you have identified the file, MWSRSTR loads the specified personality into the workstation.

The MWSLOAD utility loads your default personality file without any user interaction. MWSLOAD assumes that your default personality file resides in a file identified by your User ID located in the MWS library on your System Volume.

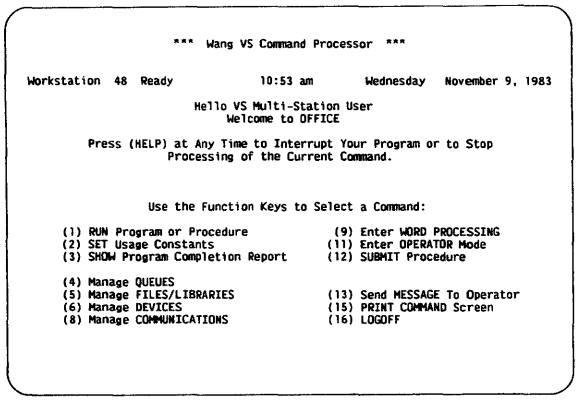
MWSRSTR is useful when you want to load a personality that is different from your default personality. MWSLOAD is appropriate when you want to easily load your default personality. For example, MWSLOAD is typically run as part of a Logon procedure.

The MWSSAVE utility allows you to save the personality that is currently loaded in the workstation in a separate file. When MWSSAVE processing begins, it requests you to enter the file, library, and volume names of the output personality file. When you have identified the file, MWSSAVE saves the personality in the specified file location.

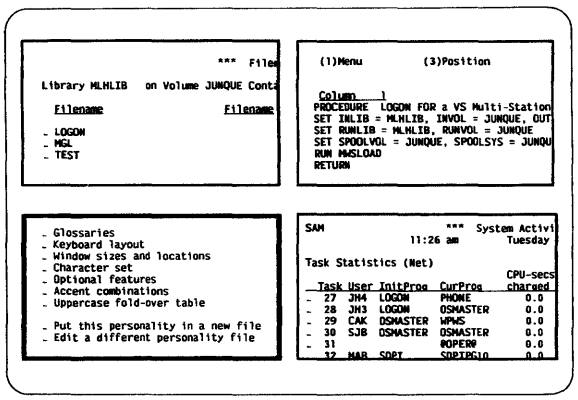
Refer to Appendix B for details about MWSSAVE and MWSRSTR GETPARM requirements and to the VS Procedure Language Reference for details about Procedure language syntax.

2.3 MANAGING WINDOWS

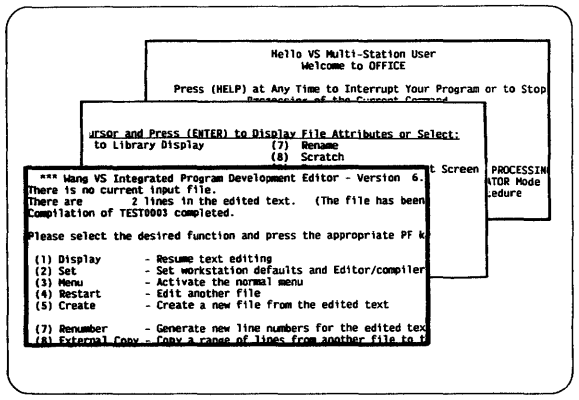
Each Multi-Station has a fixed number of windows that is determined by the system configuration. You can display the windows simultaneously or separately. In either case, each window is a full-powered VS task, regardless of how much of the screen is displayed. If less than the full screen is displayed, the Multi-Station ensures that you can always see the cursor (unless you indicate otherwise). Through the window function keys defined in Chapter 3, you can move the physical window on the screen or move the contents of the window within the boundaries of the window. Figure 2-1 demonstrates several window configurations.



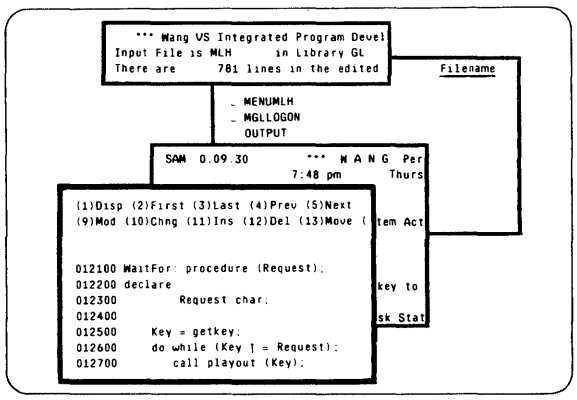
Example A



Example B



Example C



Example D

Figure 2-1. Sample Window Configurations

In Example A, each window occupies a full screen, and only one window is displayed at a time. In Example B, each of the four windows occupies a quarter of the screen. Note that the active window has a bold border. In Example C, the three windows resemble a set of three pages spread on a desk. The active window is always on top. Example D demonstrates a configuration of four windows where three windows are stacked as in Example C, but a fourth window displays a selected line of information (such as the status of an on-going compilation, the time, or the Received message list of a task running VS Office).

The particular window configuration for a Multi-Station is a matter of personal choice. If you define a default window configuration in your workstation personality (refer to Chapter 3 for instructions), the windows are displayed in the default configuration when you load the personality. Otherwise, each window is displayed as a full screen as in Example A. You can always dynamically adjust the window sizes and locations through window function keys defined on the keyboard or issued through a glossary.

NOTE

Your workstation personality must include window function keys or glossary definitions to access more than one window.

Window function keys allow you to move a window about the physical screen or move the contents of the window relative to the window. Other window function keys enable you to change the size of the displayed window. All the window function keys are defined in Chapter 3; however, a few of the keys are described in this section to demonstrate the available window operations.

The Next Window key makes the next higher-numbered window the active window, complete with its window boundaries. Figure 2-2 depicts the result of a Next Window key operation on a Multi-Station with three windows.

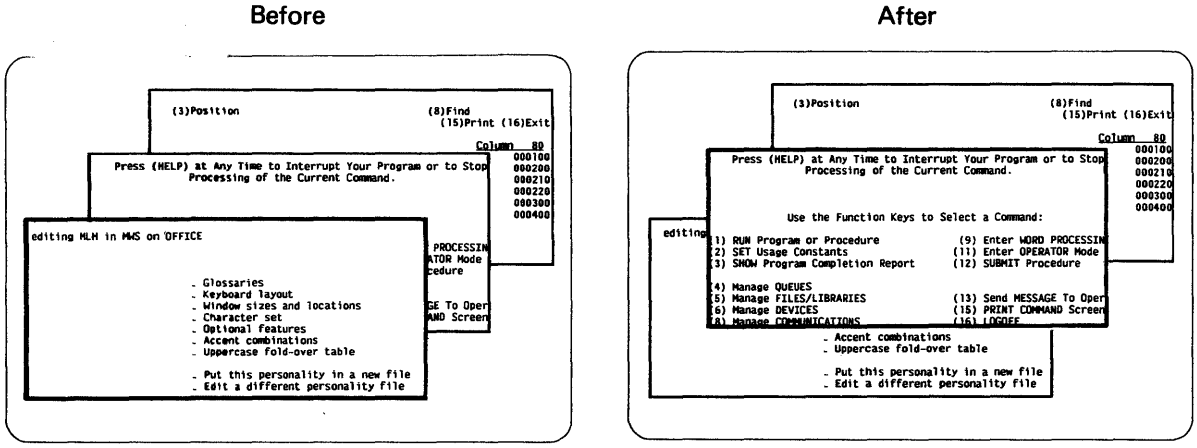


Figure 2-2. Next Window Key Operation

The Get Next Window key also makes the next higher-numbered window the active window, but moves the window into the window size and the location of the current window. Figure 2-3 demonstrates the operation of a Get Next Window key.

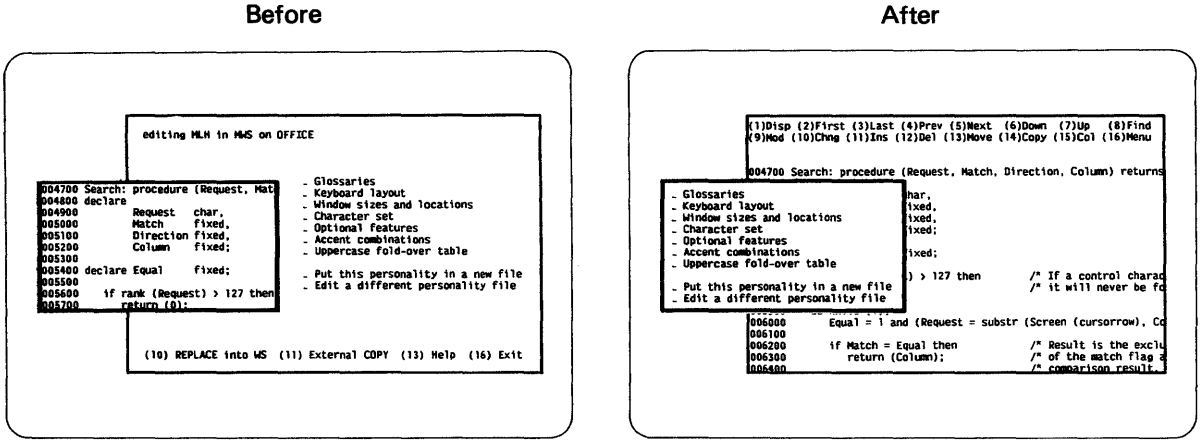


Figure 2-3. Get Next Window Key Operation

The Look Left key displays the previously obscured contents of the window to the left of what is currently displayed. Figure 2-4 demonstrates the operation of the Look Left key.

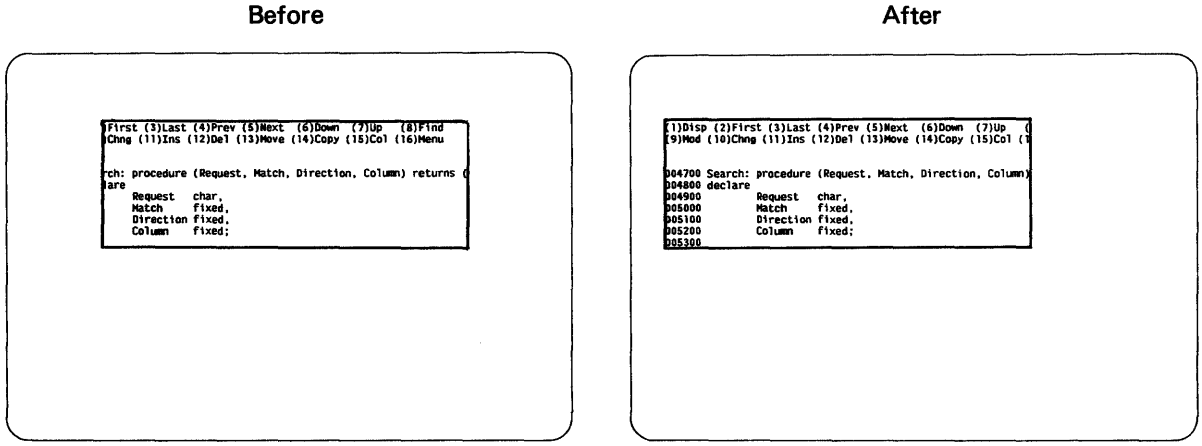


Figure 2-4. Look Left Key Operation

A large number of window functions is available for window manipulation. While dedicating each function to a key through the PERSON utility would reserve a large portion of the keyboard for window management, you can use a Glossary program to issue the window function keys without reserving keyboard locations. You can write a separate Glossary procedure for each window function key you want to use or write a Glossary procedure that performs one of several functions, depending on an additional keystroke that you enter. Refer to Chapter 4 for details on the Glossary language; the Glossary program in Appendix E contains an example window management procedure.

2.4 RUNNING INDEPENDENT MICROCODE PROGRAMS

VS Word Processing and VS Alliance load their own microcode, replacing any workstation personality you loaded in the workstation. As a result, you must run these programs from window one of the Multi-Station. Because they load their own microcode, the Multi-Station is controlled by the current microcode loaded by the program, and you cannot access the other windows. Any processes that are running in windows two through four continue processing until they need to update the screen. When you return from VS Word Processing or VS Alliance, the Multi-Station automatically loads the personality in effect when you ran them.

2.5 USING GLOSSARIES

The Multi-Station includes two types of glossary facilities: compiled glossaries and glossary-by-example. Compiled glossaries are a permanent part of the workstation personality and perform many glossary functions. Glossary-by-example allows you to create a temporary glossary that performs one function. Each Multi-Station can have one permanent and one temporary glossary.

Section 2.5.1 discusses the process of invoking compiled glossaries. (Chapter 3 descibes the process of incorporating a Glossary program into the workstation personality and Chapter 4 describes the Glossary language.) Section 2.5.2 describes the process of creating and using a glossary-by-example.

2.5.1 Invoking Compiled Glossaries

When a compiled glossary program is attached to a workstation personality, all of the individual glossary functions are available to the Multi-Station user. You can run any glossary in the active window or in all windows simultaneously. You run a glossary in the active window by pressing the Glossary key followed by the key that identifies the function; you run a glossary in all the Multi-Station's windows by pressing the Global Glossary key followed by the key that identifies the function.

You can cancel an executing glossary by pressing the Reset key. If the glossary is running in each window, you must press Reset in each window. When a glossary executes a STOP statement (refer to Section 4.4.11), you can continue the glossary's execution by pressing the Glossary key followed by PF16.

2.5.2 Glossary-by-Example

A glossary-by-example allows you to store a sequence of keystrokes in the workstation and to reissue the sequence at any time. A Multi-Station can store one glossary-by-example at a time. The glossary-by-example is available as long as the current personality is loaded in the workstation and can be executed any number of times. Thus, a glossary-by-example is particularly useful for automating repetitive keyboard operations in a single workstation session. If you want to permanently store the glossary-by-example, you can attach it to your Glossary source program, as described in Chapter 3.

The Multi-Station reserves 2048 bytes for the compiled glossary and the glossary-by-example. Because the two types of glossaries share the same space, the maximum size of either type of glossary depends on the present size of the other type. For example, if your compiled glossary occupies 2000 bytes, your glossary-by-example cannot exceed 48 bytes.

You create a glossary-by-example by pressing the Glossary key followed by PF7 (the word processing Note key). If you selected the Status feature descibed in Chapter 3, the Note symbol (!!) is displayed in the status column while the Multi-Station is recording keystrokes. The Multi-Station places every keystroke you enter in the glossary-by-example until you press the Reset key or the Help and Reset key. After you have created the glossary-by-example, you can execute the keystrokes in the active window by pressing the Glossary key twice or in all windows by pressing the Global Glossary key followed by the Glossary key.

2.6 CHANGES IN VS MENU OPERATION

VS menus that allow you to select an option by positioning the cursor to a nonmodifiable numeric field containing a hexadecimal 04 (→), 05 (␣), or 0B (■) operate differently on a Multi-Station. Such menus include, for example, the Manage Files and Libraries screens, PERSON screens, and some Wang OFFICE screens.

On a standard workstation, you select options on such screens by using the Tab, Back Tab, cursor control, and Home keys to position the cursor to the entry and pressing ENTER. On a Multi-Station, however, you can also use the space bar and back space key to move the cursor among the entries. In addition, you can type the first letter of any corresponding option to move the cursor to the entry, provided that the first letter is located two positions to the right of the menu pick. Thus, a Multi-Station automatically converts many VS menus to word processing style menus.

CHAPTER 3
THE PERSON UTILITY

3.1 INTRODUCTION

The PERSON utility, also called the Personality Editor, allows you to define and maintain your workstation personality. The Personality Editor manages each element of the personality through the following functions:

- Creates and maintains workstation personalities
- Allows you to customize the keyboard
- Allows you to edit and compile Glossary language programs and to incorporate them in the workstation personality
- Allows you to modify the character set
- Allows you to set the default window configuration
- Allows you to select workstation features such as keyboard click and type-ahead
- Allows you to define accent key combinations
- Allows you to modify the default capitalization rules

Through PERSON, you can edit all or some of the elements in a single session. Because you define the personality by modifying a standard or an existing personality, you need only define the elements that you wish to change. The Personality Editor merges all your definitions with the default selections and creates a single workstation personality.

The changes you make in the edited personality do not take effect until you load the updated personality into the workstation. (Refer to Chapter 2 for details on loading workstation personalities.) Through PERSON, you can load the personality that you are editing at any stage, allowing you to test your changes before you save them.

An overview of PERSON processing is provided in Figure 3-1.

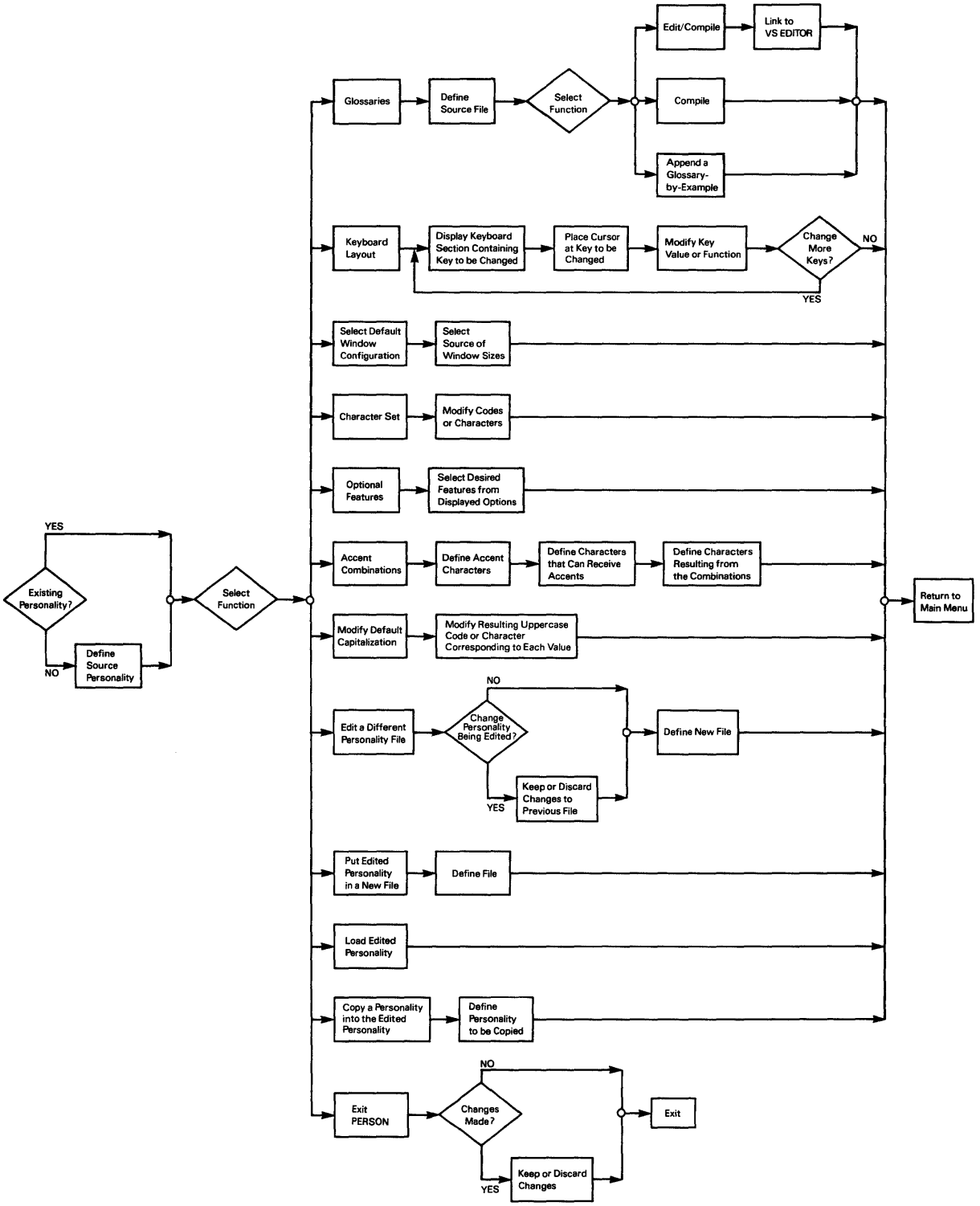


Figure 3-1. PERSON Processing

3.2 RUNNING THE PERSONALITY EDITOR

You run PERSON like any VS utility, that is, through the Command Processor RUN command (PF1) or a VS Procedure language RUN statement. You enter PERSON in the FILE parameter; you do not have to specify a library or volume name because the VS searches the @SYSTEM@ library on the System Volume when the specified file name cannot be located in the indicated library and volume location. Because the Personality Editor is intended for interactive use, you cannot write a procedure to supply values to any of PERSON's screens.

> **NOTE**
>
> You must have at least 304K of Segment 2 address space in order to use the Glossary editing features of the PERSON utility. You need only about 144K to run PERSON if you do not want to edit a Glossary file.

The Personality Editor uses word processing style menus. If the workstation is already configured as a Multi-Station (refer to Appendix A), you move to an option with the space bar or by typing the first letter of the entry, and select the option by pressing ENTER. If the workstation is not a Multi-Station, you move to an option with the Tab, Home, or cursor control keys, and select the option by pressing ENTER. The PF key values for PERSON utility options correspond to similar word processing functions.

Workstation personalities reside in VS files, typically named with your User ID and located in the MWS library on the System Volume. However, a personality file can reside in any VS file location. When PERSON processing begins, the utility searches the MWS library on the System Volume for a personality file named with your User ID. If such a file does not exist, PERSON requests you to identify the personality you want to edit on the Initial Personality Definition screen, shown in Figure 3-2. You can either create a new personality based on one of four provided prototypes or edit an existing file. You can also press PF13 to display more information or press PF16 to exit PERSON.

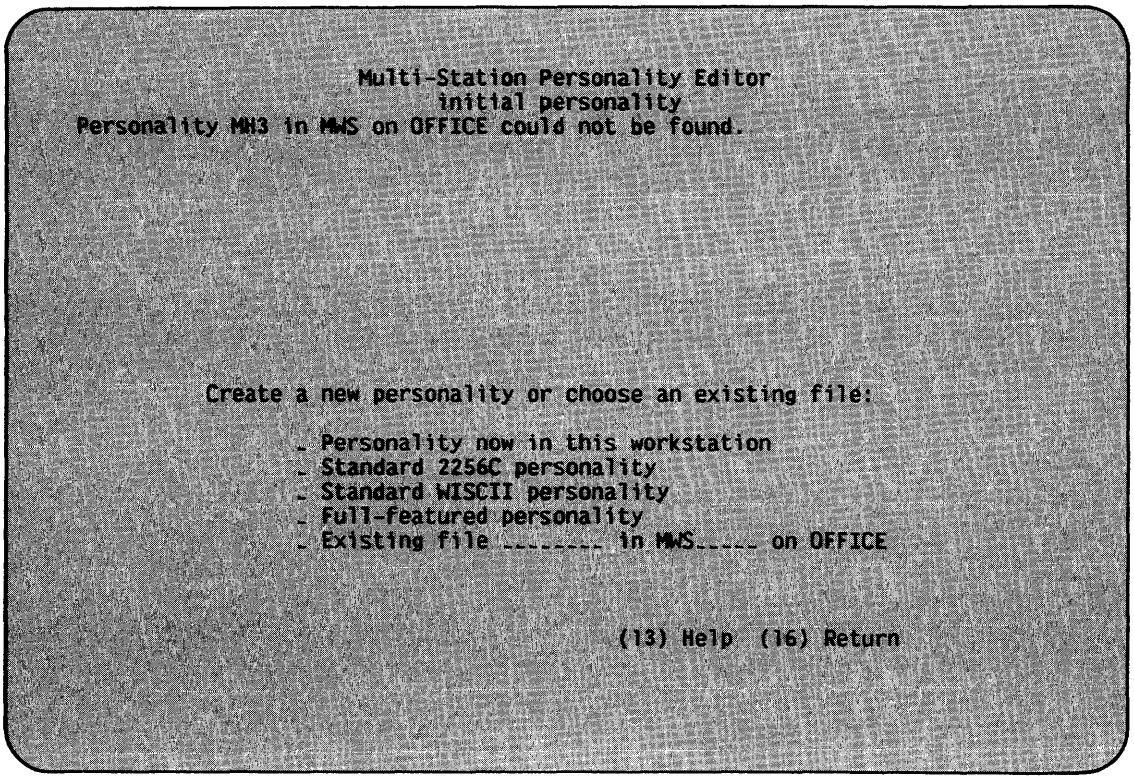


Figure 3-2. The Initial Personality Definition Screen

The four prototype personalities are:

- The personality currently loaded in the workstation
- The standard 2256C personality
- The WISCII personality (a 256 font-position character set used by model 4230 workstations)
- The full-featured personality (an example personality based on commonly-used keyboard modifications)

NOTE

You are editing only a copy of your personality; the changes are not permanent until you replace the personality file at the end of the editing session.

After you identify the personality you want to edit, PERSON displays the main menu, shown in Figure 3-3. The main menu allows you to edit each of the seven personality elements. Pressing PF13 displays more information; pressing PF16 terminates PERSON processing.

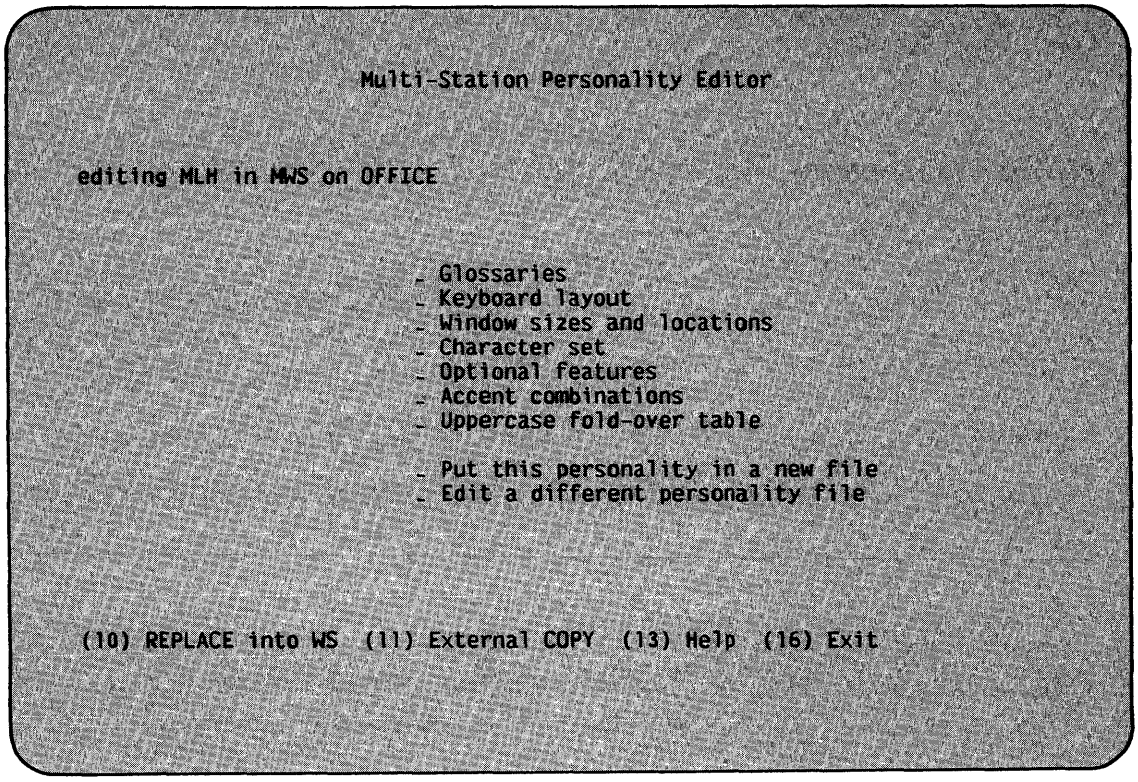


Figure 3-3. The Personality Editor Main Menu

From the main menu, you can also load the personality you are editing into the workstation by pressing PF10. PERSON loads the personality without replacing the file, allowing you to test your changes before you save them. By pressing PF11, you can copy one of the prototypes or a specified existing personality into the edited personality. When you press PF11, the copied personality entirely overwrites the personality you are editing. This feature is useful when you want to replace an existing personality with a different existing personality.

The main menu also provides two personality file maintenance functions. Because PERSON always edits your default personality if it exists, you can specify a different personality file from the main menu. If you have changed the current personality, PERSON asks whether to keep or discard the changes before it edits the new personality. You can also place the current edited personality in a new file; this option allows you, for example, to easily create several similar personalities.

3.3 DEFINING THE KEYBOARD

When you select the Keyboard Layout entry from the main menu, PERSON allows you to interactively define the keyboard. PERSON successively displays a representation of each unshifted and shifted section of the keyboard in the following order:

1. Unshifted main keyboard (alphabetic) section
2. Shifted main keyboard section
3. Unshifted keypads section
4. Shifted keypads section
5. Unshifted PF keys section
6. Shifted PF keys section
7. Unshifted extended universal section (found on 4230 and on some Ergo 3 workstations)
8. Shifted extended universal section (found on 4230 and on some Ergo 3 workstations)

You can move to the next keyboard section by pressing PF5 or to the previous section by pressing PF4. When you press PF5 from the last keyboard section, PERSON displays the first section. The Personality Editor displays key representations for all types of VS keyboards; you should define only those keys that correspond to the workstation you are using. For example, if you have a Model 2256C workstation, you define the numeric portion of the keypad section through the shifted and unshifted keypad section; if you have a Model 4230 workstation, you define the universal keys using the universal section and the numeric keys using the numeric section. In either case, you define the cursor key area through the keypad section.

You can modify the function of any key on the keyboard. To modify a key, you display the appropriate keyboard section and position the cursor next to the key representation. You can move within the keys in the keyboard representation with all cursor control keys (e.g., left, tab, and new line).

There are two types of workstation keys: alphanumeric keys, which write characters, and function keys, which perform workstation functions. You can redefine keys in two ways, depending on the type of the original key. For alphanumeric keys, you can enter new alphanumeric values by typing the new value on the key representation. For example, you can change a QWERTY keyboard to an AZERTY keyboard by typing the new key value in the appropriate location. Note that the changes do not take effect until you load the edited personality.

NOTE

You should take care when changing alphanumeric key values, because you can inadvertently remove a letter from the keyboard if you load a personality that does not contain a key for that letter. If this occurs, you should load one of the prototype personalities to recover the full character set.

For function keys or to assign a function to an alphanumeric key, you assign a key a new function by pressing PF9 when the cursor is located at that key's representation. Because one of the available functions allows you to assign an ASCII value to the key, pressing PF9 also allows you to assign an alphanumeric value to a function key.

When you press PF9, PERSON displays a set of key functions. An example Keyboard Definition screen is displayed after you press PF9 as shown in Figure 3-4. You select a function by positioning the cursor next to the entry and pressing Enter. You can select from two pages of key functions; you can display the second set of functions by pressing PF4 or PF5 from the first screen.

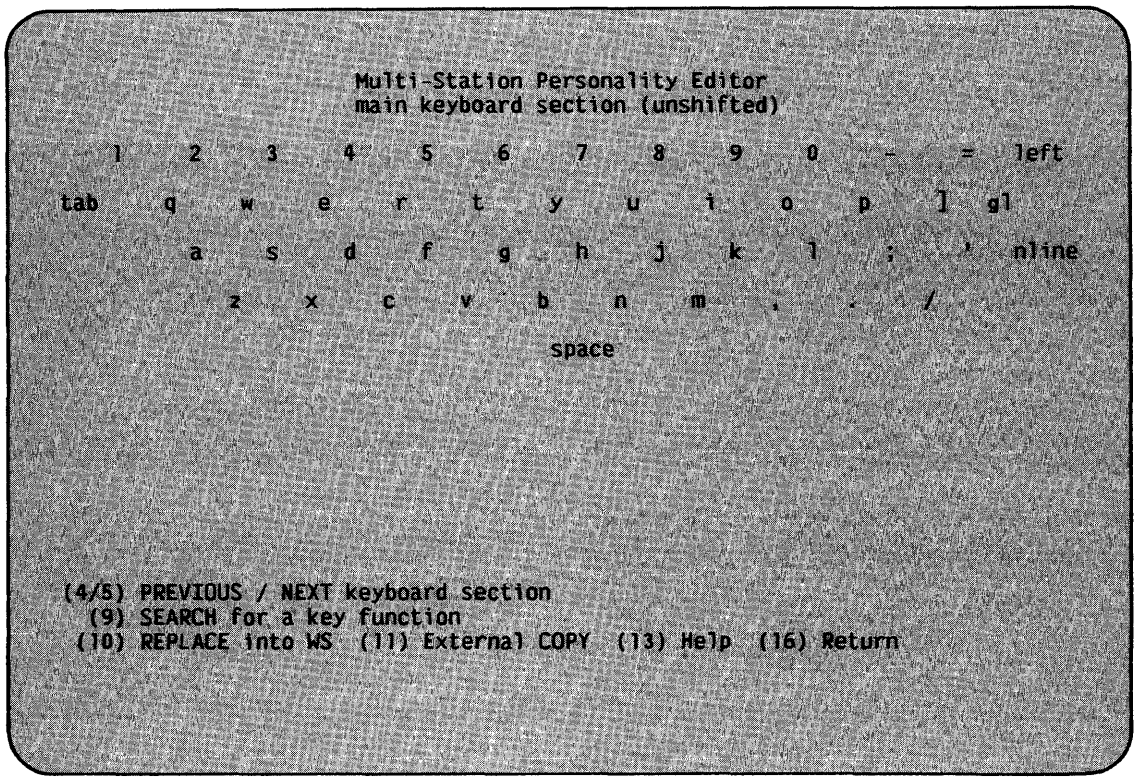


Figure 3-4. An Example Keyboard Definition Screen

You can assign the following functions to workstation keys:

Function	Description
add col left	Adds one column to the left of the window, increasing the window size.
ADD col left	Adds ten columns to the left of the window, increasing the window size.
add col right	Adds one column to the right of the window, increasing the window size.
ADD col right	Adds ten columns to the right of the window, increasing the window size.
add row down	Adds one row to the bottom of the window, increasing the window size.
ADD row down	Adds three rows to the bottom of the window, increasing the window size.
add row up	Adds one row to the top of the window, increasing the window size.
ADD row up	Adds three rows to the top of the window, increasing the window size.
again	Repeats the previous keystroke or glossary invocation from the keyboard. For glossaries, the Again key only reinvokes the glossary and does not repeat any keystrokes you entered while the glossary was executing.
anchor	Causes the workstation to ignore commands issued by executing programs to move the cursor. Thus, you can keep selected information visible in a window, regardless of the progress of the executing program.
ascii x	Types the ASCII character x. This option allows you to assign a character value to any key.
back line	Moves the cursor to the first tab stop on a previous line. If the screen contains no tab stops on previous lines, the cursor moves to the first tab stop on the last line of the screen that contains a tab stop.
back space	Moves the cursor one column to the left and types a blank (or pseudoblank).
back tab	Moves the cursor to the beginning of the previous tab stop. If the screen contains no previous tab stops, the cursor moves to the last tab stop on the screen.

Function	Description
buffered help	Issues a HELP command, which returns you to the Command Processor or to the Operator's Console, depending on the screen from which you pressed the key. If type-ahead is enabled (refer to Section 3.7), the keystroke is processed when all previous keystrokes in the type-ahead buffer have been processed.
caps lock	Converts typed lowercase characters to uppercase, even when typed in a field that accepts lowercase characters. The Caps Lock function affects only the alphabetic characters.
caps unlock	Allows uppercase and lowercase keyboard entry.
cursor to 1,1	Moves the cursor to row 1 and column 1.
del row down	Deletes one row from the bottom of the window, decreasing the window size.
DEL row down	Deletes three rows from the bottom of the window, decreasing the window size.
del col left	Deletes one column from the left of the window, decreasing the window size.
DEL col left	Deletes ten columns from the left of the window, decreasing the window size.
del col right	Deletes one column from the right of the window, decreasing the window size.
DEL col right	Deletes ten columns from the right of the window, decreasing the window size.
del row up	Deletes one row from the top of the window, decreasing the window size.
DEL row up	Deletes three rows from the top of the window, decreasing the window size.
delete	Removes the character at the cursor location, shifts each character in the field one position to the left, and adds a blank (or pseudoblank) at the right edge of the field.
down	Moves the cursor down one row. If wraparound is enabled (refer to Section 3.7) and the cursor is on row 24, the cursor moves to row 1.
enter	The standard ENTER key.

Function	Description
erase	Erases the contents of the modifiable field beginning from the cursor location. The field is filled with blanks or pseudoblanks, as appropriate.
freeze screen	Disables any screen updates in this window until the Reset key is pressed.
full size wnd	Expand the current window to full screen size.
get next wnd	The contents of the next higher-numbered window are moved into the current window; if present, window borders do not move. If you do not have window borders, a Get Next Window command is identical to a Next Window command.
get prev wnd	The contents of the next lower-numbered window are moved into the current window; if present, window boundaries do not move. If you do not have window boundaries, a Get Previous Window command is identical to a Previous Window command.
global glossary	Executes in all windows the Glossary program referenced by the next key pressed.
glossary	Executes in the current window the Glossary program referenced by the next key pressed.
help and reset	Performs a reset operation (clearing the type-ahead buffer) and issues a HELP command.
hex xx	Types the ASCII character represented by the indicated hexadecimal code. You can enter a value ranging from 00 to 7F. This function allows you to directly type characters that are not normally part of the keyboard. Refer to Appendices C and D for the hexadecimal codes for each displayable character.
home	Moves the cursor to the first tab stop on the screen.
insert	Inserts a blank (or pseudoblank) at the cursor location, pushing each character to the right one position to the right. The last character in the field is deleted; if the deleted character is not a blank or pseudoblank, the workstation alarm sounds.

Function	Description
insert mode	Places the workstation in insert mode or, if the workstation is already in insert mode, cancels it. When in insert mode, the keystrokes automatically push the text ahead of them. Nonblank characters at the end of the modifiable field are lost. While in insert mode, the Back Space key performs a shift left and delete operation. If you have selected the window status option (refer to Section 3.7), the Multi-Station displays a circumflex character (^) while you are in insert mode.
invisible window	Makes the active window invisible (zero size). If you selected the window status option in the workstation features (refer to Section 3.7), an invisible window displays as a degree symbol (°). You can restore an invisible window with the Recall Size key. If you press the invisible window key twice when removing the window, the Recall Size key does not restore the window. You can always restore an invisible window through the add and full size window function keys.
invoke gl x	Invokes the indicated glossary in a single keystroke. The specified glossary value must range from 1 to 9.
left	Moves the cursor one column to the left. If wraparound is enabled and the cursor is at column 1, the cursor moves to column 80 of the previous row. If wraparound is enabled and the cursor is at column 1 of row 1, the cursor moves to row 24, column 80.
look down	Moves a previously obscured lower portion of the screen into view when you are working within a window that is smaller than a full screen.
look left	Moves a previously obscured left portion of the screen into view when you are working within a window that is smaller than a full screen.
look right	Moves a previously obscured right portion of the screen into view when you are working within a window that is smaller than a full screen.
look up	Moves a previously obscured upper portion of the screen into view when you are working within a window that is smaller than a full screen.
mov wnd down	Moves the current window down one row on the physical screen, without changing the contents of the window.
MOV wnd down	Moves the current window down three rows on the physical screen, without changing the contents of the window.

Function	Description
mov wnd left	Moves the current window one column to the left on the physical screen, without changing the contents of the window.
MOV wnd left	Moves the current window ten columns to the left on the physical screen, without changing the contents of the window.
mov wnd right	Moves the current window one column to the right on the physical screen, without changing the contents of the window.
MOV wnd right	Moves the current window ten columns to the right on the physical screen, without changing the contents of the window.
mov wnd up	Moves the current window up one row on the physical screen, without changing the contents of the window.
MOV wnd up	Moves the current window up three rows on the physical screen, without changing the contents of the window.
new line	Moves the cursor to the first tab stop at or following the start of the next line. If the screen contains no other tab stops on subsequent lines, the cursor moves to the first tab stop.
next window	Makes the next higher-numbered window the active window. For example, if the active window is window two, pressing a next window key makes window three the active window. If you have three windows and press next window while window three is active, window one becomes active.
pf-xx	Issues a PF key, with xx in the range 1 to 32.
pick up	Places the characters from the cursor to the end of the row in the pick-up buffer. Thus, the pickup buffer cannot exceed 80 characters in length.
previous window	Makes the next lower-numbered window the active window. For example, if the active window is window two, pressing a previous window key makes window one the active window. If you have three windows and press the previous window key when window one is active, window three becomes active.

Function	Description
put down	Places the contents of the pick-up buffer in the modifiable field, beginning at the cursor. You can put down the contents of the pick-up buffer any number of times and on any window. If the contents of the pick-up buffer exceed the length of the modifiable field, only as much of the buffer as will fit is put down in the field. Lowercase text that is put down when the workstation is in Caps Lock mode is automatically capitalized.
recall	Within a modifiable field, returns the portion of the field to the right of the cursor to its state when you entered it. Once you leave a field and type something else, you can no longer recall the prior field's former contents.
recall wnd	Restores a window to its former size. When you modify a window size, the Multi-Station saves the most recent size. However, when window size changes involve multiple operations, recall wnd can only restore the most recent window size.
reset	Clears the type-ahead buffer, sets blinking fields to high-intensity fields, and cancels any executing glossaries.
right	Moves the cursor one column to the right. If wraparound is enabled and the cursor is at column 80, the cursor moves to column 1 of the next row. If wraparound is enabled and the cursor is column 80 of row 24, the cursor moves to row 1 and column 1.
space	Types a blank or a pseudoblank, as appropriate.
tab	Moves the cursor to the beginning of the next modifiable field, numeric-protected field, or soft tab stop. Soft tab stops include such tabs as the tabs defined within the VS EDITOR for program formatting. If the screen contains no more tab stops, the cursor moves to the first tab stop on the screen.
up	Moves the cursor up one row. If wraparound is enabled (refer to Section 3.7) and the cursor is on row 1, the cursor moves to row 24.

3.4 MANAGING GLOSSARIES

When you select the Glossaries entry from the main menu, PERSON allows you to edit and compile glossaries through the VS EDITOR, directly compile an existing Glossary source file, and to append saved keystrokes (glossary-by-example) to Glossary source code. The Glossary Management screen, shown in Figure 3-5, requests you to identify the Glossary source code that you want to edit or compile. The file name defaults to your User ID residing in the GL library on the System Volume but you can change the value to the name of any source code file. You must also indicate whether you are editing and compiling the source code, just compiling the source code, or appending the glossary-by-example. Section 3.4.1 discusses editing and compiling the source file; Section 3.4.2 describes the process of directly compiling an existing source file. Section 3.4.3 describes appending a glossary-by-example.

Because Glossary object programs cannot exceed 2048 bytes in length, the Glossary Manangement screen also displays the amount of space that is used and the number of characters remaining for glossary definition.

The Glossary Management screen also allows you to load the edited personality (including the most recently compiled Glossary program) by pressing PF10. You can copy the compiled Glossary program from one of the four prototype personalities, from a specified personality, or from a previously compiled Glossary object file if you press PF11. Pressing PF13 displays a screen of more information; pressing PF16 returns control to the main menu.

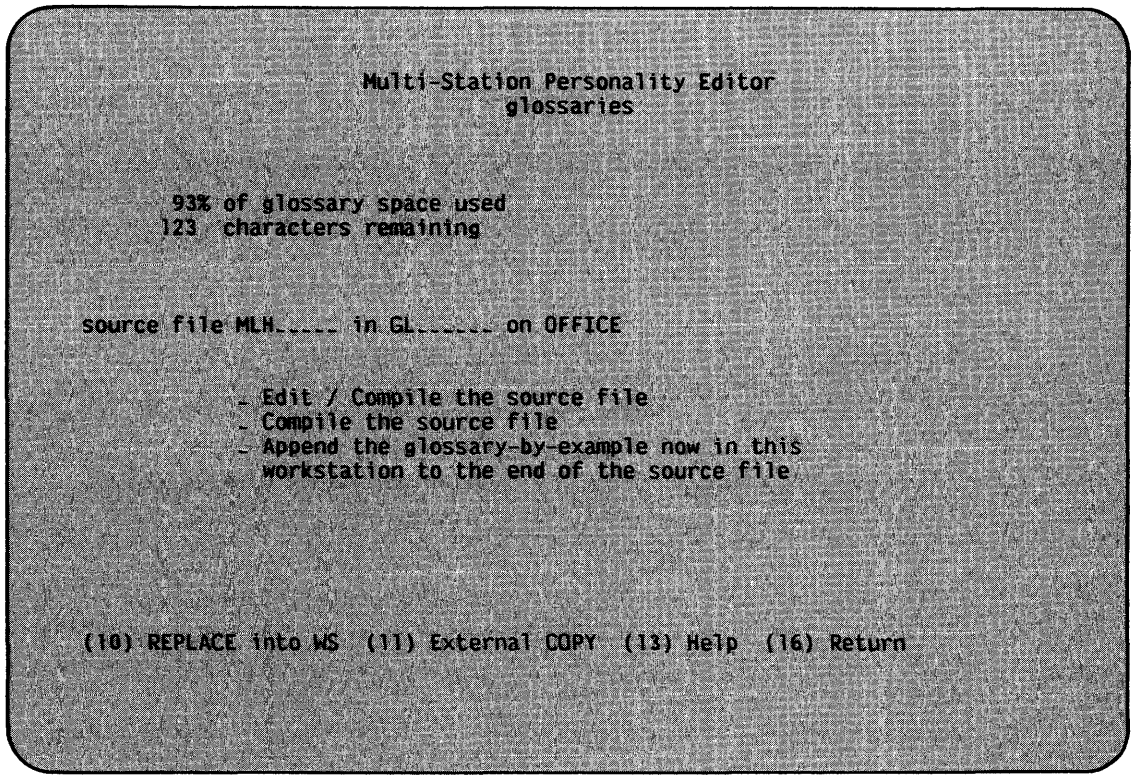


Figure 3-5. The Glossary Management Screen

3.4.1 Editing and Compiling Glossary Programs

When you select the edit and compile option from the Glossary Management screen, PERSON links to the VS EDITOR. If the specified Glossary source file exists, PERSON immediately displays the EDITOR main menu. If the specified file does not exist, PERSON displays the EDITOR Input screen, allowing you to respecify the file name from within the EDITOR. Refer to the VS Program Development Tools Reference for details on the use of the VS EDITOR.

You create and modify the Glossary source code using the VS EDITOR functions. After you have completed your editing, pressing PF9 from the EDITOR special menu compiles and, if the compilation is successful, loads the object file into the edited personality. The object file is placed in a file named with your User ID in the GLOBJ library on the System Volume. You can also press PF10 to compile the source code. You create or replace the source file with the EDITOR Create (PF5) and Replace (PF6) functions. The compiled glossary is loaded into the workstation personality that you are currently editing; the glossary is not a permanent part of the personality until you save the entire personality and does not operate until you load the edited personality into the workstation.

3.4.2 Compiling the Source File

When you select the Compile option, you can directly compile the source file specified on the Glossary Management screen. You cannot specify any compilation options; no source listing is produced. The object file is placed in a file with the same name as the source file in the GLOBJ library on the System volume. If errors occur, the error listing is placed in a file with a file name created by appending "ERRS" to the first four characters of the source file name in your default print file library.

If the compilation completed without errors, PERSON loads the resulting object program into the Glossary area of the current workstation personality. If errors were found, PERSON displays the compilation return code on the Glossary Management screen and does not update the workstation personality.

3.4.3 Appending a Glossary-by-Example

When you select the Append the glossary-by-example option, PERSON translates the keystrokes that you have stored in the workstation into the appropriate syntax for the Glossary language and places them at the end of the source file indicated on the Glossary Management screen. (You cannot append a glossary-by-example if you are running PERSON from a workstation that is not configured as a Multi-Station because you can only store a glossary-by-example on a Multi-Station.) PERSON places the keystrokes in a procedure labelled GlossaryByExample with an options clause that specifies the Glossary key.

NOTE

Unless you modify the key specification in the OPTIONS clause, you cannot invoke a glossary-by-example.

Refer to Chapter 2 for details on creating glossaries-by-example. When the keystrokes have been appended, PERSON displays a completion message on the Glossary Management screen. If no glossary-by-example is currently stored in the workstation, PERSON displays an error message.

3.5 MODIFYING THE CHARACTER SET

When you select the Character Set option from the main menu, PERSON allows you to interactively define the workstation's character set. After you modify the character set, the characters are displayed on the screen according to your selections. For example, changing the character "a" to "b" causes all occurrences of "a" on the screen to be displayed as "b". However, you are only changing the way the character is displayed on the screen; the actual character code is not changed.

PERSON displays a Character Set Definition screen, shown in Figure 3-6, when you select the Character Set option. The Character Set Definition screen displays 8 groups of 3 columns, each containing 16 entries. The first column in each group represents the character code sent by or to the VS (the index character). The second column represents the hexadecimal code displayed on the screen. The third column represents either the displayed ASCII value for the index character or the workstation value, depending on the screen mode.

When you first display the Character Set Definition screen, the second column in each group is modifiable (Hexadecimal mode). Pressing PF5 from the Character Set Definition changes the screen mode successively from Hexadecimal mode to Display mode and then to ASCII mode. Pressing PF5 when the screen is in ASCII mode updates the character set and returns the screen to Hexadecimal mode. Note that if you return to the main menu and then reselect the Character Set option, the Character Set Definition screen is first displayed in the mode in which you exited the option.

You can modify the character set in Hexadecimal or ASCII mode. You modify the character by typing the new hexadecimal code or the new character in the modifiable field that corresponds to the index character to be changed and pressing ENTER or PF5 successively. The character displayed in the ASCII column is not changed until the character set is updated. When the screen is in Display mode the displayed character corresponds to the character set currently loaded in the workstation rather than to the hexadecimal code in the second column.

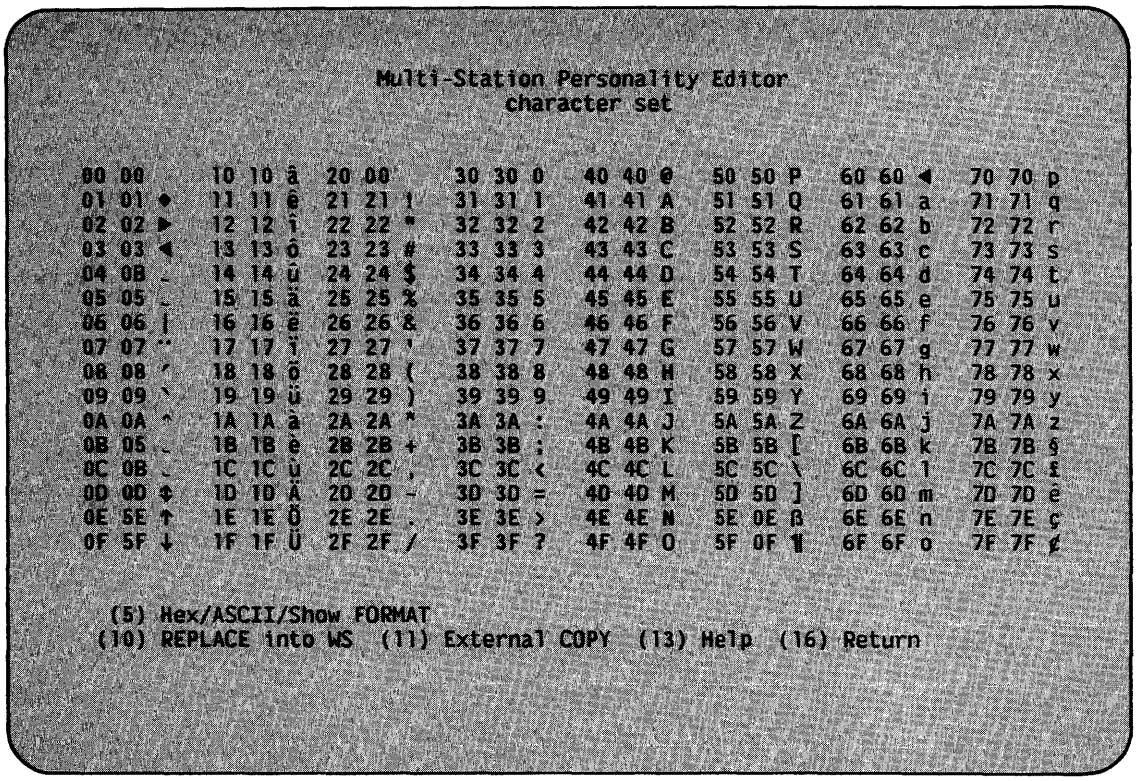


Figure 3-6. The Character Set Definition Screen

You can load the personality with the current version of the character set into the workstation by pressing PF10 from the Character Set Definition screen. You can also copy the character set from an existing or prototype personality by pressing PF11. (Note that the prototype full-featured personality uses the WISCII character set.) You can display instructions by pressing PF13 or return to the main menu by pressing PF16.

3.6 SETTING THE DEFAULT WINDOW CONFIGURATION

Although you can always dynamically adjust the window sizes and locations through the window function keys defined in Section 3.3, this option allows you to define a default window configuration that is loaded with your personality. When you select the Window Sizes and Locations option from the main menu, PERSON displays the Window Configuration Selection screen, shown in Figure 3-7.

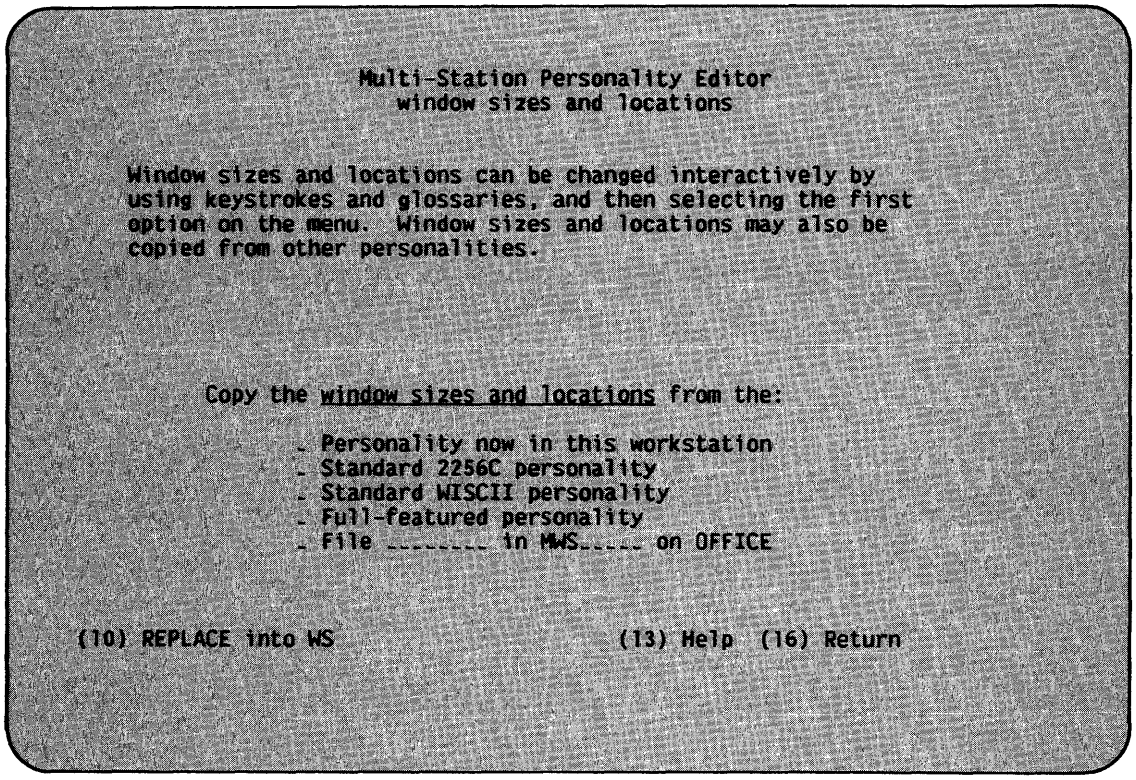


Figure 3-7. The Window Configuration Selection Screen

The Window Configuration Selection screen allows you to define your own window configuration or to copy a window configuration from an existing personality. To define your own window configuration, you use the window function keys (or any glossaries you have defined to perform window management functions) to display the desired configuration. After you have arranged the default configuration, you select the Personality Now in This Workstation option from the Window Configuration screen.

You can also copy the window configuration from the prototype 2256C, WISCII, or full-featured personalities, or from a specified existing file by selecting the corresponding screen options. You can load the current personality into the workstation by pressing PF10 or display information about the prototype personalities by pressing PF13. You can also return to the main menu without setting a default window configuration by pressing PF16. After you set the window configuration, PERSON displays the main menu.

3.7 SELECTING WORKSTATION FEATURES

When you select the Optional features option from the main menu, you can control such features as the alarm and the keyboard click through your workstation personality. PERSON displays the Optional Features screen, shown in Figure 3-8, when you select this option. The Optional Features screen displays which features are currently selected and allows you to modify the selections. You select an option by moving to the option with the space bar or the first character of the option and then pressing the Insert key. You remove an option by pressing the Delete key when the cursor is located at that option. You cannot modify the Optional features if you are not running PERSON on a Multi-Station.

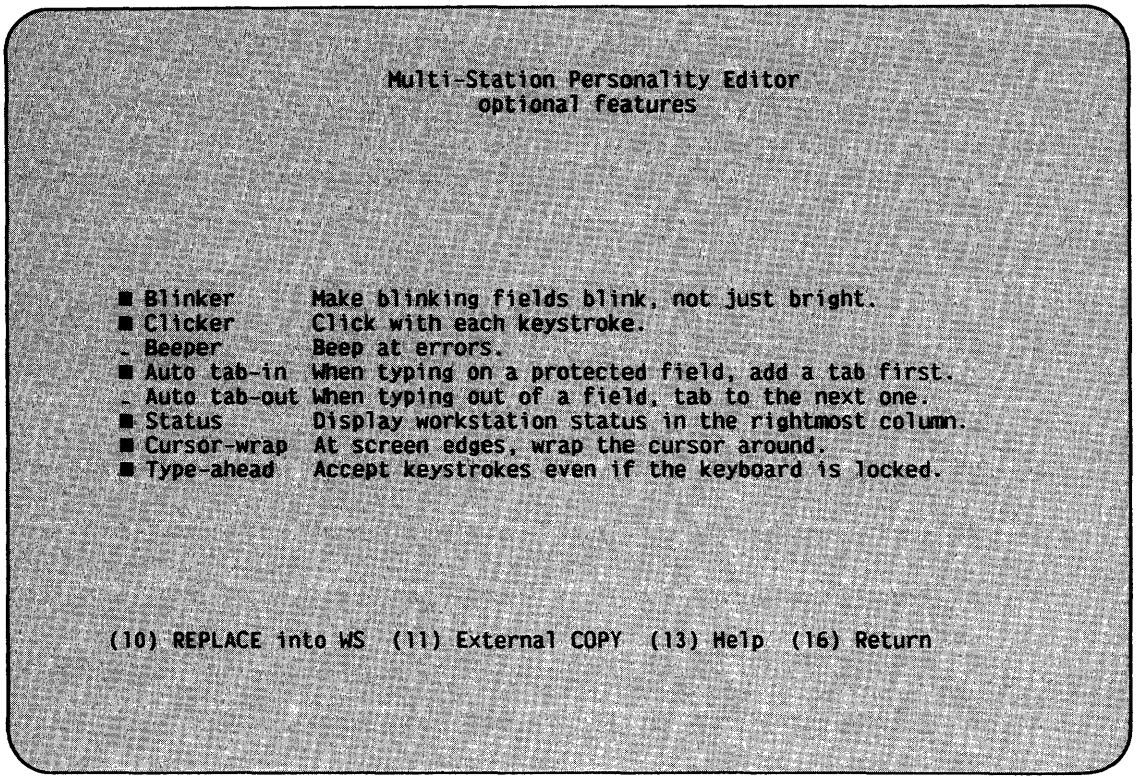


Figure 3-8. The Optional Features Screen

The Optional Features screen allows you to control the following workstation features:

Feature	Description
Blinker	Determines whether fields with the blink attribute set are displayed as blinking fields or as bright fields. If you select this option, blinking fields blink.
Clicker	Determines whether the keys click when you press them. If you select the option, the keys click. On Model 4230 workstations, the volume of the key click is no longer under software control. On other workstations, however, the volume of the click is still controlled by the Clicker control on the back of the workstation.
Beeper	Determines whether the workstation alarm sounds when an error occurs. If you select this option, the alarm sounds when an error occurs. On Model 4230 workstations, the volume of the alarm is no longer under software control. On other workstations, however, the volume of the alarm is still controlled by the Tone control on the back of the workstation.

Feature	Description
Auto tab-in	Determines whether the workstation automatically tabs to the next modifiable field when you begin typing. If you select this option, the workstation automatically adds the tab.
Auto tab-out	Determines whether the workstation automatically tabs to the next modifiable field when you fill the previous modifiable field. Standard workstations have this feature. The workstation issues a tab after a completed field if you select this option.
Status	Displays the window status in rows 1 through 9 of column 80. Table 3-1 summarizes the status symbols displayed for the indicated states.
Cursor-wrap	Determines whether the cursor wraps at the screen edges. If you select this option, the cursor moves, for example, from row 3, column 80 to row 4, column 1 when you press the Right key and from row 24, column 5 to row 1, column 5 when you press the Down key.
Type-ahead	Determines whether the workstation stores keys that you enter while the keyboard is locked. If you select this option, the keystrokes are stored in a buffer and processed successively. This feature allows you to continue typing while you are waiting for system response.

Table 3-1. Window Status Symbols

Status	Row	Displayed Symbol	4230 Displayed Symbol
Active window	1	1, 2, 3, 4	1, 2, 3, 4
Keyboard locked	2	♦	
Entries in type-ahead buffer	2	■	
Insert mode	3	^	^
Caps Lock mode	4	↑	↑
Glossary key pressed	5	§	§
Invisible window	6	°	

Table 3-1. Window Status Symbols (continued)

Status	Row	Displayed Symbol	4230 Displayed Symbol
Entries in pick-up buffer	7	↕	∧
Anchor key pressed	8	.	‡
Glossary executing	9	ASCII value	ASCII value

After you select or remove each optional feature with the Insert or Delete key, pressing ENTER updates the edited personality with your selections. You can press PF10 to load the personality immediately. You can also press PF11 to copy the optional features selections from a prototype or specified existing personality. Pressing PF13 displays instructions; pressing PF16 returns control to the main menu.

3.8 DEFINING ACCENT KEY COMBINATIONS

When you select the Accent Combinations option from the main menu, you can define accented characters as combinations of an accent key and a character key. You invoke an accent combination by pressing the accent key (note that the keyboard must have a key dedicated to the particular accent key) and the key to be accented. For example, you generate a lowercase ä by pressing a key assigned (through Keyboard Layout) the hexadecimal value 07 and a lowercase a. In this way, the Multi-Station does not have to dedicate a key to each accented character combination, freeing the keyboard to perform other functions.

PERSON displays the Accent Combination screen, shown in Figure 3-9, when you select this option from the main menu. The Accent Combination screen allows you to modify a table of accent key and character combinations. You can define the accent keys used, the characters that can be accented, and the characters resulting from the accent combination. You can modify the default table in Hexadecimal or ASCII mode. Pressing PF5 displays the Accent Combination screen successively in Hexadecimal and ASCII modes, updating the table each time PF5 is pressed. You modify the table by positioning the cursor at the value you want to change, entering the new value in hexadecimal or ASCII (depending on the current screen mode), and pressing ENTER or PF5.

You can load the edited personality with the current accent combinations immediately by pressing PF10. You can also press PF11 to copy the accent table from a prototype or specified existing personality. Pressing PF13 displays instructions; pressing PF16 returns control to the main menu.

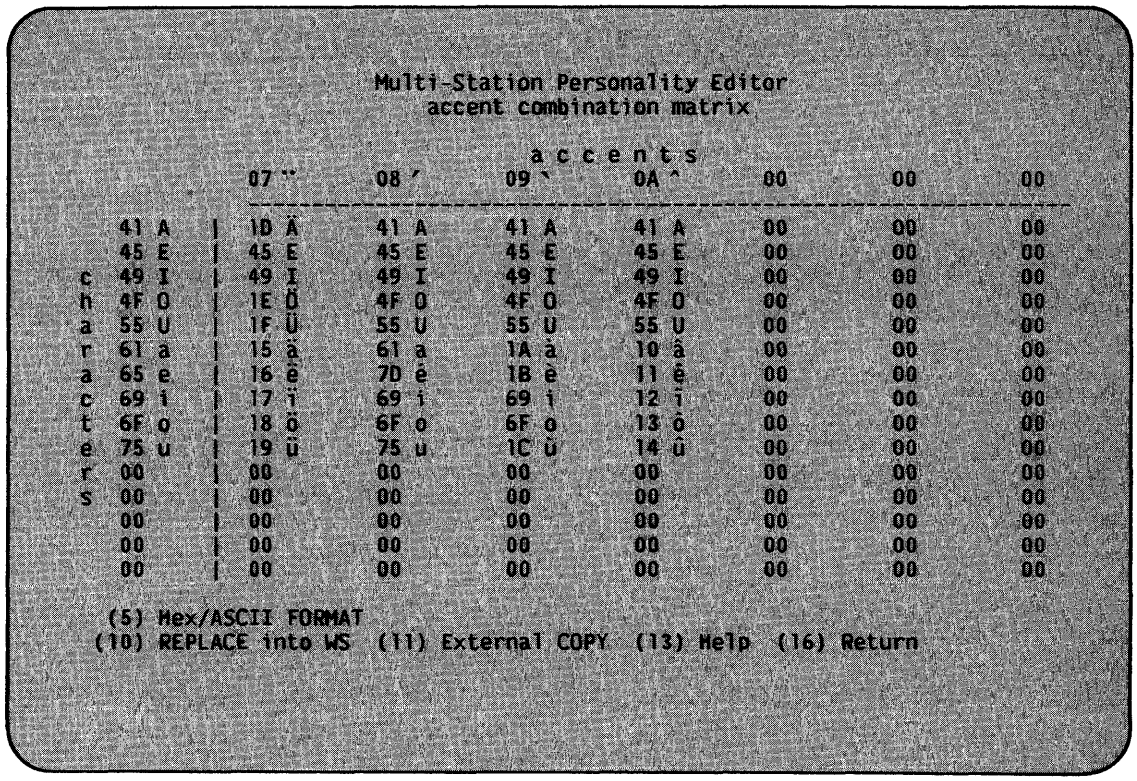


Figure 3-9. The Accent Combinations Screen

3.9 MODIFYING THE DEFAULT CAPITALIZATION RULES

When you select the Uppercase Fold-over Table option from the main menu, you can modify the way in which the workstation automatically capitalizes text you enter in uppercase-only fields. You can also indicate that a character cannot be automatically capitalized by setting the uppercase value equal to its lowercase value. PERSON allows you to modify the uppercase result of all characters in the character set; typically, however, only the lowercase accented characters require any modification.

PERSON displays the Fold-over Definition screen, shown in Figure 3-10, when you select the Uppercase Fold-over Table option. The Fold-over Definition screen displays 8 groups of 3 columns, each containing 16 entries. The first column in each group represents the character code entered on the workstation (the index character); the second column represents the corresponding hexadecimal code sent by the workstation to the screen and the VS. The third column represents the displayed value for the resulting value.

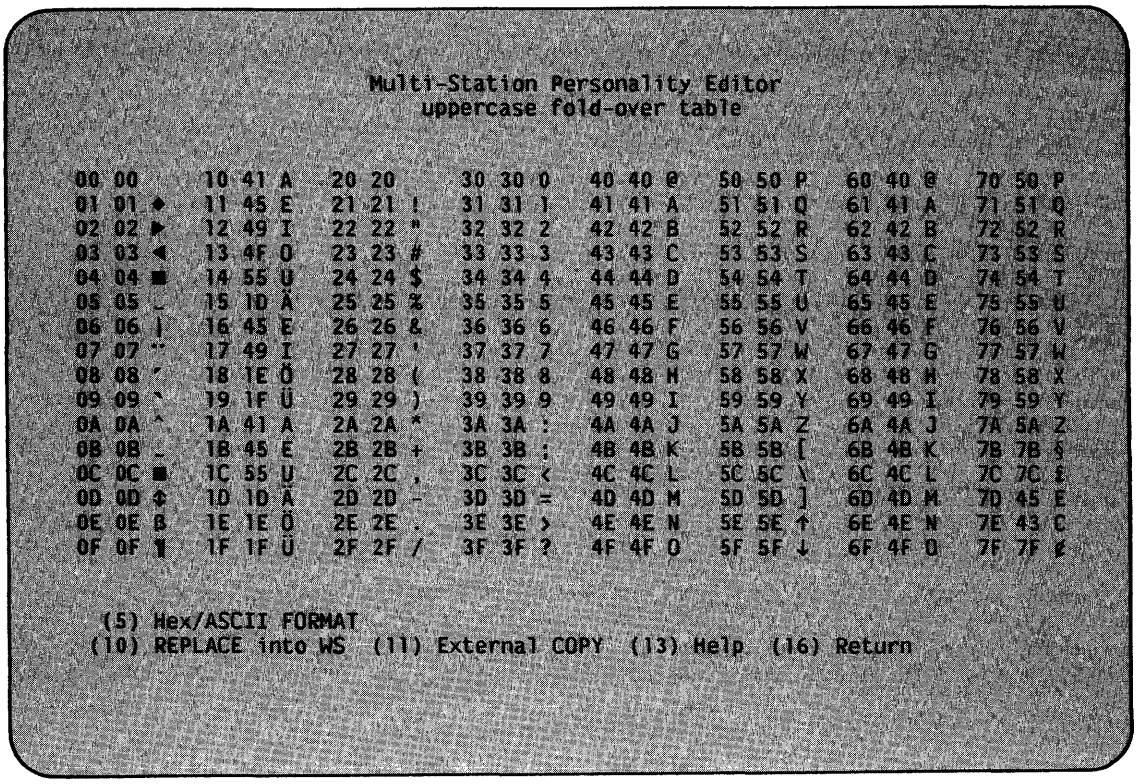


Figure 3-10. The Fold-over Definition Screen

When you first display the Fold-over Definition screen, the second column in each group is modifiable (Hexadecimal mode). Pressing PF5 from the Fold-over Definition screen changes the screen mode successively from Hexadecimal mode to ASCII mode and updates the table. If you return to the main menu and then reselect the Uppercase Fold-over option, the Character Set Definition screen is first displayed in the mode in which you exited the option.

You can modify the capitalization in Hexadecimal or ASCII mode. You modify the character by typing the hexadecimal code or the character corresponding to the desired uppercase result in the modifiable field corresponding to the index character to be changed and pressing ENTER or PF5 successively. The character displayed in the ASCII column is not changed until you update the table.

You can only modify the values corresponding to the ASCII character set because the VS uses the ASCII character set. However, if you have a Model 4230 workstation, which uses the WISCII character set, you can assign any WISCII value to any ASCII character position. Refer to Appendix C for the hexadecimal codes corresponding to the ASCII character set and to Appendix D for the hexadecimal codes corresponding to the WISCII character set.

You can load the personality with the current version of the character set into the workstation by pressing PF10 from the Fold-over Definition screen. You can also copy the table from an existing or prototype personality by pressing PF11. You can display instructions by pressing PF13 or return to the main menu by pressing PF16.

CHAPTER 4
THE GLOSSARY LANGUAGE

4.1 INTRODUCTION

The VS Glossary language is a simple, block-structured language that allows you to write programs that process workstation keystrokes and text. Through the Glossary language, you can receive, manipulate, and send keystrokes to the workstation. Because Glossary contains operators and functions, you can perform a full set of programming operations on the keystrokes.

The Glossary language resembles a greatly-simplified PL/I. Thus, PL/I programmers can write Glossary language programs almost immediately; programmers in other languages can learn the Glossary language quickly due to its simple structure.

Although your workstation personality contains only one object program, the block structure of the Glossary language allows a single program to perform a variety of functions. Each glossary function is written as a single procedure (subroutine). The program consists of a collection of procedures; refer to Section 4.3 for details on the structure of a Glossary program.

4.2 PROGRAM FORMAT

A Glossary language source program consists of a sequence of variable or procedure names, constants, comments, keywords, punctuation marks, and compiler-directing statements arranged on lines that conform to the VS EDITOR source format. The items that make up a program are known as tokens. The VS EDITOR reserves 8 columns for line numbers, allowing 72 columns of space on each line for programming. The line numbers used by the VS EDITOR are used only for editing convenience and have no significance in the source file.

Tokens must be organized into valid Glossary language syntax as described in this chapter. Tokens are separated by one or more blanks or by punctuation; the actual number of separating blanks is not significant, allowing you to format your program text for readability. You can include blank lines in your Glossary program.

4.2.1 Names

You can assign names to variables and procedures. Names consist of a sequence of letters, digits, and the underscore character and must begin with a letter. Names cannot contain embedded blanks or hyphens. A name can contain any number of characters, but the compiler checks only the first 32 characters. You can use both uppercase and lowercase text in variable or procedure names; however, because the compiler maps all values (including keyword specifications) to lowercase, you cannot define distinct names with values that differ only in case. Example Glossary language names follow:

```
a
TEXT
Key
OFFset
Move_Down
```

The Glossary language has a set of reserved words, listed in Appendix F.

4.2.2 Constants

Constants are a sequence of characters that represent specific values. Numeric constants can range in value from -32768 to 32767; character string constants represent alphanumeric values and are enclosed in matching quotation marks. Differences in uppercase and lowercase are significant in character string constants. Refer to Section 4.3 for more information on constants and data types. The Glossary language compiler considers the following values to be constants:

```
5
"this is a string constant"
'(-down-)'
```

4.2.3 Comments

Comments allow you to annotate the program text. The /* sequence signals the beginning of a commment, and the */ sequence indicates its end. Comments are equivalent to a space and can span one or more program lines. You can place a comment in any location where you can place a space. A comment can include any characters (including /*) except */. Example program comments follow.

```
a = b;      /* this is a comment */
if /* so is this */ a = b then call playout (a);
```

4.2.4 Keywords

The Glossary language contains a number of names, which when placed in a statement, indicate a specific language function or option. By placing keywords together, you construct statements, which indicate programming instructions. Glossary language keywords are case insensitive. Thus, THEN, Then, or then are all equivalent keyword specifications. The Glossary language keywords are described in the context of their containing statements; Appendix F contains a list of Glossary language keywords.

4.2.5 Punctuation

Glossary program text is separated with punctuation marks under certain conditions. When used, a punctuation mark serves as a separator and eliminates the need for a blank to separate the tokens. Punctuation marks include operator symbols (refer to Section 4.5 for a complete list of Glossary language operators), the semicolon (;), colon (:), the comma (,), and parentheses [()].

4.2.6 Compiler-Directing Statements

The VS Glossary language contains compiler-directives that alter the program format or perform compile-time operations on the source listings. All Compiler-directing statements begin with the % character and must begin in column 1 of the source file. The available statements are described as follows:

Statement	Description
%include id	Inserts the text of the file identified by id at this location in the source file. The value id should represent a file name in the same library as the source file. If the value does not reflect a file name, the compiler generates a request for the file location at compilation time. You can nest %include statements; that is, a file that is included can itself contain a %include statement.
%define id text	Substitutes the value text for the value id from this point on in the text. You must separate the value text from id by at least one blank. The text can extend to the end of the line, with trailing blanks removed.
%page	Issues a page feed when printing the program listing.
%noprint	Does not print the text following this statement when printing the source listing.
%print	Resumes printing the source listing (if such printing has been suppressed by %noprint)

Statement	Description
%pmap	Enables the preparation of object listing from this point in the text.
%nopmap	Disables the preparation of object listing from this point in the text.
%control	Enables the printing of compiler directives from this point in the text.
%nocontrol	Disables the printing of compiler directives from this point in the text. The compiler directives are not printed by default.

4.3 PROGRAM STRUCTURE

A Glossary language program consists of a collection of procedures and optional global declarations. A procedure is a group of statements that falls between a PROCEDURE statement and a corresponding END statement, and resembles a BASIC or FORTRAN subroutine, a COBOL Procedure Division, or PL/I procedure. Declarations are recognized as global by falling outside of any containing procedures. Only DECLARE statements can reside outside of a procedure in a Glossary language program.

Each procedure has a name that is determined by the label on the PROCEDURE statement. The name of the procedure is implicitly declared as a procedure name when used as a procedure label. The procedure's END statement can optionally specify the procedure name; the compiler checks the name if it is supplied. In the following example, procedure A is identified by a PROCEDURE statement and ends with a labeled END statement.

```
A: PROCEDURE;
    ...
END A;
```

A Glossary program does not execute each procedure sequentially from beginning to end. Glossary procedures are only executed when another procedure calls them or you invoke them by pressing designated workstation keys (refer to Section 4.3.3). Thus, a single Glossary program that is loaded into your workstation can perform a variety of functions.

Each procedure is a self-contained program, consisting of declarations and executable statements. Procedures can contain other procedures; such procedures are referred to as nested procedures. The following example demonstrates nested procedures:

```
A: Procedure;
    C: Procedure;
    ...
    End C;
    B: Procedure;
        ...
    End B;
End A;
```

Procedures define blocks in which variables are known. A variable is known in the procedure in which it is declared and in any procedures that are nested within the declaring procedure. If a nested procedure redeclares the variable, the nested procedure can no longer reference the original variable. The range of procedures in which a variable is known is the scope of the variable. In the following example, character variable A is known in procedures B and C, but not in procedures D and E.

```
B: Procedure;
    declare A char;
        C: Procedure;
            ...
        End C;
        D: Procedure;
            declare A fixed;
            ...
        End D;
End B;
E: Procedure;
    ...
End E;
```

Because global declarations do not fall within a procedure, they are known to all procedures within the Glossary source file that physically follow the declaration.

Like subroutines in other languages, you can also pass values to Glossary language procedures, allowing you to override scope restrictions. Because the Glossary language only passes the value of the variable, the called procedure cannot directly modify the variable in the calling procedure through the argument list. The number and the type of the arguments in the procedure call must match the number and the type of procedure parameters. An example procedure call with an argument list follows:

```
CALL A (B, c, 10);
```

The Glossary language contains four types of procedures: subroutines, functions, glossaries, and auto-start glossaries. You invoke each type of procedure differently. Subsections 4.3.1 through 4.3.4 describe subroutine, function, glossary, and auto-start procedures, respectively.

4.3.1 Subroutine Procedures

A subroutine procedure cannot be directly invoked from the workstation but is invoked by a CALL statement that specifies the procedure name and, optionally, an argument list. A subroutine procedure can only return values to the calling procedure by modifying common variables. A subroutine procedure cannot contain an OPTIONS or a RETURNS clause. In the following example, procedures A and B are subroutine procedures and must be invoked by a CALL statement.

```
B: PROCEDURE(x,y);
    ...
END B;
A: PROCEDURE;
    ...
    CALL B(Key,Buffer);
    ...
END A;
```

4.3.2 Function Procedures

Function procedures must contain a RETURNS clause in the procedure declaration. Functions produce a single value in the expression from which they are called. If the function appeares on the right side of an assignment statement, the resulting value is assigned to the variable on the left. Functions are called by a reference to the procedure name with its optional argument list. Because the Glossary language passes arguments by value, function procedures represent the only way of returning a value to a calling procedure.

The RETURNS clause identifies the data type of the value that the procedure returns; the data type is specified as fixed, character, or character(*). You cannot specify a specific length for character values because it is computed when you invoke the procedure. The following example illustrates the calling and declaration of function procedures:

```
lowercase: procedure(buffer) returns(char(*));
    dcl buffer ...

End lowercase;
A: Procedure;
    declare text char(50);
    ...
    text = lowercase(keybuffer);
    ...
End A;
```

4.3.3 Glossary Procedures

Glossary procedures are identified by the OPTIONS clause and are invoked when the user presses the Glossary key followed by the key indicated in the OPTIONS clause. A CALL statement can also invoke a Glossary procedure. You must specify the key in the OPTIONS clause as a keystroke string in the format described in Section 4.8.1 (e.g., (-pf-1-) or as a character string of length one (e.g., 'B').

NOTE

You will not be able to invoke the glossary unless the key specified in the OPTIONS clause is defined in your personality's keyboard layout. Also, if you use PF7 (Note), PF16, or the Glossary key as the key in the OPTIONS clause, you will not be able to create (PF7, PF16) or use (Glossary key) glossaries-by-example.

A Glossary procedure cannot contain an argument list. The following example illustrates a Glossary procedure that you invoke by pressing the Glossary key followed by B.

```
Run_BACKUP: Procedure Options ('B');
    call playout ('(-pf-1-)!!BACKUP!!(-enter-)');
End Run_BACKUP;
```

4.3.4 Auto-Start Procedures

An auto-start procedure is identified by the OPTIONS(MAIN) clause. You can invoke an auto-start procedure in all windows whenever you load the personality into the workstation. A Glossary program can contain only one auto-start procedure. An Auto-start procedure is typically used to log on all the windows on a Multi-Station or to initialize global variables. An Auto-start procedure can also be called by other procedures but cannot contain an argument list. An example Auto-start procedure declaration follows; refer to Appendix E for an example auto-start glossary that logs on all of a user's windows.

```
Auto_Logon: Procedure Options(Main);
    ...
End Auto_Logon;
```

4.4 PROGRAM CONTROL

The Glossary language supports the following statements that allow you to govern the flow of program control and to assign values to variables:

```
PROCEDURE
DECLARE
Assignment
IF ... THEN ... [ELSE]
DO (statement grouping)
DO (iterative)
DO WHILE(expression)
DO FOREVER
CALL
Function calling
LEAVE
RETURN
STOP
END
```

This section discusses each Glossary language statement in a separate subsection; the subsections are arranged alphabetically by statement name. Each subsection begins with the general format of the statement and follows with a description and syntax examples.

4.4.1 The Assignment Statement

```
variable_name = expression;
```

The assignment statement places the value of the expression on the right of the equals sign in the indicated variable. The data type of the receiving variable must match the expression's result because the Glossary language performs no automatic data type conversions. For assignments to a variable with the CHARACTER data type, the expression result is truncated or padded on the right with blanks to match the declared length of the variable name. Example assignment statements follow:

```
X = A + B;
Y = SUBSTR(x,1,3);
Key = Getkey;
```

4.4.2 The CALL Statement

```
CALL proc_name [(arg_list)] ;
```

The CALL statement invokes the indicated procedure and passes any specified arguments to the procedure. The referenced procedure must not be a function procedure. You can call only procedures that precede the CALL statement in the source listing; that is, you cannot call procedures that have not yet been defined. A procedure cannot call itself. When the procedure is called, the arguments are moved into the static locations reserved for the procedure parameters. Example CALL statements follow.

```
CALL Key_Process (Key);
Call Test;
```

4.4.3 The DECLARE Statement

```
{DECLARE} name type [{INITIAL(value)}] [, name type [{INITIAL(value)}] ... ] ;
{ DCL   }          [{INIT(value)    }]
```

The DECLARE statement assigns a data type to a name. A DECLARE statement that is within a procedure defines a name within the scope of that procedure. A DECLARE statement that has no containing procedure performs a global declaration. Refer to Section 4.3 for information on scoping rules and refer to Section 4.5 for information on data types and declaration.

You can also specify an initial value for the variable through the INITIAL attribute. For FIXED variables, the initial value can be any signed or unsigned numeric constant. For CHARACTER variables, the initial value can be a character string constant or an implied (without the concatenation operator) concatenation of character string constants. Example variable initializations follow:

```
INITIAL (5)
INITIAL ('stuff')
INITIAL ('(-pf-1-)')
INITIAL ('implicitly ' 'concatenated ' 'strings')
INITIAL ('(-pf-1-) BACKUP (-enter-) (-pf-1-)')
```

Example declaration statements follow:

```
declare A char(5);
dcl B fixed initial(5), C character(10), D char init('(-enter-)');
```

4.4.4 The DO Statement

The Glossary langauge supports four forms of the DO statement.

```
Form 1:     DO;
Form 2:     DO index_var = value_1 to value_2;
Form 3:     DO WHILE (expression);
Form 4:     DO FOREVER;
```

Form 1 allows you to group a sequence of statements. A statement grouping DO signals the beginning of a group of statements. The end of the group is signalled by a corresponding END statement. A statement grouping DO statement is required when more than one statement is to be processed after a THEN or an ELSE clause. An example of a statement grouping DO statement follows:

```
IF A = B THEN DO;
                CALL PLAYOUT ('(-pf-1-)');
                CALL WAITFORUNLOCK;
              END;
```

Form 2 allows you to repeat the sequence of statements between the DO statement and its corresponding END statement a specified number of times. Form 2 is known as an iterative DO statement and corresponds to a BASIC FOR ... NEXT loop, a FORTRAN DO ... CONTINUE loop, or a PL/I iterative DO loop. The index variable must be declared with the fixed data type and can be used within the DO-loop. The starting index value (value_1) is increased by one each time the loop is processed. You cannot specify the increment step as you can in some other languages. If value_2 is less than value_1, the loop is never executed. Unlike PL/I iterative DO statements, the Glossary language reevaluates value_2 each time the loop is executed. An example iterative DO statement follows:

```
DO I = 1 to 10;
    A = A +I;
END;
```

Form 3 of the statement allows you to process the group of statements between the DO statement and its corresponding END statement, while a specified relation is true. The specified relation must produce a fixed value and is interpreted as a logical result. The relation is tested before the loop is executed. The DO WHILE construction corresponds to its PL/I equivalent. An example DO WHILE group follows:

```
DO WHILE (A < B);
    A = A+1;
END;
```

Form 4 of the statement allows you to continue processing a DO group until an unrelated exit condition arises. If you do not include an exit condition within the DO-group, you create an infinite loop. The DO FOREVER construction is equivalent to a DO WHILE(1) statement.

An example DO FOREVER group follows:

```
DO FOREVER;
    Key = GetKey;
    If GetKey = '(-PF-1-)' then leave;
END;
```

4.4.5 The END Statement

```
END [procedure_name] ;
```

The END statement signals the end of a DO-group or a procedure. If you specify a procedure name, the compiler checks the value for consistency. You must have an END statement for each DO-group and procedure in the program. An END statement is associated with the nearest open DO-group or procedure unless it is labelled. The name is not required to end a procedure. Example END statements follow.

```
END Key_Process;
END;
```

4.4.6 Function Calling

```
Variable = function_name [(argument_list)] ;
```

You invoke function procedures by specifying the name of the function and its optional argument list on the right-hand side of an assignment statement or in an expression. You can call any function that precedes the calling procedure or that is a Glossary language built-in function. When the function is called, the arguments are moved into the static locations reserved for the function's parameters. Example function calls follow.

```
A = GETKEY;       /*  Returns the key the user entered  */
B = Test(Z, 1);   /*  Returns a value computed using Z and 1  */
```

4.4.7 The IF Statement

```
IF relation THEN statement ; [ELSE statement;]
```

The IF statement allows you to conditionally process a statement (or group of statements if you specify a DO-group) according to a specified relation. The specified relation must produce a fixed value and is interpreted as a logical result. If the relation is true, the statement specified in the THEN clause is executed. If the relation is false, execution continues with the evaluation of any ELSE clause. If no ELSE clause is present, the next statement in the procedure is processed. Example IF statements follow:

```
if A=B then call getkey;
IF (X/2 < 3) THEN DO;
                    ...
                  END;
             ELSE DO;
                    ...
                  END;
```

4.4.8 The LEAVE Statement

```
LEAVE;
```

The LEAVE statement transfers control outside its containing DO-loop (not DO-group). The LEAVE statement is only allowed in iterative DO, DO WHILE, and in DO FOREVER constructions. When a LEAVE statement is encountered, control passes out of the nearest containing DO-loop, regardless of the location of the LEAVE statement. For example, if a LEAVE is located in a DO-group that is within a DO-loop, control passes out of the DO-loop. Example LEAVE statements follow:

```
do forever;
    if a<b then leave;
end;

do i = 0 to 10;
    a = a +i;
    if a > 100 then do;
        a = 100;
        leave;    /* leaves the do i loop */
    end;
end;
```

4.4.9 The PROCEDURE Statement

```
name: {PROCEDURE} [(parameter_list)] [OPTIONS {(string)}] [RETURNS(type)];
      {PROC     }                    [       {MAIN   }]
```

The PROCEDURE statement defines a program block that is one of the four types of procedures described in Section 4.3. The name used to label the PROCEDURE statement is assigned as the name of the procedure. If you specify a parameter list, calls to the procedure must pass it a corresponding list of arguments. You cannot specify a parameter list or the RETURNS attribute if you specify the OPTIONS clause. The string parameter in the OPTIONS clause must be one character long. Example PROCEDURE statements follow:

```
A: PROCEDURE;
B: Procedure Options (main);
C: Procedure (x,y) returns (char(*));
D: Procedure (a,b,c,10);
E: Proc options('3');
```

4.4.10 The RETURN Statement

```
RETURN [(result)];
```

The RETURN statement returns control to the procedure that called the current procedure. You can only specify a result when returning from a function procedure. If you return a result, its data type must match that specified in the function's PROCEDURE statement RETURNS clause. All function procedures must execute a RETURN statement before an END statement. Example RETURN statements follow:

```
RETURN;
RETURN (x);
RETURN ('string');
```

4.4.11 The STOP Statement

```
STOP;
```

The STOP statement suspends execution until the user presses the Glossary key followed by PF16 or runs another glossary. An example STOP statement follows:

```
STOP;
```

4.5 DATA TYPES AND DECLARATION

The data type determines which Glossary language operations can be performed on a value and the amount of space the value reserves in the object program. Values can be named as variables or used as constants. You must assign all variable names a data type through the DECLARE statement. This section discusses the data types supported by the Glossary language, constants and variables, and the declaration process.

4.5.1 Data Types

The Glossary language contains two distinct data types: fixed integer and character string. Integer and Boolean values are declared with the FIXED data type; character values are declared with the CHARACTER data type.

FIXED Data

Values with the FIXED data type can only contain integer values. The Glossary language stores FIXED values as signed 16 bit numbers, equivalent to PL/I FIXED BINARY(15) values or COBOL BINARY integers. Thus, the Glossary language can represent integers ranging from -2^{15} to $2^{15}-1$ or from -32768 to 32767.

CHARACTER Data

The CHARACTER data type represents alphanumeric data. A CHARACTER value can contain any value in the VS character set, including spaces and quotation marks. You specify CHARACTER values within matching single or double quotation marks. You can place quotation marks in the CHARACTER value by specifying the quotation mark twice. CHARACTER values are often referred to as character strings; example character strings follow.

```
'Tristan'
"He said, ""Don't go"""
'value'
```

The length attribute determines the maximum number of characters that the character string can contain. The string always has the declared length, regardless of the number of characters in the string. Character strings assigned to variables with a longer declared length than the string are padded on the right with blanks; character strings assigned to variables with too small a declared length are truncated on the right. For example, a value of 'text' is interpreted as follows by character values with the indicated declared lengths:

Declared Length	Result
3	'tex'
4	'text'
10	'textbbbbbb'

CHARACTER values are limited to 255 characters in length. A character string can have a length of zero; such a string is referred to as a null string and is represented as '' or "". If you are specifying a string that exceeds the line length of the VS EDITOR, you can use the concatenation operator or a space (refer to Section 4.6) to continue the string. Key function specifications (refer to Section 4.8.1) compile into a single character. For example, the string "(-pf-1-)(-enter-)(-pf-1-)" is three characters long.

Boolean Data

Boolean values are represented as the least significant bit of a FIXED value. In general, even integer values are interpreted as a Boolean 0 and odd integer values are interpreted as a Boolean 1. The result of a successful logical operation is a -1; the result of an unsuccessful logical operation is a 0. Programs that depend on receiving a +1 as the result of a successful logical operation can convert the value to a +1 by anding the result with 1. Refer to the Search procedure in Appendix E for an example that converts the result of a relational operator.

4.5.2 Constants and Variables

The Glossary language supports two data representations: constants and variables. Arrays and structures are not supported.

A constant is a value in a Glossary language program that does not have an associated data name and that does not change in value. Constants can have the CHARACTER or FIXED data type. Numeric values have the FIXED data type; character strings have the CHARACTER data type. You cannot explicitly declare constants with a data type; the compiler assigns the data type to the constant when it allocates the constant's storage. The compiler assigns the length of the character string constant as the length of the string. Glossary language constants and their implicitly assigned data types are illustrated in the following example:

```
N = A*5;     /*  5 is a FIXED constant  */
C = "enter" /*  "enter" is a CHARACTER(5) constant  */
```

A data name that represents a single item with a declared data type and with a value that you can change is a Glossary language variable. A Glossary language variable can have the FIXED or CHARACTER data type.

4.5.3 Declaration

Each programmer-defined name in a Glossary language program, except procedure names and names created through %define, must be explicitly declared with a data type through the DECLARE statement. The data types supported by the Glossary language are described in Section 4.3.2; this section describes the declaration process.

The DECLARE statement associates a data type with a variable name and, optionally, can assign an initial value. The general form of the DECLARE statement is:

```
{DECLARE}name type [{INITIAL(value)}] [,name type [{INITIAL(value)}] ... ];
{DCL    }          [{INIT(value)   }]  [          [{INIT(value)   }]     ]
```

The data type must follow the name. If you omit the data type, the compiler issues an error messsage and does not assign a default data type. A single DECLARE statement can assign data types and initial values to as many variables as desired; each name, data type, and, if specified, initial value combination is separated from other name declarations by commas. PL/I programmers should note that you cannot factor attributes in the Glossary language.

You declare integer and Boolean variables by specifying the type FIXED. Example FIXED variable declarations follow:

```
DECLARE A FIXED;
DECLARE CURSOR_COLUMN_COUNT FIXED, CURSOR_ROW_COUNT FIXED;
```

You declare character string variables by specifying the type CHARACTER or its abbreviation CHAR. If you specify a string length, it must immediately follow the CHARACTER keyword and must be an integer constant enclosed in parentheses. If you omit the string length, the string has length one. Example string variable declarations follow:

```
DECLARE LINE_BUFFER CHAR(80);
DECLARE FIELD CHARACTER;
DECLARE A CHARACTER, B CHARACTER (10), C CHARACTER(5);
```

You can optionally declare an initial value: a numeric constant for a FIXED variable, a character string for CHARACTER variables. If you do not initialize the variable, the Glossary language automatically initializes FIXED variables to 0 and CHARACTER variables to spaces.

Variable declarations are subject to the scoping rules described in Section 4.2.

4.6 EXPRESSIONS

The Glossary language contains arithmetic, relational, and string operators to allow you to change or compare values. You can apply Glossary operators to constants, to variable references, functions, and built-in functions, and to other expressions. Operators can precede the operand (prefix operators) or link two operands (infix operators). The Glossary language supports the following operators:

Symbol	Function	Class
-	Sign change	Prefix arithmetic
+	Addition	Infix arithmetic
-	Subtraction	Infix arithmetic
*	Multiplication	Infix arithmetic
/	Division	Infix arithmetic
mod	Modulo	Infix arithmetic
=	Equal to	Infix relational
↑=, <>	Not equal to	Infix relational
>, ↑<=	Greater than	Infix relational
<, ↑>=	Less than	Infix relational
>=, ↑<	Greater than or equal to	Infix relational
<=, ↑>	Less than or equal to	Infix relational
!!	Concatenation	Infix string (character)
&, and	Boolean and	Infix string (fixed)
↑, not	Boolean not	Prefix string (fixed)
!, or	Boolean or	Infix string (fixed)

Because expressions can contain more than one operator and evaluation order affects the result, Glossary assigns operator priority to determine the order of expression evaluation. The Glossary language operators are listed from highest to lowest priority as follows; operators that appear on the same line have the same priority:

```
- (prefix), ↑, not
*, /, mod
+, -
!!
=, ↑=, <, >, <=, >=, <>, ↑<, ↑>, ↑<=, ↑>=
&, and
!, or
```

Infix operations having the same priority are evaluated from left-to-right; prefix operations with equal priority are evaluated from right-to-left.

You can alter the evaluation order by enclosing portions of the expression in parentheses. The operations within parentheses are performed prior to those outside parentheses; nested parentheses are evaluated from the inside out. Within parentheses, evaluation order is determined by standard operator priority. In the following example, which demonstrates evaluation order, variables A through E have values 1 through 5, respectively:

Expression	Result
X = A+B	X = 3
X = A+B*C	X = 7
X = A+(B*C)	X = 7
X = (A+B)*C	X = 9
X = (A*(B+C))+D	X = 9
X = A*B/C*D	X = 2
X = A*B/(C*D)	X = 0
X = A+B>C*E	X = 0

The arithmetic operators perform the standard algebraic functions on values with the FIXED data type. The Glossary language does not contain an exponential operator. Arithmetic operators require integer operands and always produce integer results.

The relational operators compare operands with the same data type. Divisions that produce fractional results are truncated to the next smaller integer. Regardless of the data types of the operands, however, the result of a relational operator is a fixed number with a value of 0 for unsuccessful comparisons and -1 (FFFFh) for successful comparisons.

Relational operators compare fixed and character operands differently. Fixed values are compared algebraically. Character values are compared left-to-right, one character at a time, only for equality or inequality. If the string lengths differ, the shorter value is extended on the right with blanks to the length of the longer value. Boolean values are logically limited to one bit; relational operators compare only the least significant bit of the fixed value. The value is considered true if the bit is 0 and false if it is 1.

For example, the following expressions have the indicated results:

Operand Values	Expression	Result
A=35; B=42	A > B	0
A='CAT'; B='DOG'	A ↑= B	-1
A = 5;	DO WHILE(A);	The loop is executed.

The Glossary language supports one string operator: concatenate, which is an infix operator with two character operands. The concatenate operator, !!, joins two strings, producing a single string result. The length of the resulting string is the sum of the lengths of the input strings. The resulting string cannot exceed 255 characters. The Glossary compiler also assumes a concatenate operation whenever two string constants are separated by one or more spaces or comments. The following example illustrates the operation of the concatenate operator.

Operand Values	Expression	Result
A='CAT'; B='DOG'	A !! B	'CATDOG'
	"CAT" "DOG"	"CATDOG"

The logical operators, & (and), ! (or), and ↑ (not), perform Boolean operations on fixed operands. The & and ! operators are infix operators requiring two operands, while the ↑ operator is a prefix operator that requires only one operand.

The & and ! operators compare each bit in the fixed operands. The & operator generates a true bit (1) only if the corresponding bits in each operand are 1; the ! operator generates a true bit if either operand has a 1 in the tested bit.

Examples of the & and ! operators follow:

Operand 1	Operand 2	& Result	! Result
0	0	0	0
1	-1	1	-1
1	0	0	1
7	5	5	7

The ↑ prefix operator logically negates (one's complement) the value of the operand. Thus, true values are converted to false results. Examples of ↑ operations follow:

Operand	↑ Result
0	-1
-1	0
5	-6

4.7 GENERAL BUILT-IN FUNCTIONS

The Glossary language incorporates a set of built-in functions that allows you to perform general text manipulation and value testing. Each built-in function is described separately in the following subsections. The subsection begins with the general form of the function followed by a description and examples.

4.7.1 BINARY

```
BINARY(string_value)
```

The BINARY built-in function converts an input character string to a value with the FIXED data type. For example, an input string of "80" is converted to the numeric value of 80. BINARY converts the null string, or any string that does not represent a numeric value, to a value of 0. Example results of the BINARY built-in function follow:

Example	Result
BINARY('72')	72
BINARY('4.5')	4
BINARY("")	0

4.7.2 BYTE

```
BYTE(numeric_value)
```

The BYTE built-in function converts a small positive integer (ranging from 0 to 255) into a 1-byte ASCII character. Example results of the BYTE built-in function follow:

Example	Result
BYTE(65)	'A'

4.7.3 CHAR

```
CHAR(numeric_value)
```

The CHAR built-in function converts the input numeric value to a character string representation. If the input number is negative, the output character string begins with a minus sign. Example results of the CHAR built-in function follow:

Example	Result
CHAR(-321)	'-321'
CHAR(29)	'29'

4.7.4 INDEX

```
INDEX(Searched_string, Substring)
```

The INDEX built-in function searches the first input string for an exact occurrence of the second input string. If the second string is embedded in the first string, INDEX returns its starting location as an integer constant. If the search was unsuccessful, INDEX returns 0. Example results of the INDEX built-in function follow:

Example	Result
INDEX('abcdefg', 'de')	4
INDEX('123','5')	0

4.7.5 LENGTH

```
LENGTH(string)
```

The LENGTH built-in function returns the length of the input string as an integer constant. For string variables, LENGTH returns the declared length. Example results of the LENGTH built-in function follow:

Example	Result
LENGTH(b)	5 (the variable b was declared as CHAR(5))
LENGTH('abc')	3
LENGTH('abc'!!"def")	6

4.7.6 RANK

```
RANK(string)
```

The RANK built-in function returns the ASCII character code (as an integer constant) of the first character of the input string. Example results of the RANK built-in function follow:

Example	Result
RANK('A')	65
RANK('AB')	65

4.7.7 SUBSTR

You can use SUBSTR as either a function or pseudovariable. The syntax for using SUBSTR as a function follows:

```
SUBSTR(string, first, length)
```

When you use SUBSTR as a function (e.g., on the right of an assignment statement), SUBSTR returns the portion of the input string from the indicated starting location for the designated length. If the designated length exceeds the remaining length of the string, the resulting string is padded on the right with blanks. Example results of the SUBSTR function follow:

Example	Result
SUBSTR('abcde', 2, 2)	'bc'
SUBSTR("ABC", 2, 3)	"BC "

The second form of the SUBSTR function allows you to replace the indicated portion of the string with a specified value.

```
SUBSTR(destination_string, first, length) = value;
```

When you use SUBSTR on the left of an assignment statement, SUBSTR places the indicated value in the portion of the destination string beginning at the indicated starting location (first) for the designated length. The value is padded on the right with blanks, if necessary. PL/I programmers should note that the length cannot be omitted. If the specified length exceeds the length of the subject string, the entire string is used. Example results of SUBSTR assignment follow; in each case the original value of the destination string is "xxxxxxxxxx".

Statement	Resulting Destination String
SUBSTR(string, 3, 5) = 'aaa';	'xxaaa xxx'
SUBSTR(string, 1, 3) = 'yyy';	'yyyxxxxxxx'

4.7.8 VERIFY

```
VERIFY(Searched_string, Set_string)
```

The VERIFY built-in function compares the first input string byte-for-byte with the second string and returns the byte position (as an integer constant) of the first byte in the first string that does not occur in the second string. If the string lengths differ, the shorter string is padded on the right with blanks to match the length of the second string. If all of the characters in the first string are found in the second string, VERIFY returns 0. Example results of the VERIFY built-in function follow.

Example	Result
VERIFY('abc','def')	1
VERIFY('1.36x','0123456789.')	5

4.8 ACCESSING THE WORKSTATION

The Glossary language incorporates a set of functions and subroutines that allow you to control the workstation. You can write to and read from the screen in a variety of manners and communicate workstation information to and from the program. The language also includes a syntax for specifying keystrokes. This section describes the Glossary language workstation interface.

4.8.1 Keystroke Syntax

The Glossary language has a specific syntax for keystrokes that allows you to specify both workstation key functions and any alphanumeric string that corresponds to a value typed on the workstation. You can specify any Multi-Station key function through the Glossary language, whether or not you dedicate a key to the function through the PERSON utility. Note, that a key specified in the OPTIONS clause must be defined on the keyboard in order to invoke the glossary.

Keystrokes that correspond to data entered on the workstation are specified as character strings. For example, the string "BACKUP" is equivalent to typing the text on the screen.

You distinguish key functions from ordinary keystrokes by enclosing the functions in parentheses and setting them off by dashes. Thus, the general form for a key function entry is as follows:

```
(-function name-)
```

A key function specification is equivalent to one character in a character string. As a result, you can specify a single character string as "(-pf-1-)BACKUP(-enter-)". Table 4-1 relates the Glossary language syntax for each Multi-Station key function; refer to Chapter 3 for definitions of all the key functions.

Table 4-1. Glossary Language Key Function Syntax

Key Function	Syntax
add col left	'(-ad-lf-)'
ADD col left	'(-AD-lf-)'
add col right	'(-ad-rt-)'
ADD col right	'(-AD-rt-)'
add row down	'(-ad-dn-)'
ADD row down	'(-AD-dn-)'
add row up	'(-ad-up-)'
ADD row up	'(-AD-up-)'
again	'(-again-)'
anchor	'(-anchor-)'
ASCII x	'x'
back line	'(-bklin-)'
back space	'(-bkspc-)'
back tab	'(-bktab-)'
help	'(-help-)'
caps lock	'(-lock-)'

Table 4-1. Glossary Language Key Function Syntax (Continued)

Key Function	Syntax
caps unlock	'(-unlock-)'
cursor to 1,1	'(-1,1-)'
del row down	'(-dl-dn-)'
DEL row down	'(-DL-dn-)'
del col left	'(-dl-lf-)'
DEL col left	'(-DL-lf-)'
del col right	'(-dl-rt-)'
DEL col right	'(-DL-rt-)'
del row up	'(-dl-up-)'
DEL row up	'(-DL-up-)'
delete	'(-delet-)'
down	'(-down-)'
enter	'(-enter-)'
erase	'(-erase-)'
freeze screen	'(-freez-)'
full size wnd	'(-full-)'
get next wnd	'(-get-n-)'
get prev wnd	'(-get-p-)'
global glossary	not allowed
glossary	'(-gl-)'
help and reset	'(-help!-)'
hex xx	'(-0xxh-)'
home	'(-home-)'
insert	'(-insrt-)'
insert mode	'(-insmd-)'
invisible window	'(-invis-)'
invoke gl x	not allowed
left	'(-left-)'
look down	'(-lk-dn-)'
look left	'(-lk-lf-)'
look right	'(-lk-rt-)'
look up	'(-lk-up-)'
mov wnd down	'(-mv-dn-)'
MOV wnd down	'(-MV-dn-)'
mov wnd left	'(-mv-lf-)'
MOV wnd left	'(-MV-lf-)'
mov wnd right	'(-mv-rt-)'
MOV wnd right	'(-MV-rt-)'
mov wnd up	'(-mv-up-)'
MOV wnd up	'(-MV-up-)'
new line	'(-nline-)'
next window	'(-n-wnd-)'
pf xx	'(-pf-xx-)' or '(-pf-x-)'
pick up	'(-picup-)'
prev window	'(-p-wnd-)'
put down	'(-putdn-)'
recall	'(-rclsz-)'
reset	'(-reset-)'
right	'(-right-)'
space	'(-space-)'
tab	'(-tab-)'
up	'(-up-)'

4.8.2 Workstation Subroutines

The Glossary language incorporates a number of subroutines that control the keyboard. You can invoke these subroutines directly with the CALL statement. The available subroutines are described as follows in alphabetical order:

Delay

```
CALL DELAY(value);
```

When the DELAY subroutine is called, the Glossary processor pauses for the approximate amount of time you specify in the value argument. The value is expressed as an integer value in tens of milliseconds. Example calls to the DELAY subroutine follow:

```
Call Delay(100);  /* pause for 1 second */
Call delay(x/2);
```

Highlight

```
CALL HIGHLIGHT(Row, Column, Length);
```

The HIGHLIGHT subroutine causes the area of the screen represented by the argument list to blink. Note that the Highlight subroutine overrides the Blink specification in the PERSON utility. The row and column specifications represent the first screen location to be highlighted; the length determines the number of additional characters on that row which should also be highlighted. Example calls to the HIGHLIGHT subroutine follow:

```
Call highlight(1,10,70);
call HIGHLIGHT(3,CursorCol,1);
```

Playout

```
CALL PLAYOUT(string);
```

The PLAYOUT subroutine allows the procedure to issue keystrokes. You specify the keystrokes to be issued as string constants or variables in the call to PLAYOUT. A single call to PLAYOUT can specify up to 255 keystrokes. Each keystroke is a unique character string. Keystroke character strings can be separated by blanks or the concatenate operator within the argument list. If you specify the keystroke as a variable reference, you must use the concatenate operator to specify additional keystrokes in the call; additional keystrokes that you specify as constants do not require the concatenate operator and can be separated by blanks (or a comment, which is equivalent to a blank). PLAYOUT sends each separate keystroke to the screen in a manner equivalent to a user typing the keys directly on the keyboard. Example calls to PLAYOUT follow:

```
CALL PLAYOUT ('(-pf-1-)BACKUP(-tab-)@SYSTEM@(-tab-)SYSTEM');
CALL PLAYOUT ('A'!!substr(screen(5),1,2));
```

SendPFKey

```
CALL SENDPFKEY(aid_value);
```

The SENDPFKEY subroutine issues the PF key represented by its AID value. You specify the AID value as an integer value. Refer to Appendix G for a list of the AID values for each PF key. Example calls to the SENDPFKEY subroutine follow:

```
CALL SENDPFKEY(65);  /* Issues PF1 */
CALL SendPFKey(66);  /* Issues PF2 */
```

WaitForUnlock

```
CALL WAITFORUNLOCK;
```

The WAITFORUNLOCK subroutine causes the procedure to stop executing until the keyboard is unlocked. The subroutine is useful for ensuring that the system has responded to previous writes to the screen before reading from it. For example, the following Glossary program (invoked while in the VS EDITOR) could produce unexpected results if the call to WAITFORUNLOCK were omitted because the system may not have completed writing 'abc' before the next call to playout attempts to read what was written.

```
a: proc;
    call playout('(-pf-11-)abc(-enter-)');
    call waitforunlock;
    call playout('(-pickup-)');
end a;
```

4.8.3 Workstation Functions

The Glossary language includes a set of special built-in functions that are designed specifically to extract workstation information for the program. The provided functions are describes as follows in alphabetical order:

Clock

```
CLOCK;
```

The clock function returns a time in tens of milliseconds. The receiving variable must have the fixed data type. An example statement that uses the CLOCK built-in function follows:

```
A = clock;
```

CursorChar

CURSORCHAR

The CURSORCHAR function returns the character at the current cursor location. If you assign the value to a variable, the receiving variable must have the character data type. Example references to the CURSORCHAR function follow:

```
A = CursorChar;
B = 'A'!!CursorChar;
```

CursorCol

```
Form 1:     variable_name = CURSORCOL;

Form 2:     CURSORCOL = expression;
```

Form 1 of the CURSORCOL function allows you to extract the current column location of the cursor and place it in a variable with the fixed data type. Form 2 of the CURSORCOL function moves the cursor to the column indicated by the specified integer-valued expression. Examples of the CURSORCOL function follow:

```
CURSORCOL = 5;      /* Moves the cursor to column 5 */
X = CursorCol;
```

CursorRow

```
Form 1:     variable_name = CURSORROW;

Form 2:     CURSORROW = expression;
```

Form 1 of the CURSORROW function allows you to extract the current row location of the cursor and place it in a variable with the fixed data type. Form 2 of the CURSORROW function moves the cursor to the row indicated by the specified integer-valued expression. Examples of the CURSORROW function follow:

```
CURSORROW = 5;      /* Moves the cursor to row 5 */
X = CursorRow;
```

GetKey

GETKEY

The GETKEY function causes the procedure to wait for the user to enter a keystroke and then returns the keystroke to the program. Thus, you can set a variable equal to the key that a user entered. Example GETKEY references follow:

```
Key = GetKey;
NewKeys = GetKey!!'1';
```

Screen

```
SCREEN (row_number);
```

The SCREEN function returns the contents of the screen on the specified row. If used in an assignment statement, the receiving variable must have the character data type; if the receiving variable has a length less than 80, normal string truncation occurs. You can also use the SCREEN function anywhere a character string reference is valid. SCREEN always returns the entire screen line; the SUBSTR function allows you to extract a portion of the line. Example references to the SCREEN function follow:

```
A = SCREEN(1);
PickUp = substr(screen(4),22,3);
```

TypingRate

```
TYPINGRATE = numeric_expression;
```

The TYPINGRATE function allows you to set the approximate typing rate of a PLAYOUT operation. The numeric expression must evaluate to an integer constant that expresses the rate in tens of milliseconds per character. The TYPINGRATE function is simulating different typing rates. An example reference to TYPINGRATE follows:

```
Typingrate = 100/2;
```

Window

```
WINDOW
```

The WINDOW function returns the value of the current window as a fixed integer. Example references to the WINDOW built-in function follow:

```
call playout ("ML"!!char(window));
b = window;
```

4.9 PROGRAMMER'S NOTES

4.9.1 Boolean Values in Glossary Procedures

Boolean values are represented in Glossary procedures as variables with the FIXED data type. The Glossary language uses -1 (FFFFh) to indicate True (success) and 0 to indicate False (failure). Thus, a comparison of equal values yields -1 and a comparison of unequal values yields 0. To convert the result to the usual PL/I values (0 for false, 1 for true), AND the result with 1. The AND, OR, and NOT operations are performed in a bit-wise fashion on values of FIXED type.

4.9.2 Simultaneity in Glossaries

A single Glossary procedure can run independently in each window simultaneously. Auto-start glossaries (the procedure containing the options(main) clause) and glossaries invoked through the Global Glossary key run in all windows. The variables in a Glossary program are equally accessible to all windows because they have the equivalent of the PL/I STATIC storage class. As a result, values set by a procedure running in one window can be destroyed by a procedure running in another window. However, except for using the same variables, there is no way for what is happening in one window to affect what is happening in another.

Each window runs its glossary, executing a certain number of instructions (its time-slice), and then switches to another window in a round-robin fashion. If a glossary exceeds its time-slice, stops (through the STOP statement or the Reset key), or is forced to wait (waitforunlock, getkey, delay, or playout), a glossary in another window can begin executing. If a glossary is waiting, it will automatically resume after the waiting condition is satisfied. If it is stopped, you can either invoke a fresh glossary on top of it or you can restart the glossary from where it was stopped by pressing the Glossary key followed by PF16 while its window is active.

Independently of what is occurring in other windows, you can invoke a Glossary procedure by pressing the Glossary key (and stop the glossary by pressing Reset) in the active window. You can also change the active window by pressing the Get Next Window, Get Previous Window, Next Window or Previous Window keys, which act directly, and bypassing the GetKey function. It is often hard to predict the results if a glossary is playing out window switching commands and they are also being simultaneously entered from the keyboard.

APPENDIX A
SYSTEM ADMINISTRATION

A.1 INTRODUCTION

The system configuration determines which workstations can support the VS Multi-Station software and how many windows a particular Multi-Station has. This appendix describes the system requirements, the procedure for configuring Multi-Stations, and the impact of Multi-Stations on the number of available tasks.

A.2 SYSTEM REQUIREMENTS

The VS Multi-Station package runs on the following VS systems:

VS-25
VS-45
VS-50, VS-60, or VS-80 with a 22V17 IOP at Revision 1 (R1) or higher
VS-85, VS-90, or VS-100 with a 22V27 IOP at R3 or higher

The VS Multi-Station package operates on the following workstations:

2256C
5300/VS-IIS64 Ergo 3
2866C4 Ergo 2
Wang PC in VS emulation mode with PC-PM041 local communications
4230

The VS Multi-Station package requires Release 6.20 or later of the VS Operating System.

A.3 CONFIGURING A VS WORKSTATION AS A MULTI-STATION

You define a workstation as a Multi-Station through the GENEDIT utility. The VS System Administrator's Reference describes the general use of the GENEDIT utility; this section describes the steps you must follow to configure the system with Multi-Stations and assumes familiarity with GENEDIT.

You can consider a Multi-Station to be one physical device that is connected to up to four logical tasks. Through GENEDIT, you assign a port and a device type to a device number (task). A Multi-Station with four windows is defined as four separate device numbers. However, because the Multi-Station is only one physical device, each device number is assigned to the same port.

The device type of the Multi-Station depends only on the window number represented by that particular task. All Multi-Stations, regardless of the workstation model, have the device type MULTIWSx, where x ranges from 0 to 3. The task with device type MULTIWS0 corresponds to window 1; MULTIWS2 corresponds to window 3. You must assign the device types in ascending order; for example, a Multi-Station with two windows must have device types MULTIWS0 and MULTIWS1. You cannot define a Multi-Station with only device types MULTIWS1 and MULTIWS2. Note that the device numbers do not have to be consecutive; three windows on a Multi-Station can have, for example, device numbers 7, 23, and 11. You should also note that because IOPs typically have only twice as many device numbers as ports, you may have unused ports on an IOP that supports a large number of Multi-Stations with a large number of windows.

NOTE

You cannot define the main Operator's Console (device number 0 and port number 0) as a Multi-Station.

You must initialize the system to activate the modified configuration file. When the system initialization is complete, each configured Multi-Station contains a default personality that readily identifies it as Multi-Station by its use of the Dec Tab character instead of a pseudoblank.

The following example illustrates the process of configuring a system with three Multi-Stations with differing numbers of windows:

Device Number	Device Type	Port Number
1	MULTIWS0	1
4	MULTIWS1	1
5	MULTIWS2	1
7	MULTIWS3	1
8	MULTIWS0	2
9	MULTIWS0	3
11	MULTIWS1	3
12	MULTIWS2	3

The Multi-Station attached to port number 1 has four windows, with device numbers 1, 4, 5, and 7. The Multi-Station attached to port 2 has one window, with device number 8. The Multi-Station attached to port 3 has three windows, identified by device numbers 9, 11, and 12, respectively.

Figure A-1 shows two sample screens from GENEDIT that contain the configuration for a serial IOP that controls workstations, Multi-Stations, and printers.

```
                    ***  WANG VS GENEDITOR  ***

                        Extended Serial IOP #4 BA#1
Dev Type     Description          Port   WP
 34 2246C___ Combined Wrk Station   _0
 35 MULTIWS0 Multi-task W/S         _1
 36 MULTIWS0 Multi-task W/S         _2
 37 MULTIWS0 Multi-task W/S         _3
 38 2246C___ Combined Wrk Station   _4
 39 2246C___ Combined Wrk Station   _5
 40 MULTIWS0 Multi-task W/S         _6
 41 MULTIWS0 Multi-task W/S         _7
 42 6581W___ Daisy Prtr, 40 CPS     _8    Y
 43 6581W___ Daisy Prtr, 40 CPS     _9    Y
 44 6581W___ Daisy Prtr, 40 CPS     10    Y
 45 6581W___ Daisy Prtr, 40 CPS     11    Y
 46 DWDS20__ Daisy Prtr, 20 CPS     12    Y
 47 MULTIWS0 Multi-task W/S         13
 48 MULTIWS0 Multi-task W/S         14
 49 MULTIWS0 Multi-task W/S         15

                    Press (ENTER) to continue
( 2) Show valid device types     ( 5) Next devices on IOP   (31) IOP     menu
                                 (16) Summary menu          (32) General menu
```

```
                    ***  WANG VS GENEDITOR  ***

                        Extended Serial IOP #4 BA#1
Dev Type     Description          Port   WP
 50 MULTIWS1 Multi-task W/S         15
 51 MULTIWS1 Multi-task W/S         _2
 52 MULTIWS1 Multi-task W/S         _3
 53 MULTIWS1 Multi-task W/S         _7
 54 MULTIWS2 Multi-task W/S         _7
 55 MULTIWS2 Multi-task W/S         15
 56 MULTIWS3 Multi-task W/S         15
 57 MULTIWS1 Multi-task W/S         _1
 58 MULTIWS2 Multi-task W/S         _2
 59 MULTIWS3 Multi-task W/S         _2

                    Press (ENTER) to continue
( 2) Show valid device types                          (31) IOP     menu
( 4) Previous devices on IOP     (16) Summary menu    (32) General menu
```

Figure A-1. Sample GENEDIT Screens

A.4 TASK RESTRICTIONS

Although a Multi-Station only occupies one port on an IOP, it reserves as many tasks as it has windows. The maximum number of tasks on the system is unchanged, but the number of physical workstations that can be supported is directly reduced by each additional window on a Multi-Station. For example, if a system could previously support 32 workstations and configures 3 Multi-Stations, each with four windows, the newly-configured system can support only 23 workstations. The effect on system performance of four programs running on a Multi-Station with four windows is roughly comparable to that of those programs running on four individual workstations.

APPENDIX B
MULTI-STATION UTILITY GETPARM REQUIREMENTS

B.1 INTRODUCTION TO GETPARMS

The VS Operating System supports a supervisor call routine (SVC), the GETPARM SVC, that solicits and accepts runtime parameter information, and displays and awaits acknowledgement of runtime messages. GETPARM-generated prompts are displayed on the workstation screen during normal execution. These prompts solicit parameter information from you or from a controlling procedure. Values entered from either source are verified. If the values entered are not acceptable, the GETPARM SVC responds with an error message.

GETPARM processing is distinguished from other methods of obtaining runtime information primarily because it can interface with a procedure. (Refer to the VS Procedure Language Reference.) A procedure is the preferred source of information for a GETPARM request; GETPARM prompts never appear on the workstation screen when they are satisfied by a Procedure language ENTER statement. Therefore, you can precisely control the interaction between a user and an executing program. GETPARM requests are used wherever possible by the VS system programs to solicit parameter information. GETPARM processing enables you to run system programs with little or no user interaction by supplying most or all of the required program parameters from procedures.

B.2 THE STRUCTURE OF A GETPARM

Each GETPARM request in a program is identified with a parameter reference name (prname). The prname for each request is, in general, unique within that program. System programs generally observe certain conventions when identifying GETPARM requests with prnames; for example, a GETPARM request soliciting information for an input file is usually identified with the prname INPUT, while a GETPARM soliciting parameters for an output file is named OUTPUT.

Many GETPARM requests for information contain one or more modifiable fields into which you or a procedure can enter information. Each field is labeled with a keyword. When a GETPARM request is displayed, the keyword for each field provides a description of the information to be supplied for that field. Certain conventions are commonly used in keyword naming. For example, a request for a file name often uses the keyword FILE. Also, many GETPARM requests solicit a PF key response (such as 16 = Exit Program). No keyword is associated with a PF key choice; only the PF key number itself is specified.

APPENDIX D
THE WISCII CHARACTER SET

Low order 4 Bits hex-digit		High order 4 bits hex-digit: 0_{0_0}	0_{0_1}	0_{1_0}	0_{1_1}	1_{0_0}	1_{0_1}	1_{1_0}	1_{1_1}	0_{0_0}	0_{0_1}	0_{1_0}	0_{1_1}	1_{0_0}	1_{0_1}	1_{1_0}	1_{1_1}
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0000	0	Reserved and should not be used.		SPACE	0	@	P	`	p	°	Â	â	Ğ	ğ	Þ	þ	£
0001	1			!	1	A	Q	a	q	♦	À	à	IJ	ij	Đ	ð	*f*
0010	2			"	2	B	R	b	r	▶	Á	á	İ	i	Ý	ý	¥
0011	3			#	3	C	S	c	s	◀	Ä	ä	Î	î	Ş	ş	¼
0100	4			$	4	D	T	d	t	→	Ã	ã	Ì	ì	‘	’	½
0101	5			%	5	E	U	e	u	↵	←	→	Í	í	Û	û	¾
0110	6			&	6	F	V	f	v	|	Å	å	Ï	ï	Ù	ù	ˆ
0111	7			’	7	G	W	g	w	■	↓	†	L·L	I·I	Ú	ú	ˋ
1000	8			(	8	H	X	h	x	!!	Æ	æ	Ñ	ñ	Ü	ü	´
1001	9			)	9	I	Y	i	y	↕	Ç	ç	Ô	ô	©	™	¨
1010	A			*	:	J	Z	j	z	↓	‡	□	Ò	ò	®	¤	˜
1011	B			+	;	K	[	k	{	↑	●		Ó	ó	₨	↔	¸
1100	C			,	<	L	\	l	|	←	Ê	ê	Ö	ö	ª	º	˅
1101	D			-	=	M	]	m	}	±	È	è	Õ	õ	«	»	ˇ
1110	E			.	>	N	↑	n	~	¡	É	é	Œ	œ	§	ß	
1111	F			/	?	O	_	o	¢	¿	Ë	ë	Ø	ø	¶	.	

APPENDIX E
A SAMPLE GLOSSARY PROGRAM

The following example Glossary program contains a variety of functions and illustrates the power of the Glossary language:

```
/*                  G L O B A L   K E Y   N A M E S                 */
/*                  Accessible from all procedures.                 */

%define OneOne      '(-1,1-)'
%define AddDown3    '(-AD-dn-)'
%define AddLeft10   '(-AD-lf-)'
%define AddRight10  '(-AD-rt-)'
%define AddUp3      '(-AD-up-)'
%define DelDown3    '(-DL-dn-)'
%define DelLeft10   '(-DL-lf-)'
%define DelRight10  '(-DL-rt-)'
%define DelUp3      '(-DL-up-)'
%define MovDown3    '(-MV-dn-)'
%define MovLeft10   '(-MV-lf-)'
%define MovRight10  '(-MV-rt-)'
%define MovUp3      '(-MV-up-)'
%define AddDown1    '(-ad-dn-)'
%define AddLeft1    '(-ad-lf-)'
%define AddRight1   '(-ad-rt-)'
%define AddUp1      '(-ad-up-)'
%define Again       '(-again-)'
%define Anchor      '(-anchr-)'
%define BackLine    '(-bklin-)'
%define BackSpace   '(-bkspc-)'
%define BackTab     '(-bktab-)'
%define Delete      '(-delet-)'
%define DelDown1    '(-dl-dn-)'
%define DelLeft1    '(-dl-lf-)'
%define DelRight1   '(-dl-rt-)'
%define DelUp1      '(-dl-up-)'
%define Down        '(-down-)'
%define Enter       '(-enter-)'
%define Erase       '(-erase-)'
%define Freeze      '(-freez-)'
%define Full        '(-full-)'
%define GetNext     '(-get-n-)'
%define GetPrev     '(-get-p-)'
```

```
/*                  G L O B A L   K E Y   N A M E S                  */
/*                           (continued)                             */

%define Gl           '(-gl-)'
%define Help         '(-help-)'
%define OhHelp       '(-help!-)'
%define Home         '(-home-)'
%define InsertMode   '(-insmd-)'
%define Insert       '(-insrt-)'
%define Invisible    '(-invis-)'
%define Left         '(-left-)'
%define LookDown1    '(-lk-dn-)'
%define LookLeft1    '(-lk-lf-)'
%define LookRight1   '(-lk-rt-)'
%define LookUp1      '(-lk-up-)'
%define CapsLock     '(-lock-)'
%define MovDown1     '(-mv-dn-)'
%define MovLeft1     '(-mv-lf-)'
%define MovRight1    '(-mv-rt-)'
%define MovUp1       '(-mv-up-)'
%define NextWindow   '(-n-wnd-)'
%define NewLine      '(-nline-)'
%define PrevWindow   '(-p-wnd-)'
%define PF1          '(-pf-1-)'
%define PF2          '(-pf-2-)'
%define PF3          '(-pf-3-)'
%define PF4          '(-pf-4-)'
%define PF5          '(-pf-5-)'
%define PF6          '(-pf-6-)'
%define PF7          '(-pf-7-)'
%define PF8          '(-pf-8-)'
%define PF9          '(-pf-9-)'
%define PF10         '(-pf-10-)'
%define PF11         '(-pf-11-)'
%define PF12         '(-pf-12-)'
%define PF13         '(-pf-13-)'
%define PF14         '(-pf-14-)'
%define PF15         '(-pf-15-)'
%define PF16         '(-pf-16-)'
%define PF17         '(-pf-17-)'
%define PF18         '(-pf-18-)'
%define PF19         '(-pf-19-)'
%define PF20         '(-pf-20-)'
%define PF21         '(-pf-21-)'
%define PF22         '(-pf-21-)'
%define PF23         '(-pf-23-)'
%define PF24         '(-pf-24-)'
%define PF25         '(-pf-25-)'
%define PF26         '(-pf-26-)'
%define PF27         '(-pf-27-)'
%define PF28         '(-pf-28-)'
%define PF29         '(-pf-29-)'
%define PF30         '(-pf-30-)'
%define PF31         '(-pf-31-)'
%define PF32         '(-pf-32-)'
```

```
/*                  G L O B A L   K E Y   N A M E S                  */
/*                          (continued)                              */

%define PickUp       '(-picup-)'
%define PutDown      '(-putdn-)'
%define RecallSize   '(-rclsz-)'
%define Recall       '(-recal-)'
%define Reset        '(-reset-)'
%define Right        '(-right-)'
%define Space        '(-space-)'
%define Tab          '(-tab-)'
%define CapsUnlock   '(-unloc-)'
%define Up           '(-up-)'


/*                  G L O B A L   V A R I A B L E S                  */
/*                  Accessible from all procedures.                  */


declare Key          char;            /* Generic character buffer   */
declare KeyRank      fixed;           /* Rank of the current char   */

declare CurrentRow   fixed;           /* Saved cursor coordinates   */
declare CurrentCol   fixed;           /* These are saved by the     */
declare EndRow       fixed;           /* CursorSave routine and     */
declare EndCol       fixed;           /* restored by CursorRestore. */

declare PickupBuffer char (80);       /* Secondary pickup buffer    */
declare PickupSize   fixed;           /* Current pickup text size   */

declare Line1        char (6);        /* Editor line number 1       */
declare Line2        char (6);        /* Editor line number 2       */

declare Address      char (6);        /* Current debugger address   */

declare Total        fixed;           /* Arithmetic accumulator     */

declare HexText      char (6);        /* Hex/decimal conversion     */
declare Counter      fixed;           /* work areas                 */
declare Digit        fixed;

declare Column       fixed;           /* Generic column save area   */
```

```
/*
   This routine searches right (Direction = 1) or left (-1) for a
   match (1) or a mismatch (0) of the requested character starting
   in the specified column, and returns the column number in which
   the character was found, or returns zero if no match.
*/
Search: procedure (Request, Match, Direction, Column) returns (fixed);
declare
        Request   char,   Match  fixed,
        Direction fixed,  Column fixed;

declare Equal     fixed;

   if rank (Request) > 127 then        /* If a control character,    */
      return (0);                      /* it will never be found.    */

   do forever;
      Equal = 1 and (Request = substr (Screen (cursorrow), Column, 1));

      if Match = Equal then            /* Result is the exclusive or */
         return (Column);              /* of the match flag and the  */
                                       /* comparison result.         */
      Column = Column + Direction;

      if Column < 1 ! Column > 80 then /* Search a single line only  */
         return (0);
      end;

end Search;


/*
    This routine searches left from the current cursor position
    to the last nonblank character.
*/
SearchLeft: procedure options ('1');

   Column = Search (' ', 0, -1, cursorcol);

   if Column < 80 then                 /* Set cursor only if the     */
      cursorcol = Column + 1;          /* character was found.       */

end SearchLeft;


/*
    This routine searches right from the current cursor position
    to the first nonblank character.
*/
SearchRight: procedure options ('3');

   Column = Search (' ', 0, 1, cursorcol);

   if Column > 1 then                  /* Set cursor only if the     */
      cursorcol = Column - 1;          /* character was found.       */

end SearchRight;
```

```
/*
    These routines save and restore the current cursor position.
    Note the use of the saved positions in the "Names" routine.
*/
CursorSave: procedure options (Anchor);

   CurrentRow = cursorrow;
   CurrentCol = cursorcol;

end CursorSave;


CursorRestore: procedure options (Home);

   call WaitForUnlock;
   cursorrow = CurrentRow;
   cursorcol = CurrentCol;

end CursorRestore;


/*
    These routines set the cursor to column 1, (in the Editor) column 48,
    and column 80, respectively.
*/
LeftMargin: procedure options ('4');

   cursorcol = 1;

end LeftMargin;


MidScreen: procedure options ('5');

   cursorcol = 48;

end MidScreen;


RightMargin: procedure options ('6');

   cursorcol = 80;

end RightMargin;
```

```
/*
    This routine waits for the requested character and echoes
    all other keystrokes to the workstation.  It also saves the
    ending cursor row and column.
*/
WaitFor: procedure (Request);
declare
        Request char;

    Key = getkey;                       /* Get a keystroke.          */
    do while (Key ↑= Request);          /* If it's not the one being */
       call playout (Key);              /* waited for, just echo it  */
       Key = getkey;                    /* back, and keep waiting.   */
       end;

    EndRow = cursorrow;                 /* Save cursor coordinates   */
    EndCol = cursorcol;                 /* for the calling routine.  */

end WaitFor;


/*
    This routine updates a running display once every 5 seconds.
*/
Wait: procedure options ('w');

   do forever;                          /* Loop until cancelled.     */
      call playout (Enter);             /* Strike ENTER.             */
      call delay (500);                 /* Wait 5 seconds (Z80 time) */
      end;                              /* and repeat until cancelled */

end Wait;
```

```
/*
   This routine provides a second, variable-length pickup buffer.
   The pickup string is from the cursor position at the moment of
   invocation, up to the cursor position when ENTER is struck.
*/
Remember: procedure options (Pickup);
declare Start fixed;

   Start = cursorcol;                  /* Pickup from here ...       */
   call WaitFor (Enter);               /* to current cursor column.  */

   PickupSize = EndCol - Start + 1;
   PickupBuffer = substr (Screen (EndRow), Start, PickupSize);

end Remember;


/*
   This routine plays out the contents of the pickup buffer
   created by the "Remember" routine above.
*/
PlayBack: procedure options (PutDown);

   call CursorSave;
   call playout (substr (PickupBuffer, 1, PickupSize));
   call CursorRestore;

end PlayBack;
```

```
/*
   These routines implement a simple column adding facility.
*/
Zero: procedure options ('0');
   Total = 0;                          /* Clear the accumulator      */
end Zero;


Count: procedure options ('+');
declare
        Check fixed;                   /* Check for invalid input    */

   do forever;
      Check = 0;
      Check = binary (substr (Screen (cursorrow), cursorcol, 5));

      if Check = 0 then                /* If invalid input, stop.    */
         stop;

      Total = Total + Check;           /* Accumulate the total       */
      cursorrow = cursorrow + 1;       /* Go to next row             */
      call waitforunlock;              /* Wait for screen to settle  */
      end;

end Count;


/*
   This routine implements a simple column numbering facility.
*/
Number: procedure options ('#');

   do forever;
      call CursorSave;                 /* Remember starting column   */

      Total = Total + 1;               /* Increment total            */
      if Total < 10 then               /* If only one digit in it,   */
         cursorcol = cursorcol + 1;    /* right-justify it.          */

      call playout (char (Total));     /* Play out the total         */

      cursorcol = CurrentCol;          /* Return to starting column  */
      cursorrow = cursorrow + 1;       /* Move down to next row.     */
      call waitforunlock;              /* Wait for screen to settle. */

      if cursorrow < CurrentRow then   /* If cursor has wrapped      */
         stop;                         /* around, stop.              */
      end;                             /* Else, do the next line.    */

end Number;
```

```
/*
    This routine switches the case of the character(s) at the
    current cursor position from lowercase to uppercase.
*/
UpCase: procedure options (Up);

   KeyRank = Search (' ', 0, 1, cursorcol);
   if KeyRank ↑= 0 then
      cursorcol = KeyRank;

   do while (cursorchar ↑= ' ');
      KeyRank = rank (cursorchar);

      if KeyRank >= rank ('a') and KeyRank <= rank ('z') then
         call playout (byte (KeyRank - rank (' ')));
      else
         cursorcol = cursorcol + 1;
      end;

end UpCase;


/*
    This routine switches the case of the character(s) at the
    current cursor position from uppercase to lowercase.
*/
DownCase: procedure options (Down);

   KeyRank = Search (' ', 0, 1, cursorcol);
   if KeyRank ↑= 0 then
      cursorcol = KeyRank;

   do while (cursorchar ↑= ' ');
      KeyRank = rank (cursorchar);

      if KeyRank >= rank ('A') and KeyRank <= rank ('Z') then
         call playout (byte (KeyRank + rank (' ')));
      else
         cursorcol = cursorcol + 1;
      end;

end DownCase;
```

```
/*
    This routine is called from all Editor glossaries to ensure
    that the glossaries only work in the Editor modes for which
    they are appropriate.  This depends on Editor version 6.9.10
    or higher which does not display the word 'mode' in the upper-
    right corner.
*/
StopIfNotEditor: procedure;

   call WaitForUnlock;

   if substr (Screen (1), 74, 7) ↑= 'Display' then
      stop;

end StopIfNotEditor;


/*
    This routine puts the Editor into Modify mode if it isn't already.
    Note that the current cursor position is saved for later use by
    the calling routine.
*/
ModifyMode: procedure;
                                    /* Save the cursor.  Other    */
   call CursorSave;                 /* routines depend on this.   */

   if substr (Screen (1), 75, 6) = 'Modify' then
      return;

   call StopIfNotEditor;            /* Must be in the Editor.     */

   call playout (PF9);              /* Invoke Modify mode.        */
   call WaitForUnlock;              /* Wait until WS settles down */

end ModifyMode;
```

```
/*
    This routine splits a single Editor line into two.  Position the
    cursor at the text to be split.  Press (-gl-)s.  The text to the
    right of the cursor is picked up and erased and a new line is
    inserted.  Position the cursor to the desired text position on the
    new line and press ENTER to drop the text onto the new line.
*/
Split: procedure options ('s');

   call ModifyMode;

   call playout (PickUp Erase Enter PF11);

   CurrentRow = CurrentRow + 1;
   call CursorRestore;

   call WaitFor (Enter);
   call playout (PutDown Enter PF1);

end Split;


/*
    This routine merges two Editor lines.  Position the cursor on the
    second line, at the text which is to be joined to the first line.
    Type (-gl-)t.  The text of the second line is joined to the first
    following the first nonblank.
*/
Join: procedure options ('t');

   call ModifyMode;

   call playout (Pickup Up);

   cursorcol = Search (' ', 0, -1, 79);

   call playout (Right PutDown Enter NewLine PF12 Enter);

end Join;
```

```
/*
    This routine performs a (somewhat) WP-style delete function.
    Position the cursor to the first character to be deleted,
    type (-gl-)(-delete-), and type the letter after the end of the
    text to be deleted.  If the letter is not found, try again.
*/
WPDelete: procedure options (Delete);

   Key = getkey;                          /* Search for boundary char.  */
   if 0 ↑= Search (Key, 1, 1, cursorcol) then
      do while (cursorchar ↑= Key);       /* Delete the required        */
         call playout (Delete);           /* characters.                */
         end;

end WPDelete;


/*
    This routine copies text lines downward until the cursor hits
    a nonblank character.  If the cursor character at invocation
    is blank, one key is accepted as the text to be copied.
*/
Propogate: procedure options ('p');

   if cursorchar = ' ' then do;           /* If the cursor character is */
      Key = getkey;                       /* a blank, get one from the  */
      call playout (Key);                 /* keyboard, and use that for */
      call playout (Left);                /* the output text.           */
      end;

   call playout (PickUp Down);            /* Pick up the string and go  */
                                          /* down one row.              */
   do while (cursorchar = ' ');
      call playout (PutDown Down);        /* Repeat until a non-blank   */
      end;                                /* is encountered.            */

   call CursorRestore;                    /* Restore the cursor.        */

end Propogate;


/*
    This routine duplicates the current Editor line.
*/
Duplicate: procedure options ('i');

   call StopIfNotEditor;
   cursorcol = 9;

   call playout (Pickup PF11 PutDown Enter PF1 PF9);

end Duplicate;
```

```
/*
    This routine swaps the current Editor line with the line below it.
    It requires a blank line somewhere on the screen for temporary use.
*/
Swap: procedure options (PF14);
declare Row fixed;

   call CursorSave;
   call playout (Pickup);

   do Row = 5 to 24;
      if substr (Screen (Row), cursorcol, 81-cursorcol) = ' ' then do;
         cursorrow = Row;
         call playout (PutDown);
         call CursorRestore;
         call playout (Down Pickup Up PutDown);
         cursorrow = Row;
         call playout (PickUp Erase);
         call CursorRestore;
         call playout (Down PutDown);
         call CursorRestore;
         return;
         end;
      end;

end Swap;


/*
    This routine checks if the parentheses on a line are balanced.
    It is NOT smart enough to check for them within quotes.
*/
Balance: procedure options ('(');
declare Count fixed,
        Lcol  fixed,
        Rcol  fixed;

   Count = 0;
   do Column = 9 to 80;
      if substr (Screen (cursorrow), Column, 1) = '(' then do;
         Lcol = Column;
         Count = Count + 1;            /* Count left parentheses     */
         end;
      if substr (Screen (cursorrow), Column, 1) = ')' then do;
         Rcol = Column;
         Count = Count - 1;            /* Count right parentheses    */
         end;
      end;
                                       /* Call the discrepancy to    */
   if Count > 0 then                   /* the user's attention.      */
      cursorcol = Rcol;                /* Point to the rightmost or  */
   if Count < 0 then                   /* leftmost parenthesis that  */
      cursorcol = Lcol;                /* is unbalanced.             */

end Balance;
```

```
/*
    This routine marks positions in the Editor for later use
    with the FIND, MOVE, COPY and DELETE commands.
*/
PickRange: procedure options ('8');

   Line1 = substr (Screen (cursorrow), 2, 6);
   call highlight (cursorrow, 2, 6);
   Line2 = Line1;

   call WaitFor (Enter);
   Line2 = substr (Screen (cursorrow), 2, 6);
   call highlight (cursorrow, 2, 6);

end PickRange;


/*
    This routine uses the LINE markers saved by the above routine
    to invoke the FIND, MOVE, COPY and DELETE commands.  It waits for
    the specific command key and then plays out the line numbers into
    the prompt fields.
*/
DoIt: procedure options ('7');
declare
        Offset fixed;

   call StopIfNotEditor;                 /* Must be in the Editor      */

   Key = getkey;                         /* Wait for pf key            */
   Offset = index (Delete PF9 PF11 PF12, Key);

   if Offset = 0 then
      stop;
   else
      call playout (substr (PF12 PF8 PF14 PF13, Offset, 1));

   call playout (Erase !! Line1);        /* Clear the prompt field     */
                                         /* Play out the first line #  */
   if Line1 ↑= Line2 then
      call playout (Tab Erase !! Line2);
                                         /* If the line numbers are    */
   else if                               /* the same, only do one.     */
      Key ↑= PF9 then                    /* If copy, clear 2nd field   */
      call playout (Tab Erase);

   call playout (Enter);                 /* Do the command.            */
   call WaitForUnlock;
                                         /* Handle "Invalid Range"     */
   if substr (Screen (3), 10, 1) = 'I' then
      call playout (Home Erase !! Line2 !!
                    Tab  Erase !! Line1 !! Enter);

end DoIt;
```

```
/*
    This procedure indents a line 3 characters without messing up
    the comments starting in column 40.
*/
Indent: procedure options (Right);

   call CursorSave;

   if substr (Screen (cursorrow), 48, 2) = '/*' then do;
      cursorcol = 45;
      call LineUp (-3);
      end;

   cursorcol = 9;
   call LineUp (3);
   call CursorRestore;

end Indent;

/*
    This procedure "exdents" a line 3 characters without messing up
    the comments starting in column 40.
*/
Exdent: procedure options (Left);

   call CursorSave;

   cursorcol = 9;
   call LineUp (-3);

   if substr (Screen (cursorrow), 45, 2) = '/*' then do;
      cursorcol = 45;
      call LineUp (3);
      end;

   call CursorRestore;

end Exdent;

/*
   This routine finds the correct level of indentation beneath
   the current source line.
*/
FindIndentation: procedure options (Tab);

   if substr (Screen (cursorrow), 9, 70) = ' ' then
      cursorrow = cursorrow - 1;

   call CursorSave;

   CurrentCol = 3 + Search (' ', 0, 1, 9);
   CurrentRow = CurrentRow + 1;

   call CursorRestore;

end FindIndentation;
```

```
/*
    This routine is called from "Align" to move the current text
    line a specified amount to the left (-) or to the right (+).
*/
LineUp: procedure (Count);              /* Line up a single text line */
declare
        Count fixed;

declare Loop fixed;

   if Count = 0 then                    /* If nothing to move,        */
      return;                           /* just return.               */

   if Count < 0 then do;                /* If count is negative, the  */
      Count = -Count;                   /* text is moving to the left */
      Key = Delete;                     /* so use the delete key.     */
      end;
   else                                 /* Else, must use insert.     */
      Key = Insert;

   do Loop = 1 to Count;                /* Play out the required      */
      call playout (Key);               /* number of inserts or       */
      end;                              /* deletes.                   */

end LineUp;


/*
    This routine aligns text lines left or right of their current
    position.  Set the cursor to the starting row to be aligned,
    type (-gl-)a, move the cursor left or right the number of
    columns to be added or deleted, and up or down the number of
    rows to be aligned, and press ENTER.
*/

Align: procedure options ('a');
declare Amount fixed,
        Row    fixed;

   call ModifyMode;                     /* Get into modify mode       */
                                        /* Wait for X,Y coordinate    */
   call WaitFor (Enter);                /* for number of rows to do,  */
   Amount = EndCol - CurrentCol;        /* and left or right shift.   */

   if Amount > 0 then                   /* Correct cursor position    */
      cursorcol = CurrentCol;           /* if aligning from left to   */
   else                                 /* right.                     */
      cursorcol = EndCol;

   do Row = CurrentRow to EndRow;       /* From the starting position */
      cursorrow = Row;                  /* to the ending row, do left */
      call LineUp (Amount);             /* or right shifts.           */
      end;

   call CursorRestore;                  /* Restore the cursor.        */

end Align;
```

```
/*
    This routine justifies multiple lines of text by adding blanks
    between words.  The scanning direction alternates between lines.
*/
Justify: procedure options ('j');
declare Row       fixed, Width  fixed,
        Start     fixed, Spaces fixed,
        Direction fixed;

   call CursorSave;                     /* Let operator mark the      */
   call WaitFor ('(-enter-)');          /* last row and column.       */

   Direction = 1;                       /* Start from left to right   */
   Width = EndCol - CurrentCol;         /* Calculate the text width   */

   do Row = CurrentRow to EndRow;       /* Justify requested rows     */
      Direction = -Direction;           /* in alternate directions.   */
      cursorrow = Row;                  /* Set to the current row.    */

      if Direction = 1 then             /* Choose the appropriate     */
         Start = CurrentCol;            /* limit column.              */
      else
         Start = EndCol;

      Spaces = Search (' ', 1, Direction, Start);
                                        /* Look for interword spaces  */
      if  Spaces ↑= 0                   /* and some text to justify.  */
      and substr (Screen (Row), CurrentCol, Width) ↑= ' ' then do;

         do while (substr (Screen (Row), CurrentCol, 1) = ' ');
            cursorcol = EndCol;
            call playout (Insert);
            cursorcol = CurrentCol;     /* Remove leading spaces.     */
            call playout (Delete);
            end;

         do while (substr (Screen (Row), EndCol, 1) = ' ');
            Spaces = Search (' ', 0, Direction, Spaces + Direction);
            Spaces = Search (' ', 1, Direction, Spaces);

            if Spaces <= CurrentCol     /* Skip intervening spaces    */
            or Spaces >= EndCol then    /* and check if another pass  */
               Spaces = Start;          /* is needed for this line.   */
            else do;                    /* Else, create an extra      */
               cursorcol = EndCol;      /* interword space.           */
               call playout (Delete);
               cursorcol = Spaces;
               call playout (Insert);
               end;
            end;
         end;
      end;

   call CursorRestore;                  /* Restore original cursor    */

end Justify;
```

```
/*
   This routine converts the hexadecimal number at the cursor
   into its decimal equivalent.
*/
HexToDecimal: procedure options ('d');
declare
        Last    fixed;
                                        /* Locate the hex number      */
   HexText = substr (Screen (cursorrow), cursorcol, 6);
   Last = verify (HexText, '0123456789ABCDEF');
   if Last = 0 then
      Last = 7;                         /* If all hex, then 6 digits  */

   Counter = 0;
   Last = Last - 1;                     /* Loop backwards thru digits */
   do Digit = 1 to Last;                /* converting them to base 10 */
      Counter = Counter * 16 + index ('0123456789ABCDEF',
                              substr (HexText, Digit, 1)) - 1;
      end;

   call playout (char (Counter));       /* Play out the result        */

end HexToDecimal;


/*
    This routine converts the decimal number at the cursor
    position to its hexadecimal equivalent.
*/
DecimalToHex: procedure options ('x');

                                        /* Fetch the binary number    */
   Counter = binary (substr (Screen (cursorrow), cursorcol, 6));

   Digit = 4;                           /* Max number of hex digits   */
   do while (Digit >= 1);
      substr (HexText, Digit, 1) = substr ('0123456789ABCDEF',
                                           1 + (15 and Counter), 1);
      Counter = Counter / 16;
      Digit = Digit - 1;                /* Convert to base 16         */
      end;

   call playout (HexText);              /* Play out the result        */

end DecimalToHex;
```

```
/*
    These routines perform window movement and sizing.
*/
WindowManager: procedure options (PF2);

Perform: procedure (Operations);
declare
       Operations char (8);

declare Direction  fixed;              /* Perform individual sizing  */
                                       /* operations using the arrow */
   do forever;                         /* keys.                      */
      Key = getkey;
      Direction = index (Up      Down      Left      Right
                         LookUp1 LookDown1 LookLeft1 LookRight1,
                         Key);
      if Direction = 0 then            /* If key is invalid, return  */
         leave;                        /* it to the caller.          */

      call playout (substr (Operations, Direction, 1));
      end;
end Perform;

                                       /* Start of window manager    */
   Key = getkey;
   do while (1);
      if Key = InsertMode then do;                 /* Full size window   */
         call playout (Full);  stop; end;
      else
      if Key = Recall then do;                     /* Recall prior size  */
         call playout (RecallSize); stop; end;
      else
      if Key = Erase then do;                      /* Delete the window  */
         call playout (Invisible); stop; end;
      else
      if Key = Insert then                         /* Make window grow   */
         call Perform (AddUp1 AddDown1 AddLeft1 AddRight1
                       AddUp3 AddDown5 AddLeft10 AddRight10);
      else
      if Key = Delete then                         /* Make window shrink */
         call Perform (DelUp1 DelDown1 DelLeft1 DelRight1
                       DelUp3 DelDown3 DelLeft10 DelRight10);
      else
      if Key = PF12 then                           /* Make window move   */
         call Perform (MovUp1 MovDown1 MovLeft1 MovRight1
                       MovUp3 MovDown3 MovLeft10 MovRight10);
      else
         Key = getkey;                 /* Invalid key.  Get another. */
      end;

end WindowManager;
```

```
/*  The following procedure automatically logs on all workstations  */
/*  whenever the personality is loaded.  The procedure first        */
/*  checks to make sure that the window is displaying the logon     */
/*  screen.  It then issues a HELP to ensure that the cursor is     */
/*  located at the User ID field and enters MGL on window 1 and     */
/*  ML concatenated with the window number on the other windows.    */
/*  The procedure then enters the password and logs on.             */

auto_logon: procedure options (main);
       if substr(screen(1), 41, 5) = "Logon" then do;
           call playout("(-help!-)");
           if window = 1 then do;
               call playout("MGL");
           end; else do;
               call playout("ML"!!char(window));
           end;
           call playout("(-tab-)PASSWORD(-enter-)");
       end;
   end auto_logon;

   /*  The following procedure logs off the current window or all the  */
   /*  windows when invoked globally.  It first exits any program by   */
   /*  issuing a series of PF16 keys until the Command Processor is    */
   /*  displayed (identified by the value 'Workstation' on Row 4).     */
   /*  The procedure then issues a PF16 and an Enter, logging off the  */
   /*  window.                                                         */

   auto_logoff: procedure options ('k');
       do while (substr(screen(4), 2, 11) ↑= 'Workstation');
           call playout ("(-pf-16-)");
           call waitforunlock;
       end;
       call playout ("(-pf-16-)(-enter-)");
   end auto_logoff;
```

APPENDIX F
GLOSSARY LANGUAGE KEYWORDS

The following words are reserved for use by the Glossary compiler and should not be used as identifiers in your program.

AND
CALL
DCL
DECLARE
DO
ELSE
END
IF
LEAVE
NOT
OR
RETURN
STOP
THEN

APPENDIX G
AID CHARACTER REPRESENTATIONS

Table G-1. AID Character Representations

AID Function	Hexadecimal Value	ASCII Equivalent
ENTER key	40	@
PF1	41	A
PF2	42	B
PF3	43	C
PF4	44	D
PF5	45	E
PF6	46	F
PF7	47	G
PF8	48	H
PF9	49	I
PF10	4A	J
PF11	4B	K
PF12	4C	L
PF13	4D	M
PF14	4E	N
PF15	4F	O
PF16	50	P
PF17	61	a
PF18	62	b
PF19	63	c
PF20	64	d
PF21	65	e
PF22	66	f
PF23	67	g
PF24	68	h
PF25	69	i
PF26	6A	j
PF27	6B	k
PF28	6C	l
PF29	6D	m
PF30	6E	n
PF31	6F	o
PF32	70	p

INDEX

S

T

U

V

W

WANG

Customer Comment Form

Publication Number 800-1149-01

Title VS MULTI-STATION USER'S REFERENCE

Help Us Help You . . .

We've worked hard to make this document useful, readable, and technically accurate. Did we succeed? Only you can tell us! Your comments and suggestions will help us improve our technical communications. Please take a few minutes to let us know how you feel.

How did you receive this publication?

- ☐ Support or Sales Rep
- ☐ Wang Supplies Division
- ☐ From another user
- ☐ Enclosed with equipment
- ☐ Don't know
- ☐ Other _____

How did you use this Publication?

- ☐ Introduction to the subject
- ☐ Classroom text (student)
- ☐ Classroom text (teacher)
- ☐ Self-study text
- ☐ Aid to advanced knowledge
- ☐ Guide to operating instructions
- ☐ As a reference manual
- ☐ Other _____

Please rate the quality of this publication in each of the following areas.

	EXCELLENT	GOOD	FAIR	POOR	VERY POOR
Technical Accuracy — Does the system work the way the manual says it does?	☐	☐	☐	☐	☐
Readability — Is the manual easy to read and understand?	☐	☐	☐	☐	☐
Clarity — Are the instructions easy to follow?	☐	☐	☐	☐	☐
Examples — Were they helpful, realistic? Were there enough of them?	☐	☐	☐	☐	☐
Organization — Was it logical? Was it easy to find what you needed to know?	☐	☐	☐	☐	☐
Illustrations — Were they clear and useful?	☐	☐	☐	☐	☐
Physical Attractiveness — What did you think of the printing, binding, etc?	☐	☐	☐	☐	☐

Were there any terms or concepts that were not defined properly? ☐ Y ☐ N If so, what were they? _____

After reading this document do you feel that you will be able to operate the equipment/software? ☐ Yes ☐ No ☐ Yes, with practice

What errors or faults did you find in the manual? (Please include page numbers) _____

Do you have any other comments or suggestions? _____

Name _____ **Street** _____

Title _____ **City** _____

Dept/Mail Stop _____ **State/Country** _____

Company _____ **Zip Code** _____ **Telephone** _____

Thank you for your help.

Printed in U.S.A. 14-3140 7-83-5C

WANG

Fold

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD

FIRST CLASS PERMIT NO. 16 LOWELL, MA

POSTAGE WILL BE PAID BY ADDRESSEE

WANG LABORATORIES, INC.
TECHNICAL PUBLICATIONS
ONE INDUSTRIAL AVENUE
LOWELL, MASSACHUSETTS 01851

Fold

Cut along dotted line.

WANG

ONE INDUSTRIAL AVENUE
LOWELL, MASSACHUSETTS 01851
TEL. (617) 459-5000
TWX 710-343-6769, TELEX 94-7421

Printed in U.S.A.
800-1149-01
12-83-7M