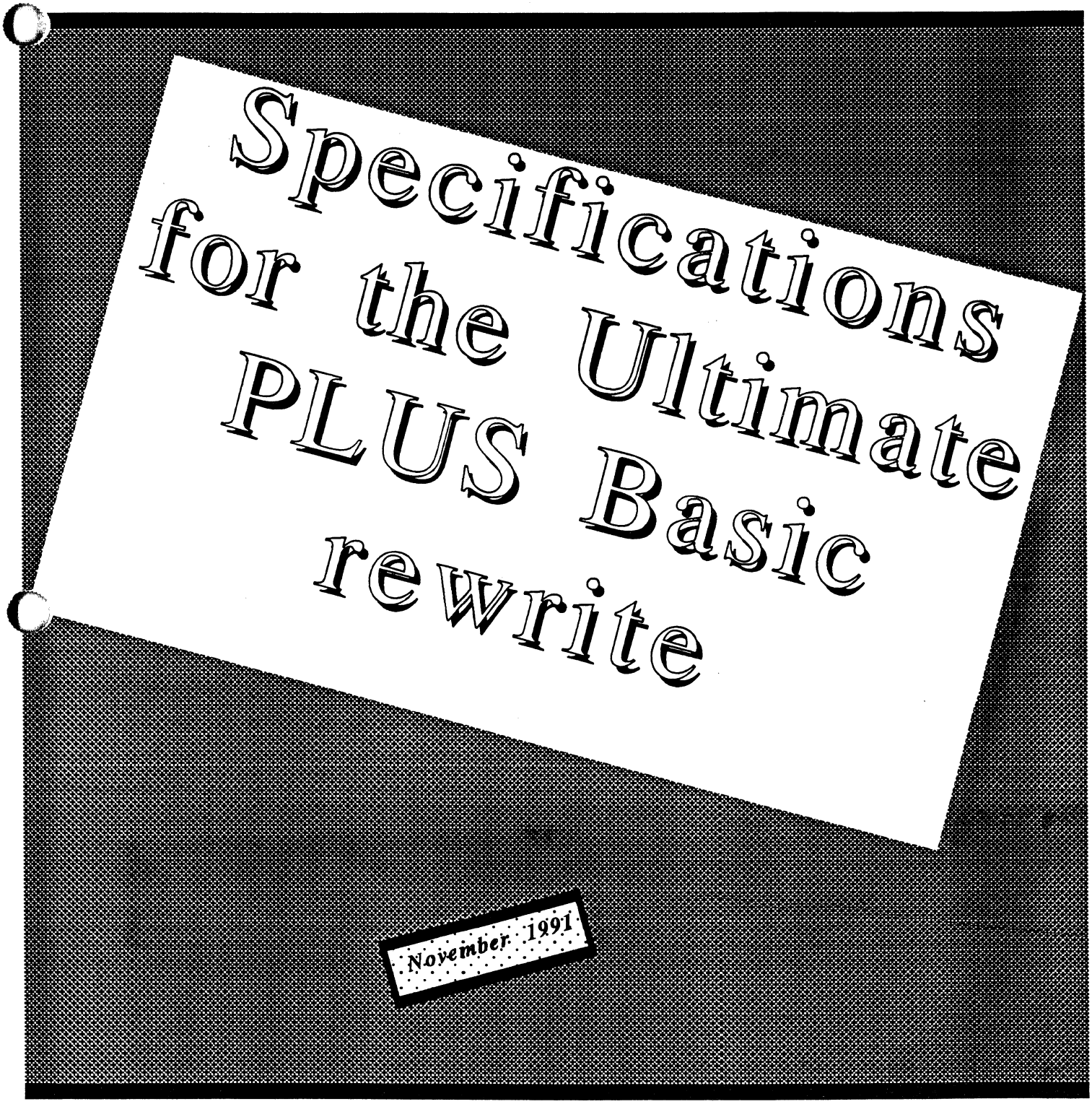

Specifications
for the Ultimate
PLUS Basic
rewrite
November 1991

The programmers that contributed to this document are:

Johnson Terry
Mohammadi Farzin
Necker Lee
Riscalla Daniel
Truyen Frank

This document was prepared using Wordperfect on the HP 835, Wordperfect on the IBM and Wordperfect on the Macintosh.

The first draft for this document was completed on December 2, 1991

The Times Roman font was used and a Macintosh Laserwriter to print it.

Table of contents:

I. Introduction:

The purpose of this project is to achieve a level of performance of the Basic runtime package that equals or surpasses the speed of our major competitors, as measured through benchmark programs such as the X-rating and Probench.

This has to be attained with no compatibility loss (or very little if unavoidable). This document lists all cases where the proposed implementation does not follow this constraint.

The project is to be viewed as the first but very important step towards the re-architecture of the Ultimate PLUS operating environment. This re-architecture in general will migrate the Ultimate PLUS environment to an architecture that is more seamless with the UNIX operating system. Re-writes of other parts of the system will definitely follow (the file system, recall etc...) as they make sense to occur as well as the introduction of new features like SQL or relational database access. From this project on, C (or C++) will be the programming language of choice. Virtual assembly programming will not be used unless absolutely necessary.

Therefore, this project being the first of a few, it has to be carefully designed to not introduce any architectural limitations that would stand in the way of some of the following projects (file system rewrite, SQL, ...).

The ultimate goal of the Basic rewrite is to provide a new Basic runtime that at least equals our competitors in performance. The C code developed will replace all current Virtual code related to the Basic runtime. Links with other software modules as well as time constraints may limit the scope of the first phase, resulting in some of the less frequently used or very complex Basic operations to remain in Virtual.

In addition to the runtime re-write, an optimizer will be developed in this project. That optimizer will use conventional compiler optimization techniques (similar to the ones used today in most C compilers) to optimize the user's Basic code. This should help us meet our performance goals more easily.

Unless opposite advice from our Marketing people, the old runtime environment will be available in parallel with the new Basic, providing a backup for the user in case of emergency. It is understood that at some point in time however, the old code is to be completely removed from the system.

In order to limit the development cycle, and also to address the issue of customers with no source code to their application, the basis for the performance enhancement code is the output of the current compiler. Options will be provided with the current compiler to produce a new object code format and to enable the optimizer.

II. Current performance analysis:

When considering rewriting areas of Ultimate PLUS for performance gains, the question that comes to mind is *which area, if rewritten, would provide the most gains in performance?* To better answer that question, various part of Ultimate PLUS were compared to Ult/ix. The Basic module appears to be the one in which Ultimate PLUS suffers the most. Ultimate PLUS seems to be about 2.5 times slower than Ult/ix when compared on the same machine. Recall on the other hand is comparable between the two products. Considering that most dealers have a significant investment in Basic programming (most dealer applications are written in Basic), it was clear that enhancing the performance of Basic was the right thing to do.

The performance of Ultimate Plus' file system appears to be comparable to the performance of Ult/ix's file system. In some test cases Ultimate PLUS is faster than Ult/ix and in some other cases Ult/ix is faster than Ultimate PLUS. We will try to identify small enhancements that will make us over all faster than Ult/ix for file accesses.

III. New Basic overview:

This section describes the architecture of the New Basic at a very high level. From here on, *Old Basic* and *Old object code* will refer to the Basic and object code that are currently released with Ultimate PLUS. *New Basic* and *New Object Code* will refer to the Basic and object code that will result

from this rewrite.

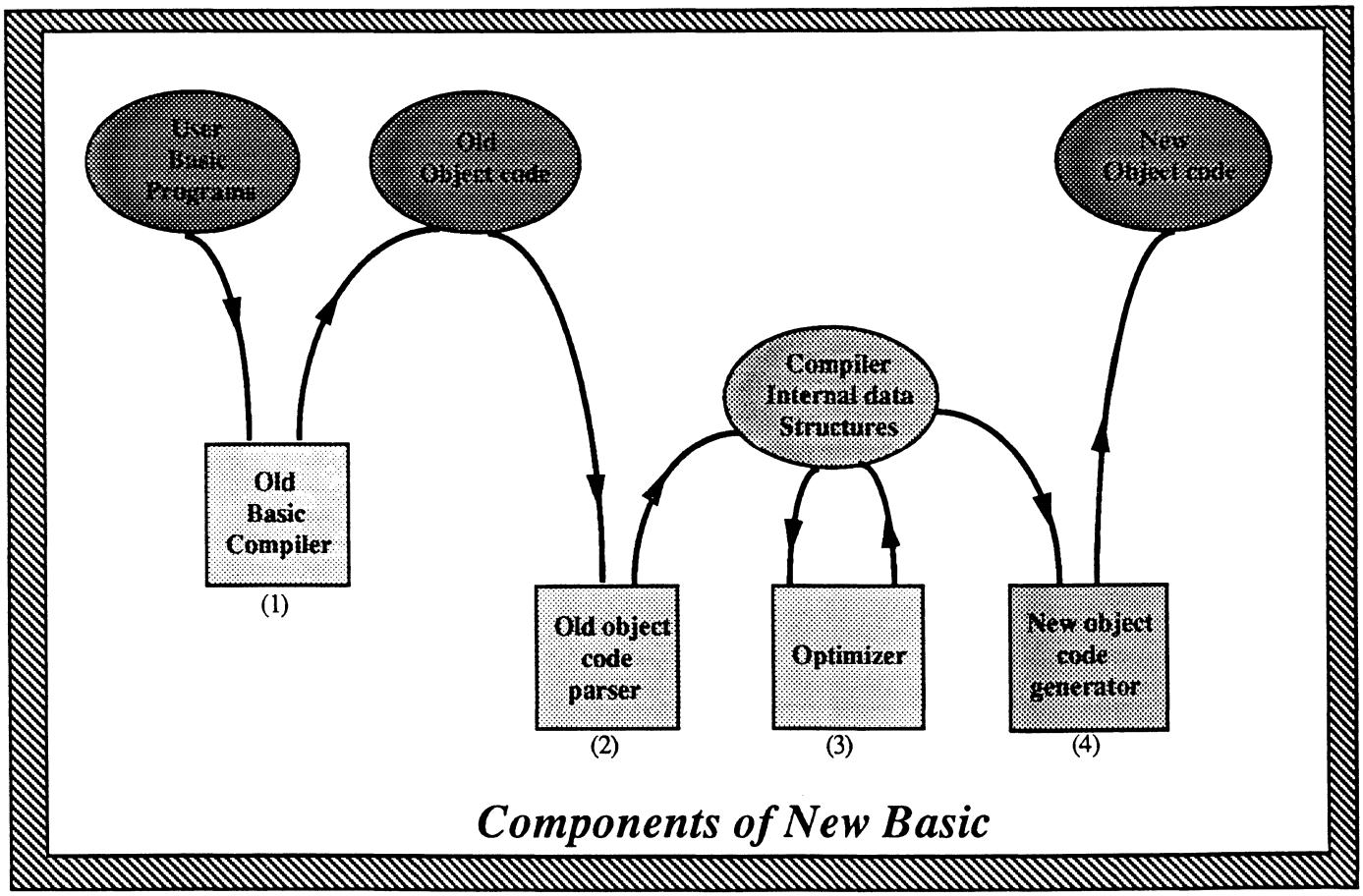
The modules that make the *New Basic* are as follows:

(1) The old Basic compiler

(2) A new parser to parse the old object code

(3) An optimizer

(4) New object generator



Components of New Basic

As this list of modules seems to imply, the old compiler remains as part of

the current system. Here follows a description of every module:

The old Basic compiler:

This module is practically unchanged from the current Ultimate PLUS release. It is the good old Basic compiler that takes the user's Basic sources and generates old object code. Some minor changes might be made to support additional options to better control calls and the behavior of subsequent modules. The reason the compiler was kept is to simplify the scope of the project as we do not have to worry about Basic parsing issues we rather can concentrate on the runtime itself. There is not doubt that at some point in the future, that old compiler will be taken out and a new Basic parser will be implemented.

Parser of old object code:

After the user programs have been compiled into object, the new compiler will go through the old object recognizing opcodes and their arguments and building some data structures in memory that form a representation of the program. These data structures are appropriate for optimization algorithms (see detailed section about optimizer). As the object is parsed, the parser will assemble some significant information about the program being compiled and store them in the mentioned data structures. That information will be useful later on for a better code generation.

Optimizer:

The optimizer is then optionally called. It is not clear yet if the default will be to call it or vice-versa, that can be determined later on. When the optimizer is called, it will apply some traditional compiler optimization techniques against the built data structures. It will therefore massage the data structures resulting in a smaller, more efficient, functionally equivalent set of data structures that represent the original user program. This optimizer can be very simple just including some obvious and easy techniques or can be extremely complex. Its complexity will depend on the time we can afford spending on it without compromising the runtime rewrite.

New object code generator:

This module is invoked last whether the optimizer was called or not. It goes against the built data structures and produces object code that is the new format. This module will take advantage of information gathered throughout the compilation to try to generate smarter object code.

Keeping the new compiler modularized into the modules just described should help building a more stable product that is more easily modified. For example, the compiler should be fully functional before any of the optimizer is coded as the *old object code parser* produces data that is input to the *new object code generator.*

IV. Internal architecture:

This section goes into a more detailed discussion of the internal architecture of the Basic rewrite. It is sub-divided into three sections.

The first one addresses various architectural concepts and why one solution was chosen instead of another. Most of the topics addressed have an **appendix** that discusses the subject in a much more detailed fashion.

The second section will talk about the actual implementation architecture getting into things like data structures, macros and templates. Whereas the first section talked about concepts, this one talks about implementation; they both relate though to the runtime.

The optimizer being a little special beast, it deserved a section all by itself. That third section describes the optimizer architecture, the optimization techniques it would use and how they would be implemented. This section is merely a summary for numerous compiler material that was gathered and analyzed.

A. Runtime Architecture:

1. Interpreted object versus direct C code:

When developing a *language processor*, there is always the issue of whether the product should be an interpreter or a

compiler. Both have advantages and disadvantages. Whereas an interpreter is more flexible and more powerful because better in control of the runtime environment, a compiler compensates by producing a faster model where the interpretation of the instructions was done at compile time.

If implemented right, one can modify an interpreter to a compiler and vice-versa with some limited efforts. That is to say that conceptually they both are not so much apart (they really both end up being interpreters one way or another, the question is at what level is the interpretation being done).

We are choosing to remain an interpreter instead of translating into C code. Translating into C code would most definitely produce faster code but the difference should be under 10 percent. The disadvantage of translating into C code is that our dependency on the C compiler of the machine would go up. It is important to notice for example, that today, for the Ultimate PLUS implementation, not all compiler options are valid. Some options lead the C compiler (and this is true for all machines we run on) to produce code that would fail. Controlling the way the C compiler is invoked on a customer machine would therefore become an additional responsibility (it is true though that sooner or later we would have to deal with that issue as we would allow customers to intermix C code with their applications).

Moreover, our compile times would become significantly dependant on the speed of the C compiler and that is not something to take lightly. We have seen under different implementation the C compile time represent over ninety percent of the **Basic to object code** compile process.

One last issue is the issue of providing for dynamic linking (indirect subroutine calls). That technique is not mature enough today to the point where we can provide an easily portable solution across all the platforms we support. UNIX System V release 4 does make dynamic linking a standard but this is not true for release 3.

I guess the conclusion is that a compilation in to C although

possible would end up presenting more problems than the performance gain is worth. Moreover some tests we have done seem to indicate that the performance gain between the two schemes might not end up be higher than a few points.

2. Non-stack architecture:

The Unix systems that currently run Ult/PLUS are RISC based platforms that use a fast stack architecture at the hardware level, geared towards running C programs. Our goal is to take the most advantage of this, without adding a software stack on top of it.

Also from a performance angle, using the machine stack shows a clear advantage over using a software stack.

Some tests we have run indicate that the performance loss because of a software stack can be as much as four times slower. Implementing a model without a software stack will definitely lead to a more complex structure but the performance gains seem to make it worthwhile.

Refer to Appendix M for a more in depth discussion.

3. Descriptor and string space management:

The descriptor space is implemented as a piece of memory that comes out of the UNIX/C process heap space. When a program is started, a chunk of memory is allocated to fit the necessary number of descriptors. A descriptor is defined as a C structure that can hold different types of data representation. These structures allow for a natural representation of native C data types, such as floating point numbers, and are generated by the C compiler with the proper alignment thereby resolving the bus exception issue on the HP series.

String space is also acquired from the heap of the UNIX process.

The heap itself is manipulated either through the use of standard C library calls (malloc()/free()/realloc) or through a

super set of these functions, under Ultimate's control.

Please refer to Appendix K for more details, and for some side effects of this approach in regards to the debugger.

4. Math operations, numeric storage & representation:

In the new Basic runtime, numbers are represented as doubles. Those are floating poin numbers with at least 32 (typicallyc 64 bits). Each program can run in one of two flavors, Ultimate plus flavor, or Ult/ix flavor. The flavor is determined at basic compile time. This means a compilation is required to switch the flavor. The flavor can be controlled via an environment variable, basic compile option, or a system wide option. In both flavors the numbers are represented as doubles. However, for the Ultimate flavor, each time a numeric variable is updated, the variable is adjusted. The adjustment is based on the scaling factor of the basic program and produces a considerable overhead.

The basic math operations such as addition, subtraction, multiplication and division are accomplished by using the C operators '+', '-', '*' and '/' on doubles. If string math is necessary then the library routines to be coded by us will be used to do the job.

Other math operations such as SIN, COS, LN, PWR, EXP, SQRT etc. are accomplished by using the C library routines sin(), cos(), log(), pow(), exp(), sqrt() etc.

Please refer to appendix G for a more detailed discussion of the math issue.

5. Interface between the new Basic environment and the virtual system:

A uniform mechanism is going to be developed to interface between the new Basic runtime and the virtual system.

Data structures are being represented differently in the new

Basic runtime (no or very rare use of PCB data elements, no use of linked frames or of workspaces etc...). Therefore, a documented (and coded if any code is necessary) API (application programmer interface) is necessary to go from the new Basic environment to virtual code or vice-versa. That interface has to promote readability and ease of maintenance. For example, if a piece of virtual code that is called from the new Basic environment, is being modified, the virtual coder needs to know that this is the case and the virtual coder needs to understand how the interface is defined to make sure that his changes preserve that interface.

The biggest beneficiary of that API will be the File System as the new runtime will call virtual code for database access and the Ultimate PLUS modules that call Basic but will not be rewritten for the time being (like Recall and Update).

More detailed description of the interface between the new Basic and the virtual system can be found in appendix R.

6. Basic Debugger:

The current debugger functionalities and commands will be preserved in the new Basic debugger. This document is written in complete accordance with the current debugger requirements. However, to correct the cryptic nature of the current debugger, the new Basic debugger has two command modes of operation. They are the Ultimate command mode and the verbose command mode. In the Ultimate command mode, the debugger takes traditional debugger commands. In the verbose command mode, a new English-like syntax is used to issue the same commands and all the new commands. Users should be encouraged to switch to the new verbose command mode. Eventually, we may take away the Ultimate command mode.

The debugger can perform symbolic source debugging for non-optimized programs. For optimized programs, the debugger can only provide effectively a subroutine or an event level debugging because the one-to-one relationship between object and source code does not exist anymore. The capability to

point out which line caused a runtime error is to be implemented for both types of programs. In addition, the debugger also provides an opcode level debugging of all programs for use by people who are so inclined (for internal use mostly).

Currently, the Basic debugger can set a breakpoint on a line number, on logical conditions and on CALL/RETURN instructions. To further facilitate debugging of a Basic program, there is now conditional break point on a source line, break point on a variable when it changes, break point inside a subroutine when it is called. The user may switch the debugging terminal to another port on the system which will help debugging screen type application. The entire debug session may be logged to a Unix file. Furthermore, there is an indexed help facility for the verbose set of commands.

Our design takes all of the desired new features into consideration. Those new features though, will only be implemented as time permits.

7. File system access changes:

a. Current performance analysis: File I/O in Basic is roughly the same speed on Ultimate Plus as on Ult/ix, so we are not proposing major changes to the Ultimate file system now.

In general, Ult Plus is fast in these cases:
- Random access
- Large items
- Repeated access on other than the RS/6000
- The Basic SELECT statement
- Very large files

Ult/ix is fast in these cases:
- Any Recall statement
- Sequential access

Just where the dividing lines between the above categories lie is unclear. Some tests show that Ult Plus becomes fast when item sizes approach 100 bytes, while other tests show the

dividing line to be around 500 bytes.
Which operations are fast and which are slow sometimes depends on the hardware platform. For example, Ult Plus uses shared memory as a cache of file data, so repeated accesses to a file do not require disk reads, which would generally give ULT Plus a speed advantage over Ult/ix. This is true on HP machines. However, the RS/6000 can use all of memory as a cache of file data, so repeated accesses to a file are improved in Ult/ix, making the speeds of the systems comparable.

b. Proposed implementation: Webb has just changed item retrieval to only use read locks when there is a conflict with another process updating the group that is being read. Retix now returns a byte value representing the condition of the group before the item was retrieved and a pointer to the current byte value within the group. The caller of RETIX may choose to copy the item from the file, then compare the values, and if they are different, redo the item retrieval with read locks. Basic and Recall now do this instead of always setting read locks. There were some difficult problems (Webb cannot remember exactly what they were) with putting all of the responsibility for data integrity inside the file system.

The elimination of setting read locks most of the time has resulted in about a 20% improvement in file accessing speed, so this feature will be included in Ult Plus.

The format of the interface to New Basic is still under design because the present code requires a moderate amount of interaction between Basic and the file system. For now, the interaction will be minimized by making the Basic virtual code that interfaces to the file system into a routine that can be called from New Basic. It is possible that the interaction between New Basic and the file system will be increased as the design of the interface evolves.

We will therefore:

- Implement the new interface
- Tune the new interface
- Tune the file system, itself

Tuning the file system will be done as time permits and as required to improve performance. This step consists of making changes to file structure, such as storing the item size in binary instead of ASCII.

B. New Basic Runtime implementation:

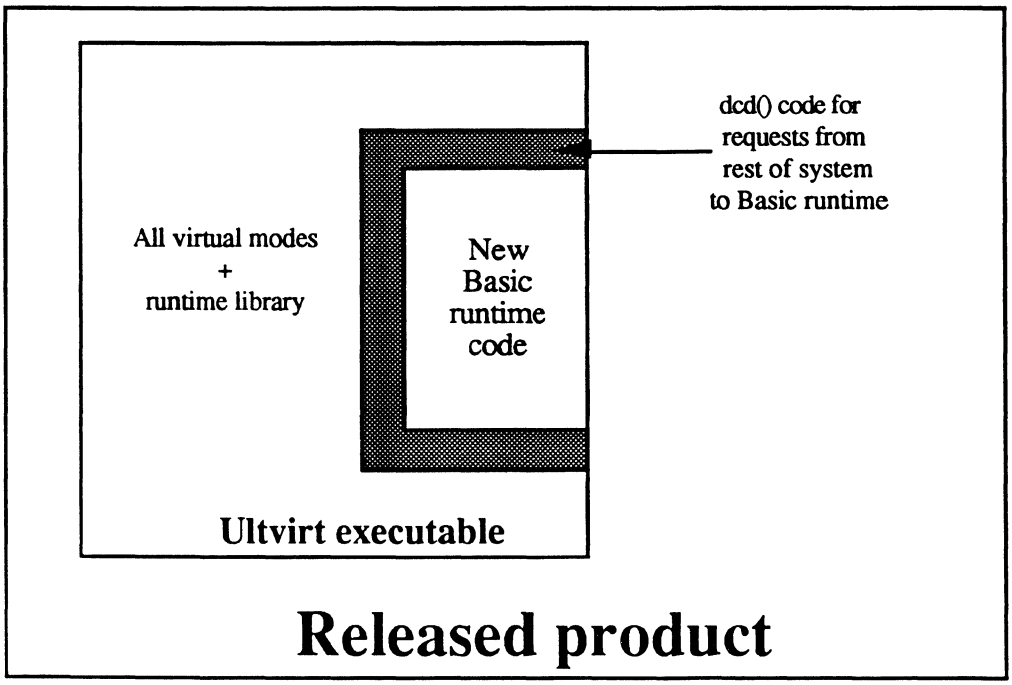
This section is more down to earth than the previous ones. It will describe the environment necessary for developing code for the new runtime. Data structures, templates and macros are among the topics addressed in this section. Do not underestimate the importance of the material in this section as significant R&D resources have been spent to come up with the described environment. this was done to ensure a stable and consistent environment that programmers will work in. This will pay off with faster code development in the next few months, more readable code and more consistent coding practices.

1. Calling the New Basic runtime:

Any programmer developing opcodes for the new Basic, will go through a particular development cycle:

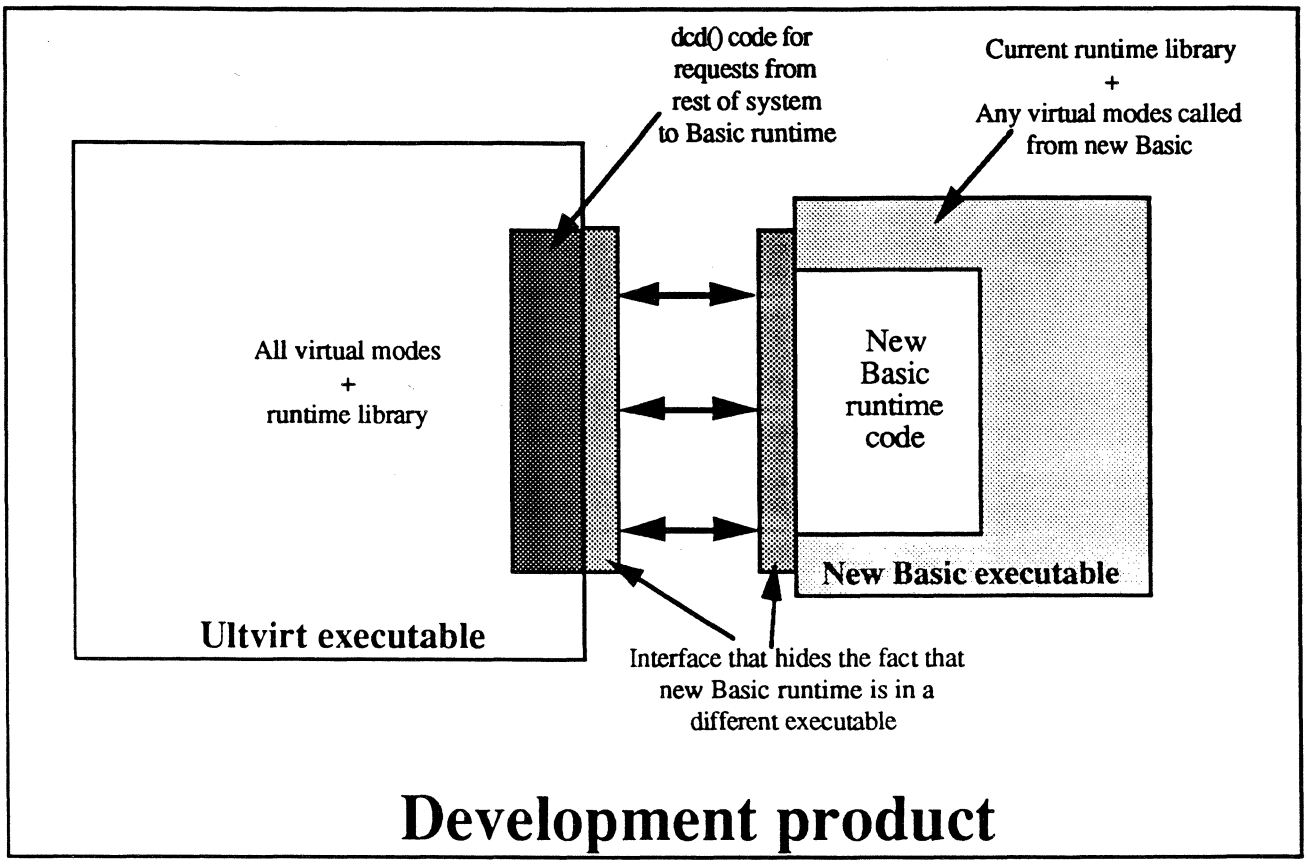
develop code - compile opcode - relink it with ultvirt - test - make changes in code - compile opcode - relink ...

Being many of us (us = opcode developers), there will typically be more than one person going through that cycle on the some machine. The problem is that the *relink it with ultvirt* portion of the cycle is time and machine resources consuming. Therefore we want to develop an environment that will make that procedure less painful for everybody.



The way this is going to be done is by splitting the new opcodes (just for the purposes of development) into a separate executable that is a lot easier to link (a few seconds compared to a many minutes!).

The previous picture highlights how the code will be architectured for the released product. The following picture highlights how it will be architectured during the development cycle. The switch from one to the other is completely transparent to the new Basic runtime code.



As you can see in the picture, for development purposes, the new runtime will be in a separate executable that contains all of the runtime library plus whatever virtual modes are directly referenced (those would certainly be less than the 700 or so linked today in **ultvirt**). An interface will be developed between the **dcd()** code in **ultvirt** and that executable making it completely transparent to the new Basic runtime that it is running in a separate executable.

2. Descriptor data structure:

The following data structure is the preliminary layout for the Basic variable and constant data elements. It can be found in the 'descriptor.h' file. The data structure has basically two main pieces to it:

- The first one is the fixed portion. It contains the fields that are always applicable no matter what type the descriptor is. Those fields are the descriptor type, a string pointer and the length of that string pointed to.

Basic Variable Data Structure

```
struct{
        struct{                         Part of the structure that is NOT overlaid.
                union{                  Structure to hold the descriptor type.

                        unsigned short  class_type;     Both class and type fields
                        struct{
                                unsigned char   class;  upper byte of flags; class of data
                                unsigned char   type;   lower byte of flags; the type
                        } byte;
                } flags;
                struct{                 Structure to hold any type of string data information.
                        char                    *ptr;   pointer to character string.
                        int                     len;    length of the string.

                } string;
        } fixed;
        union{                          Part of the structure that gets overlaid.
                struct{                         structure to hold a numeric value.
                        double                  float_val;      floating point numeric
                                                                value.
                } numeric;
                struct{                         Structure for a select list variable.

                        int                     count;          count of elements in select
                                                                list.
                        char*                   current_ptr;    pointer to last element
                                                                extracted.

                } select;
                struct{                         Structure for a file variable.
                        int                     fcb1;           open file FCB1 value;
                        int                     fcb2;           open file FCB2 value;
                } file;
                struct{                         Structure for conversion constants.
                        unsigned int            flags;          conversion flags
                        union{
                                struct{                         numeric format mask
                                                                structure.
                                        short   decimal;        decimal digits to output.
                                        short   scale;          scaling factor;
```

```
                        } format;
                        struct{                         Structure for a date format
                                                        mask.
                                short   group;          group extract count.
                                char    year;           number of year digits to
                                                        print.
                                char    group_sep;      group extract separator.
                                char    output_sep;     date output separator.
                        } date;
                        struct{                         Structure for group extract.
                                short   skip;           number of fields to skip.
                                short   extract;        number of fields to extract.
                                char    group_sep;      group extract separator.
                        } group;
                } type;
        } conv;
} overlay;
} desc_struct;
```

- The second portion is the overlay part of the structure. That part will be used differently depending on the type of variable used. For most variables, it will contain the floating point field for the numeric value. For file variables, select variables, string format constants (and some more special variables), this portion of the data structure is overlaid with a structure that is more appropriate for that type of descriptor.

The whole data structure and especially its sub-structures and unions will be hidden from the programmer through the use of various macros that are described later.

3. Basic variables and constants:

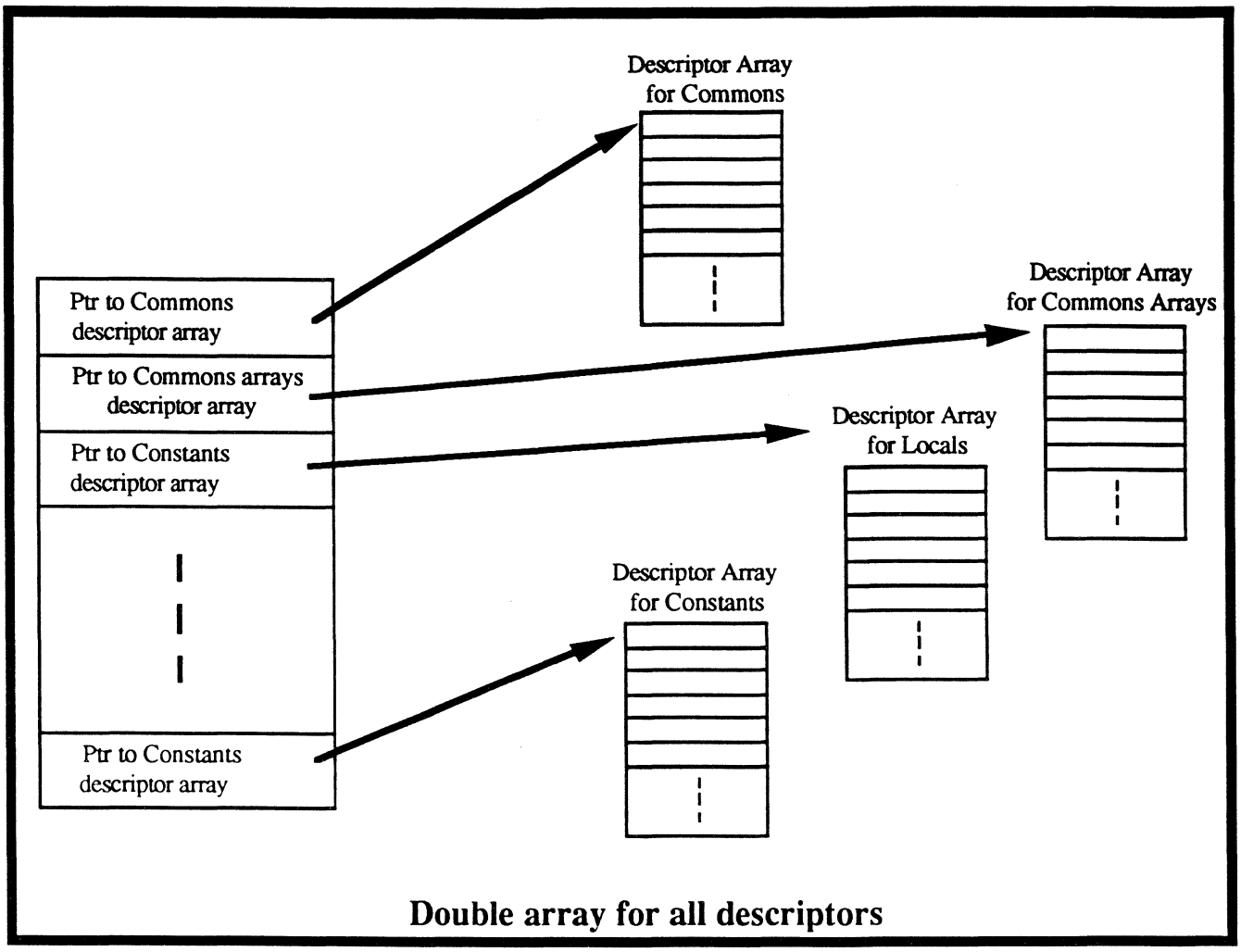
There are various types of descriptors that we have to deal with. They are various not with respect to their type in Basic (select variable etc...) but rather with respect to their nature like local variables, constants, commons, arrays etc... In trying to design an appropriate data structure for these different types, one major goal was to again promote readability of the code and ease of programming opcode to accelerate the development cycle.

It would obviously be an additional headache if for every opcode coded one had to worry where does the variable come from. If it is a constant, I reference it this way; if it is a common I reference it that way etc... We therefore came up with a data structure that makes it completely transparent to the opcodes where the variable comes from. Meaning, the opcode routines will not have to explicitly know that this variable is a common or that one is a constant unless specifically required by the nature of the opcode (some opcodes for example will require an array as an argument).

The data structure in question is a double array. The first array is an array of pointers to descriptor tables. A descriptor is the data structure that was described in the previous section. Every descriptor table represents one type of descriptors. For example, the first one is for commons; the second one is for arrays in commons (if we decide to split between the two); the third one is for local variables; the last one is for constants etc... Then all variables are represented in the same manner. Each one is uniquely identified by a tuple (a,b): a is the index of the descriptor table it belongs to in the first array and b is the index in the descriptor table itself.

For example, a local variable A would be represented by the tuple (2, 4) where two indicates that A is a local variable because all local variables belong to the third table (indexed as 2 because we start at 0) and A is the fifth local variable in the descriptor table for local variables.

The *new code generator* module of the compiler will be responsible for producing correct references in the object code to the variables. The compiler will therefore evaluate the characteristics of this double array as it has parsed the Basic program old object code. It will determine how many elements to each descriptor table, how many descriptor tables etc... It will also produce at the end of the *new object code* a copy of the constant descriptor table so it can be loaded directly.



Double array for all descriptors

One significant advantage of this data structure is how **call** becomes simplified with respect to handling local variables. To give the program a new set of local variables, all that has to happen is save the address of the caller's local variables descriptor table and make the corresponding entry in the first array point to a newly allocated callee local variables descriptor table.

A different approach could have been used for representing the descriptors with a single array. The problem would have been that arrays and commons would have probably required some handling and special cases in the opcodes. The advantage of such an approach is that only one indirection is necessary to

get to the descriptor instead of the two in the other scheme. We have done some timings and determined that the performance difference between the two is **null** as the additional de-referencing has no weight on the overall time spent in an opcode.

4. Opcode templates:

As the largest portion of the Basic rewrite will consist of rewriting hundreds of opcodes, we have identified the need to develop a template for those opcodes. That template would provide the programmers with a starting point when they want to write the code for a new opcode shortening the development cycle.

This section presents the template for an opcode. We have actually coded a real opcode and done numerous passes on it to determine what is the best way for the code to look, best macros etc...

a. An opcode template:

Here follows the template for a typical opcode. The various concepts and macros used throughout this listing are explained in the following few sections.

```
/*****************************************************************************/
/*                                   OPCODE.c                                */
/*                              Proprietary Information                      */
/*                          Copyright (C) Ultimate Corporation               */
/*                            (This work is unpublished)                     */
/*                                 All Rights Reserved                       */
/*****************************************************************************/
/* Description:                                                              */
/* ------------                                                              */
/*      This file contains the code for the OPCODE function.                 */
/*****************************************************************************/
#include "basic.h"
#include "opcode.h"


/*****************************************************************************/
/* External variables specific to the instruction.                           */
```

```
/*****************************************************************************/


/*****************************************************************************/
/* Purpose:    This routine implements the OPCODE function for basic.        */
/*             It expects to receive from the object stream N                 */
/*             arguments.                                                     */
/*             Some optimization are done within this implementation          */
/*             to handle some common cases faster:                            */
/*                     - If both the first and second arguments are           */
/*                     of type string, we can make some assumptions...        */
/*                     - If both the first and second arguments are           */
/*                     of type string, and it is a full moon outside,         */
/*                     we can make some assumptions...                        */
/*             The algorithm used to implement this opcode is blabla...       */
/* Arguments:  No arguments for the C function but this opcode expects        */
/*             the following arguments from the object stream:                */
/*                     - Arg1 is the descriptor of the variable to            */
/*                             return the result into.                        */
/*                     - Arg2 is the mask for ...                             */
/*                     - Argn is the ...                                      */
/* Return:     ...                                                            */
/*****************************************************************************/
int
OPCODE()
{
        char            *first_arg_ptr;         /* Ptr to first argument        */
        int             first_arg_length;       /* length of first argument     */
        char            *second_arg_ptr;        /* Ptr to second argument       */
        double          third_arg_dbl;          /* Double value of third arg    */


        /*****************************************************************/
        /*****************************************************************/
        /* PARSE SOURCE ARGUMENT FLAGS.                                  */
        /* ASSIGN LOCAL DATA STRUCTURES FROM SOURCE ARGUMENTS.           */
        /* FORCE DATA TYPES IF NECESSARY.                                */
        /*****************************************************************/
        /*****************************************************************/


        /*****************************************************************/
        /* There are four arguments :                                    */
        /* Arg 0: the destination argument:                              */
        /*      This argument is the descriptor where the result of the  */
        /*      operation should go.                TO_BE_FILLED_IN      */
        /* Arg 1: source arg 1:                                          */
        /*      The first argument for this opcode ...  TO_BE_FILLED_IN  */
        /* Arg 2: source arg 2:                                          */
        /*      The second argument for this opcode ... TO_BE_FILLED_IN  */
```

```
/* Arg N: source arg N:                                                    */
/*       The Nth argument for this opcode ...        TO_BE_FILLED_IN       */
/*************************************************************************/


/*************************************************************************/
/* Prepare the first argument which is               TO_BE_FILLED_IN     */
/*************************************************************************/
if ( TYPE( OBJ_DESC( 1 ) ).type != STRING_TYPE )
        FORCE_DESC_TO_STRING( OBJ_DESC( 1 ) );


/*************************************************************************/
/* Prepare the second argument which is              TO_BE_FILLED_IN     */
/*************************************************************************/
if ( TYPE( OBJ_DESC( 2 ) ).type != STRING_TYPE )
        FORCE_DESC_TO_STRING( OBJ_DESC( 2 ) );


/*************************************************************************/
/* Prepare the third argument which is               TO_BE_FILLED_IN     */
/*************************************************************************/
if ( TYPE( OBJ_DESC( 3 ) ).type != NUMERIC_TYPE )
        FORCE_DESC_TO_NUMERIC( OBJ_DESC( 3 ) );


/*************************************************************************/
/* Setup some local variables:                                           */
/*************************************************************************/
first_arg_ptr = STRING( OBJ_DESC( 1 ) ).ptr;
first_arg_len = STRING( OBJ_DESC( 1 ) ).len;
second_arg_ptr = STRING( OBJ_DESC( 2 ) ).ptr;
third_arg_dbl = NUMERIC( OBJ_DESC( 3 ) ).float_val;




/*************************************************************************/
/*************************************************************************/
/* INSTRUCTION SPECIFIC CODE                                             */
/*************************************************************************/
/*************************************************************************/
{

/*      CODE SPECIFIC TO THIS OPCODE                                     */

}
```

```
        /*****************************************************************/
        /*****************************************************************/
        /* UPDATE THE DESCRIPTOR VALUE FOR THE RESULT.                   */
        /*****************************************************************/
        /*****************************************************************/

        if ( result_string_ptr != NULL ){
                /*********************************************************/
                /* COPY A NON NULL RESULT STRING TO THE TARGET           */
                /*  DESCRIPTOR.                                          */
                /*********************************************************/
                DESC_MALLOC( OBJ_DESC( 0 ), DISCARD, result_len );
                strncpy( STRING( OBJ_DESC( 0 ) ).ptr,
                                        result_string_ptr, result_len );
                STRING( OBJ_DESC( 0 ) ).ptr[ target_string_len ] = '\0';
                TYPE( OBJ_DESC( 0 ) ).type = STRING_TYPE;
        }
        else{
                /*********************************************************/
                /* SET THE DESTINATION DESCRIPTOR TO A NULL STRING.      */
                /*********************************************************/
                SET_DESC_TO_NULL_STRING( ARG( 0 ) );
        }                                       ⇓
                                                OBJ-DES

        /*****************************************************************/
        /*****************************************************************/
        /* INCREMENT THE OBJECT CODE POINTER PAST THIS INSTRUCTION       */
        /*****************************************************************/
        /*****************************************************************/
        INCREMENT_OBJECT_PTR;
}
/*****************************************************************/
/*                          OPCODE.c(end)                        */
/*****************************************************************/
```

b. Accessing the Basic object and opcode arguments:

An external **uchar** * named **object_code_ptr** will always be pointing into the object code. The main parsing loop will decode the opcode and position the **object_code_ptr** at the byte following the opcode.

Each *opcode.c* file should have a corresponding *opcode.h* file. That include file should have in it first a

definition for the object code format for that opcode. For example if my opcode expects two variables and a two byte length after the opcode in the object, my structure definition should look as follows:

```
typedef struct{
        arg_struct      arg[ 2 ];       /* Two variables                */
        short           length;         /* two bytes length             */
}opcode_object_struct;
```

Any variables should always be put together into an array in that structure and should be of type **arg_struct**. The structure **arg_struct** is defined for the time being as follows (it could very well change as the project evolves):

```
typedef struct{
        short           vector_index;   /* index of the descriptor table in  */
                                        /*  the main vector                  */
        short           desc_index;     /* index of the variable in the      */
                                        /*  descriptor table                 */
}arg_struct;
```

Notice that this structure basically defines the way a variable is represented in the object stream.

Note that throughout the code the word ***desc*** *will always refer to a descriptor!*

The file *opcode.h* will also have one more declaration. That is the declaration for the variable **object_code_ptr**. Here we are going to cheat. That variable was defined as a **char** * but we are going to declare it differently in every *opcode.h* file. It will be declared as of type **opcode_struct** *. This allows us in the C code for that opcode (*opcode.c* file) to be able to say **object_code_ptr->length** for example to get the two bytes field from the object stream. The advantage of

doing that is that our code will not do any byte arithmetic as the C compiler will align the data structures accordingly (and on the HP automatically avoid bus exceptions!). **The draw back is that if the compiler on different machines aligns the data differently, the new Basic object will not be compatible between those different machines.** Notice though that some effort can be invested to attempt to provide object code compatibility on the machines we support through the way we declare the data structures. If some incompatibility can happen though, the runtime should be capable of identifying the fact that the object code that is attempted to run is not compatible with the machine being run on.
Therefore we will do the work to ensure that the data structures are aligned similarly on different machines.

As you notice, the template has as a last line **INCREMENT_OBJECT_PTR** (a macro). That positions the **object_code_ptr** past the opcode arguments onto the next opcode. The reason this is done in the opcode code itself is that because of the way the **object_code_ptr** variable was declared, this macro equates to **++object_code_ptr**. The C compiler calculates how many bytes it is appropriate to increment the pointer by.

On the other side of the fence, the *new object code generator* will be using the same data structures and include files *opcode.h* to produce the object ensuring proper handshake between the compiler and the runtime.

We consider the concepts described in this section as pretty critical. That is because we will not be doing any byte arithmetic in this project **minimizing** the number of bugs we will have in our code and increasing our productivity!!!

So we basically have to deal with three types of elements: object code, arguments and descriptors. We

have defined macros to go from one type to another.

```
#define     OBJ_ARG( n )        object_code_ptr->arg[ (n) ]
#define     ARG_DESC( arg )     vector[ (arg).vector_index ][ (arg).desc_index ]
#define     OBJ_DESC( n )       ARG_DESC( OBJ_ARG( n ) )
```

Vector is the pointer to the base of the two dimentional array that represents all of the variable on the system.

Note that the first element of the *arg* array in the *opcode_struct* structure should always be the descriptor to where the result is going if that is applicable for that opcode. This allows us to rely on the fact that the destination will always be **OBJ_DESC(0)**.

c. Accessing descriptors:

The following macros give us access to the different fields in the **desc_struct** structure. Those macros practically hide the different levels in the **desc_struct** structure (unions and sub-structures) giving a single level of access (there will be one occurence of the 'dot' command)
They are all defined in the *desc.h* source file.

```
#define  TYPE( desc )        OBJ_DESC( (desc) ).fixed.flags.byte.type
#define  STRING( desc )      OBJ_DESC( (desc) ).fixed.string
#define  NUMERIC( desc )     OBJ_DESC( (desc) ).overlay.numeric
#define  FILE( desc )        OBJ_DESC( ( desc ) ).overlay.file
#define  SELECT( desc )      OBJ_DESC( ( desc ) ).overlay.select
#define  FORMAT( desc )      OBJ_DESC( ( desc ) ).overlay.conv.format
```

There will definitely be more definitions in that file as the *desc_struct* structure evolves. The concept though should remain the same. Notice that we practically can get to any field in the structure by using a reference of the form **DESC_type(ARG(n)).field**. That makes it consistent throughout the code.

d. Dynamic memory allocation:

Heap management suddenly becomes an important issue with this rewrite. Up till now, most heap requirements for Ultimate PLUS were channeled through the **frame manager** as all of the memory used has been in frames. With this rewrite, workspaces for Basic programs are no more in frames but rather in the process's heap. Dynamic memory management is one area where bugs in C get pretty nasty. For that reason, we are going to channel all of our dynamic memory requests through an interface that will evolve to provide elaborate debugging capabilities. No code in the system should be doing any *malloc()* or *realloc()* or *free()*. Instead the following set of macros should be used:

DESC_MALLOC(arg, flags, length)

The first argument is of type *arg_struct*, the second is a flag that affects the behavior of the heap manager and the last one is the desired length of the memory piece to allocate. That macro will allocate a piece of memory of size *length* and will make the string pointer field of the descriptor point to it. If the string field of the descriptor was already pointing to a piece of memory of less or equal size, the macro will just return without doing much. The flag can be used to indicate if the data that is currently in the string field of the descriptor needs to be preserved or not. If the flag **PRESERVE** is used, after the new chunk of memory is allocated the data from the old string field of the descriptor is copied into the new chunk of memory. If the flag **DISCARD** is used no data is copied.

If the string field of the descriptor is already pointing to a previously allocated chunk of memory, **DESC_MALLOC** will take care of freeing it.

DESC_FREE(arg)

The macro **DESC_FREE** takes an argument of type **arg_struct** and frees the memory that is pointed to by string pointer field for the passed descriptor.

Both of those macros might evolve some more over the next month or two as people start programming and further needs are identified. It is in our intention to provide tracing and accounting capabilities within those two heap management macros. This should for example help identify a piece of code that does not free memory as it should making the process grow his heap indefinitely.

The macros in the first cut will most probably make use of the C runtime library *malloc()* , *free()*, and *realloc()* although this might change if those routines are found inappropriate in the future for performance reasons. The use of the macros allows us to modify the heap management implementation without having to modify the source code.

e. Useful macros:

A macro that is worth mentioning is the one used at the end of the template:

SET_DESC_TO_NULL_STRING(arg).

That macro will set the string field of a descriptor to an empty string.

f. Procedure for developing an opcode:

Here is a brief listing of the steps to follow when a new opcode is being coded:

- If this opcode has a corresponding opcode in the old

object code, print out a listing of the virtual mode(s) for this opcode and go through it making sure you get a thorough understanding of the functionality of that opcode. Not all the functionality is highlighted in the user manual!! That is why going through the virtual code is important.

- Run some benchmarks on Ultimate PLUS 21xEn and on a competitive implementation (Ult/ix or Pick Blue) that exercises this opcode. The purpose of this exercise is to set a performance goal for this opcode rewrite! We have to be able to be at least equal to our competitors performance.

- Make a copy of the files *opcode.c* and *opcode.h* and rename them according to the opcode being developed.

- Define the opcode structure *opcode_struct* in the include file by understanding the arguments that this opcode is being passed.

- Communicate to the programmer coding the compiler that this new data structure exists as he will need to fill it at compile time.

- Customize the comments in the *opcode.c* file describing the functionality of this opcode, its arguments, its result and the algorithm used and anything else appropriate. Do not leave the comment to the end!!!

- Customize the first section of *opcode.c* where all the arguments are forced to the appropriate type.

- Develop the code specific to the opcode.

- Do some preliminary testing.

- Print out a listing for the newly written code and choose another programmer from the Basic rewrite team and review the code with him. Make sure that the

person you picked is responsive and helpful otherwise it is of no benefit to anybody.

C. The optimizer:

1. Architecture of the optimizer:

What has commonly been referred to so far as the optimizer is really a little bit more than that. The compiler for the new Basic is really made of the old Basic compiler plus some additional modules that we are to develop. These additional modules have been referred to so far as the optimizer although the optimizer is really just a piece of it (a significant one). This new piece is made of three modules:

- **A parser**: The parser will parse the old object code and generate some data structures that are suitable for optimization. That parser is pretty straight forward to implement. The hard part is the data structures that it will be filling out. Those will change as they are found appropriate for the optimization algorithms.

- **An optimizer:** This is really the optimizer. It will use traditional compiler optimization techniques to squeeze as much performance as possible from the user's code. The fact that this optimizer should produce some measurable performance gains is in no way a reflection on the user's bad coding. Some optimizations that the optimizer is capable of doing are pretty hard to foresee for a human being. Moreover as dealer's applications grow with time, they will fatten with inefficiencies because the code gets modified in pieces and stops being as compact as on the first day.

The optimizer will operate on the data structures that were built by the parser and will result in the same data structures but may be shrinked or reshuffled.

- **A code generator:** This module will run against the data structure that the parser generated and that the optimizer optimized. It will produce out of it a stream

of object in the new Basic object code format. The most critical responsibility of this module is to generate object code that looks like the runtime expects it (the *opcode.h* structures are used for that purpose). It is also important that this code generator can be easily modified to generate different object. This is important because we foresee that the object code format might change over the life of this project and especially in the first few weeks.

2. Support for the New Basic Runtime:

Out of the three modules described in the previous section, the **parser** and the **code generator** are in the critical path for the release (with the runtime rewrite). We therefore will complete those two before we work on any optimization techniques. The optimizer will only be implemented as time permits.

Other than just producing object code in a format that is appropriate for the runtime, some other tasks are expected from the code generator:

- **Temporary variables management:** The object should contains explicit references to temporary descriptors in the *temporary descriptor table*. This will be done by the code generator.

- **Constants descriptor table**: A constant descriptor table will be produced at the end of the object to be loaded by the runtime at initialization time. The object will contain the proper references to those constants where appropriate.

- **Source/Object map:** The new Basic debugger will expect a table at the end of the object that will contain a mapping of line number's between the initial Basic sources and the produced object. This is to help symbolic debugging.

- **Symbol table:** A symbol table will also be produced

at the end of the object to help symbolic debugging with variable names...

- **Debugging:** The code generator should be capable of producing a dump of its internal data structures in a humanly readable form to help track down any problems with it.

3. Optimization techniques used:

V. Compatibility with current Basic:

This section discusses topics that are a little bit touchy as they may represent design or implementation challenges in our seek for a fully compatible implementation. Any things that we do not think we will support are highlighted in their respective sections. Most of the topics covered in this section are backed by a full appendix that reflects the research that was done.

A. Basic source compatibility:

It is the intent of this project to maintain **full** compatibility with Ultimate's Basic implementation. All the functionality should be the same. It is not clear what should be done about things that turn out to be bugs in the old Basic as some customers could be relying on them. Those need to be reviewed and assessed one by one.

B. Named Commons:

The 'named common' data concept is fully supported by optimized Basic. Data can be shared between both the old and the new runtime environments.

The implementation requirements are as follows:

The compiler part of the optimizer stores, in the constants section of the new object code, the name of the common

block(s) that the runtime may access.

During runtime, when the 'named common' instruction is executed, a new routine loads the current values of the named common block into the runtime environment. These values are stored in the old ten byte descriptor format. Conversion to the appropriate descriptor type and scaling factor occurs at that time.
This new piece of code interfaces to the current virtual modes in order to scan the 'named common' table and fetch the block address.

As part of the clean up code when the program terminates (via opcodes EXIT or CHAIN, or via the debugger), a similar routine writes out the new data values of the common block, in the old descriptor format, with all numeric values converted to string.

A special case is the EXECUTE instruction which may invoke a Basic program using the same common data block. In this phase of the project we are going to update and restore the data values of the 'named common' block for this instruction.

Refer to Appendix E for an in depth overview of the 'named common' feature.

C. Calls:

Optimized Basic can not CALL old object code, and vice versa, because of the complexity involved in building a bridge interface between the two environments, especially in regard to common variables.

Both direct and indirect call formats are supported.

The implementation requirements are:

The CALL opcode interfaces with parts of the current virtual code to retrieve the object code for the called routine.

A descriptor is kept for each direct call, so that the object location only needs to be resolved once.

For indirect calls, the OPEN 'SUB' instruction is supported.

The called routine inherits the primary descriptor vector table (refer to Section IV.B.3 about *Basic variables and constants*) from the caller (like other global variables) thereby gaining access to the 'common' data variables. Sections of the vector table that are unique to the subroutine (local variables descriptor table, local constants table,...) are saved and initialized to the new addresses.

The CALL instruction and the subroutine both contain a count of the number of arguments passed, which must match. Arguments are copied between environments in a manner similar to the current implementation.

Refer to Appendix N for an in depth discussion of CALL.

D. Basic Debugger:

All of the current debugger functionality will be provided in the rewrite. Some more features have identified as desirable and may be implemented as time permits. It is desirable for some of those features to even replace some old ones as time goes by. The debugger will look different in functionality when a program has been fully optimized as some capabilities are either too hard or impossible to provide in that situation. Please refer to appendix F for any more detailed information.

E. Library Calls:

The majority of the existing library calls do not have to be supported since they are used for hardware specific tasks that are not available on the Ult/PLUS platform (Vterm, 1400 diskette driver, IBM performance monitor,...). However a number of them are used in general purpose Basic routines and need to be supported.

The interface between the new runtime and the virtual code is the

same as for other instances where virtual code needs to be invoked.

In the new environment, the advantage of using library calls over conversion user exits is not so strong because descriptor values can no longer be updated in place.

In a few words, there does not appear to be any problem in implementing the library interface. The code for the generic interface to a virtual mode will be used here to interface to the virtual code for libraries.

Refer to Appendix J for a more detailed description of the 'library call' interface.

F. Execute / Chain / Data Statements:

Execute: New and old format Basic programs can be executed from the optimized runtime environment.
The implementation requirements are:

Using the standard interface from optimized Basic to current virtual, a number of arguments need to passed to virtual, either as data or as pointers.

Through simulation of the current interface, or through a new routine, the 'execute' Tcl level needs access to some of the passed arguments from the Basic runtime.

With the use of routines to update descriptor data, information returned by the execute command needs to be passed back to the Basic runtime environment.

For a detailed description, refer to Appendix O.

Chain/Enter: The 'CHAIN ...(I)' command and the 'ENTER' instruction are NOT supported because:

The complexity of the code involved and the likelihood of causing problems & bugs;

It makes optimization at the descriptor level impossible since there is no way to know if and when a program will be ENTERed, and how the previous descriptor space is going to map to the new one.

The reason for the existence of these features is that in the old days the object size was limited to 32K, and this was the only way to bypass that restriction. In the new environment object sizes can be large enough so that this is no longer an issue. Source code can easily be kept in small portions via the $INCLUDE directive.

So the "CHAIN...(I)" and the ENTER instruction will not be implemented in the first phase. If there is a need for any of those two features, it will be implemented in phase 2. Other 'CHAIN' commands are supported.

G. Interface from Recall and Update to Basic:

The complexity involved in pre-initializing a descriptor table, accessing it from within virtual and maintaining the runtime environment in between calls, all make these features difficult and time consuming to implement.

In the case of Recall subroutines, the time usually spent inside the Basic routine is very short, and the performance gain to be expected from optimization can only be small compared to all of the Recall processing. Because of the common variables section that must be included, these subroutines can only be used from Recall.

The easy solution will be to take advantage of the fact that the old Basic runtime will remain part of the Ultimate PLUS system. We would therefore still invoke the same virtual code as today. In the second phase of the project we can make sure that these are implemented.

Refer to Appendix H for a detailed description of the Recall calling Basic interface and of the issues involved in optimizing it.

VI. Issues and questions that need to be answered:

This section lists issues and questions arising from this design effort. These need to be resolved as soon as possible to ensure that the project is on the right track. Every one's input will be valuable from our management to our marketing department.

- How much more space is it acceptable for the new object to occupy without becoming an issue! If a Basic program produces today a 1Kb object item, how big can it be for this rewrite? Same? 10 Kb? ...

- In view of the analysis done on the math, how many flavors of arithmetic do we really need to implement: Ult/ix flavor, Ultimate flavor, correct math, others...

- Is string arithmetic still a requirement for this implementation considering that the overflow situation in the new implementation is different (see Appendix G).

- Is Basic object code compatibility required between different Ultimate PLUS implementations? See section about *Accessing the Basic object.*

- Testing the compatibility of the rewrite will be a real challenge. The ideal case would be for us to have some test software (automated test) that can validate the rewrite. Such a product would help the stability of the rewrite tremendously. It is not clear at this point if this product could be purchased or has to be completely developed in house.

- Documentation, QA, alpha and Beta are topics that complement the R&D work to make the Basic rewrite become a releasable product. Therefore a commitement and a final schedule are necessary from the responsible groups.

- Supporting the **"chain ... (I)"** and the **"enter"** statements requires some significant effort. Is it acceptable to not provide support for these statements in the first cut? Is it acceptable to not provide support for those statements ever?

VII. New Basic performance analysis:

The questions that every one is going to ask when reading this document are *how fast will new Basic be? How do you know you are going to reach your goals? etc...*

These are tough questions to answer as they will commit the team to specific performance numbers before most of the code has been written. So let me try to summarize our thoughts about this topic. There is no doubt what so ever that the rewrite will produce a system that is substantially faster than the current basic. Our goal is to be comparable with our competition meaning worst case is same speed as competitors and best case is faster. There is very little choice about whether or not we will reach the goal; **we have to be comparable.**

Some facts now to try to substantiate the feeling that we will meet our performance goals. First, since many of our constraints (machine, memory, compiler, math routines ...) are the same as our competitors, we should be able to produce comparable results. Second we have actually rewritten some opcodes already. For some of those rewritten opcodes, (very few) out first pass implementation obtained a performance level that is much higher than todays Ultimate PLUS but still slower than Ult/ix's. For most of the opcodes that were rewritten, we did end up with a performance level equal or higher than Ult/ix. Generally the rewritten opcode was two to three times faster than the current Ultimate PLUS implementation. Those timings though do not guarantee the overall desired result, as the architecture was not fully defined when they were done. But those timings do confirm our gut feeling that we can meet the goals. The month of February will be critical in producing more firm timings information that will better reflect the end result.

One last thing; the optimizer is our wildcard. None of our competitors currently have a real optimizer. So the optimizer will help us gain some additional performance in our new implementation. The question here is how much of the optimizer we will be able to implement in the first phase.

VIII. Project Implementation:

The project is basically divided in two phases. The first one to complete in

the first half of 1992 will contain a fully rewritten Basic runtime. Some odd pieces might have not been implemented in the first phase to be able to meet the required deadline. Two such pieces will be be Recall calling Basic and perhaps the optimizer (the part that does the real optimization).

A. Quality plan:

It is our goal during the implementation of this project to maximize as much as possible the quality of the code produced to allow the product to stabilize as quickly as possible. We all are human, and therefore the product will have bugs (specially considering the scope of the rewrite). But through good methodology and some procedures we will hopefully succeed in keeping the programming mistakes to a minimum. Here is a list of the things we are doing or we plan on doing during the implementation of this project:

1. **Templates:** Considering that many many opcodes are going to be written, it made sense to define an *opcode template*. That template is a piece of code that contains the framework that would be common for most opcodes. On one hand it will standardize the way the newly written opcode routines look and on the other hand it will minimize mistakes in the portion of the code that is similar for most opcodes. Templates also shorten the time the programmers spend on each opcode as they would not have to worry about code that is common to many opcodes.

2. **Macros:** Instead of defining macros, data structures and include files as the project evolves, we have decided to try to do the most important ones up-front.. We have invested a considerable amount of time up front to design the macros, data structures and include files that will be useful throughout the project. This is leading to better thought and better designed macros. These are important as every programmer will use them throughout the project. When those are designed, emphasis is being put on their readability.

3. **Code reviews:** As a programmer on the Basic project develops new opcodes, he will choose another programmer from the team that he wants to review his work. This is not an easy concept to implement as it has to not be confrontational but

rather productive in finding deficiencies and educational as various programmers share their knowledge. All Basic team members have agreed to have their code reviewed by others.

4. **Virtual reference:** Considering that we are rewriting an existing piece of code, it is very important that we take advantage of the information that is in the old code. We will therefore, for every opcode that we rewrite, use the existing *virtual code* as a reference to ensure that the new version behaves as the old one. The appendices at the end of this document indicate that we have already thoroughly examined various pieces of the current virtual implementation to better understand the pitfalls.

5. **Technical documentation:** During the work for this document, a lot of technical information about the current system was gathered and summarized in the appendixes at the end of this document. As we proceed with the implementation, more knowledge will be gathered about the current system and about the new system. We will make sure that the knowledge gathered is put back into the appendixes to this document.

B. Phase 1:

1. Goals:

The main goal of this first phase is to provide a Basic implementation for Ultimate PLUS that is comparable to the Basic performance of our competitors namely Universe and Pick Systems and at least two and a half times faster than today's Ultimate PLUS Basic performance. This phase has to be on schedule and has to stabilize fairly quickly! Any tasks that are identified to not be critical to the overall performance can be delayed to phase two of this implementation. This will allow us to maximize our chances of meeting our goals. One good example can be update calling Basic. If that represents a significant piece of work we might delay its implementation until phase two.

As it is most important for the runtime implementation to

complete (there is not much of a product without it), we will delay any work on the optimizer until we feel comfortable that the project is proceeding on schedule. Not all of the optimization techniques need to be implemented in the first phase. Some of the more complex ones can be delayed until the second phase.

2. Resources:

From an R&D stand point, the five people currently on the project are enough for a timely completion of the development. We will have to be carefull that these resources do not get side tracked on other emergencies frequently.

From a machine stand point, it does not seem necessary at this time to have more resources than what we currently are working with. We will try to scatter our efforts on more than one machine.

Testing is an issue at this time. It would be very helpfull to have a test suite for Basic to help the product stabilize more rapidely. If such software can not be purchased, it might make sense to allocate some resources (at a later time, may be when the Mips project is complete) to develop such test suite.

It is not clear yet what resources would be required from documentation and from QA. This would have to be determined at a later time.

3. Schedule:

Dec. 3	Coding starts
Jan. 31	Functional prototype for the new Basic.
Feb. 28	The C-rating runs with the new Basic without the **execute** part and the spooler part of the test.
April 20	Demoable system, ready to be QAed for the dealer show

April 20	Start work on the functional specification for the compatibility document between the old and the new Basic. See Appendix Q.
May 1	Probench runs with the new Basic.
May 15	Working demo version for the dealer show.
June 1	Ultimate PLUS release 227 with the Basic rewrite goes to QA.
July 1	227 Alpha starts
July 15	227 Beta starts.
August 1	227 is ready to ship!!

The RS6000 release (228) will lag behing the HP release of 227 by a month. Note that the documentation, QA, alpha and beta schedules commit resources that are outside of the Basic rewrite team. Therefore the dates that relate to QA, alpha and beta are suggested dates that can change.

The order of 227 and 228 releases can be switch if necessary.

C. Phase 2:

1. Goals:

Complete any pieces that were left out in the first phase. This would include enhancing the optimizer with any optimization techniques we would not have had the time to put in. Other modules that might have not been implemented in the first phase could be Recall calling Basic or further enhancements in the basic debugger.

2. Schedule:

It is not clear yet what the deadline will be for this phase as we

do not know yet for sure which things would be included in there.

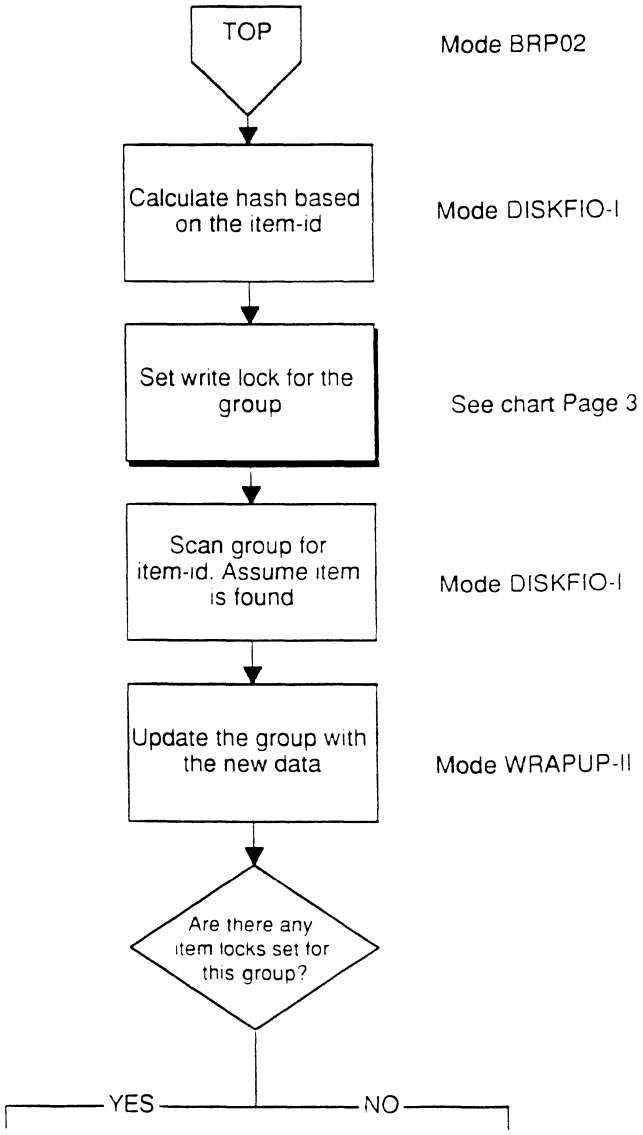
Appendix A: Basic READ/READU/WRITE flowcharts, as per release 210E

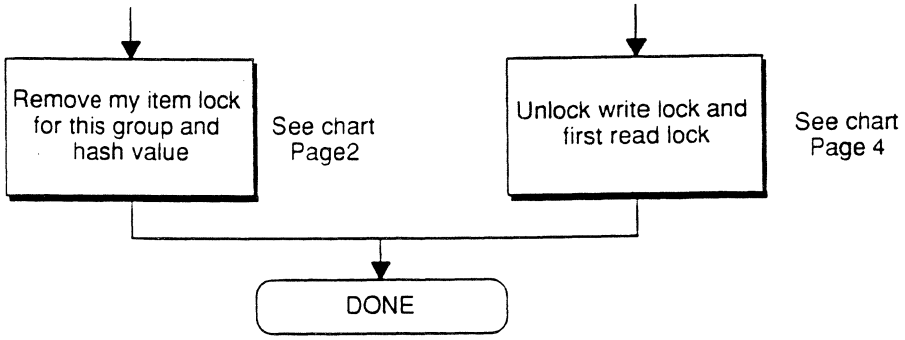
Topic: Steps taken by the Basic WRITE instruction, specifically with regards to locking.

Thursday, September 5, 1991

Page 1

Basic write



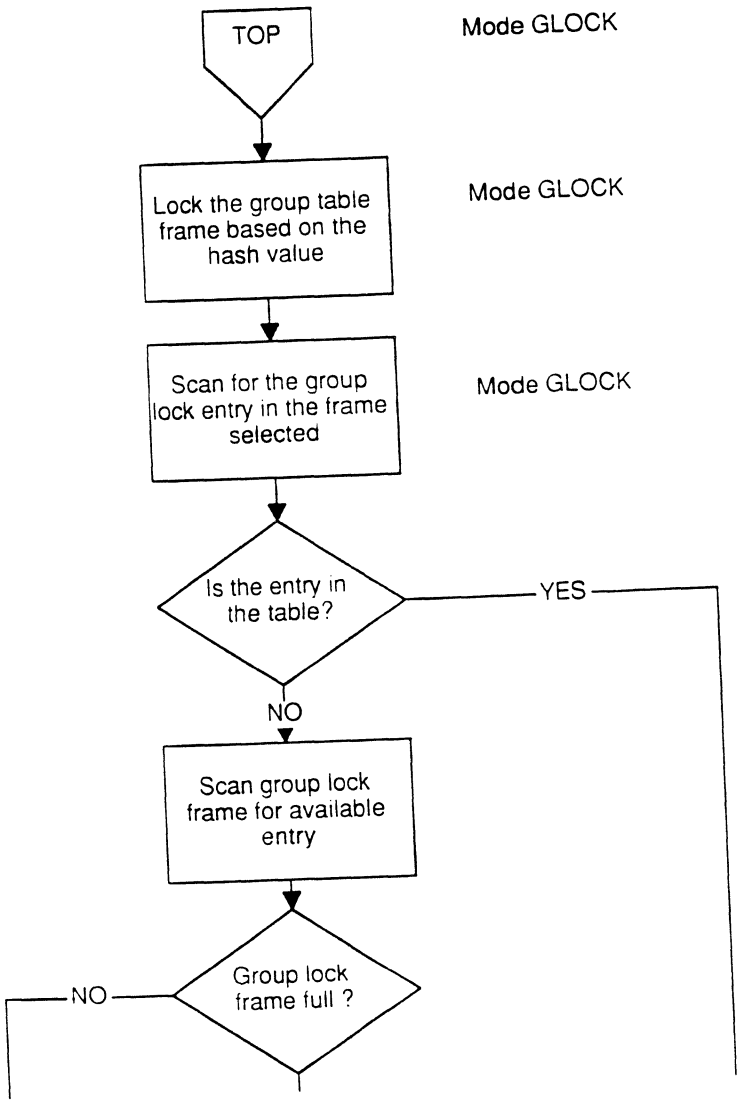

Remove my item lock for this group and hash value
See chart Page2
Unlock write lock and first read lock
See chart Page 4
DONE

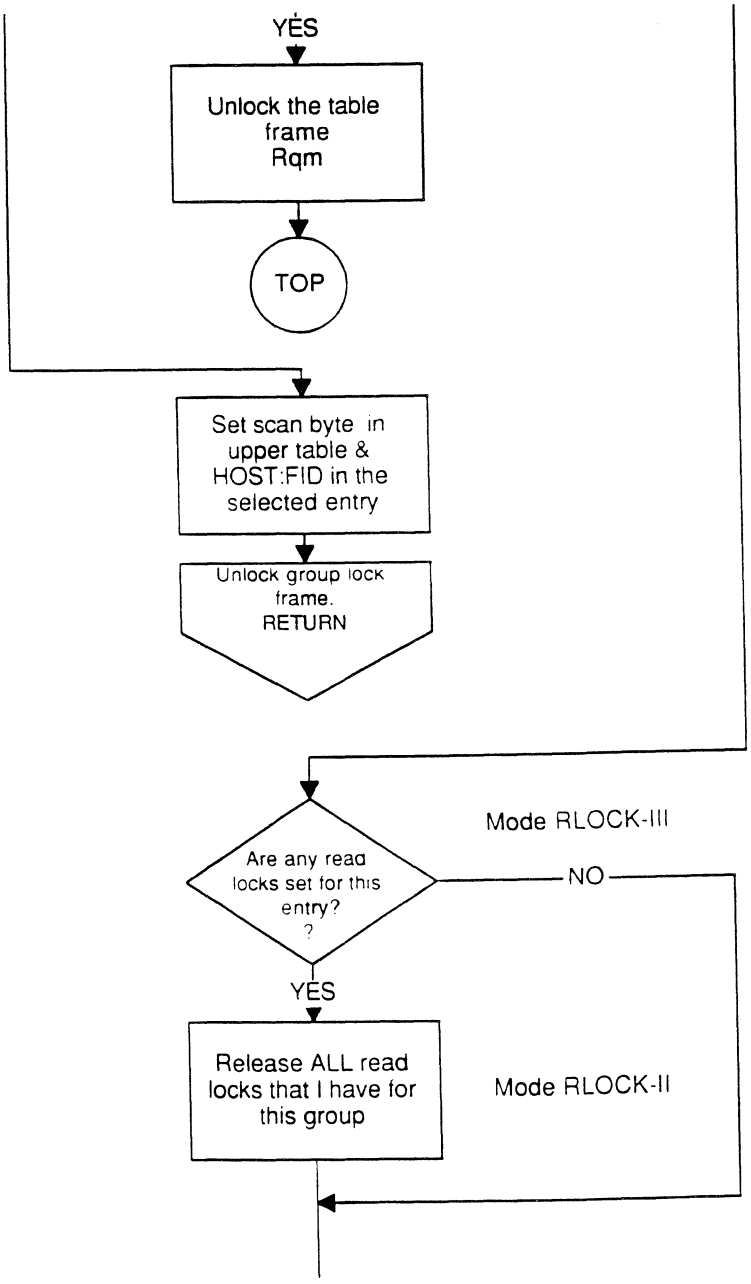
Topic: Steps taken by the Basic WRITE instruction, specifically with regards to locking.

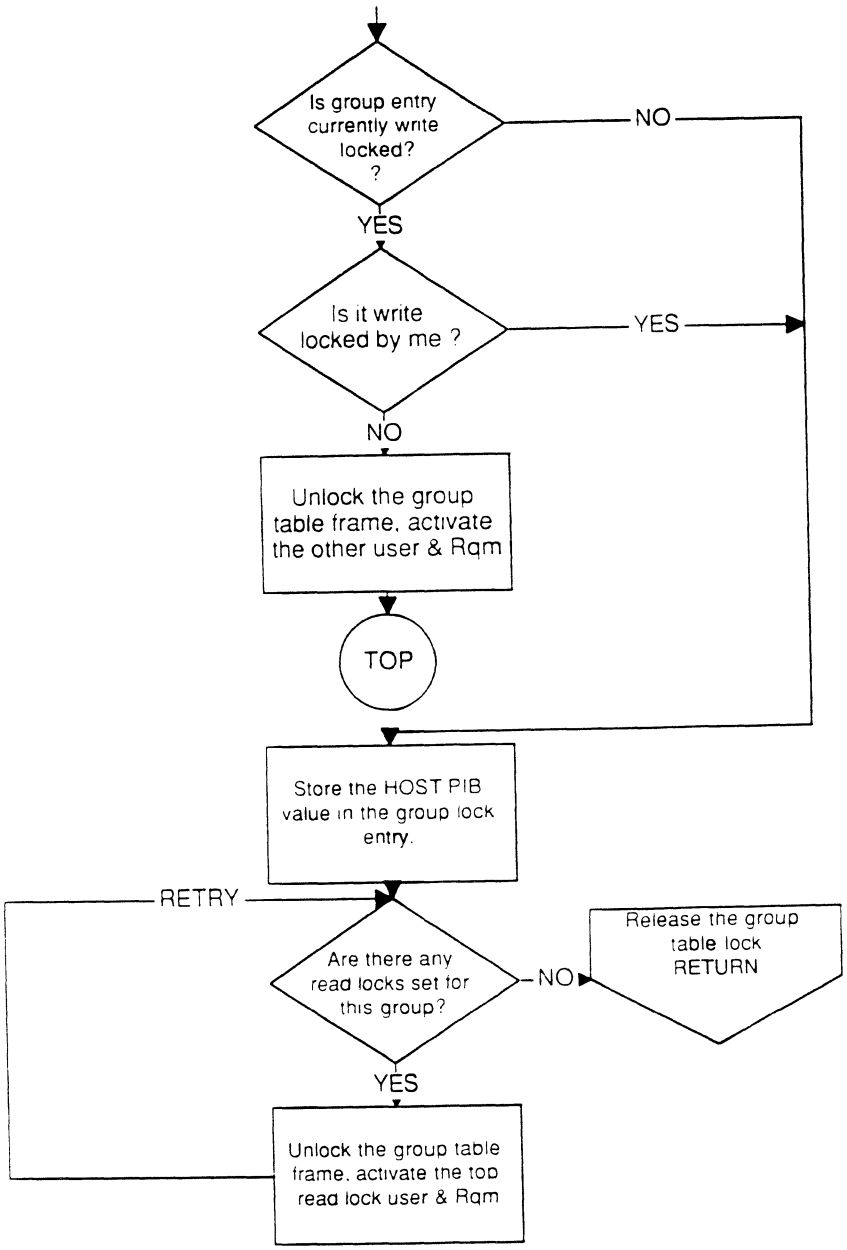
Thursday, September 5, 1991

Page 3

Set write lock for the group




YES
Unlock the table frame Rqm
TOP
Set scan byte in upper table & HOST:FID in the selected entry
Unlock group lock frame.
RETURN
Mode RLOCK-III
Are any read locks set for this entry?
NO
YES
Release ALL read locks that I have for this group
Mode RLOCK-II

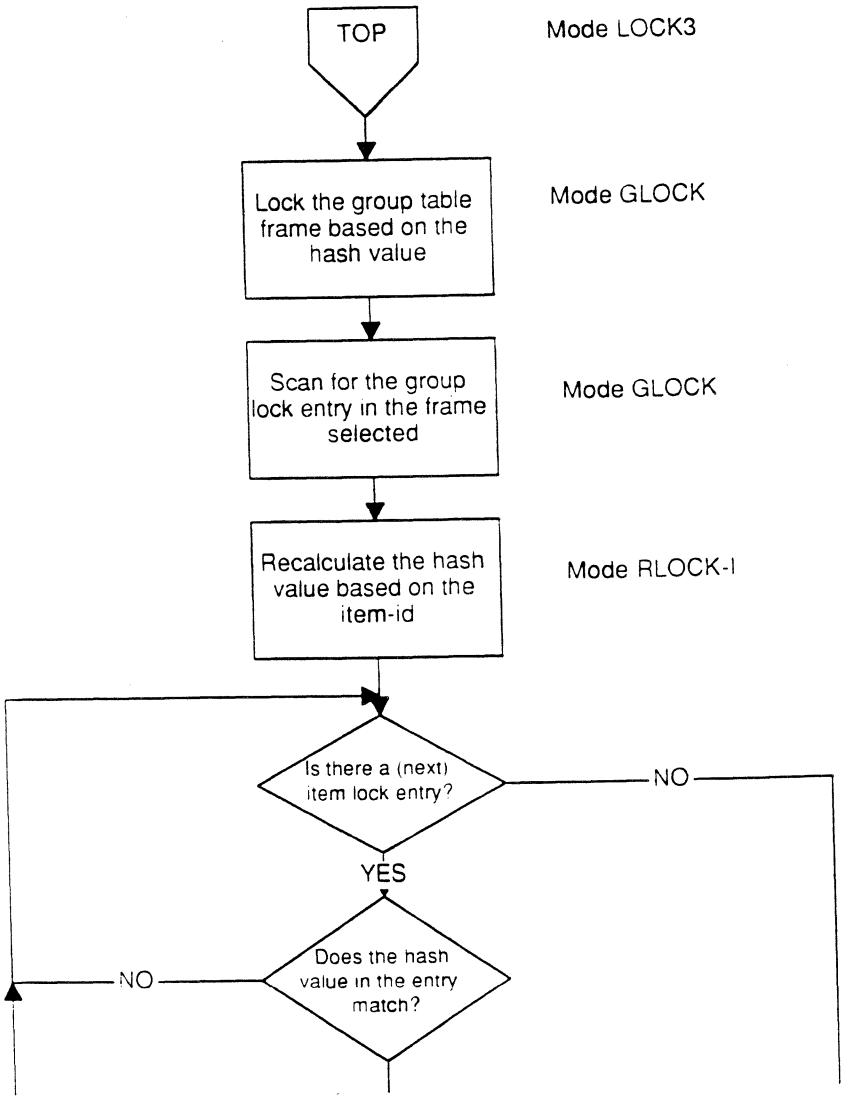

Is group entry currently write locked? ?
NO
YES
Is it write locked by me ?
YES
NO
Unlock the group table frame, activate the other user & Rqm
TOP
Store the HOST PIB value in the group lock entry.
RETRY
Are there any read locks set for this group?
NO
Release the group table lock RETURN
YES
Unlock the group table frame, activate the top read lock user & Rqm

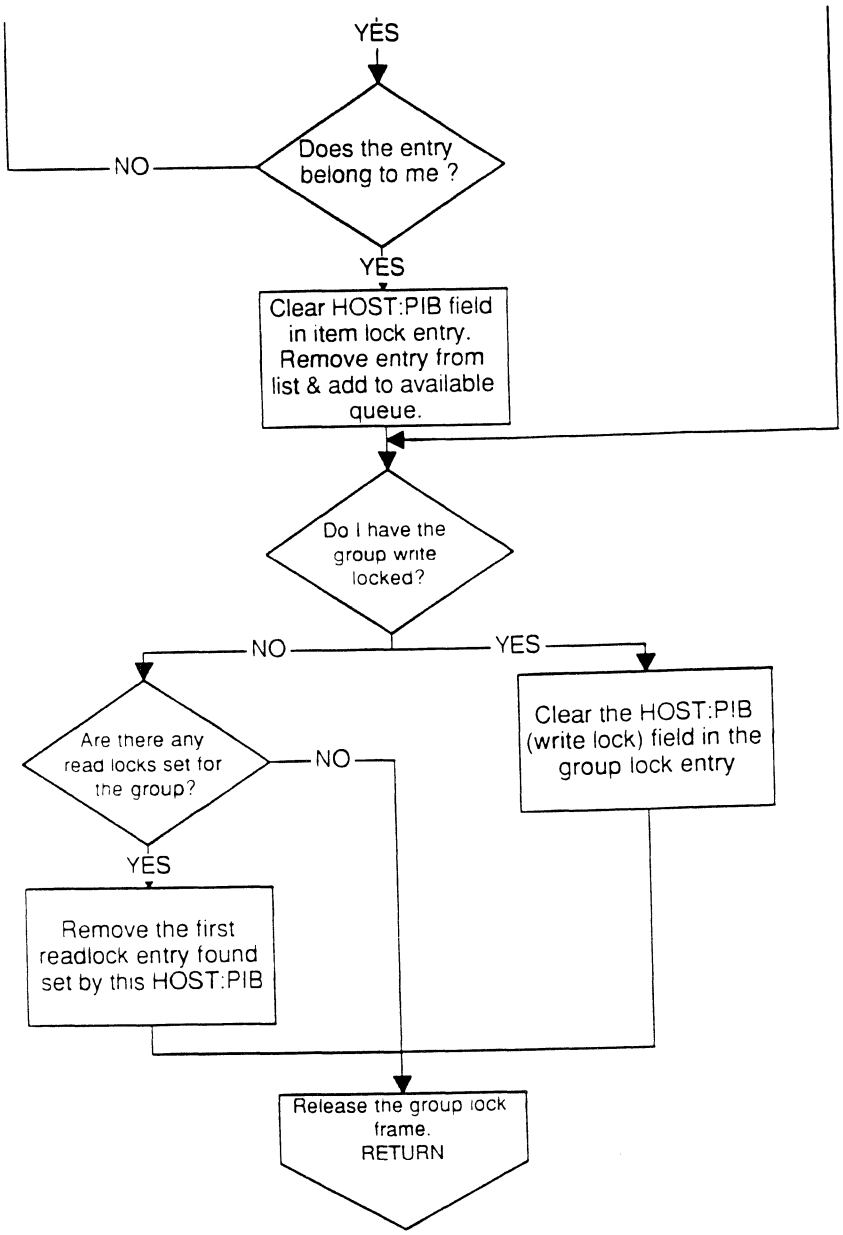
Topic: Steps taken by the Basic WRITE instruction, specifically with regards to locking.

Thursday, September 5, 1991

Page 2

Remove my item lock for this group and hash value



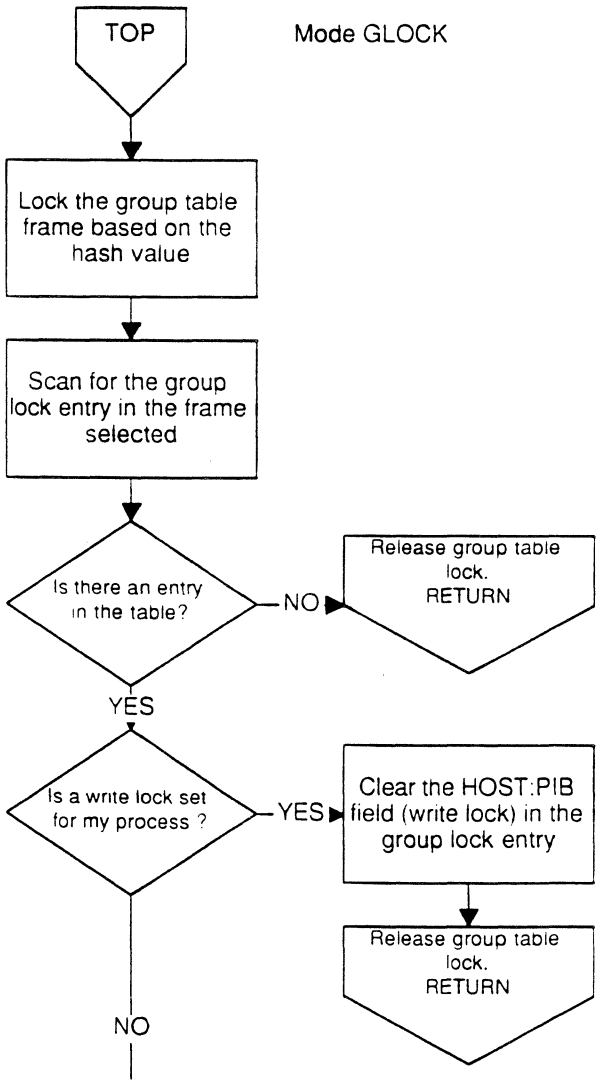

YES
Does the entry belong to me ?
NO
YES
Clear HOST:PIB field in item lock entry. Remove entry from list & add to available queue.
Do I have the group write locked?
NO
YES
Are there any read locks set for the group?
NO
YES
Clear the HOST:PIB (write lock) field in the group lock entry
Remove the first readlock entry found set by this HOST:PIB
Release the group lock frame.
RETURN

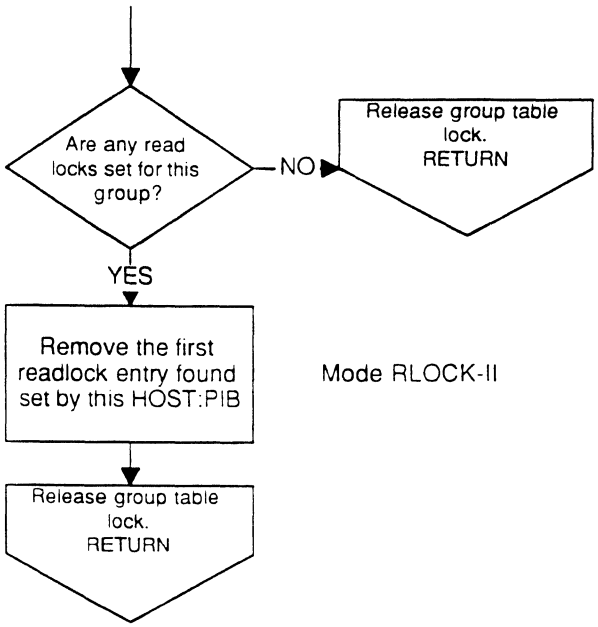
Topic: Steps taken by the Basic WRITE instruction, specifically with regards to locking.

Thursday, September 5, 1991

Page 4

Unlock write lock and first read lock





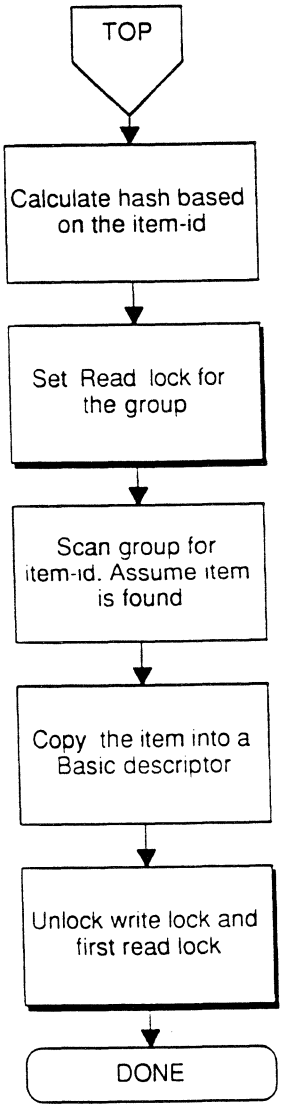
Mode RLOCK-II

Topic: Steps taken by the Basic READ instruction, specifically with regards to locking.

Thursday, September 5, 1991

Page 1

Basic read



Mode BRP02

Mode DISKFIO-I

See chart Page 3

Mode DISKFIO-I

Modes BRP02 & BRP24

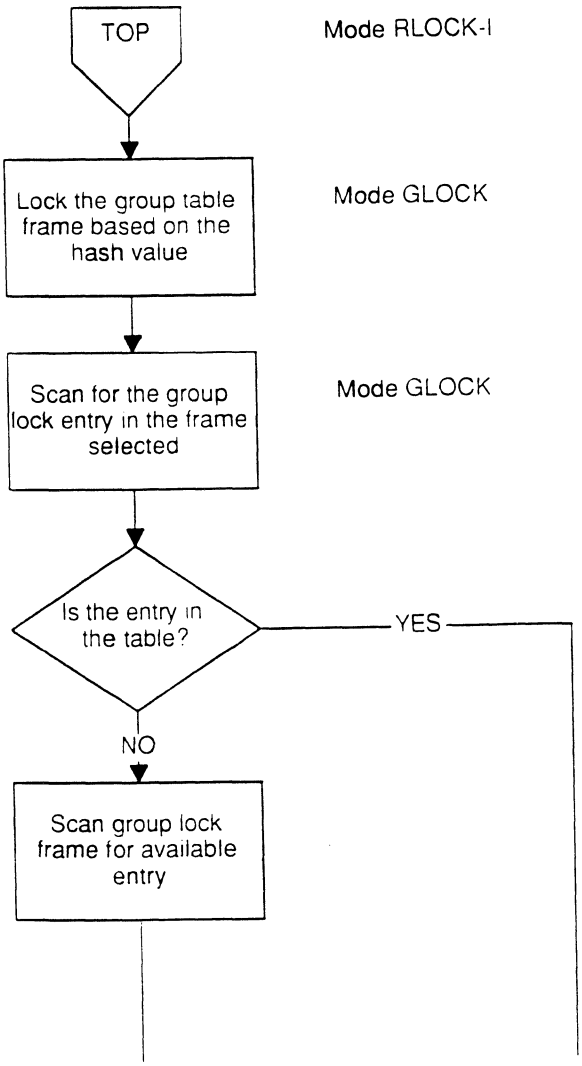
See chart Page 4

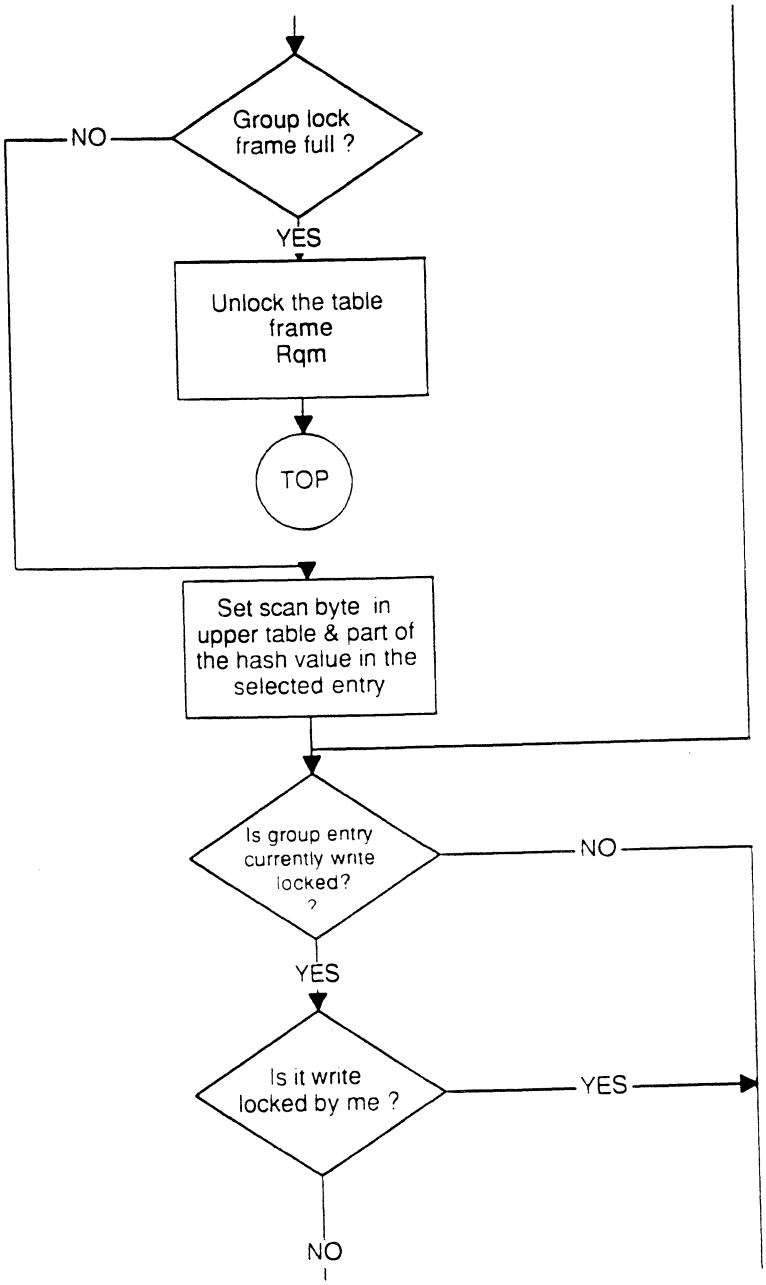
Topic: Steps taken by the Basic READ instruction, specifically with regards to locking.

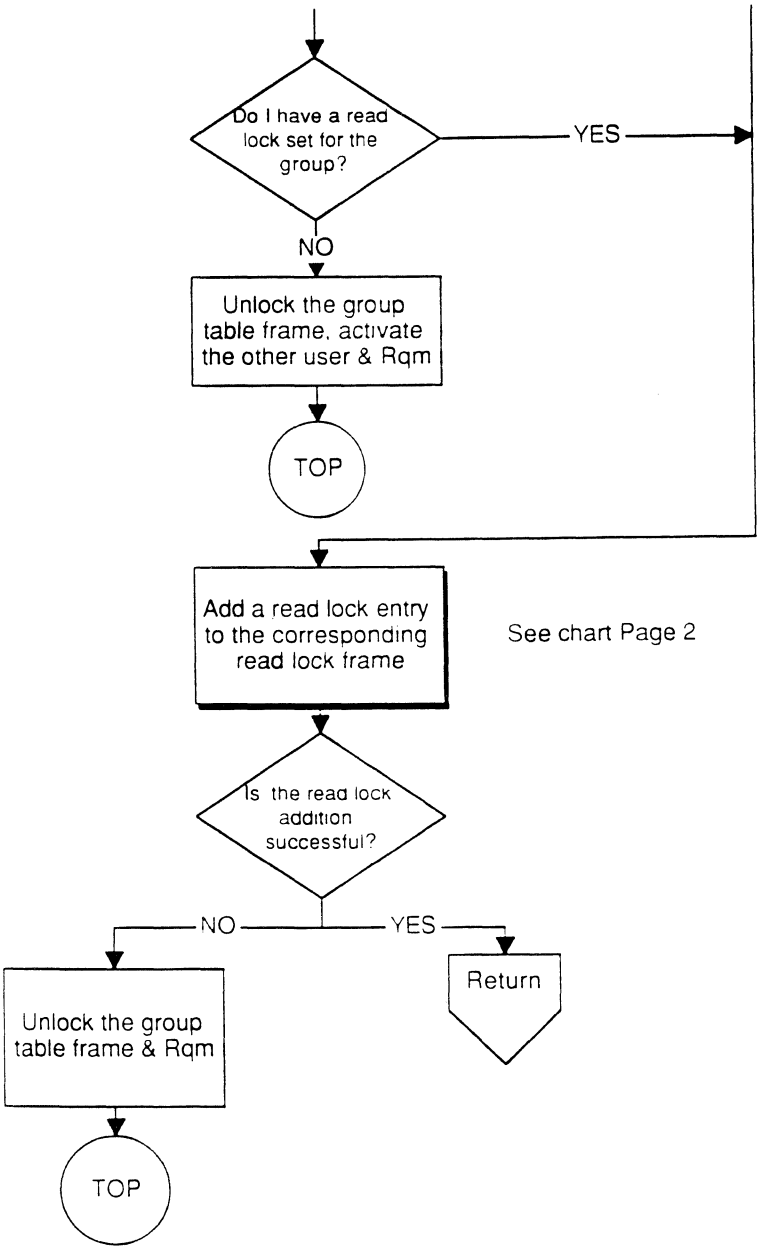
Thursday, September 5, 1991

Page 3

Set Read lock for the group



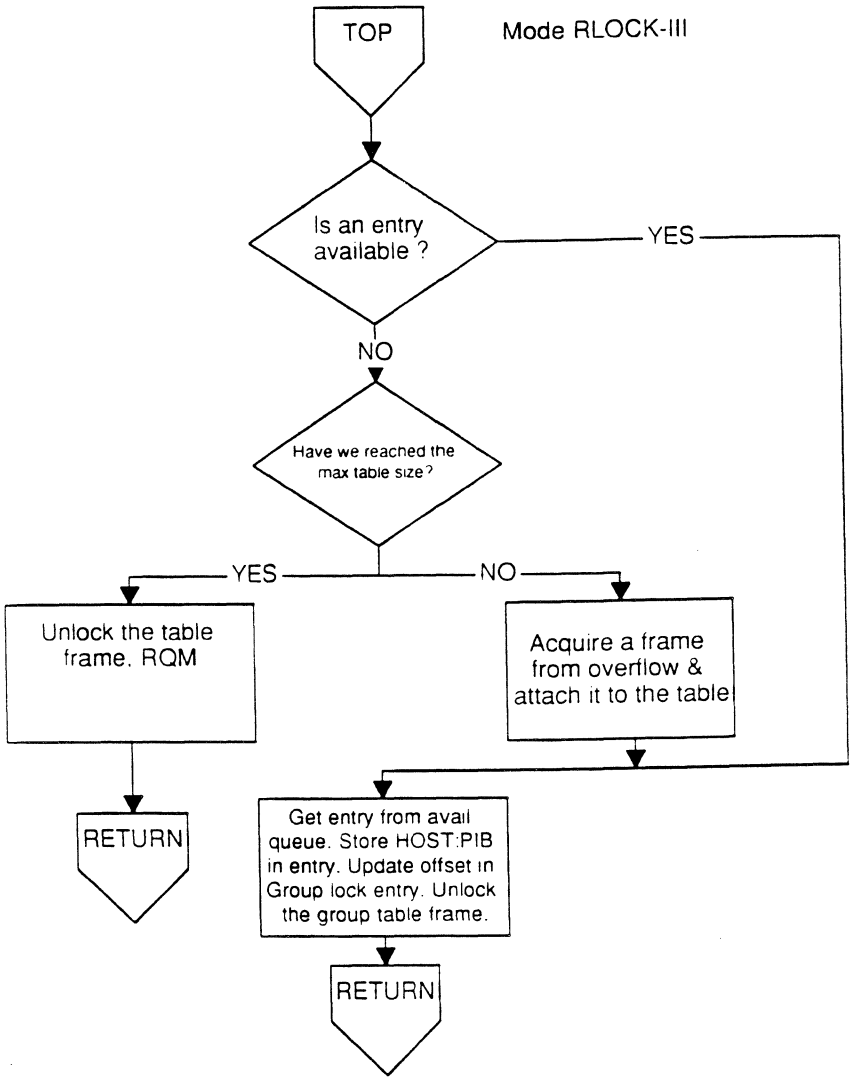

Group lock frame full ?
NO
YES
Unlock the table frame Rqm
TOP
Set scan byte in upper table & part of the hash value in the selected entry
Is group entry currently write locked?
NO
YES
Is it write locked by me ?
YES
NO


Do I have a read lock set for the group?
YES
NO
Unlock the group table frame, activate the other user & Rqm
TOP
Add a read lock entry to the corresponding read lock frame
See chart Page 2
Is the read lock addition successful?
NO
YES
Unlock the group table frame & Rqm
TOP
Return

Topic: Steps taken by the Basic READ instruction, specifically with regards to locking.

Thursday, September 5, 1991

Add a read lock entry to the corresponding read lock frame

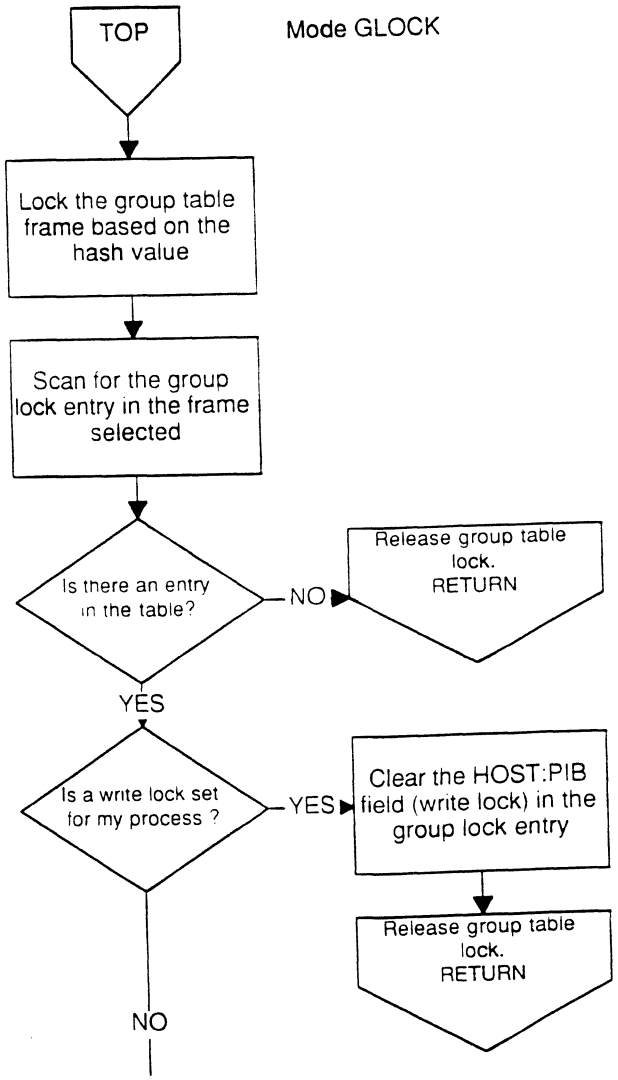


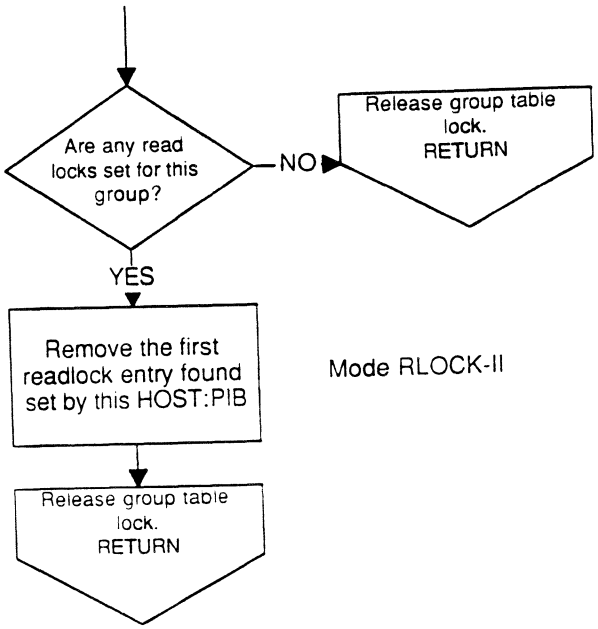
Topic: Steps taken by the Basic READ instruction, specifically with regards to locking.

Thursday, September 5, 1991

Page 4

Unlock write lock and first read lock





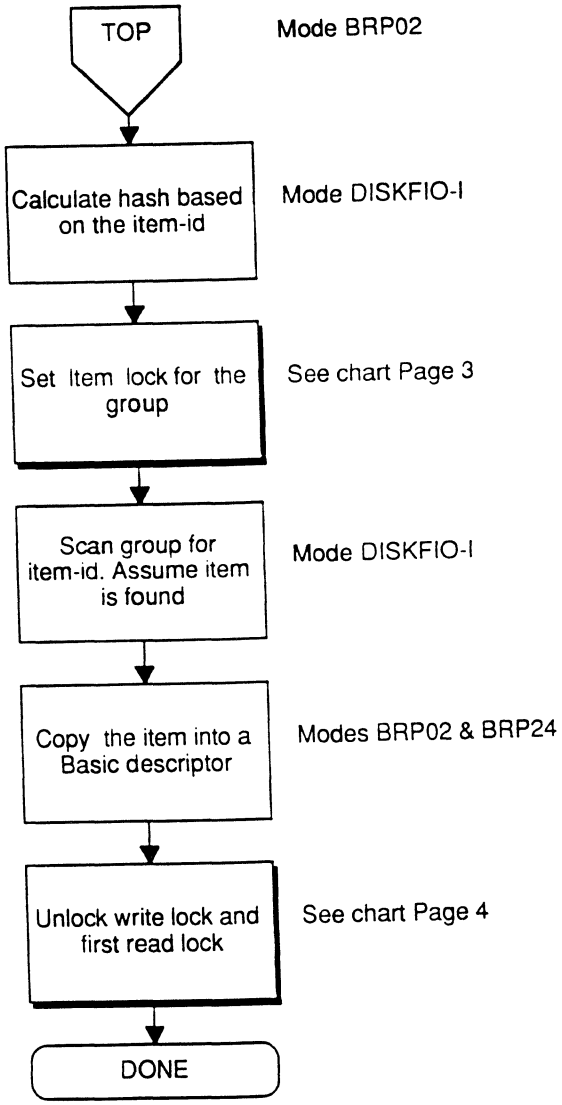
Mode RLOCK-II

Topic: Steps taken by the Basic READU instruction, specifically with regards to locking.

Thursday, September 5, 1991

Page 1

Basic readu

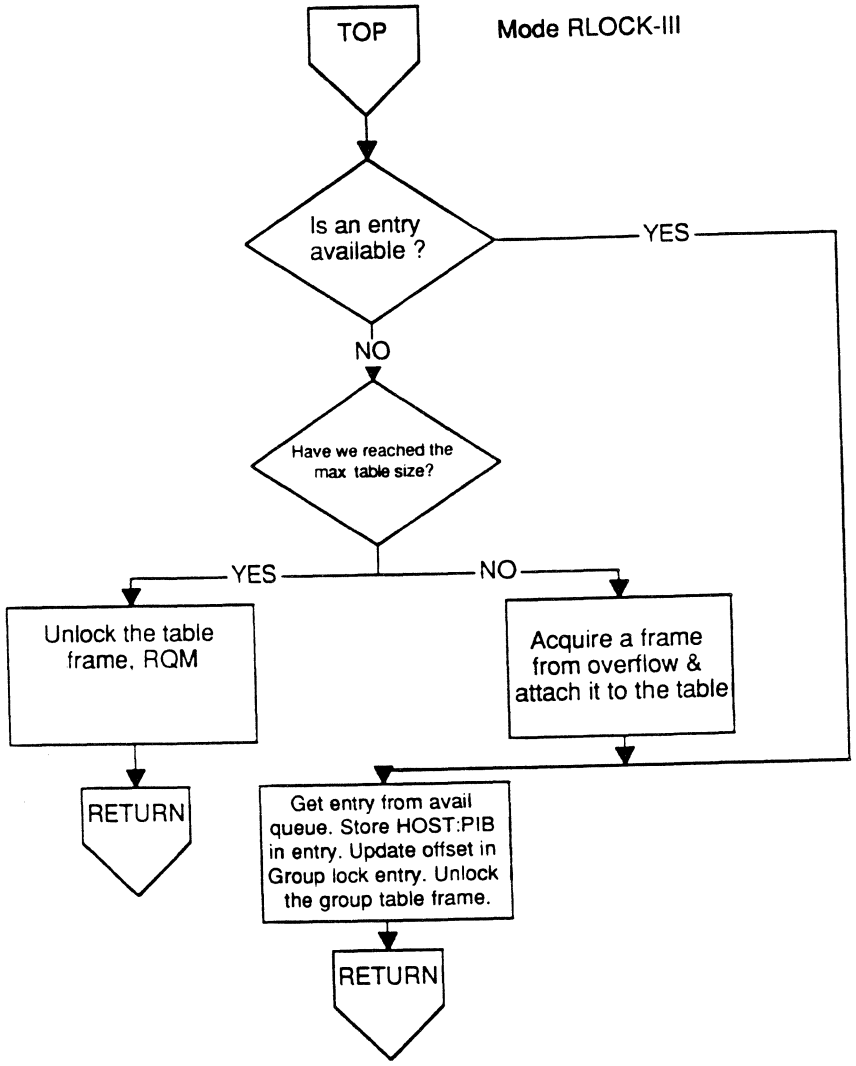


Topic: Steps taken by the Basic READU instruction, specifically with regards to locking.

Thursday, September 5, 1991

Page 2

Add a read lock entry to the corresponding read lock frame

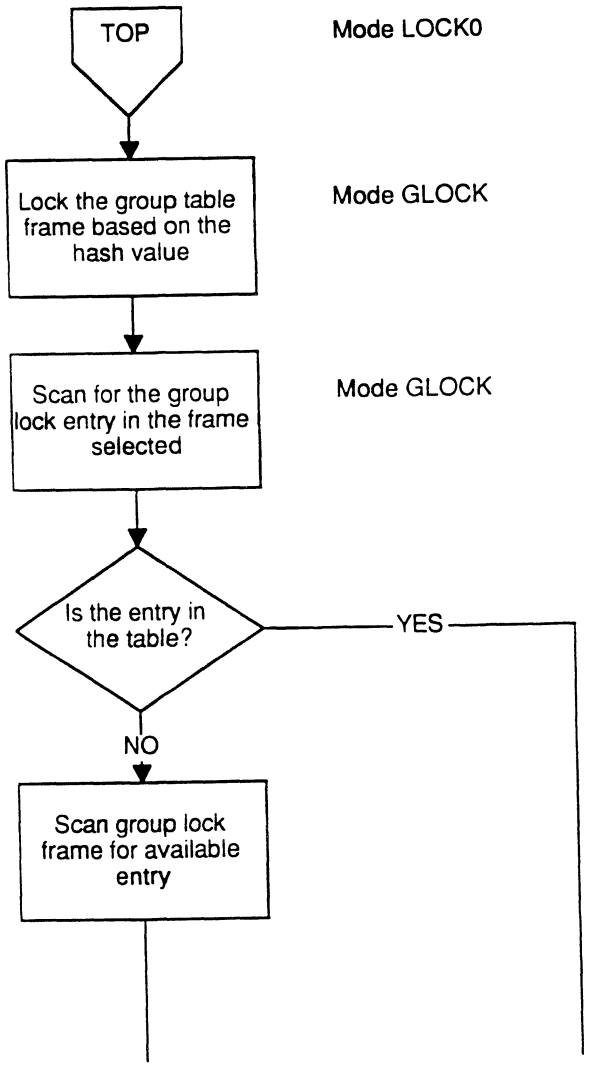


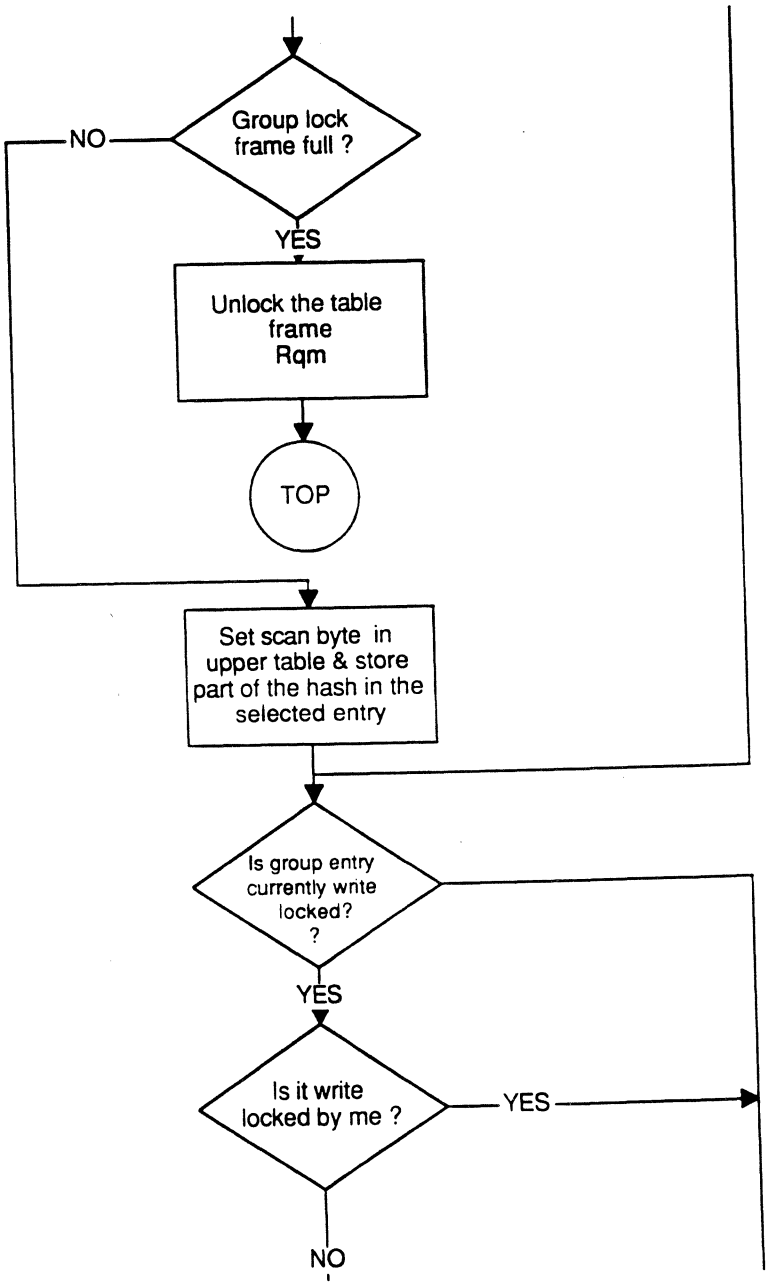
Topic: Steps taken by the Basic READU instruction, specifically with regards to locking.

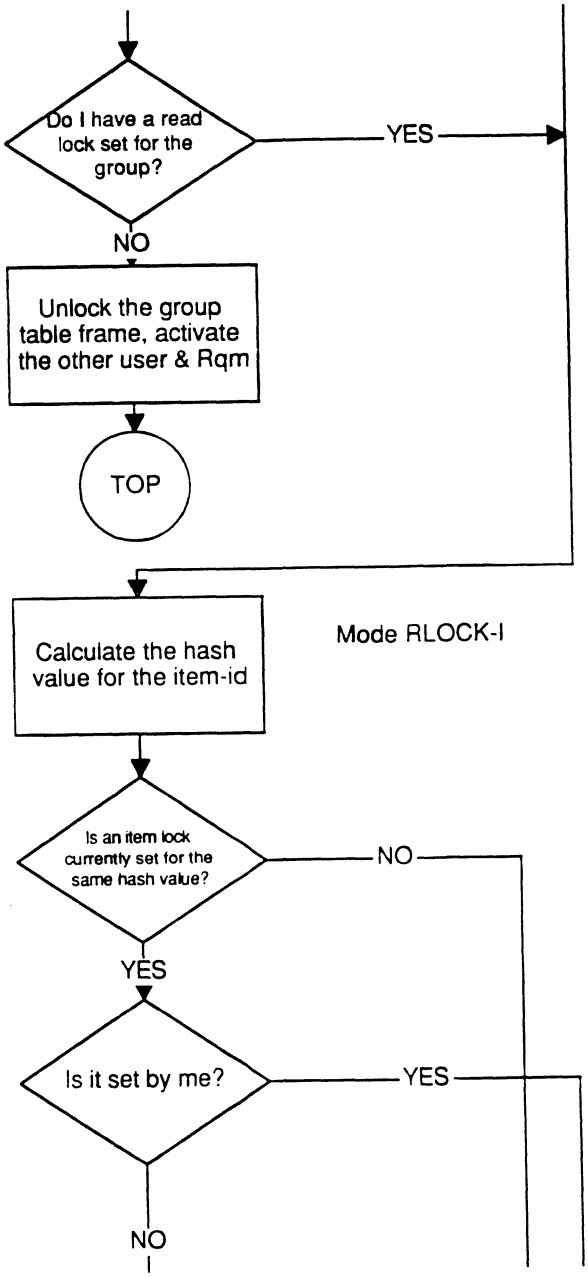
Thursday, September 5, 1991

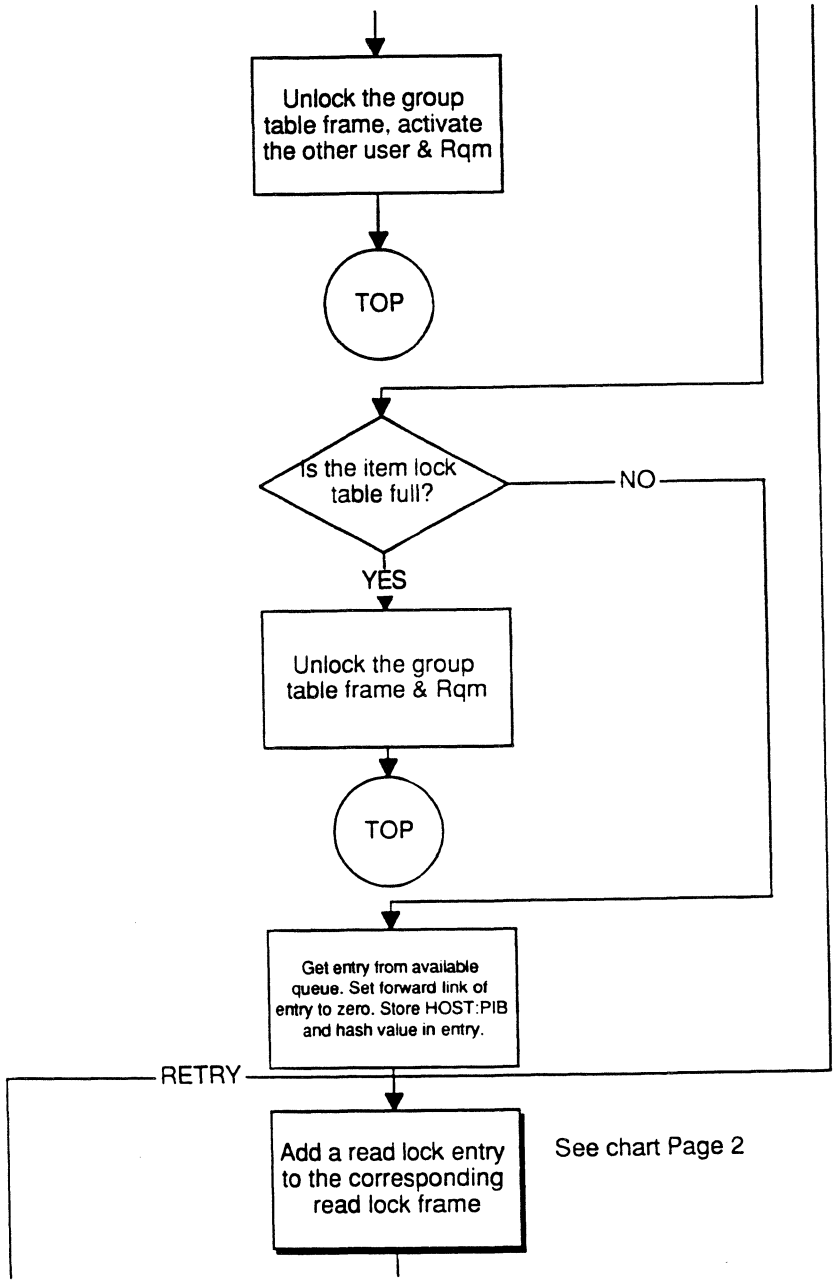
Page 3

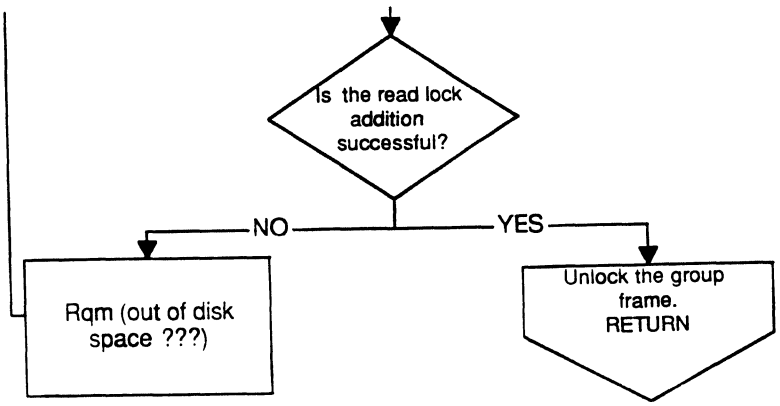
Set Item lock for the group




Group lock frame full ?
NO
YES
Unlock the table frame Rqm
TOP
Set scan byte in upper table & store part of the hash in the selected entry
Is group entry currently write locked? ?
YES
Is it write locked by me ?
YES
NO


Do I have a read lock set for the group?
YES
NO
Unlock the group table frame, activate the other user & Rqm
TOP
Calculate the hash value for the item-id
Mode RLOCK-I
Is an item lock currently set for the same hash value?
NO
YES
Is it set by me?
YES
NO


Unlock the group table frame, activate the other user & Rqm
TOP
Is the item lock table full?
NO
YES
Unlock the group table frame & Rqm
TOP
Get entry from available queue. Set forward link of entry to zero. Store HOST:PIB and hash value in entry.
RETRY
Add a read lock entry to the corresponding read lock frame
See chart Page 2

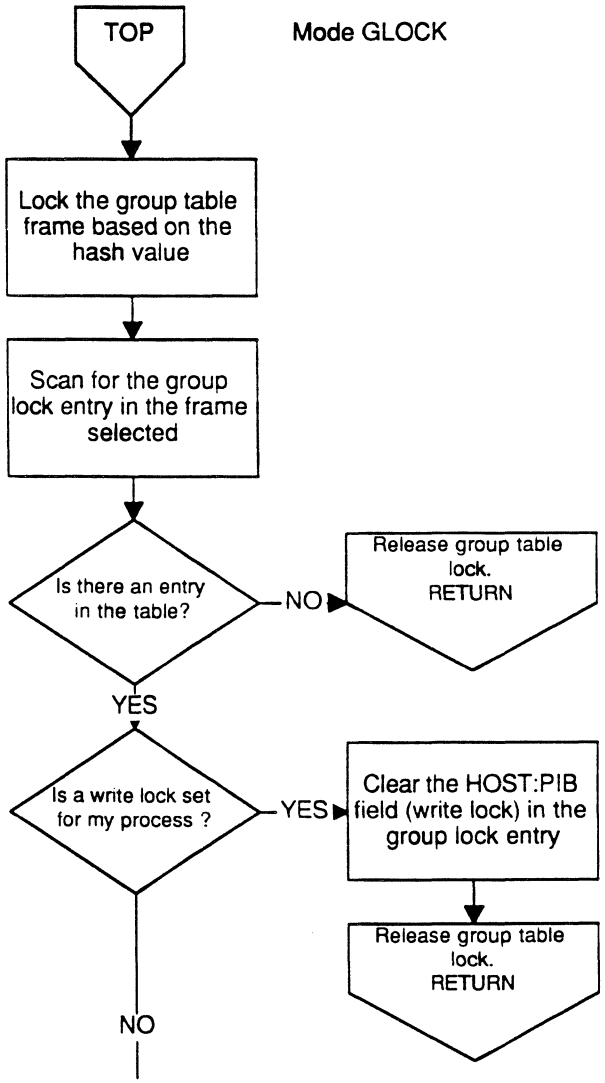

Is the read lock addition successful?
NO
YES
Rqm (out of disk space ???)
Unlock the group frame.
RETURN

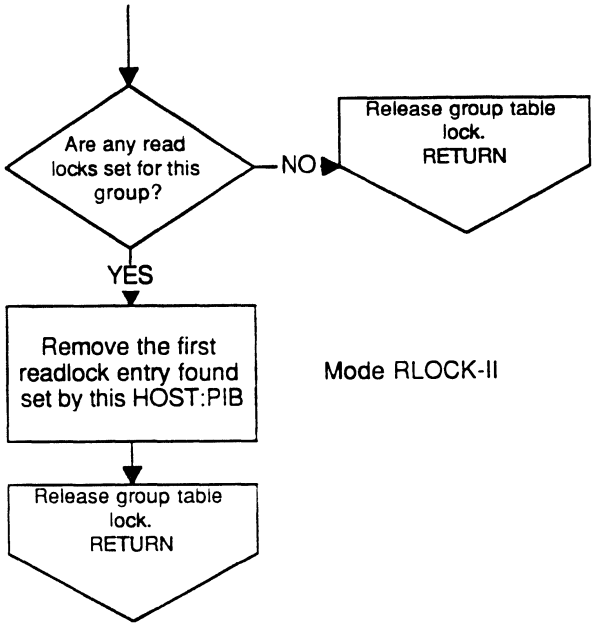
Topic: Steps taken by the Basic READU instruction, specifically with regards to locking.

Thursday, September 5, 1991

Page 4

Unlock write lock and first read lock





Mode RLOCK-II

Appendix B: Conversion format string

If the second argument of a ICONV/OCONV instruction is a literal string, the optimizer can parse it , initialize a data 'constant' data structure setting values and flags, and change the argument type (possibly the argument field also) to reflect the optimization.

The following is a suggested parser for some of the most used conversion codes.

Check the first character of the string for:

M Mask. Accept any of the following sub-codes:

R

L

D numeric and/or string format mask. Handle the same way as Basic FMT instruction (see 'Format' document).

T time format. Can be optionally followed by:

H twelve hour format;

S include seconds;

Cx some of the character conversion codes could be handled easily (ex: MCU, MCL).

D Date format. Handle the same way as in the Basic FMT instruction (see 'Format string parsing' document)

G Group extract. Is followed optionally by:

m a number indicating the count of fields to skip before doing the extract;

Is followed mandatorily by:

<sep>n a single character group separator (any non numeric except a minus sign or a system delimiter), followed by the count of fields to extract.

L Length. Not used frequently but easy to implement.

U User exits. The following is a list of user exits that could either be replaced by 'in line' code fairly easily or be converted to an already existing system function (see the 'System' document):

U5072 Return current system privilege level (0,1,2). Can be done in line by looking at PCB bits SYSPRIV1 and SYSPRIV2, or by using SYSTEM function 23;

U8072 Return a null string or a '1', depending on the status of PCB bit DOCCFLG;

U50BB Return process number and current account name (like WHO verb). We could use a combination of SYSTEM functions 19 and 26. The process number can also be fetched from the PIB directly.

U60BB Return the current ~~account name~~ process number. Use SYSTEM 26 instead.

U4117 Return the current PCB fid as a 6 byte ASCII hex string. Can be fetched directly from the PIB.

U5117 Return the PCB fid of line 0 (the LOGON PCB) as a 6 byte ASCII hex string. Can be fetched out of frame 127.

U3121 Return the system REV level, as a string. Can be fetched out of frame 127.

U0122 Return current number of PIBS. Can be fetched out of frame 127 or from the shared memory structure.

U5158 Return input from specified line number, if any is available. Equivalent to peripheral read. Call the code in test_inp.c and periph.c directly.

U6158 Write given output to specified line, then return input from that line , if any is available. Equivalent to peripheral write followed by peripheral read. Call the code in test_inp.c and periph.c directly.

U7158 Return the type-ahead count of a given line. Use the test_inp.c code directly.

U018D
U218D Increment the Basic break-off counter and set the INHIBIT bit in the PCB. Assuming the break counter is kept in a global variable this can be done in line.

U118D
U318D Decrement the Basic break-off counter and reset the INHIBIT bit in the PCB. Assuming the break counter is kept in a global variable this can be done in line.

Remarks:

Even though no statistics are available it is reasonable to assume that output conversions are used much more frequently that input conversions (most user exits work the same either way). Therefore we should first implement OCONV optimization of the above codes and perhaps only speed up a subset for ICONV.

Appendix C: Format string parsing rules

Introduction:

If the second argument of a FORMAT instruction is a literal string, the optimizer can parse it, initialize a data 'constant' data structure setting values and flags, and change the argument type (possibly the argument field also) to reflect the optimization.

The following is a suggested parser:

if the first character is a **D** then
parse for a date mask,
else
if the first character is an **M** ignore it;
if the next character is a **D** (**MD**), treat it the same as **R**;
accept any of the following justification codes:

R,L,V,U,T

R sets right justification, all other codes indicate left justification.
Parse the rest of the string for a numeric format mask.

After parsing for the date mask, if any data is left in the mask, check for the following justification codes ONLY:

R,L,V

If the first remaining character matches any of these three then:
- set the justification flag accordingly ;
- set another flag to indicate a string format mask is present;
- store a pointer to the remainder of the mask.

Any other code causes the rest of the mask to be ignored.

After parsing for a numeric format, if any data is left in the mask, then set a flag to indicate a string format mask is present and save a pointer to the

remaining string.

Parsing a date format string:

Accept any of the following options, setting appropriate flags in the data structure, in the sequence that they are listed here (note: none of these options are mandatory):

n number of digits to show the year in (between 0 and 4, the default is 4);

<sep>m a group extraction separator followed by a count. This separator is a single character which can be anything except a ';' or a system delimiter. The count specifies the number of fields to skip prior to the start of the date information. It needs to be converted to binary and saved in the data structure, along with the separator.

<sep> a single character, to become the separator between the day, month and year information. If present, the month shows as a 2 byte number. Otherwise it is printed in a 3 character abbreviation format.

OR exclusive condition: either <sep> or one of the next codes:

DJMQY respectively show only the Day of month, Julienne date, Month of year, Quarter or Year. Only one of these can be present.

Parsing a numeric format string

Past the justification code:

check for a single digit specifying the maximum number of decimal positions to output. If omitted, the default is zero;

a single digit precision. The default is zero when used inside a conversion, or equal to the precision value if called as a Basic

instruction.

Accept and set flags for any of the following codes, which appear in any order:

$	precede output with a monetary sign;
,	separate every 3 integer positions with a comma;
Z	suppress leading zeroes from the output;
N	do not output the sign indicator if negative;
CDME	one of four credit indicators.

If any of the above conditions is true (at least one of the digits or one of the flags settings) the format mask is considered to be of a numeric type. Note that this mask may be followed by an output format string.

Appendix D: SYSTEM functions that are good candidates for optimization

What follows is a list of SYSTEM functions that could be easily done in line:

1. Return the NUMBER 0 or 1, depending on the value of the PCB bit LPBIT.
2. Return the current value of PCB element OBSIZE, as a NUMBER.
3. Return the current value of PCB element PAGSIZE, as a NUMBER.
4. Return the current value, as a NUMBER, of
SCB elements FOOTCTR-LINCTR,
if FOOTCTR > PAGSIZE;
SCB elements PAGSIZE-LINCTR otherwise.
5. Return the current value of SCB element PAGNUM, as a NUMBER.
6 Return the current value of SCB element LINCTR, as a NUMBER.
7. Return the current terminal type value of PCB element TERMTYPE, as a STRING.
8. Return the current tape record length of SCB element TPRECL, as a NUMBER.
10. Return the current system type from frame 127, as a STRING.
16. Return cause of ABORT when in TRAP subroutine. If stored in global space, this can be done in line.
19. Return the current process number, from the PCB or the PIB, as a NUMBER.
21 Return the current execute level from the QCB, as a NUMBER.
22 Return the current spooler hold file number. If stored in global space, this can be done in line.
23 Return system privilege level (0,1,2), from the PCB settings of bits SYSPRIV1 and SYSPRIV2.
25 Return the item count from the currently active select list. If stored in global space, this can be done in line.

Remarks:

The functions for which a reference is made to 'global space' currently have the data stored in the Basic stack workspace frame, prior to HSBEG.

Appendix E: Named commons

Introduction:

At the contrary of a regular COMMON declaration, named COMMON is a runtime statement.

The variables declared in named common are stored in a permanent area of the user's workspace, for the duration of a LOGON session. The data can thus be shared among various independently called programs and even across TCL levels.

The total number of variables declared in a named common block must be the same in each of the programs that wish to use it. Each needs to execute a COMMON statement before any of the variables can be accessed. The precision level does NOT have to be identical for each of the programs.

Any type of variable can be declared in named common, with the exception of variably dimensioned arrays.

A maximum of 50 named common blocks can be active at any point in time. The total length of all the names may not exceed the size of one frame.

Executing the named COMMON statement (Opcode x'ED'):

Three entries have been pushed on the stack. From top to bottom:

- The number of columns in the common block (always 1 and ignored by virtual);
- The number of rows (the number of descriptors in the block);
- The name of the common block.

The opcode is followed by the offset to the target descriptor.

Two tables are used to manage the common blocks for each process:

The first one, hereafter referred to as TABLE1, is located in frame PCB+48. It contains a list of all the names of the currently active common blocks.

The second one, hereafter referred to as TABLE2, is located in frame PCB+49 and contains the primary descriptor for each of the blocks, in the same sequence as the names in TABLE1.

The runtime for opcode x'ED' scans TABLE1 for the name that it found on the stack.
If the name can not be located (new declaration):

the named is added into TABLE1;

a contiguous block of frames is obtained from overflow large enough to fit the count of descriptors (row count from stack);

the top frame is initialized as a type x'60' descriptor with a subtype of 5. The primary descriptor type is later changed to x'20', to uniquely identify a pointer to a named common block;

in the header portion of the top frame, the following information is saved:

the total descriptor size, in bytes;
the row and column counts for the block;
the precision value of the current program.

the target descriptor, now initialized as a type x'20' with a SR pointer to the top frame, is copied into TABLE2, at an offset relative to its position in TABLE1, each entry taking 10 bytes.

If the name is found in the list:

the position in TABLE1 is multiplied by 10 to obtain the offset to the primary descriptor in TABLE2;

via the SR in that descriptor, the header section of the top frame is examined to check if:

the number of elements in the block is the same as the count declared in the current program (number of rows). If not, the

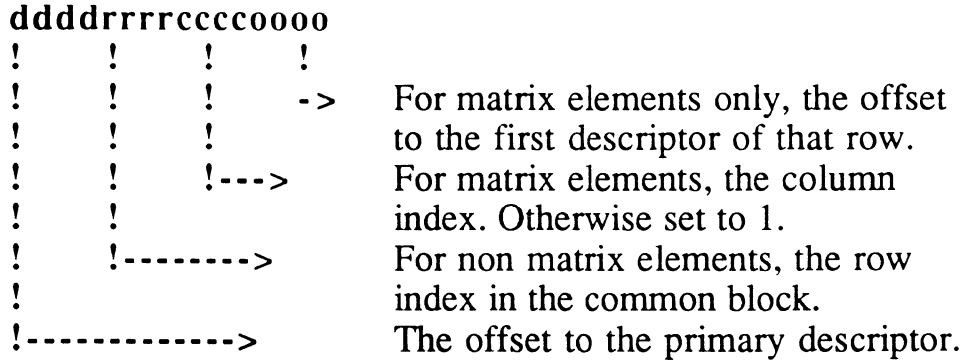
program abandons execution and drops into the Basic debugger;

the precision value in the block is the same as the one in the current program (SCALE#). If not, the current scale and precision values are temporarily restored to the values from the common block, after which each of its descriptors that has a type code x'01' (direct number) is converted (via MBDNATURAL) to a string. The string is stored either inside the descriptor if it fits (type x'02') or inside a frame obtained from overflow and initialized as a type x'60' descriptor.

The descriptor from TABLE2 is copied into the current program's descriptor space, at the offset found in the object code.

Storing a value in a named common variable (Opcode x'E8').

The opcode is followed by 8 bytes of vector/matrix information:



The top of the stack, which contains the value to be stored, is pushed up one entry, to satisfy the STORE interface, and a vector (type x'10') is pushed in its place, containing the address of the target descriptor inside the common block.
The address is computed based on the row/column/offset information from the object code.

The subtype of this vector entry is set to x'C0', as a flag to the STORE routine: when a string needs to be stored in a named common block, it either needs to fit inside the descriptor or it must be stored inside a type x'60' descriptor. It can NOT be stored in freespace.

Exit through Basic wrapup (see mode BRP11)

Each of the descriptors in the block that has a type code x'01' (direct number) is converted (via MBDNATURAL) to a string. The string is stored either inside the descriptor if it fits (type x'02') or inside a frame obtained from overflow and initialized as a type x'60' descriptor.

Implementation of named common in the optimized Basic environment:

A program can be developed to load the descriptors from the virtual modes into the C runtime data structures.

The major issue is when to write the descriptors back out to virtual keeping in mind that 'chained' or 'executed' programs may be not in 'Optimized Basic' format. Here are some possibilities:

update virtual every time a named common variable gets updated. This is equivalent to the current implementation;

update virtual only when encountering a CALL, EXECUTE or CHAIN instruction. For CALL and EXECUTE, the values also need to be reloaded when returning back to the program;

store the named common blocks in shared memory, and have the runtime check for previously loaded blocks there before going to virtual;

store the common block in heap space, but when invoking CALL, EXECUTE,... pass a pointer to a list of common variable blocks, as part of the argument list. This way the data can be shared and only on the final exit would we need to update virtual.

Note that if we want non optimized programs to access the named common blocks we must write the descriptors out to virtual in their current format, and also do the number to string conversions.

Appendix F: New Basic debugger full specifications

I. Objectives:

To provide 2 levels of debugging of New Basic programs. They are debugging for (1) programs with no optimization, (2) programs with optimizations.
Symbolic source level debugging is possible in the first level of debugging. As for the second level of debugging, it would not be source level debugging at all.

II. Assumptions:

Some assumptions are as followed:

1. A **map** of detailing relationship between source line number and pc.

2. A **symbol table**.

3. A flag (e.g. run_slow) to allow debugger to check for breakpoints and other conditions to drop into the debugger and some way to tell a break-key has been pressed (break_pressed).

4. A mechanism for the Basic runtime to call debugger directly when certain error conditions occurs and when a DEBUG statement is inserted in the source code (invoke_debugger).

5. A **programs** array that contains programs that the Basic runtime has loaded so far on the execute level. In the each programs array entry, there is information of filename and item that is loaded. A **call stack** has a programs_index that tells which program is at the stack entry. In the each programs entry, there is at least a pointer to the breakpoint and trace tables.

```
prgrams_struct {
        ...
        int             max_breaks;
        int             max_trace;
        break_struct    *breakpoints;
        trace_entry     *trace_table;
        char            source_file[52];
        char            source_item[52];
}
```

6. Some **standard file subroutines** to get at the source. This should exists for the regular Basic already.

e.g. *status = open_pfile(filename, descriptor)*
status = read_pitem(descriptor, item)

7. System debugger interface from Basic debugger.

8. A way to access ERRMSG file and also way to pass parameters to standard error processing routines.

9. It would be very helpful for all concerned if the object generator would output an essentially disassemble listing of the object file. For example:

line#	**pc>**	**opcode**	**mnemonic**
0001	70>	000F	Loadstring 9 A "abcdefghi"
0002	100>	000A	Store A B
0003	106>	000A	Store B C
0004	112>	000F	Loadstring 4 $0 KILL
0004	122>	006E	Call $0 1
0005	132>	00A4	PrintCRLF A
0006	136>	0074	Exit

10. A **terminal I/O interfaces**(includes interface to Ultimate Spooler) for read/write to the terminal so that we can have a focal point to access terminal and utilize paging in our output or even windowing in the future. A control block for each of our_fd so that

characteristics of the 'virtual' terminal can be kept (current lines, pager_on, spooler, etc). There will be at least 3 our_fd's. They are ULT+, sapphire debugger and basic debugger. On each new execute level, there are 3 more.

High-level:

uprintf(our_fd, "format string", arg...);

Low-level:

term_set(our_fd, FLAGS);
term_open(our_fd, path);
*term_reopen(our_fd, path); /*switch phy. device*/*
term_write(our_fd, buffer , len);
term_read(our_fd, buffer, len);
term_writeread(our_fd, buffer, wlen, rlen);

11. Some cleanup routines needs to de-allocate memory allocated by the debugger. (e.g. breakpoint and trace tables).

III. Symbolic source debugging:

As noted earlier, the symbolic source level debugging is possible in the first level of debugging. Since we can implement all of the commands and features of the current debugger, all commands and features are identical to the current debugger. See 6929-3 Ultimate Basic (p.g. 4-3) for more information.

A. Commands and features:

Some of the more important commands are discussed here:

1. Upward compatibility:

a. Breakpoint : Bvoc{&voc} or B$on
Set breakpoint on logical condition where
v is variable
o is logical operator <,>,=,#
c is condition to meet
n is line number when preceded by B$o
b. Call/Return breakpoint : C
c. Escape to system debugger : DE{BUG}
d. Single/multiple step execution : E{n}

e. Continue program execution at specific line : G{n}
f. Remove breakpoints : K{n} or K{{/}var}
g. Display specified source code current lines : L
h. Toggle output of Basic PRINT statement between terminal and printer : LP
i. Bypass breakpoints/steps before reentering debugger : Nn
j. OFF
k. Inhibit/enable output from the program : P
l. Printer-close output spooler : PC
m. Display GOSUB return stack : R
n. Toggle display of source code lines and line numbers : S
o. End program execution; if executed from PROC, return to PROC, return to PROC : STOP
p. Turn trace table on/off : T
q. Turn specified variable 'v' : T{/}v
r. Remove traces : U{n} or U{{/}v}
s. Display current program name and line number; verify object code : $, *, ?
t. Display value of variable or if dimensioned array, entire array with paging : /m
u. Display value of element in array : /m(x{,y})
v. Display value of element in dynamic array : /m<a{,v{,s}}>
w. Display entire symbol table : /*
x. Specify substring to display in subsequent variable display : [x,y]
y. Specify substring to be display in whole : [

2. New Commands to implement when time permits:

a. Provide a verbose mode so that a more verbose commands set for the above functions can be used. Like BREAK for B, TRACE for T, STEP for E and so on.
In addition, new features will only available on this verbose mode to avoid conflicting commands. It is not necessary for user to type the whole word(command or modifier) in most cases. User just have to type enough of the word to make it unique among the many commands that we provide. Briefly, they are:

BREAK set breakpoint by line, opcode, variable, pc

	conditionally or unconditionally; display breakpoints and turn breakpoints on/off.
CLEAR	Clear selectively different types of breakpoints; clear all breakpoints.
DISPLAY	Display value of a variable by name or by descriptor number.
END	leave debugger and exit program.
GO	resume the program at a different line number or just continue.
HELP	Get help on any topic or command.
IGNORE	ignore breakpoint; only display trace table information if any.
INFO	display current line and optionally include all call and return stack information.
LIST	list object in a disassemble form.
LOG	Log all output to a specified unix file.
MODIFY	Change value of a variable.
PROC	terminate basic program and return to PROC if any.
QUIT	same as END
SETBREAK	set size of the breakpoint to a larger size than number of used entries.
SETTRACE	set size of the trace table to a large size than number for used entries.
SHOW	various table at the end of the object.
SOURCE	Display source program at any line number; change source file; source on
STEP	step through program by # of lines or # of opcodes
SWITCH	switch debugging terminal to another port under control of ULT+.
TRACE	display values of on variables at breakpoints as well as turning trace on and off.
UNTRACE	untrace variable(s)
VERBOSE	turn this verbose mode on/off.

b. Repeat the last command for repeatable commands.

GO :	<CR> will automatically execute the 'GO' debugger command. (G command)
SOURCE :	<CR> will cause same number of lines to be

display as the last LIST command.(L command)
DISPLAY : <CR> will cause to display the next variable

c. Log all debugger output to specified Unix file : LOG unix_file

d. From a port in Basic debugger, we can switch the debugging terminal to another port : SWITCH p

e. Break on a line n:

BREAK [PC] <n> [IF voc [{AND / OR} voc]]
BREAK IF voc ...

Break on a line number or Break on a line number if certain condition(s)(e.g. A=10) is true. It can be used to break at pc (PC modifier is used).

f. Break on an opcode : BREAK OPCODE opcode-name

g. Break when a variable changes its value : BREAK VARIABLE p

h. A way to turn breakpoint selectively on/off and turn on/off all breakpoints : BREAK [<n>] { ON | OFF }
i. Getting into the Basic debugger from the sapphire debugger. To be defined.

3. Some of the more important Basic debugger features:

a. DEBUG statement in source code which allows a program execution be transferred to debugger if the program execution is invoked with option D.
b. Pressing break-key to drop program into debugger.
c. Certain errors causes program to drop into debugger.

[B5]	Incorrect number of subroutine parameters
[B12]	File has not been opened
[B17]	Array subscript out of range
[B18]	Attribute less than 1 is specified in READV or and attribute number less that -1 is specified in WRITEV statement.
[B22]	STORAGE parameter less that 10 or not a multiple of 10.

[B25]	A subroutine specified in a CALL statement was not found in the program file and was not cataloged or if cataloged, not found in the file specified in the MD.
[B26]	'UNLOCK C' attempted before LOCK. (does not issue this error any more)
[B27]	RETURN executed with no GOSUB
[B31]	Workspace underflow, register B. (FLZ; Reg=15 Abort @1066.103)
[B33]	Precision declared in subprogram 'C' is different from that declared in the mainline program.
[B36]	Arrays in calling program and subroutine must both be either fixed dimensions or both variable dimensions
[B37]	Variable dimensioned array element referenced before array was initialized by a DIM statement
[B41]	Lock number is greater that 47. (It now take the modulo of 47 of that number and use it as the lock number. e.g. LOCK 48 is equivalent to LOCK 1)
[B42-44]	About named COMMON block
[B45]	Program named in CALL statement is not a subroutine.
[B107]	LOOP statements is more that 50 levels deep.
[B209]	File is access protected.

d. It is unfortunate that we are going from a Ultimate object file to generate our object file because source line information is lost on the Ultimate object file for all included files. One would think that it is possible to improve the traditional compiler so that it put line number information in the object and includes source file information at the end of the object.

B. Debugger Data Structures:

In order to support commands and features of the debugger, Several data structures are needed to be kept in each execute level.

1. breakpoint table : breakpoint[]

There is actually a breakpoint table for each program in which the Basic runtime has ever loaded during a run. It can be part of the programs array we mentioned earlier. This is needed because breakpoints exists across subroutine calls. There is no practical limit on the number of breakpoints one can have in a program.

```
typedef struct {
        short       type;           /* types of breakpoint          */
        OPCODE      opcode;
        INT         line_number;
        INT         pc;             /* program counter              */
        CHAR        var[52];        /* variable name                */
        INT         descriptor_number;
        descr_struct des;           /* descriptor                   */
        cond_struct cond1;
        cond_struct cond2;
} breakpoint_struct;
```

type: not_used, break_on_line_number, break_on_pc, break_on_opcode, break_on_descriptor_mod, break_on_logical_cond, break_on_subr

opcode : New Basic opcode number.

line_number: hold a line number before it is converted to pc.

pc : program counter or something equivalent

var : for storing variable name for display purpose or it contains the subroutine itemname for **break_on_subr**.

descriptor_number : **break_on_descriptor_mod** : internal descriptor number of var

des : a copy of the descriptor used for comparing current value with.

if_cond : Used verbose mode : hold the logical operators (0 = unconditional 1 = AND or 2 = OR).

cond1/cond2: (see (6) below)

Note : type break_on_logical_cond is for Bvoc[&voc] in the traditional debugger.

2. RUN options : options[]

options[] : an array where we can check what options are turned on. For example: "RUN BP TEST (E" will cause options[OPT_E] to have value of 1. This is more for regular Basic runtime than debugger.

3. Internal debugger information : db

The debugger will be driven by this structure.

```
typedef struct {
        INT             verbose, source_on, break_on,bases, last_steps, debugger_steps,
                        call_return, entry_param, substring, substring_len, ignore_entry,
                        ignore_count, switched, tracing, logging;
        CHAR            entry_code;
        FILE            *log_stream, *istream, *ostream.
} debug_info_struct;
```

verbose: flag whether verbose is on.
verbose -> DEBUG> as prompt and accept new verbose set of commands.
not verbose -> * as prompt and the good old debugger is in effect.

break_on: flag whether we want to break on breakpoints

source_on: flag to whether to display next source line when the program drops into the debugger.

bases: flag whether to display variable values in hexadecimal(16),octal(8),binary(2) or decimal(0)

last_steps: how many steps last time

debugger_steps: current debugger steps as in the E command(< 0 for line stepping and > 0 for opcodes stepping).

entry_param: Used in conjunction with entry_code that tells which breakpoint send program into debugger

call_return: flag whether CALL/RETURN breakpoint is turned on

substring: the index of the string to start display

substring_len: the length from which the substring starts to display

ignore_entry: number of time entry to the debugger to be ignored (as in the N command)

ignore_count: just counting number of times debugger did ignore entry to the debugger

switched: 0 or contains the port + 1 that the debugger has been switched to.

tracing: flag whether tracing is on
logging: logging to a unix file. affects termio
entry_code: option D (D), error_cond(E), break_interrupt(I), break_point(B), call(C), return(R)
streams: streams for input/output and log file.

4. symbol table : symbol_table

```
typedef struct {
        char            name[50];
        short           length;
        short           common;
        int             address;
} symbol_table_struct;
```

(Question : How do we represent named COMMON?)

5. opcode information : opcode[opcode_number]

```
typedef struct {
        int             opcode_length;
        char            opcode_name[16];
} opcode_entry;
```

opcode_length : Fix-length opcode : the length of the opcode in bytes.
Variable length opcode: -63. The real length of the opcode is to be calculated from the opcode fixed part. For example :

[<---------- fix part --------->]
Loadstring length_of_string target string.....

opcode_name : a name given to an opcode.

6. logical expression : cond_struct

This struct is used to hold the conditions to be tested.

```
typedef struct {
        int             first_value;
        short           log_op;
        short           desc_flag;
        union {
        DESCRIPTOR      value;
        int             desc;
        }
} cond_struct;
```

first_value: a symbol_table index of the variable.
log_op: EQ, LT, GT, NE, LE, GE for =, <, >, #, <=, >= respectively. LE and GE are set in verbose mode only.
desc_flag: flag whether second value is a variable (desc) or a literal(value).
value/desc: is the value to which first value is compared to. It could be a string literal or a numeric literal defined by value or a symbol_table index of the variable.

7. Trace table : trace_table[]

There is actually a trace_table for each program that Basic runtime has even loaded during a run. Therefore, this can be part of the programs array element. There is no practical limit on the size of the table.

```
typedef struct {
        int             symbol^num;
        int             dim1, dim2;
} trace_entry;
```

symbol_num: index of the symbol table of the variable
dim1: row dimension
dim2: column dimension

8. Number of loaded symbols : loaded_symbols

Since it is not necessary to load symbol table for every program we run, we will use this variable to flag whether debugger needs to load symbol table before going further. This variable has to be updated by CALL/RETURN opcode or by a generalized subroutine that changes current program.
The value of -1 means symbol table need to be loaded. The value of 0 means program has no symbol table. A positive value will indicate that the symbol table for the current program is loaded and it has a symbol table.

C. Debugging an non-optimized program:

In this section, we will describe how the control is given to debugger and how the data structures are used to provide features and commands of the new Basic debugger for debugging an non-optimized program.

1. Control Flow:

Initially, the basic debugger can be entered via a break-key, an error conditions or an option D at the command line. The debugger can be re-entered via a breakpoint, CALL/RETURN breakpoint, error condition , a break-key or a DEBUG statement (with option D).

The following is to appear within the Basic runtime opcode decode loop. In addition, for CALL/RETURN breakpoint to work, those two opcodes should look at db.call_return and set run_slow, options[OPT_D] and db.entry_code appropriately.

```
if ( run_slow ) {                                   /* see note (a)  */
        if ( break_pressed ) {                      /* see note (b)  */
                break_pressed = 0;
                options[OPT_D] = 1;
                db.entry_code = 'I';                /* Interrupt          */
        }
                                                    /* see note (c) */
        if ( db.break_on && programs[cp].breakpoints && !options[OPT_D] ) {
                for ( i=1;i<=programs[cp].max_break;i++){
                        switch ( programs[cp].breakpoint[ i ].type ){
```

```
                break_on_line_number:
                break_on_pc:
                        if ( ..breakpoint[i].pc==Codeptr ){
                                if ( ..breakpoint[i].if_conf ){
                                        checkcond();
                                        ...
                                } else {
                                        db.entry_param = i;
                                        options[OPT_D] = 1;
                                }
                        }
                        break;
                break_on_opcode:
                        if      ( ..breakpoint[ i ]. opcode = Codeptr->opcode ){/* note
                                (d)     */
                                < see above >
                break_on_descriptor_mod:
                        cur=    ref( ..breakpoint[i].descriptor_number);
                        if ( cur == ..breakpoint[i].des ) ||
                                ( they are string and they are the same ){
                                /* no modification */
                        } else {
                                db.entry_param = i;
                                options[OPT_D] = 1;
                                ..breakpoint[i].des <- cur;
                        }
                        break;
                break_on_logical_condition:
                        p1=ref(..breakpoint.cond1.first_value);
                        ...obtain p2 from desc/value...
                        cond1_met = 0;
                        if ( evaluate( p1,p2 ) ){
                                cond1_met=1;
                        }

                        if (..breakpoint[i].cond2.first_value){
                                < evaluate cond2 >
                        } else
                                cond2_met = 1;

                        if ( cond1_met && cond2_met ){
                                db.entry_param = i;
                                options[OPT_D] = 1;
                        }
                        break;
                }/* end of switch */
        } /* end of for */
        if ( options[OPT_D] )
                db.entry_code = 'B';
}
```

```
        if ( db.debugger_steps > 0 && !options[OPT_D] ){
                /*      step by opcode                  see note (f)            */
                db.debugger_steps--;
                if ( !db.debugger_steps ){
                        options[OPT_D]=1;
                        entry_code='E';
                }
        }
        if ( db.debugger_steps < 0 && !options[OPT_D] ){
                /* step by line                                                 */
                <determine whether this is EOL>
                if ( EOL )
                        db.debugger_steps++;
                }
                if ( !db.debugger_steps ){
                        options[OPT_D]=1;
                        entry_code='E'
                }
        }

        if ( options[OPT_D] && < !not_entering_debug > )
                invoke_debugger;                        /* see note (g) */

        if ( run_slow && db.debugger_steps == 0 &&
                db.breakpoints == 0 )
                run_slow = 0;
}

<dcd begins...>
```

Notes:

(1) run_slow is a flag (does not have to be boolean) that tells the Basic runtime to check conditions that may suspend Basic programs. In our case, we check for break-key, breakpoints, debug steps and options[OPT_D]. As you may have suspected, run_slow needed to be set before options[OPT_D] is checked(Initially, if options[OPT_D], sets run_slow).

(2) break_pressed or other equivalent flag that tells Basic a break-key was pressed. It has to be set by a interrupt catching routine (e.g. checkpoint).

(3) cp is the index into the programs array for which the Basic is running.

(4) Just to illustrate one way to access current opcode. In reality, the opcode may be accessed in entirely

different way. The Codeptr is a prototype opcode pointer.
(5) If first_value is not a symbol table index, it must have been a B$on source line breakpoint. pc and Codeptr are used interchangeably because at this point it is not known how we access opcode in object file.
(6) As noted earlier, debugger_steps is actually two mutually exclusive counters. When it is > 0, it is a opcode counter. When it is < 0, it is a source line counter. In both case, it is time to enter debugger once the counter reaches zero.
(7) a macro that do all the necessary steps to invoke debugger. It includes 1) update trace_table in the db structure 2) update the last_steps in the db structure after debugger has return.

2. Debugger:

Once Basic runtime decided to drop into the debugger, the debugger will take control until a GO command is initiated. While in the debugger, user can display variable information, set breakpoint for next entry into the debugger and turn on/off debugger options. Here we will give detail description of what debugger has to do to accomplish features and commands that has been described in earlier sections.

Call Parameter: The Basic debugger is called with db structure as parameter because each Basic runtime has its own debug environment. This is apparent when you execute a Basic program. For example : debugger(db) where db is defined inside Basic runtime. Another way will be to allocate db structure each execute level and save/restore db pointer every time Basic enter/exit an execute level. Case in point, db is a small structure.

Debugger skeleton: When called, it has to initialize itself. It returns if it should ignore this call. Then it takes commands from terminal and execute them. It relinquishes control when the command is GO. Note : we only depict how the verbose mode is going to look here. A parallel structure will be in for the non-

verbose(ultimate compatible) mode.

```
init();                                 /* note (a)                    */
if ( db.source_on )
        display_stats();                /* note (b)                    */
else
        display_program_info();
display_trace_table();
db.ignore_count++;
if ( db.ignore_count <= db.ignore_entry )
        return;
db.ignore_count = 0;

while ( options[OPT_D] ){
        prompt("DEBUG>");
        term_read(command_line,command_len);
        check_break;...
        if ( !command_len and repeatable ){/* note (c)        */
                command_len = last_command_len;
                command_line <- last_command_line;
                repeatable = 0;
        }
        last_command_len = command_len;
        last_command_line <- command_line;
        if ( command_len ){
                /* note (d)                                             */
                CtUsed = 0;
                Nextc();
                /* note (e)                                             */
                perform_Vcommand( NextToken(word) );
        }
}
```

Notes:

(a) read symbol table if needed, initialize last_command if db.last_steps is set, locate source file, etc.

(b) display 1 source line if possible or display 1 opcode information based on optimization levels.

(c) repeatable is set by preform_Vcommand to tell debugger that this is a repeatable command.

(d) set CtUsed to the beginning of the command_line for Nextc(). The Nextc() will return the first character in c and the type of

character in Ctype.
(e) One of the command that perform_Vcommand does is the GO command. The GO command will reset options[OPT_D] to cause debugger to return to Basic. It also handles errors and tokens that are not commands.

D. Debugging an optimized program:

An optimized program has object that does not necessary have one to one relationship with source code and basic variables attain values not necessary in the order that appeared in the source code. These facts make 1) stepping through the source program impossible 2) displaying values of a variable hazardous.

There may be 4 ways we can do this.

a. No debugging for an optimized program or only provides low level (opcode level) debugging.
Problem with this approach is that nobody really want to debug at such a low level.

b. Only let basic program drops into debugger at the end of a basic block. At which point, we have good idea where the program is at and values of variables are known.
Problem with this approach is that program may well have runtime error within a basic block. At which point debugger lacks the knowledge to point out exactly which source line is the cause of the error and Basic runtime would not let the program run until the end of the basic block before it lets program drops into debugger anyway. Because of this problem, it will not work as is. It can probably be part of (3) below. That means (3) can allow user to set break point at the end of a basic block.

c. Embedded in the Basic object the dags of all basic blocks annotated with information about which variables hold the value corresponding to a node in the dags at what times in both the source and optimized program. With this information, a) we can infer from the information which line of the source

code caused the runtime error and b) gives values of variables in most cases while program is in debugger (due to error or otherwise).
Problem with this approach is that the dags of all basic blocks does not seem to be easy to generate and the size of this information is probably as big as the object.

d. A map detailing what source line(s) corresponding to a pc is kept at the end of the object. With this information, a) we can probably tell which line caused an runtime error if we pick the first source line that is involved. b) However, we cannot give the user with any degree of certainty that a variable having a certain value at any point of a source program (including where the program apparently has an runtime error). We can tell them however in the optimized world what the values of variables are now.

The option 4 is agreed upon and adopted. User should be aware of all this happen because it is an optimized program. As mentioned above, we have very limited capability to debug a program with this information. User may be forced to do opcode level debugging as described in section 4.0.

IV. Non-source level debugging:

Debugging without source is needed because there is no source, the line number information is not in object or the object is optimized to include global code movement. User may get a copy of the object listing and step through the program if desired.
Since all correspondence to the original source code is too convoluted to try to sort out, the only recourse for users to debug their programs will be to obtain disassemble listings of their programs from the BASIC verb or from the debugger LIST command. Users will be restricted to the opcode level debugging. That means if users use command that refers to a line number, they will be warned that this is an optimized program and the command will be rejected.

Additionally, the debugger can provide a new breakpoint called opcode breakpoint (BREAK OPCODE opcode). This is useful for user to get some idea what the program is doing at certain part of the program. For

example, user can stop at a READ opcode (i.e. BREAK OPCODE READ) and an ICONV (i.e. BREAK OPCODE ICONV) opcode and display values of variables before and after the opcode operation.

V. Major components of the debugger:

There are 8 major components in the Basic Debugger. They are 1) initialization 2) code that exists in the dcd code to determine whether to drop into the debugger; 3) many Basic opcode calls to debugger (or debugger utility routine like pc_to_line) due to runtime errors; 4) parser for standard Ultimate commands; 5) command processor or perhaps subroutines that execute the Ultimate debugger commands; 6) token parser for the new verbose mode of the debugger; 7) the command processor or perhaps subroutines that execute new verbose set of commands; 8) utilities (e.g. pc_to_line, line_to_pc).
In the first phase of the implementation, the verbose mode may not be implemented. That means only the 6 components needed to be implemented by Apr 1992.

A note for implementation, the debugger should have a focal point to do terminal I/O so that LOG and SWITCH commands will not cause too much duplicated code.

A. Initialization:

Initialization of all debugger data structures should be done every time the Basic runtime is invoked. In addition, the debugger has to do the following every time it is called:

```
init()
{
        SAVE_IO_ENV;                    /* note (a)                    */
        system_mode = In_debugger; /* note (b)                    */
        if ( loaded_symbol = -1 )
                read_symbol_table();
        program_source();               /* note (c)                    */
        db.debugger_steps = 0;
        last_command_len = 0;
        if ( db.last_steps ){           /* note (d)                    */
                if ( db.verbose ){
                        repeatable = 1; /* <CR> => step  cmd           */
                        last_command_line <- "STEP";
```

```
                last_command_len = 4;
        }
        else
                db.debugger_steps = db.last_steps;
    }
}
```

Notes:

a. This is needed only if we have a terminal I/O environment that is shared by debugger and regular Basic. The debugger should not affect any current terminal I/O environment variable. (e.g. paging, redirecting Output to spooler and so on)

b. system_mode or something equivalent that tells the rest of the system that we are running with the Basic debugger. At the least, this affects Basic runtime error handling. See the section 5.3.

c. get the source file names and 'open' the file.

d. last_steps is made equal to the debugger_steps last time. As you may recall that debugger_steps is actually two mutually exclusive counts in one. One is for opcode stepping and the other is for lines stepping.

B. DCD interface between Basic and the Debugger:

This interface has been described in detail in section 3.3.1. However, the data structures like descriptor space and structure, call_stack, programs and data elements like run_slow, options and break_pressed need to be finalized before one can proceed to implement this piece of code.

C. Basic runtime error interface:

All Basic opcodes should go to a common routine for error handling. We will call this **basic_error(error_number)**. The reasons for this are:

a. The same Basic runtime error requires different treatments based on run option(s). For example, when the option E is used to run the program, the program will go into the

debugger for any runtime error.)
b. Some errors will cause the program to go into debugger. All errors mentioned under Basic debugger features in section 3.1 are this category.
c. Some errors are reported and 'ignored' by Basic. For example, an unassigned variable is used as zero.
d. Based upon whether the program is in debugger already, an runtime error needs different treatments. Any error that normally causes program to abort will go into the debugger if the program is running with the debugger.With this common error handling routine, we can handle all the above mentioned situations in one place and call the debugger if situation warrants.

Food for thought: Situation arises when a variable is used before it is assigned. We gives an error pointing out the error and uses an zero for its value. The program keeps running. But the variable in this case does not get assigned a value of zero. The variable is still unassigned. What is the mechanism to tell opcode to use this zero value not the unassigned variable for its operation?

D. Ultimate debugger commands parser:

Most of the Ultimate debugger commands are single character commands and there is no space(s) between parameter(s). This setup does not lean itself to a tokenized parser. Therefore, this parser is just going to look at the first character and call the designated command subroutines. Any further parsing is deferred to the subroutines that execute the command. Obviously, it has to report error if the first character does not matches any command.

E. Ultimate commands processor:

It is made up of many subroutines that is going to parse the rest of the command line and execute the command or report an error. These subroutines are as followed :

Commands	Subroutines
$,*,?	U_program_info()

```
/ :             U_display_var()
[ :             U_substring()
B :             U_breakpoint()
BYE:
C :             U_callreturn()
D :             U_display_table()
DE:             U_system_debugger()
E :             U_set_step()
END:
G :             U_go()
H :             U_help()
HX:
K :             U_clear()
L :             U_line_number()
LP:             U_output()

N :             U_set_ignore()
O :             U_display_options()
OFF:
P/PC:           U_printer()
R :             U_display_gosub()
S :             U_source()
STOP:           U_return_to_proc()
T :             U_trace()
U :             U_untrace()
V :             U_program_info()
Z :             U_change_source()
```

F. Parser for verbose set of commands:

Verbose commands:

keys:	{ }	: have to choose one
	[]	: optional
	<>	: user supplied argument
	v	: variable
	n	: number

```
BREAK {      [ PC ] <n> | OPCODE <opcode> }
       [IF <logical_cond> [{ AND | OR } <logical_cond>]]
BREAK [ { VARIABLE <v> [ {,<v>}.. ] | SUBR <subroutine>
       | [ <n> ] { ON | OFF } } ]
```

```
CLEAR [ { <n> | LINE | PC | OPCODE | VARIABLE | SUBR} ]

DISPLAY { ALL | <v> | <n> }

END

GO [ <n> ] [ PC ]

HELP [ { COMMANDS | <command> } ]

IGNORE <n>

INFO [ ALL ]

LIST <pc1> - <pc2>

LOG { ON | OFF | TO <filename> }

MODIFY <v>

PROC

QUIT

SETBREAK <n>

SETTRACE <n>

SHOW { SYMBOL | MAP | ? }

SOURCE {    [ <n> [ FOR <n> ] ] |
            OPEN <filename> <itemname> |
            ON | OFF }

STEP <n> [ OPCODES ]

SWITCH [ <port-number> ]

TRACE VARIABLES <v>[ {,<v>}.. ]

TRACE [ { ON | OFF } ]

UNTRACE [ { <n> | VARIABLES <v> [,{<v>}..] } ]

VERBOSE
```

It is a token parser. It consists of Nextc() and NextToken(type). The

NextToken(type) returns a token from the words matrix. The Nextc returns character types (Ctypes : white space, letters, digits, period, prefix, single, illegal, eol) to NextToken and also the character itself (c).

The words matrix has an entry for each command and modifier. In each entry, it includes the name, number of characters make the name unique and the help file index.

token#	name		unique	help index
t_NE or 1	"#	"	1	0
t_LEFTP	"(	"	1	0
t_RIGHTP	")	"	1	0
t_COMMA	",	"	1	0
t_LT	"<	"	2**	0
t_EQ	"=	"	2**	0
t_GT	">	"	2**	0
t_GE	">=	"	2	0
t_LE or 8	"<=	"	2	0
..	"ALL	"	2	0
	"AND	"	2	0
	"BREAK	"	1	1
	"CLEAR	"	2	2
	"COMMANDS	"	2	0
	"DISPLAY	"	1	3
	"END	"	3*	4
	"FOR	"	1	0
	"GO	"	1	5
	"HELP	"	1	6
	"IF	"	2	0
	"IGNORE	"	2	7
	"INFO	"	2	8
	"LIST	"	3	9
	"LINE	"	3	0
	"LOG	"	2	10
	"MAP	"	2	11
	"MODIFY	"	2	12
	"OFF	"	3*	13
	"ON	"	2	0
	"OPCODES	"	3	0
	"OPEN	"	3	0
	"OR	"	2	0
	"PC	"	1	0
	"PROC	"	2	14
	"QUIT	"	4*	4
	"SETBREAK	"	4	15
	"SETTRACE	"	4	16

	"SHOW	"	2	17
	"SOURCE	"	2	18
	"STEP	"	2	19
	"SUBR	"	2	0
	"SWITCH	"	2	20
	"SYMBOL	"	2	0
	"TRACE	"	2	21
	"TO	"	2	0
	"VARIABLES	"	2	0
	"UNTRACE	"	1	22
t_VERBOSE or 45	"VERBOSE	"	2	23
t_number	---			
t_variable	---			
t_string	---			
t_opcode	---			
t_eoi	---			

**: Needs special handling because they can be part of other token. (prefix Ctype for ">" and "<")

*: Make sure user means it.

Hint: enum tokens { t_NE=1, ... }
char dlms[] = "#(),<=>";

For example : BREAK 100 IF B = 10

The NextToken would return in successive calls the following: t_BREAK, t_number, t_IF, t_variable, t_EQ, t_number and t_eoi. In case of t_number, the break subroutine has to convert the 100 number in LexBuf to numeric line number and the 10 to a scaled number. The t_eoi is returned when NextToken reaches the end of input. In case of t_variable, another variable has the symbol table index (call it symbol_index).

```
NextToken( type = { word, symbol, opcode } ){

#DEFINE AppendC = {
        LexBuf[LexLen] = c;
        LexLen++;
        if ( c != '\n' )
                Nextc;
}
```

```
LexLen = 0;
WHILE ( Ctype == white_spaces )
        Nextc();
if ( Ctype == eol )
        return ( t_eoi );
CASE Ctype of {
        letter : {
                        Do {
                                if ( $ALPHA(c) )
                                        c = c & %xdf; /* upshift */
                                AppendC;
                        }
                        until ( Ctype != letter && Ctype != digit
                                                && Ctype != period );
                        CASE type of {
                        word :  token = Lookup_wordmatrix();
                        symbol :        token = t_variable;
                                        symbol_index = Lookup_symbol();
                        opcode :        token = t_opcode;
                                        opcode_index = Lookup_opcode();
                        }
                        return( token );
                }
        digit,
        period :        {       While ( Ctype == digit )
                                        AppendC;                /* integer part */
                                IF ( Ctype == period ){
                                        {       AppendC;
                                                WHILE ( Ctype == digit )
                                                        AppendC;
                                        }

                                return ( t_number );
                        }
        quote : {       DO AppendC;
                                until ( c == LexBuf[0] || Ctype == eol );
                                if ( Ctype == eol ){
                                        <error>
                                } else
                                        Nextc();
                                return( t_string );
                        }
        prefix :        {       AppendC;
                                LexBuf[LexLen] = c;
                                LexLen++;
                                if ( !(token == Lookup_wordmatrix() ){
                                        /* single delimiter */
                                        for ( i = 0; dlms[i] != '\n'; i++){
                                                if ( dlms[i] == LexBuf[0] )
                                                        break;
                                        }
```

```
                                        token = i + 1;
                                } else
                                        Nextc();
                                return( token );
        single : {
                                for .. <see above>
                                Nextc();
                                return( token );
                        }
        default :       return( 0 );
        }
}
```

G. Verbose commands processor:

It is made up of command subroutines. Each command subroutine is going to structure around the command syntax. It would call the NextToken every time it needs one. If it finds the token returned is out of place or illegal, it would report an error and returns. Otherwise, it would process the entire input and execute the command.

For example :

```
V_clear(){

/* CLEAR [ { <n> | LINE | VARIABLE | OPCODE | PC | SUBR} ] */

        token = NextToken(word);
        switch token {
        t_eoi:  for ( i=0;i<=programs[cp].max_break,i++)
                                clean_break(programs[cp].breakpoint[i]);
                        break;
        t_number:       number = atoi( LexBuf );
                        clean_break(programs[cp].breakpoint[number]);
        t_LINE:         for ( ... )
                          if ( programs[cp].breakpoint[i].type ==
                                break_on_line_number )
                                clean_break(programs[cp].breakpoint[i]);
        t_VAR:          for ( ... )
                                  if    (    programs[cp].breakpoints[i].type    ==
                                clean_break(programs[cp].breakpoint[i]);
```

```
                break;
t_OPCODE: ...
t_PC:  ...
t_SUBR:        ...
default: <error>
}

if ( NextToken(word) != t_eoi )
        <error>
}
```

There are 22 commands in the verbose set of commands. They are as followed :

BREAK	Set breakpoint by line, opcode, variable, pc conditionally or unconditionally; display breakpoints and turn breakpoints on/off. Additionally, the debugger can break inside a subroutine every time a subroutine is called.
CLEAR	Clear selectively different types of breakpoints; clear all breakpoints.
DISPLAY	Display value of a variable by name or by descriptor number.
END	Leave debugger and exit program.
GO	Resume the program at a different line number or just continue.
HELP	Get help on any topic or command.
IGNORE	Ignore breakpoint; only display trace table information if any.
INFO	Display current line and optionally include all call and return stack information. In case of the program has been optimized, the debugger will show the current opcode.
LIST	List object in a disassemble form.
LOG	Log all output to a specified unix file.
MODIFY	Change value of a variable.
PROC	Terminate basic program and return to PROC if any.
QUIT	Same as END
SETBREAK	Set size of the breakpoint to a larger size than number of used entries.
SETTRACE	Set size of the trace table to a large size than number

	for used entries.
SHOW	Show various tables at the end of the object.
SOURCE	Display source program at any line number; change source file; source on/off. In case of an optimized program, typing "SOURCE" to display from current line will cause a warning. If it is still possible to come up with one line number, the debugger will display from that line for 12 line(s). But if it cannot, it will not display a line.
STEP	Step through program by # of lines or # of opcodes.
SWITCH	Switch debugging terminal to another port under control of ULT+. (This port can first trap the target port and switch the debugging terminal to the target port afterward. This way there is no security breach because the original port is already in debugger.) To switch back, type "SWITCH".
TRACE	Set trace on variables; displays trace table, trace on/off. A trace on a variable would cause debugger to display the value of the variable at breakpoints.
UNTRACE	Untrace variable(s)
VERBOSE	Turn this verbose mode on/off.

H. Utilities:

pc_to_line : maps a particular pc to the source line number. It would use the map at the end of the object for such a calculation. If the object is non-optimized, there is exactly one line number for one specific pc. There may be more the one line number for a pc in a optimized program. This routine should be used by runtime error reporting as well as by the debugger.

line_to_pc : maps a user specified line number to a pc for setting breakpoints. This is only meaningful it the program is not optimized. It should not even be called if it is a optimized program.

terminal I/O routines : It may be a good idea to have the Basic debugger's own input stream and output stream because they are not in any way related to the program input and output streams other

than the fact that initially they refer to the same terminal. The P command in the debugger will suppress all output from the rest of the ULT+ system but not from the Basic debugger. The new LOG command will redirect debugger output to a Unix file. The new SWITCH command will switch the input and output streams of the debugger to another terminal altogether.

If we are going to implement a standard TERMIO for the whole system, the TERMIO had better be able to deal with different streams, characteristics and these streams may be changed to reference another physical device during the life of the ULT+ system. This is unlikely to be implemented for this project.

If we are going to use the existing terminal I/O code, the Basic debugger may be forced to create its own set of routines to deal with its terminal I/O needs:

Initially, istream = stdin, ostream = stdout. The SWITCH command obtains the terminal path to the new physical device and makes istream and ostream to come from there. When the user desire to SWITCH back, the debugger will close the istream and ostream and makes them stdin and stdout again (remember to flush ostream). The new LOG command will cause output to go to the ostream as well as to the Unix file. Somewhat related, the Ultimate debugger P command will cause the debugger to toggle a flag in virtual space to tell VIRTUAL to suppress output to the terminal. The Ultimate debugger LP command will use whatever mechanism that is set up for the PRINTER ON/OFF Basic statements to redirect the output of the Basic PRINT statements between terminal and printer.

Complication: the output of /var command and many others commands start at the end of the input command line and it is not affected by the Ultimate TERM type. It can be done with no echo of <CR/LF> at the end of the input command line. But there is no such stty attribute to help us do that. Maybe, (1) a transparent mode character-by-character input is needed; (2) We may just have to live with the <CR/LF> at the end of the input; or (3) The debugger uses the Unix $TERM to pick up the escape sequence for cursor-up and cursor-right. Here are the needed information:

Using curses termcap emulation:

a) codename for cursor-up : up
b) codename for parm_right_cursor : RI
(Not all terminals support this)
c) codename for cursor-right : nd

```
#include <curses.h>
char up[25], rc[25];    /* we could allocate these      */
char    *str, *area;

int our_putc( c )
char c;
{       /* if tty */
        putc( c, ostream );
}
        /* Do this once at system initialization                */
        initscr();
        area = up;
        str = tgetstr( "up", &area );
        area = rc;
        str = tgetstr( "nd", &area );
        endwin();
        ...
        /* to cursor up                                                 */
        tputs( up, 1, (int (*)() )our_putc );
        /* to cursor right non-destructively 14 spaces          */
        for ( i = 1; i <= 14; i++ )
                tputs( rc, 1, ( int (*)() )our_putc );
```

VI. Time estimates:

The coding for the debugger should ideally begin when the rest of the system has been finalized. But since time is the essence, about 80% of the code can be written and tested without much difficulty. For every place that interfaces with the rest of the system, a stub is used for it and a comment of "/*^*/" is marked on the source files so that they can be easily identified. The debugger subroutines should have broken down into functional entities so that when the time comes to integrate, it should not present much difficulty (e.g. load_symbol_table, find_symbol, U_breakpoint, V_breakpoint, clear_breakpoint, pc_to_line, display_variable, term_write, term_read, etc).

Development :

Tasks	man-hrs	subtotals		
0) Getting Ready	04			
1) Init	04			
2) DCD interface	16			
3) Basic Errors	16			
4) ULT parser	04			
5) ULT commands	118			
6) utilities	40	202		
7) V. cmd parser	16			
8) V. Commands	110	126	328	
Testing :				
1) ULT commands	32			
2) Verbose	32	64	64	
			392	or 49 days

Appendix G: Analysis of Math in Basic

Introduction:

This document discusses the implementation of math operations. First today's implementation is discussed and then an alternative implementation is proposed for Optimized Basic.

Today implementation:

Format of the numeric variables:

- The numeric variables are stored as a six-byte scaled numbers, representing the numbers in the range (-(2^47)/10^SCALE) to ((2^47-1)/10^SCALE).
- The numbers that are outside of the specified range are represented as strings.

Algorithms:

- The virtual code performs the basic math operations, such as addition or multiplication in numeric format. If any of the operands, or the result of the operation, are too big(small) to be represented as a scaled 48-bit binary representation, the operands are converted to strings and string math is used.

- The string math operations such as, SADD or SMUL use the paper and pencil algorithm to calculate the result as a string.

- The other math operations such as SIN, COS, EXP, PWR, LN, and SQRT are implemented as subroutines in the virtual code. Each of these subroutines uses an algorithm associated with that operation to calculate the result. The EXP, PWR, LN use their operands as string, consequently they are very slow.

Optimized assembly code(Ultimate PLUS only):

For performance reasons the basic math operations are implemented in assembly code. The assembly code is used only if the two operands

are numeric. If the result of the operation overflows the assembly code bails out to virtual code.

Issues:

Ultimate PLUS way of representing numbers:

- Since the numbers internally are presented as scaled integers, precision is lost during the computations. For example 3.1479 * 10000 = 31400 if the precision is 2. This occurs because 3.1479 is internally stored as a scaled number 314.

Ult/ix way of representing numbers:

- The numbers are represented in double. The data type double is a 64-bit quantity, where 53 bits are used to represent the mantissa, and 11 bits are used to represent the exponent.

- The computations are more precise for numbers with mantissa in the range (-4,503,599,627,370,496 to 4,503,599,627,370,495). The precision is guaranteed only for up to 15 digits, because of the number of bits used in representing the mantissa.

The following piece of code demonstrates the difference between the Ultimate and Ult/ix. This program will print 1234 on Ult/ix implementation and 1200 on Ultimate implementation.

```
PRECISION 2
A = .1234
B = A * 1000
PRINT B
```

- No string math is used for big numbers. Consequently, Ult/ix does not come up with right values for very big numbers (numbers with more than 15 digits for mantissa). The double type can represent numbers as large as 1.79 +308(The 11 bits are used to represent exponents -1023 to 1024, the exponent represents a binary base).

- The doubles are difficult to use in logical operations. Two numbers may be extremely close but not equal. Ult/ix uses a delta for comparisons. For example the two variable A and B are compared to see if they are equal:

```
if ( abs( a - b ) < delta )
        TRUE
else
        FALSE
```

The value for delta is set by an environment variable. This can be tuned per terminal. The value of the delta is selected independently from the precision of the basic program. This causes some of the comparisons to fail. They resolve the problem by suggesting a different value for delta. The following example demonstrates the problem with delta selection:

```
A = .1
B = 0
FOR I = 1 TO 10000
        B = B + A
B = B * 100000000000
A = A * 10000000000000000
PRINT A
PRINT B
IF A = B THEN PRINT "EQUAL" ELSE PRINT "NOT EQUAL"

OUTPUT:
        1000000000000000
        1000000000000016
        NOT EQUAL
```

This occurs because the precision is guaranteed for up to 15 digits, and the multiply operation magnifies the discrepancy between A and B. No reasonable delta can be selected to avoid this problem.

Proposed implementation for the BASIC project:

The proposed implementation can run in two flavors, Ultimate plus flavor, or Ult/ix flavor. It is compatible with Ult/ix, because the variables are internally stored in the same form. However, in order to be compatible with Ultimate PLUS the double representation of the numbers need to be adjusted each time they are modified. The adjustment directly depends on the scaling factor of the basic program(see detailed explanation on the following pages). The following sections will include the data structures used, and the algorithms, and issues.

Data structure for variables:

The variables in basic will be presented as a structure with a minimum of 3 fields.

```
typedef     struct{
        int             type;
        double dbl_value;
        char            *str_value;
        .
        .
} var_type;
```

type field:
Specifies the type of the variable, string or double

dbl_value field:
If the type field is DBL_TYPE, then this field represents the current value of the variable.

str_value field:
If the type field is STR_TYPE, then this field represents the current value of the variable.

Type conversion:

During the execution of a basic program a variable may be converted from string to double or double to string several times. This conversion can be accomplished via C library routines, or they can be coded by us if the C library routines prove to be costly. A flag field in the structure can be used to indicate if the string value

and the double value are up to date to avoid extraneous conversions.

Flavor binding:

The flavor is determined at basic compile time. This means a compilation is required to switch the flavor. The flavor will be controlled via an environment variable, basic compile option, or a system wide option. It is also possible to set the flavor in any of the following 2 forms:

- The flavor is determined at runtime. This means no recompilation is required to switch the flavor. The flavor will be controlled via an environment variable, runtime option, or a system wide option.

- Bind the flavor per variable. This will require change to the BASIC language, to somehow identify those variables.

1) **Ult/ix flavor:**

- The numeric variables are always presented as double, meaning they are not converted to strings to gain more precision. In the proposed implementation we can go to string math subroutines to get better precision for huge numbers.

- The scale factor of the BASIC has no effect in the internal representation of the number, and it will only be used for external representation of the numbers.

- The Ult/ix flavor of the proposed implementation differs with the Ult/ix implementation, only in selection of delta, since the selection of delta is not a compatibility issue. The proposed implementation will avoid some of the problems encountered during logical operations of Ult/ix implementation. The delta is selected based on the value of the precision.
delta = 0.5/10^precision

2) **Ultimate flavor:**

- Each time a numeric variable is updated, the variable is adjusted based on the following formula:

```
A = floor( A * SCALE + delta ) / SCALE

SCALE       10^PRECISION (10000 for precision 4)
floor()  truncates the number
delta       0.5/SCALE
```

Where floor() is C library routine to truncate, by elimination the fraction portion of the number. On some machines such as HP the floor() routine will be rewritten in assembly to get a better performance.

The string field will be used if the operands, or the result are too big to be represented as a double or if the precision of the double is not acceptable.

Operators and C library routines used for computation:

- The basic math operations such as addition, subtraction, multiplication and division are accomplished by using the C operators '+', '-', '*' and '/' on doubles. If string math is necessary then the library routines to be coded by us will be used to do the job.

- Other math operations such as SIN, COS, LN, PWR, EXP, SQRT etc. are accomplished by using the C library routines sin(), cos(), log(), pow(), exp(), sqrt() etc.

Performance Data(Basic Math):

The first two columns are today implementation of Ultimate plus and Ult/ix. The next two columns are the proposed implementation in both flavors. The last column is today assembly code called by a stand alone C program, to help estimate Basic runtime overhead. The numbers are in units of microseconds per operation.

Operation	Today Ultimate PLUS	Today Ult/ix	Ultimate PLUS Flavor	Ult/ix Flavor	Today Assembly Code
Add(HP)	67	28	14.4	7.2	9.3
Mul(HP)	78	25	14.2	7.0	18.5
Div(HP)	76	27	14.5	7.3	16.5
Add(RIOS)	----	----	4.0	1.2	1.3
Mul(RIOS)	----	----	4.1	1.2	7.9
Div(RIOS)	----	----	4.7	1.7	8.3
EXP(HP)	6100	30	36	30	----
LOG(HP)	12500	30	37	30	----
PWR(HP)	10300	90	98	88	----
SQRT(HP)	4700	30	37	31	----
SIN(HP)	1000	31	36	30	----
COS(HP)	1000	30	37	31	----

Note that the first two columns were timed within the real systems (Basic programs) where as the last three columns were timed in stand alone C programs. The difference between the first and the last column highlights the overhead due to the architecture of Ultimate PLUS where as the difference between the second and the fourth highlights the anticipated overhead of Ult/ix's architecture (anticipated because the test of column four was done based on our general knowledge of the way Ult/ix does math).

Issues:

- If the purpose of the string math is to take care of overflow the Ultimate flavor will not test for precision each time. However, if string math is required to obtain more precision, the variables must be tested after each math operation to see if they need to be converted to string. If the test is not required the math operations will have 5-6 percent less overhead for all numbers . For example 14.4 will become 13.7 in the previous table.

Appendix H: Recall calling Basic subroutine

Format of this document

This document describes how the current interface between Recall and Basic works.
Whenever it appears that the Optimized Basic may require changes to the existing code, the document will go into a columnar format, with on the left side the current step and on the right side a proposal of a new step to take by Optimized Basic, based on the following conventions:

Keep	step should be done the same way in the new runtime;
Equivalent	step should be done but the new architecture imposes different code;
Out	no longer needed;
?	undetermined at this time;
Note n	See note number 'n' at the end of the list;

Steps taken at compile time:

For efficiency reasons, most of the work required to initialize the Basic workspace is done during the compilation phase of the Recall sentence. Each time a conversion of the type 'B' is encountered, the code (in RCL-SUBS3) which is explained here gets called.
The very first time only, the following steps are taken:

A frame is obtained from overflow and becomes the Recall Control Block (RCB) header. A pointer to it (in linked format) is saved in SCB element RCTLBSR (overlays S9).

An automatic Xmode handler is enabled for this frame, to let the control block grow up to a maximum of 128K, as required by Recall.

The first ID.DATFRM.SIZE bytes of the control block are reserved

for global element storage.

The next 4 sections, each ID.DATFRM.SIZE bytes long, are:

- a save area for the Recall PCB when switching from Recall to Basic;
- a save area for the Recall SCB;
- a save are for the Basic PCB when switching from the Basic environment back to Recall;
- a save area for the Basic SCB;

Pointers to the start of each area are saved in the RCB.

A pointer is set to the remaining space, to be used as TS workspace by Recall, which can grow to a maximum of 128K bytes.

A wrapup routine is setup in WMODE to release these frames when done.

The current (Recall) PCB/SCB are saved in the RCB.

Various bits and flags, WMODE and XMODE are reset.

RCLFLG is turned on, as a flag to the Basic initialize and to both runtime environments.

The HS, IS, OS and TS workspace pointers are re-initialized via ISINIT and TSINIT calls.

Current step	New step
Set OSBEG and R6 to dummy object code (descriptor size = 1000 bytes) and call a subroutine of the Basic initialize code ...	Note 1
... BSL B.INIT which does:	
BSL INITSTK : initialize STKBEG, STKEND for DATA stmts.	Keep
BSL BD-SETUP : initialize Basic debugger.	Note 2
Initialize CTR0 to 20000 bytes to preserve in the IS/OS buffers	Note 3

BSL HSSET : set R3 to the start of the HS buffer (PCB + 64)	Keep
MOV R3,PAGHEAD; INC R3,1500 : initialize 1500 byte area for heading/footing	Keep
MOV R3,UPDBEG; MOV BUFSIZES,C567: initialize free space pointer & sizes	Out/Note 4
INC R5,CTR0 : set R5 at start of temp space	Keep/Note 3
MOV ISBEG,R4; INC R4,CTR0; MOV R4,BDESCTBL : initialize start of the descriptor table	Out/Note 5
BSL DESCINIT : initialize the descriptor space + 90 bytes for the debugger to 0x02FF (null string)	Notes 5&6
BSL FREEINIT : initialize freespace pointer and buffer sizes	Out/Note 4
BSL INITTSTACK :	
Set R3 to special stack frame & setup automatic xmode	Out/Note 7
MOV BWRAPUP,WMODE : set Basic wrapup	Equivalent
Initialize 200 byte save area prior to R3 (for COL1, COL2, INMAT...)	Note 8
MOV R3,HSBEG; MOV R3,HSEND	Out/Note 7
MOV BSPACE,XMODE : for temp space	Keep/Note 3
Store the following elements in the save area: BASE of program file; BASE of SYSLIB file; INHIBITH;	Notes 8&9
Reserve a 1500 byte area prior to the start of temp space (ISEND) for the INPUT@ messages	Keep
Store scaling factor from object header in PCB SCALE# (for format mask)	Keep/Note10
Calculate the scale value and store in PCB SCALE.	Notes 10&11
MOV X'8000',COMDSP: for access to COMMOM variables	Out/Note 12
ZB SB8; ZB SB9	?????
ZERO PFILE, init PRMPC, SB BASICFLG, SB NOBLNK, set ICB bits	Keep
Copy 10 byte object header	

into descriptor 0	Out/Note 13
BSL INPUT@.INIT : initialize the INPUT@ message area with the ERRMSG text strings.	Keep
Core lock the OPCODES frame & save the buffer address in BOPS for the firmware	Out
INC R6 : set on dummy byte x'01'-why?	??-Chandru
MOV BDESCTBL,BAS.DESCTBL : save start of common descriptor space in RCB	Note 5
SB PAGINATE : different for normal init	??-Chandru
MOV R.WRAPUP,WMODE : set wrapup mode	Equivalent
BSL U-STOREFILEVARS (in UPD-C1): Initialize common descriptor at offset 40 as a FILE DESCRIPTOR and store BASE & MODSEP of the current file (FFCB1) into it. Clear the write enabled bit.	Note 14
Initialize common descriptor at offset 30 as a FILE DESCRIPTOR and store BASE & MODSEP of the current dict (DFCB1) into it. Clear the write enabled bit.	Note 14
Initialize common descriptor at offset 110 as a STRING DESCRIPTOR and and store the Recall file name into the descriptor (short string) or get a freespace buffer for it !!	Note 14 Note 15
MOV HSEND,R15; INC R15; MCI SM,R15 put SM in Basic stack to mark the subroutine table empty.	Keep/Note16
Corelock OPCODES frame in memory and save the buffer address in BOPS: already done in B.INIT. Why again?	??-Chandru

Notes, as they appear in the 'New step' column:

1: we could do something similar, but a less kludgy method would be nice.

2: if we keep the current debugger, the same

```
!       routine can probably be used. If not, we may                    !
!       need to write equivalent code.                                  !
!                                                                       !
!  3:   the OS buffer is likely to be used for temporary                !
!       storage by some of the instructions that will                   !
!       remain in virtual.                                              !
!       IS becomes available because the descriptors are                !
!       kept in the Kernel.                                             !
!                                                                       !
!  4:   based on the assumption that string buffers are                 !
!       managed inside the new runtime and garbage                      !
!       collect is either done there or not needed.                     !
!                                                                       !
!  5:   a routine needs to initialize the new runtime                   !
!       version of the descriptor table. This implies                   !
!       that the descriptor entries are static variables                !
!       What is initialized here is the common area for                 !
!       all the subroutines that will be called. The new                !
!       routine could return to virtual a pointer (32                   !
!       value in D0 ? ) to the initialized common                       !
!       descriptor space. This pointer can be stored in                 !
!       the RCB and later on be passed as an argument to                !
!       the runtime code.                                               !
!                                                                       !
!  6:   we may have to initialize some data structures                  !
!       for the debugger.                                               !
!                                                                       !
!  7:   based on the assumption the virtual stack is no                 !
!       longer needed.                                                  !
!                                                                       !
!  8:   for fast access, these elements should be kept                  !
!       as global variables in the C runtime. They can                  !
!       be initialized at the same time as the                          !
!       descriptor table, as 'static' elements.                         !
!                                                                       !
!  9:   the interface later on (see note 14) requires                   !
!       the capability of updating Basic descriptors                    !
!       with values from virtual. A similar routine can                 !
!       be used here to update these global elements.                   !
!                                                                       !
!  10:  the dummy object header contains a zero scale                   !
!       value: all Basic subroutines called from Recall                 !
!       run with precision zero.                                        !
!                                                                       !
!  11:  the value can be initialized when the new runtime               !
!       gets started, based on a dummy header or by                     !
!       looking at PCB SCALE#.                                          !
!                                                                       !
!  12:  the mechanism by which we can avoid the check                   !
!       for common on each descriptor access is                         !
!       explained in the document relative to the CALL                  !
```

```
!          interface.                                                    !
!                                                                        !
!   13:    the copy of the object header into descriptor 0               !
!          is specifically done for 'Recall calling Basic'               !
!          and is used to store the TOTAL size of the                    !
!          descriptor space ( common + 90 bytes for the                  !
!          debugger + each subroutine's descriptor size).                !
!          In our new runtime this should not be needed but              !
!          there could be some value in knowing the LARGEST              !
!          descriptor size so that one CONTIGUOUS block of               !
!          memory could be initialized large enough to                   !
!          contain the      COMMON descriptors and ANY of the            !
!          subroutine's descriptors.                                     !
!                                                                        !
!   14:    a routine is needed to pre-initialize certain                 !
!          descriptors with values from virtual. The input               !
!          interface could be flag bits in T4 and data in                !
!          D0 and D1.                                                    !
!                                                                        !
!   15:    Note that the space acquired to store the                     !
!          data must somehow be 'static'.                                !
!                                                                        !
!   16:    During Recall compilation a table is built                    !
!          the Basic stack containing the names of the                   !
!          subroutines that have been encountered. See                   !
!          following steps below.                                        !
!                                                                        !
--------------------------------------------------------------------------
```

If not the first time, the current Recall PCB/SCB is saved in the RCB and the Basic PCB/SCB is restored, to switch to the Basic environment.

For each of the 'B' conversions found, the following steps are taken:

if the subroutine is preceded by a file name (not catalogued), open the DICT of the file.

locate the subroutine name in the table built inside the Basic stack. Compute an index value based on the position in the table, each entry accounting for 20 bytes. If the name is NOT located (not been called previously):

add the entry to the table;

use a subroutine in mode BCALL to fetch the object

code.

Current step	New step
BSL CHECKREV :	
Check if the precision level from the object header matches the value in PCB SCALE# (always = 0 here).	Note 20 Equivalent
Check the Basic revision level by comparing the value in the object header to the one stored in mode BRP00	Note 20 Equivalent
MOV BDESCTBL,R15; MOV R15,R14: Set both registers to the object header copy in descriptor 0.	Note 13
BSL SET.DSCRP: set the subroutine descriptor space start (ISBEG) past the last 'main' descriptor, as defined in the 'main' header	
calculate the new COMDSP based on the offset to ISBEG.	
get the subroutine descriptor size, +90 for the debugger and initialize the descriptors to NULL STRING.	Note 21
Calculate the offset from the 'main' descriptor start to the last subroutine descriptor, -90 bytes for the debugger which is already allocated. Update the dummy header in descriptor 0 with that offset	Out/Note 13
Starting at byte offset 300 in the 'main' descriptor table, use the subroutine index value calculated earlier to position into this array of 'subroutine pointers'. The format of each 20 byte entry is: ----------	

!0102 030405060708 0910!	
!02ff object ptr repeat!	
!02ff desctbl ptr comdsp!	
The entry is updated by: saving the pointer to the object code (R6); setting the repeat count for multivalues to 0 saving the pointer to the subroutine's descriptor's table start (ISBEG); saving the subroutine's COMDSP value.	Note 22
MOV X'8000',COMDSP: restore COMDSP	Out
MOV BSPACE,XMODE: set for tempspace	Keep

Notes, as they appear in the 'New step' column:

20: The assumption here is that the new object header will have a different format than the old one.

21: We need a routine to initialize a number of descriptors to NULL string. It could return to virtual a 32 bit address pointer which can then be saved in the RCB. The space must be 'static'. We may have to initialize some debugger variables also.

22: The current code stores the table in an unused portion of the 'main' descriptor table, probably for convenience. There is no real reason to have it there, and for our new runtime it makes little sense to do it the same way. We do however need to store the object code pointer and, if kept in virtual, the descriptor table pointer. One area we could use is the portion of the first 1000 bytes in the RCB that is currently available.
Another possibility is the IS workspace. If the latter solution is taken, we need to skip the first 20000 bytes of IS which is reserved for Recall.

Store the subroutine index value inside the compiled Recall string, so that it can be used during the runtime to jump quickly to the appropriate table entry.

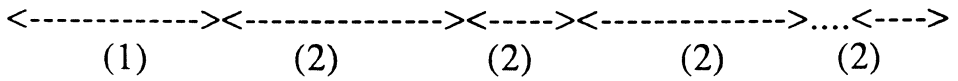
Switch the PCB/SCB environment back to Recall, after saving the current Basic PCB/SCB.

Current step	New step
For the first 'B' conversion only, MOV BAS.DESCTBL,BDESCTBL : store the pointer to the 'main' descriptor table in the Recall PCB.	Note 25

Notes, as they appear in the 'New step' column:

25: This is done so that from the Recall PCB the entry for a subroutine can be found in the table located inside the 'main' descriptor space. If we move this table to some other area (see note 22) we can remove this step.

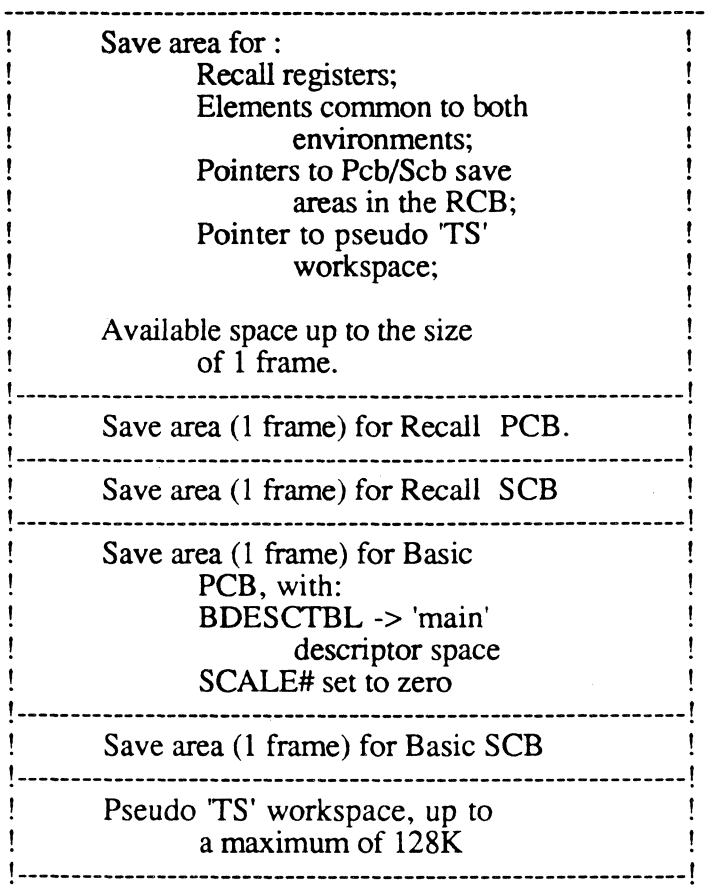
Summary of the current interface, at the end of the Recall compilation:



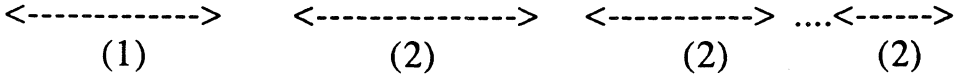
(1): Common or 'main' descriptor table, including 90 bytes reserved for the debugger. Some of the descriptors are pre-initialized with values. Part of this space is used as a table where information is kept regarding each of the subroutines, for a maximum of 35 entries.

(2): Subroutine's descriptor space, adjacent to the common block or to the last subroutine called. All descriptors are set to 'null string'.

Layout of the Recall Control Block:



Summary of the proposed new interface, at the end of Recall compilation:



(1): Common or 'main' descriptor table, possibly including some area reserved for the debugger. Some of the descriptors are pre-initialized with values. If possible, enough contiguous

space has been obtained to fit the common block and ANY of the subroutines that may be called. That way the common area never has to move (see also the CALL specification document).

(2): Subroutine's independent descriptor space. All descriptors are set to 'null string'.

Layout of the Recall Control Block:

Basically the same as before. If the pointers to the common and subroutine descriptor blocks are kept in virtual (which is not mandatory), some elements will have to be defined differently (32 bit values instead of storage registers). Also, the SR type elements in the control block are not properly aligned for the HP systems. This needs to be fixed.

Steps taken at execution time:

The code starts at label 'CALL.BASIC' in RCL-SUBS1. The input data has been stored by Recall in the 'pseudo' TS buffer, pointed to by TSBEG.

Copy the current item-id into the BMS buffer, which would normally already be there except in the case we come from a SELECT.

Save the Recall PCB/SCB and registers into the RCB. Do NOT switch to the Basic environment yet.

Pick up, from the compiled string, the index number for this subroutine, as saved during compilation (see above).

Current step	New step
MOV BDESCTBL,R15; INC R15,R.SUBPTRS, ... MOV R15,SUBS.PTR : set the register to the 'main' descriptor table, position to the 'subroutine pointer' table, use the index to position to the appropriate entry and save the resulting address in	

the RCB.	Notes 22/25

If the 'repeat value' in the current entry is zero (no other multivalues to extract from previous data), the following steps are taken:

a pointer to the current data string (TSBEG) is saved in the RCB.

the values of CTR5, CTR12 and CTR13 (current attribute, value and subvalue position) are saved in the RCB.

the values of CBIT/DBIT/C1 are examined and a resulting value is saved in the RCB.

if not previously done, a pointer to the entire item (SR0) is saved in the RCB.

the environment is switched to Basic by restoring the PCB/SCB from the save areas in the RCB.

Current step	New step
SRA R6,XFF; MOV R6,SYSR0; MOV SYSR0,R6 DCD R6,R3 : detach and reattach R6 and execute a dummy opcode to force a reload of the Basic elements known to the 7000 firmware	Out
If not done on previous calls for the same item: BSL R.SAVE.ITEM: Initialize common descriptor at offset 60 with a pointer to the entire item (SR0), as saved in the RCB.	Notes 14/15
Initialize common descriptor at offset 50 with a pointer to the item id, as saved in the BMS buffer.	Notes 14/15
MARKRTN : start with a 'clean' stack	Keep

MOV x'7C01',R3;T0;...BSL SAVE.RTN.PTRS INC R3,50 : push a return marker on the stack, and save ISBEG, OSBEG COMDSP, R6 and DAF10 on the stack	Note 30
Using the pointer to the 'subroutine entry' saved in the RCB, set R6, ISBEG and COMDSP to the values stored in the table.	Notes 22/31
Initialize common descriptors at offset 70, 80, 90 and 100 with the respective values of CTR5, CTR12, CTR13 and CBIT/DBIT/C1 as saved in the RCB.	Note 14
Initialize the SUBROUTINE descriptor at offset 30 (first variable) with a pointer to the Recall data (TSBEG) as saved in the RCB.	Note 14/15
DEC R3,50 : reset the stack	Note 30
SB RCLFLG : set for runtime	Keep
MOV R6,OSBEG : save pointer to object code start (including header). Done mostly for the debugger.	Keep
INC R6,object_header_size+size_of_abort_ instruction : skip the header and the abort instruction. The abort is there to prevent the subroutine to be executed as stand alone code.	Note 20/ Equivalent

Notes, as they appear in the 'New step' column:

30: The elements are pushed on the stack to simulate the CALL interface and to comply with the code done for RETURN. We should be able to avoid this (DAF10 may have to be saved and restored) especially since there are no 'main' values to restore. Refer to the CALL specification.

31: The new runtime needs to fetch the object code pointer and, if kept in virtual, the pointers to the 'main' and 'subroutine' descriptor tables, from the save area that we decided to use (see

note 22).	

The code then falls into the CALL routine to validate argument passing (not needed in this case) before executing the first DCD instruction.

If the repeat value in the subroutine table entry is greater than zero, it means that the return string from the last call contained multivalues which have not been passed to Recall yet. Until these multivalues are exhausted, there is no need to call the subroutine again:

Current step	New step
A register is set pointing to the data string from offset 30 in the subroutine's descriptor space, which is the data last returned by the subroutine (C.DATA). The repeat value is then used to scan to the appropriate multivalue and the resulting pointer is saved in the RCB.	Note 35

Notes, as they appear in the 'New step' column:	
35:	We need a routine that can fetch the data from a descriptor and return it to virtual. In the case of a string, it should be able to return a pointer to the string as well as to copy the string into a virtual buffer. Refer to the LIBRARY CALL specification for some thoughts on this subject.

The code returns to Recall via the 'Steps taken on exit' (see below).

Steps taken upon return from the subroutine call:

Current step	New step
In the final RETURN opcode execution R6, OSBEG, DAF10, ISBEG and COMDSP are restored with the values saved on the stack.	Note 30
BSL R.SAVE.BAS : Save the Basic PCB/SCB in the RCB.	Keep
A register is set pointing to the data string from offset 30 in the subroutine's descriptor space, which is the data returned by the subroutine (C.DATA). The register is saved in the RCB.	Note 35
RTNMARK : clear return stack up to marker	Keep

Steps taken on exit:

The Recall PCB/SCB and registers are restored from the RCB.

The Item-id is copied back into the BMS buffer, in case the subroutine clobbered it (this buffer is used in both environments).

Current step	New step
The return string, as pointed to by the saved SR in the RCB, is copied into Recall's TS buffer, up to the first SVM/VM/AM/SM found.	Note 35
Skip any subvalues and position at the next VM/AM/SM	Keep
Using the pointer to the subroutine table	

entry saved in the RCB, increment the repeat flag if the trailing marker is a VM (more multivalues to extract) or zero it otherwise.	Note 22	
Set HBIT = 1 if more multivalues to extract.	Keep	

Conclusions:

Some of the details of the current implementation need to be reviewed with Chandru.

There is a need for a routine that can initialize a descriptor table (to either 'non assigned variable' or to null string). This descriptor space must remain in place until the Basic runtime execution starts, and also in between each call to it.

There is a need for a routine that can update a given descriptor with alphanumeric or string value. If the descriptor initialization routine returns a starting address to Virtual, the latter can then use this as an input argument to this new code, since we may have multiple descriptor spaces floating around.

There is a need for a routine that can pass to Virtual either a pointer or a copy of a given descriptor value.

The format of the call from Virtual to start the runtime execution could be something like:

```
runtime( object_ptr, common_ptr, subr_ptr, flags );
                !          !       !        !
                !          !       !        ->whatever;
                !          !       -------->pointer to the
                !          !                subroutine's
                !          !                descriptor space;
                !          ->pointer to the common
                !            variables descriptor space;
                ->pointer to the object code, including
```

the header.

For normal Basic execution (ex: RUN verb), the pointers to the common and subroutine descriptor tables would be null since they are not pre-initialized.

Appendix I: Runtime Initialization

Tasks performed by the Basic runtime initialization:

What follows is a list of all the current steps the Basic initialize code goes through, when called by the RUN verb, before executing the first DCD instruction. A column to the right indicates the action to take by Optimized Basic, based on the following conventions:

Keep	step should be done the same way in the new runtime;
Equivalent	step should be done but the new architecture imposes different code;
Out	no longer needed;
?	undetermined at this time;
Note n	See note number 'n' at the end of the list;

The list starts after the retrieval of the object code from the file system (see mode BRP00, label '!RUN3'). At that point the new code can split away from the existing one to handle the Optimized Basic object code.

The 'new steps' are set based on the following assumptions:

- the descriptor table is no longer kept in virtual space;
- strings are also kept outside virtual space and 'garbage collect' is either not needed or done inside the new runtime;
- the Basic stack is either no longer used or is private to the new runtime;
- the (I) option on the RUN is not supported;
- the object header for the new runtime will have a different format than the one from the current object.

Current step	New step
Check Basic revision level by comparing the revision in the object header with the value stored in BRP00.	Keep
BSL INITSTK : initialize STKBEG, STKEND for	

DATA statements.	Keep
BSL BD-SETUP : initialize Basic debugger	Note 1
Initialize CTR0 with the count of bytes in the primary workspace (6 frames) to preserve in the IS/OS buffers	Note 2
BSL HSSET : set R3 to the start of the HS buffer (PCB + 64)	Keep
MOV R3,PAGHEAD; INC R3,1500 : initialize 1500 byte area for heading/footing	Keep
MOV R3,UPDBEG; MOV BUFSIZES,C567 Initialize free space pointer & sizes	Out
INC R5,CTR0 : set R5 at start of temp space	Keep/Note3
MOV ISBEG,R4; INC R4,CTR0; MOV R4,BDESCTBL Initialize the start of the descriptor table.	Out/Note 4
BSL DESCINIT : initialize the descriptor space + 90 bytes for the debugger to 0	Notes 4&5
BSL FREEINIT : initialize freespace table and counters	Out
BSL INITARGS : initialize argument fields for GET(ARG.) & pointer for GET(MSG.)	Keep
BSL INITTSTACK :	
Set R3 to special stack frame & setup automatic xmode for it	Out
MOV BWRAPUP,WMODE : set Basic wrapup	Equivalent
Initialize 200 byte save area prior to R3 (for COL1, COL2, INMAT...)	Note 6
MOV R3,HSBEG; MOV R3,HSEND	Out
MOV BSPACE,XMODE : init for temp space	Keep/Note 3
Store the following elements in the save area: BASE of program file; BASE of SYSLIB file; INHIBITH;	Notes 6&7
Reserve a 1500 byte area prior to the start of temp space (ISEND) for the INPUT@ messages	Keep
Store scaling factor from object header in PCB SCALE# (used in format mask)	Keep
Calculate the scale value and store	

in PCB SCALE.	Note 8
MOV X'8000',COMDSP : initialize for access to COMMON variables	Out/Note 9
ZB SB8; ZB SB9	?????
ZERO PFILE, init PRMPC, SB BASICFLG SB NOBLNK, set ICB bits	Keep
Copy 10 byte object header into descriptor 0	Note 10
BSL INPUT@.INIT : initialize the INPUT@ message area with the ERRMSG text strings.	Keep
Core lock the OPCODES frame & save the buffer address in BOPS for the firmware	Out
BSL SETLPTR; SB PBIT : init printer settings if PFLG set.	Keep
ZB PAGINATE : turn pagination off	Keep
Detach & reattach R6 for 7000 firmware	Out
BSL INITSV : if an external select list is present (DAF10 = 1), get a frame from overflow, initialize it as a type 60 descriptor pointing to a select list, set the data start & end to the top of the list (NEXTITM), and update descriptor 2 (SV) as a type 60. Also set @SELECT count in the save area = D9.	Note 7 Note 11

Notes, as referred to in the 'New step' column:

1: if we keep the current debugger, the same routine can probably be used. If not, we may need to write equivalent code.

2: keep for OS buffer. IS becomes available because the descriptors are kept in the Kernel.

3: the temp space is likely to be used in some of the instructions that will remain in virtual.

4: a routine needs to initialize the new runtime version of the

descriptor table. This implies that the descriptor entries are static variables.
Careful : SR9 still needs to be set = ISBEG for the INITARGS routine!

5: we may have to initialize some data structures for the debugger.

6: for fast access, these elements need to be kept as global variables in the C runtime. They can be initialized at the same time as the descriptor table, as 'static' elements.

7: the interface for 'Recall calling Basic' requires the capability of updating Basic descriptors with values from virtual. A similar routine can be used here to update these global elements.

8: the value can be initialized when the new runtime gets started, based on the object header contents.

9: the mechanism by which we can avoid the check for common on each descriptor access is explained in the document relative to the CALL interface.

10: this is done for 'Recall calling Basic'. This should not be needed anymore. Refer to the RECALL document.

11: something equivalent needs to be done. Depending on how the READNEXT instruction is handled (in virtual or inside the new runtime) it may make sense to copy the data into a C variable, for efficiency.

Conclusions:

I suggest that we do NOT support a mix of old and new program types (old object CALLing optimized object, optimized object CHAINing old object,...) to avoid the complexity in the code.

I also suggest to make a clean cut between the existing code and the new initialize routines, instead of having the same piece of code handle both environments. With a minimum effort in rearranging the current code, we should be able to avoid most of the duplication.

There is a need to pre-initialize certain descriptors.

There is a need to pre-initialize certain global variables.

The issues raised in this document need to be addressed in conjunction with the ones from the 'Recall calling Basic', 'Update' and 'Call' documents.

Appendix J: Library calls

Significance of Library calls to the Basic optimization project:

The vast majority of the current Library calls are not currently used in the Ult/PLUS environment since they relate to Vterm, the Performance Monitor and the 1400 Diskette driver. The following is a list of the calls that are likely to be used in Ult/PLUS:

upd$setupdflg
upd$zeroupdflg
vf$valprc
vf$valaccount
vf$valuser
vf$getmax
sf$bwhere
sf$logoff
sf$logon
sf$sec.status
sf$getxcb
sf$getrtn
sf$getwsp
sf$translate.on
sf$translate.off
sf$create.iotrans
sf$getlocks

Current interface between Basic and the Library calls:

The Basic compiler generates code similar to the normal CALL instruction. The Library definition in the R-module can request, for each argument in the call, to have it passed by value or by reference. The best way to illustrate the current mechanism is with an example:

Library call 'sf$logoff' is defined as having 4 arguments:

10	push dope vector on stack
0C	push argument value on stack

0C	same
0C	same

The LOGOFF program in SYSLIB initializes 3 arguments, for example

```
PROCESS=10
OPTIONS=0
TIME=200
```

then calls the function

```
CALL sf$logoff(USER,TIME,OPTIONS,PROCESS)
```

The object code produced is

```
BD10xxxx0Cxxxx0Cxxxx0Cxxxx09mmmmFF54
```

where

xxxx	is a descriptor offset
mmmm	the ep and mode-id for the call
BD	markstack
54	Library call opcode.

The operands pushed on the stack are

```
0100- 6 byte number-0000
0100- 6 byte number-0000
0100- 6 byte number-0000
1000- descriptor address-
7C00- bottom stack mark-
```

To parse the arguments, the runtime does

```
POPN R3
DIVX SCALE
```

To update the return value, from FP0, it does

```
MULX SCALE
BSL STOREN
```

Interface with Optimized Basic:

Since the old interface uses the stack, the proposed interface is the same as for other cases where Virtual code needs to be invoked.
Two possible interfaces are described here.

***Transparent interface to Virtual*:**

Inside the new runtime, set up arguments from the object code:

```
arg1 = VALUE_FROM_DESCRIPTOR( INDEX1 );
arg2 = VALUE_FROM_DESCRIPTOR( INDEX2 );
....
```

then call a C function that is the front end to the Virtual code:

```
logoff( *arg1, arg2, arg3, arg4 )
```

This C code is located inside the Virtual mode, and initializes the Pcb/Scb elements:

```
CTR10 = arg2;
T4 = arg3;
T5 = arg4;
```

It then falls through into the Virtual code. At the end of it, another small piece of C code provides the return interface:

```
*arg1 = CTR11;
return();
```

The runtime now updates the descriptor table with the new value:

```
DESCRIPTOR_VALUE( INDEX1 ) = arg1;
```

NOTE : the C code shown here is for illustration purposes only. If implemented, it is likely to look quite different, for example by avoiding some of the assignments.

Advantage:

The interface is clean, providing a clear distinction between the new Basic runtime and existing virtual code. A virtual coder does not need to modify the Basic runtime in order to change the Virtual element usage of the routine.

Disadvantage:

Performance loss due to the extra interface layer (kernel_table -> C argument -> Virtual argument).

Direct interface to Virtual:

Inside the new runtime, directly initialize the Virtual elements:

```
CTR10 = VALUE_FROM_DESCRIPTOR( INDEX1 );
T4 = VALUE_FROM_DESCRIPTOR( INDEX2 );
....
```

then call the virtual code:

```
pick_subcall( BSL_TYPE, ep_from_obj, fid_from_obj )
```

The virtual code ends with a RTN instruction. The runtime updates the descriptor table directly from virtual:

```
DESCRIPTOR_VALUE( INDEX3 ) = CTR11;
```

NOTE : the C code shown here is for illustration purposes only. If implemented, it is likely to look quite different, depending for example on how numeric values are stored internally.

Advantage:

This interface is fast.

Disadvantage:

The new runtime and the Virtual code are completely imbricated. Changes to the element usage require modifications in both environments.

String argument passing

Either scenario needs to deal with passing strings from the C runtime environment to virtual. The easy way is to write a routine that copies

data from one environment to the other and systematically copy the data.
In some instances however, when it is determined that the string is not passed down to processors other than Basic, it would be nice if it did not need to be copied.

Here are some thoughts:

- keep a flag associated with the register that points to the string, indicating it is located outside virtual space. All string instructions must be changed to look at that flag. Instead of a flag we could also use register numbers greater than 15.

- provide a new class of instructions that know how to deal with C strings. This implies that changes need to be made to virtual to specifically use these instructions.

Conclusion:

I suggest a Team meeting to discuss the issues listed above and possibly consider some alternatives.
Three considerations should guide the final decision:

- performance,
- transparency of the code to the virtual coder (make it easy to change),
- readability of the code.

Appendix K: String and space management

Introduction:

This paper discusses the implementation of Basic descriptor space and Basic string space. Both today are implemented as frames in shared memory. The descriptor space is implemented in the user secondary workspace where as the string space is taken out of the workspace and the overflow.

Seamlessness:

One major reason for the slowness of Ultimate PLUS is the fact that the product was not originally architectured for UNIX. Tools were architectured to allow the traditional architecture to run under UNIX. The effect of that is that some operations that used to be native operations on the traditional machine became software operations in UNIX that do not necessarily take full advantage of the native operations in UNIX.

One area that falls in this category is descriptor and heap space management. Basic makes use of both descriptors and a heap. Both are implemented in virtual frames using concepts such as register attachments and frame faulting.

The idea is to try to implement both concepts by using features that are a lot more natural to UNIX.

Space management in Basic:

The descriptor space can be implemented as a piece of memory that comes out of the UNIX/C process heap space. When a Basic program is started, a chunk of memory would be allocated to fit the necessary number of descriptors. A descriptor would be defined as a C data structure that can hold all the different types of descriptors (we would not be limited to the ten byte descriptor length anymore). The same approach would be used for

strings.

This approach will allow the Basic runtime to avoid dealing with attaching registers and frame faulting. Further more, because C structures are being used to represent descriptors, the C compiler would generate proper alignment resolving the bus exception issue on the HP series. C structures will also allow a more natural representation of native C types that would be used in Basic (floating point is an example if it turns out to be feasible to use).

Basically, as we have concerned ourselves so far with the seamlessness of Ultimate PLUS with UNIX at the user level, this type of design will allow us to be more seamless at an architecture level. The heap of a running program becomes the same as the heap of that UNIX process instead of being some data structure implemented on top.

Heap manipulation:

The traditional way to manipulate the UNIX heap is through the use of the *malloc()/free()/realloc()* set of C library calls. In our case, these routines might turn out to not be powerful enough for two types of reasons: efficient garbage collection and powerful debugging capabilities.

Whereas the C library routines will do fine to get the project moving, I anticipate a time where we will have to write a customized version that would resolve those deficiencies. This task is a tricky one and can require as much as two weeks of someone's time.

Debugging:

One important side effect of this implementation is the fact that both descriptor and string space are being moved from a memory that all users running on that same virtual machine can see and reference to one that is local to the process. This might force us to put in place a mechanism by which a program running Basic can copy his descriptor and string space to a piece of shared memory when requested. This would allow another process to be able to look at him.

Fairness:

To be fair more than one approach should have been considered. But, in this case, the old mechanism is so obviously deficient under UNIX and the proposed one so natural, that no other solution could be figured out as of now.

Appendix L: Current opcode table

.00 MODEM ERROR	
.01 MODEM BD-EOL	EOL
.02 MODEM 3,BRP06	POP NUMBER (POPN) error handler
.03 MODEM 6,BRP07	PUSH DESC (PUSHD) error handler
.04 MODEM 9,BRP04	POP STRING (POPS)
.05 MODEM ERROR	PUSH NUMBER (PUSHN)
.06 MODEM 3,BRP41	BRANCH
.07 MODEM 15,BRP06	PUSH 1
.08 MODEM 0,BRP06	PUSH 0
.09 MODEM 6,BRP43	PUSH STRING
.0A MODEM 8,BRP41	COND BRANCH TRUE
.0B MODEM 9,BRP41	COND BRANCH FALSE
.0C MODEM 3,BRP31	PUSH ADDRESS
.0D MODEM 6,BRP01	STORE at ADDRESS (STOREA)
.0E MODEM 0,BRP31	EQUAL
.0F MODEM 6,BRP06	NOT(..)
.10 MODEM 0,BRP37	PUSH ABS ADDRESS
.11 MODEM 1,BRP49	INCREMENT
.12 MODEM 6,BRP41	BRANCH EQUAL LIT STRING
.13 MODEM 6,BRP41	BRANCH UNEQUAL LIT STRING
.14 MODEM ERROR	STRING BRANCH TRUE (Defer)
.15 MODEM ERROR	STRING BRANCH FALSE (Defer)
.16 MODEM ERROR	
.17 MODEM ERROR	
.18 MODEM ERROR	
.19 MODEM 5,BRP41	Fast FOR
.1A MODEM 2,BRP43	MULTICAT (1 byte subopcode follows)
.1B MODEM 6,BRP34	PASSMAT
.1C MODEM ERROR	
.1D MODEM 7,BRP41	GOSUB
.1E MODEM 4,BRP41	FORNEXT
.1F MODEM 10,BRP41	RETURN TO
.20 MODEM ERROR	DUMMY AFTER REMOVE
.21 MODEM 5,BRP06	OR
.22 MODEM ERROR	
.23 MODEM ERROR	
.24 MODEM ERROR	

.25 MODEM 1,BRP31	LESS THAN OR EQUAL
.26 MODEM 8,BRP06	AND
.27 MODEM 2,BRP53	PROMPT
.28 MODEM ERROR	
.29 MODEM 3,BRP02	READVU
.2A MODEM 0,BRP27	MULTIPLY
.2B MODEM 0,BRP30	ADD
.2C MODEM 1,BRP36	MATREADU
.2D MODEM 1,BRP30	SUBTRACT
.2E MODEM ERROR	
.2F MODEM 0,BRP32	DIVIDE
.30(0) MODEM 3,BRP05	OPEN x,y ...
.31(1) MODEM 1,BRP02	DELETE [fileitem]
.32(2) MODEM 15,BRP08	CLEARFILE
.33(3) MODEM 0,BRP16	SELECT (1 byte subopcode follows)
.34(4) MODEM 6,BRP02	READU
.35(5) MODEM 3,BRP02	READV
.36(6) MODEM 4,BRP02	WRITEV
.37(7) MODEM 0,BRP58	SELECT BY index
.38(8) MODEM 6,BRP02	READ
.39(9) MODEM 7,BRP02	WRITE
.3A MODEM 9,BRP18	INPUTCLEAR
.3B MODEM 4,DYNAMIC	PVFA push value from address
.3C MODEM 2,BRP31	LESS THAN
.3D MODEM 1,BRP05	OPEN x ...
.3E MODEM 1,BRP36	MATREAD
.3F MODEM 0,BRP57	MATWRITE
.40(@) MODEM 3,BRP50	ONGOTO
.41(A) MODEM 3,BRP26	ABS(...)
.42(B) MODEM 1,BRP03	PRINTER
.43(C) MODEM 7,BRP18	INPUT ONE BYTE (INPUTZERO)
.44(D) MODEM 0,BRP17	INPUT
.45(E) MODEM 4,BRP11	EXIT
.46(F) MODEM 2,BRP15	GET(...)
.47(G) MODEM 0,BRP24	SUBSTR
.48(H) MODEM 10,BRP03	HEADING FOOTING (1 byte subopcode follows)
.49(I) MODEM ERROR	
.4A(J) MODEM 0,BRP10	FMT(...)
.4B(K) MODEM 11,BRP51	INPUTCONTROL
.4C(L) MODEM 1,BRP16	READNEXT VAR (1 byte subopcode follows)
.4D(M) MODEM 1,BRP59	READPREV VAR (1 byte subopcode follows)

.4E(N) MODEM 1,BRP27	NEGATE
.4F(O) MODEM 0,BRP04	TAB
.50(P) MODEM 8,BRP03	PRINTCRLF
.51(Q) MODEM 9,BRP03	CRLF
.52(R) MODEM 0,BRP34	RETURN
.53(S) MODEM 4,BRP43	SPACE(...)
.54(T) MODEM 4,BCALL1	LIBRARY function call
.55(U) MODEM ERROR	reserved for RETURN expression
.56(V) MODEM 0,BRP54	BITAND(..)/BITOR(..)/BITXOR(..) one byte subcode follows
.57(W) MODEM 2,BRP13	TAN(...)
.58(X) MODEM 4,BRP18	PRINTERR
.59(Y) MODEM 2,BRP06	NUM(...)
.5A(Z) MODEM 8,BRP03	PRINTCAT
.5B MODEM 0,BRP48	RELEASE COMMON
.5C MODEM 5,BRP03	PRINT ON
.5D MODEM 7,BRP06	ERRSETUP
.5E MODEM 2,BRP03	RQM SLEEP
.5F MODEM 6,BRP01	STORE
.60 MODEM 8,BRP08	RND(...)
.61 MODEM 3,BRP43	TRIM{B,F}(...) (1 byte subopcode follows)
.62 MODEM 2,BRP04	MATCH(...)
.63 MODEM 3,BRP04	CHAR(...)
.64 MODEM 2,BRP54	DATE()
.65 MODEM 1,BRP52	CHAIN
.66 MODEM 0,BRP43	FIELD(...)
.67 MODEM 4,BRP04	COL1()
.68 MODEM 5,BRP04	COL2()
.69 MODEM 2,BRP50	INT
.6A MODEM 7,BRP11	CLEAR
.6B MODEM 8,BRP04	SEQ
.6C MODEM 1,BRP50	LEN(...)
.6D MODEM 5,BRP50	MAT
.6E MODEM 2,BRP21	ICONV(...)
.6F MODEM 3,BRP21	OCONV(...)
.70 MODEM 6,BRP49	PAGE (1 byte subopcode follows)
.71 MODEM 4,BRP09	ASCII(...)
.72 MODEM 5,BRP09	EBCDIC(...)
.73 MODEM 5,BRP43	STR(...)
.74 MODEM 3,BRP54	TIMEDATE()
.75 MODEM 1,BRP54	TIME() (1 byte subopcode follows)

.76 MODEM 6,BRP20	CLOSE
.77 MODEM 3,BRP36	MATCOPY
.78 MODEM 10,BRP18	INDEX(...)
.79 MODEM 6,BRP18	CURSOR(...), @(...)
.7A MODEM 15,BRP18	CURSOR(x) (one parameter)
.7B MODEM 8,BRP03	DISPLAYCAT
.7C MODEM 8,BRP03	DISPLAYCRLF
.7D MODEM 3,BRP34	SUBR
.7E MODEM 0,BRP50	SORT(...)
.7F MODEM 2,BRP07	WEOF
.80 MODEM 0,BRP13	SIN(...)
.81 MODEM 1,BRP13	COS(...)
.82 MODEM 0,BRP25	LN(...)
.83 MODEM 0,BRP26	EXP(...)
.84 MODEM 3,BRP13	SQRT(...)
.85 MODEM 0,BRP14	PWR(...)
.86 MODEM 3,BRP19	FILEINFO(.) INDEXINFO(.) (1 byte subopcode follows)
.87 MODEM 2,BRP05	COUNT(...)
.88 MODEM 0,BRP22	SYSTEM (...)
.89 MODEM 0,BRP32	REM(...) MOD(...)
.8A MODEM 3,BRP25	STOP
.8B MODEM 1,BRP53	RELEASE ALL
.8C MODEM 3,BRP42	DATA
.8D MODEM 0,BRP56	BEGIN SCREEN
.8E MODEM 4,BRP50	ONGOSUB
.8F MODEM 1,BRP21	BREAK ON-OFF
.90 MODEM 6,BRP21	ECHO ON-OFF
.91 MODEM 0,DYNAMIC	INSERT(...) INS
.92 MODEM 0,BRP29	REPLACE(...) <.>=
.93 MODEM 1,BRP28	EXTRACT(...) <.>
.94 MODEM 0,BRP28	DELETE(...) DEL
.95 MODEM 0,BRP61	LOCATE(...)
.96 MODEM ERROR	
.97 MODEM 3,BCALL	CALL @ (INDIRECT)
.98 MODEM ERROR	
.99 MODEM 4,BRP13	ALPHA(...)
.9A MODEM 7,BRP02	WRITEU
.9B MODEM 4,BRP02	WRITEVU
.9C MODEM 0,BRP57	MATWRITEU
.9D MODEM 0,BCALL	CALL direct

.9E MODEM 4,BRP11	ABORT
.9F MODEM 2,BRP05	DCOUNT(...)
.A0 MODEM DEBUG	DEBUG
.A1 MODEM 5,BRP08	LOCK
.A2 MODEM 7,BRP08	LOCK ELSE
.A3 MODEM ERROR	
.A4 MODEM ERROR	
.A5 MODEM ERROR	
.A6 MODEM 6,BRP08	UNLOCK UNLOCKALL (1 byte subopcode follows)
.A7 MODEM ERROR	
.A8 MODEM 2,BRP52	ENTER
.A9 MODEM ERROR	
.AA MODEM 6,BRP02	RELEASE
.AB MODEM 0,BRP07	READT/TX/TL (1 byte subopcode follows)
.AC MODEM 1,BRP07	WRITET/TX/TL (1 byte subopcode follows)
.AD MODEM 3,BRP07	REWIND
.AE MODEM 9,BRP51	PUSH ABSOLUTE DOPE VECTOR
.AF MODEM 9,BRP51	PUSH ABSOLUTE ARRAY ADDRESS
.B0 MODEM 6,BRP03	PROCREAD
.B1 MODEM 7,BRP03	PROCWRITE
.B2 MODEM 1,BRP61	LOCATE
.B3 MODEM ERROR	
.B4 MODEM 2,BRP38	EXECMARKSTACK (1 byte subopcode follows)
.B5 MODEM 0,BRP15	PUT(...)
.B6 MODEM ERROR	
.B7 MODEM 6,BRP02	READU LOCK
.B8 MODEM 3,BRP02	READVU LOCK
.B9 MODEM 1,BRP36	MATREADU LOCK
.BA MODEM ERROR	reserved for FILE LOCK/UNLOCK
.BB MODEM ERROR	
.BC MODEM 11,BRP41	MODIFY DOPEVECTOR
.BD MODEM 1,BRP34	MARKSTACK
.BE MODEM 1,BRP15	SEEK(...)
.BF MODEM 3,BRP15	EOF(...)
.C0 MODEM 0,XMATH6	STORAGE
.C1 MODEM 1,BRP45	MATCHFIELD(...)
.C2 MODEM 0,XMATH1	FADD(...)
.C3 MODEM 1,XMATH1	FSUB(...)
.C4 MODEM 2,XMATH1	FMUL(...)
.C5 MODEM 3,XMATH1	FDIV(...)

```
.C6 MODEM 2,XMATH2      FCMP(...)
.C7 MODEM 1,XMATH2      FFIX(...)
.C8 MODEM 0,XMATH2      FFLT(...)
.C9 MODEM 3,XMATH4      SADD(...)
.CA MODEM 4,XMATH4      SSUB(...)
.CB MODEM 2,XMATH5      SMUL(...)
.CC MODEM 3,XMATH6      SDIV(...)
.CD MODEM 1,XMATH6      SCMP(...)
.CE MODEM 1,BRP39       ERRTEXT(...)
.CF MODEM 3,BRP39       USERTEXT (...)
.D0 MODEM 6,BRP42       TRAP ON
.D1 MODEM 1,BRP43       MULTI FIELD(...)
.D2 MODEM 0,BRP35       FIELD STORE
.D3 MODEM 2,BRP28       LEFT EXTRACT
.D4 MODEM 2,BRP36       MATPARSE
.D5 MODEM 1,BRP42       CONVERT
.D6 MODEM 2,BRP42       CLEARDATA
.D7 MODEM ERROR         reserved for ATAN(...)
.D8 MODEM 0,BRP52       SUBSTRING STORE
.D9 MODEM ERROR         reserved for MAXIMUM(...)
.DA MODEM ERROR         reserved for MINIMUM(...)
.DB MODEM 5,BRP42       LET THEN
.DC MODEM 6,BRP04       INMAT()
.DD MODEM ERROR
.DE MODEM 1,XMATH5      ONECAT
.DF MODEM 2,BRP34       REUSE()
.E0 MODEM 4,BRP42       Duplicate stack
.E1 MODEM 0,BRP42       REMOVE
.E2 MODEM 0,BRP44       SUM()
.E3 MODEM 3,BRP37       VECTOR DESC
.E4 MODEM 4,BRP37       VECTOR ADDR
.E5 MODEM 5,BRP37       MATRIX DESC
.E6 MODEM 6,BRP37       MATRIX ADDR
.E7 MODEM 2,BRP49       NUMOBJ INCREMENT
.E8 MODEM 8,BRP37       NUMOBJ STORE
.E9 MODEM 9,BRP37       NUMOBJ PUSH DESC
.EA MODEM 10,BRP37      NUMOBJ PUSH ADDR
.EB MODEM 1,BRP38       EXECUTE
.EC MODEM 0,BRP23       Runtime variable DIM
.ED MODEM 0,BRP47       NAMED COMMON
*
```

```
* END run-time OPCODES - The remainder are reserved by the compiler
* for use during pass 2 - 4 of compilation.
* ** except F2, F3, F6, F7, FA, FB, FD **
*
.EE MODEM ERROR          ! DIMENSION
.EF MODEM ERROR          ! EQU NUMERIC LIT
.F0 MODEM ERROR          ! EQU STRING LIT
.F1 MODEM ERROR          ! EQU CHAR(XX)
.F2 MODEM 6,BRP41        EXTENDED BRANCH EQ LIT ! EQU SIMPLE
                         VAR
.F3 MODEM 6,BRP41        EXTENDED BRANCH NE LIT ! EQU SIMPLE
                         TO ARRAY ELEMENT
.F4 MODEM ERROR          ! EQU ARRAY OVERLAY
.F5 MODEM ERROR          ! COMMON SIMPLE VAR
.F6 MODEM 12,BRP41       EXTENDED BRANCH ! COMMON ARRAY
.F7 MODEM 10,BRP41       EXTENDED RETURN TO
.F8 MODEM ERROR          ! STOREA OF ARRAY WITH NUM SUBS
.F9 MODEM ERROR          ! LOAD OF ARRAY WITH NUM SUBS
.FA MODEM 14,BRP41       EXTENDED COND. BRANCH TRUE ! NOP !
.FB MODEM 15,BRP41       EXTENDED COND. BRANCH FALSE
.FC MODEM ERROR          ! LABEL AT COMPILE TIME
.FD MODEM 13,BRP41       EXTENDED GOSUB
.FE MODEM ERROR          unusable
.FF MODEM 4,RCL-SUBS1
```

Appendix M: Machine stack versus software stack

Advantages of the current stack architecture:

it provides a convenient vehicle for passing values and pointers from one opcode to the next;

our firmware implementations can execute primitives that manipulate the stack directly, without going to virtual.

Disadvantages, and reasons for not perpetuating the current design:

software implementations, such as Ult/PLUS, do not have proprietary hardware available to execute stack related instructions;

loading values, pointers or descriptor indexes from the object code requires pushing these elements on the stack, when they could be accessed directly, after being loaded once into a constants area;

often results of functions can be stored directly in the target descriptor. Having to push the result first, then pop it off adds overhead for no purpose;

the Unix systems that currently run Ult/PLUS are RISC based platforms that use a fast stack architecture at the hardware level, geared towards running C programs. We should try to take the most advantage of this, without adding a software stack on top of it.

from a performance angle, using the machine stack shows a clear advantage over using a software stack. Take the following example:

```
FOR I = 0 TO 500000
        X = I
        Y = I + 2
        Z = Y - 3
        R = X + Y + Z
```

```
NEXT I
```

Tested with a stack architecture, the code is:

```
Main:                    ADD subroutine:
PUSH X                       POP A
PUSH Y                       POP B
ADD                          PUSH( A + B)
PUSH Z                       RETURN
ADD
POP R
```

On the HP 835, the timing for it varies between 4.20 and 4.40 seconds.

Tested with the machine stack, the code is:

```
Main:
R = ADD( X, ADDIT( Y, Z ) )

ADD( A, B ) subroutine:
RETURN( A + B )
```

On the HP 835, the timing varies between 1.10 and 1.20 seconds, showing an almost 4 to 1 gain.

Disadvantages of not using the software stack:

The optimizer, the object code format and the runtime code require a more complex design in order to eliminate the need of such a stack.

Appendix N: Call interface

Introduction

This document first lists the current steps done by CALL, then proposes an alternative solution in the framework of Optimized Basic.

There are two CALL opcodes:

> Opcode x'9D' is a DIRECT call: the name of the subroutine to invoke is a string constant which follows the opcode. The top entry on the stack is an offset to a reserved descriptor for this subroutine call.
>
> Opcode x'97' is an INDIRECT call: the top of the stack is a copy of a descriptor which points to the subroutine name (or contains it in case of a short string). If the subroutine is opened, prior to the call, by an " OPEN 'SUB', name TO var " type instruction, the subroutine name has already been resolved and the 'var' descriptor will contain a pointer to the object code, with a displacement value of x'8001'. The high order bit is set by the OPEN routine as a flag to the CALL interface.

If any arguments are passed to the subroutine they are already on the stack by the time the CALL opcode gets executed. If an argument is a variable, the stack will contain a dope vector (pointer to the descriptor) for it. If it's a constant, the stack will either contain the value (numeric) or a pointer to the constant (string).
The bottom entry on the stack is always a stack marker, even if no arguments are passed.

Different cases of CALL are examined here. Portions of the code that are common to all cases are explained in the later sections.

One case is not documented : the call to an assembler mode (CM pointer in MD). This interface has never been documented and, to the best of my knowledge, is not used internally. Its functionality has been superseded by the Library function call. I suggest that we do not support it in the new implementation.

Section 1 : direct call- subroutine is first invoked:

Position R6 past the name in the object code.

Set R12, via SETDD, to the descriptor who's offset is on top of the stack. If the calling program is the TRAP handler (YBIT=1), set R12 via SR8 instead if ISBEG (refer to the 'Trap handler' specification).

Verify the descriptor type is not x'40' (subroutine called before).

Copy the subroutine name into the BMS buffer.

Save elements on the Basic stack, to restore when RETURNing (see Section 6 below).

If the current descriptor type is greater than x'48', call the ABANDON routine to free the descriptor. This is ONLY possible if the calling program was ENTERED from a routine with a different descriptor table layout.

Fetch the object code address of the subroutine (see Section 7 below).

Store the object code address (R6) in the descriptor and set the type to x'40', so that the next call (if any) to this routine knows the name has been resolved.

Join common code - see Section 5.

Section 2 : direct call- subroutine has been called before:

Position R6 past the name in the object code.

Set R12, via SETDD, to the descriptor who's offset is on top of the stack. If the calling program is the TRAP handler (YBIT=1), set R12 via SR8 instead if ISBEG.

Verify the descriptor type is x'40' (subroutine called before).

Save elements on the Basic stack, to restore when RETURNing (see Section 6 below).

Join common code - see Section 5.

Section 3 : indirect call- subroutine has not been opened:

Ensure the top stack entry is not a number (type x'01') followed by x'8001'.

Set a register, via POPS, to the subroutine name.

Set R12 to a scratch location (AF buffer) because indirect calls do not have a dedicated descriptor. Fake the descriptor type by storing a x'00' at R12. This is done so that common code can be used later on.

Copy the subroutine name into the BMS buffer.

Save elements on the Basic stack, to restore when RETURNing (see Section 6 below).

Fetch the object code address of the subroutine (see Section 7 below).

Store the object code address (R6) in the fake descriptor and set the type to x'40'.
This is NOT used later on.

Join common code - see Section 5.

Section 4 : indirect call- subroutine has been previously opened:

Ensure the top stack entry is a number (type x'01') followed by x'8001'.

Save elements on the Basic stack, to restore when RETURNing (see Section 6 below).

Reset the high order bit of the displacement field in the stack, and set R6 to the object code.

Join common code - see Section 5.

Section 5 : common code:

Check if the calling program is a Recall subroutine and, if so,
set the descriptor table start pointer (R14) to the value of BDESCTBL, which in fact points to the 'main' descriptor table;

set the object header pointer (R15) to the same location, since descriptor 0 of the 'main' table contains a fake object header;

set COMDSP to x'8000';

The reason for all this (refer to the 'Recall calling Basic' document) is because:

the new descriptor space needs to be at the end of the TOTAL descriptor space (main + all the subroutines), not just at the end of the current subroutine's descriptor table;

the fake object header is the only place where the TOTAL descriptor size is stored;

the offset to the COMMON block needs to be calculated from the top of the 'main' descriptor space.

else,
set R14 at the start of the caller's descriptor table (ISBEG);

set R15 at the start of the caller's object code (OSBEG);

The address of the descriptor table (R14) is incremented by the descriptor table size from the header (R15) added with 90 bytes (for use by the debugger). The resulting address becomes the start of the subroutine's descriptor space (ISBEG).

COMDSP is decremented by the same offset.

The table size from the new object header (R6) is used to initialize all the subroutine's descriptors to zero ('non assigned variable').

The start of the object stream (R6) is saved in OSBEG.

R6 is incremented by the header size plus two, to point one byte before the first opcode to execute. The two bytes skipped are for the first instruction which is an ABORT. This prevents subroutines to be executed directly, without being called.

The object register (R6) is detached and reattached to ensure element attachment on the 7000 firmware systems.

XMODE is setup with the BSPACE routine, so that 'temp space' can grow as needed.

The number of arguments passed in the CALL is compared with the number of SUBROUTINE arguments, in the following way:

the subroutine's object code is scanned until a x'7D' opcode is found, looking for
x'0D' : store,
x'1B' : mat.
Each occurrence accounts for one argument.

The Basic stack is scanned until a stack marker is found (type x'7C'). Each entry accounts for 1 argument.

If the numbers do not match the program aborts in the Basic debugger.

Finally, the first opcode gets executed via a DCD instruction.

Section 6 : elements pushed on the Basic stack:

Three entries are pushed on the stack, past the arguments if any, containing information pertinent to the calling routine:

the descriptor table address (ISBEG);
the offset to common variables (COMDSP);

the address of the object code header (OSBEG);
the external select list flag (DAF10);
the current object code address (R6).

The latter entry is pushed with a stack descriptor type x'18', as a flag to the RETURN code (see below).

The top of the stack is saved in SR19, after which it is reset prior to these entries: the subroutine's object code expects the top of the stack to be just above the first argument to copy.

Section 7 : finding the object code from the subroutine name:

The PCB's BASE and MODSEP are set to the user's MD, to first look for a catalogued program. The name in the BMS buffer is used to retrieve the item (via RETIX).

If found and the first attribute matches the string 'PC',

the name is saved in the CS buffer;

the 'catalogue' pointer is scanned until attribute 5 where the program file name is stored;

if the program name starts with a blank,
the file defaults to SYSLIB. The SYSLIB BASE is copied from the save area prior to HSBEG (refer to the Basic initialization document) to the PCB's BASE, and the item is looked up in that file.

else,
the DICT of the file name found is opened, the program name is restored from the CS into the BMS buffer and the item is looked up in that file.

else,

the program is presumed to reside in the same program file as the current one. The BASE for this file is copied from the save area

prior to HSBEG (refer to the Basic initialization document) to the PCB's BASE and the name is looked up in that file.

The item, if found, must be a pointer item with the first attribute matching the string 'CC'. The starting fid of the object code is converted from attribute 2 of the pointer.

The precision set in byte 5 of the object header must match the value of SCALE# in the PCB.

The revision level set in byte 4 of the header must match the value store in frame 230 (BRP00).

The first opcode in the object must be x'9E' (ABORT).

If the program can not be found, or if any of the above conditions fails, the program aborts in the Basic debugger.

Section 8 : executing the SUBROUTINE opcode:

It's only function is to reset the stack pointer (R3) past the argument list and past the saved elements, using SR19.

Section 9 : passing arguments from the caller to the subroutine:

With the exception of common variables, a source descriptor, once it has been copied to the subroutine's descriptor, is changed to a type x'04' and its contents are modified to become a pointer to the target descriptor. This is done because strings are not normally duplicated in the new environment: the descriptor is set to point to the caller's data. In the case of a type '82', the backward link of the buffer in freespace is set to the new descriptor. The problem arises when an argument is passed multiple times: in that case the string does need to get copied into a new buffer. The x'04' type, which is only used for this purpose, allows the STORE routine (see code in BRP01) to detect this condition and force the data copy.

If the value fits inside the descriptor, multiple copies of the same variable cause no problem: the value is copied into each of the target descriptors. When returning (see Section 10) the caller's descriptor will have the last

subroutine.

The subroutine's descriptor type is set to 0, to prevent multiple string assignments from the same source, and also as a flag to the ABANDON routine so as to not release the data buffer.

x'1B' : return an array. For each element in the array:

The caller's descriptor is overlaid with the one from the subroutine.

If the descriptor to copy is a type x'82', the backward pointer of the freespace buffer is changed to point to the caller's descriptor.

The subroutine descriptor is changed to a type x'04' to prevent multiple updates from the same source, and also as a flag to the ABANDON routine.

If the array is of the 'variably dimensioned' type, clear the 'array inherited' flag. This indicator is used by the DIM instruction to prevent subroutines from re-dimensioning the array while the caller's stack contains an absolute pointer to it.

A stack marker must now be on top of the stack. The subcode following the type can be:

x'00' :	regular CALL instruction;
x'01' :	RECALL calling Basic. Execution continues at a specific location (refer to the 'Recall calling Basic' specification);
x'82'	
x'84'	
x'88' :	from TRAP subroutine. Refer to the 'Trap handler' specification.

All 'freespace' buffers used by the subroutine are abandoned.

that in the new environment we ensure object sizes can be large enough so that this restriction does not become an issue. Source code can easily be kept in small portions via the $INCLUDE directive.

Optimized Basic can not CALL old object code, and vice versa, because of the complexity involved in building a bridge interface.

Proposed object code for a direct call:

DIR_CALL:CONSTANT(X):DESCRIPTOR(Y):ARGUMENTS..END_MARKER

If DESCRIPTOR(Y) is not currently assigned:

call a Virtual routine similar to the one described in Section 7, passing it a pointer to the program name in CONSTANT(X).

store the OBJECT_PTR setup by Virtual (R6 typically) inside the descriptor and set the type code appropriately.

else

pickup the OBJECT_PTR from DESCRIPTOR(Y).

Proposed object code for an indirect call:

IND_CALL:DESCRIPTOR(Y):ARGUMENTS...END_MARKER

If DESCRIPTOR(Y) has a 'opened subroutine' type code,

fetch the OBJECT_PTR from the descriptor,

else

scan the 'cached calls buffer' for this program name. I suggest a small buffer, with a most 5 entries.
If found,

fetch the OBJECT_PTR from the cache area.

The constants table is loaded from the OBJECT_PTR.

Global variables are initialized.

Calling from a Recall subroutine:

Besides the fact that their descriptor space is static and pre-initialized, Recall subroutines are not different from other Basic programs. There should be no special code required to execute the CALL instruction.

Passing COMMON variables between programs:

COMMON variables are stored in the top portion of the descriptor table. The number of common variables needs to be retrieved from the object header, and may not necessarily be the same as the count from the calling program: their size equivalence is not enforced although it probably should.
There are several ways the data can be shared:

copy the individual descriptors in each direction. Clearly not very efficient;

copy the table itself, in each direction, for the portion of the COMMON descriptors that is used in both programs.

use the common area from the caller, overlaying the descriptors that are not shared with the ones from the subroutine. Obviously the overlaid portion needs to be saved first, and restored when the subroutine terminates.

use a separate descriptor table for the common descriptors. If this can be achieved in a way transparent to the runtime code (for example by passing a pointer to the descriptor argument instead of an index into a unique table) then this is clearly a better solution, since it avoids data copy and/or extra memory allocation.

Some of the proposed solutions assume that the 'string space' can be freely shared between programs;

only those descriptors that were modified need to be copied. This is not as desirable because of the overall performance impact on descriptor updates.

Detecting the return condition:

When a RETURN instruction is encountered and the local stack of GOSUB return addresses is empty, then the subroutine must return to the calling program.

Returning values back to the calling program:

The argument list at ARG_PTR is scanned again for arguments that require to be passed back:

if the 'has been copied' flag is set in the caller's descriptor entry:

reset the flag;
blindly copy the descriptor, since any space reallocation has been taken care of inside the subroutine.

else

'free' the current descriptor if needed, then overlay it with the one from the subroutine.

Reset whatever necessary in the subroutine's descriptor so that the string space passed back to the caller does not get released.

Open issues:

Do we need special code to handle the TRAP subroutine ?.

How do we pass certain global variables that are currently kept in the area prior to HSBEG ?. The ones that are only used by code that is going to remain in Virtual can certainly stay there. Others, like the COL1() and COL2() values, are specific to each runtime environment and do not need to be copied. But what about the break on/off counter?

Appendix O: The execute instruction

Scope of this appendix:

This appendix lists the major steps taken by the virtual code upon execution of a Basic EXECUTE statement. The code examined is as of release 210F.

Basic stack layout when the opcode (x'EB') is invoked:

Top->	0x....SR.....	storage pointer to the command string to be executed.
Followed by any of:		
	0x..21..SR.....	storage pointer to STACKed data.
	0x..22..offset..	descriptor offset to return CAPTURed data into.
	0x..23..SR.....	storage pointer to the PASSLIST data to be passed.
	0x..24..offset..	descriptor index to return RETURNING data.
	0x..25..offset..	descriptor offset to return the RTNLIST data into.
Bottom->	0x7e.....	stack marker.

Steps taken during runtime execution:

the command to execute is copied into the OS buffer (PCB+5).

the CHAINFLG bit in the PCB is set true.

the Basic break off counter is copied into the PCB INHIBITH byte.

execution is started in the new workspace:

the Basic stack from the caller's workspace is scanned, until a stack marker is found, for any of the following sub-types:

the current pointer is copied to SCB element NXTITM, the current pointer in the descriptor is set to null (at the SM) and PCB bit DAF10 is set true.

else, if an external select list is active in the caller's workspace (bit DAF10 is true):

the caller's SCB pointer NXTITM is copied to the current SCB's NXTITM pointer, the caller's NXTITM frame id is zeroed and the PCB bit DAF10 is set true.

the statement to execute is copied from the caller's OS workspace to the current OS and PCB bit CHAINFLG is set true.

other non Basic related elements are passed from the old to the new environment (see code in mode PSPACES0).

the statement is executed, via the MD1 routine.

through the code in WRAPUP-I execution resumes in the previous TCL level.

non Basic related elements are copied from the execute workspace to the current one.

any message string from the PROC secondary input buffer in the execute workspace is copied to the current IS buffer (for the GET(MSG.) instruction) and, if the RETURNING clause is in effect, is also copied into the descriptor pointed to by SCB storage register SR3. In that case the target descriptor is forced to be of type x'60'.

if a list generation statement was executed (PCB bit SELECTFLG is true) and items are selected (PCB bit DAF10 is true):

if the RTNLIST clause is in effect:

the descriptor pointed to by SCB element SR2 if forced to be of type x'60', and the returned list, as pointed to by the

Issues in regard to Basic optimization:

The interface between optimized Basic and virtual needs to provide all the necessary data elements and descriptor pointers so that information can be easily accessed and/or updated.

The part of the code executed by the execute TCL level to parse the Basic stack can either:

be accommodated by simulating a stack interface, in which case the old and new runtime environments could share this code,

or can be replaced in the optimized runtime environment by new code that accesses the data elements via a to be designed interface.

Appendix P: Enhancements survey results

On July 9th 1991, an R&D meeting was held in the large conference room to discuss the Basic performance rewrite. Fifteen persons were present, mostly programmers. A list of suggested enhancements was compiled throughout the meeting. Those are enhancements that would possibly speed up Basic. The list was put on a memo and sent back to all fifteen people asking each one to distribute 20 votes across the items. One could give every item from 0 to 20 votes with a total for all of his votes of twenty. Eight persons responded. Here follows the compiled results of those votes.

The table has three columns:

- the first column indicates how many persons voted for that item

- The second column indicates how many votes were totaled for that item.

- the third is a description of the enhancement.

Votes		Suggested enhancement
2	4	Compiler could type some variables by looking at all uses of each variable. This could avoid some of the type code tests at runtime.
2	2	Rewrite mask/format routines to be more efficient
5	9	Speed up CALLs: Do not return variables that don't change - Compiler could type some of them.
0	0	Rewrite POPN/POPNS from virtual to C.
2	5	Speed up passing MATs in CALLs.
2	7	Eliminate double data manipulation on MATWRITE by modifying UPDITM to understand arrays.
5	11	Implement Webb's File I/O speed enhancements (eliminate locking on READs).
4	8	All dynamic arrays change to runtime dimensioned arrays.
1	1	Compiler could change EXECUTE 'WHO' to using @WHO, etc..
5	16	Implement file I/O locking mechanisms directly in C.
2	2	Execute needs to be speeded up (do not reinitialize the workspace

Appendix Q: Compatibility between old Basic and new Basic

At this point in time, it is intended to make both basics completely compatible. As the project will proceed though, we may identify areas of the system that may not be compatible anymore. Some utilities for example may not be applicable anymore or there may be new tools with the rewrite etc... A functional spec for the compatibility issue will need to be developed later on when we closer to completing the development. That spec will become this appendix.