Ultimate

THE ULTIMATE CORP.

Assembly Language Reference Guide

The Ultimate Corp.
East Hanover, NJ

Version 1

Ultimate Assembly Language Reference Guide
Version 1

How to order this guide:

The Ultimate Assembly Reference Guide is a restricted document. For information on ordering, call the Ultimate Administration Department.

Publication Information

Operating System Release 10, Revision 210

Document No. 6973

Contents

Figures

Tables

How to Use this Manual

This manual is intended as a reference for programmers using the Ultimate Assembly language. Although not a tutorial, it covers all aspects of using Assembly language with the Ultimate system file structure and operating system. The material is presented in a structured format, with text and figures integrated into single-topic units.

The Ultimate operating system is written mainly in the Ultimate assembly language. Users may also write their own programs in this language. This manual assumes that the reader has some familiarity with the Ultimate computer system and with programming concepts in general. For an overview of the system hardware and software components, see the Ultimate System Overview manual. For a description of the various programming languages and Ultimate-supplied system and application programs, see the appropriate user reference manuals.

How the Manual is Organized

This manual contains nine chapters, five appendices, a glossary, and an index. The following describes each of these components.

Chapter 1, Introduction to the Assembler, gives an overview of programming with Ultimate Assembly language. It covers the virtual system architecture, kernel software, and management of virtual memory.

Chapter 2, The Assembler, explains how the assemblers operate, including use of the symbol files, the format and editing of instructions in source items, assembler options and directives, and the assembly process itself. It also summarizes programming conventions

Chapter 3, Addressing and Representing Data, describes how data can be represented, addressed, and manipulated in an assembly language program. It covers the topics needed to write an assembly language program for the Ultimate operating system. This includes the formats of linked and unlinked frames, data formats and the use of registers to address data. It also discusses the Ultimate system conventions for writing assembly language programs, such use of global variables, control blocks, and workspace buffers, re-entrancy, PCB fields, and SCB fields.

Chapter 4, The Instruction Set and Directives, details each instruction and assembler directive in the assembly language set in alphabetical order.

Chapter 5, System Subroutines, lists, in alphabetical order, the system subroutines that users may call, with one listing for each *root* routine. The root entry contains its associated routine names (different suffixes). These subroutines perform specific functions such as reading command lines or taking care of file management tasks. The standard system elements used as inputs and outputs are listed, and subroutine operations are explained.

Chapter 6, System Software Interfaces, discusses the Ultimate system flow of control, and conventions for interfacing between the system and a user-written assembly program. When a program is ready to run, it must be integrated to work within the system control flow. This chapter

discusses the various ways a program can be executed; for example, as a type of system command or as a subroutine called from an appropriate system indicator or flag.

Chapter 7, References for Programmers, gives some guidelines on recommended methods for using the instruction set. It also contains examples of programs and their interfaces with the system. This chapter is intended as a transition for programmers who are new to the Ultimate system.

Chapter 8, The Assembly Language (System) Debugger, explains the tools available for program testing and debugging in the Assembly (System) Debugger. The Debugger messages are also included.

Chapter 9, MCALs contains a list of the system monitor calls.

Chapter 10 contains details about internal instructions. This chapter will eventually be merged into chapter 4.

Conventions

This manual presents the general syntax for each BASIC statement and function. In presenting and explaining the syntax, the following conventions apply:

Convention	Description
UPPER CASE	Characters printed in upper case are required and must appear exactly as shown.
lower case	Characters or words printed in lower case are parameters to be supplied by the user (for example, line number, data, etc.).
{ }	Braces surrounding a parameter indicate that the parameter is optional and may be included or omitted at the user's option.
bold	Boldface type is used for section and unit headings. It is also used in examples to indicate user input as opposed to system displayed data.
RETURN	The RETURN symbol indicates a physical carriage return pressed at the keyboard. A RETURN is required to complete a command line, and signals the system to begin processing the command.
<key>	Angle brackets are used to indicate a key other than letters or numbers; for example <ESC>.
enter	The word enter is used to mean "type in the required text, then press RETURN."
X'nn'	This form is used to define a hexadecimal number where 'nn' is the hex value; for example, X'0B', X'41', X'FF'.
`Enter option`	This typeface is used for messages and prompts displayed by the system.

1 Overview of Assembly Language

Ultimate assembly language is a generalized language that is not tied to any specific CPU type. The assembly language program source code is the same on any Ultimate system, regardless of the underlying hardware. After the source program is written, an assembler process, provided by Ultimate, compiles it into object code for specific hardware. A different assembler process is needed for each type of hardware.

Assembly language programming on any computer requires greater attention to detail than programming in higher-level languages, but it also provides more control over the machine. Also, assembly programs tend to be much longer in source form than equivalent programs written in a high-level language such as BASIC, but the generated object code is often shorter and more efficient.

The Ultimate operating system is written mainly in assembly language.

The main features of the assembly language are

- symbolic addressing, which allows locations to be addressed by a symbolic name as well as by an absolute number
- bit, byte, word, double-word, and triple-word operations
- memory to memory operation using relative addressing on bytes, words, double-words, and triple-words
- bit operations permitting the setting, resetting, and branching on condition of a specific bit
- branch instructions which permit the comparison of two relative memory operands and branching as a result of the comparison
- addressing register operations for incrementing, decrementing, saving, and restoring addressing registers
- byte string operations for the moving of arbitrarily long byte strings from one place to another
- byte string search instructions
- buffered terminal Input/Output instructions, with selectable type-ahead

- all data and program address references handled by virtual memory
- operations for the conversion of binary numbers to printable ASCII characters and vice versa
- arithmetic instructions for loading, storing, adding, subtracting, multiplying, and dividing the extended accumulator and a memory operand
- control instructions for branching, subroutine calls, and program linkage

The Ultimate Virtual System Architecture

The concept of virtual memory used by Ultimate is that all data on disk, including files, is addressable by any assembly program. Any process on the system can address the entire disk in exactly the same manner. Software conventions are used to control and limit a particular process from using space that belongs to some other process, but there is no hardware enforced "memory exception" type of error.

This concept of virtual memory differs from that used by other systems where each process has its own process area and cannot address any other area and where files are not part of the addressable area.

Figure 1-1 shows a typical layout of an Ultimate virtual memory system.

Virtual memory is organized into blocks called frames . A frame is a fixed block of data resident on the disk, which can be transferred between disk and main memory. The size of a frame may vary from one hardware implementation to another; on firmware machines it is 512 bytes.

All frames are uniquely identified by a frame number, or frame-identifier (called the FID). Frame numbers start at one and continue to the last available frame in the disk set. The physical limit on the frame number is (2**24)-1, or 16,777,215. The frame numbers map directly into disk addresses.

For additional information on the Ultimate virtual memory system, see Chapter 3.

Caution! *This ability to address any data in virtual memory, which gives assembly programming its power, can also be dangerous. Unlike BASIC, which tends to affect only the account or terminal on which it is run, assembly programs can affect several terminals or even destroy data throughout the system (including most of the operating system itself).*

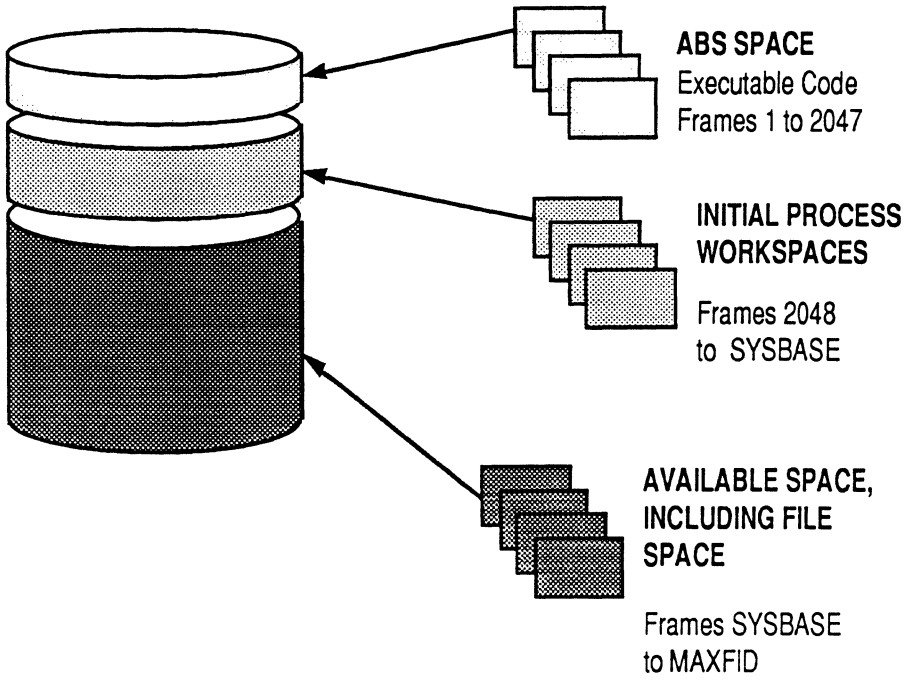


Figure 1-1. Virtual Memory System

User Processes in a Multi-User System

A process is an operating entity within the system that has its own functional elements and workspace. A virtual process is typically attached to a line on one of the asynchronous communication channels available on the system, and is therefore often called a channel or, more commonly in Ultimate, a port or line.

Each user interacts with the system via an assigned process line. A peripheral device connected to the line, usually a terminal, is the user's means for interaction with the system. All Ultimate systems can support one or more users at a time; the maximum number of users and/or processes that can be running at one time is a function of the system's configuration.

Note: *In addition to processes that are assigned to physical lines, the print spooler and network processes are always assigned to "phantom lines".*

Figure 1-2 illustrates an Ultimate system with many processes.

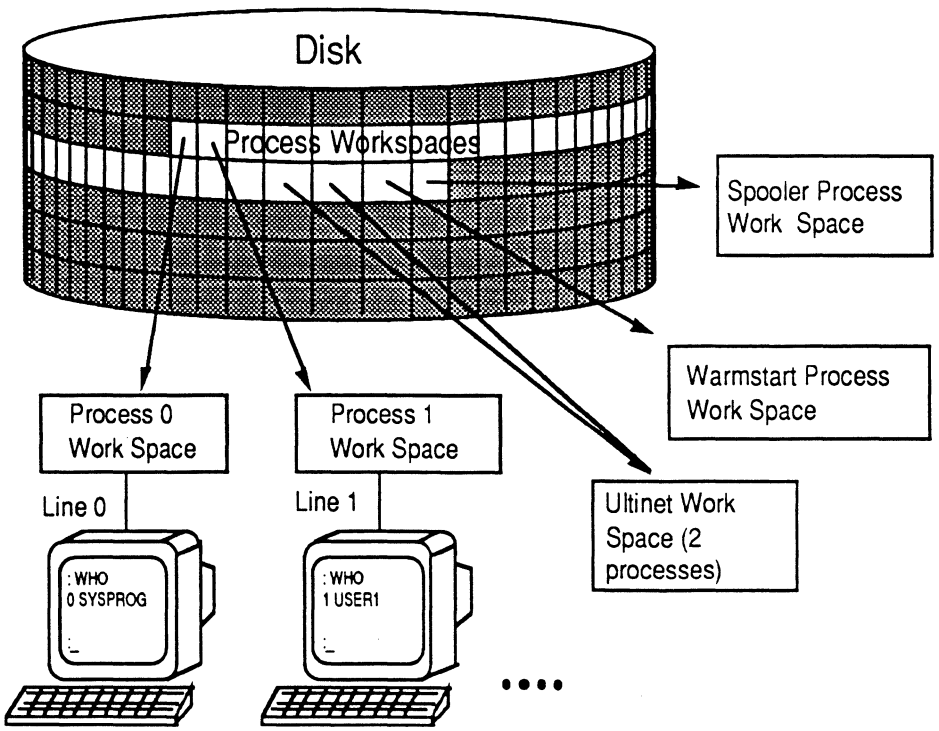


Figure 1-2. Processes

Process Workspaces

Each process has a dedicated area of virtual memory called the process workspace (see Figure 1-3). Approximately 256K bytes of workspace are reserved for each process.

The first frame of each process work space is called the Primary Control Block (PCB). The PCB is used for assembly program "housekeeping" requirements such as registers for manipulating data, stacks for program loops, and an accumulator for arithmetic functions. When a process executes an assembly instruction that references an element in this housekeeping area, the reference is always relative to the beginning of the workspace assigned to that process. This allows several processes to execute the same program simultaneously.

The format of a PCB is shown in Appendix B.

In addition to the workspace in virtual memory,each process has a dedicated block of space in main memory called the Process Identification Block (PIB). The PIB is a fixed block of main memory that serves to define the status of a virtual process. It is used by the Kernel for process scheduling and input/output operations associated with a process, and contains all information necessary for process activation.

The PIB and its extensions constitute the only elements of a process that are always in main memory. All other information associated with a process is in virtual memory, and can remain on disk if the process is not active.

For more information about PIBs, see Appendix C.

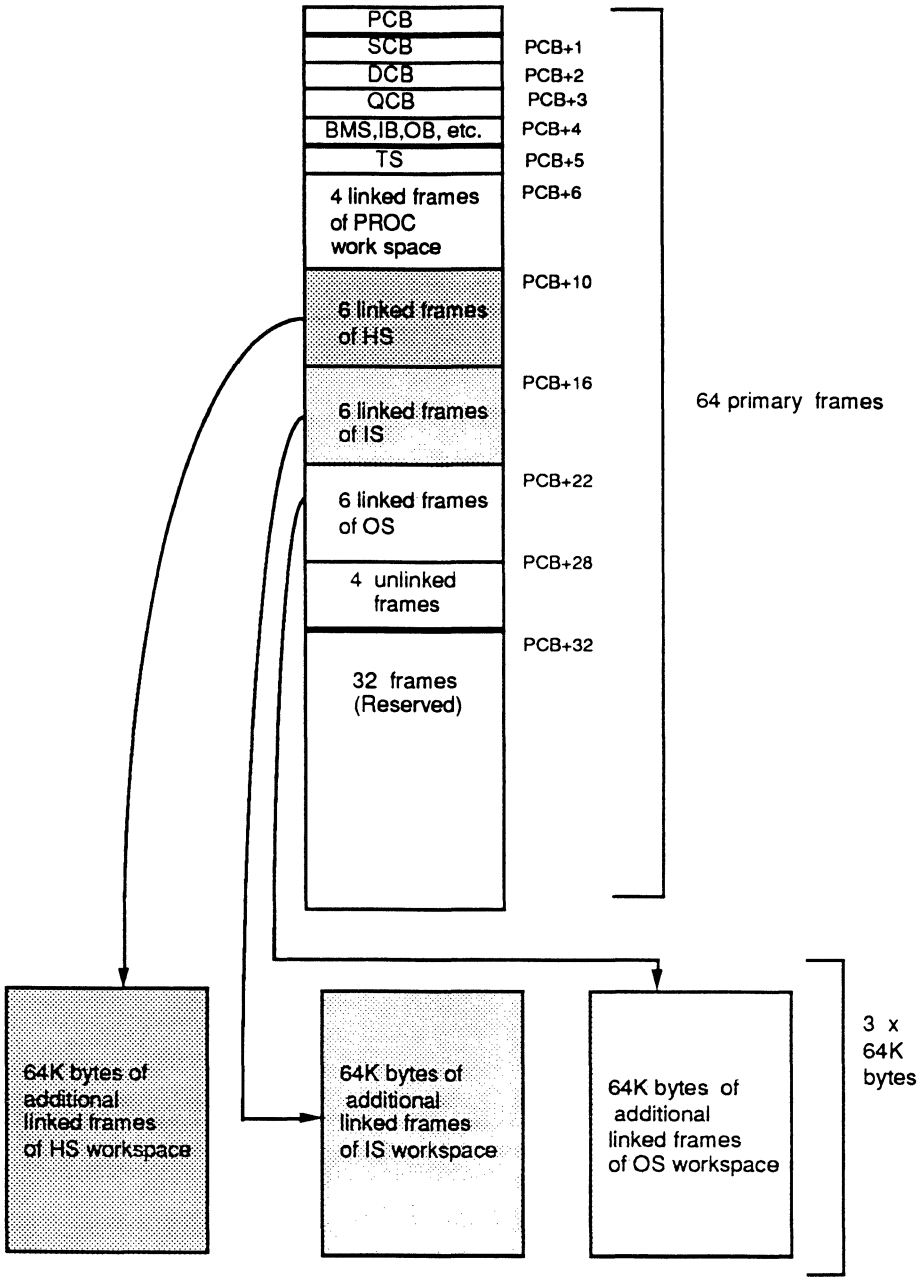


Figure 1-3. Process Work Space

The Kernel Software

The Kernel is the executive program of an Ultimate system. It is responsible for virtual process scheduling, all I/O, monitor calls, and management of memory tables.

The Kernel software differs from other assembly language software in the following respects:

- it is resident in main memory
- it is usually written in the "native" language of the machine (Honeywell Level 6, DEC LSI-11, Motorola 68000, etc.), unlike virtual software which is written in Ultimate assembly language
- it can address any location in memory directly

All input and output (I/O) from an Ultimate system to the disk is under control of the Kernel. No other process can explicitly perform any I/O to the disk. For example, when a user process issues a write command, a flag is set in main memory to indicate that a disk write is required. The actual writing of data to disk happens at some time later as determined by the state of the memory buffer and the Kernel (and is transparent to both user and process).

At system startup, the Kernel process is used to coldstart the system. This involves loading all system software and starting up the processes that make up a multi-user computer system.

When the system is running, the Kernel is called whenever the following tasks are needed:

- process scheduling
- frame faults
- automatic disk writes
- special functions that are requested by a user process via an assembly language Monitor Call (MCAL) instruction
- terminal input/output

Figure 1-4 shows the main memory portion of an Ultimate computer system. Note that the fixed portion contains the Kernel software and a few other control tables (which are discussed later in this section). The

virtual portion is simply storage space to be used as needed by user processes.

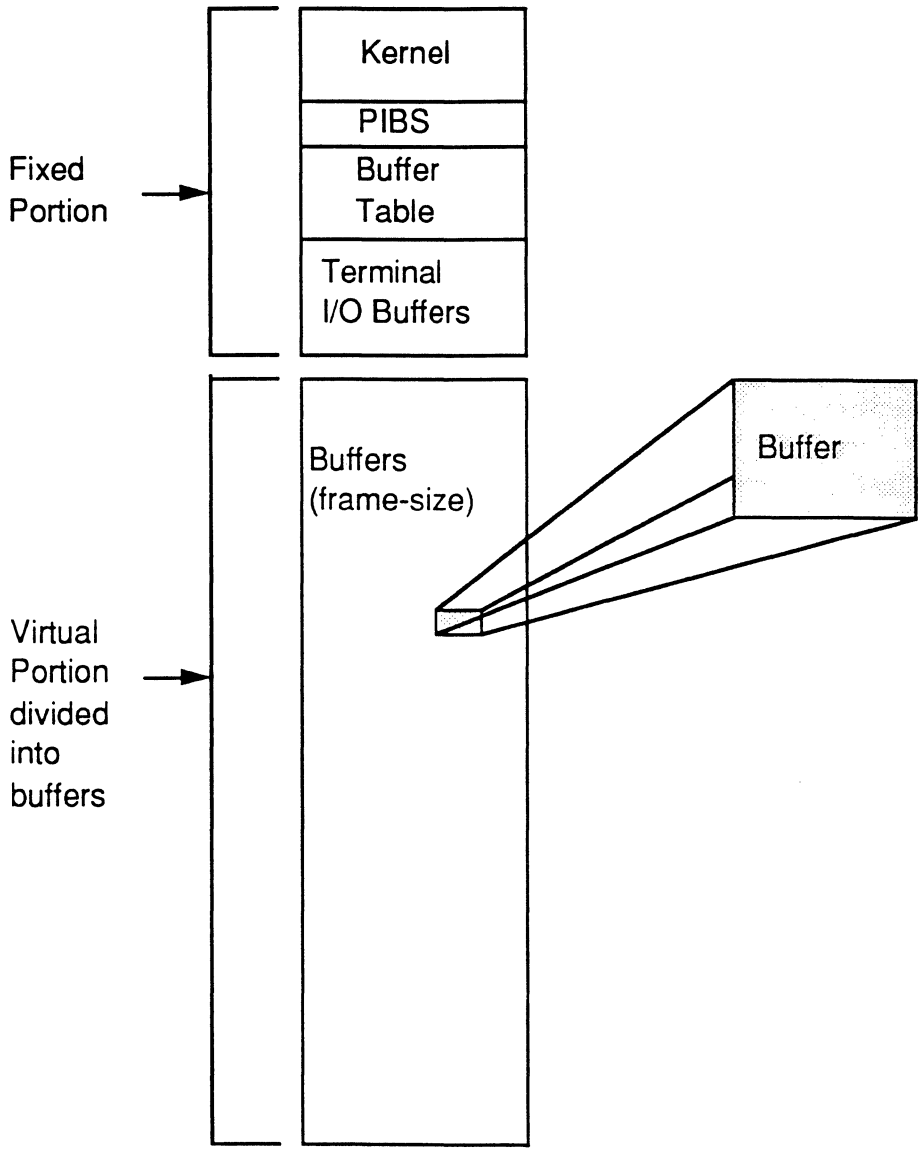


Figure 1-4. Main Memory Layout

Process Scheduling

A process may be active or inactive . The Kernel maintains a schedule of available processes, their current statuses (active or inactive), and their relative priority to be activated.

When the Kernel turns over control by selecting the virtual process that is next in line, with no roadblocks to prevent activation, that process is said to be active .

A process is inactive, but eligible to be activated, if it has returned control to the Kernel due to one of the following events:

- The process has executed a Monitor Call instruction. Normally, when the Kernel has completed the function that it was called upon to perform, it reactivates the virtual process immediately.
- The process was terminated by some external interrupt such as a timeslice runout.

A process is inactive and roadblocked if it has returned control to the Kernel due to one of the following events; the process will not be eligible to be activated until the roadblock is resolved:

- The process has made reference to data which is not in main memory. This causes a frame fault trap to the Kernel.
- The process has executed a READ (asynchronous channel byte) instruction when the terminal input buffer is empty.
- The process has executed a WRITE (asynchronous channel byte) instruction when the terminal output buffer is full.
- The process has executed a SLEEP and the time has not elapsed.

Frame Faults

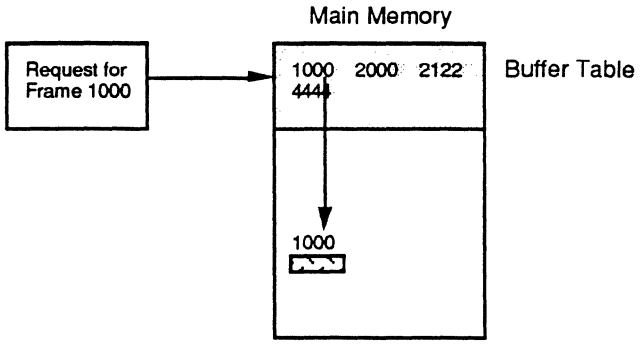
The Kernel handles disk scheduling, which involves bringing data from the disk into main memory for processing. This mechanism is called a "frame fault".

Data is transferred between disk and the main memory, frame by frame (one frame at a time). Each frame is stored in memory in a block of the same size; the memory block is called a "buffer".

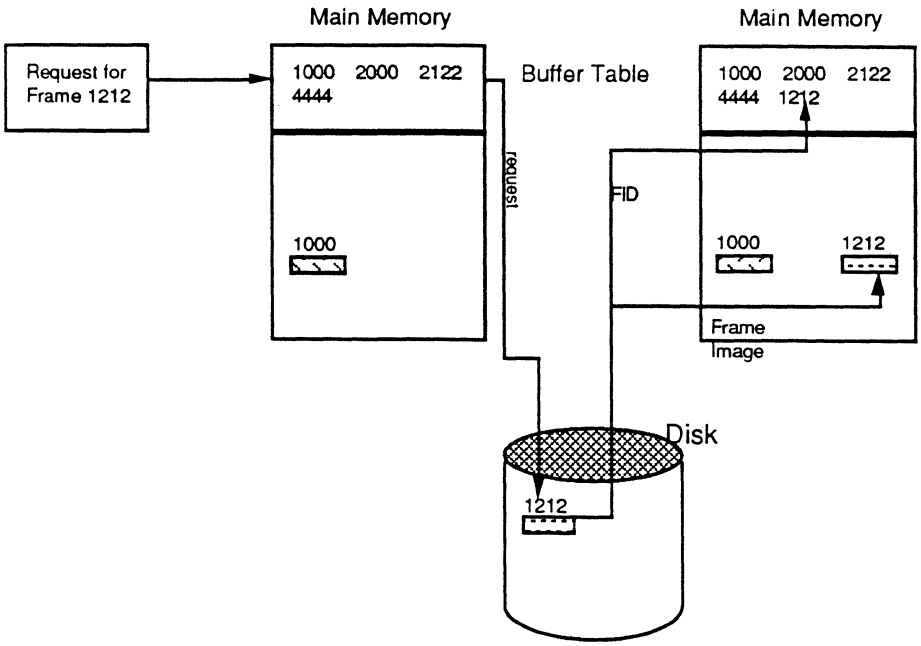
The FID of each frame that has been brought into memory from the disk is kept in the buffer table. Each time a frame is referenced, the system checks for its FID in the buffer table; if the FID is not there, the frame must be retrieved from disk (a "frame fault" occurs). After the frame is brought into memory, its FID is put into the buffer table.

Before the system brings the frame into memory, it checks to see if the buffer table is full. If it is full, the least recently used frame is written to disk if necessary, its FID removed from the table, and the new FID is added.

Figure 1-5 illustrates the retrieval of frames. The first figure shows the case of a frame already in memory. The second figure shows the case of a frame fault.



FID in buffer table - no frame fault



FID not in buffer table - frame fault

Figure 1-5. Frame Fault

Automatic Disk Writes

Periodically, such as whenever the system is idle, the Kernel attempts to "flush" memory by writing buffers to disk which have their write-required flags set. This ensures that updated data is safely on disk in case of a power failure, which could destroy the contents of main memory.

If uninterrupted, the Kernel writes one or more write-required buffers at a time to disk and resets the write-required flags, until memory is flushed. Various types of interrupts, however, such as frame faults from virtual processes, can suspend the automatic-write mechanism. During this time, the disk is kept busy reading in requested frames, and writing other frames out as needed on a least-recently-used basis. When the system again becomes idle, the automatic-write mechanism is restarted.

The precise criteria for determining when the system is idle is subject to variation according to configuration and operating system release.

Calls (MCALs) from Processes

User processes communicate with the Kernel via assembly language instructions called Monitor Calls (MCALs). Each Ultimate implementation has its own set of MCALs that allow assembly language programmers to call the Kernel whenever any I/O functions are needed.

All I/O operations initiated at the virtual level, except those to or from the asynchronous communication channel, are accomplished through the MCALs.

The format and meaning of these Monitor calls depend on the particular Ultimate implementation being used; no details are given here. However, standard system subroutines are provided in Section 6 for programmers to use with common devices such as tape drives and line printers (e.g., TPREAD, SETLPTR, WRTLIN, etc.).

Main Memory Management

In main memory, several kilobytes are reserved for use by the Kernel for its resident software, tables, etc. Other areas of memory contain the variable-size memory mapping table, the extent of which is dependent on the size of main memory. All remaining main memory is available as buffers for disk frames.

In order to manage the main memory, the Kernel uses several tables that contain information regarding the buffers. These tables may be accessed by memory management firmware as well as by the Kernel software. They are not accessible to the virtual processes.

The protection afforded to the tables is set up by the initial condition of the tables themselves. Since the memory map indicates the relationship between a disk address and a main memory location, the protected areas of memory do not have corresponding disk addresses, and therefore cannot be addressed by a virtual process.

2 The Assembler

The Ultimate operating system is configured on a wide variety of computers. On some computers, such as the Honeywell Bull DPS-6 and various Digital Equipment Corporation (DEC) models, a firmware implementation is used. On others, such as the IBM 4300 or 9370, a software implementation is used. The assembly language program source code is the same for all implementations.

A firmware implementation is one in which the virtual machine language is directly executed by underlying firmware. In addition to instruction decoding, the firmware also aids in virtual memory management.

A software implementation is one in which the virtual machine language is translated to the native machine language of the computer by the assembly process.

The assembly language program is assembled at the TCL level using the process referred to as the assembler. The assembler generates the machine-specific object code that is needed to execute the program on a given implementation. There is one assembler for firmware implementations and a different assembler for each software implementation.

At this time, there is no assembler for Ultimate PLUS implementations.

The Components of an Assembly Program

Assembly language programs are stored as items in disk files. A program is made up of assembly language instructions, as well as directives that are interpreted and used by the assembler.

An assembly *instruction* tells the system to perform a specific program operation, for example, move an element. An assembly *directive* tells the assembler to perform a specific function about the way the program is assembled (for example,.define and reserve space for symbols).

An instruction or directive must contain an operation code mnemonic (opcode), and may also contain a label, operands, and comments. Only one instruction or directive can appear on a program line. The general format is:

```
{label} opcode {operand{,operand...} {comments}}
```

Only the opcode is required; operands may be required, depending on the instruction. Labels and comment fields are optional. One or more blanks are needed to separate label from opcode, opcode from first operand, and last operand from comments.

If a program line has a label, the label must start at the first character position in the line. If a line does not have a label, there must be at least one blank space before the opcode. A label may be composed of either alphabetic or numeric characters.

The comment field can be used to explain or document the program operation. It allows the programmer to keep a running commentary on the meaning or purpose of each line of code.

In a program item, extra blank spaces surrounding the opcode or operands in a line are ignored; however, all-blank lines or null lines are illegal.

Displaying the Program

The MLIST command and the line editor AS command can be used to produce a formatted listing of the program. Figure 2-1 shows a sample excerpt from an assembly program's source code, formatted using MLIST. (For more information on MLIST, see the section, Assembler System Commands.)

```
!START  EQU   *
        BSL   LOGHDR
        BBZ   RMBIT,RTN       rtn if error
        MCI   SM,R15          mark header end in CS
        BSL   INITTAPE
        BZ    TCTLBSRF,RTN    tape problem
        BNZ   REJCTR,RTN      tape problem
        INC   INHIBITH
        MOV   OSBEG,OS
        INC   OS,1+ID.PWS.SZ and stay here until wrapup
        FAR   OS,4
        MOV   XPFID,D8        save until wrapup
        MOV   OSFID,RECORD
        INC   RECORD,-1+ID.WS.FRAMES [abt0387]
        MOV   RECORD,XPFID    link first to last
        BSL   RDLINK
        MOV   OSFID,XNFID     link last to first
        MOV   SHED,MAP
        LOAD  PRECL
        STORE BLOCKSIZE
        LOAD  PROCESS#        get my PIB#
        STORE CAMP#
        MOV   OS,DECKBEG
```

Figure 2-1. Sample Assembly Program Source Code Lines

Creating an Assembly Language Program

An assembly language program, also called a *mode*, is created using either the line editor or the screen editor. However, only the line editor provides assembly formatting.

The line editor can be set to display the lines of code in assembly listing format, using the following commands:

AS assembly listing format on/off switch; default OFF.

M macro expansion display on/off switch; default OFF.

S suppress object code on/off switch; default OFF.

In addition, the following command can be used to locate a line of object code in a previously assembled mode:

Q/loc#/ locates the line that contains object code location 'loc#', which is specified as a hexadecimal byte offset in the current mode (for example: 005D). Differs from L/string/ in that only object code is searched, and the match is on a location, not a string value.

For more information on the editors, see the *Guide to the Ultimate Editors*.

Assembly Structures

An assembler mode item has a specific overall structure and each program line within the mode has a specific structure. The assembler program checks for this structure. In addition, the line editor uses this structure to display assembly source code lines in a standard assembly listing format, including object code, if any is present.

Mode Structure

The assembler expects an assembly source mode to begin with comment lines. The comments may use as many lines as needed. Following the initial comments, the assembler looks for the beginning of the Entry Point Branch Table, followed by the directives that are used to define symbols and registers in the program. This section of the program is then followed by the main program instruction routines. This structure is similar to the following:

```
001  FRAME directive.
002 * Comment line.  By convention, program type/purpose.
003 * Comment line.  assembler places current system date
004 * Comment line.  By convention, these lines contain
005 * Comment line.   revision level, author, and other
006 * Comment line.  explanatory comments.
 .
 .
0nn entry point 1
 .
 .
nnn final entry point
xxx symbol definitions
 .
 .
yyy main program
 .
 .
zzz END
```

Each of these elements is discussed on the following pages.

The end of the program can be indicated by an END directive, but this is not actually required by the assembler.

FRAME Directive

The FRAME directive specifies the frame in which this program mode is to be loaded. FRAME also sets the assembler's location counter to 1 or 2, depending on the implementation. You may use an ORG directive to reset the location to ORG 0 if you wish to use the first 1 or 2 bytes.

The frame number must be within the limits for ABS frames. For release 200, the limits are frame 0 to frame 2047. In general, user-written code should be loaded into frames 400-599; Ultimate reserves these frames for user modes. It is possible that other user modes and applications already have used some of these frames, so be sure to check that the frame is free before using it.

Note: The USER-MODES file in the SYSPROG account contains user modes that are loaded by the COLD-START PROC. This is a good starting point in detecting used program frames.

Comment Lines

A comment line is defined by an asterisk (*) in column 1 or by the CMNT directive. The * comment line has no tabbing performed; it is one long line of text comments. The CMNT directive must be in column 2 or beyond; everything else on that line is considered to be comments. A CMNT directive may be preceded by a label.

*Note: The assembler puts the system date in line 3 only if it is a comment line that begins with an *.*

Entry Point Branch Table

This is a sequence of up to 16 Entry Point (EP) instructions that defines the entry points (numbered 0-15) into the mode. The entry points may be given sequential labels such as 0, 1, 2, etc., or alpha labels. (For information on using the entry points to execute the program, see Chapter 6.)

The entry points must be the first instructions that generate object code.

By setting the entry points up as a series of branches, you can later change the program and reassemble it without affecting the entry points.

Note: Although no entry points are required to be defined after the last used entry point, it is usually safer to put NEP instructions in place of all unused entry points.

Mode-ids - External Program References

All assembly language programs to be executed must be identified by a mode-id in order for the system to access the correct frame (FID) and entry point in memory where the program is located.

A mode-id is a 16-bit field (that is, it fits in a tally), and is composed of one hex digit for the entry point and three hex digits for the frame number (FID). Together these make up an address to which execution control can be transferred in a program.

Every program needs to have a defined mode-id; however, the mode-id is actually stored in different places, depending on the system interface being used to initiate the program:

- If the program is to be executed as a verb (system command) from TCL, the mode-id is stored (in ASCII character format) in the verb definition item in the Master Dictionary (MD) of each account that runs the program.
- If the program is to be executed via the CONV (Conversion) interface, the mode-id is given as part of the 'Unxxx' conversion code in the BASIC ICONV or OCONV function that calls it. If the program is associated with Recall attributes, the mode-id is given in the 'Unxxx' (User Exit) Correlative or Conversion code (line 7 or 8) in a dictionary attribute definition item.
- If the program is to be executed from PROC, the mode-id is given as part of the 'Unxxx' or 'Pnxxx' PROC command that calls it.

  In all 'Unxxx' specifications, the 'nxxx' is four hexadecimal digits of mode-id, which immediately follow the 'U' conversion code letter. 'Unxxx' means entry point 'n' (0-F) of frame 'xxx' (1-FFF, which is 1-4095 in decimal). For more information on BASIC, Recall, and PROC, please see the appropriate reference manual.

Due to the mode-id format, assembly programs must be loaded into frames 1-4095, with up to 16 entry points. The actual number of frames may be less, depending on the operating system release. Frames

above 1023, especially, are typically used for purposes other than assembly programming.

In assembly language programming, when a program needs to branch to an entry point in another frame, a symbol should be predefined as a mode-id that points to the desired entry point in the desired frame. If a symbol already exists in the PSYM file which defines this mode-id, then that symbol may be used. Otherwise, both the entry point and FID of the mode-id should be explicitly specified in the calling program.

A mode-id may be defined in two ways:

- DEFM directive (defines a symbol; no object code)
- MTLY or MTLYU directive (defines a symbol and reserves storage, word-aligned only if MTLY)

The DEFM method may be used to simply define a synonym for a location already allocated storage (or that will be allocated storage before the program calls it). For example, the following defines the symbol EXT.SUB as a mode-id whose value is entry point 4 in frame 500:

```
EXT.SUB   DEFM  4,500
```

EXT.SUB may then be used as an operand in instructions such as the following:

```
          BSL   EXT.SUB       Call external subroutine
          ENT   EXT.SUB       Branch with no return
```

The MTLY directive should be used when storage needs to be reserved. MTLY and MTLYU are less frequently used, except when constructing tables of mode-ids. For example:

```
EXT.SUB   MTLY 4,500
    .
    .
          LOAD  EXT.SUB       Get mode-id in accumulator
          BSLI  *             Call subroutine referenced
          CMNT  *             by accumulator
```

Program Line Structure

A source line may contain up to five fields of information:

- label field
- source code operation field (opcode mnemonic)
- source code operand field
- comment field
- object code generated by assembler

Label field

The optional label, if present, must begin in column 1 of an input line and must begin with an alphanumeric character. Labels may be up to 50 characters in length, although only 10 columns are reserved for the format on an assembly listing.

Labels should not contain an asterisk (*), a slash (/), or a plus sign (+). A label is separated from the opcode mnemonic by a space.

Labels are locally defined symbols used to address locations in the program, or to define other symbol types. A label must be used as the target of all branch instructions (conditional or unconditional). Examples are:

```
LOOP
!STARTIT
TOTAL-X
TEST123
```

Opcode field

The opcode is separated from the label and the operands by at least one space. If there is no label, at least one space must precede the opcode.

Opcodes may be primitive or macro instructions, or directives. They consist of the opcode mnemonic and usually one or more operands. Examples of mnemonics are:

```
MOV
```

```
INC
BSL
```

The valid opcodes are described in Chapter 4.

Operand field

The operands are separated from the opcode by at least one space. Multiple operands are separated by commas, and no spaces are allowed within the field (except in quoted character literals). Operands may be literals, symbols, or the current location counter, using the forms shown below:

Form	Description

C'xxxx'	Text string; example:
	`C'NOT AGAIN'`
	If a single quote (') is needed as a literal, two adjacent single quotes must be used; example: for JOE'S, use the operand
	`C'JOE''S'`
	For just a single quote, use
	`C''''`

n	Decimal integer; examples: 120 or -42
X'xxxx'	Hexadecimal constant; example: `X'FE'` or `X'8100FF'` If an odd number of hex characters is used, a leading zero is assumed to fill the leftmost nibble
symbol	Symbol name predefined in the PSYM file or defined in the label field of the source program
*	Current byte location in frame; uses the assembler program location counter to return the first byte of the current location or address being assembled
*n	Current location in units of 'n' bits; examples: *1 = loc. in bits; *8 = *; *16 = loc. in words This location counter advances as instructions are assembled; the counter can be altered only via an Origin (ORG) directive.

literals +/- loc	Literals or * locations combined with a plus (+) or minus (-). Symbols cannot be used here; examples:
	`*+2`
	`*-1`
	`-1+ID.ABSFRM.SIZE`

Comments field

The optional comments field follows the last operand, separated by at least one space, and may be of any length.

Object Code field

The first four columns of the object code field contain the byte offset (displacement) in the frame, followed by a space, followed by the actual object code. The object code is separated from the source code by a subvalue mark, placed there by the assembler.

Displaying Assembly Programs in the Editor

The line editor has three commands that can be used to display assembly language programs:

AS displays source code in pre-sized fields

M displays macro expansions

S suppresses object code (if any) in object field

Source code lines may be displayed on the screen with all fields shown when the Editor is used with the assembly listing switches AS and M turned on and S turned off.

If both the AS switch and the S switch are off, each line is displayed as entered. Macro expansions and error messages, if any, follow the source code and are separated from it by value marks. Object code, if any, follows the source code and any macro expansion code; it is separated by sub-value marks. For example,

```
013  SAVE MCC R4,R5 MOVE THE TERMINATOR\0056 645D
014  MCC R4,R16 SAVE IT ALSO]*ERR: REF:UDEF, REF:UDEF
015  B OK] B: OK\0058 1E45
```

When AS is on, the assembly listing format is as follows:

Col	Field description
1-15	object code
16	blank
17-25	label field; contains one of the following: label * (comment line) (null) neither label nor comment
26	blank
27-31	opcode field
32	blank
33-49	operand field
50	blank
51-75	comment field

The following example shows a program in the editor with AS on, but with S and M off (the editor item line numbers are shown to the left of the program line itself).

```
column:      1         2         3         4         5  ...
    12345678901234567890123456789012345678901234567890123...

001 0001 7FF001D7       FRAME 471
002                *SAVE/RESTORE
003                *24 APR 1990
...
013 0000                ORG   0
014 0000 FE             CHR   AM
015                AM   EQU   R1
016                *
017 0001 1E27      0    EP    !LOG
018 0003 1E38      1    EP    !CMDLOOP
...
073 0028           !LOG EQU   *
074 0028 A00200         ZERO  PRMPCH
...
085 0049 1172B2         B     CMD200
...
243 01CC 0309           END
```

If S is on (suppress object code), lines 13-18 would list as:

```
          1         2         3         4         5  ...
 12345678901234567890123456789012345678901234567890123...
013         ORG    0
014         CHR    AM
015 AM      EQU    R1
016 *
017 0       EP     !LOG
018 1       EP     !CMDLOOP
```

If M is also on (display macro expansions), line 85 would list as:

```
085         B      CMD200
            +B:    CMD200
```

The Assembler Program

The assembler translates source code statements into object code. The source mode may be stored as an item in any file. In firmware implementions, the object code is assembled in place; that is, at the conclusion of the assembly process, the item contains both the original source code and the generated object code. In software implementations the destination of the object code must be specified; it can be a separate file or it can be in the current file.

The assembled object code must be less than or equal to one ABS frame in size. On all machines the operative frame size is stored in the PSYM file as the symbol ID.ABSFRM.SIZE; on firmware machines, this is 512 bytes.

Each implementation has its own version of the assembler and is invoked as follows:

firmware systems	use the AS verb
S/370 systems	use the OPT verb.
1400 systems	use the ASM verb

When a program is assembled, the generated object code is stored along with the source statement and system delimiters are used to separate the components on each line. On firmware machines, the object code is stored back into the source file. On 1400 and S/370 systems, it is stored a separate file. On a firmware system, while you are editing an already assembled program, you can ignore any data beyond the source statement, because the assembler examines only the source data on each line as it performs the assembly; any existing object code and other characters are discarded.

Object code and associated addresses are stored as hexadecimal digits in ASCII character format. These are converted to binary values when the program is loaded.

Listing Assembled Programs

The following system commands can be used to generate listings using an assembled program item:

MLIST generates a formatted listing

MLOAD loads the program for execution

MVERIFY verifies the loaded code

CROSS-INDEX generates concordance listings

X-REF generates a cross reference by symbol name

XREF enhanced version of X-REF

These system commands are described in in this chapter in the section, Assembly Program Listings.

Executing Assembled Programs

An assembled program is not automatically ready to execute. In order to run an assembled program, you must create a verb definition item in the account's Master Dictionary (MD), or call the program from BASIC, PROC, Recall, or another assembly language program.

The following interfaces can be used between user-written programs and the Ultimate operating system. Each interface is designed for a particular function or type of program.

Interface	Function
CONV	For subroutine calls from BASIC or Recall. Used when a conversion needs to be performed.
PROC	For routines called from PROC.
RECALL	For verbs that use Recall's data base reporting capabilities.
TCL-I	For verbs that use the TCL-I form (no filename)
TCL-II	For verbs that use TCL-II form (filename).
WRAPUP	For exiting verbs, or anywhere if a program may exit on an error condition.
XMODE	For handling Forward Link Zero register conditions (that is, to add frames to a linked set during program operation).

When an assembled program is ready for production, the appropriate interface must be selected and programmed. Most user-written programs use the TCL-I, TCL-II, or the CONV interfaces. The TCL interfaces involve defining the program as a verb in the MD. Once the verb definition is stored, the program can be executed by entering the verb name at the TCLor specifying the name anywhere a system command is valid.

All interfaces are described in Chapter 6, System Software Interfaces.

The AS Command - Firmware Assemblies

The AS command is used to assemble programs for a firmware machine.

Syntax

AS filename {itemlist} {(options)}

filename name of file that contains items to be assembled

itemlist names of items to assemble; may be one or more explicit item-IDs, or an asterisk (*) to specify all items in the file; may be omitted if a select-list is active

(options the following options are available:

- **E** when used in conjunction with the **L** option, lists only errors
- **L** generates a listing equivalent to the MLIST command during assembly
- **N** inhibits waiting at end-of-page during listing to terminal; useful in conjunction with **Z** option
- **P** routes output to print spooler
- **Q** specifies that messages are not to be displayed nor the editor entered if assembly errors are found; normally, this is used when multiple items are being assembled
- **Z** specifies that, if assembly errors are found, the editor is not to be entered; normally, this is used when multiple items are being assembled

Description

The AS command requires three files to be defined on the user's account:

- OSYM opcode symbol file; contains all the opcodes and valid symbol types for each opcode
- PSYM permanent symbol file; contains the global symbols available to all assembly language programs
- TSYM temporary symbol file; used by the assembler to store the symbols used in the mode currently being assembled

OSYM and PSYM are typically Q-pointers to the Ultimate-supplied OSYM and PSYM files, but TSYM must be created for each account. For more information on the symbol files, see the section, Symbol Files.

Only one user at a time in an account can use the AS command.

The AS command is table driven and performs two passes over the source code. During the first pass, all instructions that have undefined and forward references are flagged as requiring re-assembly. Local labels are stored in the temporary symbol file (TSYM) during this first pass, along with the literal definitions that need to be created.

As the assembler processes items, it outputs an asterisk (*) after every ten source statements are assembled. At the end of the first pass, the literals are generated and added to the end of the current object code.

On pass two, a new line is started and an asterisk is printed for each ten statements reassembled.

If there are any assembly errors, the assembler enters the editor so that the program may be conveniently corrected for reassembly (unless suppressed by the Q or Z option).

If there are no errors, the following message is displayed (unless the Q or Z option is used):

```
[236] No errors
```

The AS command is table driven and performs two source code passes:

1. In the first pass, all instructions haveing undefined and forward references are flagged as requiring re-assembly. Local labels are stored in the temporary symbol file (TSYM), along with the literal definitions that need to be created. At the end of the first pass, the literals are generated and added to the end of the current object code. As the Assembler processes items, it outputs an asterisk (*) after every 10 source statements assembled.
2. In pass two, a new line is started and an asterisk is printed for each 10 statements reassembled.

Assembly errors cause the Editor to be entered for program correcttion for reassembly (unless suppressed by the Q or Z option). If no errors, the following message displays (unless the Q or Z option is used):

```
[236] No errors
```

Assembler error messages are stored as part of the source line in error. Undefined symbols are stored as a message list in the last line of source. Assembler error messages are explained below.

Message	Description
OPCD?	opcode mnemonic is missing
OPRND REQD	instruction is missing at least one operand
ILGL OPCD: *xxxx*	either the opcode mnemonic, or operands specified are not valid for this opcode
LBL REQD	an Equate (EQU) directive does not contain a symbol in the label field, so there is nothing to equate the value to
MUL-DEF	label is defined more than once
OPRND DEF	either the operand is defined improperly or is not valid for this instruction
OPRND RNGE	operand's numeric value is not within the valid range for this instruction
REF: UDEF	instruction references an undefined symbol
TRUNC	an operand is out of range. Typically this error occurs when a program exceeds the size of a frame and an instruction tries to reference an Assembler-generated literal beyond the last location of an ABS frame
UNDEF: *xxx* {,*xxx*..}	list of undefined symbols found

```
:AS  SM  PROG1
******************          (pass 1 output from assembler)
***                         (pass 2 output from assembler)
[236] No errors
```

The OPT Command - S/370 Assemblies

The OPT command is used to be assemble a program for S/370 implementations of the Ultimate Operating System. The command itself is a cataloged BASIC program and is included in the SYSPROG account.

Syntax

OPT filename {itemlist} {(L}

filename name of file that contains items to be assembled

itemlist names of items to assemble; may be one or more explicit item-IDs, or an asterisk (*) to specify all items in the file; may be omitted if a select-list is active

(L generates an instruction that allows a BREAK at each label.

Note: Once a program has been debugged, it should be assembled without the L option in order to run more efficiently.

Description

OPT requires the following symbol files to be defined on the account doing the assembly:

I1.PSYM permanent symbol file used in pass 1 of the assembly

I1.OSYM opcode symbol file used in pass 1 of the assembly

IPSYM permanent symbol file used in pass 2 of the assembly

IOSYM opcode symbol file used in pass 2 of the assembly

ISM file used by the assembler to store assembled object code; must be created by user

TSYM temporary symbol file; used by the assembler to store the symbols used in the mode currently being assembled. This file must be created with a data section modulo of 31.

The I1.PSYM, I1.OSYM, IPSYM and IOSYM files are delivered on the SYSPROG account. To assemble from another account, Q-pointers should be set to the file in the SYSPROG account.

The OPT command uses the following two verbs, which must be defined on the account doing the assembly:

AS.IBM

XP

The OPT version of the assembler makes two passes. Pass 1 converts the Ultimate source code to S/370 source code. Pass 2 assembles the S/370 source code into S/370 object code.

When the assembly is complete, the following message is displayed:

```
[206] 'itemname' assembled
```

The object code is stored in the ISM file under the item name used in the assembly. If there were errors or undefined references, these are also stored in the item in the ISM file.

Each assembled item should be edited to determine if errors exist. The following shows an assembled item with errors:

```
          .
          .
*ERR    MOV BMS,15
*ERR:   ILGL OPCD:  MOV:RN
```

The ASM Command - 1400 Assemblies

ASM is used to assemble a program for 1400 implementations of the Ultimate Operating System. The command itself is a cataloged BASIC program.

Syntax

ASM filename {item-ist} {(options)}

filename name of file that contains items to be assembled

itemlist names of items to assemble; may be one or more explicit item-IDs, or an asterisk (*) to specify all items in the file; may be omitted if a select-list is active

(options the following options are available:

- **C** retains comment lines from the source code instead of suppressing them from the assembled program. By default, ASM deletes comment lines in the source, and converts source code into comment lines; the assembled code is indented beneath the source code that generated it. With the C option, a comment line is converted as a comment with a greater-than sign (that is, "*> comment-text").
- **E** when used in conjunction with the **L** option, lists only errors
- **L** generates a listing equivalent to the MLIST command during assembly.
- **N** inhibits waiting at end-of-page during listing to terminal; useful in conjunction with **Z** option.
- **P** routes output to print spooler.
- **Q** specifies that messages are not to be displayed nor the editor entered if assembly errors are found; normally, this is used when multiple items are being assembled
- **V** inserts a V.TRAP instruction into the native code before each source instruction instead of just at labels. By default, ASM inserts a V.TRAP only at a label, to enable single-step debugging with the debugger E1 command. With the V option, the single-step is from the source's instruction to instruction instead of label to label.

Note: *Once a program has been debugged, it should be assembled without the V option in order to run efficiently.*

Z specifies that the editor is not to be entered if assembly errors are found; normally, this is used when multiple items are being assembled

Description

ASM requires the following symbol files to be defined on the user's account:

M1.OSYM	opcode symbol file; used by pass 1 of the assembler program to convert Ultimate assembler code to 1400 assembler code
M1.PSYM	permanent symbol file; contains all predefined symbols available to the assembly language programmer
M2.OSYM	opcode symbol file; used by pass 3 of the assembler program to convert 1400 assembler code to object code
M2.PSYM	permanent symbol file; used by the assembler program
OPT.ERRORS	optimizer errors; used by optimizer to store errors encountered during pass 2
TSYM	temporary symbol file; used by the assembler to store the symbols used in the mode currently being assembled

The symbol files are described in the section, Symbol Files.

The OPT command uses the following two verbs, which must be defined on the account doing the assembly:

ASM1

ASM2

When ASM is invoked, it first prompts for a destination file name:

```
To: {(filename} {item-list}
```

The response to the To: prompt may be a filename or item-IDs or both; pressing RETURN with no response cancels the ASM command. All destination file/item name forms that are valid for a COPY command may be used. For example:

```
ASM BP *
To:(BD
```

assembles all items in the BP source file to the BD file using the same item-IDs as in BP. Another example:

```
ASM SSM T
To:T.OBJ
```

assembles one item in SSM to the same file as item 'T.OBJ'.

After a valid destination file has been specified, the ASM command starts the assembly.

ASM uses three passes:

1. Executes the ASM1 verb, which assembles virtual code into native machine source code, using M1.OSYM and M1.PSYM.

2. Executes a BASIC program in the SYSPROG-PL file called OPT. The symbol files are used to construct the destination file items. Any errors encountered are logged in the OPT.ERRORS file, described below. The optimized items are output to the specified destination file.

3. Executes the ASM2 verb, which assembles the object code, using the output of the optimizer and the M2.PSYM and M2.OSYM symbol files. The items are updated in the destination file.

This sequence is looped through for all items in the list.

The Optimizer

Object code for an assembled item must fit into one ABS frame. If the object code generated by the optimizer in pass 2 does not fit in an ABS frame in the first try, the optimizer reassembles the code using a compression algorithm. Level 0 is "no compression"; level 9 is "maximum compression". The higher the compression level, the smaller but less efficient the resultant code. The optimizer tries up to ten levels of compression; if it reaches level 10 without fitting the object code into a frame, it gives up and goes on to pass 3.

Each level of compression specifies which instructions are to be compressed by the optimizer. An instruction is compressed by moving most of its code to the Kernel, leaving only code to set up a call to the Kernel in the assembled object code.

The level of compression for a program is stored in attribute 2.of the destination item in the following format:

```
002 * compression level = n
```

n number of iterations used by the Optimizer to get the object code to fit into an ABS frame

If the destination item already exists, the value in attribute 2 is used as the compression level for the program being assembled. The Optimizer does not reduce the compression level value previously stored. This means that even if code has been removed to make the program smaller, the optimizer starts the assembly at the previous level of compression. To overcome this restriction, you should either delete the old destination item, or edit the old item and set the compression level back to 0 before reassembling the program.

The OPT.ERRORS File

This file contains an item for each source file item that has been assembled. OPT.ERRORS stores the time and date of the assembly in the item, as well as any errors that the optimizer found while processing the source file item.

Assembler Error Messages

Assembly error messages are stored as part of the source line in error. If undefined symbols exist, a list of these symbols is stored as a message in the last line of source. If any assembly errors are found, the Editor is called as a convenient way to edit the source item, unless the Q or Z option was specified with the assembler command.

Message	Description
OPCD?	The opcode mnemonic is missing.
OPRND REQD	The instruction is missing at least one operand.
ILGL OPCD: xxxx	Either the opcode mnemonic is not valid, or the operands specified are not valid for this opcode.
LBL REQD	An Equate (EQU) directive does not contain a symbol in the label field, so there is nothing to equate the value to.
MUL-DEF	The label is defined more than once.
OPRND DEF	Either the operand is defined improperly or is of an invalid type for this instruction.
OPRND RNGE	The operand's numeric value is not within the valid range for this instruction.
REF: UDEF	The instruction references an undefined symbol.
TRUNC	An operand is out of range; typically this occurs when a program exceeds the size of a frame and an instruction tries to reference an assembler-generated literal beyond the last location of an abs frame.
UNDEF: xxx {,xxx..}	List of undefined symbols found.

OSYM Errors

The following error messages are issued when the assembler detects errors in the OSYM file definitions:

```
FRMT. A-FIELD
FRMT. B-FIELD
OPCD TYP
MACRO DEF
# MOD WORDSIZE  > 32 BITS
EXIT DEFN
```

To correct the OSYM file, perform a selective restore (SEL-RESTORE command) of the OSYM file using the latest SYSGEN tape.

Generating Object Code

The output of the assembly procedure is object code that the Ultimate machine can directly execute. The actual object code on Ultimate software implementations depends on the native code of the system; however, all firmware machines generate the same object code.

The assembly procedure performs two distinct tasks on source code, determined by the type of operation:

- **directives** are processed to set up the program structure and generate object code where needed
- **instructions** are assembled into object code

Directives and Object Code

Directives do not generate executable code. They may, however, generate object code in the sense that symbol definitions may reserve space in the program frame and may also assign a value which is in "object" format.

The following directives do generate object in that sense:

ADDR	generates a 6-byte storage register*
ALIGN	may generate 1 byte (0)
DTLY	generates a 4-byte double tally*
FTLY	generates a 6-byte full tally*
HTLY	generates a 1-byte half tally
MTLY	generates a 2-byte tally with even-byte alignment*
MTLYU	generates a 2-byte tally without alignment
SR	generates a 6-byte storage register*
TEXT	generates the number of bytes in specified string
TLY	generates a 2-byte tally*

Instructions and Object Code

Instructions normally generate executable code. Each source code instruction is assembled into 1-6 bytes of object code that can be directly executed by the operating system.

---

* may first generate a byte of 0 to align the operand on a word (tally) boundary

The first byte of the object code for all instructions is the primary opcode (1 byte). In addition, depending on whatever is necessary to access the specified data and perform the specified operation, the object code may have up to 5 more bytes for secondary opcodes, address registers, byte addresses of relative operands, a code for the type of symbol used as an operand, immediate data, and/or the offset of a local label.

The primary opcode is the only byte that is generated for all instructions. The other parameters may or may not be applicable to a particular instruction. Chapter 4 discusses each instruction in alphabetical order.

Generating Object Code

In order to generate directive object where needed, the assembler interprets the directive and converts the value to hexadecimal for storage in the frame. The object is stored at the current program counter location. If a symbol is locally defined (that is, it is not in the PSYM file), it is added to the TSYM file during the assembly procedure.

To generate instruction object, the assembler searches the OSYM file for the particular instruction form and uses the primitive layout(s) to convert the source to object code in the frame. The object is stored at the current program counter location in the frame.

Symbol Files

The Ultimate system assemblers use several symbol files in assembling a source program. The file types and names for each of the implementations are given in Table 2-1.

Each file performs a different function during program assembly.

The permanent and opcode symbol files must be defined in the master dictionary (MD) of the user account. These may be actual files in the account, but usually they are Q-pointers to the files supplied in the SYSPROG account. TSYM, however, must be an actual file defined in the MD of any user account that uses the assembler.

Table 2-1. Symbol Files

File Type	Firmware	1400	S/370
permanent symbols	PSYM	M1.PSYM M2.PSYM	I1.PSYM IPSYM
temporary symbols	TSYM	TSYM	TSYM
opcode symbols	OSYM	M1.OSYM M2.OSYM	I1.OSYM IOSYM

The PSYM File Layout

The permanent symbol files contain the set of permanent or global symbols available to all assembly programs. While symbols in these files may be redefined locally in a program, it is best to treat them all as reserved.

The item-ID of a permanent symbol file entry is the symbol name. Attribute 1 of each symbol item has a symbol type code, which the assembler uses to determine the amount of space to assign for the symbol. Table 2-2 lists the symbol type codes and storage allocation. The specific format of each symbol type is shown in the Table 2-3. Values are in hexadecimal.

Table 2-2. Symbol Type Codes and Storage Allocation

Symbol Type	Name	Storage Allocation
B	bit	1 bit
C	character	1 byte
D	double tally	4 bytes
F	triple tally	6 bytes
H	half tally	1 byte
L	label	(none)
M	mode entry point	(none)
N	literal number	variable
R	address register	(none)
S	storage register	6 bytes
T	tally	2 bytes
X	external address register	8 bytes

Table 2-3. Format of Symbol File Item

Attribute	Description			
item-ID	symbol name	symbol name	symbol name	symbol name
001 (symbol)	M	N	R	all other symbols
002	entry point number	literal value	register number	offset
003	frame number	not used	not used	base register

The TSYM File Layout

The TSYM file is used by the assembler to hold the set of symbols in the program currently being assembled. It is always cleared by the assembler before the start of each assembly.

As the assembler finds labels and symbols in the source program, it stores the label in the TSYM file for future use. If a reference is made to an undefined symbol, it is also stored in the TSYM file. Undefined symbols are converted to defined symbols if they are later found in the label field of a source statement. If not used, an undefined symbol is reported as an assembly error.

The format of the entries in the TSYM file is identical to that of entries in the PSYM file.

A symbol in the TSYM file overrides a corresponding symbol in the PSYM file; that is, local definitions override global ones.

The TSYM file cannot be shared. Therefore, only one user at a time can use the assembler on an account. Each account should have its own TSYM file, and not a Q-pointer to another account's TSYM.

The modulo of the data section of the TSYM file must be 31, due to the method the assembler uses in generating literals. If a program is loaded and then reassembled with a different TSYM modulo it will not MVERIFY, even though the source statements are identical.

The OSYM File Layout

The opcode files contain the set of Ultimate opcode mnemonics.

The item-ID of an entry in one of these files has one of two forms:

- the opcode mnemonic itself; for example, B for branch.
- the opcode mnemonic concatenated with the symbol type of each operand. For example, MOV:RR (move register to register) and MOV:SR (move storage register to register)

The second form is used to distinguish different opcode-operand combinations, which may generate completely different machine instructions, as well as to validate the operands used in the instruction. For example, the MOV opcode with operands of types B and H would result in an OSYM file lookup of MOV:BH, which is nonexistent and therefore invalid.

An item in the opcode files has two or more attributes:

Attribute	Description
001	type of instruction; valid codes are P primitive; the following lines in the item are used to generate object code or perform other symbol manipulation functions. M macro; each succeeding line in the item is used to generate a new source line that is in turn assembled just as any source line. Q synonym; the following line in the item is used as an item-ID to continue processing. This is used to "link" from one item to another to save duplicate definitions.
002 and on	assembly operation appropriate for type; may be • primitive instruction layout (attribute 1 = P) • list of component instructions (attribute 1 = M) • synonym item-ID (attribute 1 = Q)

Primitive Instructions

Each primitive entry in an opcode file contains a definition for generating object code from the source statement. The definition is divided into argument fields, where each argument defines the object code for that particular component of the instruction.

The term "argument field" (AF) refers to the fields in the original source statement being assembled as follows:

label AF(0)

opcode AF(1)

operands AF(2) through AF(9), if they exist.

For example, in the following source statement:

```
LOOP    BCE   R11,C'A',STOPIT
```

the AF values are:

AF(0) LOOP

AF(1) BCE

AF(2) R11

AF(3) C'A'

AF(4) STOPIT

Each line in a primitive OSYM definition has one of the following formats:

Entry	Description
G,a1,a2,... b1,b2,...	generates object code. There is a one-to-one correspondence between the a1, a2, etc., and the b1, b2, etc. There is one blank space between the 'a' and 'b' fields. The 'a's are bit counts, and refer to the size in bits of the object code to be generated by the corresponding 'b' expressions. The sum of the 'a' fields must be a multiple of 8, and must be in the range 8-32. Valid b field expressions are given in Table 2-4.
R,a1 b1,b2...	redefines a TSYM file entry. The TSYM file item is referenced using AF(a1) (normally, a1 is zero, to reference the label field of the source statement). Successive lines in the TSYM file entry are replaced with the data generated by the expressions b1, b2, ...
E:xxxx	specifies an exit to an assembly subroutine whose mode-id is xxxx.
Q opcode	transfers control to OSYM entry specified by opcode. There is one blank space between the Q and the opcode name.
O text	generates the specified text as source code, in the macro expansion portion of the statement. This is used in assembling programs on 1400 systems (ASM command).
*comment	used to include comments in OSYM entries.

Table 2-4. Expressions to Generate Object Code
('b' Field Expressions)

Code	Description
n	decimal constant.
X'nn'	hexadecimal constant.
=c	single byte character constant.
*{n}	current location counter, where the optional n is 1 for location in bits, 8 in bytes, 16 in words
An;m	references AF(n); if a symbol, returns the value from the 'm'th line of the PSYM/TSYM file definition; if a literal constant, returns the value of the literal.
B	Current base register (see literals below).
E:xxxx	Exit to assembly subroutine whose mode-id is xxxx.
Jn	Returns branch (or jump) address of local label referenced by AF(n).

The 'b' field expressions may be composed of sub-expressions joined with the following operators:

+	addition
-	subtraction
*	multiplication
/	division (integer)
&	logical AND
!	logical OR
Rn-n	lower-upper range limit on previous expression
Uxxxx	assembly subroutine call (mode-id: xxxx) after evaluating previous expression

(...) enclose expression in parentheses to alter expression evaluation

The precedence of the operators is as follows:

1 expressions within parentheses are evaluated

2 R and U operators

3 & and ! operators

4 * and / operators

5 + and - operators.

Operators with the same precedence in an expression are applied left to right; for example:

`A2;2-*`	difference between value of AF(2) and current location
`A4;2R0-3`	value of AF(4) or assembly error if this is not 0, 1, 2, or 3
`(A2;2+1)/2`	half of one more than the value of AF(2); remainder from division is discarded

Macro Definitions

A macro definition has the code M in attribute 1 of the OSYM file item. Each succeeding line generates a new line of source. All text in the macro definition is literal and copied without change, except for the following:

Text	Description
(n)	references AF(n), which is copied to the source line.
(*)	references all AF entries, starting with AF(2); this may be used to copy all references to the macro-generated source line.
(L), (L+n) or (L-n)	If present in the label field of the macro-generated statement, this creates a unique label by incrementing the macro's internal label count, and storing that as the generated label. The +n and -n forms are not allowed here. If not in the label field, the current internal label count, modified by the +n or -n, is used to generate a label.

The following example explains how a macro is created. Suppose a new instruction which tests a signed integer to see if it is in a specified range is to be created, using the following syntax:

```
RANGE x,low,high,label
```

x	signed integer
low	minimum value
high	maximum value
label	label to branch to if x is in range

An example of this instruction, its OSYM macro definition, and the generated code would be:

```
RANGE CTR0,CTR1,CTR2,INRANGE
```

OSYM file format	Generated source code (assume macro label count = 14 at start)

```
RANGE:TTTL
001 M
002  BL (2),(3),(L+1)          BL CTR0,CTR1,=L15
003  BLE (2),(4),(5)           BLE CTR0,CTR2,INRANGE
004 (L) EQU *             =L15  EQU *
```

Note that (L) is in the label field because no space precedes the "(".

```
MCI SC0,R11                    Original source line

SC0     R11                    PSYM file entries
001 C   001 R
002 3   002 00B
003 0   003 B

MCICR                          OSYM file entry
  001 M
  002 INC (3)
  003 MCC (2),(3)

+INCR R11                      Resulting macro source statements
+MCCCR SC0,R11
```

```
MCC SC0,R11                          Source line

SC0         R11                      PSYM file entries
001 C       001 R
002 3       002 00B
003 0       003 B

MCCCR                                OSYM file entry
001 P
002 G,4,4,8,4,4 13,A2;3,A2;2,1,A3;2
```

Object code generation:

a-field	b-field expression	symbol ref	result
4	13	-	D
4	A2;3	SC0	0
8	A2;2	SC0	03
4	1	-	1
4	A3;2	R11	B

```
D0031B                               Final result
```

```
NEW       DEFH R15,5                Source line

 R15                                PSYM file entries
 001 R
 002 00F
 003 F

 DEFHRN                             OSYM file entry
 001 P
 002 R,0 =H,A3;2,A2;2
 003 E:5019

 NEW                                TSYM file entry after Pass 1
 001 L                              symbol NEW is stored as type L
 002 xxxxxxxx                       offset equal to the current location
 003 1                              base register of 1

 NEW                                TSYM file entry after Pass 2
 001 H
 002 5
 003 4
```

Symbols and Literals

A symbol is a named reference to one of the fields that can be addressed by the system. Symbols can be defined in the following ways:

- a globally defined symbol, stored in PSYM
- a locally defined symbol; one that appears in the label field of the current program
- a shared symbol; one that appears in the label field of a program that is named in an INCLUDE assembler directive in the current program.
- an immediate symbol; one that is explicitly stated in the instruction.

The symbol name is of the same format and has the same restrictions as a label field.

A symbol name should not begin with one of the following characters:

$ dollar sign
# pound or number sign
!! double exclamation mark

Certain symbols that start with these characters are used by the kernel on some systems. To avoid possible conflict, select symbol names that do not begin with these characters.

If you attempt to assemble a program whose code includes a definition of a symbol used by the kernel on 1400 systems, the ASM command displays the following message as a warning to change the specified symbol name to a different "safe" name:

```
Redefinition of symbol used by kernel: symbol
```

Locally Defined Symbols

To define a symbol in the program for local usage, use one of the DEF directives. To reserve storage in the object code, use one of the TLY type directives.

For example, the following instruction defines CNTER as a symbol of type T, with a specific base register of 4 and an offset of 5:

```
CNTER   DEFT R4,5
```

However, the following instruction defines it implicitly at the current location in the object code, and stores a value of 1234 at that location in the object code:

```
CNTER   TLY  1234
```

This symbol is now a literal or constant in the program.

Literals

The assembler automatically assembles certain types of literals. Such literals are fields that can be addressed using a base register and an offset displacement. When a program is executing, address register 1 (R1) points to byte zero (0) of the frame. Therefore, this may be used by the assembler as the default base register to address literal fields that it creates and stores in the frame.

Symbols of types T and D can be automatically generated as part of an instruction, but types H and F cannot. This is because half tallies (H) can only be offset up to 255 bytes from the base register's address, and literals are only generated at the end of the object code. If the object code is greater than 255 bytes, half tally literals would cause a truncation error. F-type (triple) tallies cannot be generated automatically due to an assembler limitation. If a program needs to use half or F-type tally literals, they must be defined explicitly with the HTLY or FTLY instructions.

In addition, in order for the assembler to generate a literal, the instruction must be a macro. The instruction itself should simply specify the literal value (for example., ADD 3); the macro uses the following form to generate the symbol:

```
=x(AFn)
```

where

x D or T

AFn number of argument field in the instruction that contains the literal value

For example, to generate a tally of the value in argument field 2, the assemble sets up the following:

```
=T(2)
```

The assembler stores this symbol (if not already present) as an undefined type in the TSYM file. At the end of pass one, the TSYM is searched sequentially for undefined symbols that match the above pattern, and the literals are assembled. This is done by internally generating source statements using special opcodes of the form ":x" (:D, :T, etc.), which actually generate the literal and redefine the symbol to the correct type and location.

The literal thus generated at the end of the program has the following form:

```
=xvalue   :x    value
```

For example, the following generates a tally with the literal value 3:

```
=T3       :T    3
```

The following is step-by-step example of literal generation on a firmware implementation:

Step 1

MOV 100,COUNTER — Source line

PSYM file entry:

item-ID	COUNTER
001	D
002	1F
003	0

OSYM file entry

item-ID	MOVND
001	M
002	MOV =D(2),(3)

Step 2

MOV =D100,COUNTER — Resulting macro source statements

TSYM file entry

item-ID	=D100
001	U
002	0
003	1

OSYM file entry

item-ID	MOVDD
001	P
002	G,4,4,8,4,4,8 15,A3;3,A3;2,8,A2;3,A2;2

Step 3

At the end of pass one, an internal source statement is assembled:

=D100 :D 100	Source line

OSYM file entry

item-ID	:D	
001	E:101B	
002	1F	Forces word alignment in object code
003	R,0 =D,*16,B	
004	G,32 A2;2	Generates double tally object code

00000064	Resulting object code

TSYM file entry:

Before instruction		After instruction	
item-ID	=D100	item-ID	=D100
001	U	001	D
002	0	002	xxx
003	1	003	1

xxx offset appropriate to the current location.

Step 4

The MOV 100,COUNTER instruction is reassembled on pass two.

Shared Symbols (INCLUDE Directive)

The main reason for the INCLUDE directive is to be able to place a set of shared definitions in one item and then use the definitions in any other program. Typically, variables and mode-ids that are local to a set of programs are placed in a single program for inclusion during assembly. The advantage of this method is that the definitions are not duplicated in every program that uses them. Such duplicate definitions can lead to errors and are in general more difficult to maintain than if they were all in one program.

The format of the INCLUDEd program is identical to that of any other program, though typically it consists of only DEFx (definition) assembler directives.

Immediate Symbols

Normally, a symbol must be in PSYM or must appear as an entry in the label field of the program or in an included program.

In some instructions, however, an *immediate symbol* may be defined as an operand. This may be useful when a symbol is only used once; it may be simpler than having to define the symbol in a separate line. However, because these symbols have a quirk in their syntax that makes them different from the PSYM/TSYM equivalents, they are not recommended except to reference bits. They are documented here for compatibility only.

The general form of an immediate symbol is:

```
Rn;xd
```

Rn address register R0-R15

x symbol type (B, C, D, F, H, S,or T)

m decimal value that generates the offset displacement

The offset displacement is equal to m * field.length

In other words, m is the displacement in units of immediate symbols. For example, the immediate symbol R0;B32 addresses bit 32 displaced from R0; and R2;T10 addresses the tally displaced from R2 at bytes 20 and 21 (same as PSYM/TSYM entries). However, R2;D10 addresses the double tally displaced from R2 at bytes 40 through 43 (not 20 through 23, as generated by PSYM/TSYM entries).

Following are examples of immediate symbols and their equivalent DEF instructions (see the DEFx directive in Chapter 4 for a full discussion).

Immediate Symbol	Displacement from Base Register		Equivalent DEF Instruction
R0;B0	0	HIBIT	DEFB R0,0
R15;B7	7	LOBIT	DEFB R15,7
R2;C100	100	CHARACTER	DEFC R2,100
R15;T10	20-21	TALLY	DEFT R15,10
R0;D10	40-43	DTALLY	DEFD R0,20
R0;S10	60-65	STORAGE	DEFS R0,30
R0;F15	90-95	F-TALLY	DEFF R0,45

Assembler System Commands

After an assembly language program has been assembled, a number of system (TCL) commands are available to bring the program up to a production mode of operation.

CROSS-INDEX

The CROSS-INDEX command creates a cross-reference of all symbols used in an assembly language program or set of programs.

Syntax

CROSS-INDEX filename {itemlist} {(F)}

filename name of file that contains items to be indexed

itemlist names of items to index; may be one or more explicit item-IDs, or an asterisk (*) to specify all items in the file; may be omitted if a select-list is active

(F prompts for the name of a symbol file to use instead of PSYM; if not specified, the symbols are searched for in PSYM

Description

CROSS-INDEX checks each program in the specified file and builds an item in the CSYM file. (The CSYM file must already exist).

The name of the program is used as the item-ID in the corresponding item in the CSYM file. Each attribute in the item contains information about one type of symbol. The item has the following format:

AMC	symbol type	
1	B	bits
2	C	characters
3	H	half tallies
4	T	tallies
5	D	double tallies
6	F	f-type tallies
7	S	storage registers
8	R	address registers
9	M	mode-ids
10	N	literals or constants

The name of each symbol and the number of times it occurs in the program are kept together as a value in the corresponding attribute.

Symbol references are only checked in the PSYM file, or if the F option was used, in the specified file. To cross-reference local definitions

(such as from an INCLUDEd program) as well as the standard global definitions, a temporary symbol file containing both the global and local definitions must be created, as follows:

1. Copy all items from the regular PSYM file into the temporary symbol file.

2. Assemble program that contains the local symbols, for example, the INCLUDEd program

3. Copy all items from the TSYM file copied into the temporary symbol file.

4. Use the F option when invoking the command, and specify the name of the temporary symbol file at the prompt.

```
:CROSS-INDEX MODES DLOAD          Cross-indexes the item DLOAD in
                                  the MODES file.

:CT MODES DLOAD
    DLOAD
001 LISTFLAG 001]RMBIT 002
002 CHB 001
003 NNCF 002
004 CTR1 002]MODULO 007]OBSIZE 001]RSCWA 001]SEPAR 010]
005 BASE 008]DO 001]OVFLW 001]R15FID 001]RECORD 005
006 FP1 001
007 BMSBEG 001]CSBEG 001]ISBEG 002]OBBEG 001]S2 002
008 CS 006]IS 021]OB 005]R14 003]R15 006]TS 001
009 CRLFPRINT 001]CVDR15 003]CVTNIS 002]GETBLK 001
010 AM 002
```

MLIST

The MLIST command lists an assembly language program.

Syntax

MLIST filename {itemlist} {(options)}

filename name of file that contains items to be listed

itemlist one or more explicit item-IDs, or an asterisk (*) to specify all items in the file; may be omitted if a select-list is active

(options

n-m	lists only line numbers n through m, inclusive
E	prints error lines only
J	enables page eject if EJECT directive is in program being listed
M	prints macro expansions of source statements
N	inhibits waiting at end-of-page when listing to the terminal
P	routes output to print spooler
S	suppresses the display of object code

Description

The MLIST command generates a program listing with one instruction per line. Each line shows a statement number, location counter, object code, and source code, with the label, opcode, operand and comment fields aligned. A page heading is output at the top of each new page.

Errors, if any, are displayed on the line following the line that contains the code. Macro expansions, if requested, are displayed as source code, but with the opcodes prefixed by a plus sign (+).

```
:MLIST  MODES  LIST4
PAGE   1            LIST4          FRAME 511         14:40:16  29 JAN 1991


001 0001 7FF301FF            FRAME 511
    0001
002                  *
003      T 5F 0      HSENDDSP  DEFTU HSEND            DISP FIELD OF HSEND
004                  *
005 0001 70BE                ZB    SB30              INTERNAL FLAG
006 0003 90A1040F            BBS   SB1,NOTF          NOT FIRST TIME
007                  * FIRST TIME SETUP
008 0007 F21A412B            MOV   4,CTR32
009 000B 117009              BSL   PRNTHDR           INITIALIZE AND PRINT HEADING
010 000E 80A1                SB    SB1
011                  *
012 0010 909E0A1B   NOTF      BBZ   RMBIT,OP          LAST ENTRY
013 0014 A21A644E            BDNZ  CTR32,RETURN      NOT YET 4 ITEMS OBTAINED
014 0018 F21A412B            MOV   4,CTR32           RESET
015 001C E05CEE     OP        MOV   HSBEG,R14
  .
  .
```

MLOAD

The MLOAD command loads an assembly language program mode (item) into the frame specified in the mode's FRAME opcode.

Syntax

MLOAD filename {itemlist} {(options}

filename name of file that contains items to be loaded

itemlist names of items to load; may be one or more explicit item-IDs, or an asterisk (*) to specify all items in the file; may be omitted if a select-list is active

(options

E prints only messages relating to errors

I prints item-IDs if more than one is MLOADed

N inhibits load but prints message

P routes output to print spooler

Description

The mode to be loaded must contain no more bytes of object code than are in an ABS frame (ID.DATFRM.SIZE in PSYM). The first statement assembled in the mode must be a FRAME statement.

If the load is successful, a message is displayed:

```
[216] Mode 'item-ID' loaded; Frame =nnn Size =sss Cksum =cccc
```

nnn frame number into which the mode has been loaded; nnn is decimal.

sss number of bytes of object code loaded into the frame, expressed in hexadecimal

cccc byte checksum for the object code in the loaded mode.

The program then becomes part of the ABS software.

MVERIFY

The MVERIFY command checks previously loaded object code against the assembled source item.

Syntax

MVERIFY filename {itemlist} {(options}

filename name of file that contains items to be verified

itemlist names of items to verify; may be one or more explicit item-IDs, or an asterisk (*) to specify all items in the file. May be omitted if a select-list is active.

(options

A	displays all error bytes
E	prints only messages relating to errors
I	prints item-ID if more than one item is MVERIFYed
P	routes output to the print spooler

Description

MVERIFY is used to verify assembly language object code in a program item, or mode, against the actual code loaded in the ABS frame specified by the FRAME opcode in the mode.

If the item verifies, a message similar to the following is displayed:

```
[217] Mode 'item-ID' verified; Frame =nnn Size sss Cksum=cccc
```

nnn	frame number into which the mode has been loaded; nnn is decimal.
sss	number of bytes of object code loaded into the frame, expressed in hexadecimal
cccc	byte checksum for the object code in the loaded mode.

If the process finds mismatches, they are displayed along with an error status message:

```
aaa bb cc
[218] MODE 'item-ID'   Fram e=  nnn  has  xx mismatches
```

aaa	location of first error

bb value that should be in the first byte of that location

cc value that is currently there

nnn frame number into which the mode has been loaded; nnn is decimal.

xx total number of errors

```
:MVERIFY  SM  EXAMPL1

[217] Mode 'EXAMPL1" verified; Frame=511 Size=1FB Cksum=A03C

:MVERIFY  SM  EXAMPL2

014 0C 18
[218] Mode 'EXAMPL2'   Frame=511  has 78 mismatches

 MVERIFY SM EXAMPL2 (A                list all mismatches

LOC  SB AB  LOC  SB AB  LOC  SB  AB  LOC   SB AB
014  OC 18  015  13 17  016  0E  0D  017   3A 3C
...

[218] Mode 'EXAMPL2'   Frame=511  has  78  mismatches
```

SET-SYM

The SET-SYSM command is used to specify symbol names for display and data change.

Syntax

SET-SYM filename {(T)}

filename name of file that contains symbols

(T indicates that filename is secondary file and that previously specified symbol file is also to be used

Description

Normally, PSYM is used as the symbol file so that all the global PSYM symbols can be referenced. The Coldstart procedure supplied by Ultimate on the SYS-GEN tape initially sets up the symbolic debugging capability for all symbols in the PSYM file. (The command :DEBUG-PSYM is used by the Coldstart procedure to set up PSYM as the symbol file for the debugger.)

Users are therefore not required to use the SET-SYM command before referencing PSYM elements symbolically in the debugger. However, the SET-SYM command is required if a user wishes to specify the T option, or when using a symbol file other than PSYM.

Local references can be made to another file by using the SET-SYM verb with the (T) option. This is useful when working with numerous local symbols, such as those defined in INCLUDEd programs. For example, immediately after an assembly, the TSYM file has all the local symbols in it and it can be specified in the SET-SYM command. However, the contents of the TSYM change after an assembly. To preserve the local symbols, copy them to a more permanent file, then that file can be used with the SET-SYM with the T option.

X-REF

The X-REF command creates a cross-reference listing of symbols and stores it in an XSYM file.

Syntax

X-REF filename {itemlist}

filename name of file that contains items to be cross-referenced; this is usually the CSYM file, but can be any file in same format as CSYM file

itemlist names of items to cross-referernce; may be one or more explicit item-IDs, or an asterisk (*) to specify all items in the file; may be omitted if a select-list is active.

Description

The X-REF command uses the CSYM file created by the CROSS-INDEX command, or another similarly formatted file, for input. It creates a cross-reference listing by symbol name; the listing includes all symbol names and stores the result in a file called XSYM. (The XSYM file must already exist).

The symbol name is used as the item-ID. Each program that uses that symbol name is stored as a value in attribute 1 of the file. Each item has only one attribute.

Note: The CSYM file is composed of program name items that have symbol names as data. The XSYM file is composed of symbol name items that have program names as data.

To list the file, you should create an attribute definition item similar to the following:

```
REFERENCES  item-ID
       001  A
       002  1
       003  References
       004
       005
       006
       007
       008
       009  L
       010  70
```

```
:SORT  XSYM
PAGE   1                                  17:39:08  29 JAN 1991


XSYM : CTR32
References   LIST4

XSYM : CTR9
References   CHARGES

XSYM : CVDR15
References   CHARGES

XSYM : D0
References   CHARGES

XSYM : D1
References   CHARGES

XSYM : D2
References   CHARGES

XSYM : D3
References   CHARGES
  .
  .
```

XREF

The XREF command is a PROC which clears the XSYM file, executes an X-REF command, then produces a sorted listing of the XSYM file.

Syntax

XREF {filename {itemlist {options}}}

filename name of file that contains items to be cross-referenced; this is usually the CSYM file, but can be any file in same format as CSYM file; if not specified, it is prompted for

itemlist names of items to cross reference; may be one or more explicit item-IDs, or an asterisk (*) to specify all items in the file; if not specified, it is prompted for

options any option that is valid for the SORT command

Description

Before using XREF, an attribute called REFERENCES must be defined in the file dictionary. (REFERENCES is described in the description of X-REF)

```
:XREF  CSYM  *
PAGE   1                                17:43:08  29 JAN 1991

XSYM : CTR32
References    LIST4

XSYM : CTR9
References    CHARGES

XSYM : CVDR15
References    CHARGES

XSYM : D0
References    CHARGES

XSYM : D1
References    CHARGES

XSYM : D2
References    CHARGES
  .
  .
```

Notes

3 Addressing and Representing Data

This chapter discusses the general concepts of how data is addressed and the symbol types used to describe data. The following topics are covered:

- frame formats
- data formats in a frame
- virtual addresses
- understanding address registers
- understanding storage registers
- addressing modes in an instruction0
- symbol types
- addressing the PCB fields
- addressing the SCB fields
- addressing conventional buffer workspaces
- programming conventions

Frame Formats

As a virtual system, Ultimate programs and data reside on disk. Each addressable section of disk memory is called a frame. By convention, there are two logical types of frame formats: linked and unlinked.

A linked frame is part of a chained set. The first few bytes of a linked frame contain fields for two links: a forward link to the next frame of data, and a backward link to the previous frame of data. Linked frames are used primarily for files of data (which are variable in size) and for the larger workspaces.

An unlinked frame stands alone; that is, it has no forward or backward links to other frames. The entire frame is used for data; there are no link fields. Unlinked frames are used primarily for programs, short workspaces and control blocks.

Physically, there is no difference between a linked frame and an unlinked frame—nothing in the frame itself indicates whether it should be viewed as linked or unlinked. The distinction is made by software, when a program attaches an address register to point within a frame. (For information on attaching registers, refer to the section, Attaching an Address Register.) If a frame is attached in linked mode, the register can be incremented or decremented to point to any byte within the set of linked frames, as if the frames were a single area of contiguous storage. The operating system automatically points the register into the correct frame at all times, by reading the link fields. If a frame is attached in unlinked mode, an address register can only reference data in the current frame, although this includes all bytes in the frame.

Frame Size

Originally, in the Ultimate operating system, all frames contained 512 bytes. Linked frames had an addressable size of 500 bytes and a 12-byte link field; unlinked frames had an addressable size of 512 bytes. Now, however, frame sizes and the link field sizes in linked frames are variable, depending on the system implementation.

The frame and link field sizes for a particular implementation are stored in special ID.symbols in the Permanent Symbol (PSYM) file, as follows:

specifies the length of ABS frames (for example, 512)

ID.DATFRM.SIZE	specifies the length of an unlinked data frame
ID.DATA.SIZE	specifies the length of the data portion of a linked frame (for example, 500)
ID.LINK.SIZE	specifies size of link area (difference between unlinked and linked sizes)

Note: *The value of ID.ABSFRM.SIZE and ID.DATFRM.SIZE is always a power of 2.*

Figure 3-1 illustrates the layouts of linked and unlinked formats.

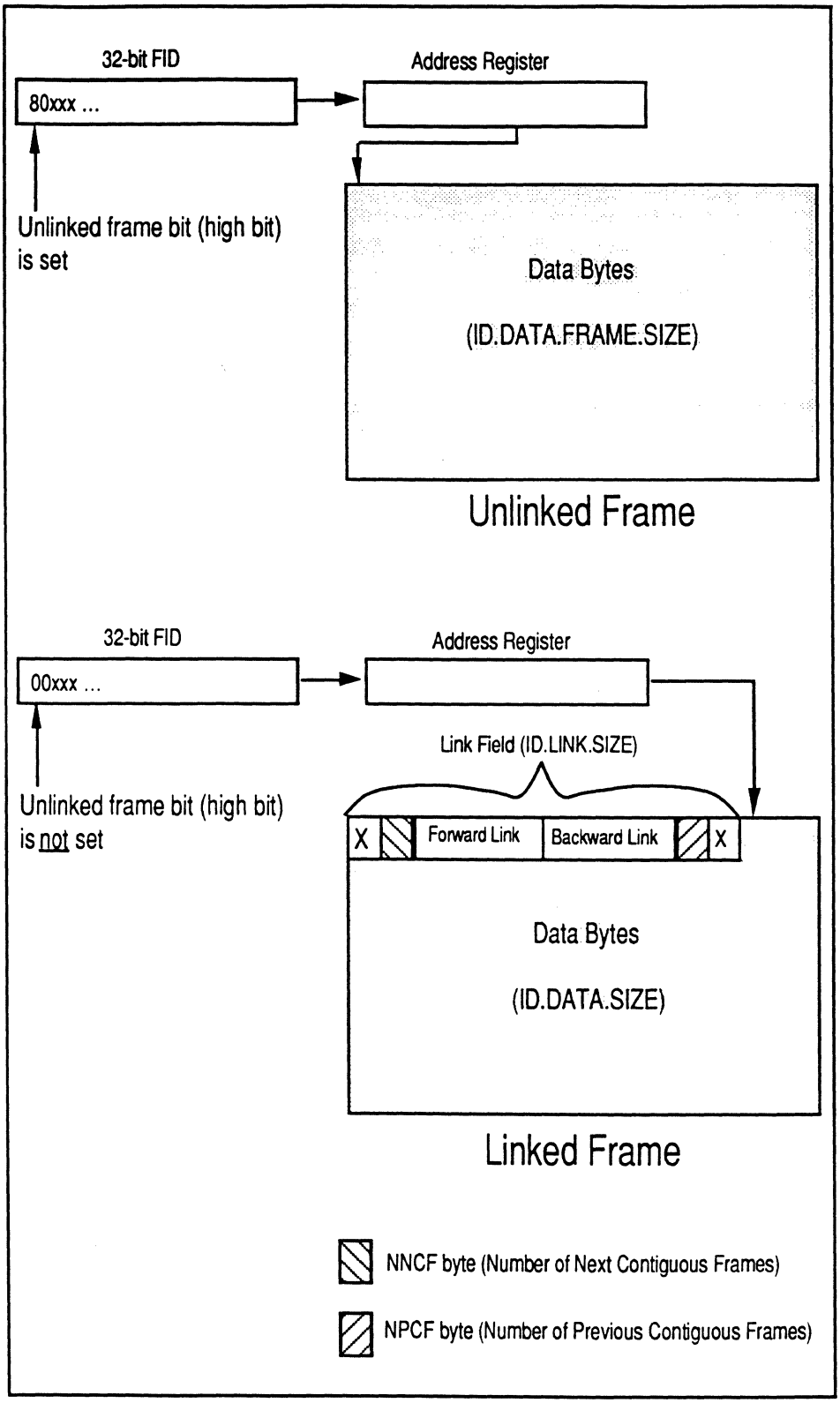
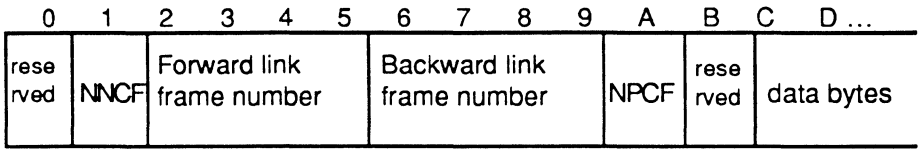


Figure 3-1. Frame Formats

Link Fields

In a linked frame, the first ID.LINK.SIZE bytes (12 on a firmware system) make up the link field, which contains link information. Following the link field are ID.DATA.SIZE bytes (500 on a firmware system) of data. The link fields contain a count of the number of sequential forward and backward linked frames, and the next and previous frame numbers (FIDs) in this linked set.

The following describes the format of a linked field (assuming 12 bytes in the link field):



Byte	Description
0	reserved
1	NNCF (number of next contiguous frames); count of the number of sequential frames linked after this one.
2-5	forward link frame number (FRMN); contains the frame number of the next frame in this logical set. (These bytes are zero if this is the last frame in the set.)
6-9	backward link frame number (FRMP); contains the frame number of the previous frame in this logical set. (These bytes are zero if this is the first frame in the set.)
X'A'	NPCF (number of previous contiguous frames); count of the number of sequential frames linked previous to this one.
X'B'	reserved; sometimes referred to as a **dummy data byte.**

The Purpose of NNCF and NPCF

When a frame boundary is reached, the link information is examined to determine which frame is to be addressed next. Depending on the direction of movement in the logical chain, either the forward link or the backward link is used to continue in the chain.

If the required address is more than ID.DATA.SIZE bytes ahead or behind the boundary of the current frame, the contiguous count plays a role. If the contiguous count is non-zero, it may be used to compute the next frame to be addressed since it is known that the frame numbers are contiguous or sequential; that is, one or more intervening frames may be skipped over.

This scheme obviously results in considerable savings in frame faulting when indexing into large contiguous blocks of frames, or skipping over large segments of data in such frames.

It is possible that a frame links to a sequential frame, but that the NNCF or NPCF is zero. While this reduces efficiency, it is not an error.

```
:DUMP  6520  L
  .
  .
+ FID: 6520 : 7  6521  6519 120 (1978 : 7  1979  1977 78)
+ FID: 6521 : 6  6522  6520 121 (1979 : 6  197A  1978 79)
+ FID: 6522 : 5  6523  6521 122 (197A : 5  197B  1979 7A)
+ FID: 6523 : 4  6524  6522 123 (197B : 4  197C  197A 7B)
+ FID: 6524 : 3  6525  6523 124 (197C : 3  197D  197B 7C)
+ FID: 6525 : 2  6526  6524 125 (197D : 2  197E  197C 7D)
+ FID: 6526 : 1  6527  6525 126 (197E : 1  197F  197D 7E)
+ FID: 6527 : 0   0    6526 127 (197F : 0   0    197E 7F)
```

This is an example of the end of a set of 128 contiguously linked frames. The first number in each line is the FID; the second is the NNCF; the third is the forward link FID; the fourth is the backward link FID; and the fifth is the NPCF. The numbers in parentheses are the equivalent values in hexadecimal.

```
:DUMP  12568  L

  FID: 12568 : 0    0     0     0 (3118 : 0    0     0     0)
```

This frame has no forward or backward links.

ABS Frames

The Ultimate operating system has designated a group of frames as **ABS** frames; usually these are frames 1-2047 (up to a maximum of 4095). All frames that are not ABS frames are called **data** frames and are used for files, work-spaces, etc. (Frame 0 is unused and is considered an "illegal frame.")

ABS frames are normally used to hold assembly language programs. All programs must be located in the first 4095 frames of virtual storage since instructions are referenced via a 12-bit frame number (three hexadecimal digits). (The maximum three-digit hexadecimal value (X'FFF') is equivalent to decimal 4095.)

As a rule, ABS frames are in unlinked format. Conversely, most data frames, except for small workspace areas, such as process control blocks, are in linked format.

Data Formats in a Frame

The data in a frame may be addressed in one of the following formats: bit, byte, tally (word), double tally, triple tally, or string data type.

bit	binary digit; can contain one of two values: 0 or 1. Bits are often used as switches or flags: OFF or UNSET for 0 value, and ON or SET for 1 value.
byte	eight bits, and is also known as a half tally or character. Bytes can store values in the range of -128 through +128.
tally	two bytes with a range of -32,768 through +32,767. Tallies are the basic word size in an Ultimate system and are the most frequently used format.
double tally	four bytes with a range of -2**31 through +2**31-1. Double tallies are typically used to store FIDs (base FID of a file, for instance), and to count items in a file.
triple tally	six bytes (also known as an F-tally) with a range of -2**47 through +2**47-1. Triple tallies are used for any arithmetic that requires the full 48-bit precision of the system.
string data	a sequence of characters of arbitrary length; may be delimited by any system delimiters, such as an attribute mark, value mark. A string is the only data type that may cross a frame boundary.

Figure 3-2 is an illustration of the layout of these data types, except string data. Using binary notation (base 2), each bit (that is, binary digit) may have only a 0 or 1 value. However, by defining byte and tally formats, very large values may be represented as single entities.

At the assembly program level, these information entities are called **elements** or **fields**, and are given symbolic names just as variables are named in higher level languages.

In Ultimate, the following conventions apply:

- All numbering starts at 0 and is incremented from left to right. So, bit 0 is the high order bit in a byte, and bit 7 is the low order bit.
- Decimal notation is normally used, although offsets within frames are usually expressed in hexadecimal (base 16). (In hexadecimal

notation, a single digit may be: 0 1 2 3 4 5 6 7 8 9 A B C D E or F, where A-F represent the decimal values 10 through 15.) When a hexadecimal value is used, the hexadecimal number is enclosed in single quotes and preceded by an X; for example X'1F' is 31 in decimal.

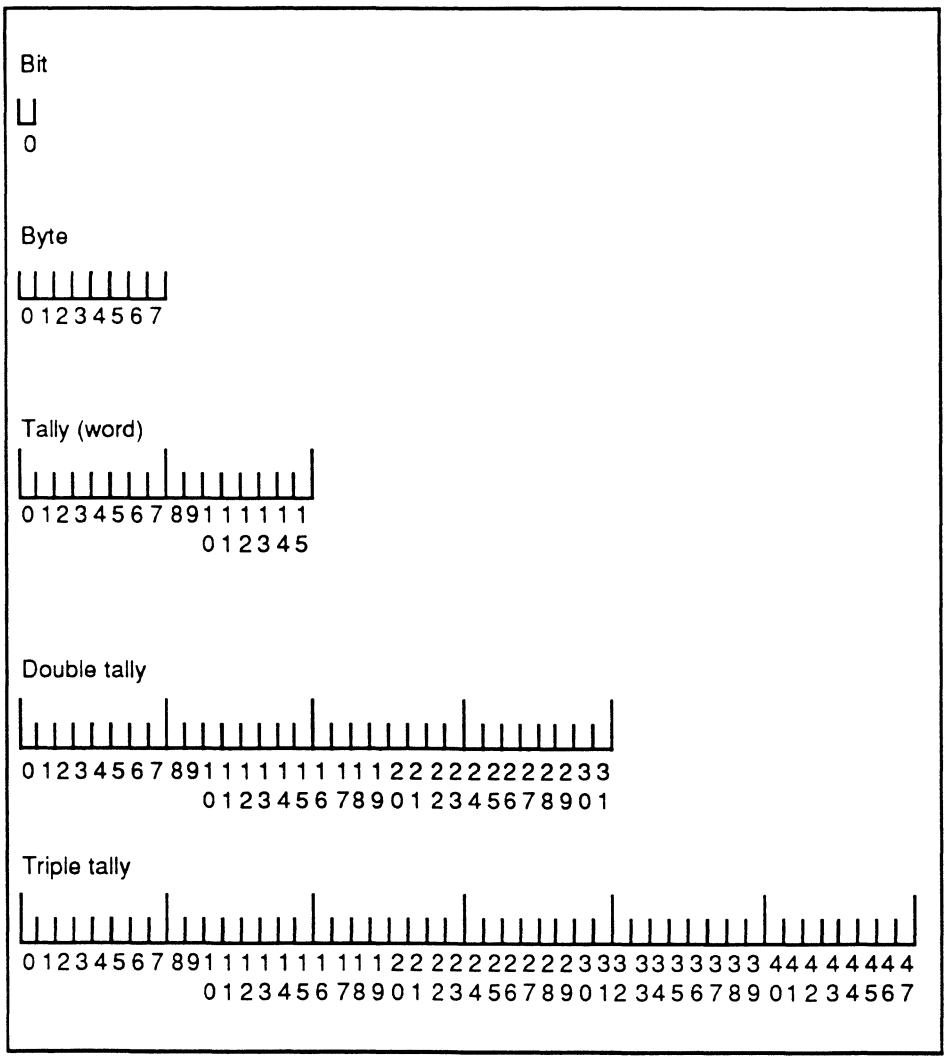


Figure 3-2. Data Formats and Bit Numberings

Virtual Addresses - Addressing Data in a Frame

All program references to data and instructions in a frame use virtual addresses. Fields in virtual storage are referenced via a frame number (FID) and a displacement of the first byte of the field within the frame. The FID and displacement together are known as the **virtual address.**

All references to data and instructions are done through address registers. Each such register contains a virtual address, which may be in either unlinked or linked format.

The number of addressable bytes in a frame depends on whether the register used is in unlinked or linked format and also on the Ultimate system implementation. (The physical number of bytes in an ABS frame is the value of the ID.symbol in PSYM called ID.ABSFRM.SIZE. The physical number of bytes in a data frame is the value of the ID.symbol in PSYM called ID.DATFRM.SIZE.)

If the register is in unlinked format, physical byte 0 of the frame is addressed by a displacement of 0. The last physical byte is addressed by a displacement of (n-1), where n is ID.ABSFRM.SIZE or ID.DATFRM.SIZE as specified above. In unlinked addressing mode, the boundaries of the frame cannot be crossed, and all bytes of the frame are addressable.

If the register is in linked format, physical byte ID.LINK.SIZE of the frame (for example, 12 on firmware systems) is addressed as byte 1. The last physical byte is addressed by a displacement of ID.DATA.SIZE (500 on firmware systems). Addresses in data frames with displacements in the range 1 to ID.DATA.SIZE are referred as **normalized.**

Displacements outside this range refer to either previous or forward frames in the logical chain (assuming that such frames exist), and such addresses are referred to as **unnormalized.** Unnormalized addresses are automatically resolved and normalized when the address register is used. Normalization consists of following the links in the appropriate direction until the displacement is reduced to the range 1 to ID.DATA.SIZE.

If the end of the linked set is reached during the normalization process, the assembly debugger is entered with a trap condition indicating either Forward Link Zero or Backward Link Zero. See the section on the debugger relating to system traps for further details.

Table 3-1 summarizes how the system resolves virtual addresses, assuming a frame size of 512 bytes. Figure 3-3 shows how virtual addresses are resolved in a linked set of frames. Virtual addresses are normally kept in address registers or storage register fields. The next topics explain more about registers.

Table 3-1. Resolution Table of Displacements and Addresses (for a 512-Byte Frame)

Displacement	Linked Mode Address	Unlinked Mode Address
less than 0	refers to previous frames in logical chain	invalid
0	If a backward link exists, a displacement of 0 is normalized to access the last byte of previous frame in chain. Also, displacement may be set to 0 temporarily in advance of using instructions that increment the register before accessing data. Data at displacement of 0 should never be accessed.	physical byte 0 of frame
1-500	physical bytes 12-511	physical bytes 1-500
501-511	refers to forward frames in logical chain	physical bytes 501-511
512 or greater	refers to forward frames in logical chain	invalid

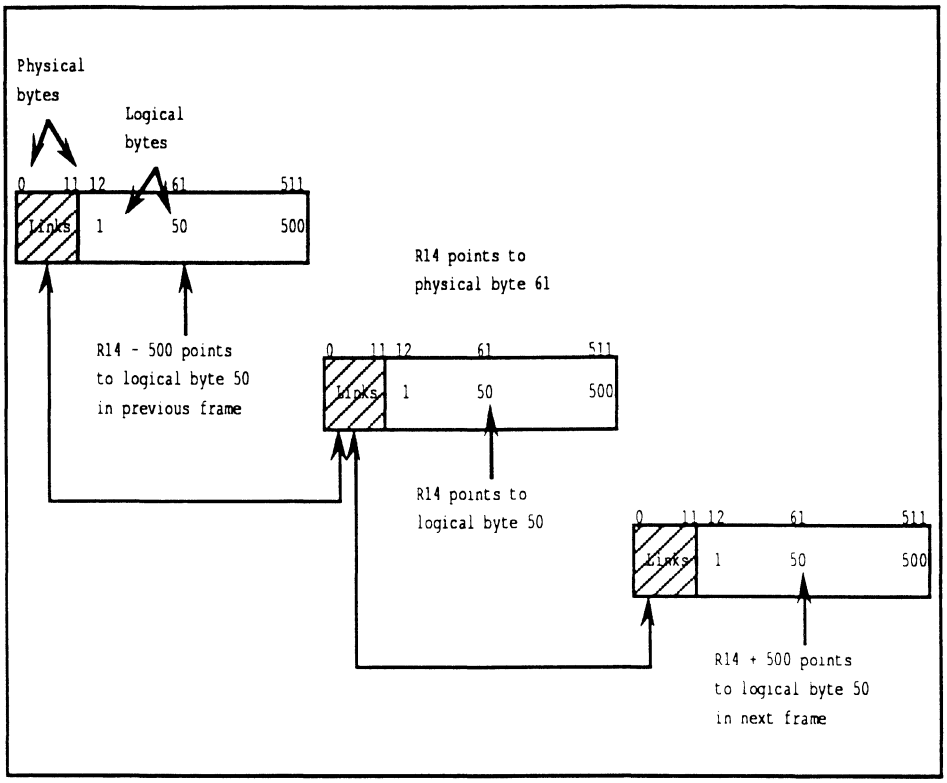
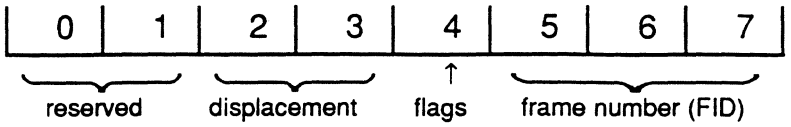


Figure 3-3. Register Displacement Involving Linked Set of Frames

Understanding Address Registers

Data within a frame is always referenced via address registers. There are no assembly language instructions that allow you to access data directly by virtual address. Instead, the location of each data element must be specified in terms of an address register and offset. A register can be thought of as pointing to a location in virtual storage. Data elements in a program are defined in terms of offsets from this location.

Every process has 16 address registers. An address register is composed of 8 bytes (see Figure 3-4) and contains a virtual address.



Byte	Description
0 and 1	reserved (used in some implementations along with bytes 2 and possibly 3 to store the main memory address when the register is attached. In other implementations, the main memory address is stored in a register that is not accessible to the programmer.)
2 and 3	displacement field, can be in range -32768 to 32767
4	flag field; contains specific bits as follows: bit 0 link mode flag for address in register 0 = linked; 1 = unlinked bit 1 special attachment flag; for internal use only (allows register to have displacement of zero when a pre-incrementing data movement instruction reaches a frame boundary. It pre-increments to the first data byte in the frame as instruction execution continues.) 0 - causes normalization and attachment to ID.DATFRM.SIZE of previously linked frame 1 - allows temporary displacement of 0 bit 2-7 reserved
5-7	frame number (FID) of address in register; can be in range 1 to (2**24)-1

Figure 3-4. Address Register Format

Attaching an Address Register

In order to use an address register, it must be **attached.** This means that the frame pointed to by the register has been copied into a main memory buffer. References to data within the frame then become references to data within this buffer. When any data in the buffer is changed, the buffer is marked *write-required*, and the Kernel schedules a disk write to copy the new version of the frame back to disk.

Address register attachment is automatic; you can use a register at any time without knowing if the frame it points to is currently in main memory. If it is, the correct memory buffer is accessed. If it is not, a **frame fault** occurs, and the kernel schedules a disk read to bring a copy of the frame into main memory.

Only one copy of a frame is ever in main memory at one time. If several address registers point into the same frame, they will point into the same memory buffer when they are attached.

In some Ultimate implementations, all 16 address registers are attached when a process is activated. In other implementations, a register may not be attached until the first instruction which tries to use the register is executed.

When a process is not active, all its address registers are detached. The contents of the registers are stored in reserved locations in the Primary Control Block (PCB) of the process. When the process is activated again, the contents of these locations are used to reattach the registers.

The PCB fields reserved for address registers are not normally referenced by assembly language programs (other than the FID field). One reason for this is that the format of these fields is not the same on all Ultimate implementations. Another reason is that the fields may not reflect or affect the true contents of an address register: some implementations maintain information about attached registers in hardware or other locations outside the PCB, and update the PCB locations only when detaching the address register.

Loading an Address Register

The standard method of loading a virtual address into an address register is to first load the virtual address into a storage register, then to move the storage register into the address register. Conversely, the standard method of obtaining the contents of an address register is to move the address register into a storage register, and then to inspect the storage register contents. (For more information on storage registers, see the section, Understanding Storage Registers.)

When a register is referenced directly by its register number (R0-R15), the reference is to the register's contents, that is, to the virtual address (FID and displacement) of the data being pointed to. For example, the following instruction causes the virtual address in R5 to be saved in storage register SR5:

```
MOV R5,SR5
```

Remember: The term "address in a register" means the FID and displacement (virtual address) of the byte that the register is referencing, **not** the data being pointed to or the location of the register itself.

Conventional Usage of Address Registers

Most of the 16 address registers have a certain conventional usage associated with them. However, only R0, R1, and R2 are system-controlled pointers; the rest are simply conventions and may be used for other purposes (at your own risk). The address registers are predefined for each user process.

Address register R0 addresses a special frame called the Primary Control Block (PCB) of the process. R1 addresses the current ABS frame being executed by the process. R2 addresses the Secondary Control Block (SCB). R3-R15 have associated conventional uses, but no predefined meanings.

The following sections describe the conventional uses of address registers. A summary is given in Table 3-2.

R0 - Primary Control Block (PCB)

R0 always addresses the Primary Control Block (PCB), which is a single frame unique to a particular process. The PCB contains address registers

R0-R15, the subroutine return stack, the accumulator, and various other data variables. The PCB of a process is the basis for every data reference that the process can make.

The PCB for each process is assigned a FID at system initialization. When the Kernel decides to turn control over to a particular process, it uses the Process Identification Block (PIB) to find the FID of the PCB for that process. It then searches the virtual memory table for that FID. If that frame is not in main memory, the process cannot be activated. An instruction to read the frame into memory is executed and the Kernel continues on to other tasks

When the PCB frame is in main memory, R0 is attached to byte zero (unlinked format) of the frame, and this main memory address is saved in a register that is inaccessible to the programmer. That register is then used to reference all other PCB elements, including the other address registers for attachment. R1 is attached first, followed by the other registers (R2-R15).

Note: Although R0 is stored in the process's PCB, it is not used for all PCB accesses. Some internal functions use a direct memory address.

The PCB is described in the section, Addressing the PCB Fields.

R1 - Program Counter

R1 has two distinct formats, depending on whether the process is active or inactive. In the inactive state, R1 is a true program counter in the sense that it addresses the location (less one byte) of the next instruction that the process will execute when it is reactivated.

In the active state, it is set pointing to byte zero of the ABS frame that the process is currently executing. This means that since R1 always addresses byte zero of the current program frame, data in that frame may be referenced relatively using R1 as a base (see the topic on Addressing Modes below). Relative addressing is the primary mode used to address literal text, symbols and other data in a program frame.

The real program counter, which actually addresses the next instruction that the process will execute, is stored in a special register and is inaccessible to the programmer.

R2 - Secondary Control Block

R2 points to another control block, called the Secondary Control Block (SCB) whose frame number is fixed as the PCB FID plus one. This block contains numerous additional elements that have both system-defined and variable uses. (The SCB layout is given in Appendix C.)

R3 through R15

Address registers 3-15 (R3-R15) are general purpose registers. However, the Ultimate system software conventions initialize R3 through R13 to specific locations (see the section Buffer Workspaces).

Understanding Storage Registers

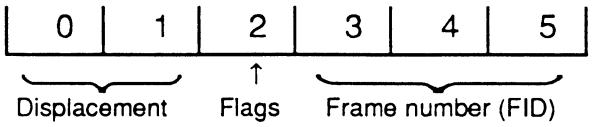
A storage register is a 6-byte field which contains a virtual address. Unlike address registers, which are fixed in number and are associated with specific locations in a process's PCB, storage registers may be defined in any frame, to store as many virtual addresses as a program needs.

A storage register may be specifically defined via an assembly language program directive (ADDR, DEFS, or SR), or it can be allocated without a symbol, as in the following:

```
MOV    R6;S0,R4
```

In this example, R6;S0 defines a storage register at the virtual address pointed to by address register R6.

The format of a storage register is as follows:



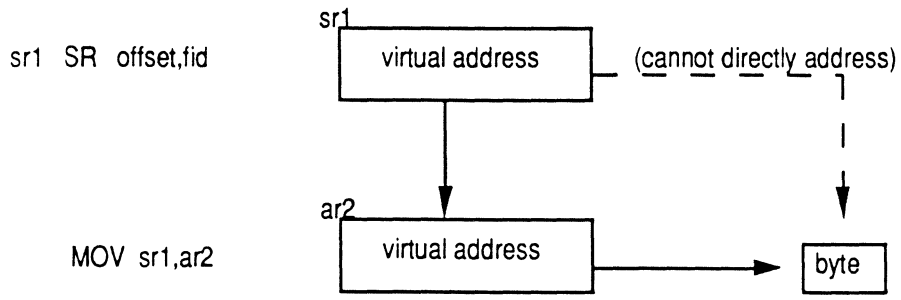
Byte	Description
0 and 1	displacement field.
2	flag field; contains specific bits as follows:
	bit 0 link mode flag for address in register 0 = linked; 1 = unlinked
	bit 1-7 reserved
3 to 5	frame number (FID) of address in register

Remember: The term "address in a register" means the FID and displacement (disk address) of the byte that the register is referencing, not the data being pointed to or the location of the register itself.

Storage registers reside in the frame in which they are defined. Standard storage register fields in the PCB and SCB are defined in PSYM.

There is no attachment associated with storage registers. In order to reference data pointed to by a storage register, the storage register must be moved into an address register, and the data referenced via the address register.

Following is an example of an instruction (MOV) that moves the virtual address from a storage register (sr1) into an address register (ar2). After the instruction is executed, both sr1 and ar2 point to the same byte.



Addressing Modes in an Instruction

The Ultimate operating system supports four addressing modes. All four modes use the address registers R0-R15.

The four addressing modes are as follows:

- immediate addressing
- relative addressing
- indirect addressing
- direct register addressing

Immediate Addressing

In the immediate addressing mode, the data is stored in the instruction itself. The candidates for immediate addressing are literal values used as operands, such as numbers or characters.

Examples of the use of immediately-addressed data are as follows:

- The source operand in character moves of a single byte (MCC, MCI instruction) where the character is immediate data.

```
MCC X'FE,' R15
```

- The mask operand in string scans and moves (MIID, SID instructions, for example) where the mask is immediate data.

```
MIID R14,R15,X'A0'
```

- For some implementations, other literal values used as operands.

```
MOV 4,CTR1
```

Note: Not all literal value operands are immediately addressed; it depends on the implementation of Ultimate for which the instruction is being assembled. Although, as a rule, one-byte literals are always assembled as immediate data, larger values may become immediate data in some implementations (for software machines) but relatively addressed data (at the end of the program) in others.

Relative Addressing

The relative mode of addressing is used to address data defined as a bit, half tally, tally, double tally, triple tally, local label, storage register, or external address register.

Relatively addressed operands are addressed via a base address register and an offset (displacement) to get the actual address. All relative addresses are computed from the virtual address pointed to by the base register. In the case of local symbols, R1 is the base register, referencing byte zero of the program frame, and the offset is simply the offset from the beginning of this frame.

To resolve the relative address, a function of the offset is added to the virtual address in the base register. The function used is dependent on the actual symbol type being addressed (which is described in the next section). Only forward addressing is allowed, and the entire element must be in the frame being addressed. (Note: some implementations check to see if the element crosses the boundary of the frame for any of the referenced field. If it does, a Crossing Frame Limit error message and an abort conditions are generated. In other implementations, no checking is done; in this case, if the element crosses the boundary of the frame, the results are unpredictable.)

Indirect Addressing

The indirect mode of addressing data is used in instructions where an address register is an operand, but the reference is to the data at the virtual address in the register. Several types of instructions use indirect addressing:

- Single-byte character moves where the destination is an unincremented register (MCC instruction).

```
MCC X'FE',R15
```

- Single-byte character moves where the destination is a pre-incremented register (MCI instruction). The destination addressed byte is located indirectly by first adding one to the virtual address in the register. The register remains altered.

```
MCI X'FE',R15
```

- String moves where the destination register is pre-incremented until a test condition is met (MIID, SID instruction, for example). The destination addressed byte is located as described for single-byte character moves above and moved, then the register is successively incremented by one and another character in the string is moved until the terminating conditions are met. The register is left addressing the last moved byte in the string.

```
MIID R14,R15,X'A0'
```

- Other instructions where an addressed byte is located indirectly by using the virtual address in the register (for example, a branch instruction where the register points to a byte whose value is being tested).

```
BCE R14,R15,LABEL
```

Direct Register Addressing

The direct mode of addressing a register (R0-R15) is confined to a group of register instructions (MOV, INC, DEC, SETR, SETDSP, etc.). In these instructions, the reference is to the contents of the register itself and the operation is on the register content, not the data at the address in the register. For example, in the following instruction, R14 is moved to replace the contents of R15, so that the two registers are then identical (that is, they contain the same virtual address):

```
MOV R14,R15
```

Symbol Types

Symbols can be either defined by a program, or predefined by the system. The predefined symbol names and criteria are stored in the PSYM file.

In assembly language programs, each symbol has an associated symbol type code. This code defines the nature of the symbol. Table 3-2 lists the symbol type codes.

As shown in the table, all symbol types, except A, M, N, and R, use the relative addressing mode. Type M symbols (mode identifiers) are used to define branches to external program subroutines, which are defined as entry points in a mode (that is, the mode-id). Addresses of this type are discussed in Chapter 2 in the section, External Program References: Mode-ids. Type N symbols (literals) are treated as immediate data. Type R symbols (address registers), which are treated directly or indirectly, have addressing modes that are discussed in the section, Addressing Modes in an Instruction.

The effects of symbol types in computing relative addresses are discussed in the next section.

Table 3-2. PSYM Symbol Type Codes

Symbol Type Code	Description	Unit of Offset	Maximum Displacement
A	virtual address (both FID and displacement)	N/A	N/A
B	relatively addressed bit	bits	255 bits - 31 bytes+7 bits
C	relatively addressed character or byte (8 bits)	bytes	255 bytes
D	relatively addressed double tally (32 bits)	words	255 words - 510 bytes
E	system message	N/A	N/A
F	relatively addressed triple tally (48 bits)	words	255 words - 510 bytes
H	relatively addressed half tally (8 bits)	bytes	255 bytes
L	locally defined label	bytes	255 bytes[1]
M	mode-id (16 bits); external FID and entry point	N/A	N/A
N	constant or literal value	N/A	N/A
R	address register	N/A	N/A
S	storage register	words	255 words - 510 bytes
T	relatively addressed tally (16 bits)	words	255 words - 510 bytes

[1]Local labels are subject to the 256-byte limitation only in the SRA instruction. In a branch instruction, it is an absolute location in the object code.

Computing Relative Addresses by Symbol Type

Symbols referenced in a relative addressing mode specify a base register and an offset displacement. The resulting address may be offset up to the maximum displacement as given in Table 3-2, although it may not cross the boundary of the frame that the register is addressing.

Offsets are in the range 0-255. The offset value is taken from the definition of the symbol in the symbol file. The column "Unit of Offset" indicates the function used to convert the offset to the effective address.

The following are examples of symbol definitions:

```
B15     DEFB  R1,15
H15     DEFH  R1,15
T15     DEFT  R1,15
D15     DEFD  R1,15
S15     DEFS  R1,15
```

The relative address computed for each of these symbol definitions is different, as illustrated in Figure 3-5.

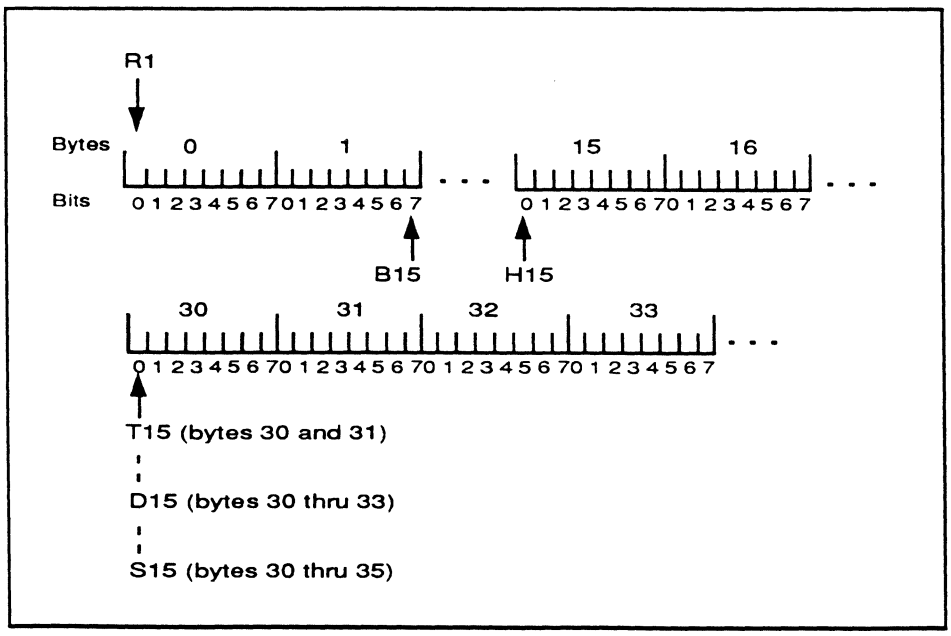


Figure 3-5. Relative Addressing of Symbols

R1 points to byte 0 of the frame. Each symbol is also shown with the effective address of that symbol:

- if the symbol being addressed is a bit (type B), the offset is also in bits, so that an offset of 15 would address the seventh bit in byte 1 displaced from the address in the register
- if the symbol is a half tally (type H), the offset is in bytes, so an offset of 15 would address byte 15 displaced from the address in the register
- if the symbol is a tally (type T), the offset is in words, so an offset of 15 would address bytes 30 and 31 displaced from the address in the register
- if the symbol is a double tally (type D), the offset is also in words, so an offset of 15 would address bytes 30 through 33 displaced from the address in the register.
- if the symbol is a storage register (type S), the offset is again in words, so an offset of 15 would address bytes 30 through 35 displaced from the address in the register.

Limits in Offsets

The reason for limits in offsets used in relative addressing is so that any relatively addressed operand can be specified by a 12-bit number. This number includes four bits for specifying an address register (0-15), leaving eight bits for an offset (0-255). In fact, this is how relatively addressed operands are coded in the object code for firmware machines.

When the maximum of 255 is applied to the three different units of offset shown in the table, you can see that the actual addressable bytes, offset from the address in the register, are different:

256 addressable bits	you can address a bit in the range of byte x'00' to x'1F' (32 bytes)
256 addressable bytes	you can address a byte in the range of byte x'00' to x'7F' (256 bytes)
256 addressable words	you can address a word in the range of byte x'00' to x'FF' (512 bytes)

A limitation of this scheme is not being able to address words (tallies, double tallies, etc.) at odd (byte) boundaries off a register; however, this scheme does allow an entire 512-byte frame to be referenced at word boundaries. In practice, the restrictions on relative addresses are not a problem.

Note that a relatively addressed character or half tally (byte operand), when the offset is zero, is the same as an indirectly addressed byte. For example, the code sequence:

```
CHR15   DEFC R15,0          C at R15, offset 0
        MCC  R14,CHR15      Move to relatively-
                            addressed byte
```

produces the same effect as:

```
        MCC  R14,R15        Move to indirectly-
                            addressed byte
```

The object code generated by the assembler may be different for the two cases, however.

Addressing the PCB Fields

The primary control block (PCB) of each process contains indicators and flags for that process, including the following:

- accumulator
- scan characters
- file control block pointers
- address register fields
- subroutine return stack

All elements in the PCB are accessed via address register zero (R0), which always addresses byte zero of the PCB in unlinked mode.

The format of the PCB may vary depending on the system implementation. The actual location of most PCB elements is irrelevant to programmers since they are referenced via their PSYM name. A sample PCB format is shown in Appendix B.

The Accumulator

The accumulator consists of an 8-byte accumulator area and a 6-byte extension (14 bytes). The accumulator is used in the following instructions:

- LOAD and STORE instructions.
- arithmetic instructions.
- LAD instruction.
- Certain string scanning and moving instructions to count the number of bytes scanned or moved.
- Certain string-to-binary and binary-to-string conversion instructions.

The primary accumulator area consists of two double tallies, labeled D1 and D0. This area is used for most arithmetic operations, except for the extended arithmetic instructions. Extended arithmetic addresses a 6-byte area (triple tally) of the accumulator, labelled FP0. Another triple tally, FPY, is a 6-byte extension which is used for extended precision division instructions only (DIVX instruction).

The primary accumulator area occupies bytes 8 through 15 of the PCB. This area may be addressed symbolically with the following units:

bits (B0-B63)

half tallies (H0-H7)

tallies (T0-T3)

double tallies (D0-D1)

triple tally (FP0).

See Figure 3-6.

The accumulator retains its last resultant value until other data is moved or loaded into it.

The 6-byte accumulator extension is called FPY and is located at bytes 498-503 (X'1F2'-'1F7') of the PCB.

The symbols in Figure 3-6 are all global variables in the PSYM file and

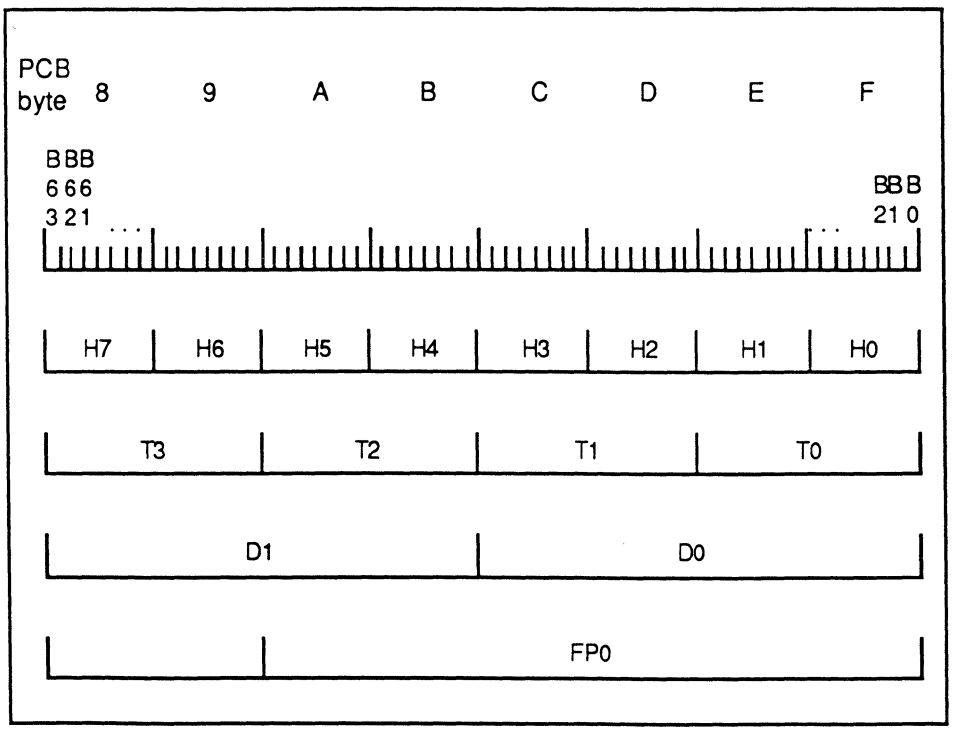


Figure 3-6. Primary Accumulator Area

may be used to address sections of the accumulator. Some instructions, such as LOAD and ADD, address the accumulator implicitly; the portion depends on the operand.

The Accumulator and Arithmetic

All arithmetic is performed in the PCB accumulator. Each arithmetic instruction operates on specific prenamed portions of the accumulator, as determined by the type of arithmetic and the operands used. For any particular instruction, only a certain portion of the whole 8 bytes is addressed. When performing arithmetic, the following symbol types used as operands cause the accumulator to be addressed in a corresponding way.

Operand	Symbol Type	Accumulator Portion Addressed
H	half tally	D0
T	tally	D0
D	double tally	D0
F	triple tally	FP0

The accumulator does all arithmetic in binary, and expects that file values have been converted to binary, if necessary. (That is, they may have been stored on disk as ASCII values.) The following examples show the value of the accumulator using hexadecimal equivalents.

For example, if the accumulator contains a value of zero:

```
| 0 0 | 0 0 | 0 0 | 0 0 | 0 0 | 0 0 | 0 0 | 0 0 |
```

the ADD instructions below would return these results from the accumulator:

```
        ADD H8                  H8 = 64 (X'40')

| 0 0 | 0 0 | 0 0 | 0 0 | 0 0 | 0 0 | 0 0 | 4 0 |
```

```
        ADD T4                  T4 = 42000 (X'A410')

| 0 0 | 0 0 | 0 0 | 0 0 | 0 0 | 0 0 | A 4 | 1 0 |


        ADD D6                  D6 = 1234567890
                                     (X'499602D2')

| 0 0 | 0 0 | 0 0 | 0 0 | 4 9 | 9 6 | 0 2 | D 2 |


        ADDX FP1                FP1 = 123456789012345
                                     (X'7048860DDF79')

| 0 0 | 0 0 | 7 0 | 4 8 | 8  6 | 0 D | D F | 7 9 |
```

Numbers are stored in two's complement form. The high-order bit of a positive number is 0. The high-order bit of a negative number is 1. This high order bit is propagated to the left when necessary to sign-extend a number within the section of the accumulator (D0 or FP0) being used. The sign is extended initially when the accumulator is loaded with a value. For example:

```
        LOAD N                  N = a half tally of X'7F'

| 0 0 | 0 0 | 0 0 | 0 0 | 0 0 | 0 0 | 0 0 | 7 F |
```

Because the high order bit is a 0, the number is positive, and the 0 sign bit is extended throughout the accumulator D0. The D1 portion is not affected.

If the same half tally were to have a value of X'80', the high order bit would be 1, which would also be sign-extended throughout the accumulator D0. For example:

```
        LOAD N                  N = a half tally of X'80'

| 0 0 | 0 0 | 0 0 | 0 0 | F F | F F | F F | 8 0 |
```

Note that the sign is extended within the accumulator D0, but the D1 portion is not affected by this instruction. However, a LOADX N

instruction would extend the sign through FP0, leaving only T3 unaffected.

In Chapter 7, Reference for Programmers, there is additional information about two's complement arithmetic.

Accumulator Usage

The following are some general guidelines for accumulator usage:

- Extended precision arithmetic instructions such as ADDX affect FP0; DIVX also affects FPY.
- Normal precision arithmetic instructions such as ADD affect D0; MUL and DIV also affect D1.
- Instructions that count string lengths, as well as the LAD instruction, use T0 only.
- Conversion instructions use FP0 for data and T3 as a parameter.

Scan Characters

Scan characters are programmer-specified characters used in string scanning and moving instructions. Three one-byte fields called SC0, SC1, and SC2 contain the characters. The fields are referenced through mask bytes.

Mask bytes are used by the following instructions:

MIID	SID
MIIDC	SIDC
MIITD	SITD
SICD	

The mask bytes used by MIID, MIIDC, MIITD, SID, SIDC and SITD instructions can specify up to seven different characters to be tested; four of them are the standard system delimiters:

segment mark	SM	X'FF'
attribute mark	AM	X'FE'
value mark	VM	X'FD'
sub-value mark	SVM	X'FC'

The other three characters are taken from the scan characters SC0, SC1, and SC2. The contents of these scan characters are specified by the programmer.

Note: The mask byte used by the SICD instruction is unique and is discussed as part of the instruction description in Chapter 4.

The low order seven bits in the mask byte are used to determine which of the seven characters are to be compared; if any bit is set (1), the corresponding character is tested; if zero (0), it is ignored.

If the high-order bit (bit 0) of the mask byte is set (1), it indicates that the string terminates on the first **occurrence** of a delimiter as specified by the setting of bits 1-7. If it is zero (0), it indicates that the string terminates on the first **non-occurrence** of a delimiter as specified by the setting of bits 1-7.

See Figure 3-7. (The parentheses around SC0, SC1 and SC2 are to indicate that it is the contents of these locations that are compared.)

The following are some examples of mask bytes:

Mask byte	Bit pattern	Meaning
X'C0'	1100 0000	Stop on first occurrence of a SM.
X'A0'	1010 0000	Stop on first occurrence of an AM.
X'C3'	1100 0011	Stop on first occurrence of an SM, or the contents of SC1 or of SC2.

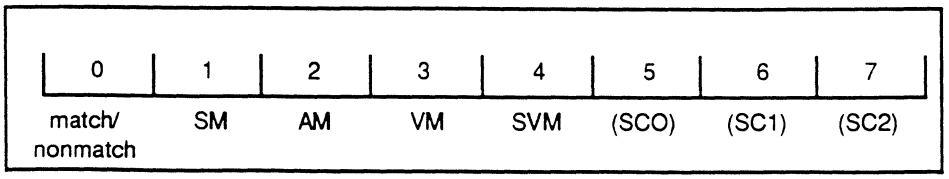


Figure 3-7. Mask Byte Format

Mask byte	Bit pattern	Meaning
X'F8'	1111 1000	Stop on first occurrence of any system delimiter - SM, AM, VM, or SVM.
X'01'	0000 0001	Stop on the first non-occurrence of the contents of SC2. For example, if SC2 contains a blank, this mask causes the instruction to terminate when the first non-blank character is encountered.

For information on the use of these fields, see the MIID, MIIDC, MIITD, SICD, SID, SIDC and SITD instructions.

File Control Block Pointers

Each file in the system has a File Control Block (FCB) that stores file access information, such as the file's base, modulo, separation, and other status information. The PCB contains the FIDs of the FCBs typically associated with the following files:

Symbols	Associated File Information
FCB1, FCB2	current file
DFCB1, DFCB2	file dictionary section (typically)
MFCB1, MFCB2	master dictionary (user's MD)
EFCB1, EFCB2	ERRMSG file
FFCB1, FFCB2	file data section (typically)

See the GETFILE and OPENDD subroutine descriptions for more information on these fields.

Subroutine Return Stack Fields

The assembly subroutine return stack in the PCB can handle up to 11 entries. An extended stack, which resides in a workspace frame defined in the item WORKSPC-DEFS in the SM file, can hold up to 125 entries.

Each stack entry is four bytes, where bytes 0 and 1 contain the FID and bytes 2 and 3 contain the displacement. The first two return stack entries in the PCB are used to handle return stack full and return stack empty processes.

When the process executes a subroutine call, the address of the last byte of the call is stored in the return stack and the stack pointer is incremented by four bytes. When the stack in the PCB is full, the routine in the first stack entry is called to move the oldest five entries to the extended workspace; the remaining entries are moved down, freeing up room for five more entries.

On executing a subroutine return instruction, the stack pointer is decremented by four bytes, then used to get the return address. If the stack in the PCB is empty, the routine in the second stack entry is called to move entries back from the extended stack.

If desired, the extended stack can be logically divided into multiple stacks. When the stack is divided into logical stacks, the entire logical stack can be moved to the PCB stack.

The following instructions can be used to access the return stack:

INITRTN	initializes return stack; can be useful in conditions where a process is to be re-initialized and all current entries in the stack are to be deleted or ignored
POPRTN	pops one entry off the return stack; this is mandatory if a subroutine is to be exited without using a RTN instruction
MARKRTN	copies all the active entries in the PCB to the extended stack, then marks them as one logical stack
RTNMRK	pops all entries in the PCB and the extended stack, up to and including the marker. Any remaining entries in the extended stack are moved back to the PCB if the return stack is empty.

XMODE Field

The XMODE tally field can be used to branch to a specified mode-id (subroutine) when a Forward Link Zero error condition occurs. This error indicates that the program has reached the end of a set of linked frames without completing the current instruction. See Chapter 6 for more information about using XMODE.

RMODE Field

When the WRAPUP software is entered to store or print messages, a return may be requested by placing a mode-id in the tally field RMODE. When WRAPUP completes the requested processing, an ENT* RMODE instruction transfers control to the program whose mode-id has been stored in RMODE. See Chapter 6 for more information about using RMODE.

WMODE Field

When WRAPUP finishes processing, just before it returns to TCL or PROC, the tally field WMODE is checked. If WMODE is non-zero, control is transferred via a BSL* WMODE instruction to the subroutine whose mode-id has been stored in WMODE.

Assembly programs that require special handling before completing may gain control in this way. The control transfer via WMODE occurs even if the process has been terminated via the debugger END command.

An example of WMODE usage is when writing to magnetic tape. If the process is stopped for any reason, an EOF mark should be written on the tape. Setting WMODE to the mode-id of the subroutine that writes an EOF mark (TPWEOF) automatically ensures this.

OVRFLCTR Field

When the system software gets space from the system's overflow space pool, the FID of the first frame so obtained is placed in the special double tally field OVRFLCTR. This is typically done by a sorting or selecting function such as SORT or SELECT. The extra space needed by the program is built up as a chain of frames obtained as needed.

Just before WRAPUP returns control to TCL, OVRFLCTR is checked, and if it is non-zero, the subroutine RELCHN is called to return the chain of frames to the overflow pool. To maintain this convention of releasing space, OVRFLCTR should not be changed by any program other than the first one that gets space and initializes it.

User code written as a TCL-I or TCL-II verb may initialize OVRFLCTR if it uses overflow space that is to be released when the process terminates by returning to WRAPUP. However, TCL-II initializes OVRFLCTR for update-class commands (that is, attribute 5 of the verb definition item contains a U) used with more than one item. In this case, user code must use another means of returning space, perhaps via WMODE.

INHIBIT and INHIBITH Fields

Normally, the terminal's BREAK key causes the process to enter the appropriate debugger (either assembly or BASIC). For sensitive processing that should not be interrupted, the bit INHIBIT (available to the user) and the half tally INHIBITH are used to prevent debug entry. If either are non-zero, such entry is prevented.

For example, INHIBITH is used by the system during overflow management. It is incremented by one during the sensitive processing, and decremented on exit. The increment is performed with an INC INHIBITH instruction. The decrement is performed by calling the subroutine DECINHIB.

Addressing the SCB Fields

The Secondary Control Block (SCB) contains additional elements that can be used by assembly language programs. All elements in the SCB are accessed via address register 2 (R2), which always addresses byte zero of the SCB in unlinked mode.

The format of the SCB may vary depending on system implementation. A sample SCB format is shown in Appendix C.

Addressing Conventional Buffer Workspaces

By convention, the system preassigns buffer workspaces such as HS, IS, and OS to a process via address registers R3-R15.

In Ultimate assembly programs, unlike other systems, program space is rarely used to store variables (other than text strings). All programs should be re-entrant and contain only code.

There are several preassigned buffer workspaces available to a process. Three linked workspaces, called the IS, OS, and HS, contain 64000 bytes each (128 frames on systems having 500-byte logical data frames). Five other workspaces, called the BMS, AF, IB, OB, and CS, vary between 50 and 140 bytes in length and are all in one frame. The TS workspace is one unlinked frame. These standard workspaces normally give ample room to store and manipulate string data. Counters, bits, and pointers are stored in PSYM-defined PCB and SCB elements, as mentioned in previous topics.

Each workspace is defined by a beginning pointer and an ending pointer (both are storage registers). The pointer to the beginning of the buffer is conventionally called xxBEG, and the pointer to the end of the buffer is called xxEND, where xx is the workspace name.

When the process is at the TCL level, these pointers are all set to an initial condition. At other levels of processing, the beginning pointers should normally be maintained; the ending pointers may be moved by system or user routines.

The xxBEG pointer (such as ISBEG) is set to point one byte before the actual data. This is to ensure correct operation of the string scanning and string moving instructions, which always increment an address register before testing or moving the next data byte.

For example, a typical sequence that initializes and moves data into the HS workspace is:

```
MOV HSBEG,HS          Set HS register to start of
MIID R15,HS,X'C0'     buffer; copy a string until
                      an SM.
```

Note that the byte at HSBEG is not affected, since the MIID instruction pre-increments and then stores the first byte.

The subroutine WSINIT may be used to reset the BMS, AF, CS, IB and OB registers and buffer pointers to their initial conditions. The subroutine ISINIT does the same for the IS, OS and HS buffers, and also calls WSINIT.

The buffer pointers are sometimes changed by system software, but reference is always made to a symbol, so this is mostly transparent. TSBEG, for example, always defines the beginning of the TS buffer, regardless of which frames are actually being used for this buffer at any given time.

The address registers associated with these workspaces (for example, R3 or HS) need not necessarily be maintained within their workspaces; however, system routines may reset the specific registers to their associated workspaces.

Table 3-3 shows the various workspace pointers, along with the size and location of the buffers (using the FID of the PCB as the reference point).

Notes to Table 3-3

"Not a buffer" indicates that there is no permanently assigned space associated with those address registers.

The Description column indicates the conventional usage of the buffer. "Freely usable" does not apply to a program entered from the

conversion interface of BASIC or Recall, both of which tend to be very possessive of all available registers, except R14 and R15.

The frames at PCB+6 to PCB+9 are reserved for the PROC software for working storage.

Table 3-3. Registers and Pointers (1 of 3)

Reg Num	PSYM Name	Beginning and Ending Pointers (SRs)	Size, Location of Buffer	Description
R0	-	-	-	points to byte 0 of user's PCB
R1	-	-	-	points to FID of currently executing ABS frame
R2	-	-	-	points to user's SCB
R3	HS	HSBEG fixed, HSEND floating; must point to current end of data in the HS buffer	64K bytes PCB+10	history string; stores messages to be printed at end of processing[1]
R4	IS	ISBEG fixed, ISEND floating; end of current data pointer	64K bytes PCB+16	input string; stores compiled string for Recall; data for Editor; no conventions

[1]area past HSEND may be used as scratch if needed to save data; conventions are:
strings separated by SMs
character after SM is an X
string terminated by a SM and a Z
HSEND points to the SM before the Z

Table 3-3. Registers and Pointers (2 of 3)

Reg Num	PSYM Name	Beginning and Ending Pointers (SRs)	Size, Location of Buffer	Description
R5	OS	OSBEG fixed, OSEND floating; end of current data pointer	64K bytes PCB+22	output string; stores compiled string for Recall; data for Editor; in Recall, area past OSEND is scratch; no conventions
R6	IR	none	not a buffer	points to beginning of current file item if using standard system file I/O subroutines
R7	UPD	UPDBEG not used; UPDEND not used	not a buffer	used as tape buffer pointer for tape I/O; otherwise, register R7 is freely usable
R8	BMS	BMSBEG fixed, BMSEND floating on last byte of item-ID	50 bytes PCB+4.00	stores item-ID when interfacing with system file I/O
R9	AF	AFBEG fixed, AFEND fixed	50 bytes PCB+4.51	scratch buffer in same frame as BMS; register R9 is freely usable
R10	IB	IBBEG fixed, IBEND floating; end of current data pointer	465 bytes PCB+46.33 (linked)	terminal input buffer used by terminal input routines to read data; not to be used for other purposes

Table 3-3. Registers and Pointers (3 of 3)

Reg Num	PSYM Name	Beginning and Ending Pointers (SRs)	Size, Location of Buffer	Description
R11	OB	OBBEG fixed, OBEND fixed	465 bytes PCB+47.1 (linked)	terminal output buffer used by terminal output routines to write data; not to be used for other purposes
R12	CS	CSBEG fixed, CSEND fixed	100 bytes PCB+4.102	scratch buffer in same frame as BMS; register R12 is freely usable as a scratch register
R13	TS	TSBEG fixed, TSEND floating; points to current end of data	512 bytes PCB+5	scratch area used by various processors; the area from TSBEG on may be treated as scratch space in the conversion interface; register R13 is freely usable as a scratch register
R14	R14	-		scratch register
R15	R15	-		scratch register

Programming Conventions

Programming in the Ultimate assembly language requires understanding and adhering to the conventions of the operating system. The primary areas where conventions apply are the use of

- global elements (variables) defined in the permanent symbol (PSYM) file
- predefined buffer workspaces, typically associated with address registers R3-R13.

Ultimate assembly language programming makes extensive use of global data areas. This reduces overhead in allocating and deallocating storage for programs when they are run, but requires the programmer to choose very carefully the data areas used by a program. Otherwise, data in use by other programs, including the operating system, can be destroyed.

Global elements such as bits, counters, and storage registers are defined as fields in the PCB or the SCB. The field definitions are in the PSYM file, and give the offset relative to R0 (if in the PCB) or R2 (if in the SCB). For more information on the PCB, see the section, Addressing the PCB Fields. For more information on the SCB, see the section, Addressing the SCB Fields.

In addition to the global elements, the system defines several buffers to use as workspaces. These workspace areas are used by system software such as BASIC, PROC, Recall, and the system debugger. For more information, see the section, Addressing Conventional Buffer Workspaces.

When a process is at the system (TCL) level, its process workspace pointers are in an initialized state, although the data in the workspace frames is whatever was left over from the last program. Also, most bit flags are cleared. These points are important to remember when first writing assembly programs, since they define initial conditions that the programmer must take into account. These initial conditions are discussed in more detail in Chapter 6, System Software Interfaces.

An active process always has access to the current account's Master Dictionary (MD) and to the ERRMSG file; that is, these files are open to the process.

A process can normally run any re-entrant assembly program to which it has access simultaneously with other users. (A re-entrant program is one which has no storage internal to the program. The section, Sharing Object Code Among Processes, discusses this concept. It also discusses how to lock a byte to prevent simultaneous access if a program requires internal storage, and cannot be re-entrant.)

Global Symbolic Elements - PSYM File

All PSYM elements (variables) are global and can be used by all routines. Some PSYM elements are used by the operating system, as well as system subroutines, and their values cannot be expected to be preserved when calling a system subroutine. Other PSYM elements are not used by the operating system or any subroutines, and are reserved for user assembly language programs.

The following PSYM elements in the SCB are unused by the system software and can be safely used by user-written assembly programs:

bits	SB24 - SB35
characters	none
double tallies	none
half tallies	none
storage registers	SR20 - SR29
tallies	CTR30 - CTR42

Note that **no** PCB elements, including address registers, are freely available; availability depends on the interface with the system software.

Additional elements may be stored by setting up an additional control block (see the section, Defining Additional Workspaces).

Elements used for temporary storage are known as **scratch** elements. Information that needs to be preserved should not be kept in a scratch element, since any subroutine that is called may use these elements.

The following scratch elements located in the PCB might be used by nearly any subroutine:

bits	SB60, SB61
tallies	T4, T5
double tallies	accumulator (D0, D1), D2

triple tallies	FPX (overlays SYSR0)
	FPY (overlays SYSR1)
registers	R14, R15
storage registers	SYSR0 (overlays FPX)
	SYSR1 (overlays FPY)

The following scratch element is located in the SCB:

storage register	SYSR2

These scratch elements are so widely used that their use is not covered in the documentation for most system subroutines in Chapter 5. However, each subroutine in Chapter 5 does specify all other system- and user-defined inputs and outputs to that routine.

Sharing Object Code Among Processes

In practically all cases, the system software is re-entrant; that is, the same copy of object code may be used simultaneously by more than one process. For this reason, programs normally do not store variable data within the program itself. Instead, each process uses its own process workspace for data storage.

The system has predefined several control blocks (frames) per process that are reserved for that process, such as:

- primary control block (PCB)
- secondary control block (SCB)
- tertiary or debug control block (DCB)
- quaternary control block

The storage space most commonly used by a process is that in its PCB and SCB. The system automatically sets up an address register to allow direct, indirect, and relative addressing of these blocks:

R0 points to the PCB, byte 0

R2 points to the SCB, byte 0

The two other control blocks, the tertiary (debug) and quaternary control blocks, have no registers pointing to them. The debug control block is used solely by the assembly debugger, and should not be used by any other programs. The quaternary control block is used by some system

software (magnetic tape routines, for example) which temporarily set a register pointing to it; its use is reserved for future software extensions.

If a program must modify fields internal to itself, the program must be made non-re-entrant in order to prevent several processes from modifying data at the same time. A common method of accomplishing this is with a *lock byte*, illustrated below. The first process to execute the code *locks* it with an XCC instruction. Any other process attempting to execute the code must then wait until the first process unlocks the program after execution is completed:

```
        ORG   0
        TEXT  X'00'             Initial condition for lock
byte
        CMNT  *                 (Note usage of storage
        CMNT  *                 internal to program)
         .
         .
LOCKED  MCC   X'01',R2          Move "lock" flag to scratch
                                location;
        XCC   R2,R1             Exchange old lock; store
                                "lock" flag;
        BCE   R2,X'00',OK       If old flag was X'00', ok
                                to continue.
        RQM   *                 Else wait a while...
        B     LOCKED            and try again.
OK      EQU   *                 Start of non-shared code
         .
         .
         .
UNLOCK  MCC   X'00',R2          Unlock the "lock" flag
        XCC   R2,R1             Set R1 to unlocked
```

Note: *The instruction MCC X'00',R2 followed by XCC R2,R1 is equivalent to the single instruction MCC X'00',R1. The reason the first form is better than the second is that the XCC instruction guarantees that the memory location of the byte is not accessed by more than one processor at a time. The MCC X'00',R1 instruction would be adequate on a single-processor system, but not on a dual-processor system.*

Defining Additional Workspace

If a program needs more workspace than is available in the system-defined areas, the system allows the program to define storage elements or buffer areas. The unused frames PCB+30 and PCB+31 may be used as additional control blocks.

The following sequence of instructions is one way of setting up an address register to a scratch buffer:

```
.
.
MOV      R0,R3
SETR0+   R3,30          Set R3 with FID of PCB+30
CMNT     *              and displacement of zero
.
.
.
```

R3 can now be used to reference areas in the additional workspace, or functional elements that are addressed relative to R3. None of the system subroutines use R3, so that a program has to set up R3 only once in the above manner. However, an exit to TCL via the WRAPUP software resets R3 to PCB+10.

Ensuring Compatibility

In order to ensure that assembly programs are compatible on all Ultimate platforms, the following rules and conventions should be applied to all assembly language programs:

- Do not use the following characters in symbols (anything that may cause a PSYM file lookup):

  ^ ( ) - ? * | < > &

- Symbols defined in INCLUDE items should not be used prior to the INCLUDE statement.
- Only the following subroutines should be used to modify the return stack:

  INITRTN POPRTN MARKRTN RTNMARK

- In Ultimate PLUS implementations, items in the SM file become files. Because of the restrictions on filename size imposed on some UNIX implementations, it is necessary that the item-ID of any new mode be less than or equal to 12 characters.

Item-IDs for items in the OSYM and PSYM files can be up to 14 characters.

Scan character definitions (such as <SM> or <SM!AM>) that exceed 14 characters have been put into an SM INCLUDE item called SCAN-DEFS.

- The Ultimate PLUS implementation on HP systems requires that all 2-, 4-, and 6-byte data fields be aligned:
  - 2-byte (TLY) fields must be word aligned;
  - 4-byte (DTLY) fields must be on a double word boundary;
  - 6-byte (FTLY or SR) fields must be word aligned but *not* double word aligned. It is the low order two words (FID in case of an SR) that need to be double word aligned.

  Use the directives ALIGND and ALIGNS to align data definitions as follows:

```
        ALIGND *            Align for Double tally
LAB1    DTLY   X'12345'
*
        ALIGNS *            Align for Storage register
LAB2    ADDR   T%CONFIG
*
        ALIGNS *
LAB3    FTLY   X'12',x'3456'
```

  On the traditional systems, ALIGNS and ALIGND are synonyms of ALIGN.

- Access on word and double boundaries are not mandatory but highly desirable, given the impact on performance.

- In cases where the definition cannot be aligned (for example, the double tallies XNFID and XPFID which reference the forward and backward links), use the special instructions in the OSYM starting with UA_ (for UnAligned) followed by the normal OSYM entry. These instructions function as a flag to the 'virtual to C' translator, indicating the non-aligned nature of this data access. Such entries can be added freely to the OSYM file when needed. For example, the following statement

```
MOV     XNFID,OVRFLW
```

  should be changed to the following:

```
UA_MOV XNFID,OVRFLW
```

On all platforms except Ultimate PLUS on the HP, the UA_MOV instruction is equivalent to MOV.

- The following instruction should be used to clear the PCB:

```
MII    R14,R15,-1+ID.DATFRM.SIZE
```

This ensures that all the bytes in the PCB are cleared, regardless of the implementation or frame size.

- If you need to copy an unknown number of bytes then set a pointer to the data, use the ID.DSP.ADJ command to align the register, as follows:

```
MIID     R14,R15,<SM>
ID.DSP.ADJ R15
MOV      R15,SR2
```

- When allocating space for an array of SRs, either reserve eight bytes for each one, leaving the first two bytes of every definition unused, or use the PSYM entries ID.SRDEF.SIZE (word size of an SR definition) and ID.SRDEF.OFFS (word offset from the start of an aligned register to the start of a storage register defined from it). On an Ultimate PLUS implementation, these reserve eight bytes; on all other implementations, where the size of the SR can remain at six bytes, these reserve six bytes. The following is an example of the use of these instructions:

```
SR.BYT.SIZ  DEFN  2*ID.SRDEF.SIZE  Byte size  of
                                   SR definition
SR.BYT.OFS  DEFN  2*ID.SRDEF.OFFS  Byte size of
                                   SR offset
*
                .
                .
* Allocate size for array of 10 storage registers
            MOV   R15,SR2         Save ptr to array
            LOAD  10              Numb of SRs defined
            MUL   SR.BYT.SIZ      Byte size (6 or 8)
            SIT   R15             Skip array area
              .
              .
* Initialize array to zero
            MOV   SR2,R15         At array start
            LOAD  10              Count of SRs
```

```
LOOP   EQU    *
INC    R15,SR.BYT.OFS Skip 0 or 2 bytes
ZERO   R15;F0         Clear SR
BDNZ   T0,LOOP
```

- No assumptions should be made regarding the physical location of PCB or SCB elements. For example, many of the PSYM entries referring to the PCB or SCB were recently redefined.
- The BASIC runtime contains string instructions of the type NO_MII. On both the HP and the RS6000, data copies using the 'memcpy' library function are not guaranteed to occur in a 'left to right' motion. This means that this function can not be used for overlapping moves. The NO_... (No Overlap) is a flag to the C translator indicating that this particular string copy involves no data overlap, therefore allowing the use of 'memcpy'. Otherwise, data is copied in a slower way, byte by byte.

Notes

4 Assembler Instruction Set and Directives

An assembly *instruction* performs one operation. Each instruction assembles to one or more machine-executable object code instructions.

An assembly *directive* reserves program space, defines symbols for use as operands, or generates literal data within a program. Directives are different from instructions in that directives do not generate executable object code.

A *program* is a sequence of instructions and directives that perform a complete job or task. For information on the structure of programs and program lines, see Chapter 2, The Assembler.

In the following topics, each instruction and directive is described in detail in its own separate topic. The topics are presented in alphabetical order, according to the root mnemonic name of the instruction or directive.

The general syntax, operands, usage, and examples are given for each instruction.

Summary of the Instructions and Directives

The following summary lists the Ultimate Assembly Language Instruction set, divided into functional groups.

Arithmetic Instructions	ADD	MUL
	ADDX	MULX
	DEC	NEG
	DIV	SUB
	DIVX	SUBX
	INC	
Bit Instructions	BBS	SB
	BBZ	ZB
	MOV	
Character Scans and Moves	MCC	MITD
	MCI	SICD
	MIC	SID
	MII	SIDC
	MIID	SIT
	MIIDC	SITD
	MIIR	XCC
	MIIT	
Character Tests	BCA	BCNA
	BCE	BCNN
	BCH	BCNX
	BCHE	BCU
	BCL	BCX
	BCLE	BSTE
	BCN	

Conversions	MBD	MFX
	MBX	MSDB
	MBXN	MSXB
	MDB	MXB
	MFD	
Data Comparisons	BDLEZ	BHEZ
	BDHZ	BHZ
	BDHEZ	BL
	BDLZ	BLE
	BDNZ	BLEZ
	BDZ	BLZ
	BE	BNZ
	BH	BU
	BHE	BZ
Data Movement	LOAD	ONE
	LOADX	STORE
	MOV	ZERO
Directives	*	EP.ADDR
	ADDR	EQU
	ALIGN	FRAME
	CHR	FTLY
	CMNT	HTLY
	DEFx	INCLUDE
	DEFM	MTLY
	DEFN	MTLYU
	DEFNEP	ORG
	DEFNEPA	SR
	DTLY	TEXT
	EJECT	TLY
	END	
Terminal I/O	INP1B	OUT1B
	INP1BX	OUT1BX

Logical Operators	AND	SHIFT
	OR	XOR
Register Operators	BE	MOV
	BU	SETDSP
	DEC	SETR
	FAR	SRA
	INC	XRR
	LAD	
Transfer	B	EP
	BSL	HALT
	BSL*	ID.B
	BSLI	ID.RSA
	ENT	NEP
	ENT*	NOP
	ENTI	RTN
System MCALs	RQM	SLEEP
	SET.TIME	TIME

Operand Types

Table 4-1 summarizes the operand types; for more information, see Chapter 3.

Table 4-1. Operand and Symbol Types

Symbol Code	Description
A	address (both FID and displacement)
B	relatively addressed bit
C	relatively addressed character or byte (8 bits)
D	relatively addressed double tally (32 bits)
F	relatively addressed triple tally (48 bits)
H	relatively addressed half tally (8 bits)
L	locally defined label in this program
M	mode-id (16 bits); FID and entry point
N[1]	constant or literal value
R	address register
S	storage register
T	relatively addressed tally (16 bits)
X	address register in an external PCB

[1]An operand of type 'N' may be any of the following:

- An actual literal such as 3, X'82', or C'A'.
- '*'; the symbol for program location counter
- A symbol defined as having a literal value (symbol code=N), such as ID.DATA.SIZE; literal symbols may be predefined in the PSYM file (such as ID.DATA.SIZE or SM), or may be defined locally with the DEFN directive.

Virtual Addresses

An operand that resolves to a virtual address can be expressed in one of the following ways

- as a symbol name defined in PSYM or locally via a DEFx directive
- as an asterisk (*); specifies the current location of the program counter in this frame
- as a special operand of the form:

  Rn;Sd

  Rn address register R0-R15

  Sd displacement from the virtual address of Rn. S specifies the symbol type units (B,C,H,T,D,F) and d specifies the relative displacement

  The following example shows several special operands:

```
R4;B12          12th bit off R4
R14;D4          4th double tally off R14
R2;T7           7th tally off R2
R8;H8           8th half tally off R8
```

For more information on special operands, see the section, Immediate Symbols, in Chapter 2.

System Delimiters

The following symbols are used to denote the system delimiters:

SM	segment mark (X'FF')
AM	attribute mark (X'FE')
VM	value mark (X'FD')
SVM	subvalue mark (X'FC')
SB	start buffer (X'FB')

ADD
ADDX

The ADD and ADDX instructions add the contents (value) of the operand to the accumulator. The ADD form adds to a 4-byte field (D0); the ADDX form adds to a 6-byte field (FP0).

Syntax

ADD d	ADDX d
	ADDX f
ADD h	ADDX h
ADD n	ADDX n
ADD t	ADDX t

d double tally

f triple tally (for ADDX only)

h half tally

n numeric literal; if used, a 2-byte field is assumed (a range of -32,768 through +32,767). If a 1-byte literal (half tally) is being referenced, it should be defined separately using the HTLY directive. If the literal is outside the range of -32,768 through +32,767, a 4-byte literal must be separately defined using the DTLY directive, or a 6-byte literal via the FTLY directive.

The n form may generate a 2-byte literal at the end of the program when assembled for certain machines.

t tally

Description

The ADD instruction adds the operand value to the 4-byte field in the accumulator called D0. If the operand is a half tally (1 byte) or tally (2 bytes), it is internally sign-extended to form a 4-byte field before the add operation takes place.

The ADDX instruction adds the operand value to the 6-byte field in the accumulator called FP0. If the operand is a half tally (1 byte), tally (2 bytes), or double tally (4 bytes), it is internally sign-extended to form a 6-byte field before the add operation takes place.

The ADD and ADDX instruction cannot detect arithmetic overflow or underflow.

The addition does not affect the original operand or the other sections of the accumulator.

```
ADD    D4
ADD    H8
ADD    T4
ADDX   D4
ADDX   FP1
ADDX   H8
ADDX   T4
ADD    11
```

ADDR

The ADDR (define address) assembler directive defines a program address and creates a storage register containing that address. The storage register (symbol type S) is in unlinked format.

Syntax

label ADDR a
label ADDR n,m
label ADDR n,n

label specifies the symbol being defined

a defines both FID and displacement to specify the virtual address. If used, the address must have been previously defined via a DEFRA directive.

n,n
n,m virtual address to reserve for the symbol. The first operand is a literal (n) value that specifies the displacement of the generated virtual address. The second operand may be a literal (n) or a mode-id (m) that specifies the frame number (FID).

Description

The ADDR instruction sets up a symbol as a storage register pointing to data in an unlinked frame. (To define storage registers in linked frame format, use the SR directive.)

Six bytes of storage for the address are reserved at the current location counter, or the current location +1 if necessary to align on a word (even byte) boundary.

The ADDR instruction can be used to refer to data in other ABS frames. However, care needs to be taken if the ABS frames contains code. Data within such a frame may be in different locations on different implementations. This is because the object code for one implementation may be of a different size from that generated for another implementation. Accordingly, it is important to know whether the location of data referred to by an ADDR may vary by implementation.

If the ABS frame contains only data (for example, it contains tables), the data within the frame will be in the same locations on all implementations. In this case, the ADDR directive can be used to specify the location of data within the frame.

However, if a frame contains both code and data, more care is needed. To refer directly to an entry point of the frame, the EP.ADDR directive can be used (the ADDR directive should *never* be used to refer directly to entry points). To refer to data which is not an entry point, one of two techniques can be used:

- Place the data far enough after the last entry point of the frame (but before the next executable instruction) so that it is not overlaid by object code no matter what machine the frame is assembled for. The lowest "safe" address can be calculated by assuming four bytes of object code for each entry point (EP instruction), and an initial location (set by the FRAME directive) of 2.

  Once this is done, the data can be referred to by a simple ADDR directive as in the case of the data only frame above.

- Place the data immediately after the last entry point of the frame, and refer to it in terms of entry points, using the ADDR directive in conjunction with the DEFNEP or DEFNEPA directive. DEFNEP defines a byte offset to an entry point and DEFNEPA defines a (word-aligned) word offset.

```
FIELD    ADDR     X'1F0',223
         MOV      FIELD,R15    point R15 to above addr

%SYSTYP  DEFRA    X'6C',127    define as address, type A
SYSTYP   ADDR     %SYSTYP
         MOV      SYSTYP,R14   point R14 to above addr

DATA2    DEFNEPA  3            word-aligned entry point 3
LABL1    ADDR     DATA2,511

DATA8    DEFNEP   3            7 or E (machine-dependent)
LABL8    ADDR     DATA8,511
         MOV      LABL8,R15    point R15 to above addr
```

ALIGN
ALIGND
ALIGNS

The ALIGN directive aligns the assembler's location counter on an even-byte (word) boundary. On Ultimate PLUS implementations, the ALIGND directive aligns the assembler's location counter on a double word boundary. On Ultimate PLUS implementations, the ALIGNS directive aligns the assembler's location counter on a word boundary, but *not* double word boundary.

Syntax

ALIGN

Description

If the location counter is currently pointing to an odd byte, the ALIGN directive creates one byte of object code (X'00') in the program in order to move the counter down to the next even byte.

The ALIGN directive is typically used before a section of definitions (DEFx directives) to ensure even byte (word) alignment.

Note: The assembler automatically word-aligns literals that it creates itself (at the end of a program). It also word-aligns storage created by TLY, DTLY, FTLY, MTLY, SR, ADDR, and EP.ADDR directives.

The ALIGND and ALIGNS directives are used to that fields align correctly on Ultimate PLUS implementations. On all other implementations, these directives are identical to the ALIGN directive.

For more information on data alignment, see the section, Ensuring Compatibility, in Chapter 3.

AND

The AND instruction logically ANDs two bytes, and stores the result in the byte referenced by the first operand. The byte referenced by the second operand is unchanged.

Syntax

AND r,n
AND r,r

r address register

n numeric literal

Description

The logical AND instruction tests two bytes, one bit at a time, for a true (1) condition. If both bits are true (1), the result is true (1). If either is false, the result is false (0). For example,

```
Byte 1:     0000 0101
Byte 2:     1111 0011
            ---------
Result      0000 0001
```

The result is stored in the byte referenced by the first operand. The byte referenced by the second operand is unchanged.

```
        AND   R14,X'EF'

        AND   R14,R15
```

B

The B (branch) instruction transfers control unconditionally to a local label in the program.

Syntax

B l

l local label; must be defined in the same frame as the B instruction

Description

The B instruction immediately resolves the effective address of the local label and transfers program control to that address. To set up branches for other situations, use one of the following:

- To transfer control to an external label, use the ENT instruction.
- To define entry points at the start of the frame, use the EP instruction.
- To define branch tables (branch on a number used as an index) within a frame, use the ID.B instruction.

The EP instruction is used to indicate an entry point because the object code may differ from a simple B instruction when assembled for certain implementations. The ID.B instruction is used in branch tables to guarantee that the object code for each branch instruction has the same length; otherwise, the assembly process for some implementations may produce shorter code for some of the branches than for others, thereby destroying the table.

```
        B   LOW
        B   HIGH
          .
          .
LOW     EQU *
        ORG X'101'
HIGH    EQU *
```

BBS
BBZ

The BBS (branch bit set) instruction tests a specified bit and transfers control to a local label if the bit is set (1). The BBZ (branch bit zero) instruction tests and transfers control if the bit is not set (0).

Syntax

BBS b,l BBZ b,l

b specifies the bit to be tested; it may be a symbol or a special operand in the form Rn;Bd. (Special operands are described in the beginning of this chapter in the section, Virtual Addresses.)

l specifies the label in the current frame of the branch destination if the result of the test is "true".

Description

For BBS instructions, the referenced bit is tested, and if its value is 1 (set), program control transfers to the specified local label. If the value is 0, execution continues with the next instruction.

For BBZ instructions, the referenced bit is tested, and if its value is 0 (off), program control transfers. If the value is 1, execution continues with the next instruction.

```
TEST    DEFB R0,10
        BBS  TEST,LABL1
          .
          .
        BBZ  R15;B3,LABL1
          .
          .
        BBS  R15;B3,LABL1
          .
          .
LABL1   EQU *
```

BCA
BCNA

The BCA (branch character alphabetic) instruction tests a specified character and branches to a local label if the character is an alphabetic letter. The BCNA (Branch Character Not Alphabetic) instruction tests and branches if the character is not an alphabetic letter.

Syntax

BCA r,l BCNA r,l

r address register (R0-R15) that contains the virtual address of the character to be tested

l the label (in the current frame) of the branch destination if the result of the test is true

Description

For BCA instructions, the referenced character is tested, and if its value is in the ASCII character range of upper case letters A-Z (X'41' through X'5A') or lower case letters a-z (X'61' through X'7A'), the program branches to the specified local label. If the value is not in the range of alphabetic characters, execution continues with the next sequential instruction.

For BCNA instructions, the referenced character is tested, and if its value is **not** in the range of alphabetic characters (X'41' to X'5A' or X'61' to X'7A'), the program branches to the specified local label. If the value is within this range, execution continues with the next instruction.

```
        BCA   R15,LABL1
          .
          .
        BCNA  R15,LABL1

LABL1   EQU   *
```

BCE
BCU

The BCE (branch character equal) instruction compares one specified character against another and branches to a local label if the characters are equal. The BCU (branch character unequal) instruction compares and branches if the characters are not equal.

Syntax

BCE c,c,l	BCU c,c,l
BCE c,r,l	BCU c,r,l
BCE n,r,l	BCU n,r,l
BCE r,c,l	BCU r,c,l
BCE r,n,l	BCU r,n,l
BCE r,r,l	BCU r,r,l

c relatively addressed characters.

l the label (in the current frame) of the branch destination if the result of the test is true

n constant or literal

r address register (R0-R15) whose virtual address points to the character to be tested

Description

BCE and BCU compare two characters and use the results to determine program action.

For BCE instructions, the first referenced character is compared to the second referenced character, and if their ASCII values are equal, the program branches to the specified local label. If the values are not equal, execution continues with the next sequential instruction.

For BCU instructions, the referenced characters are compared, and if the values are *not* equal, the program branches to the specified local label. If the values are equal, then execution continues with the next instruction.

Note that a symbol of type c can be tested directly against another symbol of type c , but not against type n . To handle comparisons between c and n symbol types, you can use one of the following techniques:

- Use an SRA instruction to set an address register to point to the c type symbol; for example:

```
        SRA   R15,SC1          set R15 pointing to c symbol
        BCE   R15,C'$',OK
```

- Use DEFH or HTLY to define the c symbol as a half tally (symbol type h), then use a BE or BU instruction;for example:

```
HSC1    DEFH  SC1          define SC1 as a half tally
HLIT$   HTLY  C'$'         define $ as half tally literal
          .
          .
        BE    HSC1,HLIT$,OK
```

If the c,c,l form is used, a signed arithmetic comparison is made instead of an ASCII comparison. However, the result is correct since two different bit patterns are never evaluated as equal by the machine.

```
        BCE   R14,R15,LABL1
        BCU   R14,R15,LABL1
        BCE   X'20',R15,LABL1
        BCU   X'20',R15,LABL1
        BCE   PRMPC,R15,LABL1
        BCU   PRMPC,R15,LABL1
        BCU   CH0,CH9,LABL1
        BCE   CH0,CH9,LABL1
LABL1   EQU   *
```

BCH
BCHE
BCL

The BCH (branch character higher) instruction compares one specified character against another and branches to a local label if the value of the first character is greater than the second. BCHE (branch character higher or equal) compares and branches if the first value is greater than or equal to the second.

The BCL (branch character lower) instruction compares one specified character against another and branches to a local label if the value of the first character is less than the second. BCLE (branch character lower or equal) compares and branches if the first value is less than or equal to the second.

Syntax

		BCL c,c,l	BCLE c,c,l
BCH c,r,l	BCHE c,r,l	BCL c,r,l	BCLE c,r,l
BCH n,r,l	BCHE n,r,l	BCL n,r,l	BCLE n,r,l
BCH r,c,l	BCHE r,c,l	BCL r,c,l	BCLE r,c,l
BCH r,n,l	BCHE r,n,l	BCL r,n,l	BCLE r,n,l
BCH r,r,l	BCHE r,r,l	BCL r,r,l	BCLE r,r,l

c relatively addressed character

l label (in current frame) of the branch destination if the result of the test is true

n constant or literal

r address register (R0-R15) whose virtual address points to the character to be tested.

Description

BCH, BCHE, BCL, and BCLE compare two characters and use the results to determine program action.

For these instructions, the character addressed by the first operand is compared as an 8-bit logical field to the character addressed by the second operand. In a logical comparison, the lowest character is decimal 0 (X'00') and the highest character is decimal 255 (X'FF').

If the first character is higher than (BCH), higher than or equal to (BCHE), less than (BCL), or less than or equal to (BCLE) the second, then program control transfers to the third operand, which is a local label.

There are basically four cases, each with two ways of coding:

```
BCH A,B      or     BCL B,A
BCHE A,B     or     BCLE B,A
BCL A,B      or     BCH B,A
BCLE A,b     or     BCHE B,A
```

Note that a symbol of type c cannot be tested directly against another symbol of type c or type n . To handle comparisons between c and n symbol types, you can use one of the following techniques:

- Use an SRA instruction to set an address register to point to the c type symbol; for example:

```
SRA  R15,SC1       set R15 pointing to c symbol
BCH  R15,C'$',OK
```

- Use DEFH or HTLY to define the c symbol as a half tally (symbol type h), then use a BH{E} or BL{E} instruction;for example:

```
HSC1  DEFH  SC1          define SC1 as a half tally
HLIT$ HTLY  C'$'         define $ as half tally literal
        .
        .
BH    HSC1,HLIT$,OK
```

Note: *This coding performs an arithmetic comparison. In an arithmetic comparison, the lowest half tally is -128 (X'80') and the highest half tally is 127 (X'7F'). This means that the Ultimate system delimiters SM, AM, VM, and SVM (decimal 255-252, hexadecimal X'FF'-X'FB') are logically higher than all other ASCII characters but are arithmetically lower (as "negative" numbers).*

```
        BCH     R14,R15,LABL1
        BCL     R14,R15,LABL1
        BCHE    R14,R15,LABL1
        BCLE    R14,R15,LABL1
        BCH     X'20',R15,LABL1
        BCL     R15,X'20',LABL1
        BCL     X'20',R15,LABL1
        BCH     R15,X'20',LABL1
        BCLE    X'20',R15,LABL1
        BCLE    R15,X'20',LABL1
        BCH    PRMPC,R15,LABL1
        BCL    R15,PRMPC,LABL1
        BCL    PRMPC,R15,LABL1
        BCH    R15,PRMPC,LABL1
        BCLE   PRMPC,R15,LABL1
        BCLE   R15,PRMPC,LABL1
LABL1   EQU *
```

BCN
BCNN

The BCN (branch character numeric) instruction tests a specified character and branches to a local label if the character is a number. The BCNN (branch character not numeric) instruction tests and branches if the character is not a number.

Syntax

BCN r,l BCNN r,l

r address register (R0-R15) that contains the virtual address of the character to be tested

l the label (in the current frame) of the branch destination if the result of the test is true

Description

The BCN instruction tests the specified character, and if its value is in the ASCII character range of numbers 0-9 (X'31' through X'39'), program control transfers to the specified local label. If the value is not numeric, execution continues with the next sequential instruction.

The BCNN instructions tests the specified character, and if its value is *not* in the ASCII character range of numbers 0-9 (X'31' to X'39'), program control transfers. If the value is numeric (within this range), execution continues with the next instruction.

```
        BCN    R15,LABL1
        BCNN   R15,LABL1
          .
LABL1   EQU    *
```

BCNA

The BCNA instruction branches if a character if not alphabetic. See the BCA instruction for details.

BCNN

The BCNN instruction branches if a character is not numeric. See the BCN instruction for details.

BCNX

The BCNX instruction branches if a character is not hexadecimal. See the BCX instruction for details.

BCU

The BCU instruction branches if a character is not equal to another. See the BCE instruction for details.

BCX
BCNX

The BCX (branch character hexadecimal) instruction tests a specified character and branches to a local label if the character is in the hexadecimal number range (0-F). The BCNX (branch character not hexadecimal) instruction tests and branches if the character is not in the hexadecimal range.

Syntax

BCX r,l BCNX r,l

r address register (R0-R15) that contains the virtual address of the character to be tested

l the label (in the current frame) of the branch destination if the result of the test is true

Description

The BCX instruction tests the specified character, and if its value is in the ASCII character range of numbers 0-9 (X'31' through X'39') or upper case letters A-F (X'41' through X'46'), program control transfers to the specified local label. If the value is not a hexadecimal number, execution continues with the next sequential instruction.

The BCNX instruction tests the specified character, and if its value is *not* in the ASCII character range of 0-9 or A-F (X'31' to X'39' or X'41' to X'46'), program control transfers. If the value is hexadecimale (within this range), execution continues with the next instruction.

```
        BCX   R15,LABL1
        BCNX  R15,LABL1
LABL1   EQU   *
```

BDHZ
BDHEZ
BDLZ
BDLEZ

The BDHZ, BDHEZ, BDLZ, and BDLEZ instructions decrement a relatively addressed operand and then compare it to zero. BDHZ (branch decrementing higher than zero) transfers control to a local label if the resultant value is higher than zero. BDHEZ (branch decrementing higher/equal zero) transfers control if the value is higher than or equal to zero. BDLZ (branch decrementing less than zero) transfers control if the value is less than zero. BDLEZ (branch decrementing less/equal zero) transfers control if the value is less than or equal to zero.

Syntax

BDHZ d,l	BDHEZ d,l	BDLZ d,l	BDLEZ d,l
BDHZ d,d,l	BDHEZ d,d,l	BDLZ d,d,l	BDLEZ d,d,l
BDHZ d,n,l	BDHEZ d,n,l	BDLZ d,n,l	BDLEZ d,n,l
BDHZ f,l	BDHEZ f,l	BDLZ f,l	BDLEZ f,l
BDHZ f,f,l	BDHEZ f,f,l	BDLZ f,f,l	BDLEZ f,f,l
BDHZ h,l	BDHEZ h,l	BDLZ h,l	BDLEZ h,l
BDHZ h,h,l	BDHEZ h,h,l	BDLZ h,h,l	BDLEZ h,h,l
BDHZ t,l	BDHEZ t,l	BDLZ t,l	BDLEZ t,l
BDHZ t,t,l	BDHEZ t,t,l	BDLZ t,t,l	BDLEZ t,t,l
BDHZ t,n,l	BDHEZ t,n,l	BDLZ t,n,l	BDLEZ t,n,l

d double tallies

f triple tallies

h half tallies

n numeric literal

t tallies

l the label (in the current frame) of the branch destination if the result of the test is true

If operand 1 is a tally or double tally, operand 2 may be a numeric literal (n); the literal assembles as the same symbol type as operand 1.

Description

These instructions take the place of a DECrement followed by a conditional branch instruction, and are usually used in loop controls.

If only one operand is specified, the value at the effective address is decremented by one (1). If two operands are specified, the value at the effective address of operand 2 is subtracted from the value at the address

of operand 1. Then the specified condition is tested and, if true, the specified branch is taken.

Note: *If the second operand in a BDHEZ instruction is negative, decrementing the first operand by the second operand will not detect a sign change if a positive first operand overflows, producing a negative number.*

To loop through a section of code, the following can be used:

```
        MOV   COUNT,CTR1  Set loop counter for iterations
REPEAT  BDLZ  CTR1,QUITLP
          .
          .
        B     REPEAT      Repeat the cycle
QUITLP  EQU   *           Termination of loop
```

This example does not execute the loop body if the loop count is initially zero or negative. Compare this to the following example:

```
        MOV   COUNT,CTR1
XLOOP   EQU   *
          .
          .
        BDLEZ CTR1,XLOOP
```

This also loops for the count in CTR1, but always executes at least once.

BDZ
BDNZ

The BDZ (branch decrementing if zero) and BDNZ (branch decrementing if not zero) instructions decrement a relatively addressed operand and then compare it to zero. BDZ decrements, then tests and transfers control if the resultant value is zero. BDNZ decrements, then tests and transfers control if the value is not zero.

Syntax

BDZ d,l	BDNZ d,l
BDZ d,d,l	BDNZ d,d,l
BDZ d,n,l	BDNZ d,n,l
BDZ f,l	BDNZ f,l
BDZ f,f,l	BDNZ f,f,l
BDZ h,l	BDNZ h,l
BDZ h,h,l	BDNZ h,h,l
BDZ t,l	BDNZ t,l
BDZ t,t,l	BDNZ t,t,l
BDZ t,n,l	BDNZ t,n,l

d double tallies

f triple tallies

h half tallies

n numeric literal

t tallies

l the label (in the current frame) of the branch destination if the result of the test is true

If operand 1 is a tally or double tally, operand 2 may be a numeric literal (n); the literal assembles as the same symbol type as operand 1.

Description

These instructions take the place of a DECrement followed by a conditional branch instruction, and are usually used in loop controls.

If only one operand is specified, the value at the effective address is decremented by one (1). If two operands are specified, the value at the effective address of operand 2 is subtracted from the value at the address of operand 1. Then the specified condition is tested and, if true, the branch is taken.

```
        MOV   100,CTR1   Set loop counter for 100
                         iterations
REPEAT  EQU   *          Start of loop
          .
          .
        BDNZ  CTR1,REPEAT
```

Note that the body of the loop executes at least once with this logic. Compare this to the example shown for BDHZ.

BE
BU

The BE (branch equal) instruction compares two relatively addressed operands or virtual addresses and branches to a local label if the operand values are equal. The BU (branch unequal) instruction compares and branches if the values are not equal.

Syntax

BE	BU
BE d,d,l	BU d,d,l
BE d,n,l	BU d,n,l
BE f,f,l	BU f,f,l
BE h,h,l	BU h,h,l
BE n,d,l	BU n,d,l
BE n,t,l	BU n,t,l
BE r,r,l	BU r,r,l
BE r,s,l	BU r,s,l
BE s,r,l	BU s,r,l
BE t,n,l	BU t,n,l
BE t,t,l	BU t,t,l

d double tally

f triple tally

h half tally

n numeric literal

t tally

l the label (in the current frame) of the branch destination if the result of the test is true

If the operands are tally-types, they must be of the same length, that is, one byte (type H), two bytes (type T), four bytes (type D) or six bytes (type F).

If one operand is a tally or double tally, the other operand may be a literal (n); the literal assembles as the same symbol type as the relatively addressed operand.

Description

BE and BU compare two values of the same length or symbol type and use the to determine program action. If a 1-byte or 6-byte literal or constant value needs to be compared, it must be defined as a symbol

using an HTLY or DEFH directive (1-byte value), or an FTLY or DEFF directive (6-byte value).

If the register format is used, the virtual address in any storage register operands must be normalized prior to executing the BE/BU instruction; see the FAR instruction for more information.

For BE instructions, the first referenced operand is compared to the second referenced operand, and if their arithmetic values are equal, then program control transfers to the specified local label. If the values are not equal, then execution continues with the next sequential instruction.

For BU instructions, the referenced operands are compared, and if the values are not equal, then program control transfers. If the values are equal, then execution continues with the next instruction.

Note: When testing registers, this test for "equal or unequal" is the only option. There is no way to test which register is "less than" or "higher than" the other. When testing tally-type operands, however, alternative tests are possible; see the BH, BHE, BL, and BLE instructions.

If a tally-type operand is to be compared with a literal or constant (n) value of zero, it is more efficient and clearer to use another instruction that is designed for comparisons with zero (such as BZ, BNZ, or BHZ); for example, use

```
BZ CTR1,QUIT
```

rather than

```
BE CTR1,0,QUIT
```

Note that half tally and triple tally symbols (types h and f) cannot be tested directly against a constant or literal (type n). There are two ways to handle this condition:

- Use an SRA instruction to set an address register to point to the h or f type symbol; for example:

```
SRA  R15,H7          set R15 pointing to h symbol
BCE  R15,10,OK
```

- Define the constant or literal as a half tally or triple tally in the program; for example:

```
FLIT  FTLY 0,X'F23AB3FC'   Define a constant of type F
        .
        .
      BE  FP0,FLIT,OK
```

```
      BE   R14,R15,LABL1
      BU   R14,R15,LABL1
      BU   SR5,R15,LABL1
      BE   R15,SR5,LABL1
      BU   D0,D1,LABL1
      BE   D0,D1,LABL1
      BU   FP0,FP1,LABL1
      BE   FP0,FP1,LABL1
      BU   H8,H9,LABL1
      BE   H8,H9,LABL1
      BU   T0,T4,LABL1
      BE   T0,T4,LABL1
LABL1 EQU  *
```

BH
BHE
BL
BLE

The BH (branch higher) instruction compares one specified tally-type operand against another and branches to a local label if the value of the first operand is greater than the second. BHE (branch higher/equal) compares and branches if the first value is greater than or equal to the second.

The BL (branch lower) instruction compares one specified operand against another and branches to a local label if the value of the first character is less than that of the second. BLE (branch lower/equal) compares and branches if the value is less than or equal to the second.

Syntax

BH d,d,l	BHE d,d,l	BL d,d,l	BLE d,d,l
BH d,n,l	BHE d,n,l	BL d,n,l	BLE d,n,l
BH f,f,l	BHE f,f,l	BL f,f,l	BLE f,f,l
BH h,h,l	BHE h,h,l	BL h,h,l	BLE h,h,l
BH n,d,l	BHE n,d,l	BL n,d,l	BLE n,d,l
BH n,t,l	BHE n,t,l	BL n,t,l	BLE n,t,l
BH t,n,l	BHE t,n,l	BL t,n,l	BLE t,n,l
BH t,t,l	BHE t,t,l	BL t,t,l	BLE t,t,l

d double tallies

f triple tallies

h half tallies

n numeric literal

t tallies

l the label (in the current frame) of the branch destination if the result of the test is true

The operands must be of the same length, that is, one byte (type H), two bytes (type T), four bytes (type D) or six bytes (type F).

If one operand is a tally or double tally, the other operand may be a literal (n); the literal assembles as the same symbol type as the relatively addressed operand.

Description

These instructions compare two values of the same length or symbol type and use the results to determine program action.

The first referenced operand is compared to the second referenced operand. The operands are compared as two's-complement (signed) integers. If the first operand is arithmetically higher than (BH), higher than or equal to (BHE), less than (BL), or less than or equal to (BLE) the second, program control transfers to the specified local label. If the result of the comparison is "false", execution continues with the next sequential instruction.

If a 1-byte or 6-byte literal or constant value needs to be compared, it must be defined as a symbol using an HTLY or DEFH directive (1-byte value), or an FTLY or DEFF directive (6-byte value).

If an operand is to be compared with a literal or constant (n) value of zero, it is more efficient and clearer to use another instruction that is designed for comparisons with zero (such as BHZ or BLZ); for example, use

```
BHZ CTR1,QUIT
```

rather than

```
BH CTR1,0,QUIT
```

Note that half tally and triple tally symbols (types h and f) cannot be tested directly against a constant or literal (type n). Since this is an arithmetic comparison, there is only one way to handle this condition:

- Define the constant or literal as a half tally or triple tally in the program; for example:

```
FLIT  FTLY 0,X'F23AB3FC'   Define a constant of type F
        .
        .
      BL   FP0,FLIT,OK
```

```
BLE     D0,D1,LABL1
BHE     D0,D1,LABL1
BH      D0,D1,LABL1
BL      D0,D1,LABL1
BLE     FP0,FP1,LABL1
BL      FP0,FP1,LABL1
BLE     H8,H9,LABL1
BHE     H8,H9,LABL1
BH      H8,H9,LABL1
BLE     T0,T4,LABL1
BL      T0,T4,LABL1
LABL1   EQU  *
```

BHZ
BHEZ
BLZ
BLEZ

The BHZ (branch higher than zero) instruction compares one specified operand against zero and branches to a local label if the operand value is higher than zero. BHEZ (branch higher/equal zero) compares and branches if the value is higher than or equal to zero.

The BLZ (branch lower than zero) instruction compares one specified operand against zero and branches to a local label if the operand value is less than zero. BLEZ (branch lower/equal zero) compares and branches if the value is lower than or equal to zero.

Syntax

BHZ d,l	BHEZ d,l	BLZ d,l	BLEZ d,l
BHZ f,l	BHEZ f,l	BLZ f,l	BLEZ f,l
BHZ h,l	BHEZ h,l	BLZ h,l	BLEZ h,l
BHZ t,l	BHEZ t,l	BLZ t,l	BLEZ t,l

d double tallies

f triple tallies

h half tallies

t tallies

l the label (in the current frame) of the branch destination if the result of the test is true

Description

The BHZ, BHEZ, BLZ, and BLEZ instructions compare a tally-type symbol against zero to determine program action. These instructions are faster and clearer than the equivalent BH, BHE, BL, and BLE instructions used with a literal of zero as one of the operands.

The value at the effective address is tested against zero to determine program action. If the result is "true", program control transfers to the specified local label. If the result is "false", then execution continues with the next sequential instruction.

```
        BHEZ D4,LABL1
        BHEZ FP1,LABL1
        BHEZ H8,LABL1
        BHEZ T4,LABL1
        BHZ  D4,LABL1
        BHZ  FP1,LABL1
        BHZ  H8,LABL1
        BHZ  T4,LABL1
        BLEZ D4,LABL1
        BLEZ FP1,LABL1
        BLEZ H9,LABL1
        BLEZ T4,LABL1
        BLZ  D4,LABL1
        BLZ  FP1,LABL1
        BLZ  H8,LABL1
        BLZ  T4,LABL1
LABL1   EQU  *
```

BL
BLE

The BL instruction branches if an operand value is lower than another. The BLE instruction branches if an operand value is lower than or equal to another. See the BH instruction for details.

BLZ
BLEZ

The BLZ instruction branches if an operand value is less than zero. The BLEZ instruction branches if an operand value is less than or equal to zero. See the BHZ instruction for details.

BNZ

The BNZ instruction branches if an operand value is not zero. See the BZ instruction for details.

BSL

The BSL (branch subroutine location) instruction stores the address of the next sequential instruction in the return stack and branches unconditionally to a specified subroutine location.

Syntax

```
BSL l
BSL m
BSL n,m
```

l the label (in the current frame) of the subroutine

m mode-id (external entry point), which defines a frame number and offset for a subroutine located outside the current program frame

n,m n specifies the entry point (0-F) and the m is a mode-id

Description

The BSL instruction is used to branch to an internal or external subroutine when a return to the main program after the conclusion of the subroutine is desired.

If a mode-id format is used, the m operand may be a globally defined symbol of type M in the PSYM file, or it may be defined with a DEFM assembler directive (either within the local program or in an INCLUDEd program).

The BSL instruction first stores the return address where program execution will continue after returning from the subroutine. The address is stored in the next available return stack entry, which is always pointed to by the return stack pointer (RSCWA). Then RSCWA is incremented by four, to point to the next available entry. (This return address is the location, less one, of the instruction following the BSL.)

Next, the BSL instruction resolves the effective address of the label or mode-id by modifying the runtime program counter (R1). Program control is then transferred to that address.

Note that the same subroutine can be called either locally from within the frame or externally by establishing an entry point. When calling a subroutine in the same frame that happens to have an externally established entry point, the BSL executes slightly faster if the local label is used instead.

Note: *Because of the word alignment requirement on software machines, if data follows the BSL instruction, it must fill out a full word or the called subroutine must account for the possible extra filler byte. For example:*

```
BSL   CRLFPRINT
TEXT  C'1234',X'FF'
LOAD  5
```

In the above case, there are an odd number of bytes in the text string, but the LOAD code will begin on the next even address, leaving a 1-byte hole which the subroutine must deal with. (CRLFPRINT, an Ultimate system subroutine, guarantees execution at the next even address after text on software machines, by use of the ID.RSA instruction; see ID.RSA, listed in this section, for details.)

If the instruction causes more than eleven entries in the return stack, the Debugger is entered with a Return Stack Full trap condition. In this case, the first entry in the stack is overwritten with the location of the instruction causing the abort.

See also the RTN instruction to return from a subroutine.

Note: *The subroutine return stack is part of the PCB and is described in Chapter 3.*

Example of defining an external mode-id:

```
EXTS    DEFM  10,500      Define a constant of type M as
        CMNT  *           entry point #10 in frame 500.
          .
          .
        BSL   EXTS        Transfers control to FID 500,
        CMNT  *           e.p. 10, at offset 21 (X'15')
        CMNT  *           on firmware machines.  Returns
        CMNT  *           when subroutine executes RTN.
```

Example of a local/external subroutine:

```
        FRAME 500
          .
          .
        EP    EXT.S       Entry point for external call
        CMNT  *           in branch table at start of
        CMNT  *           prog
          .
          .
        BSL   EXT.S       Local call of same subroutine
          .
          .
EXT.S   EQU   *           Subroutine local label
          .               (body of subroutine)
          .
        RTN
```

BSL*

The BSL* (branch subroutine location indirect) instruction stores the address of the next sequential instruction in the return stack and branches unconditionally to the location referenced by the specified operand.

Syntax

BSL* t

t tally symbol, which contains the branch destination address (subroutine's mode-id)

Description

The BSL* instruction performs the same function as a LOAD t instruction followed by a BSLI instruction.

On firmware machines, BSL* is a macro that loads the accumulator (T0) with the current content of the t operand, and then executes the BSLI instruction. T1 is also destroyed because of sign-extension in loading the accumulator. On software machines, the same operation may occur without affecting T0 or T1. Therefore, the contents of the accumulator are not guaranteed to be in a predictable state after execution of a BSL* instruction.

See BSLI and BSL for more details about how subroutine branches operate.

BSLI

The BSLI (branch subroutine location, indirect) instruction stores the address of the next sequential instruction in the return stack and branches unconditionally to the location specified in T0 of the PCB.

Syntax

BSLI

Description

The BSLI instruction operates identically to the BSL instruction, except that the subroutine address is variable and is obtained from the low-order two bytes of the accumulator, T0, instead of from an operand.

T0 must contain the branch destination mode-id (entry point in the high-order 4 bits and FID in the lower-order 12 bits), which can be loaded into it from a local label, an external label or by converting an ASCII string.

See the BSL instruction for details on calling subroutines.

```
        ALIGN *             Ensure TABLE is word-aligned
TABLE   EQU  *              Start of table
        MTLY 0,SUB1         Define subroutine exits
        MTLY 7,SUB4
          .
          .
        SRA  R15,TABLE      Set to start of table
        INC  R15,CTR1       Index into table
        LOAD R15;T0         Load Tally from table
        BSLI *              Call subroutine
        CMNT *              Return here when subroutine
        CMNT *              executes RTN
```

BSTE

The BSTE (branch string test equal) instruction compares one string to another (character by character) until a specified delimiter is reached, then branches to a local label if the strings are equal.

Syntax

BSTE r,r,n,l

r address registers (R0-R15) that contain the virtual address of the two strings to be compared

n constant or literal (symbol type n) that specifies the delimiting value (usually a system delimiter)

l the label (in the current frame) of the branch destination if the result of the test is true

Description

The BSTE instruction compares two strings and uses the result to determine program action.

Two different registers should be used to reference the strings, since unpredictable results may occur if both register operands refer to the same register.

The two address register operands are incremented by one before the initial comparison is made.

The character addressed by the first operand is tested as a 1-byte logical field against that addressed by the second operand. In a logical comparison, the lowest character is decimal 0 (X'00') and the highest character is decimal 255 (X'FF').

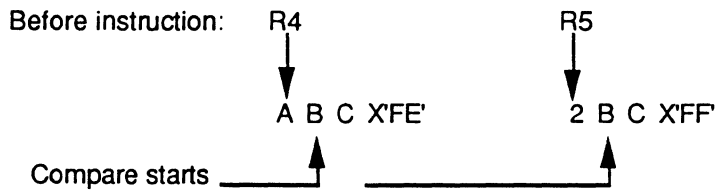
This operation is repeated until one of the following conditions is met:

- One character is logically higher than or equal to the third operand, but the other is not. BSTE terminates with the strings considered **unequal**.
- Both characters are logically higher than or equal to the third operand. BSTE terminates with the strings considered **equal**.

Note: The terminating characters need not be the same, as long as they are both higher than the third operand.

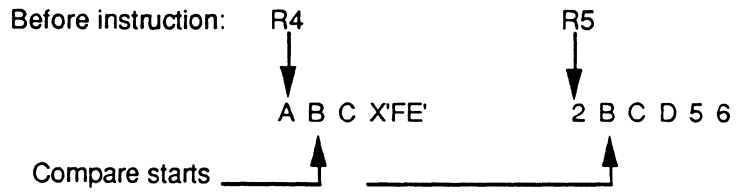
- The two characters are both less than the third operand, and are not equal. BSTE terminates with the strings considered **unequal.**

- `BSTE R4,R5,X'FE',LABEL`



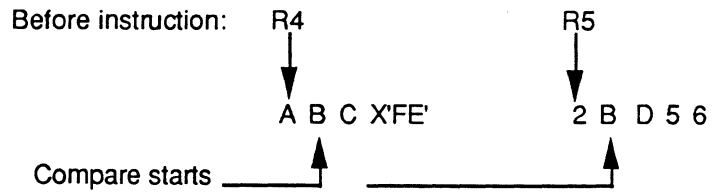
Strings are considered equal, and a branch is taken to LABEL.

- `BSTE R4,R5,X'FC',LABEL`



Strings are considered unequal, and no branch is taken.

- `BSTE R4,R5,X'FC',LABEL`



Strings are considered unequal, and no branch is taken.

BU

The BU instruction branches if an operand value is not equal to another operand value. See the BE instruction for details.

BZ
BNZ

The BZ (branch on zero) instruction compares a relatively addressed operand to zero, and transfers control if the value is zero. The BNZ (branch on not zero) compares and transfers control if the value is not zero.

Syntax

BZ d,l	BNZ d,l
BZ f,l	BNZ f,l
BZ h,l	BNZ h,l
BZ t,l	BNZ t,l

d double tallies

f triple tallies

h half tallies

t tallies

l the label (in the current frame) of the branch destination if the result of the test is true

Description

The BZ and BNZ instructions compare a symbol value against zero and use the result to determine program action. These instructions are faster and clearer than the equivalent BE and BU instructions used with a literal of zero as one of the operands.

The referenced operand is compared to zero, and if the value is zero (BZ) or not zero (BNZ), program control transfers to the specified local label. Otherwise, execution continues with the next sequential instruction.

```
        BNZ   D4,LABL1
        BNZ   FP1,LABL1
        BNZ   H8,LABL1
        BNZ   T4,LABL1
        BZ    D4,LABL1
        BZ    FP1,LABL1
        BZ    H8,LABL1
        BZ    T4,LABL1
LABL1   EQU   *
```

CHR

The CHR directive reserves one byte of storage and sets up the symbol in the label field to be of type c (Character).

Syntax

{symbol} CHR n

symbol — if present, appears in the label field of the instruction and specifies the symbol name of the character; if not present, CHR simply stores the value of the operand at the current program location counter as a single byte

n — specifies the constant or literal value to be assigned to the character symbol.

Description

The CHR assembler directive sets up a symbol of type c (character).

One byte of storage is reserved for the symbol value.

```
        CHR  AM

STAR    CHR  C'*'
```

CMNT

The CMNT (Comment) directive places a comment line in the source program.

Syntax

CMNT text

text any characters up to a maximum that fit on one program line

Description

The CMNT assembler directive is an alternative to the use of an asterisk (*) in the label field; both specify that the source line is a comment and is to be ignored by the assembler.

The first word in the text is treated as an operand by MLIST or the Editor AS format. It may be desirable to put a dummy operand (such as "*") after CMNT to force the real comment to be entirely in the comment field of the listing.

This directive can be used to align comments in the MLISTing. It can also be used to define a label as an alternative to the "label EQU *" form.

```
LABEL1  CMNT THIS            LINE HAS NO DUMMY OPERAND
LABEL2  CMNT *               However, this and following
        CMNT *               lines of comments are
        CMNT *               aligned.
```

DEC (Data)
INC (Data)

The DEC (decrement) instruction used with a symbol operand decrements the relatively addressed operand value. The INC (increment) instruction used with a symbol operand increments the relatively addressed operand value. (For information on decrementing and incrementing a register, see the next topic.)

Syntax

```
DEC d          INC d
DEC d,d        INC d,d
DEC d,n        INC d,n
DEC f          INC f
DEC f,f        INC f,f
DEC h          INC h
DEC h,h        INC h,h
DEC t          INC t
DEC t,n        INC t,n
DEC t,t        INC t,t
```

d double tally

f triple tally

h half talliy

t tally

n numeric literal

If operand 1 is a tally or double tally, operand 2 may be a numeric literal; the literal assembles as the same symbol type as operand 1.

Description

If only one operand is specified, the value at the effective address is decremented or incremented by one. The DEC and INC instructions with one symbol operand are always preferable to the logically equivalent forms "DEC operand,1", or "INC operand,1", which are slower instructions that also use more object code.

If two operands are specified, the value at the effective address of operand 1 is decremented or incremented by the value at the effective address of operand 2. The two operands must be of the same length.

The DEC and INC instructions with two operands are used whenever a value needs to be decremented or incremented by a value other than 1.

Symbols of type F and H cannot be directly decremented or incremented by a constant or literal (type N). The FTLY or HTLY directive should be used to define a local constant to use as the second operand.

Caution *PCB fields associated with address registers (RnDSP, RnFID, RnDSPFID) should not be modified with these instructions. Instead, use the INC or DEC register instructions (see next topic), or the SETR, SETDSP, or MOV instructions to change the register's virtual address.*

Arithmetic overflow or underflow cannot be detected. For example, if a DEC instruction is used with a two-byte tally, the value -32768 (X'8000') wraps around to 32767 (X'7FFF').

```
DEC  D4
DEC  FP1
DEC  H8
DEC  T4
INC  D0
INC  FP1
INC  H0
INC  T0
DEC  D4,D0
DEC  FP1,FP0
DEC  H8,H0
DEC  T1,T0
INC  RECORD,D0
INC  FP1,FP0
INC  H8,H0
INC  T1,T0
```

DEC (Register)
INC (Register)

The DEC (decrement) instruction used with a register operand decrements the virtual address in the register. The INC (increment) instruction used with a register operand increments the virtual address in a register.

Syntax

DEC r	INC r
DEC r,n	INC r,n
DEC r,t	INC r,t

r address register

n constant or literal

t tally

The first operand must be a register. If a second operand is present, it may be a tally (type t) or a constant or literal (type n).

Description

If only a register operand is specified, the virtual address in the address register is decremented (DEC) or incremented (INC) by one. These instructions are always preferable to the logically equivalent forms "DEC r,1", or "INC r,1", which are slower instructions that also use more object code.

If two operands are specified, the virtual address in the address register is decremented (DEC) or incremented (INC) by the second operand value.

If the resultant address crosses a frame boundary, and the register is in **unlinked** mode, the debugger is entered with a trap condition indicating CROSSING FRAME LIMIT.

If the resultant address crosses a frame boundary, and the register is in **linked** mode, the system may attempt to normalize the address, depending on instruction type and machine type. (Normalization of an address means to resolve the address to an offset (up to the size of a frame) within a particular frame. This may require traversing several frames in a linked set, reading the link fields to determine subsequent

frames in the chain, until the required number of bytes have been skipped over.)

For the single-operand INC and DEC register instructions, the system always attempts to normalize the resultant address if it crosses a frame boundary.

For the double-operand instructions, the address may be left unnormalized on firmware machines. In this case, normalization does not take place until the next instruction is executed which references data via the address register. In order to guarantee attachment and normalization after incrementing or decrementing an address register by the value of another operand, a FAR instruction can be used after the INC or DEC instruction. This may be useful when an XMODE routine has been set up to handle end-of-linked-chain conditions.

If the beginning of a linked set of frames is reached during the normalization process, the assembly debugger is entered with a trap condition indicating Backward Link Zero.

If the end of a linked set is reached during the normalization process, the XMODE (exception mode identifier) tally is tested to determine the program action:

- If XMODE is non-zero, a subroutine call is executed to that address, to allow special handling of this condition (usually linking additional frames).
- If XMODE is zero, the assembly debugger is entered with a trap condition indicating Forward Link Zero.

Incrementing an address register by a negative value has the same effect as decrementing it by a positive value, and decrementing a register by a negative value has the same effect as incrementing it by a positive value.

```
DEC  R15
INC  R15
DEC  R15,T0
INC  R15,T0
```

DEFx

The DEFx (define symbol) assembler directives are used to define a local symbol for a program or to associate additional symbol names and types with previously defined symbols or their locations.

Syntax

The following set of formats (Set 1) are used to define symbols in terms of literal base register and offset values:

```
symbol DEFB r,n
symbol DEFC r,n
symbol DEFD r,n
symbol DEFF r,n
symbol DEFH r,n
symbol DEFS r,n
symbol DEFT r,n
symbol DEFX r,n
symbol DEFRA n,n  (n,n = byte offset,FID)
```

The following set of formats (Set 2) are used to define symbols in terms of previously-defined symbols:

```
symbol DEFC r,c                    r,h
symbol DEFD       r,d                        r,s     r,t
symbol DEFF       r,d     r,f                r,s     r,t
symbol DEFH r,c                    r,h
symbol DEFS       r,d     r,f      r,s       r,t
symbol DEFT       r,d                        r,s     r,t
symbol DEFX       r,d                        r,s     r,t

symbol DEFC                        h
symbol DEFD                                          t
symbol DEFF       d                          s       t
symbol DEFH c
symbol DEFS               f
symbol DEFS       d                                  t
symbol DEFT       d       f                  s
```

The tally (t) and double tally (d) forms overlay the existing ??

The following special formats (Set 3) are used to define one symbol as a subfield of another:

```
symbol DEFDL  s    (overlays lower DTLY of storage reg.)
symbol DEFHL  t    (overlays lower HTLY of a TLY)
symbol DEFHU  t    (overlays upper HTLY of a TLY)
symbol DEFTL  d    (overlays lower TLY of a DTLY)
symbol DEFTL  s    (overlays lower TLY of storage reg.)
symbol DEFTM  s    (overlays middle TLY of storage reg.)
symbol DEFTU  d    (overlays upper TLY of a DTLY)
symbol DEFTU  s    (overlays upper TLY of storage reg.)
```

Description

The symbol type being defined is the last character of the instruction mnemonic (such as B in DEFB) or the next to last character (such as T in DEFTL).

The DEFx directives define a local symbol as one of the following symbol types:

a	address
b	bit
c	character
d	double tally
f	triple tally
h	half tally
s	storage register
t	tally
x	external register

upper half-tally of tally (HU)
lower half-tally of tally (HL)
upper tally of double tally or storage register (TU)
lower tally of double tally or storage register (TL)
middle tally of storage register (TM)
lower double tally of storage register (DL)

To define a mode-id, see the separate topic DEFM. To define literals, see the separate topic DEFN.

The first set of formats (Set 1) is used to define a symbol for the first time in a program. These formats have two operands:

r an address register (R0-R15 or synonym such as IS, IR, or TS)

n the offset; may be a symbol or expression that resolves to an offset.

The second set of formats (Set 2) is used to associate additional symbol names with previously-defined symbols. These formats may have one or two operands:

r an address register (R0-R15 or synonym such as IS, IR, or TS)

c character

d double tally

f triple tally

h half tally

s storage register

t tally

The third set of formats (Set 3) is used to associate additional symbol names to subfields of previously-defined symbols. These formats have one operand that identifies the symbol type of the original symbol:

d double tally

s storage register

t tally

In essence, a symbol may be defined in terms of:

- a base register and offset:

```
DEFT R2,11
```

- a previously defined symbol having a register and offset:

```
DEFT CTR17
```

- a previously defined symbol having a register and offset, with a different register:

```
DEFT R13,CTR17
```

  This is useful when accessing a table via different registers.

The symbol in the label field of the DEFx directive is created with the specified type.

Initial Symbol Definition vs. Additional Symbol Definitions

The initial definition of a symbol is used to generate a symbol in TSYM to define a location that was previously unknown and assign a symbol name to it.

Additional definitions are used to create new symbols based on an existing symbol in PSYM or TSYM. Either the symbol type, the displacement, or base register of the new symbol is different from that of the original symbol, and a new symbol name is assigned as well.

All initial symbol definitions require two operands: register and displacement.

For additional definitions, the symbol used as the operand provides the displacement and base register. The register can be overridden by specifying the optional register operand.

Evaluation of Operands

If only one operand is present, it must be a previously-defined symbol. In this case, both the base register and the offset of the new symbol are taken from those of the previously-defined symbol. This form is used to refer to a symbol by a different type code; for instance, to refer to a half tally as a character.

If two operands are present, the first indicates the base register. The second operand indicates the offset of the symbol's address, where the unit of offset depends on the symbol type.

Type	Offset Unit
bit (type B)	bits
character (type C) half tally (type H)	bytes
tally (type T) double tally (type D) triple tally (type F) storage register (type S) external register (type X)	words (16 bits each).

The second operand may be viewed as an expression which is evaluated on the basis of the following rules:

- If the first character is the '*' character, it is assumed to refer to the location counter.
- If the first character is a number or the character 'X' followed by a single quote, it is assumed to be a literal or literal expression (e.g., 79, or X'A' or 3+2).
- If the first character is any other legal character, it is assumed to be a symbol name.

`2*LOC`	evaluated as 2 times the offset in symbol LOC
`LOC*2`	evaluated as the label 'LOC*2'
`*`	evaluated as the current location counter (bytes)

Using Location Counter as Second Operand

Whenever the second operand begins with the "*" character, the expression is evaluated using the following formula:

```
*{n}{+/-m}
```

* the current location counter as maintained by the assembler.

n measurement of units, in bits, with which to express the location; valid values are 1, 4, 8, 16, or 32. If omitted, it's assumed to be 8 (in bytes). Thus, a location counter value of X'10' equates as follows:

```
*1      X'80' (bits)
*8      X'10' (bytes)
*16     X'8' (TLY or tally units)
*32     X'4' (DTLY or double tally units)
```

+/-m value by which to increment or decrement the resultant value of *{n}; 'm' must be preceded by a plus (+) or minus (-) sign. For example:

```
        ORG   X'10'
LABL1   DEFT  *16
LABL2   DEFT  *16+1
```

would define one TLY at byte offset X'10' (word X'8') and another at byte offset X'12' (word X'9').

The 'n' value must be preset for certain DEF directives:

```
DEFB *1
DEFH *                  *8 is assumed for *)
DEFT *16              } all require TLY offsets in
DEFD *16              } all instructions
DEFF *16              }
```

LOWBIT	DEFB R15,7	Defines a bit with register 15 as the base register, and an offset of 7 (low order bit in the byte addressed by the register)
XCURS	DEFT R15,7	Defines a tally with register 15 as the base register, and an offset of 7,which references bytes 14 and 15 (a tally) displaced from the virtual address in R15.
NXTFID	DEFD R15,7	Defines a double tally with register 15 as the base register, and an offset of 7, which references bytes 14-17 displaced from the virtual address in the AR; note this is not the same as a displacement of 7 double tallies, as used for immediate symbols; see the Immediate Symbols topic in Section 2 on the Assembler.
T2T1	DEFD R0,T2	Defines T2T1 as a four-byte field that overlays the fields T2 and T1 (both tallies) in the accumulator
FP0S	DEFS FP0	Redefines the six-byte accumulator FP0 as a storage register FP0S
R1XR.15	DEFX R15,R1DSP	Defines a 6-byte external register located X'10A' bytes off of R15.
SR20FID	DEFDL SR20	Defines a symbol that references the FID field of storage register SR20
SR20DSP	DEFTU SR20	Defines a symbol that references the displacement field of SR20

```
SVLOC    ORG  X'90'          * SET LOC CTR FOR DEFS
*
*************************
*
CTR.A    DEFT  R1,X'48'      * TLY DSP=HTLY DSP
CTR.B    DEFT  R1,72         * X'90' / 2
CTR.C    DEFT  R1,CTR0       * CTR0 DEF'D ASSUMING
CTR.E    DEFT  R1,*16        *  X'48' TALLY DSP--
                              NOT VALID IF CTR0
*************************     IS MOVED IN A
                              LATER RELEASE[1]
*************************
*
CTR.EUA  DEFH R1,2*CTR.E     * UPPER HTLY OF
CTR.EUB  DEFH R1,**1         * CTR.E
CTR.EUC  DEFH R1,*8          *
*
CTR.ELA  DEFH R1,1+CTR.EUA   * LOWER HTLY CTR.E
CTR.ELB  DEFH R1,*+1         * FORM: *{+m}
CTR.ELC  DEFH R1,*8+1        * FORM: *{n}(+m)
*****************************
*
SRX      DEFS FP0            * REG OPERAND INFERRED
                             * FROM FP0 SYMBOL
SRXN     DEFS R0,FP0         * REG OPERAND EXPLICITLY
         *                   DEFINED
```

[1]Ultimate recommends not hard-coding assumptions such as the CTR0 location; if CTR0 were redefined, the program may not work properly after assembly with the new PSYM definition

DEFM

The DEFM (Define Mode-id Symbol) directive defines a local symbol as a modal entry point, or mode-id.

Syntax

symbol DEFM n1,n2
symbol DEFM n1,m

n1 entry point number; must be in the range 0-15 (0-X'F')

n2 frame number

m previously defined mode-id

Description

A mode-id consists of a four-bit entry point number and a twelve-bit frame number (FID). The DEFM directive is used whenever a mode-id symbol is needed in a program.

A symbol defined by the DEFM directive can be used in the BSL and ENT instructions to transfer control to the specified location. It can also be used in the MOV and LOAD instructions, when it acts as a literal value. Depending on implementation, the assembler may actually generate a literal at the end of the object code with the value defined in the DEFM instruction.

`EXT.SYM`	`DEFM 3,133`	Defines EXT.SYM as entry point 3 in frame (decimal) 133 (for firmware machines, this is location 7 in frame).
`MYFRAME`	`DEFM 0,510`	Defines MYFRAME as entry point 0 in frame 510
`ENTRY0`	`DEFM 0,MYFRAME`	Defines ENTRY0 as entry point 0.
`ENTRY1`	`DEFM 1,MYFRAME`	Defines ENTRY1 as entry point 1.
`ENTRY15`	`DEFM 15,MYFRAME`	Defines ENTRY15 as entry point 15.

DEFN

The DEFN (define constant symbol) directive defines a local symbol as a constant.

Syntax

symbol DEFN n

n constant value which may be generated in one or two bytes (a byte or tally).

Description

The DEFN directive defines the value as a constant symbol (type n). A constant can be used in exactly the same manner as a literal value. With many instructions (on certain implementations), reference to a constant or literal causes a literal field to be generate at the end of the object code.

Constants have a maximum length of four bytes, giving a numeric range of -2,147,483,648 to 2,147,483,647 (X'80000000' to X'7FFFFFFF'). Constants more than two bytes long, however, must be explicitly defined as double tallies via the DTLY directive.

For more information about literal values, see the section on Literals in Chapter 2.

Notes: *If the value is to be used to define the location of a variable (an offset) within an ABS frame, the DEFN directive should* ***not*** *be used. The DEFNEP or DEFNEPA directives should be used instead, to define the word or byte offset relative to an entry point rather than an explicit offset.*

Due to an assembler requirement in software machines, a symbol of type n defined via DEFN, DEFNEP, or DEFNEPA (but not other symbol types) when used as an offset must be preceded by "0+" or the appropriate "n+". The symbol offset may not be used alone. For example,

```
TEN    DEFN 10
CTR    DEFT R7,0+TEN   (0+ required)
CTR1   DEFT R7,CTR     (0+ not required)
```

Using DEFN to refer to a constant value by a symbolic name rather than its actual value tends to make programs easier to read, easier to modify, and less prone to errors.

```
MAXNUM   DEFN 20
XCONST   DEFN X'8010'
DELIM    DEFN C'.'
CCONST   DEFN C'ABCD'
DCCONST  DTLY C'ABCDEF'
         .
         .
         .

         BH  T0,MAXNUM,ERR   References 2-byte literal
         MOV  XCONST,CTR30   References 2-byte literal
         MOV  DCCONST,D1     DTLY must be defined
         MCC  DELIM,R15      Immediate value
```

DEFNEP
DEFNEPA

The DEFNEP (define entry point) and DEFNEPA (define entry point aligned) directives define the offset of a program entry point that is machine independent. DEFNEP defines the byte offset to the designated entry point. DEFNEPA defines the word offset (word-aligned) to the designated entry point (that is, rounded up if necessary to the next even address if entry points begin on odd bytes, as they do on firmware machines).

Syntax

label DEFNEP ep#
label DEFNEPA ep#

label local label to assign to the entry point
ep# entry point number relative to the beginning of the framw

Description

The DEFNEP and DEFNEPA directives are used to define machine-independent address displacement (offset) values, which can then be referenced in ADDR or DEFx directives.

These directives are necessary to ensure that an assembly language program operates correctly on any Ultimate implementation, since data in ABS frames may wind up in different locations on different implementations. This is because the object code for a firmware machine, for instance, may be smaller than that generated by assembling the frame for a software machine. If a frame contains both code and data, then, the locations of entry points and data may vary between machines.

The EP.ADDR directive can be used to refer directly to the entry point of the frame. To refer to data which is not an entry point, however, it must be placed either (1) at a "safe" location via an ORG directive, or (2) immediately after the last entry point of the frame, which has been defined by a DEFNEP or DEFNEPA directive. An ADDR or DEFx directive may then be used after DEFNEP or DEFNEPA to reference the data.

The specified label is defined in the program's TSYM file as a symbol of type n. The value of the symbol is derived from the entry point number,

as the correct offset for that entry point based on the implementation for which the program is assembled.

For example, the following directive sets up a literal or constant (n) symbol at the next available location following entry point 2:

```
DATA1   DEFNEP 3
```

This defines DATA1 as symbol type n with a value of 7 (offset to byte 7) on a firmware machine. On a software machine, this same directive could result in DATA1 having a value of X'E' (offset to byte 14) if the FRAME directive ORGs to byte 2 and four bytes are reserved for each EP instruction.

The above DEFNEP-defined symbol may be used with an ADDR directive to define a location immediately following entry point 2. For example, given the following program with data:

```
        FRAME  511
*
*
*
0       EP     !LABEL0
1       EP     !LABEL1
2       EP     !LABEL2
        CMNT   *     The next available location is 007
        CMNT   *     on firmware machines
MYTXT   TEXT   C'Externally referenced string'
```

The following lines could be placed in another program to address the label MYTXT:

```
TXTDSP  DEFNEP 3              Define byte offset to EP 3
HERTXT  ADDR   TXTDSP,511
```

which references the same data as:

```
HERTXT  ADDR   7,511          (on a firmware machine)
```

or

```
HERTXT  ADDR   X'E',511       (on a 1400 machine)
```

When both programs are assembled for the same machine, they execute properly. No source code has to be changed to assemble the two frames for another machine.

If the data following the last entry point must be word aligned (a tally or greater), the above technique does not work since it produces an ADDR pointing to the first byte at or beyond the specified entry point. In this case, the DEFNEPA directive must be used instead to point to a word aligned variable following the last entry point. For example:

```
DATA2   DEFNEPA 3
```

defines DATA2 as a symbol of type n with a value of 4 on firmware machines (the first even address at or following entry point 3 is X'008'

which is word offset X'004'). On a software machine, however, DATA2 could have a value of 7.

The above DEFNEPA could be used with an ADDR directive to define a tally immediately following entry point 2. For example, given the following frame with a tally MYWORD defined at the next word offset:

```
        FRAME  511
*
*
*
0       EP     !LABEL0
1       EP     !LABEL1
2       EP     !LABEL2
        CMNT   *     The next available location is 007
        CMNT   *     on firmware machines
        ALIGN
MYWORD  DEFT   R1,*16
```

The following lines could be placed in another frame to address the label MYWORD:

```
WRDDSP  DEFNEPA 3             Define word offset to EP 3
HISWRD  ADDR    2*WRDDSP,511
```

which references the same data as:

```
HISWRD  ADDR    8,511         (on a firmware machine)
```

since WRDDSP has the value of the 4th word (2*4 = 8).

On a software machine where the first entry point starts at location 2 and each EP instruction generates 4 bytes of object code, DEFNEPA would give WRDDSP a value of X'E' (14), and the ADDR directive would be equivalent to:

```
HISWRD  ADDR  X'E',511
```

```
SYMB1     DEFNEPA 11          word offset X'C' to entry point 11
ADDR.NW   DEFS    R6,0+SYMB1  defines 3 words of storage at loc.
ADDRS.NW  DEFT    R6,3+SYMB1  defines a tally immediately after
                              ADDR.NW, or 3 words past entry
                              point

TBL       DEFNEPA 11          word offset X'C' to entry point 11
TLY1      DEFT    R6,0+TBL    tally at X'18' (2 * X'C')[1]
TLY2      DEFT    R6,1+TBL    tally at X'1A' (2+2 * X'C')[1]

TBL       DEFNEP  11          byte offset to entry point 11
HTLY1     DEFH    R6,TBL      half tally at X'17'[1]
HTLY2     DEFH    R6,2+TBL    half tally at X'19'[1]
```

[1] Note that the explicit locations in hex apply only to firmware machines and are included merely as examples. By using DEFNEPs, the appropriate offset is automatically assembled on all Ultimate implementations.

DIV
DIVX

The DIV and DIVX (divide) instructions are used to divide the contents of the accumulator by the value of the operand. The DIV form addresses the accumulator field D0); the DIVX form addresses it the field FP0.

Syntax

DIV d	DIVX d
	DIVX f
DIV h	DIVX h
DIV n	DIVX n
DIV t	DIVX t

d double tally

f triple tally (for DIVX only)

h half tally

n numeric literal); if used, a 2-byte field is assumed (a range of -32,768 through +32,767). If a 1-byte literal (half tally) is being referenced, it should be defined separately using the HTLY directive. If the literal is outside the range of -32,768 through +32,767, a 4-byte literal must be separately defined using the DTLY directive, or a 6-byte literal via the FTLY directive.

The n form may generate a 2-byte literal at the end of the program when assembled for certain machines.

t tally

If the value of the operand is zero, the results of the division are unpredictable.

Description

The DIV instruction divides the operand value into the 4-byte field in the accumulator called D0. If the operand is a half tally (1 byte) or tally (2 bytes), it is internally sign-extended to form a 4-byte field before the divide operation takes place. The integer result is stored in D0, and the integer remainder in D1. The division does not affect the original operand or the other sections of the accumulator.

The DIVX form divides the operand value into the 6-byte field called FP0. If the operand is a half tally (1 byte), tally (2 bytes), or double

tally (4 bytes), it is internally sign-extended to form a 6-byte field before the divide operation takes place. The 6-byte integer result is stored in FP0, and the 6-byte integer remainder is stored in the 6-byte field called FPY. The division does not affect the original operand, or the other sections of the accumulator.

In division involving negative numbers, the sign of the remainder follows that of the dividend, so that multiplying the quotient by the divisor and adding the remainder yields the original dividend.

These instructions cannot detect arithmetic overflow or underflow.

```
 3 / 2  =  1  remainder  1
-3 / 2  = -1  remainder -1
 3 /-2  = -1  remainder  1
-3 /-2  =  1  remainder -1
```

```
DIV   D4
DIV   H8
DIV   T4
DIVX  D4
DIVX  FP1
DIVX  H8
DIVX  T0
DIV   11
```

DTLY
FTLY
HTLY
TLY

The DTLY (double tally), FTLY (full (triple) tally), HTLY (half tally), and TLY (tally) directives reserve storage and set up the symbol in the label field to be of a specific symbol type. They can also be used to only reserve storage if there is no entry in the label field.

Syntax

{symbol} DTLY n
{symbol} FTLY n,n
{symbol} HTLY n
{symbol} TLY n

symbol — optional label name; if present, it stores the symbol name as an item in the TSYM file with the specified value.

n — value of the symbol as an alphabetic, numeric, or alphanumeric value. In FTLY instructions, the first n gives the value of the upper two bytes; the second n gives the value of the lower four bytes.

Description

The DTLY directive is used to define a double tally (four bytes), and to store a 4-byte value .

The FTLY directive is used to define a triple tally (six bytes), and to store a 6-byte value.

The HTLY directive is used to define a half tally (one byte), and to store a one-byte value.

The TLY directive is used to define a tally (two bytes, which also make up one "word"), and to store a 2-byte value.

The TLY, DTLY, and FTLY directives force the location counter to be aligned on an even-byte boundary (word alignment).

The HTLY directive can only be used when the program's location counter is less than X'100'; otherwise it will generate a TRUNCation error message. This is because the generated symbol would have an

offset of more than X'FF'. The TLY, DTLY, and FTLY directives can appear anywhere in the source program.

Due to an assembler requirement, the value stored by the FTLY directive must be specified as two values: an upper 2-byte value and a lower 4-byte value. The programmer must be especially careful with negative values. For example:

Instruction	Equivalent value	
	Hex	Decimal
`X.10 FTLY 0,1`	000000000001	1
`FTLY 0,12345`	000000003039	12345
`ABCD FTLY 0,10000000`	000000989680	100000000
`FTLY 2,X'540BE400'`	0002540BE400	10000000000
`FTLY X'FFFF',X'FFFFFFFC'`	FFFFFFFFFFFC	-3

For information on defining a storage register, see the SR directive.

The specified symbol label, if any, is added to the TSYM file. The value of the symbol is stored at the current program counter location, word-aligned if necessary, as described above.

EJECT

The EJECT (eject page) directive ejects the current page and begins a new page in an MLISTing of the program in which the EJECT directive appears. This directive is put into effect only if the MLIST command has the J option.

Syntax

EJECT

Description

The EJECT directive is used to start a new page of an MLIST for a program, where the MLIST command specifies the J option.

If the MLIST command does not contain the J option, the directive has no effect.

END

The END (end program) directive indicates the end of a source program.

Syntax

END

Description

The END directive has no effect on assembly, and is treated as a comment line (see CMNT directive).

ENT

The ENT (external branch) instruction transfers program control unconditionally to a specified location external to the current program frame.

Syntax

ENT m
ENT n,m

m mode-id (external entry point), which defines a frame number and offset for a routine located outside the current program frame

n entry point (0-F or symbol); offset specified by m is replaced by the value of n .

Description

The ENT instruction is used to unconditionally branch to an external routine when no subroutine return is needed to the current program frame. The ENT instruction resolves the effective address of the mode-id and transfers program control to that address.

The external mode-id must be defined as a globally defined symbol of type M in the PSYM file, or it can be defined with a DEFM or MTLY Assembler directive (either within the local program or in an INCLUDEd program).

If the n operand is specified, the mode-id is used only to define the frame. Conventionally, mode-ids used only to define a frame are given entry point values of zero; for example,

```
MYSUBS  DEFM 0,511
```

For information on internal branches, see the B instruction. For information on transferring to subroutines, see the BSL instruction.

```
EXTM    DEFM 10,500      Define mode-id (type m symbol)
        CMNT *           at entry point 10 in frame 500.
        .
        .
        ENT  EXTM        Transfers control to FID 500,
        CMNT *           entry point 10.
```

ENT*

The ENT* (external branch indirect) instruction branches unconditionally to the location referenced by the specified operand.

Syntax

ENT* t

t tally symbol, which contains the branch destination address.

Description

The ENT* instruction performs the same function as a LOAD t instruction followed by an ENTI instruction.

The contents of the accumulator are not guaranteed to be in a predictable state after execution of an ENT* instruction. On firmware machines, ENT* is a macro that loads the accumulator (T0) with the current content of the t operand, and then executes the ENTI instruction. T1 is also destroyed because of sign-extension in loading the accumulator. However, on software machines, the same operation may occur without affecting T0 or T1.

For more information about how external branches operate, see ENTI and ENT.

ENTI

The ENTI (external branch indirect) instruction branches unconditionally to the location specified in T0 of the PCB.

Syntax

ENTI

Description

T0 must contain the branch destination mode-id (the high 4 bits are the entry point and the low 12 bits are the FID), which may be loaded into it from a local label, an external label, or by converting an ASCII string.

The ENTI instruction operates identically to the ENT instruction, except that the address is variable and is obtained from the low-order two bytes of the accumulator, T0, instead of from an operand.

```
R15 points to a hexadecimal ASCII string :
v
|x |7 |1 |F |E |AM| ...

 BSL  CVXR15       CVXR15 is a subroutine that
 CMNT *            converts the ASCII string
 CMNT *            value to a binary value in the
 CMNT *            accumulator FP0 (that is, T0)
 ENTI *            External branch to T0 location
 CMNT *            (frame 510, entry point 7).
```

EP

The EP (entry point) instruction defines an entry point at the start of a program frame.

Syntax

EP l

l local label.

Description

The EP instruction is used to define an entry point at the start of a program frame. Up to 16 entry points can be defined. Although the assembler may not flag an error if more than 16 entry points are defined, there is no way to specify the 16th (number 15) in either the ENT or BSL instruction.

Although typically EP generates the same object code as B (branch), EP guarantees branch code of a fixed length on each Ultimate implementation and, moreover, is required by some Assemblers in order to identify program entry points.

The EP instruction immediately resolves the effective address of the local label and defines the entry point as a symbol of type L (label).

```
        FRAME 471
        *
        *
        ORG  0
0       EP   FIRSTEP
1       EP   SECONDEP
2       EP   THIRD EP
3       EP   FOURTHEP
4       EP   FIFTHEP
```

EP.ADDR

The EP.ADDR (entry point address) directive specifies an entry point address and creates a storage register containing that address.

Syntax

```
label EP.ADDR n,n
label EP.ADDR m
label EP.ADDR n,m
```

label — name of symbol to use in referring to storage register

n,n — virtual address to reserve for the symbol. The first operand specifies the entry point number (0-15) in the frame. The second operand specifies the frame number (FID)

m — mode-id symbol which contains the virtual address

n,m — entry point to be used with the FID from the mode-id m .

Description

The EP.ADDR creates a storage register in unlinked format. containing the specified entry point address. It also creates a symbol (type S) that refers to the storage register. Six bytes of storage for the address are reserved at the current location counter (word-aligned).

An EP.ADDR directive, instead of an ADDR directive, should be used to point directly to entry points. The reason is that entry points cannot be assumed to be of the same length on all machines. The EP.ADDR directive, which uses entry point numbers rather than actual displacements, generates the correct byte offset based on the machine for which the program is being assembled.

For example, on some systems, two bytes of object code are generated for each entry point. On other systems, four bytes of object code are generated for each entry point. If an ADDR directive is used to point to entry point 9 in frame 278 on one system, the instruction would be

```
ADDR    19,278
```

On other systems, the equivalent ADDR instruction would be:

```
ADDR    38,278
```

However, The EP.ADDR directive is the same on all system types.

```
EP.ADDR 9,278
```

```
MD999                                PSYM entry
001 M
002 1
003 00A

LAB1    EP.ADDR  1,10       specifies entry point 1 in frame 10
LAB2    EP.ADDR  MD999      these directives are equivalent
LAB3    EP.ADDR  1,MD99
```

EQU

The EQU (equate) directive sets up an equivalence between the symbol in the label field of the statement and the operand.

Syntax

label EQU n
label EQU symbol

label — symbol name being equated to an operand

n — constant or literal value

symbol — redefined symbol name;current program location counter (*) is often specified as the operand.

Description

The EQU directive is normally used for two purposes:

- To define a new name for an existing symbol (already defined in PSYM or via DEFx, DEFN, or DEFM
- To give a label to a location within the program.

If the operand is a literal or constant or the current location counter symbol (*), the label symbol is stored as a symbol of type L. If the operand is another symbol, the label symbol is created as an exact duplicate of the operand symbol.

Note: *It is recommended that DEFN directive be used to define constant values, and to give names to numeric or character values:*

```
OFFSET  DEFN 3
MAXNUM  DEFN X'40'
```

`LOOP`	`EQU`	`*`	creates a symbol LABEL, with the current location as its value. This is a useful way of defining labels, since the label is on a line by itself, and is therefore clearer.
`TITLE`	`EQU`	`*-1`	creates a symbol TEXTS, with the current location less one as its value. This is useful when an SRA instruction is to address a text string, and it is necessary to address the location one less than the start of the string.
`COUNT`	`EQU`	`CTR21`	creates a symbol COUNT which is equivalent to CTR21.
`PTR`	`EQU`	`SR20`	creates a symbol PTR which is equivalent to storage register 20.
`X`	`EQU`	`AM`	creates a symbol X which is equivalent to the value of the attribute mark, X'FE'. The symbol AM is predefined in PSYM as a constant (type N). X is also made a type N symbol, since its definition is copied directly from that of AM.

FAR

The FAR (force attachment of register) instruction attaches an address register (if not already attached), and thereby normalizes its virtual address.

Syntax

FAR r,n

r address register R0-R15 or synonym such as IS, IR, or TS

n constant or literal value of of 0 or 4

0 guarantees attachment and normalization

4 guarantees attachment and normalization, and sets R15 (unlinked) to the link field of the frame to which r points after normalization (r=R15 is permissible).

other values of n are undefined.

Description

The FAR instruction has the following uses:

- for compatibility between implementations
- to normalize the virtual address in a register
- to set R15 to the link field of the frame
- with XMODE for exception processing

On firmware machines, address registers are typically attached only when data is referenced through the register (via indirect or relative operands) or when either an INC/DEC Rn or a FAR is executed. The FAR is therefore necessary for correct program operation on firmware machines whenever the program may not execute properly if the register is not attached.

On software machines, however, all address registers for a process are typically automatically attached and remain attached as long as the process is running.

Consequently, the FAR instruction should be coded where needed for compatibility between firmware and software machine implementations. The "FAR r,0" instruction generates no object code when assembled for systems that keep all registers attached.

The FAR instruction is typically used to ensure that the virtual address in a register is normalized before using it in a comparison with another virtual address (without regard to the data actually addressed) or MOVing it to a storage register.

Virtual addresses in storage registers must be normalized before comparison, since the same location within a set of linked frames may be addressed in terms of several different frame-displacement combinations. If a virtual address is unnormalized, perhaps due to an "INC r,t" instruction, it may fail a "BE r,s" or "BE s,s" comparison with another (normalized) virtual address even though it logically addresses the same location.

For example, on a system having 512-byte frames, if a register such as R14 is incremented by X'200', the displacement is inaccessible until the address is normalized (if a linked frame; otherwise, it is a Crossing Frame Limit error).

Note: As an alternative, you can always move a storage register into an address register before comparing addresses:

See also Section 3, Addressing and Representing Data, and the topic on Understanding Registers.

Another use of the FAR instruction is to set Address Register 15 to the link field of a frame; that is, to byte 0, unlinked. R15 is set up in this manner if the "mask" byte (the second operand) has a value of X'04'. Other mask byte values are reserved for future use.

`INC R14,ID.DATA.SIZE`	INCs R14 by the size of one data frame, forcing a chase of the forward link
`FAR R14,X'04'`	ensures normalization of R14, and setup of R15 in the unlinked format to byte 0 of the frame referenced by R14

Finally, the FAR instruction can be used in conjunction with the XMODE field of the PCB to perform exception processing. For example, you may want to build a table in a set of linked frames and need to ensure that enough frames are available. To avoid an abort to the debugger, you can set up XMODE with the mode-id of a subroutine that links another frame onto the set, and execute another FAR instruction each time another entry is added to the table.

Note however, that (1) when an XMODE routine is entered because of a Forward Link Zero condition, the address register involved is not guaranteed to be pointing to the same location in all implementations, and (2) no XMODE routine is guaranteed to work when the register is incremented by more than one frame past the end of a linked set of frames.

```
MOV   ADDIT,XMODE
INC   R14,ENTRYSIZE
FAR   R14,0              forces attachment of R14
ZERO  XMODE              automatically calls ADDIT if a
                         forward link zero condition would
                         occur while normalizing the
                         virtual address in R14
```

The above example works in cases such as when R14 starts out pointing to (logical) byte 1 of the first frame, ENTRYSIZE is an integral divisor of ID.DATA.SIZE (such as 10, 20, 50, etc.), and ADDIT always sets R14 to (logical) byte 1 of the new frame it attaches before exiting. Also note that any reference to data off R14, not just a FAR instruction, may cause the XMODE routine to be entered; the FAR is then not needed unless it is the last instruction in the table building routine.

```
MOV   ISBEG,IS           Set IS to data start
INC   IS,CTR30           Increment by length
FAR   IS,0               Ensure normalized SR for future
MOV   IS,ISEND           tests.
```

```
MOV   SR20,R14          Get data pointer
FAR   R14,X'04'         Attach R14, set R15 to links
LOAD  R15;H1            Load nncf


MOV OSBEG,OS            example of comparison of virtual
                        addresses
INC OS,4000             skip forward at least one frame
MOV OS,OSEND            this may leave an unnormalized
                        address in OSEND if executed on a
                        machine that does not normalize
                        OS when the "INC OS,4000"
                        instruction is executed.
FAR OS,0                ensure normalization
MOV OS,OSEND            save normalized address
BE OS,OSEND,SUCCESS     OS and OSEND are equal; if the
                        FAR instruction had not been
                        included, the compare would have
                        found the values unequal

MOV OSEND,R15           An alternative way to normalize
                        OSEND.
BE OS,R15,SUCCESS       R15 normalized and the compare
                        is equal.
```

FILLCHR

The FILLCHR (fill character) instruction initializes a data segment with a particular character (typically to clear frames).

Syntax

FILLCHR r1, r2, fill.character.

r2 must be one byte past r1. The byte pointed to by r2 at the start of the instruction is initialized with the fill character, which is then propagated for the number of bytes specified in T0.

Description

This instruction expects that T0 (the accumulator low order two bytes) has been set up for the maximum string length to move.

FRAME

The FRAME (define frame) directive defines the frame number into which the object code from the program is to be loaded.

Syntax

FRAME n

n frame number (in decimal, unless explicitly specified as a hexadecimal number).

Description

The FRAME directive is normally the first statement in the program, but always must precede any statements that generate object code. For more information about how the Assembler assembles the program object code, see Section 2, The Assembler.

The FRAME directive also sets the assembler's location counter to the first byte for code to be assembled into via an ORG directive. On firmware machines, this zero'th entry point is byte X'001', but on software machines, it may be X'002' for word alignment.

If it is necessary to use byte zero of the object code, the FRAME directive must be followed by an appropriate ORG, then the value of byte zero, and then the entry points, as usual. Note that EP is word-aligned when assembled for systems that require it.

```
        FRAME   511
        ORG     0
        CHR     C'*'
STAR    EQU     R1
        EP      !ENTRY0
        EP      !ENTRY1
.
.
```

FTLY

The FTLY directive defines a triple tally (48 bits, or 6 bytes). See the DTLY directive for details.

HALT

The HALT (halt program) instruction interrupts execution of the program and unconditionally branches to the assembly debugger.

Syntax

HALT

Description

The HALT instruction is primarily used as a debugging tool. Due to software machine requirements, the HALT instruction may not be used in a branch table at the beginning of a program. HALT can be used anywhere else in a program frame, but should not be used to take the place of an EP instruction. The NEP instruction should be used to indicate an invalid entry point; this creates the same effect as HALT, but guarantees that the object code is the same length as that for EP on each system type.

The HALT instruction affects only the current process; it does not halt the entire multi-user system.

HALT interrupts execution of the current program and transfers control to the assembly debugger at entry point 11 (HALT). Program execution can be resumed only by specifying an address with the debugger "G" command. Alternatively, the program execution can be terminated with the "BYE", "END", or "OFF" commands.

See Section 6, The Assembly Language System Debugger, for more information about using the debugger.

```
HALT
```

HTLY

The HTLY directive defines a half tally (8 bits, or one byte character). See the DTLY directive for details.

ID.B

The ID.B (table branch) instruction defines the unconditional branches in an internal branch table.

Syntax

ID.B l

l the label (in the current frame) of the branch destination

Description

The ID.B instruction is used instead of a B instruction to define branch tables (branch on a number used as an index) within a frame. It guarantees that the object code for each branch instruction has the same length. Otherwise, some assemblers may produce shorter code for some branch instructions than for others. The length of an ID.B instruction's object code in bytes, for any given implementation, is the value of the symbol ID.B.SIZE in the PSYM file.

The ID.B instruction immediately resolves the effective address of the local label and transfers program control to that address.

```
        MUL ID.B.SIZE          ADJUST TO BRANCH TABLE
        BSL !GOTO              (* INDEXED BY T0)
* GOTO WILL RETURN TO ONE OF THE FOLLOWING BRANCHES
        ID.B BLNK              ' '
        ID.B BLNK              '6'
        ID.B BSPACE            '4'
        .
        .
        .
```

ID.RSA

The ID.RSA (return stack adjust) instruction ensures that the return stack contains a valid address for subroutine returns.

Syntax

ID.RSA r

r address register that points to the top stack entry.

Description

The ID.RSA instruction should be inserted in subroutines that modify a return address to ensure correct operation on both firmware and software machines. This allows for instruction alignment (on a word) for software machines, which require word alignment. Any subroutine that modifies the return address of the stack must ensure that the modified address points to an even byte for these machines.

On firmware machines, the ID.RSA instruction assembles as a null. However, on software machines, ID.RSA assembles as a macro which ensures that the return address on the stack is word-aligned. It is important to insert this instruction in the appropriate spots even though it appears as a null on firmware machines.

ID.RSA assumes that an address register has been set up pointing to the top stack entry.

For example, assume that R14 is pointing to the first byte of the top return stack entry (the first byte of the FID portion). The following instruction should be inserted before returning from the subroutine:

```
ID.RSA  R14
```

INC

The INC instruction increments a data value or a register operand. See the DEC/INC (Data) or DEC/INC (Register) instruction for details.

INCLUDE

The INCLUDE (include program) directive causes the specified program to be "included" in the program being assembled.

Syntax

INCLUDE program-name

program-name name of an assembled program in the current user account and file

Description

The main reason for the INCLUDE directive is to be able to place a set of shared definitions in one item, and then use the definitions in any other program. Typically, variables and mode-id's that are common to a set of programs are placed in a single program for inclusion during assembly. The advantage of this method is that the definitions are not duplicated in every program that uses them. Such duplicate definitions can lead to errors and are in general more difficult to maintain than if they were all in one program.

The format of the INCLUDEd program is identical to that of any other program, though typically it consists of only DEFx (definition) directives.

If the INCLUDEd program does generate code, it may be necessary to save and restore the location counter of the current program around the INCLUDE statement, as shown in the example below:

```
SAVELOC    EQU  *
           INCLUDE TABLE1
           INCLUDE TABLE2
           ORG     SAVELOC   Reset location counter
```

INP1B
INP1BX

The INP1B and INP1BX instructions replace the character addressed by the register operand with the next character (byte) from the asynchronous channel input buffer.

Syntax

INP1B r INP1BX r

r address register (R0-R15) whose virtual address is the destination of the byte being read.

Description

The INP1B and INP1BX instructions are used to input data from the asynchronous channel input buffer into buffers in memory. These instructions read the next character from the asynchronous channel input buffer and place it in the location addressed by the register. The byte previously in that location is overlaid. If the input buffer is empty, the process is suspended until a character is received from the asynchronous channel.

Characters transmitted by the channel are automatically queued in the terminal input buffer for the process, until some configuration-dependent maximum number of characters is received. If this condition occurs, no further data characters are accepted from the channel; if an attempt is made to enter more characters, a bell character (X'07') is output for each attempted input character until the condition is cleared.

The INP1B instruction also tests to determine if the character should be echoed to the terminal. The INP1B instruction does not echo control characters (X'00' through X'1F'); it echoes non-control characters unless the bit NOECHO is set.

On most machines, the INP1BX instruction never echoes characters on the asynchronous channel. However, some types of terminals on some systems perform local echoing. Because of this, it is important to use the OUT1BX instruction when attempting to echo characters read via INP1BX. Otherwise, characters may be echoed twice.

The INP1B instruction actually consists of several instructions that test whether a character should be echoed, and execute an OUT1B instruction

if so. The following example shows an INP1B instruction and its macro expansion on a firmware system:

```
         INP1B R13
         +INP1BX R13
         +BCL    R13,X'20',=L002
         +BC:    R13,X'20',=L002,3
         +BBS    NOECHO,=L002
         +OUT1B  R2
=L002    +EQUX   *
```

Caution: *The INP1B and INP1BX instructions are not compatible with the TERM-VIEW and character translation features of the Ultimate operating system. To ensure that your programs are compatible with these features, use the system subroutines READ@IB or READX@IB. These subroutines are described in Chapter 5, System Subroutines.*

LAD

The LAD (load absolute difference) instruction loads the difference between a specified address register and a specified storage register into the accumulator (T0).

Syntax

LAD r,s
LAD s,r

r address register (symbol type r for R0-R15)

s storage register (symbol type s).

Description

The LAD instruction computes the difference between the virtual addresses of the two register operands, and stores the absolute (unsigned) value in the low-order two bytes of the accumulator, T0. The result is unsigned, and may be in the range 0-65,535. The other sections of the accumulator are unchanged .

The LAD instruction can be used to compare virtual addresses of data only when the addresses can be guaranteed to be one of the following:

- in the same frame, or
- in contiguously linked frames no more than 65,535 bytes apart.

The following actions are taken:

1. If the virtual addresses are in the same frame when normalized, they can be compared directly.

2. If the frame numbers of the virtual addresses of the registers are unequal, the instruction compares correctly if the addresses are:
   - in a set of contiguously linked frames, and
   - the addresses differ by no more than 65,535.

3. If the virtual addresses are in different unlinked or non-contiguously linked frames, or more than 65,535 bytes apart in a contiguously linked set, the results of the instruction are undefined.

It is therefore strongly recommended that the LAD instruction be used with registers in the same unlinked frame . In order to determine

address differences (or string lengths) under other conditions, use either the SIDC or MIIDC type of instruction.

```
LAD   BMS,BMSBEG

LAD   BMSBEG,BMS
```

LOAD
LOADX

The LOAD (load accumulator) instruction loads a relatively addressed operand value into the accumulator. The LOAD form loads into a 4-byte field (D0); the LOADX form loads into a 6-byte field (FP0).

Syntax

LOAD d	LOADX d
	LOADX f
LOAD h	LOADX h
LOAD m	LOADX m
LOAD n	LOADX n
LOAD t	LOADX t

d double tally

f triple tally (for LOADX only)

h half tally

m mode-id

n numeric literal); if used, a 2-byte field is assumed (a range of -32,768 through +32,767). If a 1-byte literal (half tally) is being referenced, it should be defined separately using the HTLY directive. If the literal is outside the range of -32,768 through +32,767, a 4-byte literal must be separately defined using the DTLY directive, or a 6-byte literal via the FTLY directive.

The m and n form may generate a 2-byte literal at the end of the program when assembled for certain machines.

t tally

Description

The LOAD instruction loads the operand value into the 4-byte field in the accumulator called D0. If the operand is a half tally (1 byte) or tally (2 bytes), it is internally sign-extended to form a 4-byte field before the load takes place.

The LOADX form loads the operand value into the 6-byte field in the accumulator called FP0. If the operand is a half tally (1 byte), tally (2 bytes), or double tally (4 bytes), it is internally sign-extended to form a 6-byte field before the load takes place.

The load operation does not affect the other sections of the accumulator.

```
        LOAD   D4

        LOAD   H8

        LOAD   RETIX

        LOAD   T4

        LOADX  D4

        LOADX  H0               (sign-extend H0 into FP0)

        LOADX  T0               (sign-extend T0 into FP0)

        LOADX  FP1

        ORG    X'100'           what is this??
=QRETIX +:Q    RETIX            (macro expansion showing
                                literal generation of RETIX
                                mode-id, X'1007')
```

MBD

The MBD (move binary number to decimal) instruction converts a binary value into its equivalent decimal ASCII string value, and stores the resulting string, starting at the address +1 of the register operand.

Syntax

```
MBD d,r          MBD n,d,r
MBD f,r          MBD n,f,r
MBD h,r          MBD n,h,r
MBD t,r          MBD n,t,r
```

d double tally

f triple tally

h half tally

t tally

r address register (R0-R13) whose virtual address +1 is the starting location at which the converted value is to be stored; neither R14 nor R15 should be used.

n integer that specifies the minimum number of characters that the output string will contain

Note: No section of the accumulator should be used as the binary field operand (the d, t, h, or f symbol operand).

Description

The register operand is pre-incremented by one before storing the first byte (character) of the converted string. After the first character is stored, the register operand is incremented by one, and the next converted character is stored at that location. This operation is repeated until the entire string has been stored.

The length of the string is determined by the format used in the instruction as follows:

- The first set of MBD formats does not create leading zeros; the field is variable length. MBD, unlike MBX (which is described in the next topic) generates one zero for an operand value of zero.
- The second set of MBD formats stores a fixed length field, padded with leading zeros if necessary. The field is allowed to exceed the specified length if its precision requires this.

The incrementing process could generate an address that crosses a frame boundary. If the register is in linked mode, it is normalized and attaches to the next frame in the linked chain. If the end of the linked set is reached during the normalization process, the following action is taken:

- If the exception mode identifier XMODE is non-zero, a subroutine call is executed to that address, to allow special handling of this condition.
- If XMODE is zero, the assembly debugger is entered with a trap condition indicating Forward Link Zero.

If the register is in unlinked mode and the frame boundary is reached, the assembly debugger is entered with a trap condition indicating Crossing Frame Limit.

This instruction destroys the current contents of:

BKBIT

T4

D0

D1

R14

R15

FPX ('f' operand forms only; same as SYSR0)

FPY ('f' operand forms only; same as SYSR1)

SYSR0 ('f' operand forms only; same as FPX)

SYSR1 ('f' operand forms only; same as FPY)

MBD is a subroutine call, not a primitive opcode; however, it has been included as part of the instruction set for convenience.

For the first set of formats, the subroutine MBDSUB is called to convert numbers of type h, t, and d (half tallies, tallies, and double tallies). The subroutine MBDSUBX is called for numbers of type f (triple tallies).

For the second set of formats, the subroutine MBDNSUB is called to convert numbers of type h, t, and d (half tallies, tallies, and double tallies). The subroutine MBDNSUBX is called for numbers of type f (triple tallies).

The subroutine call can be coded directly, instead of being called with an MBD instruction. See the macro expansions below, as well as the chapter on System Software, which illustrate the subroutine interface.

Assume the following binary tally value is to be converted

```
0000 1000 1101 0010
```

This is equivalent to the following hexadecimal value:

```
X'04D2'
```

MBD converts that value to the following decimal ASCII string:

```
1234
```

The equivalent, character-for-character, hexadecimal ASCII string value would be:

```
X'31 32 33 34'
```

Assuming VALUE is X'04D2', the following instructions would yield different stored results:

Instruction	Stored ASCII string
`MBD VALUE,R9`	`1234`
`MBD 8,VALUE,R9`	`00001234`

The following examples show how a program can be coded to call the subroutines directly:

Using MBD with minimum number of characters not specified:

```
MBD   CTR1,R9
```

Calling subroutine directly:

```
LOAD  CTR1
MOV   R9,R15
BSL   MBDSUB
MOV   R15,R9
```

Using MBD with minimum number of characters specified:

```
MBD   4,CTR1,R9
```

Calling subroutine directly:

```
LOAD  CTR1
MOV   R9,R15
MOV   4,T4
BSL   MBDNSUB
MOV   R15,R9
```

MBX
MBXN

The MBX (move binary number to hexadecimal) and MBXN instructions each convert a binary value into its equivalent hexadecimal ASCII string value, and stores the resulting string, starting at the address +1 of the register operand.

Syntax

MBX d,r	MBXN n,d,r
MBX f,r	MBXN n,f,r
MBX h,r	MBXN n,h,r
MBX t,r	MBXN n,t,r

d double tally

f triple tally

h half tally

t tally

r address register (R0-R15) whose virtual address +1 is the starting location at which the converted value is to be stored.

n integer that specifies the number of characters to convert

Description

The MBX instruction uses the first operand to locate the rightmost digit to be converted. It then uses H0, the low-order byte of the accumulator, to determine the number of characters to convert and whether zero padding is to be used for output. H0 must be set up prior to executing the MBX instruction.

The MBXN instruction is similar to MBX, except it first sets up H0 with the number of characters to convert specified in the instruction and with zero padding. MBXN then executes the MBX instruction.

The MBX instruction assumes H0 has been set up as follows:

Bit	Contents
0	1 (set) pad string with leading zeros 0 (unset) suppress leading zeros
4-7	The number of ASCII hexadecimal digits to create. If 0, the instruction becomes a NOP. If greater than the number of nibbles in the first operand, the results are undefined. If less than the number of nibbles in the operand, the most significant nibbles are skipped so that the conversion will finish on the rightmost nibble.

Both MBX and MBXN destroy the current contents of H0. All of D0 may be affected, depending on machine type.

The MBX instruction converts a binary number to its equivalent ASCII string value in hexadecimal. The length of the result string is determined by the H0 as follows:

- If leading zeros are suppressed, the field is variable length. In this case, MBX, unlike MBD, does not generate one zero for an operand value of zero.
- If padding is specified, the field is a fixed length string, padded with leading zeros if necessary. The field is truncated on the left, if the specified length is exceeded.

The register operand is pre-incremented by one before storing the first byte (character) of the converted string. After the first character is stored, the register operand is incremented by one, and the next converted character is stored at that location. This operation is repeated until the entire string has been stored. When the instruction terminates, the register points to the last byte moved. If no bytes are generated, the register is unchanged.

The incrementing process could generate an address that crosses a frame boundary. If the register is in the unlinked mode and the frame boundary is reached, the assembly debugger is entered with a trap condition indicating Crossing Frame Limit. If the register is in linked mode, it is normalized and attached to the next frame in the linked chain. If the end of the linked set is reached during the normalization process, the following action is taken:

- If the exception mode identifier XMODE is non-zero, a subroutine call is executed to that address, to allow special handling of this condition.
- If XMODE is zero, the assembly debugger is entered with a trap condition indicating Forward Link Zero.

Assume the following binary tally value is to be converted

```
0000 1000 1101 0010
```

This is equivalent to the following hexadecimal value:

```
X'04D2'
```

Converting this value to its equivalent value as a hexadecimal ASCII string would result in a character-for character conversion that yields a string value as follows:

```
04D2  (or X'30 34 44 32')
```

Assuming VALUE is X'04D2', the following instructions would yield different stored results:

Instruction	Stored ASCII string
`LOAD X'04'`	
`MBX VALUE,R9`	`4D2`
`MBXN 2,VALUE,R9`	`D2 (truncated)`
`MBXN 4,VALUE,R9`	`04D2`

Additional examples:

```
MUL  3              multiply a value
STORE T4            store value in T4 to free accumulator
LOAD  X'84'         for MBX: zero fill, convert 4 nibbles
MBX  T4,OB          convert value in T4 to ASCII hex

LOAD X'04'          zero suppress, convert 4 nibbles
MBX  FP2,R14

MBXN 4,CTR1,R9      zero-fill, convert 4 nibbles
```

MCC

The MCC (move character to character) instruction stores the character addressed by the first operand at the location addressed by the second operand.

Syntax

MCC c,c
MCC c,r
MCC n,r
MCC r,c
MCC r,r

c relatively addressed characters

n constant or literal values

r address registers

Description

The MCC instruction copies a data character to a specified location.

The character addressed by the first operand is stored at the location addressed by the second operand. The contents of the register operands are unaffected.

Note: Half tallies (symbol type h) are not directly supported; however, they can be equated to characters, then moved accordingly.

```
MCC   X'FE',R11

MCC   AM,R11          same as previous example, except
                      uses the PSYM name for X'FE'
MCC   R14,R15

MCC   PRMPC,R15

MCC   R15,PRMPC

MCC   R15;C0,CH8
```

MCI

The MCI (move character to character, incrementing) instruction stores the character addressed by the first operand at the address +1 of the second operand. The extended MCI instruction, which uses a third operand, repeats the move a specified number of times.

Syntax

```
MCI c,r
MCI n,r
MCI n,r,n
MCI n,r,t
MCI r,r
```

c relatively addressed character

n constant or literal value

r register

t tally

Description

The first operand references the character to be moved. The second operand is an address register (R0-R15) whose virtual address +1 is the location at which the character is to be stored.

The third operand is used with the extended form, and if present, specifies the number of times the move is to be repeated.

In the extended form, the same character is moved and the second operand is incremented until the terminating condition is met as specified in the third operand. If the third operand is initially zero (0), a total of 65,536 bytes (all the same character) are moved and stored.

With both forms, address register 15 (R15) and the accumulator D0 may be used.

Note: *Half tallies (symbol type h) are not directly supported; however, they can be equated to characters, then moved accordingly.*

For the address register operand, the incrementing process could generate an address that crosses a frame boundary. If the register is in

unlinked mode, and the frame boundary is reached, the assembly debugger is entered with a trap condition indicating Crossing Frame Limit. If the register is in linked mode, it is normalized and attaches to the next frame in the linked chain. If the end of the linked set is reached during the normalization process, the following action is taken:

- If the exception mode identifier XMODE is non-zero, a subroutine call is executed to that address, to allow special handling of this condition.
- If XMODE is zero, the assembly debugger is entered with a trap condition indicating Forward Link Zero.

The extended form of the MCI instruction sets up the conditions for the MIIT instruction, which moves a string of bytes. See the MIIT instruction for details.

```
MCI  AM,R10,9          move nine attribute marks
+MOV   R10,R15         code generated on firmware
+MCINR AM,R10          by extended form of instruction
+LOAD =T-1+9
+MIIT:RR R15,R10
```

MDB
MXB

The MDB (move decimal number to binary) converts a decimal ASCII character to its equivalent binary value and accumulates it into a symbol operand. The MXB (move hexadecimal number to binary) converts a hexadecimal ASCII character to its equivalent binary value and accumulates it.

Syntax

MDB	MXB
MDB r,d	MXB r,d
MDB r,f	MXB r,f
	MXB r,h
MDB r,t	MXB r,t

d double tally

f triple tally

h half tally (MXB only)

t tally

r address register (R0-R15) that contains the virtual address of the character to be converted

Description

The first operand is an address register (R0-R15) which references the ASCII character to be converted. The second operand specifies the location into which the converted byte is to be accumulated and should be initialized before this instruction is executed.

The character addressed by the first operand is assumed to be ASCII decimal number (for MDB) or ASCII hexadecimal number (for MXB). If not, the result of the instruction is unpredictable.

The MDB and MXB instructions are normally used in a loop, with the value of the second operand initially set to zero. The ASCII characters are accumulated according to the following formulas:

MDB: operand2 = operand2 * 10 + binary equivalent of operand1

MXB: operand2 = operand2 * 16 + binary equivalent of operand1

That is, each execution of the MDB or MXB instruction multiplies the previous value in the second operand by 10 (MDB) or 16 (MXB), then

adds in the binary equivalent of the character addressed by the first operand.

Note: These instructions have been largely superseded by the equivalent string conversion instructions MSDB, MSXB, MFD, MFE, and MFX.

```
        ZERO FP0          Clear the accumulator
LOOP    INC  R15          Set on next character
        BCNN R15,QUIT     Done if not numeric character
        MDB  R15,FP0      Convert one more character
        B    LOOP
```

MFD
MFE
MFX

These instructions convert ASCII character strings to binary. MFD (move decimal to binary F-tally) and MFE (move European decimal to binary F-tally) assume a string of decimal numbers; MFX (move hexadecimal to binary F-tally) assumes a string of hexadecimal numbers. The European version, MFE, interprets a comma as a decimal point.

Syntax

MFD r MFE r MFX r

r address register (R0-R15) whose byte address +1 is the starting location of the ASCII decimal or hexadecimal character string to convert.

Description

The results of these instructions are accumulated into the accumulator FP0 as scaled integers; that is, the string is multiplied by 10 raised to the power of the scaling factor.

Before executing an MFD, MFE, or MFX instruction, the accumulator must be initialized as follows:

Address	Value
H7	scaling factor; contains the number of fractional digits expected in the string. This must be in the range 0-15 (0-X'F')[1]. The converted value stored in FP0 is scaled up if there are not enough decimal places in the string.
H6	contains the maximum number of digits allowed to the left of the decimal point; typically used with fixed length strings. A zero is equivalent to 256.
FP0	any value in FP0 is multiplied by 10 (MFD and MFE) or by 16 (MFX) before each byte is converted; the converted byte is added to the previous value; initial value is typically zero.

[1]The high order four bits have special significance and are used by certain system processes. See Table 4-2.

Table 4-2. Bits in H7 used by MFD, MFE, and MFX

Bit in H7	If Set Before Instruction	If Set After Instruction
B63	no sign digit should occur; if one is found, conversion stops and NUMBIT is reset to zero	at least one character was processed
B62	previous processing encountered a numeric digit	at least one numeric digit has been found; if not, NUMBIT is reset to zero
B61	decimal point was previously encountered and the scaling factor is greater than zero; digits currently being processed are considered fractions	indicates a scaling factor greater than zero and a decimal point has been encountered
B60	indicates a minus sign was found previously. If string being converted currently also contains a minus sign, results are undefined.	indicates a minus sign has been found; this bit is copied to NEGBIT in the ACF

The bits shown in Table 4-2 are used primarily when it is necessary to separate the processing of an input string into multiple segments. The bits are set, but never reset, by these instructions.

The conversion terminates when one of the following conditions occurs:

- When a non-numeric character (for MFD/MFE, a character not in the range 0-9; for MFX, a character not in the range 0-F), is found. A plus or minus character in the first position, or a decimal point in any position, unless H7 = 0, are not terminating characters.

  If the terminating character is a decimal point or is a system delimiter (a character in the range x'F0'-x'FF'), the flag NUMBIT is set to 1;

otherwise, NUMBIT is zeroed. The register addresses the terminating character. The result is scaled as specified by H7 even if the conversion is stopped by a non-standard delimiter.

- When the number of characters specified by H6 have been converted. NUMBIT is zeroed, and the register addresses the last character converted. There is no scaling of fractional digits in this case.
- When the number of fractional digits specified by H7 have been converted, and the next character is not a system delimiter or decimal point. NUMBIT is zeroed, the result is scaled, and the register addresses the terminating (unconverted) character.

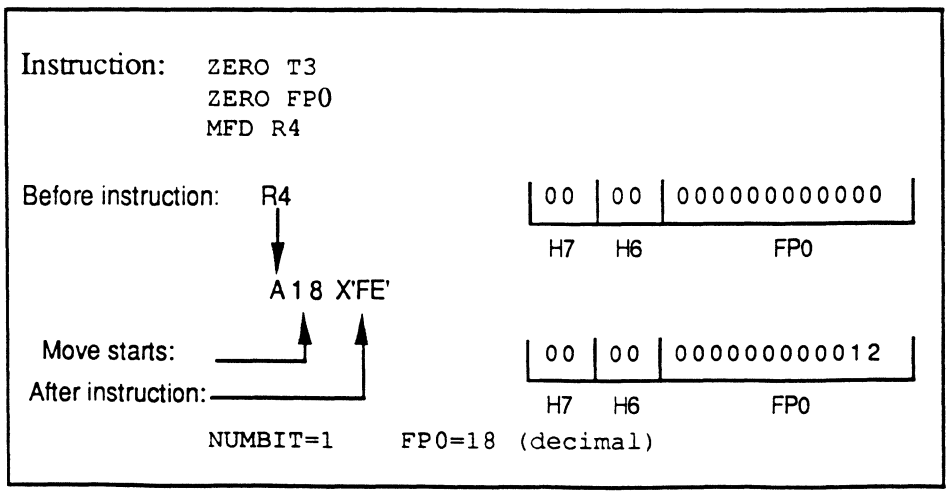
FP0 is always scaled as specified by H7 (even if no digits have been converted) except when the number of characters specified by H6 have already been converted (case 2, above).

After execution, H6 is decremented by one for each digit found to the left of the decimal point. When converting fixed length strings, then, H6 can be compared to zero to determine if an entire string was successfully converted.

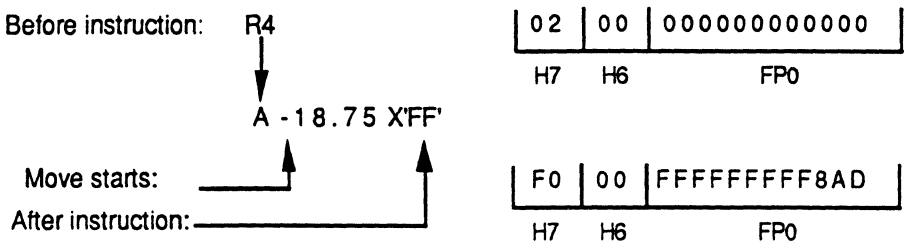
If the string is null, or if no numeric characters are found before the terminating character is encountered, NUMBIT is zeroed.

Note: *If more than one decimal point is encountered, the results are undefined.*

The following are examples of MFD and MFX usage.

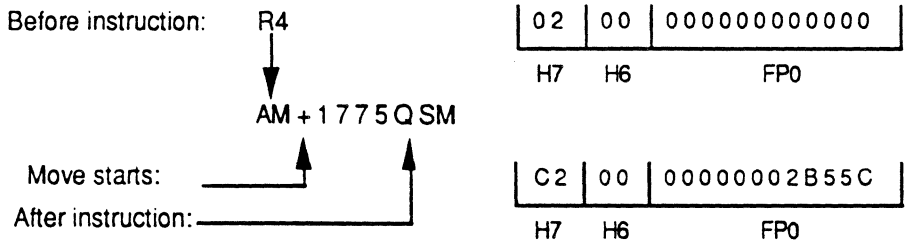


Instruction: MOV X'0200',T3
ZERO FP0
MFD R4



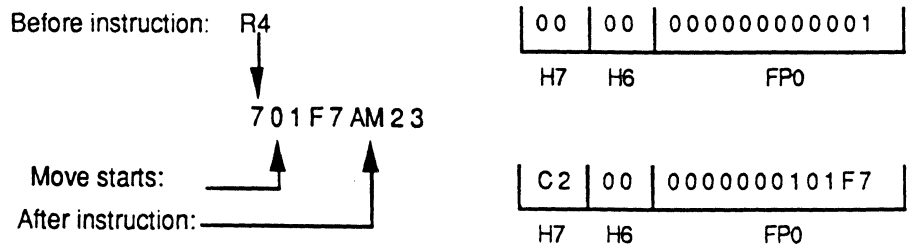
NUMBIT=1 FP0=1875 (decimal)
Note integer is scaled

Instruction : MOV X'0200',T3
ZERO FP0
MFD R4



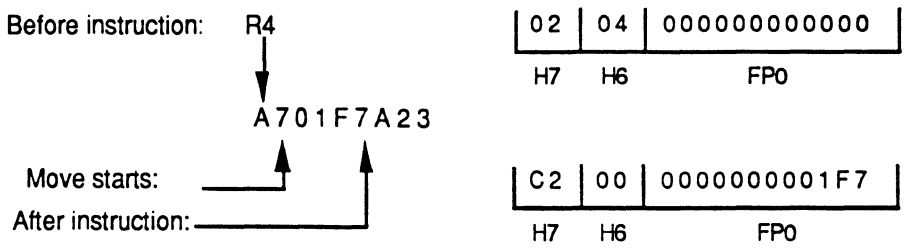
NUMBIT=0 FP0=177500 (decimal)
non-numeric character found
Note integer is scaled even though there were no fractional digits present

Instruction : MOV X'0000',T3
ONE FP0
MFX R4



NUMBIT=1 FP0=66039 (X'0000000101F7')
Note original value in FP0 is included

```
Instruction :  MOV  X'0204',T3
               ZERO FP0
               MFX  R4
```




```
NUMBIT=0     FP0=507 (X'0000000001F7')
```

Maximum string length reached, therefore, result not scaled

MIC

The MIC (move incrementing character) instruction copies one character from one location to another location.

Syntax

MIC r,c
MIC r,r

r address register (R0-R15)

c relatively addressed character

Description

The first operand is incremented by one; the character addressed by the incremented first operand is copied to the location addressed by the second.

The MIC instruction is the same as an INCrement followed by an MCC instruction.

Note: Half tallies (symbol type h) are not directly supported; however, they can be equated to characters, then moved accordingly.

For the address register operand, the incrementing process could generate an address that crosses a frame boundary. If the register is in unlinked mode, and the frame boundary is reached, the assembly debugger is entered with a trap condition indicating Crossing Frame Limit. If the register is in linked mode, it is normalized and attaches to the next frame in the linked chain. If the end of the linked set is reached during the normalization process, the following action is taken:

- If the exception mode identifier XMODE is non-zero, a subroutine call is executed to that address, to allow special handling of this condition.
- If XMODE is zero, the assembly debugger is entered with a trap condition indicating Forward Link Zero.

MII

The MII (move incrementing character to incrementing character) instruction increments two register operands, then moves the character addressed by the first operand to the location addressed by the second operand. The extended MII instruction, which uses a third operand, repeats the move a specified number of times.

Syntax

MII r,r
MII r,r,n
MII r,r,s
MII r,r,t

n constant or literal value

r address register (R0-R15)

s storage register

t tally

Description

The first operand references the character to be moved. The second operand is the location at which the character is to be stored. When the instruction is executed, both register operands are incremented by one. The character then addressed by the first operand is stored at the location addressed by the second operand.

The third operand is used with the extended form, which moves a string. If the third operand is a symbol type t or n, it specifies the number of times the move is to be repeated. If the third operand is a symbol type s, it specifies the location of the last byte in the string to be moved.

In the extended form, the assembled code sets up the conditions for the MIIR or MIIT instruction. With the MII r,r,s form, address register 15 (R15) is used. The third operand is moved into R15 and an MIIR instruction is executed (see the MIIR instruction for details). With the MII r,r,t and MII r,r,n forms, the accumulator D0 is used. The third operand is moved into D0 and an MIIT instruction is executed (see the MIIT instruction for details).

For the address register operand, the incrementing process could generate an address that crosses a frame boundary. If the register is in

unlinked mode, and the frame boundary is reached, the assembly debugger is entered with a trap condition indicating Crossing Frame Limit. If the register is in linked mode, it is normalized and attaches to the next frame in the linked chain. If the end of the linked set is reached during the normalization process, the following action is taken:

- If the exception mode identifier XMODE is non-zero, a subroutine call is executed to that address, to allow special handling of this condition.
- If XMODE is zero, the assembly debugger is entered with a trap condition indicating Forward Link Zero.

```
MII    R4,R6,T1
+LOADT T1              T1 is operand 3
+MIIT:RR R4,R6         R4,R6 are operands 1 and 2

MII    R4,R6,SR20
+MOVSR SR20,R15        SR20 is operand 3
+MIIR:RR R4,R6         R4,R6 are operands 1 and 2
```

MIID
MIIDC

The MIID (move incrementing string to incrementing string, delimiter) instruction increments two address register operands, then moves the character addressed by the first operand to the location addressed by the second operand. If the specified delimiter is not encountered, the operation is repeated. The MIIDC instruction performs the same operation, and also counts the number of characters moved.

Syntax

MIID r,r,n MIIDC r,r,n

r address register (R0-R15)

n constant or literal value that specifies the mask of delimiters to use as terminators for the string being moved

Description

The first address register's virtual address +1 references the starting character of the string to be moved. The second address register's virtual address +1 is the location at which the starting character of the string is to be stored.

The registers referenced by the first two operand fields are incremented by one; the character addressed by the first register is stored at the location addressed by the second. This operation is repeated until the condition specified by the third operand is met.

The third operand, called a "mask byte" indicates the terminating condition for the string move. The mask byte contains flags for four system delimiters plus three user-specified characters; it also has a match/nomatch flag that allows the move to terminate on either a match or nomatch with the specified delimiters.

Each byte is tested after it has been copied, to see if it satisfies the terminating condition.

Note: Because the delimiter test is done after the byte copy, the virtual addresses of the registers are always incremented by at least one.

The MIIDC instruction uses the accumulator field T0 to store the number of characters moved. As each byte is moved, T0 is decremented by one. If T0 was set to ZERO (0), its value after the instruction terminates is the negative of the length of the string, including the delimiter. If T0 was set to ONE (1), its value after the instruction terminates is the negative of the string length excluding the delimiter.

No other sections of the accumulator are affected.

For the address register operand, the incrementing process could generate an address that crosses a frame boundary. If the register is in unlinked mode, and the frame boundary is reached, the assembly debugger is entered with a trap condition indicating Crossing Frame Limit. If the register is in linked mode, it is normalized and attaches to the next frame in the linked chain. If the end of the linked set is reached during the normalization process, the following action is taken:

- If the exception mode identifier XMODE is non-zero, a subroutine call is executed to that address, to allow special handling of this condition.
- If XMODE is zero, the assembly debugger is entered with a trap condition indicating Forward Link Zero.

Mask Bytes

The mask byte can specify up to seven different characters to be tested; four of them are the standard system delimiters:

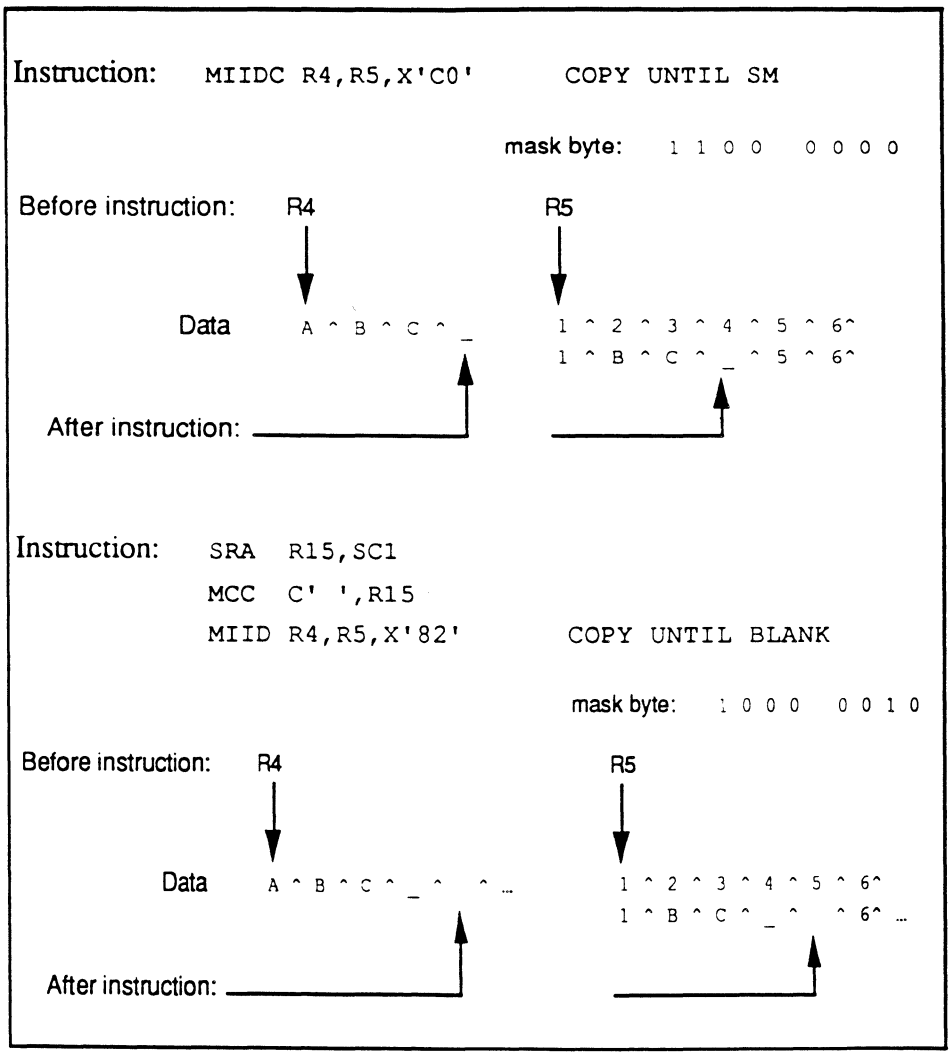
segment mark	SM	X'FF'
attribute mark	AM	X'FE'
value mark	VM	X'FD'
sub-value mark	SVM	X'FC'

The other three characters are taken from the scan character symbols SC0, SC1, and SC2. The contents of these symbols are specified by the programmer.

The low order seven bits in the mask byte are used to determine which of the seven characters are to be compared; if any bit is set (1), the corresponding character is tested; if zero (0), it is ignored.

If the high-order bit (bit 0) of the byte is set (1), it indicates that the string terminates on the first *occurrence* of a delimiter as specified by the setting of bits 1-7. If it is zero (0), it indicates that the string terminates on the first *non-occurrence* of a delimiter as specified by the setting of bits 1-7.

For more information on the use of the mask byte, see the description of SC0, SC1, and SC2 in Chapter 3.



MIIR

The MIIR (move incrementing string to incrementing string, register) instruction increments two address register operands, then copies the characters addressed from one location to another.

Syntax

MIIR r,r

r address register (R3-R14)

The first address register's virtual address +1 references the starting character of the string to be copied. The second address register's virtual address +1 is the starting location where the string is to be copied.

Description

The address of the last byte of the string to be copied is taken from address register 15 (R15).

The MIIR instruction first increments the registers referenced by the operand fields by one; the character then addressed by the first operand is stored at the location addressed by the second. A comparison is made and this operation is repeated until the first operand's address equals that of R15.

Caution: R15 should not be used as one of the two operands since it is referenced as the ending location of the string. The assembler does not check for this condition, and if R15 is used as an operand, the assembled instruction will not execute properly at runtime.

If the first operand's address equals that of R15 at the start of this instruction, no action takes place .

For all three registers (operands 1 and 2, plus R15), the incrementing process could generate an address that crosses a frame boundary. If the register is in unlinked mode, and the frame boundary is reached, the assembly debugger is entered with a trap condition indicating Crossing Frame Limit. If the register is in linked mode, it is normalized and attaches to the next frame in the linked chain. If the end of the linked set is reached during the normalization process, the following action is taken:

- If the exception mode identifier XMODE is non-zero, a subroutine call is executed to that address, to allow special handling of this condition.
- If XMODE is zero, the assembly debugger is entered with a trap condition indicating Forward Link Zero.

```
SETR R15,-1+ID.DATA.SIZE
MIIR R14,R13
```

MIIT
MIITD

The MIIT (move incrementing string to incrementing string, T0 termination counter) instruction copies a specified number of characters from one location to another. The MIITD (move incrementing string to incrementing string, T0 termination counter or delimiter) instruction performs the same operation as MIIT, but terminates the move at a specified number of characters or when a specified delimiter is encountered.

Syntax

MIIT r,r MIITD r,r,n

r address register (R0-R15)

n constant or literal value that specifies the mask of delimiters to use as terminators for the string being moved

Description

The first address register's virtual address +1 references the starting character of the string to be moved. The second address register's virtual address +1 is the location at which the starting character of the string is to be stored. These instructions also expect that T0 (the accumulator low order two bytes) has been set up for the maximum string length to move.

The mask byte contains flags for the four system delimiters plus three user-specified characters; it also has a match/nomatch flag that allows the move to terminate on either a match or nomatch with the specified delimiters.

These instructions are useful when a program expects that a frame boundary may be crossed, to access the XMODE facility.

If T0 is zero at the start of this instruction, no action takes place .

The registers referenced by the first two operand fields are incremented by one; the character then addressed by the first operand is copied to the location addressed by the second. T0 is decremented by one. A comparison is made after the copy to determine if one of the terminating conditions has occurred. If not, this operation is repeated.

The following are the terminating conditions:

- For the MIIT instruction, when T0 reaches zero. This instruction is typically used to move a fixed length string.
- For the MIITD instruction, when T0 reaches zero, or when one of the delimiter tests specified by the third operand (the mask byte), is encountered. This instruction is typically used to move a delimited string of unknown length to a location of preset maximum length. If the string is longer than the destination location, the instruction terminates without overlaying subsequent data.

  Note: *If T0 is not initially zero, the virtual addresses of the registers are always incremented by at least one, because the delimiter test is done after the byte copy.*

For the address register operand, the incrementing process could generate an address that crosses a frame boundary. If the register is in unlinked mode, and the frame boundary is reached, the assembly debugger is entered with a trap condition indicating Crossing Frame Limit. If the register is in linked mode, it is normalized and attaches to the next frame in the linked chain. If the end of the linked set is reached during the normalization process, the following action is taken:

- If the exception mode identifier XMODE is non-zero, a subroutine call is executed to that address, to allow special handling of this condition.
- If XMODE is zero, the assembly debugger is entered with a trap condition indicating Forward Link Zero.

Mask Bytes

The mask byte can specify up to seven different characters to be tested; four of them are the standard system delimiters:

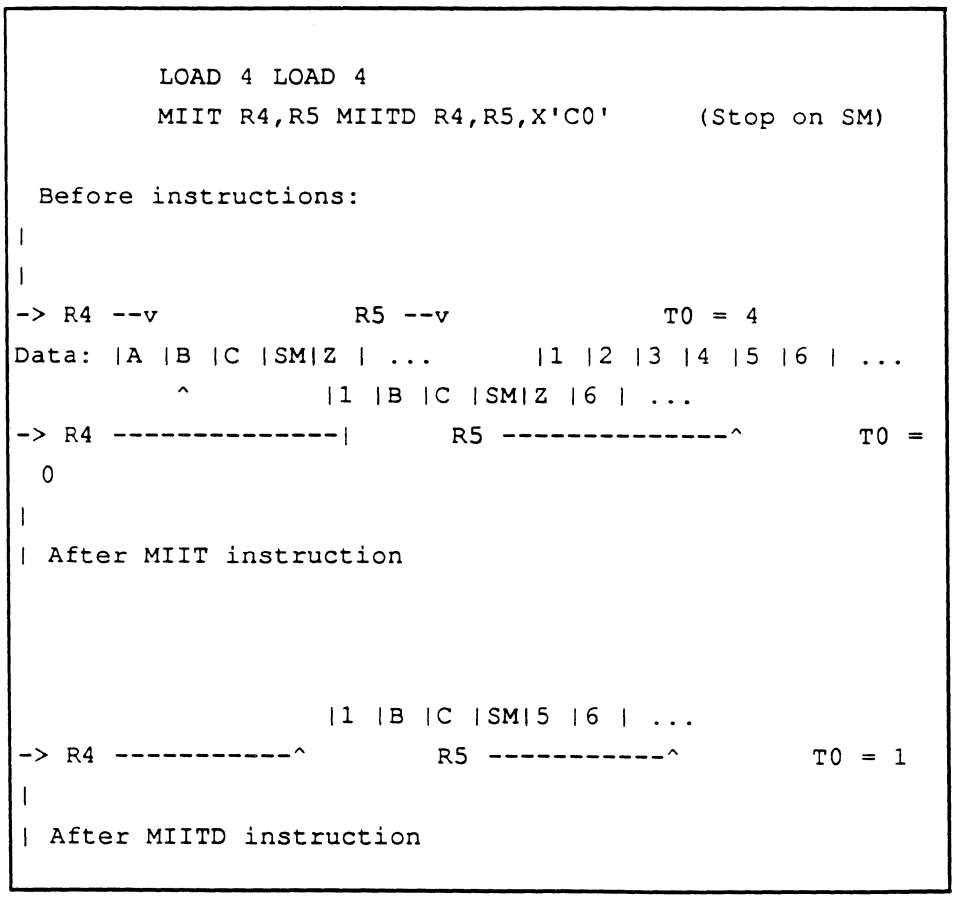
segment mark	SM	X'FF'
attribute mark	AM	X'FE'
value mark	VM	X'FD'
sub-value mark	SVM	X'FC'

The other three characters are taken from the scan character symbols SC0, SC1, and SC2. The contents of these symbols are specified by the programmer.

The low order seven bits in the mask byte are used to determine which of the seven characters are to be compared; if any bit is set (1), the corresponding character is tested; if zero (0), it is ignored.

If the high-order bit (bit 0) of the byte is set (1), it indicates that the string terminates on the first *occurrence* of a delimiter as specified by the setting of bits 1-7. If it is zero (0), it indicates that the string terminates on the first *non-occurrence* of a delimiter as specified by the setting of bits 1-7.

For more information on the use of the mask byte, see the description of SC0, SC1, and SC2 in chapter 3.



MOV (Operand)

The MOV (operand) instruction copies the contents of the first operand to replace the contents of the second operand. For MOV register instructions, see MOV (Register).

Syntax

```
MOV b,b
MOV d,d
MOV f,f
MOV h,h
MOV m,t
MOV n,d
MOV n,t
MOV s,s
MOV t,t
```

b bits

d double tally

f triple tally

h half tally

t tally

s storage register

m mode-id

n constant or literal

Description

In the MOV (operand) instruction, the contents of the first operand replace the contents of the second operand. The two operands must be of the same length.

Note: *A constant or literal cannot be moved directly to a half tally (h) or triple tally (f). Use the FTLY or HTLY directive to define a local constant as a symbol, then use the appropriate MOV.*

If the first operand is a literal, constant, or mode-id, a literal may be generated at the end of the program when assembled for some machines.

To change the virtual address of an address register, use the MOV (Register) instruction or the DEC/INC Register instruction or the SETR/SETDSP instructions. MOVing values to the PCB fields associated with address registers (RnDSP, RnFID, RnDSPFID) is illegal and may cause unpredictable results.

```
MOV   D1,RECORD

MOV   H8,H9

MOV   SR4,HSEND

MOV   T4,T3
```

MOV (Register)

The MOV (register) instruction copies the virtual address of the first register operand into the second register operand.

Syntax

MOV r,r	MOV s,r	MOV x,r
MOV r,s		MOV x,s
MOV r,x	MOV s,x	

r address register (R0-R15)

s storage register

x external address register (in another PCB)

Description

In the MOV (register) instruction, the first operand contains the address to move, and the second operand is the destination of the move.

If one of the operands is an address register that is in another PCB, use the DEFX directive to define the external register symbol. This causes the MOV instruction to be assembled with the x type operand and guarantees that the correct object code is generated no matter which machine type the code is assembled for. The assembler does not check for this condition, and if the instruction is executed on a software system, the assembled instruction will not execute properly at runtime.

When an address register is moved to a storage register, the virtual address of the address register replaces the value in the storage register. If the address register was attached, the address is converted to the detached form before the move. The address register remains unchanged.

When a storage register is copied to an address register, the address register is first detached, then the virtual address from the storage register replaces the value in the address register.

When moving one address register to another, the second register may or may not be attached after the instruction is executed, depending on the machine type.

In a multiple-processor machine, an external register is not guaranteed to be detached. Otherwise, an external register may be regarded simply as another storage register.

```
        SETR0+ R14,R0FID,2    Set R14 to DCB
R8.14   DEFX   R14,R8DSP      Define R8 in PCB that R14
                              points to
        MOV    R8.14,R15      Point my R15 to what his R8
                              points to
```

MSDB
MSXB

The MSDB (move ASCII string decimal to binary) instruction converts an ASCII decimal string into its equivalent value as a binary number, which remains in the accumulator FP0. The MSXB (move ASCII string hexadecimal to binary) instruction converts an ASCII hexadecimal string into its equivalent value as a binary number, which remains in the accumulator FP0.

Syntax

MSDB r MSXB r

r address register (R0-R15) whose virtual address +1 is the starting location of the ASCII decimal or hexadecimal character string to convert.

Description

These instructions clear the entire accumulator (T3 and FP0), then the MSDB instruction executes an MFD, and the MSXB instruction executes an MFX.

MSDB assumes a string of decimal numbers; MSXB assumes a string of hexadecimal numbers. The result is an integer stored in FP0.

See the MFD/MFX instruction for details.

MTLY
MTLYU

The MTLY (mode-id) and MTLYU (mode-id unaligned) directives reserve storage and set up the symbol in the label field as a symbol type M. The directives can also be used to only reserve storage if there is no entry in the label field.

Syntax

{symbol} MTLY m	{symbol} MTLYU m
{symbol} MTLY n,m	{symbol} MTLYU n,m
{symbol} MTLY n,n	{symbol} MTLYU n,n

symbol	name given to tally; the value of the symbol is the current program counter location.
m	mode-id
n	literal or constant

Description

There may be one or two operands. If only one operand is present, it must be a symbol of type m (mode-id). A mode-id consists of a four-bit entry point number and a twelve-bit frame number or FID.

If two operands are present, the first operand must be a constant or literal value in the range of 0-15 (X'0'-X'F') that specifies the entry point number. The second operand may be a literal or a previously defined mode-id. If a mode-id is specified, only the FID portion is used.

The MTLY form automatically aligns the tally at an even-byte boundary. The MTLYU form does not force the tally being defined to be aligned at a word (even-byte) boundary.

These directives are typically used when creating a table of mode-ids. For more information about mode-ids, see Chapter 2.

Symbols of type M may be loaded into the accumulator for use in the BSLI and ENTI instructions to transfer control indirectly to an external program frame. See also description for the DEFM directive, which defines a mode-id without creating storage.

MUL
MULX

The MUL and MULX (multiply) instructions multiplies the contents of the accumulator by the value of the operand. The MUL form addresses the accumulator as a 4-byte field (D0); the MULX form addresses it as a 6-byte field (FP0).

Syntax

MUL d	MULX d
	MULX f
MUL h	MULX h
MUL n	MULX n
MUL t	MULX t

d double tally

f triple tally (for MULX only)

h half tally

n numeric literal; if used, a 2-byte field is assumed (a range of -32,768 through +32,767). If a 1-byte literal (half tally) is being referenced, it should be defined separately using the HTLY directive. If the literal is outside the range of -32,768 through +32,767, a 4-byte literal must be separately defined using the DTLY directive, or a 6-byte literal via the FTLY directive.

The n form may generate a 2-byte literal at the end of the program when assembled for certain machines.

t tally

Description

The MUL instruction multiplies the operand value by the 4-byte field in the accumulator called D0. If the operand is a half tally (1 byte) or tally (2 bytes), it is internally sign-extended to form a 4-byte field before the multiply operation takes place.

The 8-byte result is stored in D1 and D0.

The MULX instruction multiplies the operand value by the 6-byte field in the accumulator called FP0. If the operand is a half tally (1 byte), tally (2 bytes), or double tally (4 bytes), it is internally sign-extended to form a 6-byte field before the multiply operation takes place.

The low order eight bytes of the result are stored in D1 and D0.

These instructions cannot detect arithmetic overflow or underflow.

The multiplication does not affect the original operand.

```
MUL   D4

MUL   H8

MUL   T4

MULX   D4

MULX   FP1

MULX   H8

MULX   T4

MUL   11
```

MXB

The MXB (move hexadecimal to binary) instruction converts a hexadecimal ASCII byte to its equivalent as a binary number and adds the result to a location specified by a symbol operand. For details, see the MDB/MXB instruction.

NEG

The NEG (negate) instruction replaces the value of a specified operand with its negative (two's complement).

Syntax

NEG d
NEG f
NEG h
NEG t

d double tally

f triple tally

h half tally

t tally

Description

The NEG instruction returns the negative of a value, where the value is treated as a binary number.

The negative of a value is computed by applying the two's complement operation to it. The two's complement of a number is the result of inverting each bit, then adding 1. The high order bit of a binary number is always the sign bit.

```
Original Value:    9     In Bits:     0000 1001
Inverted Value:                       1111 0110
 Addition of 1:                              +1
                                      ---------
 Negated Value:   -9                  1111 0111


Original Value:  -63     In Bits:     1100 0001
Inverted Value:                       0011 1110
 Addition of 1:                              +1
                                      ---------
 Negated Value:   63                  0011 1111
```

NEP

The NEP (not entry point) instruction defines a program location at the start of a frame as a "non-entry point". NEP executes a HALT and transfers control to the assembly debugger.

Syntax

NEP

Description

NEP interrupts execution of the current program and transfers control to the assembly debugger at entry point 11 (HALT). Program execution can be resumed only by specifying an address with the debugger G command. Alternatively, the program execution can be terminated with the BYE, END, or OFF commands.

The NEP instruction is primarily used as a debugging tool to indicate an invalid entry point. Due to software machine requirements, the HALT instruction may not be used in a branch table at the beginning of a program; the NEP instruction is to be used instead. The NEP instruction has the same effect as HALT, but guarantees that the object code is the same length as that for EP (entry point) on each type of machine.

The NEP instruction affects only the current process; it does not halt the entire multi-user system.

See Chapter 8, The System (Assembly Language) Debugger, for more information about using the debugger.

NOP

The NOP (no operation) instruction performs no action in the program; it merely causes the program to pass on to the next instruction to be performed.

Syntax

NOP

Description

The NOP instruction can be used as a placeholder for a future instruction or when patching object code on a particular machine. It can also be used to generate a small delay. However, this is all machine-dependent; NOP is not normally useful in a general-purpose program.

ONE

The ONE (set to one) instruction replaces the contents of the operand with a binary one (1) value.

Syntax

ONE d
ONE f
ONE h
ONE t

d double tally

f triple tally

h half tally

t tally

Description

The operand value becomes a binary one.

half-tally: 0000 0001

tally: 0000 0000 0000 0001

OR

The OR instruction logically ORs two bytes, and stores the result in the byte referenced by the first operand. The byte referenced by the second operand is unchanged.

Syntax

OR r,n
OR r,r

r address register

n numeric literal

Description

The logical OR operation tests two bytes, one bit at a time, for a true condition. If either bit is true (1), the result is true (1). Otherwise, the result is false (0). For example,

```
Byte 1:     0000 0101
Byte 2:     1111 0011
            ----------
Result:     1111 0111
```

The result is stored in the byte referenced by the first operand. The byte referenced by the second operand is unchanged.

```
        OR   R14,X'FD'

        OR   R14,R15
```

ORG

The ORG (origin) directive resets the program's location counter to a specified byte offset in the current program frame for use by the assembler during program assembly.

Syntax

ORG n

n number within the size of an ABS frame; may also be current program location counter function (*). The * function can be used alone, but is normally used with a +n or -n, meaning 'n' bytes before or after the current location counter value.

The ABS frame size is stored in a PSYM symbol called ID.ABS.FRAME.SIZE, and may vary among various types of firmware and software machines.

Description

The ORG directive resets the program location counter to a specified byte location.

When a program is assembled, the FRAME directive sets the location counter ORG to the address of entry point 0:

```
ORG  2        FRAME directive on many software machines
ORG  1        FRAME directive on firmware machines
```

The location counter then advances, byte by byte, as the assembler generates object code. The current location function (*) always contains the address (byte offset) of the next byte to be generated. The * function can be specified in the operand field of various directives that define data symbols in terms of the current program location.

There are several reasons to change the location counter in an explicit manner:

- To save and restore the location counter; for example, if a program is INCLUDEd that actually generates code:

```
SLOC  EQU  *           Save location counter
      INCLUDE TABLE1   Include program to get table
      ORG  SLOC        Reset in case TABLE1 has
                       object code
```

- To use byte zero of the object code. The FRAME assembler directive typically sets the location counter to 1 or 2 (not zero) because the object code begins at one. To use byte zero for storage:

```
        FRAME xxx
        ...
        ORG   0
        TEXT X'FE'        Define an attribute mark
        CMNT *            Location counter is back to 1
AM      EQU   R1          Used to reference the byte at
        CMNT *            location zero symbolically
        CMNT *            via label AM
        EP    !ENTRY0 EP  Forces location counter to 2
        CMNT *            if necessary for this machine
```

- To leave "space" in the object code for variables that the program uses. This is not recommended, since this leads to non-re-entrant (non-sharable) code, but is not prohibited. For example,

```
COUNT DEFT R1,*16
      ORG  *+2
```

  Since the tally COUNT occupies two bytes in the object code, the ORG *+2 is used to "space" over these two bytes.

Programmers are advised to make sure that any absolute number is a safe ORG before using it. An ORG instruction with an absolute offset that is placed after code will probably be incorrect when assembled for different machine types since the object code is of different lengths. The only exception is when the offset is prior to any code, or beyond any conceivable expansion of code in the frame.

One area where a safe ORG can be easily guaranteed is past the last entry point. It can be assumed that no implementation will have entry point zero start at a location greater than X'002'. Furthermore, it can be assumed that an entry point will require no more than four bytes in any implementation. For example,

```
FRAME   511
*
EP      LABL1       0 starts no higher than X'002'
NEP     *           1 starts no higher than X'006'
                       (2+4)
ORG     10          Safe ORG (2+4+4)
```

However, it cannot be assumed that the next available location that the assembler will assign after a FRAME directive is location X'001'. If it is necessary to assign data starting at location X'001' then an ORG 1 directive should be inserted. For example:

```
FRAME 512
.                          (all comments)
ORG  1
TEXT  C'Data not after EP',X'FF'
```

```
FRAME 513
*
 ...
ORG  -11+ID.ABS.FRAME.SIZE
TEXT  X'0C0C0C0C'
TEXT  X'0C0C0C0C'
TEXT  X'0C0C0D'
```

The X'0D' byte will end up in the last byte of the frame regardless of the ABS frame size.

OUT1B
OUT1BX

The OUT1B instruction stores the character (byte) addressed by the register operand in the next location in the asynchronous output buffer. The OUT1BX form is used anywhere when a write is needed only to echo a character read in by a INP1BX instruction.

Syntax

OUT1B r
OUT1BX r

r address register (R0-R15) that contains the virtual address of the character to be output

Description

The OUT1B instruction outputs a character to a process's asynchronous channel (normally connected to a terminal). The OUT1BX instruction is used anywhere that the write is only to echo a character read in by a INP1BX instruction.

The virtual addressed by the register is stored in the next location in the asynchronous channel output buffer. If the output buffer is full, the process is suspended until characters are removed from the buffer by the asynchronous channel controller.

Caution: *The OUT1B and OUT1BX instructions are not compatible with the TERM-VIEW and character translation features of the Ultimate operating system. To ensure that your programs are compatible with these features, use the system subroutines WRITE@OB or WRITEX@OB. These subroutines are described in Chapter 5, System Subroutines.*

RQM

The RQM (release quantum) instruction releases a process's timeslice, thereby deactivating the process.

Syntax

RQM

Description

The RQM instruction is a request to the Kernel to turn over control to the next process in line. The process that executed the RQM is reactivated after other active processes in the process chain have executed their timeslices.

RQM typically causes a process to sleep (remain deactivated) for about 50 milliseconds, though this varies with machine type.

See the XCC instruction for examples of RQM usage.

RTN

The RTN (return from subroutine) instruction transfers program control to the location specified by the current entry in the return stack.

Syntax

RTN

Description

This instruction exits a subroutine that has been called via a BSL instruction. It does not matter whether the subroutine had been called locally or externally.

If there are no entries in the return stack, the assembly debugger is entered with a Return Stack Empty trap condition.

The assembly subroutine return stack is in the user's PCB. Up to 125 entries can be placed in the return stack.

An entry can be deleted from the return stack by the instruction POPRTN. This is mandatory if a subroutine is to be exited without using a RTN instruction.

The entire return stack can be reset by the instruction INITRTN, which may be useful in conditions where a process is to be re-initialized, and all current entries in the stack are to be deleted or ignored.

SB

The SB (set bit) instruction sets the referenced bit to an "on" condition (that is, 1 or true).

Syntax

```
SB b
```

b specifies the bit symbol that is to be set

The SB instruction sets a bit flag or switch in a program. The referenced bit value is set to 1.

For information on a related instruction, see ZB instruction.

SET.TIME

The SET.TIME instruction resets the system's internal time and date.

Syntax

SET.TIME

Description

The SET.TIME instruction is a monitor call (that is, it executes an external subroutine call to the operating system kernel) that is included as part of the instruction set.

SET.TIME assumes that the accumulator FP0 has been set up as follows:

T2 (upper two bytes of FP0)	contains the date as a number of days past December 31, 1967.
D0 (lower four bytes of FP0)	contains the time as a number of milliseconds past midnight.

```
SET.TIME
```

SETDSP

The SETDSP instruction sets the displacement (DSP) field of an address register to the specified value.

Syntax

SETDSP r,n
SETDSP r,t
SETDSP0 r
SETDSP1 r

r specifies the address register (R0-R15) to be set up.

n constant or literal value that is to be used as displacement

t tally symbol that contains value to be used as displacement

The SETDSP0 instruction sets the displacement to zero (0); the SETDSP1 instruction sets the displacement to one (1).

Description

The SETDSP instruction is used to set the displacement portion of an address register's virtual address to a specific value. This is an alternative to setting up a virtual address in a storage register, then using the MOV (Register) instruction to move the virtual address into the address register.

SETDSP sets the displacement field of an address register to the specified value (0, 1, or the n or t value). If an n value is specified, a 2-byte literal may be generated at the end of the program when assembled for certain machine types, and the instruction will assemble the same as if a t symbol had been specified. The register may be detached after execution of the instruction, depending on machine type.

The following example shows the effect of each form of the SETDSP instruction, regardless of the implementation details on any particular machine type (such as detaching registers):

```
SETDSP   r,m
 MOV     m,RnDSP

SETDSP   r,t
 MOV     t,RnDSP

SETDSP0  r
 ZERO    RnDSP
```

```
SETDSP1 r
 ONE     RnDSP
```

Caution: *The above merely shows the effective instruction generated by each form of the SETDSP instruction. Do* ***not*** *try to substitute the descriptive code for the SETDSP instruction, because register attachment needs to be handled correctly for each machine type.*

The following are examples of SETDSP instructions and the corresponding macro expansions on a firmware machine.

```
SETDSP0  R13
+DETZERO  R13*

SETDSP1  R13
+DETONE   R13*

SETDSP   R13,5
+DETZERO  R13
+MOV      5,R13DSP

SETDSP   R13,T5
+DETZERO  R13
+MOV      T5,R13DSP
```

* The DETONE and DETZERO instructions are for use only by the firmware assembler in macros such as SETDSP. They should not be coded directly, in order to guarantee compatibility of source code across all system types, firmware and software.

SETR

The SETR (set register) instruction sets the FID and displacement (DSP) fields of an address register to the specified values.

Syntax

```
SETR r,n
SETR r,n,n          SETR0{+/-} r,n      SETR1{+/-} r,n
SETR r,t            SETR0{+/-} r        SETR1{+/-} r
SETR+ r,n,d,n       SETR0+ r,d,n        SETR1+ r,d,n
SETR+ r,t,d,n       SETR0- r,d,n        SETR1- r,d,n
SETR- r,n,d,n
SETR- r,t,d,n
SETR{+/-} r,n,d     SETR0{+/-} r,d      SETR1{+/-} r,d
SETR{+/-} r,t,d
```

r address register (R0-R15) to be set up

d double tally

f triple tally

h half tally

n literal or constant

t tally

Description

The SETR instruction is used to change the virtual address of an address register to a specified value. It is an alternative to setting up the virtual address in a storage register, then using the MOV (Register) instruction to move the virtual address into the address register.

There may be one, two, three or four operands; the first operand always specifies the address register to set up.

With the first set of formats (SETR, SETR+, and SETR-), the second operand specifies the displacement to use. If the third operand is present, it specifies the FID to use; if not present, the FID value comes from the accumulator (INCed if SETR+ or DECed if SETR-). The fourth operand is valid only if the SETR+ or SETR- form is used; it specifies the value by which to INC or DEC the FID; if not present, 1 is used to increment or decrement the third operand.

With the second and third set of formats, SETR0 specifies a displacement of 0 and SETR1 specifies a displacement of 1. If a second operand is present, it specifies the FID to use; if not present, the FID comes from the accumulator (INCed if SETR0+/SETR1+ or DECed if SETR0-/SETR1-). The third operand is valid only if the SETR0+/SETR1 or SETR0-/SETR1- is used; it specifies the value by which to INC or DEC the FID; if not present, 1 is used to increment or decrement the third operand.

If an n value is a FID or a FID INC/DEC value, a 4-byte literal may be generated.

SHIFT

The SHIFT (shift) instruction shifts the value of the byte addressed by a register operand by one bit to the right, sets the leftmost (high order) bit to zero, and stores the new byte value at the virtual address of the second register operand.

Syntax

SHIFT r,r

r address register (R0-R15)

Description

The first operand specifies the byte to be shifted. The second operand specifies the storage location of the new byte value.

The value of the byte referenced by the first operand is logically shifted one bit; the vacated leftmost bit is set to zero. The result is stored at the location addressed by the second operand. The byte referenced by the first operand is unchanged.

```
        SHIFT   R14,R15
```

SICD

The SICD (scan string to count of delimiters) instruction scans a string starting from the character addressed by a register operand until a specified number of a delimiter has been scanned.

Syntax

SICD r,n

r address register (R0-R15) whose virtual address points to the character where the scan begins

n constant or literal value that specifies the mask of delimiter criteria to use for the string being scanned

Description

The SICD instruction scans a string until a specified count of a delimiter has been reached. The result is to position the address register at a specific point within a data structure.

The SICD instruction expects that T0, the low-order tally of the accumulator, has been set up to contain the count of delimiters to be scanned over. If T0 is initially zero, the results are unpredictable.

The register operand is incremented by one; the character then addressed is examined. This operation is repeated until the terminating condition specified by T0 and the second operand (the mask byte) is met. If the initial condition of the accumulator and the mask byte matches the terminating condition, no operation is performed. Each byte is tested after it has been scanned, to see if it satisfies the terminating condition.

Note *The mask byte used by SICD is different from the one used in the SID, SIDC, SITD, MIID, MIIDC, and MIITD instructions.*

Only **one** of six possible delimiters may be specified as the test character in the SICD instruction.

Three of the possible scan delimiters are fixed, and are the standard system delimiters (excluding the segment mark):

attribute mark	AM	X'FE'
value mark	VM	X'FD'
sub-value mark	SVM	X'FC'

The other three delimiters are variable, and the programmer may set up the desired test character in one of the scan characters SC0, SC1, and SC2.

Six bits in the mask byte are used to determine which one of the six above characters is to be compared; if a bit is set (1), the corresponding character in the scan string is tested; if zero, it is ignored.

Bits 0 and 1 set up additional criteria, as follows:

0 bit 0 (high-order bit) of the mask, if set, indicates that the accumulator T0 should be decremented by 1 before the scan is started and the terminating condition tested. If zero, T0 is not decremented.

1 bit 1 specifies the condition for abnormal termination of the scan. If set, the scan terminates abnormally if a character is found that is logically higher than the character in SC2. If zero, the scan terminates abnormally if a character is found that is logically higher than the delimiter being scanned for. If the delimiter being scanned for is in SC2, therefore, the state of this bit does not matter.

See Figure 4-1. (The parentheses around SC0, SC1 and SC2 are to indicate that it is the contents of these locations that are compared.)

The scan can terminate either normally or abnormally. It terminates normally if the number of delimiters specified in T0 (pre-decremented if required) is encountered. In this case, T0 is zero, and the register points to the final delimiter (or is unchanged if no scan takes place).

The scan terminates abnormally if a character higher than that in SC2 (mask bit 1 on) or higher than the delimiter (mask bit 1 off) is encountered. In this case, the value remaining in T0 is the number of

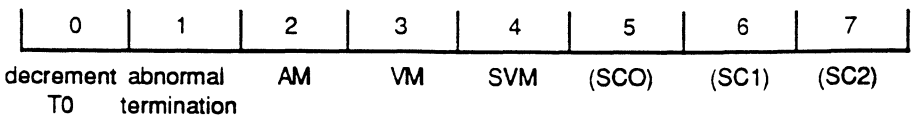


Figure 4-1. SICD Mask Byte Format

delimiters that must be inserted in the data to create the required data position, and the register points one byte before the character that caused the scan to terminate.

A few examples should make this clear:

Mask byte	Bit pattern	Meaning
X'A0'	1010 0000	Stop on nth occurrence of an AM, or on the first SM; decrement T0 by 1 before starting scan.
X'20'	0010 0000	Stop on nth occurrence of an AM, or on the first SM; do not decrement T0 before starting.
X'02'	0000 0010	Stop on nth occurrence of the contents of SC1, or on the first character higher; do not decrement T0 before starting scan.
X'42'	0100 0010	Stop on nth occurrence of the contents of SC1, or on the first character higher than the contents of SC2; do not decrement T0 before starting scan.

For the address register operand, the incrementing process could generate an address that crosses a frame boundary. If the register is in unlinked mode, and the frame boundary is reached, the assembly debugger is entered with a trap condition indicating Crossing Frame Limit. If the register is in linked mode, it is normalized and attaches to the next frame in the linked chain. If the end of the linked set is reached during the normalization process, the following action is taken:

- If the exception mode identifier XMODE is non-zero, a subroutine call is executed to that address, to allow special handling of this condition.
- If XMODE is zero, the assembly debugger is entered with a trap condition indicating Forward Link Zero.

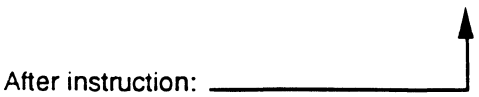
The following are some examples of SICD usage.

Scan to attribute 3:

```
        LOAD   3              T0 = 3
        SICD   R15,X'20'      Scan to AM
```

Before instruction: R15 mask byte: 0010 0000

E0 ^ E11] E12 ^ E2 ^ E31] E321 \ E322] E33 ^ _



After instruction:

T0 = 0

Scan to attribute 6:

```
        LOAD   6              T0 = 6
        SICD   R15,X'A0'      Decrement T0, scan to AM
```

Before instruction: R15 mask byte: 1010 0000

Data E0 ^ E11] E12 ^ E2 ^ E31] E321 \ E322] E33 ^ _

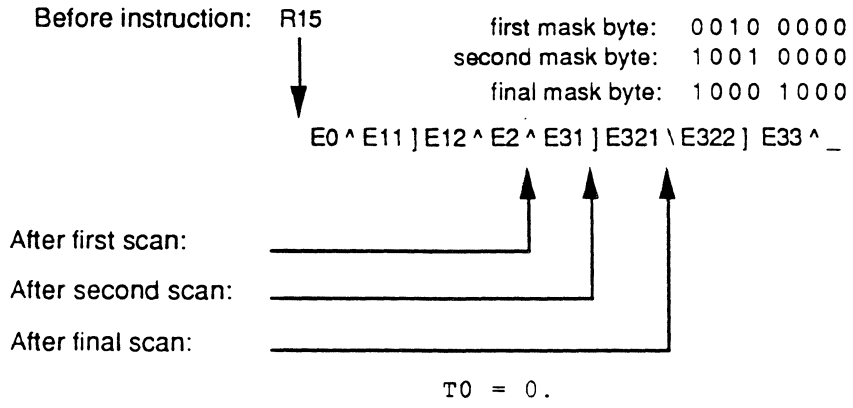
After instruction:

T0 = 2

Note that R15 has been backed off one byte from the segment mark.

Scan to attribute 3, value 2, subvalue 2:

```
LOAD   3              T0 = 3
SICD   R15,X'20'      Scan to AM
LOAD   2              T0 = 2
SICD   R15,X'90'      Decrement T0, scan to VM
LOAD   2              T0 = 2
SICD   R15,X'88'      Decrement T0, scan to SVM
```



SID
SIDC

The SID (scan incrementing string to delimiter) instruction scans a string starting from the character addressed by a register operand until a specified delimiter has been scanned. SIDC also counts the number of characters scanned.

Syntax

SID r,n SIDC r,n

r address register (R0-R15) whose virtual address +1 contains the character where the scan begins

n constant or literal value that specifies the mask of delimiter criteria to use for the string being scanned.

Description

The register operand is incremented by one; the character then addressed is examined. This operation is repeated until the terminating condition specified by the second operand (the "mask" byte) is met.

Each byte is tested after it has been scanned to see if it satisfies the terminating condition.

Note: The virtual address of the register is always incremented by at least one, because the delimiter test is done after the byte scan.

The SIDC instruction expects that T0, the low-order tally of the accumulator, has been set up (usually to ZERO or ONE) so that on termination, it indicates the number of characters scanned. With the SIDC instruction, as each byte is examined, T0 is decremented by one. No other sections of the accumulator are affected. If T0 is set to ZERO (0), its value after the instruction terminates is the negative of the length of the string, including the delimiter. If T0 is set to ONE (1), its value is the negative of the string length excluding the delimiter.

For the address register operand, the incrementing process could generate an address that crosses a frame boundary. If the register is in unlinked mode, and the frame boundary is reached, the assembly debugger is entered with a trap condition indicating Crossing Frame Limit. If the register is in linked mode, it is normalized and attaches to

the next frame in the linked chain. If the end of the linked set is reached during the normalization process, the following action is taken:

- If the exception mode identifier XMODE is non-zero, a subroutine call is executed to that address, to allow special handling of this condition.
- If XMODE is zero, the assembly debugger is entered with a trap condition indicating Forward Link Zero.

Mask Bytes

The mask byte can specify up to seven different characters to be tested; four of them are the standard system delimiters:

segment mark	SM	X'FF'
attribute mark	AM	X'FE'
value mark	VM	X'FD'
sub-value mark	SVM	X'FC'

The other three characters are taken from the scan character symbols SC0, SC1, and SC2. The contents of these symbols are specified by the programmer.

The low order seven bits in the mask byte are used to determine which of the seven characters are to be compared; if any bit is set (1), the corresponding character is tested; if zero (0), it is ignored.

If the high-order bit (bit 0) of the byte is set (1), it indicates that the string terminates on the first *occurrence* of a delimiter as specified by the setting of bits 1-7. If it is zero (0), it indicates that the string terminates on the first *non-occurrence* of a delimiter as specified by the setting of bits 1-7.

For more information on the use of the mask byte, see the description of SC0, SC1, and SC2 in chapter 3.

The following are some examples of SID(C) usage:

Instruction:

```
        SIDC R4,X'C0'          SCAN UNTIL SEGMENT MARK
                               T0 = 0
Before instruction:  R4        mask byte:     1100 0000
                     ↓
                   A^B^C^_
                     ↑
After instruction:   ——

                               T0 = -3

        SRA  R15,SC1
        MCC  C' ',R15
        SID  R4,X'02'          SCAN UNTIL NON-BLANK

Before instruction:  R4        mask byte:     0000 0010
                     ↓
Data               ^ ^ ^X^ ^...
                     ↑
After instruction:   ——
```

SIT
SITD

The SIT (scan incrementing string, T0 termination counter) instruction scans the characters addressed by the first operand until a specified number of characters has been moved. The SITD (scan incrementing string, T0 termination or delimiter) instruction performs the same operation as SIT, but terminates the move at a specified number of characters or when a specified delimiter is encountered.

Syntax

SIT r SITD r,n

n constant or literal values that specifies the "mask" of delimiters to use as terminators for the string being scanned

r address register whose virtual address +1 references the starting character of the string to be scanned

Description

The SIT and SITD instructions can be used whenever a data character string needs to be scanned or counted, if the string field is of fixed length or the string delimiters can be specified in a mask byte. The mask byte contains flags for the four system delimiters plus three user-specified characters; it also has a match/nomatch flag that allows the move to terminate on either a match or nomatch with the specified delimiters.

These instructions also expect that T0, the low-order tally of the accumulator, has been set up for the maximum string length to scan.

If T0 is zero at the start of this instruction, no action takes place .

These instructions are used as an alternative to merely INCing a register by T0 when a program intends to use the XMODE facility in cases where a frame boundary is crossed. SIT is logically equivalent to the INC r,t instruction, except that additional frames can be linked via XMODE (see below).

The register operand is incremented by one; the character then addressed is scanned and compared. T0 is decremented by one. This operation is repeated until one of the following terminating conditions is met:

- For the SIT instruction, when T0 reaches zero. This instruction is typically used to scan over a fixed length string.
- For the SITD instruction, when T0 reaches zero or when one of the delimiter bytes specified by the mask byte is encountered. The terminating condition is found by testing each byte after it has been scanned. This instruction is typically used to scan over a delimited string of preset maximum length. Additional frames can be linked on to the end of the linked set by using XMODE (see below).

Note: If T0 is not initially zero, the virtual address of the register will always be incremented by at least one, because the delimiter test is done after the byte scan.

Mask Bytes

The mask byte can specify up to seven different characters to be tested; four of them are the standard system delimiters:

segment mark	SM	X'FF'
attribute mark	AM	X'FE'
value mark	VM	X'FD'
sub-value mark	SVM	X'FC'

The other three characters are taken from the scan character symbols SC0, SC1, and SC2. The contents of these symbols are specified by the programmer.

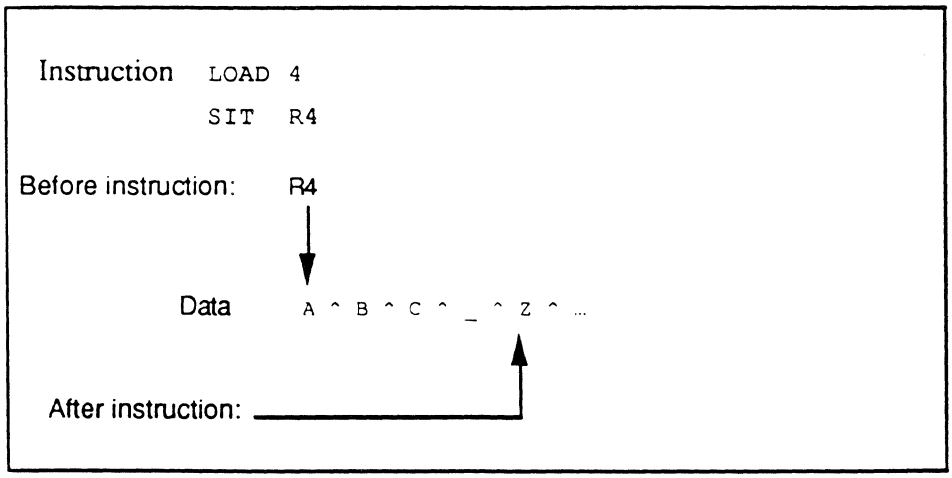
The low order seven bits in the mask byte are used to determine which of the seven characters are to be compared; if any bit is set (1), the corresponding character is tested; if zero (0), it is ignored.

If the high-order bit (bit 0) of the byte is set (1), it indicates that the string terminates on the first *occurrence* of a delimiter as specified by the setting of bits 1-7. If it is zero (0), it indicates that the string terminates on the first *non-occurrence* of a delimiter as specified by the setting of bits 1-7.

For more information on the use of the mask byte, see the description of SC0, SC1, and SC2 in chapter 3.

For the address register operand, the incrementing process could generate an address that crosses a frame boundary. If the register is in unlinked mode, and the frame boundary is reached, the assembly debugger is entered with a trap condition indicating Crossing Frame Limit. If the register is in linked mode, it is normalized and attaches to the next frame in the linked chain. If the end of the linked set is reached during the normalization process, the following action is taken:

- If the exception mode identifier XMODE is non-zero, a subroutine call is executed to that address, to allow special handling of this condition.
- If XMODE is zero, the assembly debugger is entered with a trap condition indicating Forward Link Zero.



SLEEP

The SLEEP instruction causes the process to be deactivated and put in a wait state until a specified time of day.

Syntax

SLEEP

Description

The SLEEP instruction is typically used when the process is waiting for an event to occur. If the process executes instructions continuously, it is a waste of the system's resources. The SLEEP instruction is inserted as a means of delaying the process until a specific time of day.

This instruction expects that the accumulator D0 has been loaded with the "awakening" time of day in internal system format (number of milliseconds past midnight). If D0 contains a value higher than 86,400,000, the process will sleep "forever."

A sleeping process can be awakened from the process' own terminal by the BREAK key. See also the RQM instruction.

SR

The SR (storage register) assembler directive defines a program address and creates a storage register containing that address. The storage register (data type 's') is in linked format. Six bytes of storage for the address are reserved at the current location counter.

Syntax

label SR n,n
label SR a

n the virtual address to reserve for the symbol. The first operand specifies the displacement of the generated virtual address. The second operand specifies the frame number (FID) as a decimal number (linked frame) or hexadecimal number (unlinked frame).

a defines both FID and displacement to specify the virtual address

Description

An SR directive can be used to set up a symbol as a storage register pointing to data in a specific frame, which can be in linked or unlinked mode.

In the first SR form, the second operand specifies the frame, or FID, as a 4-byte field. If the high-order bit of the second operand's value is set, the virtual address is assumed to be in unlinked format. If the high-order bit is zero, it is assumed to be in linked format.

In the second SR form, the address specified by a the a operand must have been previously defined via a DEFRA directive. In this case, the virtual address is always in linked format.

See the section in the chapter on Data Addressing for a full description of linked and unlinked modes of addressing.

Note: *The ADDR directive also creates a storage register, normally for data stored within a program. The frame is assumed to be in unlinked format (that is, an abs frame) and the displacement is relative to location 0 of the frame.*

`SR 1,100`	Addresses frame 100 in linked mode; the first data byte in the frame depends on machine type, and is equal to the value of the PSYM symbol ID.LINK.SIZE.
`SR 1,X'80000064'`	Frame is in unlinked mode; the first data byte is at location 1 in the frame.
`MOV F100U,R15`	Sets R15 to point to the above address.

SRA

The SRA (set register to address) instruction sets up an address register operand with the virtual address of a specified operand.

Syntax

```
SRA r,c
SRA r,d
SRA r,f
SRA r,h
SRA r,l
SRA r,s
SRA r,t
```

r address register (R0-R15) that is being set up

c character

d double tally

f triple tally

h half tally

l label

s storage register

t tally

The second operand may also be another register and an offset (such as R15;T9 for the 9th tally off R15).

Description

The SRA instruction is used to "point" an address register to a location that is specified by the second operand. It is typically used to address locations in the object code (text strings, for example), or to address the first byte of a symbol so that sections of it can be manipulated in ways not otherwise possible.

It is also another legitimate way of changing the virtual address of an address register in addition to MOV (register), INC/DEC (register), and SETDSP or SETR.

An SRA to a local label works only when the location of the label is less than X'100' (that is, in the first part of the frame). This is because a label is addressed relatively via a byte offset, and the maximum offset

can be 255 or X'FF'. If it is necessary to address a label at or beyond location X'100', one way is to make the label of type T using instructions of the form:

```
        ALIGN *              Need to align location on
        CMNT *               word boundary!
LAB1    DEFT R1,*16          Define LABEL as "here" (*16
        CMNT *               gets offset as words, not
        CMNT *               bytes)
```

Now a SRA r,LAB1 would work correctly.

The following are also examples of SRA usage.

```
FILENAME EQU  *-1
         TEXT C'INVN',X'FE'
         ...
         ...
         SRA  R15,FILENAME This sets R15 to address one
         CMNT *            byte before the byte 'I' in
         CMNT *            the string'INVN'. Typically,
         CMNT *            R15 then used in a MIID
         CMNT *            instruction to copy the
         CMNT *            string, up to the AM, to
         CMNT *            another location.

         SRA  R15,D0       This sets R15 to point to
         CMNT *            the first byte of the
         CMNT *            accumulator D0.

         SRA  R15,H3       Same as above (see format of
         CMNT *            accumulator).

         SRA  R15,R14;T2   This sets R15 to point to
         CMNT *            word 2 off the virtual
         CMNT *            address of R14.
```

STORE

The STORE instruction stores the contents of the accumulator in a specified operand.

Syntax

```
STORE d
STORE f
STORE h
STORE t
```

d double tally

f triple tally

h half tally

t tally

Description

The STORE instruction is used whenever an accumulator value needs to be stored as the value of a symbol in a program.

The contents of the accumulator (H0, T0, D0 or FP0) replace the contents of the operand.

The accumulator is not changed.

SUB
SUBX

The SUB and SUBX instructions subtract the contents of the operand from the accumulator. The SUB form addresses the accumulator as a 4-byte field (D0); the SUBX form addresses it as a 6-byte field (FP0).

Syntax

SUB d	SUBX d
	SUBX f
SUB h	SUBX h
SUB n	SUBX n
SUB t	SUBX t

d double tally

f triple tally (for SUBX only)

h half tally

n numeric literal; if used, a 2-byte field is assumed (a range of -32,768 through +32,767). If a 1-byte literal (half tally) is being referenced, it should be defined separately using the HTLY directive. If the literal is outside the range of -32,768 through +32,767, a 4-byte literal must be separately defined using the DTLY directive, or a 6-byte literal via the FTLY directive.

The n form may generate a 2-byte literal at the end of the program when assembled for certain machines.

t tally

Description

The SUB instruction subtracts the operand value from the 4-byte field in the accumulator called D0. If the operand is a half tally (1 byte) or tally (2 bytes), it is internally sign-extended to form a 4-byte field before the subtract operation takes place. The result is stored in D0.

The SUB instruction cannot detect arithmetic overflow or underflow.

The SUBX instruction subtracts the operand value from the 6-byte field in the accumulator called FP0. If the operand is a half tally (1 byte), tally (2 bytes), or double tally (4 bytes), it is internally sign-extended to form a 6-byte field before the subtract operation takes place. The 6-byte result is stored in FP0.

The subtraction does not affect the original operand, or the other sections of the accumulator.

TEXT

The TEXT assembler directive stores one or more text strings in a program.

Syntax

TEXT operand1{,operand2,...}

operand string in either ASCII (C'xxx') or hexadecimal (X'nnn') format

Description

The TEXT instruction is typically used to store literal strings, messages, tables of values, etc.

The specified text strings are generated at the location in which they appear in the program. ASCII strings are stored in their equivalent ASCII hexadecimal value; hexadecimal strings are stored exactly as specified.

See the SRA instruction for the method of addressing generated data.

```
        TEXT C'ABCD',X'07FF'
```

TIME

The TIME instruction retrieves the system's time and date in internal format.

Syntax

TIME

Description

The TIME instruction is a monitor call (that is, it executes an external subroutine call to the operating system kernel) that is included as part of the instruction set.

Note: The internal format for system time is in milliseconds, not seconds, and therefore differs from the BASIC function TIME(), which returns time as the number of seconds past midnight.

TIME retrieves the current system time and date and loads it into the accumulator FP0 as follows:

T2 (upper two bytes of FP0)	contains the date as a number of days past December 31, 1967.
D0 (lower four bytes of FP0)	contains the time as a number of milliseconds past midnight.

This is the same format as used by SET.TIME.

TLY

The TLY directive defines a tally (16 bits, or one word). See the DTLY directive for details.

XCC

The XCC (exchange characters) instruction replaces the character addressed by the first operand with the character addressed by the second operand, and vice versa.

Syntax

XCC r,r

r address registers (R0-R15) that contain the virtual addressess of the two characters to be exchanged

Description

The character addressed by the first operand is exchanged with that addressed by the second operand.

The XCC instruction allows the "Test and Set" function to be implemented, which can be used to prevent shared usage of sections of code, similar to the following:

```
        SRA R15,LOCKTBL    Set R15 to the Lock byte,
        CMNT *             which may contain either a
        CMNT *             X'00' (unlocked) or X'01'
        CMNT *             (locked)
LOCKD?  MCC X'01',R2       Move 'lock' flag to scratch
        CMNT *             location
        XCC  R2,R15        Exchange old lock and 'lock'
        BCE  R2,X'00',OK   flag; if old flag was X'00',
        RQM  *             continue, else wait a while
        B    LOCKD?        and try again.
OK      EQU  *             Start of non-shared code
          .
          .
        MCC X'00',R2       Set up 'unlock' flag
        XCC R2,R15         Store it at LOCKTBL
```

Anything locked via XCC should be unlocked via XCC as well, because on a multi-processor system only the XCC instruction guarantees that only one process can access the lock byte at a time. An MCC executed by one processor may change the value of a byte referenced by an XCC executed simultaneously on another processor, invalidating the lock value. But if both processors use XCC, only one at a time will be given access to the memory location of the byte.

XOR

The XOR (exclusive OR) instruction logically exclusive-ORs two bytes, and stores the result in the byte referenced by the first operand.

Syntax

XOR r,n
XOR r,r

r address register (R0-R15) that contains the virtual address of the character to be tested

n numeric literal

Description

An exclusive-OR operation tests two bytes, one bit at a time, for a true condition. If one and only one bit is true, the result is true (1). If both or neither are true, the result is false (0). For example:

```
Byte 1:     0000 0101
Byte 2:     1111 0011
            ---------
Result      1111 0110
```

The result is stored in the byte referenced by the first operand. The byte referenced by the second operand is unchanged.

XRR

The XRR (exchange registers) instruction exchanges the virtual addresses of two address register operands.

Syntax

XRR r,r

r address registers (R0-R15) that contain the virtual addresses of the character to be exchanged

There are two address register (R0-R15) operands whose virtual addresses are to be exchanged.

Description

The first operand's content (virtual address) is exchanged with that of the second operand. On firmware machines, the attached or detached state of the address registers is not guaranteed after execution of this instruction.

ZB

The ZB (zero bit) instruction clears the referenced bit to an "off" condition (that is, 0 or false).

Syntax

ZB b

b bit symbol that is to be cleared

Description

The ZB instruction can be used whenever a bit flag or switch needs to be cleared in a program.

The referenced bit value is cleared to 0.

ZERO

The ZERO (set to zero) instruction replaces the contents of the operand with a zero value.

Syntax

ZERO d
ZERO f
ZERO h
ZERO t

d double tally

f triple tally

h half tally

t tally

Description

The value of the operand is replaced by binary zero.

half-tally: 0000 0000

tally: 0000 0000 0000 0000

```
        ZERO   D0

        ZERO   FP0

        ZERO   H0

        ZERO   T0
```

Notes

5 System Subroutines

Assembly programming for Ultimate systems is facilitated by the system software routines provided to handle disk file management, terminal I/O and other system-wide functions.

The system software routines work with a standard set of address registers, storage registers, tallies, character registers, bits, and buffer pointers. These standard symbols are collectively called functional elements. In order to use any of these routines, the calling routine must set up the appropriate functional elements as required by the called routine's input interface.

The standard set of functional elements is predefined in the permanent symbol file (PSYM), and is therefore always available to the programmer. Also included in PSYM are the mode-ids (program entry points) for the standard system routines documented in this chapter.

User-written assembly programs are often written for two puposes:

- To call a system-supplied subroutine from within a user program. This chapter describes the system- supplied subroutines available to the user program once it has been invoked.
- To initiate a user program from the Ultimate operating system and, after execution, to return control to the operating system. Below are listed the user program interfaces with the operating system. Chapter 6 details the programming requirements for each interface.

CONV	interfaces to conversion software
PROC	interfaces to PROC (procedure) software
RECALL	interfaces to Ultimate RECALL software
TCL-I	interfaces to TCL with no file input
TCL-II	interfaces to TCL for programs with file input
WRAPUP	interfaces to TCL (messages, exiting)
XMODE	interfaces to routine set up by programmer for handling of end-of-linked frame set

Summary of the System Subroutines

The following section categorizes the system subroutines according to function.

Terminal and Printer I/O Routines

CRLFPRINT	(see PRINT)
GETBUF	read data from terminal
NEWPAGE	skips to new page, print heading/footing
PCRLF	prints cr/lf sequence
PERIPHREAD1	reads asynchronous channel
PERIPHREAD2	reads asynchronous channel
PERIPHWRITE	writes asynchronous channel
PRINT	print stext from object code to terminal
PRNTHDR	initializes and prints heading/footing
READ@IB	reads single character, echoes it
READIB	reads a line from terminal
READLIN	reads a line from terminal
READLINX	reads a line from terminal
READX@IB	reads single character, does not echo it
RESETTERM	(see SETTERM)
SETLPTR	sets up characteristics of printer
SETTERM	sets up characteristics of terminal
SYSTEM-CURSOR	positions cursor/sets visual effects
TERM-INFO	gets function key definitions for terminal
WRITE@OB	outputs a single character
WRITEX@OB	outputs single character, unless terminal does local echo
WRITOB	writes a line to the terminal or printer
WRTLIN	writes a line to the terminal or printer

Disk File I/O Routines

GETACBMS	opens the ACC file
GETFILE	opens dictionary *or* data section of file
GETITM	gets next sequential item from file
GLOCK	locks a file group
GUNLOCK	unlocks a file group
GUNLOCK.LINE	unlocks all group locks for a line
HASH	computes record that item-id hashes to
OPENDD	opens dictionary and data sections of file
RETIX	reads a specific item from a file
RETIXU	(see RETIX)
UPDITM	writes a specific item to a file

Tape I/O Routines

TPBCK	backspaces tape one record
TPRDBLK	reads a tape record
TPRDLBL1	reads tape label
TPREAD	(see TPRDBLK)
TPREW	rewinds the tape
TPWEOF	writes end of file
TPWRITE	writes a tape record

Stack Routines

INITRTN	initializes the return stack
MARKRTN	divides the extended return stack into logical stacks
POPRTN	pops the top entry off the return stack
RTNMARK	moves a logical stack into the PCB

Frame Management

ATTOVF	attaches overflow frame automatically
GETBLK	gets a block of overflow frames
GETOVF	gets a frame of overflow space
LINK	initializes link fields
NEXTIR	obtains next forward linked frame
NEXTOVF	attaches overflow frame via register
RDLINK	reads link fields of frame
RDREC	reads one frame
RELBLK	releases a block of overflow space
RELCHN	releases a chain of overflow frames
RELOVF	releases a single overflow frame
WTLINK	writes link fields of frame

Character Conversion

ACONV	converts ASCII character to EBCDIC
CONV	calls conversion processor
CVDxx subs	converts ASCII decimal to binary
CVXxx subs	converts ASCII hexadecimal to binary
ECONV	converts EBCDIC character to ASCII
MBDNSUB	converts binary to decimal ASCII string, padded
MBDNSUBX	converts binary to decimal ASCII string, padded
MBDSUB	converts binary to decimal ASCII string
MBDSUBX	converts binary to decimal ASCII string

Miscellaneous Routines

ANDIOFLGS	sets I/O flags
DATE	gets system tdate
DECINHIB	decrements the INHIBITH counter
GETIOFLGS	gets I/O flags
HSISOS	initializes IS, OS and HS buffer pointers
LINESUB	gets user's line number
ORIOFLGS	sets I/O flags
SLEEP	puts terminal to sleep
SLEEPSUB	puts terminal to sleep
SORT	sorts a string of keys
TIMDATE	gets system time and date
TIME	gets system time
WSINIT	initializes buffer pointers

Conventions Used to Describe System Subroutines

Each system subroutine, or related group of subroutines, is described in detail in its own separate topic. The topics are presented in alphabetical order, according to the subroutine's root name.

The system subroutines described here are supplied on every Ultimate system SYS-GEN tape, and are available for use by user-written programs. These subroutines, unless otherwise specified, are meant to be called with a BSL instruction. The subroutine returns control to the calling program via a RTN instruction.

A brief description is given for each subroutine, including a summary of function, inputs, outputs, and elements used, similar to the following:

Input (user specified):

IB	R10	points to the character to be translated

Outputs:

IB	R10	points to the converted character (location unchanged)

Elements used:

None except standard scratch elements

The **Inputs:**, **Outputs:**, and **Elements used**: headings describe the functional elements (that is, the standard PSYM symbols) used by the routine. The letter following an element name describes its symbol type, as described in Chapter 3. If the element is a register, the register number is also given.

The **Input:** elements for many routines are divided into two sections: **user specified** and **system specified**.

User specified elements are those that the programmer sets up explicitly before calling the routine. For example, when calling the routine to get a number of contiguous frames (GETBLK), the programmer must obviously specify this number as a parameter.

System specified elements are those that have been implicitly set up by the system some time prior to the call. For example, when calling the

routine to read a line from the terminal (**READLIN**), the buffer location where the data are to be stored is a system standard, and does not have to be explicitly set up by the programmer.

The following are standard scratch elements, as described in Chapter 3:

bits	SB60, SB61
tallies	T4, T5
double tallies	accumulator (D0, D1), D2
triple tallies	FPX (overlays SYSR0) FPY (overlays SYSR1)
registers	R14, R15
storage registers	SYSR0 (overlays FPX) SYSR1 (overlays FPY) SYSR2

File Control Block Symbols

When a file is opened, the system puts the FID of the file control block into one of several symbols, depending on the instruction:

FCB1

FCB2

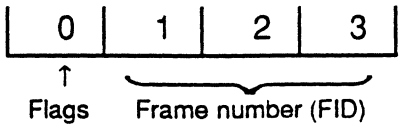
FFCB1

FFCB2

DFCB1

DFCB2

Access information for a file consists of eight bytes (FCB1 and FCB1) used by routines such as GETITM, RETIX, and UPDITM, to uniquely identify the file and to access items within it. The exact format of the 8-byte field may vary depending on operating system revision, and on whether the file is local or remote (that is, accessed via UltiNet). For a local file, however, access information consists mainly of a pointer to the file's control block (FCB), which contains such things as the file's base, modulo, and separation parameters (see subroutine HASH).



Byte	Description
0	flag field; contains specific bits as follows: bit 0, 1 reserved bit 2 remote indicator for file 0 = local file; 1 = remote file bit 3 if bit 2 is set, indicates UltiNet lock status bit 4-7 reserved
1 to 3	if bit 2 in byte 0 is not set, these bytes contain frame number (FID) of FCB

ACONV

The ACONV subroutine translates one character from ASCII to EBCDIC.

Input (user specified):

IB	R10	points to the character to be translated

Output:

IB	R10	points to the converted character (location unchanged)

Elements used:

None except standard scratch elements

ANDIOFLGS

The ANDIOFLGS command sets one or more I/O flag bits to false.

Input (user specified):

B15	B	sets the ?? flag
B14	B	sets the ?? flag
B13	B	sets the ?? flag
B12	B	sets the ?? flag

Output:

T0	T	??

Elements used:

D0

R14

R15

Description

The bits in T0 correspond to the following flag bits (**need the meaning??**):

B15	CCDEL
B14	TITFLG
B13	FRMTFLG
B12	ITABFLG

A value of 1 (one) sets the corresponding bit to true; a value of 0 (zero) maintains the current setting.

ATTOVF

The ATTOVF routine is used to obtain a frame from the overflow space pool and to link it to the frame specified in double tally RECORD.

Input (user specified):

RECORD	D	contains the FID of the frame to which an overflow frame is to be linked.

Output:

OVRFLW	D	contains the FID of the overflow frame if obtained, or zero if no more frames are available.

Elements used:

None except standard scratch elements

Description

The forward link field of the frame specified in RECORD is set to point to the overflow frame obtained, the backward link field of the overflow frame is set to the value of RECORD, and the other link fields of this overflow frame are zeroed.

CONV

The CONV interface is used to call the conversion software (input conversion or output conversion) as a subroutine or to call a user-written subroutine from BASIC or Ultimate RECALL. See Chapter 6, System Software Interfaces, for details.

CRLFPRINT

The CRLFPRINT subroutine is used to execute a carriage return and line feed, and then output a message to the terminal. See PRINT for details.

CVD
CVX

The CVD (convert decimal) and CVX (convert hexadecimal) subroutines convert ASCII numeric character strings to equivalent binary strings.

Instruction Input (user specified):

CVDIB	IB	R10	decimal number to convert
CVDIR	IR	R6	decimal number to convert
CVDIS	IS	R4	decimal number to convert
CVDOS	OS	R5	decimal number to convert
CVDR15	R15	R	decimal number to convert
CVXIB	IB	R10	hexadecimal number to convert
CVXIR	IR	R6	hexadecimal number to convert
CVXIS	IS	R4	hexadecimal number to convert
CVXOS	OS	R5	hexadecimal number to convert
CVXR15	R15	R	hexadecimal number to convert

Output:

FP0	F	contains the converted binary number
CTR1	T	contains the low-order two bytes of FP0, except after CVDR15 and CVXR15 instructions, which do not use CTR1
NUMBIT	B	set if string was terminated by a system delimiter or decimal point; zero if any other character
Register used by input	R	Points to the terminating character (usually system delimiter)

Elements used:

None except standard scratch elements

Description

Both ASCII decimal to binary and ASCII hexadecimal to binary conversions are available. The register used as the string pointer depends on the exact subroutine called (see Inputs above).

The character string to be converted starts at the byte address +1 of the register associated with the called subroutine. The string is terminated when any invalid decimal or hexadecimal character is encountered (usually a system delimiter).

The converted value is stored in the accumulator FP0; the register points to the delimiter on exit.

DATE

The DATE subroutine is used to return the current system date. See the TIME subroutine for details.

DECINHIB

The DECINHIB subroutine is called to decrement the INHIBITH half tally when the user has previously incremented it by one to prevent the BREAK key from calling the debugger.

Input (system specified):

BREAKKEY	B	set if BREAK key was pressed while debugger was inhibited
INHIBITH	H	greater than zero if debugger is currently inhibited
USER	T	contains value 7 if debugger is currently inhibited and DSR has gone false

Output:

BREAKKEY	B	see below
INHIBITH	H	decremented as described below
USER	T	see below

Elements used:

None except standard scratch elements

Description

The debugger is inhibited as long as INHIBITH is non-zero. This prevents the process from entering the debugger and also prevents the process from being logged off if DSR goes false.

The protocol of incrementing and decrementing INHIBITH ensures that several different system programs that require inhibiting of the BREAK key may call one another without fear that INHIBITH may accidentally reach zero.

DECINHIB decrements INHIBITH if its current value is non-zero. If INHIBITH is still non-zero, DECINHIB returns to the calling program with the debugger still inhibited.

If INHIBITH reaches zero, or was zero on entry to the subroutine, DECINHIB first checks USER, then BREAKKEY as follows:

- If USER = 7, indicating DSR has gone false, USER is changed to 11 and the process is sent to WRAPUP, where it is logged off.
- If USER is not 7 and if BREAKKEY is set, indicating BREAK was pressed while the debugger was inhibited, BREAKKEY is zeroed and the system debugger is entered.
- If USER is not 7 and if BREAKKEY is not set, DECINHIB returns to the calling program with the debugger no longer inhibited.

ECONV

The ECONV subroutine translates one character from EBCDIC to ASCII. Characters without ASCII equivalents are returned untranslated.

Input (user specified):

IB	R10	points to the character to be translated

Output:

IB	R10	points to the translated character; (location unchanged)

Elements used:

None except standard scratch elements

GETACBMS

The GETACBMS subroutine retrieves the file access information of the system ACC file.

Inputs (user specified): None

Output:

FCB1	D	contain the file access information for the ACC file, if found; if not, FCB1 is zero
FCB2	D	
RMBIT	B	set if ACC file access parameters were successfully retrieved

Elements used:

SR1 same as GETFILE

Additional elements used by RETIX

Description

GETACBMS sets up FCB1 and FCB2 to allow access to the system ACC file.

GETBUF

The GETBUF subroutine reads data into a buffer pointed to by R14.

Input (user specified):
(??documentation I have says same as for READLIN, except that does not make sense. READLIN moves the data into IB (R10). The description for GETBUF says the buffer pointed to by R14. Who specifies R14? It also says that T0 can be used to specify the number of characters to read.)

Elements used:

None except standard scratch elements

GETFILE

The GETFILE subroutine opens a specified file.

Input (user specified):

IS	R4	points at least one character (any number of blanks) before filename; the filename cannot contain embedded blanks, and must be followed by a blank, a system delimiter, or character specified in SC0
		Note: The filename may be preceded by either DICT or DATA (or neither, which opens the DATA section).
RTNFLG	B	set if GETFILE is to return to the calling program even if the file cannot be opened
DAF1	B	set if file is to be opened only if update access code test does not fail; if not set, file is opened but if the update access code test fails, bit 16 in FCB2 is zero
SC0	C	contains character used to delimit filename

Input (system specified):

BMSBEG	S	standard system buffer where the filename is to be copied; if IS (R4) contains DICT or DATA, they are ignored; only the filename is copied

Output:

FCB1	D	} contain the file access information for the current file, if found; if not, elements are zero
FCB2	D	
RTNFLG	B	set if GETFILE is to return to the calling program even if the file cannot be opened
DAF1	B	set if update access is required; if zero, update access is granted unless the update access code test fails
IS	R4	points to the first character after the file name
BMS	R8	points to the last character of the copied file name

RMBIT	B	set if the file parameters are successfully retrieved
SC2	C	contains a blank

Elements used:

SC1 C

Additional elements used by RETIX

Description

GETFILE uses the file name to set up the access information for a file. The name of the file is specified in a string pointed to by register IS.

If the file cannot be successfully opened, GETFILE exits to WRAPUP, unless the bit RTNFLG is set, in which case the subroutine returns to the calling program.

DICTOPEN, GETFILE, and OPENDD are the only approved methods of opening a disk file. They perform access code checking, and flag the file as being accessible for read-only, or for read-and-update, as appropriate.

If an error occurs and RTNFLG is not set (it is 0), the subroutine exits to the following entry points, depending on the error:

- MD99 with message 200 if the input string is null (blank to a SM)
- MD995 with message 201 if the string does not refer to a file (item not found or in incorrect format)
- MD995 with message 210 if the access code test fails
- MD99 with message 13 if the data section of a file is not found (no data pointer, or in incorrect format)

GETIOFLGS

The GETIOFLGS command get the current I/O flag bit settings.

Input (user specified):

B15	B	sets the ?? flag
B14	B	sets the ?? flag
B13	B	sets the ?? flag
B12	B	sets the ?? flag

Output:

T0	T	??

Elements used:

D0

R14

R15

Description

The bits in T0 correspond to the following flag bits (need the meaning):

B15	CCDEL
B14	TITFLG
B13	FRMTFLG
B12	ITABFLG

A value of 1 (one) sets the corresponding bit to true; a value of 0 (zero) maintains the current setting.

GETITM

The GETITM subroutine sequentially retrieves all items in a file. It is called repetitively to obtain items one at a time until all items have been retrieved. The order in which the items are returned is the same as the storage sequence.

Input (user specified):

DAF7	B	initial entry flag; must be zeroed on the first call to GETITM
DAF1	B	if set, the update option is in effect
FCB1	D	contain the file access information for the current file; required on first entry only
FCB2	D	

Input (system specified):

BMSBEG	S	standard system buffer where the item-id of the item retrieved on each call is copied
OVRFLCTR	D	meaningful only if DAF1 is set; if non-zero, the value is used as the starting FID of the overflow space table where the list of item-ids is stored; if zero, GETOVF is called to obtain space for the table

Output:

DOCCFLG	B	set if catalog item type or extended item type
IR	R6	points to the first AM of the item
R14	R	points one prior to the item count field
RECORD	D	contains beginning FID of the group to which the item-id hashes (set by HASH)
RMBIT	B	1 if item found; 0 if item not found
SIZE	T	item size, or 0 for extended item type
SR0	S	points one before the count field of the retrieved item
SR4	S	points to the last AM of the item

XMODE T 0

Note: The outputs are the same as for RETIX.

Elements used: (used for accessing file data)

NNCF	H	
FRMN	D	
FRMP	D	
NPCF	H	
OVRFLW	D	used by GETOVF if DAF1 is set and OVRFLCTR is initially zero

The following elements *should not be altered* by any other routine while items are being retrieved by GETITM:

DAF1	B	see Inputs:
DAF7	B	
SFCB1	D	contains the access information for the FID of the current group being processed
SFCB2	D	contains the access information for the number of groups left to be processed
FFCB1	D	contain the original (saved) access information for the file
FFCB2	D	
NXTITM	S	points one before the next item-id in the pre-stored table if DAF1 is set, otherwise points to the SM after the item previously returned
OVRFLCTR	D	contains the starting FID of the overflow space table if DAF1 is set; otherwise unchanged

Description

If the items that are retrieved are to be updated by the calling routine (using routine UPDITM), this should be flagged to GETITM by setting bit DAF1. GETITM then performs a two-stage retrieval process by first storing all item-ids (per group) in a table, then using this table to actually retrieve the items on each call. This is necessary because if the calling routine updates an item, the data within the updated group shift around; GETITM cannot simply maintain a pointer to the next item in the group, as it does if the "update" option is not flagged.

GETITM *must* be called the first time with the flag DAF7 set to zero, so that it can set up its internal conditions . It sets up and maintains certain pointers which should not be altered by calling routines until all the items in the file have been retrieved (or DAF7 is zeroed again).

GETOVF
GETBLK

The GETOVF and GETBLK routines obtain overflow frames from the overflow space pool maintained by the system. GETOVF is used to obtain a single frame. GETBLK is used to obtain a block of contiguous space.

Input (user specified):

D0	D	contains the number of frames needed (block size); for GETBLK only

Output:

OVRFLW	D	if the needed space is obtained, this element contains the FID of the frame returned (for GETOVF) or the FID of the first frame in the block returned (for GETBLK); if the space is unavailable, OVRFLW=0

Elements used:

None except standard scratch elements

Description

Note that the link fields of the frames obtained by a call to GETBLK are not reset or initialized in any way; this should be done by the caller, using subroutine LINK.

GETOVF zeros the link fields of the single frame it returns. These routines cannot be interrupted until processing is complete.

For information on related commands, see the descriptions of the following:

RELBLK
RELCHN
RELOVF

GLOCK
GUNLOCK
GUNLOCK.LINE

The GLOCK, GUNLOCK, and GUNLOCK.LINE routines are used to ensure that disk files are not updated by more than one process at a time, and are used primarily by the subroutine UPDITM.

Input (user specified):

RECORD	D	contains the beginning FID of the group to be locked (typically set by RETIX or RETIXU)

Outputs: None

Elements used:

CH9	C
R2;C0	C
CTR1	T

Plus standard scratch elements

Description

GLOCK sets a lock on a specified group within a file, preventing other processes from locking the group. If GLOCK is called and the the group is already locked by another process, the second process pauses until the lock is unlocked.

GUNLOCK frees the lock on a group (if present, and set by the calling process), allowing another process to lock it.

GUNLOCK.LINE frees all locks set by a process.

The subroutine UPDITM calls GLOCK at the beginning of UPDITM's execution, before writing an item to a file. GUNLOCK is called at the end of the UPDITM subroutine. GLOCK is also called by RETIXU, which retrieves a disk file item and leaves the group containing the item locked.

HASH

The HASH routine computes the starting FID of the group in which an item in a file would be stored, given the item-id and the access information of the file.

Input (user specified):

BMSBEG	S	Points one byte before the beginning of the item-id, which must be terminated by an attribute mark.
FCB1	D	} contain the access information for the file
FCB2	D	}

Output:

RECORD	D	contains the frame number to which the item-id hashes

Elements used:

None except standard scratch elements

HSISOS

The HSISOS routine initializes the registers for the HS, IS, and OS workspaces. It does not link frames in the work spaces.

Inputs: None

Output:

HS	R3	points to the beginning of the HS workspace (PCB+10)
HSBEG	S	
HSEND	S	
IS	R4	points to the beginning of the IS workspace (PCB+16)
ISBEG	S	
ISEND	S	points to the last data byte in the primary IS workspace (6*ID.DATA.SIZE bytes past ISBEG).
OS	R5	points to the beginning of the OS workspace (PCB+22)
OSBEG	S	
OSEND	S	points to the last data byte in the primary OS work space (6*ID.DATA.SIZE bytes past OSBEG).

Elements used:

None except standard scratch elements

INITRTN

The INITRTN subroutine initializes the return stack.

Inputs:

RCSWA-4	T	contains return address of the calling routine

Output:

none

Elements used:

QCBSR	S	used to save and restore R15

LINESUB

The LINESUB subroutine returns the line number (PIB number) of the calling process.

Inputs: None

Output:

D0	D	contains the line number associated with the process

Elements used:

None except standard scratch elements

LINK

The LINK subroutine initializes the links of a set of contiguous frames (the set may be only one frame).

Input (user specified):

RECORD	D	contains the starting FID of a set of contiguous frames
NNCF	H	contains one less than the number of frames in the set

Output:

R14	R	points one byte before the first data byte of the first frame
R15	R	points to the last byte of the last frame

Elements used:

FRMN	D
FRMP	D
NPCF	H

Description

Frames are linked contiguously backwards and forwards.

The subroutine is called with RECORD containing the starting frame number of the set, and NNCF the number of frames less one in the set (that is, NNCF contains the number of next contiguous frames).

MARKRTN

The MARKRTN subroutine divides the extended return stack into logical stacks.

Inputs:

RSCWA-4	T	return address of the calling routine

Output:

RSCWA	T	set to X'190'

Elements used:

QCBSR	used to save and restore R15

Description

All entries in the return stack in the PCB from entry three to the entry at RSCWA-4 are copied to the extended stack. A null address is added to the extended stack to mark the end of the logical stack. The return address of the routine that called MARKRTN is moved to X'18C'.

MBDSUB
MBDNSUB
MBDSUBX
MBDNSUBX

The MBD-type subroutines convert a binary number to the equivalent string of decimal ASCII characters.

Input (user specified):

D0	D	contains the number to be converted (used by MBDSUB and MBDNSUB)
FP0	F	contains the number to be converted (used by MBDSUBX and MBDNSUBX only)
T4	T	contains the minimum string length (used by MBDNSUB and MBDNSUBX only); string may exceed this length; leading zeros or blanks are added, if necessary, to ensure that the string is at least this length
BKBIT	B	set if leading blanks are required as fill; not set if zeros required as fill (used by MBDNSUB and MBDNSUBX only)
R15	R	points one prior to the area where the converted string is to be stored; the area must be at least 18 bytes in length for MBDSUBX and MBDNSUBX; MBDSUB and MBDNSUB require at most 11 bytes

Output:

BKBIT	B	=0
R15	R	Points to the last converted character

Elements used:

None except standard scratch elements

Description

MBDSUB and MBDSUBX return only as many characters as are needed to represent the number, whereas MBDNSUB and MBDNSUBX always return a specified minimum number of characters (padding with leading zeros or blanks whenever necessary).

A minus precedes the numeric string if the number to be converted is negative.

These subroutines are implicitly called by the assembler instruction MBD (move binary to decimal).

NEWPAGE

The NEWPAGE subroutine is used to skip to a new page on the terminal or printer and print a heading.

Input (user specified):

None

Input (system specified):

OB	R11	first set equal to OBBEG by this routine
OBBEG	S	used to store data for terminal or printer output
LISTFLAG	B	if set, all output to the terminal is suppressed; set and reset by the TCL P command and by the debug P command
LPBIT	B	if set, output is routed to the spooler (Note: routine SETLPTR should be used to set this bit so printer characteristics are set up correctly)
LFDLY	T	lower byte contains the number of fill characters to be output after a CR/LF; set by the TCL TERM command
OTABFLG	B	output tabs in effect if set (by the TCL TABS command)
PAGINATE	B	if set, pagination and page headings are invoked; usually set by the PRNTHDR routine in conjunction with page heading and/or footing

Output:

OB	R11	=OBBEG

Note: The output is the same as for WRTLIN.

Elements used:

BMS Used if LPBIT set

ATTOVF Used if LPBIT set

Plus standard scratch elements

Description

No action is performed by this subroutine if either the bit PAGINATE or the tally PAGSIZE is zero.

See PRNTHDR for more information on page headings and footings.

NEXTIR
NEXTOVF

The NEXTIR subroutine obtains the forward linked frame of the frame to which register IR (R6) currently points. The NEXTOVF subroutine is used to attach additional overflow space.

Input (user specified):

IR	R6	points into the frame whose forward-linked frame is to be obtained (displacement unimportant)

Output:

IR	R6	points to the first data byte of the forward linked frame
RECORD	D	contains the FID of the frame to which IR (R6) points
NNCF	H	As set by RDLINK for the FID in RECORD
FRMN	D	
FRMP	D	
NPCF	H	
OVRFLW	D	=RECORD if ATTOVF called, otherwise unchanged

Elements used:

Same as elements used by ATTOVF if a frame is obtained from the overflow space pool.

Description

For NEXTIR, if the forward link is zero, the routine attempts to obtain an available frame from the system overflow space pool and link it up appropriately (see ATTOVF subroutine). In addition, if a frame is obtained, the IR register is set up before return, using routine RDREC.

NEXTOVF may be used in a special way to handle end-of-linked-frame conditions automatically when using register IR with single- or multiple-

byte move or scan instructions (for example, MII, MCI, MIID, MIITD, SIT, and SID.). Tally XMODE should be set to the mode-id of NEXTOVF before the instruction is executed by using the following instruction:

```
MOV NEXTOVF,XMODE
```

If the instruction causes IR to reach an end-of-linked-frame condition (forward link zero), the system generates a subroutine call to NEXTOVF, which attempts to obtain and link up an available frame, then resumes execution of the interrupted instruction.

Note that the INC r,t (increment register by tally) instruction cannot be guaranteed to work in this manner.

OPENDD

The OPENDD subroutine opens both the data and dictionary portions of the file.

Input (user specified):

IS	R4	points at least one character (any number of blanks) before filename; the filename cannot contain embedded blanks, and must be followed by a blank, a system delimiter, or character specified in SC0
		Note: The filename may be preceded by DICT, in which case, only the dictionary portion of the file is opened.
RTNFLG	B	set if OPENDD is to return to the calling program even if the file cannot be opened
DAF1	B	set if file is to be opened only if update access code test does not fail; if not set, file is opened but if the update access code test fails, bit 16 in FCB2 is zero
SC0	C	contains character used to delimit filename

Input (system specified):

BMSBEG	S	standard system buffer where the filename is to be copied; if IS (R4) contains DICT or DATA, they are ignored; only the filename is copied

Output:

FCB1	D	contain the file access information for the current file, if found; if not, elements are zero
FCB2	D	
FFCB1	D	contain the file access information for the data section of the current file, if found; if not, elements are zeroed; FFCB1 and FFCB2 = FCB1 and FCB2
FFCB2	D	

DFCB1 DFCB2	D D	contain accesss information for the file dictionary, if found; if not, elements are zeroed; if IS specifies DICT, DFCB1 and DFCB2 = FCB1 and FCB2
IS	R4	points to the first character after the file name
BMS	R8	points to the last character of the copied file name
RMBIT	B	set if the file parameters are successfully retrieved; if not, it is zeroed
DAF8	B	set if only dictionary section opened; zero if data section, or both dictionary and data sections were opened
CRLFBIT	B	set if both dictionary and data sections were opened; zero if only data section was opened
NETERR	B	set if network error occurred while trying to open remote file
T0	T	contains ERRMSG number if NETERR set
SC2	C	contains a blank

Elements used:

SC1	C

Additional elements used by RETIX

Description

OPENDD uses the filename to set up the access information for a both the dictionary and data sections of a file. The file name is specified in a string pointed to by register IS.

If the file cannot be successfully opened, OPENDD exits to WRAPUP, unless the bit RTNFLG is set, in which case the subroutine returns to the calling program.

DICTOPEN, GETFILE, and OPENDD are the only approved methods of opening a disk file. They perform access code checking, and flag the file as being accessible for read-only, or for read-and-update, as appropriate.

If an error occurs and RTNFLG is not set (it is 0), the subroutine exits to the following entry points, depending on the error:

- MD99 with message 200 if the input string is null (blank to a SM)
- MD995 with message 201 if the string does not refer to a file (item not found or in incorrect format)
- MD995 with message 210 if the access code test fails
- MD99 with message 13 if the data section of a file is not found (no data pointer, or in incorrect format)

ORIOFLGS

The ORIOFLGS command sets one or more I/O flag bits to true.

Input (user specified):

B15	B	sets the ?? flag
B14	B	sets the ?? flag
B13	B	sets the ?? flag
B12	B	sets the ?? flag

Output:

T0	T	??

Elements used:

D0

R14

R15

Description

The bits in T0 correspond to the following flag bits (need the meaning):

B15	CCDEL
B14	TITFLG
B13	FRMTFLG
B12	ITABFLG

A value of 1 (one) sets the corresponding bit to true; a value of 0 (zero) maintains the current setting.

PCRLF

The PCRLF subroutine prints a carriage return and line feed on the terminal only, along with the specified number of delay characters (X'00'), as set by the TCL TERM command.

Inputs:

None

Output:

None

Elements used:

None except standard scratch elements

Description

PCRLF does not use the pagination, headings, footings, and other elements of print placement, nor does it update them. If it is necessary to ensure correct use of these elements, use the WRTLIN subroutine.

PERIPHREAD1
PERIPHREAD2

The PERIPHREAD subroutines are used to read a string of bytes from another line's asynchronous channel, on configurations which support this feature. They are analogous to the READLIN subroutine, which reads to the current line's channel only.

Input (user specified):

IBSIZE	T	maximum number of bytes to be input
T0	T	contains the number of the affected channel
SC0	C	contains the delimiter characters on which to stop the input (used by PERIPHREAD1 only)
SC1	C	
SC2	C	

Input (system specified):

IBBEG	S	standard system input buffer pointer

Output:

Same as READLIN

Elements used:

ABIT	B
CTR0	T

Plus standard scratch elements

Description

The line number of the affected channel should be loaded into T0. The affected line must have been previously set to a trapped condition by the TCL :TRAP command. If the affected line is not trapped, WRAPUP is entered with error message 540.

PERIPHREAD1 inputs data until a byte matching that in SC0, SC1, or SC2 is found. Either subroutine returns when the number of bytes specified in IBSIZE is read. The bytes that are input are stored starting at the location one past IBBEG (just as in READLIN).

PERIPHREAD2 inputs data just as READLIN does; control is returned on detecting a carriage return or line feed.

For information on writing to another line's asynchronous channel, see the PERIPHWRITE subroutine.

PERIPHWRITE

The PERIPHWRITE subroutine is used to write a string of bytes to another line's asynchronous channel, on configurations which support this feature. It is analogous to the WRTLIN subroutine, which writes to the current line's channel only.

Input (user specified):

OB	R11	pPoints to the last byte to be output
T0	T	contains the number of the affected channel

Input (system specified):

OBBEG	S	standard output buffer pointer

Output:

OB	R11	=OBBEG

Elements used:

ABIT	B
CTR0	T

Plus standard scratch elements

Description

The line number of the affected channel should be loaded into T0. The affected line must have been previously set to a trapped condition by the TCL :TRAP command. If the affected line is not trapped, WRAPUP is entered with error message 540.

PERIPHWRITE outputs data just as WRTLIN does; OBBEG points one byte before the beginning of the data, and OB points to the last byte of data.

For information on reading from another line's asynchronous channel, see the PERIPHREAD subroutines.

POPRTN

The POPRTN subroutine pops the top entry off the return stack.

Inputs:

RSCWA-4	T	return address of the calling routine

Output:

none

Elements used:

QCBSR	used to save and restore R15

PRINT
CRLFPRINT

The PRINT and CRLFPRINT subroutines send a message to the terminal from textual data in the calling program. CRLFPRINT precedes the text with a carriage return and line feed.

Input (user specified):

Message text *must* follow BSL instruction

Output:

None

Elements used:

None except standard scratch elements

Description

The message sent is a string of characters assembled immediately following the subroutine call in the calling program. The string must be terminated by one of the three delimiters SM, AM, or SVM. Control is returned to the instruction at the location immediately following the terminal delimiter. The delimiter VM inserts a carriage return/linefeed, but does not terminate the message.

The system delimiters have the following meaning:

SM (X'FF')	end of message; CR/LF printed, and return to calling program
AM (X'FE')	same as SM
VM (X'FD')	CR/LF printed, and message processing continues to next character
SVM (X'FC')	end of message; return without printing CR/LF

Caution: *These routines are not consistent with conventions regarding pagination . They should therefore be used only for error messages and short terminal prompts.*

```
BSL     PRINT                   Call to subroutine
TEXT    C'Hello',X'FDFF'        Message as a literal in object code
CMNT    *                       Note terminating SM (X'FF')
```

The above would print the message Hello, followed by a blank line.

PRNTHDR

The PRNTHDR subroutine is an entry point into the system routine for pagination and heading control of output (also used by WRTLIN and WRITOB when pagination is specified).

Input (user specified):

PAGHEAD	S	points one before the start of the page heading; if the FID of PAGHEAD is zero (initial condition at TCL), there is no heading defined
PAGFOOT	S	points one before the start of the page footing; if the FID of PAGFOOT is zero (initial condition at TCL), there is no footing defined

Output:

OB	R11	=OBBEG

Note: The output is the same as for WRTLIN.

Elements used:

BMS	used if LPBIT set
ATTOVF	used if LPBIT set

Plus standard scratch elements

Description

PRNTHDR must be called once to initialize the bits and counters needed to start pagination; it also prints the heading (if any) for the first page. PRNTHDR should not be called again unless starting a new pagination sequence; to skip to a new page at any time, NEWPAGE should be called.

A page heading or footing, if present, must be stored in a buffer defined by storage register PAGHEAD or PAGFOOT. The heading or footing message is a string of data terminated by a SM; system delimiters in the message invoke special processing as follows:

SM (X'FF')	terminates the heading or footing line with a carriage return and line feed
VM (X'FD')	prints one line of the heading or footing and starts a new line

SVM (X'FC') inserts data from various sources into the heading or footing, depending on the characters following the SVM; valid characters are as follows:

A	inserts data from AFBEG+1 through a system delimiter
D	inserts current date
F	inserts data from ISBEG+3 through a system delimiter
I	inserts data from BMSBEG+1 through a system delimiter
P	inserts page number right-justified in a field of four blanks
PN	inserts page number left-justified
T	inserts current time and date

Carriage returns, line feeds, and form feeds should not be included in heading or footing messages, or the automatic pagination will not work properly. A convenient buffer for storing headings and footings is the HS (R3).

This subroutine uses WRTLIN to print each heading or footing line as it is formatted; therefore the OB buffer is considered scratch and is destroyed.

RDLINK
WTLINK

The RDLINK and WTLINK subroutines read or write the link fields from or to a frame, to or from the tallies NNCF, FRMN, FRMP, and NPCF.

Input (user specified):

RECORD	D	Contains the FID of the frame whose links are to read or written.

Inputs (user specified for WTLINK only):

NNCF	H	number of next contiguous frames
FRMN	D	forward link
FRMP	D	backward link field
NPCF	H	number of previous contiguous frames

Outputs (from RDLINK only):

NNCF	H	number of next contiguous frames
FRMN	D	forward link
FRMP	D	backward link field
NPCF	H	number of previous contiguous frames

Elements used:

None except standard scratch elements

RDREC

The RDREC subroutine is used to set up the IR(R6) register to the beginning of the frame defined by the double tally RECORD.

Input (user specified):

RECORD	D	Contains the FID required

Output:

IR	R6	points to one before the first data byte of frame (assuming a linked frame)
R15	R	points to link portion of frame

Plus the link field outputs from RDLINK

Elements used:

None except standard scratch elements

Description

The subroutine assumes the frame has the linked format, and therefore IR is set pointing to the byte prior to the first data byte of the frame.

Additionally the subroutine RDLINK is entered to set up R15 pointing to the link portion of the frame, and to set up the link elements NNCF, NPCF, FRMN, and FRMP.

READ@IB
READX@IB

The READ@IB subroutine inputs one character at the current position of R10 and echoes the character. The READX@IB subroutine inputs one character at the current position of R10 and does not echo the character.

Input (user specified):

R10	R	

Output:

R10	R	unchanged
R14	R	

Elements used:

None except standard scratch elements

Description

The next character from the asynchronous channel input buffer replaces the byte addressed by the register. If the input buffer is empty, the process is suspended until a character is received from the asynchronous channel. Characters transmitted by the channel are automatically queued in the terminal input buffer for the process, until some configuration-dependent maximum number of characters is received. If this condition occurs, no further data characters are accepted from the channel, which will output a bell character (X'07') for each attempted input character until the condition is cleared.

For the READ@IB subroutine, control characters (X'00' through X'1F') are never echoed.

READLIN
READLINX
READIB

READLIN, READLINX, and READIB are the standard terminal input routines.

Input (user specified):

CCDEL	B	if set, control characters are deleted; this bit is normally zero
FRMTFLG	B	if set, <CTRL-X> sends a backspace instead of a CR/LF, to preserve screen format; this bit is normally zero
PRMPC	C	terminal prompt character

Input (system specified):

IBBEG	S	standard system buffer pointer; points one before where input is to be stored; the buffer is normally two bytes greater than the value in IBSIZE
IBSIZE	T	contains the maximum number of characters accepted in an input line; normally fixed at 465
LFDLY	T	contains (in the low-order byte) the number of idle characters to be output after a CR/LF; set by the TCL TERM command
BSPCH	C	contains the character to be echoed to the terminal when the backspace key is typed; is zero if no character is to be echoed; set by the TCL TERM command
STKFLG	B	if set, READLIN and READIB test for stacked input; terminal input will not be requested until stacked input is exhausted; set by the PROC processor, or the BASIC DATA statement
STKBEG	S	points to the next stacked input line; lines are delimited by AMs, with an SM indicating the end of the stack
ITABFLG	B	set for input tab stops by the TCL TABS command

Output:

IB	R10	=IBBEG
IBEND	S	points to a SM one byte past the end of input data (overwrites the CR or LF)
STKFLG	B	zeroed if the end of stacked input was reached; not changed if initially zero
STKBEG	S	points to the next line of stacked input (or end of stack) if stacked input is being processed

Elements used:

None except standard scratch elements

Description

Storage register IBBEG points to the standard buffer area where the routine will input the data. Input continues to this area until either a carriage return or line feed is encountered, or until the number of characters equal to the count stored in IBSIZE have been input. The carriage return or line feed terminating the input line is overwritten with a segment mark (SM), and storage register IBEND points to this character on return. If the input is terminated because the maximum number of characters has been input, a SM is added at the end of the line.

If READLIN or READLINX is used, a trailing CR/LF sequence is transmitted to the terminal; if READIB is used, it is not.

READLIN and READIB also provide the facility for taking input from the stack generated by a PROC (STON command) or by BASIC (DATA statement) instead of directly from the terminal. Such input lines are returned to requesting processors as if they originated at the terminal.

If a stacked input line exceeds IBSIZE, the line is truncated at IBSIZE; the remainder of the line is lost.

Terminal input resumes when the stacked input is exhausted. READLINX does not test for stacked input.

Note: READLIN does not recognize the TCL line continuation line character, which is entered as a <CTRL-_>, but may display as a back-arrow (←) on some terminals. Only the TCL software recognizes these line continuation characters.

All three routines perform terminal editing as follows:

<CTRL-H>	backspace input; echo a backspace-space-backspace unless BSPCH = 0.
character in BSPCH	as above.
<CTRL-W>	backspace word to last non-numeric, non-alpha.
<CTRL-X>	cancel line; echo CR/LF or backspaces (see FRMTFLG).
CR or LF	terminate input and return control.

READLIN and READIB also perform input tabulation as specified by the TCL TABS command, when input is from the terminal. If a tab character (X'09') is input, it is replaced by the appropriate number of blanks required to space to the next tab stop.

RELBLK
RELCHN
RELOVF

The RELBLK, RELCHN, and RELOVF subroutines are used to release frames to the overflow space pool. RELOVF is used to release a single frame, RELBLK is used to release a block of contiguous frames, and RELCHN is used to release a chain of linked frames (which may or may not be contiguous).

Input (user specified):

OVRFLW	D	contains the FID of the frame to be released (for RELOVF), or the first FID of the block or chain to be released (for RELBLK and RELCHN, respectively)
D0	D	contains the number of frames (block size) to be released, for RELBLK only

Output:

none

Elements used:

None except standard scratch elements

Description

A call to RELCHN specifies the first FID of a linked set of frames; the routine releases all frames in the chain until a zero forward link is encountered.

For information on getting frames from overflow, see the GETOVF and GETBLK subroutines

RESETTERM

The RESETTERM subroutine is used to initialize terminal and printer characteristics. RESETTERM is called from WRAPUP before reentering TCL.

Input (system specified):

OBSIZE	T	contains the size of the output (OB) buffer
OBBEG	S	points to the start of the OB buffer

Output:

TOBSIZE	T	initialized to default values, as set up by the TCL TERM command
TPAGSIZE	T	
POBSIZE	T	
PPAGSIZE	T	
PAGSKIP	T	
LFDLY	T	
BSPCH	C	
CCDEL	B	=0
FRMTFLG	B	
STKFLG	B	
PAGINATE	B	
NOBLNK	B	
LPBIT	B	
TPAGNUM	T	
TLINCTR	T	
PPAGNUM	T	
PLINCTR	T	
PAGNUM	T	
LINCTR	T	
PAGHEAD	S	contain zero in the frame field
PAGFOOT	S	

OB	R11	=OBBEG
OBSIZE	T	=TOBSIZE
OBEND	S	points to OBBEG + OBSIZE

The area from the address pointed to by OBBEG to that pointed to by OBEND is filled with blanks.

Elements used:

None except standard scratch elements

RETIX
RETIXU

The RETIX and RETIXU subroutines are the entry points to the standard system routine for retrieving an item from a file. The item-id is explicitly specified to the routine, as is the file access information.

Input (user specified):

BMSBEG	S	points one byte before the item-id, which must be terminated with an AM
FCB1	D	contain the file access information of the file to be searched
FCB2	D	

Output:

BMS	R8	point to the last character of the item-id
BMSEND	S	
NNCF	H	contain the link fields of the frame specified in RECORD; set by RDREC
FRMN	D	
FRMP	D	
NPCF	H	
RECORD	D	contains the beginning FID of the group to which the item-id hashes (set by HASH)
RMBIT	B	1 if item found; 0 if item not found
XMODE	T	0

The following have meaning only if the item was found:

SIZE	T	=item size, or 0 for extended item type (see D0 below)
R14	R	points one prior to the item count field
IR	R6	points to the first AM of the item
SR4	S	points to the last AM of the item
DOCCFLG	B	=1 if D, CC, or CL pointer; otherwise = 0
D0	D	extended item size if SIZE = 0

Elements used:

None except standard scratch elements

Description

The file access information in FCB1 and FCB2 is set up by calling GETFILE or OPENDD to open the file (see these subroutines for details).

The RETIX routine performs a hashing algorithm to determine the group (see HASH subroutine). The group is then searched sequentially for a matching item-id. If the routine finds a match, it returns the item size (from the item count field) and pointers to the beginning and end of the item.

If RETIXU is used, the group is locked to prevent other programs from changing the data; the group is automatically unlocked when the item is later written back to the file (see UPDITM), or the user may explicitly unlock the group by calling the GUNLOCK or GUNLOCK.LINE routine. The group is locked whether or not the item is found.

The item-id is specified in the system-standard buffer defined by storage register BMSBEG; it must be terminated with an AM.

If an error condition occurs, such as bad data in the group, or premature end of linked frames, or non-hexadecimal character encountered in the count field, this message is returned:

```
GFE handler invoked - record/GFE x/f.d
```

x starting FID of the group to which the item hashes

f.d approximate frame and displacement where the error was detected.

If this happens, RETIX and RETIXU return with an item not found condition. No data is destroyed, and the group format error remains.

RTNMARK

The RTNMARK subroutine moves a logical stack from the extended return stack into the PCB.

Inputs:

RSCWA-4	T	return address of the calling routine

Output:

RSCWA	T	set to X'190'

Elements used:

QCBSR	used to save and restore R15

Description

The return address of the routine that called RTNMARK is moved to X'18C'. All entries in the extended return stack from entry ?? to the first null entry are copied to the stack in the PCB.

SETLPTR
SETTERM

The SETLPTR and SETTERM subroutines are used to set output characteristics such as line width and page depth to the previously specified values for either the terminal or the printer. In addition, the current line number and page number are saved.

Input (user specified):

None

Input (system specified):

LINCTR	T	contains the current line number
PAGNUM	T	contains the current page number

Inputs (system specified) for SETLPTR only:

PPAGSIZE	T	contains the number of printable lines per page for the printer
POBSIZE	T	contains the size of the output (OB) buffer for the printer
PLINCTR	T	contains the current line number for the printer
PPAGNUM	T	contains the current page number for the printer

Inputs (system specified) for SETTERM only:

TPAGSIZE	T	contains the number of printable lines per page for the terminal
TOBSIZE	T	contains the size of the output (OB) buffer for the terminal
TLINCTR	T	contains the current line number for the terminal
TPAGNUM	T	contains the current page number for the terminal

Output:

LPBIT	B	set by SETLPTR; reset by SETTERM

PAGSIZE	T	set to the appropriate characteristics for terminal or printer output
OBSIZE	T	
LINCTR	T	
PAGNUM	T	
PLINCTR	T	=LINCTR; set by SETTERM
TLINCTR	T	=LINCTR; set by SETLPTR
OBEND	S	=OBBEG+OBSIZE

The area from the location addressed by OBBEG to that pointed to by OBEND is filled with blanks.

Elements used:

None except standard scratch elements

Description

It can be useful to save the current line and page number when switching from terminal to printer output, and then switching back. Pagination continues automatically from the previous values.

SLEEP
SLEEPSUB

The SLEEP and SLEEPSUB subroutines cause the calling process to go into an inactive state for a specified amount of time. If SLEEPSUB is used, either the amount of time to sleep or the time at which to wake up can be specified.

Input (user specified):

D0	D	for SLEEP, contains the time to wake up (number of milliseconds past midnight); for SLEEPSUB, contains the number of seconds to sleep or the time to wake up, depending on RMBIT
RMBIT	B	for SLEEPSUB, set if D0 contains the number of seconds to sleep, or zero if it contains the time to wake up (number of milliseconds past midnight)

Output:

None

Elements used:

None except standard scratch elements

SORT

The SORT subroutine is used to sort an arbitrarily long string of keys in ascending sequence only. The calling program must complement the keys if a descending sort is required.

Input (user specified):

SR1	S	points to the SM preceding the first key
SR2	S	points to the SM terminating the last key
SR3	S	points to the beginning of the second buffer

Output:

SR1	S	points before the first sorted key (the exact offset varies from case to case); the calling routine should scan from one byte past this point for a non-SB character; the end of the sorted keys (separated by SBs) is marked by a SM

Elements used:

SB1	B
SC2	C
XMODE	T
IS	R4
OS	R5
BMS	R8
TS	R13
CS	R12
S1 thru S9	S

Elements used by ATTOVF

BMS and AF (R9) workspaces

Description

The sort keys are separated by SMs (X'FF') when presented to SORT; they are returned separated by SBs (X'FB'). Any character, including system delimiters other than the SM, SB, X'F0', and X'F1', which have special meanings, may be present within the keys.

For descending sort sequencing (on non-numeric data), the individual characters of the sort key must have been one's complemented by the calling routine.

SORT performs a left-to-right character comparison, except when either of the character X'F0' or X'F1' is present:

X'F0' indicates the start of a numeric string; the string is terminated by a SVM.

X'F1' indicates the start of a numeric string that is to be compared negatively; the string is terminated by a SVM (for example, this may be set up by the Ultimate RECALL BY-DSND connective).

The purpose of this is to allow the sort keys to contain mixed left-justified (non-numeric) data and numeric (right-justified comparison) data.

For example, to sort the key ABC/9Y before the key ABC/100X, the keys should be presented to the SORT subroutine as follows:

```
ABC/[F0]100[FC]X
ABC/[F0]9[FC]Y
```

This results in sequencing ABC/9Y before ABC/100X.

The SORT subroutine uses a six-way polyphase sort-merge sorting algorithm. The original unsorted key string may grow by a factor of 10%, and a separate buffer is required for the sorted key string, which is about the same length as the unsorted key string. The growth space is contiguous to the end of the original key string; the second buffer may be specified anywhere.

SORT automatically obtains and links overflow space whenever needed. Due to this, one can follow standard system convention and build the

entire unsorted string in an overflow table with OVRFLCTR containing the beginning FID; the setup is then:

```
Start of       End of          Growth       Start of
unsorted keys  unsorted keys   space        second buffer

SM<-------/-   -/---------->SM<----------><----------/-
```

The user creates the unsorted key list and the 10% growth space. The second buffer pointer then is merely set at the end of the growth space, and SORT is allowed to obtain additional space as required.

Alternatively, the entire set of buffers may be in the IS or OS workspace if they are large enough.

SYSTEM-CURSOR

The SYSTEM-CURSOR subroutine can be used for either terminal cursor control or printer control.

Input (user specified):

T0	T	if negative, contains either a terminal control code or a printer control code; see Table 5-1 for a list of terminal control codes; see Table 5-2 for a list of printer control codes. If non-negative, it must be equal to the value of CTR11 (column position)
CTR10	T	when T0 is non-negative, specifies the row number for positioning the cursor; rows are numbered from top to bottom, starting with zero; a value less than zero indicates no row specification
CTR11	T	when T0 is non-negative, specifies the column number for positioning the cursor; columns are numbered from left to right, starting with zero; a value less than zero indicates no column specification
R15	R	points one byte before the output area to be used

Input (system specified):

TERMTYPE	C	specifies terminal type

Output:

R15	R	points to the last byte of data generated; or unchanged if the specified function is not defined for this terminal type

Elements used:

None except standard scratch elements

Description

The SYSTEM-CURSOR subroutine examines the code in T0 to determine whether terminal or printer control is indicated. The codes for terminal control are in the range -1 to -99. The codes for printer control are in the range -101 and up.

For cursor control, the SYSTEM-CURSOR subroutine generates the string of characters necessary to position the cursor or to produce a visual effect on a terminal. The terminal type is specified by the code character in TERMTYPE. TERMTYPE is initialized at logon, but may be changed by commands such as TERM and TERMINAL. The following are valid Ultimate-defined terminal type codes:

A	ADDS Regent 40 (25 line CRT)
B	DEC VT241
C	ADDS Viewpoint Color
D	DEC VT100
E	DEC VT200 8-bit mode
F	IBM 3270
G	IBM 3101
H	Honeywell VIP-7200
L	Liberty Freedom-200
P	IBM Personal Computer
R	ADDS Regent 25
S	Wyse WY-60, Native mode
U	Ultimate CRT (Volker-Craig)
V	Ultimate VDT (ADDS Viewpoint)
W	Wyse WY-50 or Ultimate ULT-50 Enhanced Viewpoint
X	Wyse WY-50 or Ultimate ULT-50, Native mode
Y	Wyse WY-85 VT220 7-bit
Z	HP 700/92

For printer control, the SYSTEM-CURSOR subroutine generates the string necessary to set up special printer features. The printer type is determined by the current PRINTER command; it is a code character stored in byte zero of the Quaternary Control Block (QCB).

If T0 indicates printer control rather than terminal control, TERMTYPE is ignored. Instead, the output of SYSTEM-CURSOR is determined by the current printer type, as set by the PRINTER system command. The following are valid Ultimate-defined printer types:

H Honeywell (NEC) letter quality (the default)

L Hewlett Packard LaserJet

Other printers, such as line printers, do not have any special functions defined that can be invoked via SYSTEM-CURSOR.

When used for terminal control, SYSTEM-CURSOR does not perform any output. Typically, a calling routine would set R15 to the current position in the terminal output buffer (OB workspace), and specify a terminal function in T0 (and possible CTR10 and CTR11). Then it would call SYSTEM-CURSOR to generate the appropriate terminal control string. After SYSTEM-CURSOR terminates, R15 would be pointing to the new end of output data. The OB register could then be set to this value and WRTLIN or WRITOB could be called to do the actual output. See the following example.

The following terminal and printer control functions require a second parameter to be specified to SYSTEM-CURSOR:

(-30,c) (-31,f) (-32,b) (-101,p) (-102,h)

In these cases, the value of the parameter must be placed in CTR10. An example is changing the background color on a terminal: T0 is set to -32, and CTR10 is set to the code for the desired color.

For more information on writing a cursor or printer control routine, and loading a TERMDEF item, see Section 7.

```
MOV     OBBEG,R15       SET R15 TO START OF OUTPUT BUFFER
LOAD    -1 LOAD         CLEAR-SCREEN CODE VALUE
BSL     SYSTEM-CURSOR   GENERATE ESCAPE SEQUENCE
MOV     R15,OB          SET OB TO END OF DATA
BSL     WRITOB          CLEAR SCREEN; NO CR-LF AFTERWARDS
```

Table 5-1. Cursor Control Values (1 of 8)

Code	Description
@(-1)	Clear the screen and positions the cursor at 'home' (upper left corner of the screen).
@(-2)	Position the cursor at 'home' (upper left corner).
@(-3)	Clear from cursor position to the end of the screen.
@(-4)	Clear from cursor position to the end of the line.
@(-5)	Start blink.
@(-6)	Stop blink.
@(-7)	Start protected field.
@(-8)	Stop protected field.
@(-9)	Backspace the cursor one character.
@(-10)	Move the cursor up one line.
@(-11)	Move the cursor down one line.
@(-12)	Move the cursor right one column.
@(-13)	Enable auxiliary (slave) port.
@(-14)	Disable auxiliary (slave) port.
@(-15)	Enable auxiliary (slave) port in transparent mode.
@(-16)	Initiate slave local print.
@(-17)	Start underline.
@(-18)	Stop underline.
@(-19)	Start reverse video.
@(-20)	Stop reverse video.
@(-21)	Delete line.
@(-22)	Insert line.

Table 5-1. Cursor Control Values (2 of 8)

Code	Description
@(-23)	Scroll screen display up one line.
@(-24)	Start boldface type.
@(-25)	Stop boldface type.
@(-26)	Delete one character.
@(-27)	Insert one blank character.
@(-28)	Start insert character mode.
@(-29)	Stop insert character mode.
@(-30,c)	Set foreground and background color:

c	background	foreground
1	black	cyan
2	black	red
3	black	blue
4	black	green
5	black	magenta
6	black	yellow
7	black	white
8	blue	red
9	blue	green
10	blue	white
11	blue	yellow
12	blue	red
13	blue	cyan
14	blue	magenta
15	white	red
16	white	green
17	white	blue
18	white	cyan
19	white	magenta
20	white	black
21	red	white
22	red	green

Table 5-1. Cursor Control Values (3 of 8)

Code	Description
@(-31,f)	Set foreground color: **f** **foreground** 1 brown (may vary on some terminals) 2 white 3 red 4 magenta 5 yellow 6 green 7 cyan 8 blue
@(-32,b)	Set background color: **b** **background** 1 brown 2 white 3 black 4 red 5 blue 6 cyan 7 magenta
@(-33)	Set 80 columns.
@(-34)	Set 132 columns.
@(-35)	Set 24 rows.
@(-36)	Set 42 rows.
@(-37)– @(-45)	Reserved

Table 5-1. Cursor Control Values (4 of 8)

Code	Description
@(-46)	Returns function key default values as a string in the following format: sFBfFBx1FA...xnFBy1FA...ynFBeFB s character sequence needed to set the overall characteristics of the function line; typically, this is null FB CHAR(252)[1] f lead-in sequence used to load function keys xn value for function key n FA CHAR(251) yn value for shifted function key n e terminator for key text
@(-47)	Returns character sequence needed to set the overall characteristics for the label line (bottom line of terminal). The following information is returned: sFBfFBxFByFBeFBr s character sequence needed to set the overall characteristics of the label line FB CHAR(252) f lead-in sequence used for label line x lead-in sequence for unshifted label line y lead-in sequence for shifted label line e terminator for text r reset label line (turn off)

[1] After the string is returned, the CONVERT function can be used to change the delimiters to attribute marks (CHAR 254) and value marks (CHAR 253) if desired. (Doing this converts the string to a dynamic array.)

Table 5-1. Cursor Control Values (5 of 8)

Code	Description
@(-48)	Returns character sequence needed to set the overall characteristics for the status line (top line of terminal). The following information is returned: sFBfFBxFByFBeFBr s character sequence needed to set the overall characteristics of the status line FB CHAR(252) f lead-in sequence used for status line x lead-in sequence for unshifted status line y lead-in sequence for shifted status line e terminator for text r reset status line (turn off)
@(-49)	Returns string that defines the graphics characters codes for the current terminal; the exact characters that will be displayed depend on the terminal type. Before the code is printed, the terminal's graphic capability must be turned on by an @(-50) statement. After the graphics have been printed, the graphic capability must be turned off by an @(-51) statement. The codes in @(-49) are single digits whose meanings are determined by the position of the code in the string. The first eleven positions in the string define the following single line graphics characters:

Table 5-1. Cursor Control Values (6 of 8)

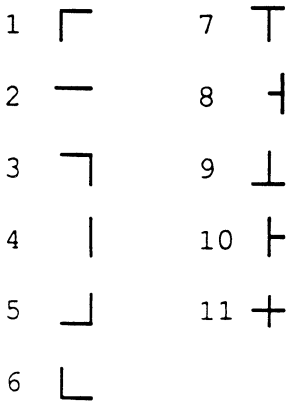
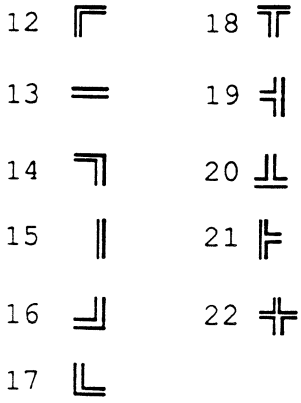
Code	Description
	 The second set of eleven positions define the following double line graphics characters:  The 23rd through 26th positions define other graphic characters, depending on the terminal type.
@(-50)	Start graphics.
@(-51)	Stop graphics.
@(-52)	Start blink.
@(-53)	Stop blink.
@(-54)	Start reverse video.

Table 5-1. Cursor Control Values (7 of 8)

Code	Description
@(-55)	Stop reverse video.
@(-56)	Start reverse video and blink.
@(-57)	Stop reverse video and blink.
@(-58)	Start underline.
@(-59)	Stop underline.
@(-60)	Start underline and blink.
@(-61)	Stop underline and blink.
@(-62)	Start underline and reverse video.
@(-63)	Stop underline and reverse video.
@(-64)	Start underline, reverse video, and blink.
@(-65)	Stop underline, reverse video, and blink.
@(-66)	Start dim.
@(-67)	Stop dim.
@(-68)	Start dim and blink.
@(-69)	Stop dim and blink.
@(-70)	Start dim and reverse video.
@(-71)	Stop dim and reverse video.
@(-72)	Start dim, reverse video, and blink.
@(-73)	Stop dim, reverse video, and blink.
@(-74)	Start dim and underline.
@(-75)	Stop dim and underline.
@(-76)	Start dim, underline, and blink.

Table 5-1. Cursor Control Values (8 of 8)

Code	Description
@(-77)	Stop dim, underline, and blink.
@(-78)	Start dim, reverse video, and underline.
@(-79)	Stop dim, reverse video, and underline.
@(-80)	Set 80 columns
@(-81)	Reserved
@(-82)	Set 132 columns

Table 5-2. Letter-Quality Printer Control Values

Code	Description
@(-101,p)	Set VMI (Vertical Motion Index) to p.
@(-102,h)	Set HMI (Horizontal Motion Index) to h.
@(-103)	Set alternate font.
@(-104)	Set standard font.
@(-105)	Generate a half line-feed.
@(-106)	Generate a negative half line-feed.
@(-107)	Generate a negative line-feed.
@(-108)	Print black ink.
@(-109)	Print red ink.
@(-110)	Load cut sheet feeder.
@(-111)	Select feeder1.
@(-112)	Select feeder2.
@(-113)	Select standard thimble.
@(-114)	Select proportional space thimble.
@(-115)	Start automatic boldfacing.
@(-116)	Stop automatic boldfacing.
@(-117)	Start automatic underlining.
@(-118)	Stop automatic underlining.

TERM-INFO

The TERM-INFO subroutine returns information about the specified terminal type, such as the function and cursor key definitions.

Input (user specified):

R15	R	points one byte before the location where the 38 flag characters are to be stored.

Input (system specified):

TERMTYPE	C	specifies terminal type

Output:

R15	R	points to the SM after the last character of generated 38-byte string (or unchanged for null value)
D0	D	32-bit flag word

Elements used:

None except standard scratch elements

Description

The TERM-INFO subroutine returns a 32-bit flag word in D0, and 38 bytes of function and cursor control key definition characters at the byte address of R15. Flag 0 is the high-order bit of D0 (the PSYM name is B31), flag 1 is the next-to-high-order bit (B30), and so on. However, if flag bit 0 (B31) is not set, indicating an undefined terminal type, no function key or cursor key definition information is returned.

R15 is returned pointing to the SM after the 38 characters. The characters are stored beginning one byte after the original byte address of R15.

For more information on writing a cursor or printer control routine, and loading a TERMDEF item see Section 7.

The flag bits are specified in the TERMDEF OPTION specification and are returned as follows in D0:

Flag	Description	Flag	Description
0	terminal is a CRT	9	insert character
1	terminal has Aux port	10	no-roll status Line
2	programmable Aux port	11	multi-byte function keys
3	transparent Aux port	12	multi-byte cursor keys
4	discrete visual attribute	13	RETURN follows function key
5	attribute takes a space	14	multi-byte cursor lead in
6	delete line	15	multi-byte function lead in
7	insert line	16	has reverse scroll
8	delete character	17	block mode terminal (such as IBM 3270)

The function and cursor key definitions are returned in the following order at R15:

Byte	Description
0	function key lead in character (normally ESC; but STX for Viewpoint)
1-16	function keys 1-16 (second byte if flag bit 11 set)
17-32	shifted function keys 1-16
33-37	cursor home, left, up, down, right (second byte if flag bit 12 set)

Note: *TERMTYPE, which specifies the terminal type, is initialized at logon, but may be changed by commands such as TERM and TERMINAL.*

TIME
DATE
TIMDATE

The TIME, DATE, and TIMDATE subroutines return the system time and/or the system date, and store them in the buffer area specified by register R15.

Input (user specified):

R15	R	Points one prior to the buffer area

Output:

R15	R	Points to the last byte of the data stored

Elements used:

D2	D	Used by TIME and TIMDATE only
D3	D	Used by TIME and TIMDATE only

Description

The time is returned as on a 24-hour clock.

Entry	Buffer size required (bytes)	Format
TIME	9	hh:mm:ss
DATE	12	dd mmm yyyy
TIMDATE	22	hh:mm:ss dd mmm yyyy

TPBCK

The TPBCK subroutine backspaces the tape one physical record, or block. The tape must be attached to the process via the TCL T-ATT command.

Inputs:

None

Output:

None

Elements used:

None except standard scratch elements

Description

Not all tape drives can backspace. On some Ultimate systems, calling TPBCK causes a logical backspace, which resets a pointer into a tape data buffer. Multiple backspaces are not guaranteed to work.

TPREAD
TPWRITE
TPRDBLK

The TPREAD subroutine reads a specified number of bytes from tape into a buffer area. The TPWRITE subroutine writes a specified number of bytes to tape. The TPRDBLK reads one physical tape record, or block, into an internal tape buffer, and returns a pointer to the data along with the number of bytes read.

Input (user specified):

R15	R	for TPREAD and TPWRITE, points to one byte before the source or destination buffer area
D0	D	for TPREAD and TPWRITE, contains the number of bytes to be transferred to or from the tape

Output:

R15	R	for TPREAD and TPWRITE, points to the last character transferred to or from the source or destination buffer area
R7	R	for TPRDBLK, points to one byte before the beginning of data in the internal tape buffer
D0	D	for TPRDBLK, contains the number of bytes read
EOFBIT	B	for TPREAD and TPRDBLK, indicates end-of-file if set

Elements used:

The tape handler stacks and restore most of the elements which it uses. The following elements are modified, however:

YMODE	T	Used to save and restore XMODE; the XMODE routine, if any, on entry to the tape routines, is not guaranteed to work until the particular routine is exited and XMODE has been restored
R7	R	Tape buffer pointer; must be maintained between calls to TPREAD and TPWRITE
T5	T	

T6	T	
T7	T	
D2	D	
OVRFLW	D	Used by routine GETBLK
RECORD	D	
FRMN	D	
FRMP	D	
NNCF	H	
NPCF	H	

Description

All three routines use a virtual tape drive. The initial execution of any one of them causes initialization of a buffer in virtual space used for transferring tape records between the controller and main memory. This buffer typically consists of a set of contiguous frames obtained from the system overflow pool, linked together to form a block large enough to accommodate the maximum block size of the tape drive. These frames are automatically released during WRAPUP processing, just before return to TCL.

For TPREAD and TPWRITE, the contents of the accumulator, D0, is the number of characters to transfer to or from the tape buffer. Also, for these routines, Register R7 is used as the tape buffer pointer and must be preserved from one call to the next . For TPRDBLK, R7 is reset on each call.

Bit EOFBIT is set when the tape mark is reached on reading a tape. End of tape conditions are automatically handled by the tape routines.

If D0 is zero on a write, TPWRITE fills the rest of the tape buffer with the character pointed to by R15, which causes the buffer to be written to tape. This is recommended in order to send the last partial tape record to the tape, after which TPWEOF should be called.

The tape drive must be attached before calling these routines, otherwise they exit to WRAPUP with an error message. The TCL T-ATT command is used to attach a tape, and also to set the block size for TPWRITE.

These routines may be used with either labeled or unlabeled tapes. For labeled tapes, the routines TPRDLBL, TPRDLBL1, TPWTLBL, and TPWTLBL1 may be used to read and write the labels. See the documentation on these routines for more information.

Note: *Some tape drive interfaces require the number of bytes to be specified when reading tape. This means that TPRDBLK, on some Ultimate systems, always returns a maximum-size logical block rather than a physical block of data. The Ultimate 1400 system, for example, always returns the number of characters in the block size set by the current T-ATT command.*

TPREW

The TPREW subroutine is used to rewind the tape. The tape must be attached to the process via the TCL T-ATT command.

Inputs:

None

Output:

None

Elements used:

None except standard scratch elements

TPWEOF

The TPWEOF subroutine is used to write a tape mark on the tape. The tape must be attached to the process via the TCL T-ATT command.

Inputs

None

Output:

None

Elements used:

None except standard scratch elements

UPDITM

The UPDITM subroutine performs updates to a disk file. If the item is to be deleted, the routine compresses the remainder of the data in the group in which the item resides. If the item is to be added, it is added at the end of the current data in the group. If the item is to be replaced, a delete, then add function takes place.

Input (user specified):

BMSBEG	S	points one prior to the item-id of the item to be updated; the item-id must be terminated by an AM
TS	R13	points one prior to the item body to be added or replaced (no item-id or count field); not needed for deletions; the item body must be terminated by a SM
CH8	C	contains the character D for item deletion, U for item addition or replacement
FCB1	D	contain the access information for the file being updated
FCB2	D	

Output:

None

Elements used:

NNCF	H	Scratch
NPCF	H	
XMODE	T	
D2	D	
D3	D	
RECORD	D	
FRMN	D	
FRMP	D	
IR	R6	
BMS	R8	
UPD	R7	

Description

If the update causes the data in the group to reach the end of the linked frames, NEXTOVF is entered to obtain another frame from the overflow space pool and link it to the previous linked set; as many frames as required are added.

If the deletion or replacement of an item causes an empty frame at the end of the linked frame set, and that frame is not in the primary area of the group, it is released to the overflow space pool. Once the item is retrieved, processing cannot be interrupted until completed.

Note that this routine does not perform a merge with the data already on file. In order to change an item, it must first be read and copied to the user's workspace, changed there, and then updated back to the file using UPDITM.

If a group format error is encountered (premature end of linked frames, or non-hexadecimal character found in an item count field), an error message is printed and the group is terminated at the end of the last good item before processing continues.

WRITE@OB
WRITEX@OB

The WRITE@IB subroutine stores the character at the current position of R11 in the next position of the asynchronous output buffer. The WRITEX@IB subroutine stores the character at the current position of R11 unless current terminal performs a local echo (such as IBM 3270 types), in which case it does nothing.

Input (user specified):

R11	R	

Output:

R11	R	unchanged
R14	R	

Elements used:

None except standard scratch elements

Description

If the output buffer is full, the process is suspended until characters are removed by the asynchronous channel controller.

The WRITEX@OB routine must be used whenever code is writing only to echao a character that was read in by READX@IB. This is because some terminals do a local echo and the systems that support those terminals must be able to distinguish instructions used for only echoing.

WRTLIN
WRITOB

The WRTLIN and WRITOB subroutines are the standard routines for outputting data to the terminal or printer. WRTLIN deletes trailing blanks from the data, and increments the internal line counter LINCTR; WRITOB does neither.

Input (user specified):

OB	R11	points to the last character in the OB buffer; the buffer must extend at least two characters beyond this location
NOBLNK	B	if set, blanking of the output buffer is suppressed; this bit is normally zero
PFILE	T	contains the spooler print file number; meaningful only if LPBIT is set; unless more than one print file is being simultaneously generated, the normal value of zero may be used
PAGHEAD	S	points one byte before the beginning of the page heading message; meaningful only if PAGINATE is set. If the FID field of this storage register is zero, no heading is printed; this is the default condition
PAGFOOT	S	points one byte before the beginning of the page footing message; meaningful only if PAGINATE is set. If the FID field of this storage register is zero, no footing is printed; this is the default condition

Input (system specified):

OBBEG	S	standard system buffer used to store data for terminal or printer output
LISTFLAG	B	if set, all output to the terminal is suppressed; set and reset by the TCL P command and by the debug P command

LPBIT	B	if set, output is routed to the spooler (Note: routine SETLPTR should be used to set this bit so printer characteristics are set up correctly)
LFDLY	T	lower byte contains the number of fill characters to be output after a CR/LF; set by the TCL TERM command
PAGINATE	B	if set, pagination and page headings are invoked; usually set by the PRNTHDR routine in conjunction with page heading and/or footing
OTABFLG	B	output tabs in effect if set (by the TCL TABS command)

The following specifications are meaningful only if PAGINATE is set:

PAGSIZE	T	contains the number of printable lines per page; set by the TCL TERM command
PAGSKIP	T	contains the number of lines to be skipped at the bottom of each page; set by the TCL TERM command
PAGNUM	T	contains the current page number
PAGFRMT	B	if set, the process pauses at the end of each page of terminal output, until the user enters any character to continue; a <CTRL-X> returns the process directly to TCL; normally set, this bit is zeroed by the N option at TCL for most commands, or by the NOPAGE modifier in Ultimate RECALL
FOOTCTR	T	contains the number of lines to the point in the page where the FOOTING is to print, if a footing is in effect; set by the PRNTHDR routine; if the footing is changed by the user, the subroutine SETFOOTCTR must be called to reset this tally

Output:

OB	R11	=OBBEG

Elements used:

BMS	used if LPBIT set
ATTOVF	used if LPBIT set

Plus standard scratch elements

Description

OBBEG points to the beginning of the data to be output; the last byte of data is at the address pointed to by OB. This is for convenience in calling the subroutines; normally they are called just after data has been transferred to the OB buffer, in which case the OB address register is on the last byte copied. On return, the OB address register is set back to OBBEG, again for convenience; the output buffer area is blanked unless bit NOBLNK is set.

The data is output to the terminal if bit LPBIT is off; otherwise the data is stored in the printer spooling area.

Pagination and page heading and page footing routines are automatically invoked if bit PAGINATE is set. If headings or footings are also needed, the page heading and page footing buffers must be set up by the user; see the PRNTHDR topic.

If PAGINATE is set, the end of page is checked for, and action is taken to automatically print the page footing (if it exists), skip to the next page, and print a new page heading (if it exists). The end of page is triggered when either LINCTR reaches PAGSIZE (when there is no footing), or reaches FOOTCTR (when there is a footing). However, a value of zero in PAGSIZE suppresses pagination, regardless of the setting of PAGINATE.

WRTLIN and WRITOB also perform output tabulation as specified by the TCL TABS command, when output is to the terminal. In this case, blank sequences in the output are checked against the output tab stops; if a sequence of blanks crosses a tab stop, a tab character (X'09') is output instead of the blanks.

WSINIT

The WSINIT subroutine initializes the following process workspace pointer triads:

AF, AFBEG, AFEND

BMS, BMSBEG, BMSEND

CS, CSBEG, CSEND

IB, IBBEG, IBEND

OB, OBBEG, OBEND

TS, TSBEG, TSEND

It also initializes PBUFBEG and PBUFEND.

Inputs

None

Output:

R9	R	AF workspace pointer
R8	R	BMS workspace pointer
R12	R	CS workspace pointer
R10	R	IB workspace pointer
R11	R	OB workspace pointer
R13	R	TS workspace pointer

The first byte of the AF, CS, IB, and OB workspaces is set to X'00'. The OB workspace is filled with blanks (X'20'). IBSIZE and OBSIZE are set to 465 if initially greater.

Elements used:

None except standard scratch elements

Description

For each workspace, the beginning storage register (and associated address register, if present) is set pointing one before the first byte of the workspace, and the ending storage register is set pointing to the last data byte.

All workspaces except the TS and PROC (PBUFBEG to PBUFEND) are contained in the frame at PCB+4; PBUFBEG and PBUFEND define a 4-frame linked workspace; TSBEG and TSEND define a single unlinked frame.

For more information on workspaces, see the section in Chapter 3 entitled, "Addressing Conventional Buffer Workspaces."

WTLINK

The WTLINK subroutine writes link field information. See RDLINK for details.

6 System Software Interfaces

When you log on to the system, you are normally at the TCL level of operation, which is the primary user interface with the Ultimate system. The TCL processor is the main way the Ultimate operating system exercises its flow of program control.

All system commands are processed by the TCL processor. At the conclusion of the command, it goes through WRAPUP software before returning to TCL, or if the PROC program was in control, to PROC. (WRAPUP is described later on in this chapter.)

The following are forms of commands that can be executed from TCL:

PROC — The PROC program gains control; it may call the TCL software as a subroutine by generating any TCL command and executing it via the PROC P command, or it may return to TCL via the PROC exit (X) command.

TCL-I verb — Typically these do not require file I/O; examples are TIME, SLEEP, and POVF.

TCL-II verb — These verbs always access a file and usually items in that file; examples are EDIT, COPY and AS. The TCL-II file-handler opens the file and retrieves the item, and then transfers control to the specific processor. The latter returns to the file-handler after completion of processing for that item.

Ultimate RECALL or Ultimate UPDATE Command — The Ultimate RECALL compiler is first called to compile the command. The compiled string is passed to the selection program, which acts as the file-handler for the Ultimate RECALL run-time. If the command requires sorting (SORT, SSELECT), the SORT program is called to sort the selected items on the specified sort-keys. The Ultimate RECALL run-time program is called as each item is selected, and it returns control to the selection program after completion of processing for that item.

Cataloged BASIC Program	Cataloged BASIC programs function similar to assembly language programs, but are written in BASIC. Examples include CREATE-FILE and TERM-INIT.
Compile-and-Go BASIC Program	The BASIC compiler is first called to compile the source statements in the item. Then the generated object code is executed, after which it is discarded.

Interfaces Between TCL and User Programs

Control can be transferred to a user-written program and then returned to the system using any of the following interfaces:

CONV interface	allows subroutines to be called from BASIC or RECALL, and allows parameters to be passed back and forth.
PROC interface	used when an assembly program is called as a subroutine via a PROC.
RECALL	provides the full power of the Ultimate RECALL selection, sort, and correlative or conversion processing.
TCL-I interface	used when no disk file I/O is to be done
TCL-II interface	used when a file or items in a file need to be accessed; TCL-II relieves the program of responsibility of file opening and item retrieval.
XMODE interface	used to set up a special routine that handles the Forward Link Zero abort condition (end of current set of frames) without terminating the program. Instead, the user can set up XMODE to branch to a subroutine that links additional frames to the end of the current frame set, complete the disk write, and then continue with the program

Please see Chapter 7 for examples of programs and their associated interfaces.

The Initial Conditions of a Process at TCL

When a process is at TCL, its process workspace pointers are at an initialized state; however, the data in the workspace buffers is whatever has been left over from the last program.

The initial conditions of a process at TCL are as follows:

PCB Symbols	Contents
MFCB1 MFCB2	Access information for the Master Dictionary (MD)
EFBC1 EFCB2	Access information for ERRMSG file
USER	Describes current log status: 5 = logged on 3 = logging off.
Scan character SC0 SC1 SC2	 Contains X'FB' (SB) blank blank
ABIT through ZBIT AFLG through ZFLG SB0 through SB35	Zeroed Zeroed Zeroed
Workspace pointers	Initialized to beginning and end of buffer spaces.

Output Devices	Contents
Terminal and printer characteristics (such as paper depth and width)	Initialized by the TERM command.

Note that the initial conditions of a process mean that the process has access to the user's MD and to the ERRMSG file.

CONV Interface

The CONV interface may be used to call the conversion software as a subroutine or to call a user-written subroutine from BASIC or Ultimate RECALL.

Two distinct but closely connected interfaces are covered in this topic:

- calling conversion program as a subroutine
- calling a user-written subroutine from BASIC or Ultimate RECALL

The entire section should be read carefully for a complete understanding of the methods involved.

Conversions can be either input or output, as follows:

input conversions:	conversion of a user input to intermediate (processing) format according to specified code.
output conversions:	conversion of data in intermediate (processing) format to output format according to specified code.

Calling Conversion Program as a Subroutine

The CONV entry point may be used to call the entire conversion program as a subroutine (BSL CONV), which will perform any and all valid conversions specified in the conversion string. It is normally used when the user writes an assembly program that requires one of the standard user conversions (see Figure 6-1 for conversion codes; for details, see the *Ultimate RECALL and UPDATE User Guide*).

Inputs (user specified):

TSBEG	S	Points one before the value to be converted; the value is converted in place, and the buffer is used for scratch space; therefore it must be large enough to contain the converted value; the value to be converted must be terminated by any of the standard system delimiters (SM, AM, VM, or SVM)

IS	R4	Points to the first character of the conversion code specification string for CONV. If CONVEXIT is to be used to check for more conversion codes (see below), IS points at least one before the next conversion code (after a VM) or AM at the end of the string, or to the AM; the code string must end with an AM; initial semicolons (;) are ignored
MBIT	B	Set if input conversion is to be performed; zero for output conversion

Outputs:

IS	R4	Points to the AM/VM terminating the conversion codes
TSBEG	S	Points one before the converted value
TS	R13	Point to the last character of the TSEND converted value; a SM is also placed one past this location;
TSEND	S	=TS

If a null value is returned, TS=TSEND=TSBEG

Elements used:

SB10	B
SB11	B
SB12	B
SC2	C
CH0	T
CTR1	T
CTR20	T
CTR21	T
CTR22	T
CTR23	T
S4	S
S5	S
S6	S
S7	S

Code	Description
D	convert date to internal or external format
G	extract group of characters
L	test string length
MC	mask characters by numeric, alpha, or upper/lower case
ML	mask left-justified decimal data
MP	convert integer to packed decimal or vice versa
MR	mask right-justified decimal data
MT	convert time to internal or external format
MX	convert ASCII to hexadecimal or vice versa
P	test pattern match
R	test numeric range
T	convert by table translation; the table file and translation criteria must be given *Note: This type of conversion is inefficient if several items or attributes will be accessed.*
U	convert by subroutine call to assembly routine, either system- or user-defined. The hexadecimal mode-id (absolute address) of the routine must be given (as discussed above). The INPUT.VAL may be a parameter to be passed to the subroutine or a null string if none is needed. If two or more parameters are to be passed, they must be compressed into a single string in INPUT.VAL and parsed by the called routine.

Figure 6-1. Processing Codes

Calling a User-Written Subroutine

The CONV entry point may also be used to call a user-written subroutine from BASIC or Ultimate RECALL. This conversion program interface is the single most useful interface in the system. The interface to both these programs is identical.

- To call a user subroutine from BASIC, use the ICONV or OCONV function as follows:

```
CONVERTED.VAL = OCONV(INPUT.VAL, 'Uxxxx')
CONVERTED.VAL = ICONV(INPUT.VAL, 'Uxxxx')
```

  The value 'xxxx' in 'Uxxxx' is the hexadecimal mode-id of the user-written subroutine. The unconverted or raw value in variable INPUT.VAL is passed as a parameter to the user assembly code, which performs such action as needed, and the result is returned to the BASIC program as a value (in the above example) stored in the variable CONVERTED.VAL.

  ICONV or OCONV is used depending on whether the user wants input or output conversion to be performed (if the distinction does exist).

- To call a user subroutine from Ultimate RECALL, a correlative or conversion code of the following format should be used:

```
Uxxxx
```

  The value 'xxxx' in 'Uxxxx' is the hexadecimal mode-id of the user-written subroutine. Ultimate RECALL passes the unconverted value (which may have previous conversions or correlatives already applied, since these fields in the dictionary item may be multiple valued) as a parameter to the user assembly code; the latter performs such action as needed, and the result is returned to Ultimate RECALL.

In both cases, the unconverted value is a string stored in the buffer defined by TSBEG. It is important to note that the actual location of the buffer is irrelevant. The actual TS buffer, as initialized at TCL, is only 512 bytes in length. This buffer is rarely used, since TSBEG can be freely moved around to point to any scratch space available. However, the symbolic reference via TSBEG always locates the data, so the physical location need be of no concern.

All conversion programs adopt the convention that the converted value is returned in the same location, overlaying the original value. User-written conversions *must* follow this convention. The space available beyond the original unconverted parameter is considered scratch, and may be used freely.

Chapter 7 contains an example of an assembly language program called from a conversion program.

Inputs (user specified):

IS	R4	points to non-hexadecimal character following the Uxxxx string
MBIT	B	set for ICONV function or from selection software in Ultimate RECALL.; reset if OCONV function or LIST output software in Ultimate RECALL.
TSBEG	S	points one before unconverted parameter from BASIC or Ultimate RECALL; value is terminated by any system delimiter

Outputs:

IS	R4	Points to the AM/VM terminating the conversion codes
TSBEG	S	Points one before the converted value
TS	R13	Point to the last character of the TSEND converted value; a SM is also placed one past this location
TSEND	S	=TS

If a null value is returned, TS=TSEND=TSBEG

Exit Conventions:

One of two methods of exit can be used:

- The conventional exit is to entry CONVEXIT, which will process further conversion codes, if any. In this case, the IS register must point either to the delimiter terminating the Uxxxx code, which can be a VM or an AM, or to anywhere before it.
- If it is known that no further codes exist, or if these codes are not to be processed, a RTN instruction may be executed. In this case, it is irrelevant where the IS register points.

PROC Interface

The PROC interface is used when the program is to be called during the execution of a PROC.

When a program is called from PROC, the PROC software regains control after the assembly subroutine terminates execution. PROC may call the TCL software as a subroutine by generating any TCL command and executing it via the PROC P command, or it may return to TCL via the PROC exit (X) command.

A user-written program can gain control during execution of a PROC by using the Pxxxx or Uxxxx command in the PROC, where 'xxxx' is the hexadecimal mode-id of the user routine (*user exit*). These PROC commands operate as follows:

Pxxxx executes current primary output buffer data and performs the normal TCL initialization, then transfers control to the specified mode-id instead of to TCL

Uxxxx transfers control to the specified mode-id without any TCL involvement or initialization.

The routine can perform special processing, and then return control to the PROC software. Necessarily, certain elements used by PROC must be maintained by the user program; these elements are marked with an asterisk in the table below.

Inputs (system specified):

FCB1	D	} contain the file access information for the MD
FCB2	D	}
PQBEG*	S	points one prior to the first PROC statement
PQEND*	S	points to the terminating AM of the PROC
PQCUR	S	point to the AM following the Pxxxx or Uxxxx statement
IR	R	=PQCUR

PBUFBEG*	S	points to the buffer containing the primary and secondary input buffers; buffer format is SB ... primary input... SM SB ... secondary input ... SM
ISBEG*	S	points to the buffer containing the primary output line
STKBEG*	S	points to the buffer containing "stacked input" (secondary output)
SBIT*	B	set if a STON command is in effect
SC2*	C	contains a blank
IB	R10	current input buffer pointer (may point within either the primary or secondary input buffers)
IS	R4	if SBIT is set, points to the last byte moved into the primary output buffer; if SBIT is not set, points to the last byte moved into the secondary output buffer
UPD	R7	if SBIT is set, points to the last byte moved into the primary output buffer; if SBIT is not set, points to the last byte moved into the secondary output buffer

Outputs:

IR	R	points to the AM preceding the next PROC statement to be executed; may be altered to change PROC execution
IS	R4	may be altered as needed to alter data within the input and output buffers, but the formats described above must be maintained
UPD	R7	
IB	R10	

Exit Convention

The normal method of returning control to the PROC software is to execute an external branch instruction (ENT) to 2,PROC-I. To return control and also reset the buffers to an empty condition, entry 1,PROC-I may be used.

If it is necessary to abort PROC control and exit to WRAPUP, bit PQFLG should be reset before branching to any of the WRAPUP entry points (see WRAPUP topic).

When a PROC eventually transfers control to TCL via the P operator, the following elements are expected to be in an initial condition:

bits ABIT through ZBIT

bits AFLG through ZFLG

bits SB0 through SB30

scan character SC0 must contain a SB, scan characters SC1 and SC2 must each contain a blank

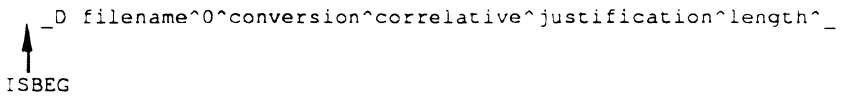
It is best to avoid usage of these bits in PROC user exits. However, if a user routine does use any of these elements, they should be reset before returning to the PROC, unless the elements are deliberately set up as a means of passing parameters to other programs.

RECALL Interface

The Ultimate RECALL interface may be used whenever a program needs to use the Ultimate RECALL capabilities for output. The Ultimate RECALL interface requires more care than the TCL-I and TCL-II interfaces because it uses most of the available global elements, but it provides the full power of the Ultimate RECALL selection, sort, and correlative or conversion processing.

Ultimate RECALL software uses a compiled string that is stored in the IS work space. String elements are separated by SMs. There is one file-defining element in each string, an element for each attribute specified in the original statement, and an element for each explicitly listed item-ID.

A file defining element has the following format:



An attribute defining element has the following format:

```
_c attribname^amc^conversion^correlative^justification^length^_
```

where c is one of the following:

- A regular attribute
- Q controlling attribute
- B dependent attribute
- Bx SORT connectSORT-BY, SORT-BY-DSND, etc; x is from attribute 1 of the connective

An explicit item-ID element has the following format:

```
_I item-id _
```

An end-of-string element has the following format:

```
_ Z
```

A typical Ultimate RECALL statement passes through the compiler and the selection programs before entering the program that produces the

final report. All statements must pass through the first two stages, but control can be transferred to user-written programs from that point onward. It is possible to interface with the Ultimate RECALL software at following points:

- after selection and sorting, if specified, but before processing codes have been applied
- after processing codes have been applied

Gaining Control After Selection

A user program can get control directly from the selection software or from GOSORT if a sorted retrieval is required.

The selection software performs the actual retrieval of items that meet the selection criteria, if specified. Each time an item is retrieved, the software at the next level is entered with bit RMBIT set; a final entry with RMBIT zero is also made after all items have been retrieved. If a sorted retrieval is required, the selection software passes items to the GOSORT mode, which builds up the sort-keys preparatory to sorting them. After sorting, GOSORT retrieves the items again, in the requested sorted sequence.

To define verbs that gain control after selection, use the following format:

attribute number	unsorted	sorted
001	PB	PB
002	35	35
003	xxxx	4E
004		xxxx

xxxx hexadecimal mode-ID of the user program

The mode-ID is loaded into the tally MODEID2 for later use.

Note: Attribute 1 must be PB.

In this method of interface, only item retrieval has taken place; none of the conversion and correlative processing has been done. For functional element interface, the column headed "Selection Software" in the table shown later must be used.

Exit Convention

On all but the last entry, the user routine should exit indirectly via RMODE (using an ENT* RMODE instruction); on the last entry, the routine should exit to one of the WRAPUP entry points. Processing may be aborted at any time by setting RMODE to zero and entering WRAPUP. Bit SB0 must also be set on the first entry.

Gaining Control After Processing Codes

A user program also gain control in place of the normal LIST formatter, to perform special formatting. The advantage here is that all conversions and correlatives have been processed, and the resulting output data has been stored in the history string (HS area).

Ultimate RECALL software obtains the data in output form (correlated and converted), creates a dummy item in the HS buffer, then turns control over to the user program.

To define verbs that gain control after processing codes, use the following format:

attribute number	unsorted	sorted
001	PA	PA
002	35	35
003	4D	4E
004	xxxx	xxxx

xxxx hexadecimal mode-ID of the user program

The mode-ID is loaded into the tally MODEID3 for later use.

Note: Attribute 1 must be PA.

Output data is stored in the HS area; data from each attribute is stored in the string, delimited by AMs, as follows:

```
X item-ID^value.one^ ... ^value n ^_Z
```

The X denotes a new item. The program must reset the history string pointer HSEND as items are taken out of the string. HSEND must be reset to point one byte before the next available spot in the HS work space, normally one before the first X code found.

Exit Convention

The exit convention for the LIST program is the same as for the selection software (see previous page).

Example

The following is an example of an assembly program that gains control from the LIST software to print item-IDs four at a time across the page.

The format of the verb is which uses the program is as follows:

```
item-ID     LIST4
  001       PA
  002       35
  003       4E        sorted; for unsorted, substitute 4D
  004       01FF      MODEID3 exit to frame 511, entry point 0
```

```
         FRAME 511
         *
HSENDDSP DEFTU HSEND          DISP FIELD OF HSEND
         *
         ZB    SB30           INTERNAL FLAG
         BBS   SB1,NOTF       NOT FIRST TIME
* FIRST TIME SETUP
         MOV   4,CTR32
         BSL   PRNTHDR        INITIALIZE AND PRINT HEADING
         SB    SB1
*
NOTF     BBZ   RMBIT,OP       LAST ENTRY
         BDNZ  CTR32,RETURN   NOT YET 4 ITEMS OBTAINED
         MOV   4,CTR32        RESET
OP       MOV   HSBEG,R14
LOOP     INC   R14
         BCE   C'X',R14,STOR  FOUND AN ITEM
         BCE   C'Z',R14,ENDHS END OF HS STRING
SCANSM   SID   R14,X'C0'      SCAN TO NEXT SM
         B     LOOP
STOR     BBS   SB30,COPYIT    NO FIRST ID FOUND
         SB    SB30           FLAG FIRST ID FOUND
         MOV   R14,SR28       SAVE LOCATION OF FIRST
         CMNT  *              "X"
COPYIT   MIID  R14,OB,X'A0'   COPY ITEM-ID TO OB
         MCC   C' ',OB        OVERWRITE AM
         INC   OB,5           INDEX
         B     SCANSM
ENDHS    BSL   WRTLIN         PRINT A LINE
         MOV   SR28,HSEND     RESTORE HS TO FIRST X CODE
         DEC   HSENDDSP       BACK UP ONE BYTE
         BBZ   RMBIT,QUIT
RETURN   ENT*  RMODE          RETURN TO SELECTION
         CMNT  *              PROCESSOR
QUIT     ENT   MD999          TERMINATE PROCESSING
         END
```

Element Usage

The following table summarizes the functional element usage by the selection and LIST software. Only the most important usage is described; elements that have various usages are labeled scratch. A blank in a column indicates that the program does not use the element. Since the LIST program is called by the selection program, any element that is not to be used by others in the former is indicated by a blank usage in the latter column.

In general, user routines may freely use the following elements:

bits	SB24 upwards
tallies	CTR30 upwards
double tallies	D3 thru D8
storage registers	SR20 upwards

SB0 and SB1 have a special connotation: they are zeroed by the selection program when it is first entered, and not altered thereafter. They are conventionally used as first-time switches for the next two levels of processing. SB0 is set by the LIST program when it is first entered, and user programs that gain control directly from selection should do the same. SB0 may be used as a first-entry switch by user programs that gain control from the LIST program.

An Ultimate RECALL verb is considered an update type of verb if the SCP character from line one of the verb definition is B, C, D, E, G, H, I, or J. These SCP characters are reserved for future Ultimate RECALL verbs.

The following should also be considered:

- If a full file retrieval is specified, the additional internal elements as used by GETITM are used. If explicit item-ids are specified, RETIX is used for retrieval of each item.
- Most elements used by the CONV and FUNC programs have been shown in the table; both may be called either by the selection program or the LIST program.
- Since the ISTAT, SUM, and STAT commands are independently driven by the selection program, the element usage of these programs is not shown.

Bits	Selection program	LIST program
ABIT	scratch	non-columnar list flag
BBIT	first entry flag	
CBIT	scratch	scratch
DBIT	scratch	dummy control-break
EBIT	reserved	control-break flag
FBIT	reserved	scratch
GBIT	reserved	scratch
HBIT	reserved	scratch
IBIT	explicit item-ids specified	
JBIT	reserved	D2 attribute in process
KBIT	by-exp flag	by-exp flag
LBIT	scratch	left-justified field
MBIT	CONV interface; zero	zero
NBIT	scratch	scratch
OBIT	selection test on item-id	
PBIT	scratch	scratch
QBIT	scratch	scratch
RBIT	full-file-retrieval flag	
SBIT	selection on values (WITH)	
TBIT	scratch	print limiter flag
UBIT	scratch	reserved
VBIT	reserved	scratch
WBIT	scratch	reserved
XBIT	scratch	reserved
YBIT	left-justified value test	left-justified print limiter
ZBIT	left-justified item-id	

Bits	Selection program	LIST program
SB0	unavailable	first entry flag,level one
SB1	unavailable	first entry flag, level two
SB2	reserved; zero	
SB3-SB17	scratch or reserved	scratch or reserved
WMBIT	FUNC interface	FUNC interface
GMBIT	FUNC interface	FUNC interface
BKBIT	scratch	scratch
RMBIT	set on exit if an item was retrieved; zero on final exit	
VOBIT	set for WRAPUP interface	

Bits	Selection program and LIST program
DAF1	set if SCP = B, C, D, E, G, H, I, or J
DAF8	set if accessing a dictionary
CFLG	set if C option or COL-HDR-SUPP specified
DFLG	set if D option or DET-SUPP specified
HFLG	set if H option or HDR-SUPP specified
IFLG	set if I option or ID-SUPP specified
CBBIT	set if BREAK-ON or TOTAL specified
DBLSPC	set if DBL-SPC specified
LPBIT	set if P option or LPTR specified
PAGFRMT	set unless N option or NOPAGE specified
TAPEFLG	set if T-LOAD verb (SCP = T) or TAPE specified

Tallies	Selection program	LIST program
C1; C3-C7	scratch	scratch
C2	contents of MODEID2	
CTR1-CTR4	scratch	scratch
CTR5	scratch	AMC of current element in IS
CTR6	reserved	scratch
CTR7	reserved	AMC corresponding to IR
CTR8	reserved	scratch
CTR9	reserved	scratch
CTR10	reserved	scratch
CTR11	reserved	scratch
CTR12	FUNC interface	current subvalue counter
CTR13	FUNC interface	current value counter
CTR14	reserved	scratch
CTR15	reserved	item size
CTR16	reserved	scratch
CTR17	reserved	reserved
CTR18	reserved	scratch
CTR19	reserved	sequence number for BY-EXP
CTR20-CTR23	CONV interface	CONV interface
CTR24	reserved	scratch
CTR25	reserved	scratch
CTR26	reserved	scratch
CTR27	reserved	current max-length
CTR28	reserved	scratch

Storage Registers	Selection program	LIST program
S1	points to the next explicit item-id	
S2-S9	scratch	scratch
SR0	points one before item count field	
SR1	points to the correlative field	current correlative
SR2	scratch	scratch
SR3	reserved	scratch
SR4	points to the last AM of the item	
SR5	reserved	points to the next segment in the IS
SR6	points to the conversion field	current conversion field
SR7	reserved	scratch
SR8-SR12	reserved	reserved
SR13	next sort-key; used by GOSORT only:	reserved
SR14-SR19	reserved	reserved
PAGHEAD	heading in HS if HEADING was specified	generated heading in HS

Address Registers	Selection program	LIST program
AF	scratch	scratch
BMS	within the BMS area	scratch
CS		scratch
IB		scratch
OB		output data line
IS	compiled string	compiled string
OS		scratch
TS	within the TS area	within the TS area
UPD		within the HS area
IR	within the item	within the item

Other Storage	Selection program	LIST program
D9	count of retrieved items	
D7	FUNC interface	FUNC interface
FP1-FP5	FUNC interface	FUNC interface
RMODE	return mode-id	
SIZE	item-size	scratch
FFCB1 FFCB2	access information for file	

Workspace Usage	Selection program	LIST program
BMS	contains the item-id	
HS	heading data	heading data; attribute data for special exits
TS	scratch	current value in process

TCL-I and TCL-II Interfaces

To initiate a user program as a TCL-I or TCL-II verb, a verb definition item must be created in the user's Master Dictionary (MD). The item-ID selected for the definition item becomes the command name to enter at TCL. The TCL software uses the information in the verb definition item to transfer control to the user program.

Figure 6-2 shows the structure of a TCL-I verb definition item. Figure 6-3 shows the structure of a TCL-II verb definition item.

Attribute	Content	Meaning
001	P{x}	P identifies item as a verb; x, if present, is the SCP character
002	X'nnnn'	Program primary mode-id.
003	{X'nnnn'}	Optional secondary program mode-id or a subroutine parameter.
004	{X'nnnn'}	Optional tertiary program mode-id or a subroutine parameter.

Figure 6-2. TCL-I Verb Definition Item Format

Attribute	Content	Meaning
001	P{x}	P identifies item as a verb; x, if present, is the SCP character
002	2	Constant 2; mode-id of TCL-II (entry point 0,frame 2)
003	X'nnnn'	Program primary mode-id
004	{X'nnnn'}	Optional secondary program mode-id or a subroutine parameter.
005	{options}	Code characters for TCL-II options: C copy retrieved items to the IS workspace F file access only; file parameters are set up, but any item-list is ignored by TCL-II. If present, any other options are ignored N new item acceptable; if the item specified is not on file, the secondary program still gets control (the EDITOR, for example, can process a new item) P display item-id on a full-file or item-list (more than one item) retrieval, or if a select list is in effect U updating sequence flagged; this option is required if items are to be updated as retrieved Z Final entry required; the secondary program is entered once more after all items have been retrieved (the COPY program, for instance, uses this option to print a final message)

Figure 6-3. TCL-II Verb Definition Item Format

SCP Character

The SCP character is an optional character that follows the P in verb definitions. If present, it may be any character except the following, which have special meaning:

G	send input line directly to program; do not parse it
O	ignore options
Q	identifies command as a PROC
U	update processor

The TCL software loads the SCP character into the byte in your PCB called SCP. It can be accessed by your program as necessary. It is generally used to allow verbs that are very similar to share mode-ids and entry points. The SCP character can be used to distinguish between the verbs. For examples, see the verb definitions for ADDD, DIVD, MULD, and SUBD.

TCL-I Interface Requirements

MD1B is the point where TCL attempts to retrieve a verb (first set of contiguous non-blank data in the input buffer) from a user's MD, and validate it as such. If no errors are found, the rest of the data in the input buffer is edited and copied into the IS work space, and control passes to the software specified in the primary mode-id attribute of the verb, or to the PROC software if the data defines a PROC; that is, attribute 1 = PQ{U}.

If the TCL statement contains options (an alphabetic character string and/or numbers enclosed in parentheses at the end of the statement), the options are parsed as described below.

Inputs (user specified):

IB	R	Points one character before the input data

Outputs:

FCB1	D	contain the file access information for the MD
FCB2	D	
IB	R10	points to the SM at the end of the input line
IBEND	S	=IB
BMS	R8	points to the last character of the item-id
BMSEND	S	=BMS
IR	R6	points to the AM following attribute 4 of the verb item, or to the end-of-data AM in attribute 1 if the item defines a PROC
SR4	S	points to the AM at the end of the verb item in the master dictionary

The following are meaningful only if the first two input characters are not PQ (that is, the item is not a PROC). In addition, AFLG through ISBEG are meaningful only if the first two input characters are not PG (reserved for operating system software):

SCP	C	contains the character immediately following P in the verb definition, if present; otherwise contains a blank

CTR0	T	contains the program mode-id specified in the verb definition item (attribute 2)
MODEID2	T	contains the secondary mode-id from the verb definition item (attribute 3), if present; otherwise, contains 0
MODEID3	T	contains the tertiary mode-id from the verb definition item (attribute 4), if present; otherwise, contains 0
OS	R	=OSBEG
AFLG thru ZFLG	B	option flags; AFLG set for A option, BFLG for B option, etc., thru ZFLG for Z (numeric options are stored as shown below)
NUMFLG1	B	set if any numeric option was present
NUMFLG2	B	set if second number was present
D4	D	contains first number (if NUMFLG1 is set)
D5	D	contains second number (if NUMFLG2 is set)
IS	R	=ISBEG
ISBEG	S	points one character before the beginning of the edited input line; characters are copied from the IB, subject to the following rules: • all control characters and system delimiters (SB, SM, AM, VM, SVM) in the input buffer are ignored • redundant blanks (two or more blanks in sequence) are not copied, except in strings enclosed by single or double quote marks • strings enclosed in single quote marks are copied as: SM I string SB • strings enclosed in double quote marks are copied as: SM V string SB • end of data is marked as: SM Z • If SCP is G, no input editing or parsing is done.

When TCL-I has finished parsing the statement, it exits as follows:

- if the verb is not found in the Master Dictionary, or has a bad format, control passes to MD99 in the WRAPUP software, which prints an error message.

Error number	Error type
2	uneven number of single or double quote marks in the input data
3	verb cannot be identified in the MD
30	verb format error (premature end of data or a non-hexadecimal character present in the mode-id)

- if the first verb line contains "PQ{U}, control passes to 0,PROC-I
- otherwise, control passes to the entry point set up in CTR0

TCL-II Interface Requirements

TCL-II should be used whenever a verb requires access to a file, or to all or explicitly specified items within a file. It is entered from the TCL-I software after the verb has been decoded and the primary mode-id has been identified as that of the TCL-II software (that is, the mode-id in attribute 2 of the verb definition is 2).

The input data string to TCL-II consists of the file-name, followed by a list of items, or an asterisk (*) specifying retrieval of all items in the file. If a SELECT, SSELECT, GET-LIST or QSELECT has immediately preceded the TCL-II statement and no item list is present, item-ids are obtained from the select-list instead of from the statement.

TCL-II exits to the software whose mode-id is specified in MODEID2. Typically, programs such as the editor use TCL-II to feed them a set of items which is specified in the input statement. TCL-II uses RMODE to gain control from WRAPUP after each item is processed.

On entry, TCL-II checks the verb definition for a set of option characters in attribute 5; verb options are single characters in any sequence and combination, as shown in Figure 6-3.

If the C option is specified in the verb definition, TCL-II copies to the IS workspace as follows:

```
Item-id:AM: ...Item body ...end
↑                            ↑
ISBEG                        ISEND
```

Inputs (system specified - from TCL-I):

IR	R	points to the AM before attribute 5 of the verb
SR4	S	points to the AM at the end of the verb
MODEID2	T	contains the mode-id of the program to which TCL-II transfers control (assuming no error conditions are encountered)
BMSBEG	S	standard system buffer where the file-name is to be copied, if the F option is present; otherwise where item-ids are to be copied

ISBEG	S	standard system buffer where items are to be copied, if the C option is present

Outputs:

DAF1*	B	set if the U option is specified
DAF2*	B	set if the C option is specified
DAF3*	B	set if the P option is specified
DAF4*	B	set if the N option is specified
DAF5*	B	set if the Z option is specified
DAF6*	B	set if the F option is specified, or if a full file retrieval is specified (item list is * and no F option)
DAF8	B	set if a file dictionary is being accessed, otherwise reset (from GETFILE)
DAF9	B	=0
DAF10	B	set if more than one item is specified in the input data, but not a full file retrieval
IS	R	points one past the end of the file name in the input string if the "F" option is present; points to the SM in the copied item if the "C" option is present, otherwise to the end of the input string
RMBIT	B	set if the file or item is successfully retrieved
FFCB1	D	contain access information for the current file
FFCB2	D	
FCB1	D	contain access information for the current file on the first exit only
FCB2	D	
DFCB1	D	contain access information for the dictionary of the current file
DFCB2	D	

*These elements must not be changed by the next level of software.

The following are meaningful only when the F option is *not* present:

SR0	S	points one prior to the count field of the retrieved item
SIZE	T	contains the item size in bytes (one less than the value of the count field)
SR4	S	points to the last AM of the retrieved item
ISEND	S	if the C option is present, points to the SM terminating the item data
IR	R	if the C option is present, points to the last AM of the retrieved item; otherwise, points to the AM following the item-id on file
RMODE	T	if items are left to be processed, =MD201 (TCL-II re-entry point); otherwise =0 (must not be changed by the next level of software)
XMODE	T	=0

Flags as set up by TCL-I if the input data contains an option string.

Error Conditions

The following conditions cause an exit to the WRAPUP program with the error number indicated:

Error number	Error type
13	data pointer item not found, or in bad format
199	IS work space not big enough when the C option is specified
200	no file name specified
201	filename illegal or incorrectly defined in the MD
202	item not on file; all messages of this type are stored until all items have been processed; items which are on file are still processed
203	no item list specified

WRAPUP Interface

All TCL-interfaced programs must exit to the WRAPUP software when processing is completed. In addition, various entry points to the WRAPUP software provide convenient message printing, if needed.

WRAPUP performs the following functions:

- prints messages and resets HS
- writes abort information, if any, into SYSTEM-ERRORS file
- re-initializes workspaces using the WSINIT subroutine
- closes open print jobs
- cancels interrupts on all virtual devices
- if tape is attached, resets tape buffers and flags
- resets stack counter, STACKPTR, STKFLG
- resets storage registers in the QCB
- resets terminal and printer fields such as page and line counters, page width and depth, and heading and footing text
- releases space to overflow
- resets XMODE field
- resets any read locks or group locks
- resets INHIBITH
- closes remote files, if any, and if specified in the UltiNet setup

  if RMODE is non-zero, it returns to the location specified in RMODE
- if WMODE is non-zero, it performs the processing specified in RMODE
- if mode was EXECUTEd from BASIC, it releases the EXECUTE level and returns to BASIC program at the previous level
- if BREAK and END was used to abort process and if account indicates to do so, it executes to LOGON proc

WRAPUP has several entry points that are used to print messages under different conditions. In all cases, the messages (and parameters) either may be stored in the HS buffer or may be immediately printed. If bit

VOBIT is set, the messages are stored; if VOBIT is zero or if RMODE is zero, they are printed.

If messages are stored in the HS buffer, HSEND points to the next available spot in the buffer. The message string is copied to this location with a SM (segment mark) and an O preceding it and AMs (attribute marks) separating each attribute; the message is terminated with a SM and a Z.

The HS string format is:

```
... _O^ERRMSG item-id^param1^param2^ ...^paramn^_O message _O message ... _Z
                                                                          ↑
                                                                        HSEND
```

Note that HSEND points to the SM, not the Z. This is so that on the next entry, the Z is overwritten with the next O.

On final entry to WRAPUP, the HS buffer is scanned for sequences of SM followed by O, and the messages are printed. (However, if HSEND = HSBEG, no messages are printed.)

If WRAPUP returns via RMODE, the subroutine return stack is cleared, and the workspace pointers and address registers AF, BMS, CS, TS, IB and OB are reset to standard conditions.

WRAPUP Entry Points

The following entry points to WRAPUP may be called by an ENT MDxxx instruction from the user program, depending on the message needed:

Entry point	Description
MD999	terminates processing and returns to TCL-I
Note:	*All entries below eventually enter MD999.*
MD99	enter with REJCTR, REJ0 and REJ1 containing up to three message numbers; no parameters
MD995	enter with C1 containing the message number; string parameter is at BMSBEG thru an AM; typically used to print a message after a file I/O

routine has failed, since the item-id is in the BMSBEG buffer at this point

MD994 — enter with C1 containing the message number; string parameter is at IS thru an AM

MD993 — enter with C1 containing the message number; numeric parameter is in C2; typically used to print a message with a count less than 32,767

MD992 — enter with C1 containing the message number; numeric parameter is in D9; typically used to print a message with a count that may go higher than 32,767

MD1

When WRAPUP is finished, it goes to the entry point to the TCL software is known as MD1.

When MD1 is entered, TCL checks for PROC control, and if this is present, enters the PROC software. If a PROC is not in control (and bit CHAINFLG is zero), an input line is obtained from the terminal, and control passes immediately to MD1B (see TCL Interfaces).

Inputs (user specified):

CHAINFLG	B	If set, terminal input is not obtained (as when chaining from one BASIC program to another)
PQFLG	B	Set to indicate PROC control

XMODE Interface

The XMODE interface is used for processing in the case that a Forward Link Zero condition occurs during a program.

A subroutine mode-id can be placed in XMODE before a pre-incrementing instruction such as MIID or MIIT is executed. This subroutine would gain control if a Forward Link Zero condition is reached.. It can then process the end-of-frame condition and return to the calling routine via a RTN instruction. On return from the subroutine, execution continues at the interrupted instruction.

The XMODE interface has two purposes:

- to allow the standard "Forward Link Zero" system abort message to be replaced with a more formal message
- to attach frames automatically when building a table or string of unknown length

The following is an example of the use of XMODE:

```
MOV   NEXTOVF,XMODE   NEXTOVF is a standard system
CMNT  *               routine that can be used to
CMNT  *               add frames to register 6
 .
 .
MIID  R5,R6,X'C0'     Copy a string until R5 reaches
CMNT  *               a SM
ZERO  XMODE           Reset XMODE
```

The MIID instruction above will automatically generate a subroutine call to NEXTOVF if either register reaches the end of the linked set of frames. If R5 does so, the NEXTOVF subroutine will exit to the debugger to print the Forward Link Zero abort message. However, if register 6 does so, a new frame from the system's overflow space will be linked to the last frame in the linked set addressed by R6. The MIID instruction will then continue as if nothing happened.

The following instructions can be handled by the XMODE interface:

INC register	MIIT	SIT
MCI	MIIDC	SITD
MIC	MIITD	SICD
MII	MIIR[1]	SIDC
MIID	SID	

In the cases where the accumulator is used to count characters, save the accumulator.

If you write your own subroutine, note the following system specified input:

ACF	C	contains the number of the register that caused the Forward Link Zero trap; this should be checked to ensure that the correct register is being handled.

[1]Take care to save R15 in the subroutine.

```
         FRAME 511
* example of user written subroutine
* NEXTOVF could have been used instead
*
         EP    ENTRY0        Entry point is 01FF
         EP    !TRAPSUB      Entry point is 11FF
*
TRAPSUB  DEFM  1,511         Define trap subroutine mode-id
*
ENTRY0   EQU   *
          .
          .
         MOV   TRAPSUB,XMODE  Initialize XMODE with mode-id
          .
          .
         MIITD R5,R6,X'C0'   This may reach end of frame on R6
          .
          .
*  (end of mainline program)
!TRAPSUB EQU   *             Subroutine entry point 11FF
         SRA   R15,ACF       Reference ACF for test
         BCU   R15,6,NOT6    Cannot handle if not R6!
         STORE D4            Save accumulator because
         CMNT  *             subsbelow will destroy it !
         SETDSP R6,ID.DATA.SIZE Set displacement to last
         CMNT  *             byte of this frame, so on
         CMNT  *             return will increment to
         CMNT  *             first byte, next frame
         MOV   R6FID,RECORD  Pickup FID from register
         BSL   ATTOVF        Attach another frame from ovflw.
         BZ    OVRFLW,NOT6   Abort if no more space
         LOAD  D4            Restore accumulator
         RTN   Return to interrupted inst.
NOT6     ZERO XMODE          Kill XMODE; when instruction
         CMNT  *             is re-executed, Debug will
         CMNT  *             be entered to print
         RTN   *             Forward Link Zero message
```

7 Programmer's Reference

This section provides some guidelines for programmers who are new to the Ultimate system, and some examples of assembly programs. The following topics are covered:

- hints
- guidelines for data moves and string conversions
- guidelines for selecting directives and symbol names, and defining a value as a symbol
- background on two's complement binary arithmetic, which is used for all arithmetic operations
- examples of programs and interfaces with system software

Hints

The following describe useful techniques to follow when writing an assembly program.

Setting Up Entry Points

Conventionally, at the beginning of a program, the entry points are defined via EP and NEP instructions; this sequence is called the *branch table*. These unconditional branch instructions allow the program body to be changed and reassembled without affecting the entry points.

The branch table can contain up to 16 entry points. It is recommended that all possible entry points be defined. Use EPs for valid entry points, and NEPs for invalid entry points.

PSYM Elements Reserved User Programs

The following PSYM elements are reserved for user-written assembly routines:

- bits: SB24 to SB35
- characters: none
- tallies: CTR30 to CTR42
- double tallies: none
- triple tallies: none
- storage registers: SR20 to SR24

Note that no address registers are freely available; availability depends on the interface with the system software.

Additional elements may be stored by setting up an additional control block (see Section 3.13).

Making Programs Run Faster

A program runs faster executing sequential instructions than it does executing branches. It is, therefore, recommended that test and branch instructions be constructed so that the most likely result of the test causes the program to continue in sequence; the less likely result should cause the branch.

Handling Terminal Input and Output

It is recommended that the system subroutines READLIN and WRTLIN be used for terminal I/O instead of the subroutines READ@IB, READ@IB, WRITE@OB, and WRITEX@OB (which handle just one character at a time).

Branching to External Frames

If a program needs to branch to an external frame, use the ENT instruction. To execute a subroutine call to an external frame, use a BSL instruction.

Defining Literals

Operands of the form =Dxxxx or =Txxxx should not be used without defining them first in the program. This restriction is made because the intended literal values may not be generated when the program is assembled on software machines.

On a software machine, the assembler may generate literals in-line as part of the object code, rather than at the end of the program. This would render =D4 an undefined symbol unless it is explicitly defined earlier via a DTLY, that is, "=D4 DTLY 4".

In summary, then, no symbol should be used without defining it first. Instead, specify the literal value and let the assembler generate the correct code for that instruction on that particular machine, as in:

```
ADD 10
```

instead of

```
ADD =T10
```

Guidelines for Data Moves and String Conversions

The Ultimate assembly language provides a number of data move and conversion instructions, each designed to be used under certain conditions. These instructions start with the letter M for move.

Programmers need to become familiar with the spectrum of instructions available for data moves and conversions. Once the special features of each instruction are clear, each instance where data needs to be moved or converted can be handled efficiently in an assembly language program.

The source location and destination location of the data should be two different areas. The source data value is unchanged in all cases.

Ultimate file data is stored in ASCII string format; however,the assembly language arithmetic instructions expect binary numbers as operands. Therefore, the appropriate conversion must take place before performing the arithmetic. Conversely, after performing the arithmetic, the resulting binary number must be converted back to its ASCII string equivalent in order to output the value.

All data instructions assume that the programmer knows the type of source data and the desired resulting data type.

- a binary number
- an ASCII decimal number
- an ASCII hexadecimal number

Data stored as a single character is usually retrieved at the virtual address of a register operand R0-R15 (non-incremented address), whereas string data is usually retrieved starting at the virtual address plus 1 (incremented address). A non-incremented address is referred to as AR, and an incremented address as AR+1.

Table 7-1 lists data conversion instructions. Table 7-2 is a similar chart for data move instructions.

Table 7-1. Data Conversion Instructions

MBD	converts binary value into its equivalent decimal ASCII string value, and stores the resulting string, starting at the address +1 of the register operand
MBX MBXN	converts binary value into its equivalent hexadecimal ASCII string value, and stores the resulting string, starting at the address +1 of the register operand
MDB	converts a decimal ASCII character to its equivalent binary value and accumulates it into a symbol operand
MFD MFE	converts decimal ASCII character string value to its equivalent binary value
MFX	converts hexadecimal ASCII character string value to its equivalent binary value
MSDB	converts decimal ASCII string into its equivalent value as a binary number, which remains in the accumulator FP0
MSXB	converts hexadecimal ASCII string into its equivalent value as a binary number, which remains in the accumulator FP0
MXB	converts hexadecimal ASCII character to its equivalent binary value and accumulates it

Table 7-2. Data Move Instructions

MCC	stores the character addressed by the first operand at the location addressed by the second operand
MCI	stores the character addressed by the first operand at the address +1 of the second operand
MIC	copies one character from one location to another location
MII	increments two register operands, then moves the character addressed by the first operand to the location addressed by the second operand
MIID	increments two address register operands, then moves the character addressed by the first operand to the location addressed by the second operand; continues until delimiter encountered
MIIDC	increments two address register operands, then moves the character addressed by the first operand to the location addressed by the second operand; continues until delimiter encountered; counts the number of characters moved
MIIR	increments two address register operands, then moves the character addressed by the first operand to the location addressed by the second operand; continues until address in first register = address in R15
MIIT	copies a specified number of characters from one location to another
MIITD	copies characters from one location to another; terminates the copy at a specified number of characters or when a specified delimiter is encountered.

Guidelines for Defining Symbols

The Ultimate assembly language provides two types of directives for defining symbols: TLY-type directives and DEFx directives.

The TLY-type directive reserves the specified number of bits or bytes in the program frame .

The DEFx directive only defines a symbol name, without reserving storage space in the system.

If you need to reserve space for a literal value as well as defining the symbol name, user the TLY-type directive. If you need to refer to an already existing value, or define a name for a value that will exist elsewhere, use the DEFx directive.

In selecting symbol names, keep in mind the following criteria:

- A symbol name may begin with any character or character sequence except the following:

  $ dollar sign

  # pound sign

  !! double exclamation point

- A symbol name should not begin with a number. The assembler assumes that any operand that starts with a number is a literal number, not a symbol. Consequently, the assembler does not check the PSYM or TSYM file, and the defined symbol value would not be found or used.

Two's Complement Arithmetic Concepts

The Ultimate system performs arithmetic in binary, using the two's complement method. The two's complement method of arithmetic provides the most efficient arithmetic calculations.

The two's complement of a binary number is obtained by changing every 1 in its binary representation to 0, and every 0 to 1, then adding 1 to the result. For example, suppose you want the two's complement of 6. As an 8-bit binary number, 6 is 00000110. Complementing this yields 11111001. Adding 1 to the result yields 11111010.

If a binary number is added to its two's complement, the result (ignoring carry-over) is zero:

```
  00000110
+ 11111010
  --------
(1)00000000
```

Therefore, negative numbers are stored as the two's complements of positive numbers in computer systems such as Ultimate. Two's complement representation allows both positive and negative numbers to be stored in binary. The high-order bit of each negative number is 1, and the high-order bit of each non-negative (positive or zero) number is 0. This means that in an eight-bit field, only seven bits are available to indicate the magnitude of a number.

The range of numbers that can be represented in a field of n bits can be determined by the following formula (where n is greater than 1):

$$-(2^{n-1}) \text{ through } (2^{n-1})-1$$

For example, using n=8:

$$-(2^{8-1}) \text{ through } (2^{8-1})-1 = -128 \text{ through } 127$$

This shows that one byte can contain numeric values that range from -128 through +127. If a number has a value of +128, then it must be contained in a field of at least two bytes. (The value +128 in binary is 10000000. But as an eight-bit value, this is interpreted as a negative number (-128), because the high-order bit is 1. As a sixteen-bit value,

however, +128 is correctly interpreted because its high-order bit is now 0: 0000000010000000.)

Programmers should make sure they use elements large enough to contain all conceivable values of symbols in their programs. Overflow is not detected by the system, and any element having its high-order bit set is treated as a negative number by the arithmetic instructions.

Use these ranges as the basis for selecting the appropriate data type for numeric symbols in a program:

B (bit)	0 to 1
H (half tally)	-128 to +127
T (tally)	-32,768 to +32,767
D (double tally)	-2,147,483,648 to +2,147,483,647
F (triple tally)	-140,737,488,355,327 to +140,737,488,355,327

Examples

This section contains examples of the following:

- TCL-I verb and BASIC program
- TCL-II verb and BASIC program
- conversion subroutine
- setting up heading and footing area
- PROC user exit
- cursor and printer control
- returning a port's logon PCB frame
- returning time in milliseconds
- handling BREAK key activity
- changing width on Wyse terminals

Before you can use an assembly program, you must first assemble it, MLOAD it, and put the verb definition in your master dictionary. Then, to invoke the program, enter the item-ID of the verb definition at TCL. For information on assembling programs, see Chapter 2.

Before you can use a BASIC program, you must first compile it. For more information on BASIC, refer to the *Ultimate BASIC Language Reference Guide.*

TCL-I Verb and BASIC Program

This example shows a simple assembly language program that is run from the system (TCL) level as a TCL-I verb. A BASIC program that performs the equivalent functions is given following the assembly language program.

```
          FRAME 511
*
*
          EP      ENTRY0          Entry point is 01FF
ENTRY0    EQU     *
          SRA     R15,PRMPC
          MCC     C'+',R15        PROMPT '+'
          BBZ     PFLG,LOOP       "P" option not used
          BSL     SETLPTR         PRINTER ON
LOOP      BSL     READLIN         INPUT x
          CMNT    *               Note no initialization for above
          INC     IB              Set on first character input
          BCE     IB,SM,STOP      If null line entered, quit
          DEC     IB              Backup to one before first byte
          MIID    IB,OB,X'C0'     Copy string through SM
          DEC     OB              Backoff SM to set up interface to
          CMNT    *               WRTLIN
          BSL     WRTLIN          PRINT
          CMNT    *               Note no initialization for above
          MOV     IBBEG,IB        Set back to one before first byte
          BSL     CVDIB           Convert numeric to binary
          BZ      T0,LOOP         If zero or non-numeric do nothing
          BLE     T0,140,OK       This test ensures number < 140
          LOAD    140             Else setup to limit to 140 bytes
OK        MOV OB,R15              OBBEG=OB=R15 now WRTLIN reset OB
          MCI     C'+',OB         Move first +, pre-incrementing OB
          DEC     T0              Adjust for above move
          MIIT    R15,OB          Propagate + as many times as value in
          CMNT    *               T0. Note that R15 always pre-increments
          CMNT    *               to a + and OB is always 1 ahead of R15.
          BSL     WRTLIN          PRINT
          B       LOOP            REPEAT
STOP      ENT MD999               Conventional return to TCL via WRAPUP
```

This program is called by a verb defined as follows:

```
     COPYIT
001  P
002  01FF
```

To invoke the program, enter the following at TCL:

```
COPYIT  (P)
```

Use the P option if you want to send the results to the printer.

The following BASIC program is equivalent to the preceding assembly language program.

```
PROMPT '+'
LOOP
      INPUT X
UNTIL X = '' DO
      PRINT X
      IF NUM(X) THEN IF X<=140 THEN PRINT STR('+',X) ELSE
         PRINT STR('+',140)
      END
REPEAT
```

TCL-II Verb and BASIC Program

This example shows an assembly language program that is run from the system (TCL) level as a TCL-II verb. It strips comments from BASIC source file items. The stripped source is written back to the same file, with an item-ID of STRIP- concatenated with the original item-ID.

```
          FRAME 511
*
*
*  This routine is called once as each item is read.
*
          EP      ENTRY0          Entry point is 01FF
STRIPX    EQU     *-1             For SRA instruction below
          TEXT    C'STRIP-'
UCHAR     CHR     C'U'            For MCC below
*
ENTRY0    EQU     *
          MOV     BMSBEG,BMS      Interface to UPDITM start of item-ID
          SRA     R15,STRIPX      Set R15 one before STRIP- string above
          MII     R15,BMS,6       Copy 6 bytes
          MOV     ISBEG,IS        Location of item copied to IS buffer
          MIID    IS,BMS,X'A0'    Concatenate original item-ID, thru AM
          MOV     OSBEG,OS        Scratch location for copy of item
LOOP      INC     IS              To look at first byte of next line
          BCE     IS,SM,ITMEND    If SM found, end of item reached
          BCE IS,C'*',SKIPIT      Asterisk in column one; delete line
          DEC     IS              Backoff first byte for MIID below
          MIID    IS,OS,X'A0'     Else copy rest of line
          B       LOOP            REPEAT
SKIPIT    SID IS,X'A0'            Scan to end of line (AM)
          B       LOOP
TMEND     EQU     *               End of item body reached
          MCI     SM,OS           Interface to UPDITM; end of new item
          MCC     UCHAR,CH8       Interface to UPDITM; update flag
          MOV     OSBEG,TS        Interface to UPDITM; start of new item
          BSL     UPDITM          WRITE
          ENT     MD999           Rtn via WRAPUP to TCL-II for next item,
          CMT     *               if any
```

This program is called by a verb defined as follows:

```
    STRIPIT
001 P
002 2           TCL-II verb
003 01FF
004 0
005 CU          Copy item to IS buffer; verb may update file
```

To invoke the program, enter the following at TCL:

```
STRIPIT  filename  {itemlist}
```

The itemlist contains the names of the items to be stripped.

The following BASIC program is equivalent to the preceding assembly language program. (It assumes that a select statement was specified at TCL before invoking the program.

```
      OPEN 'filename' TO FILE ELSE STOP 201,'filename'
100   READNEXT ID ELSE PRINT 'DONE'; STOP
      I=0
      READ ITEM FROM FILE,ID ELSE
         PRINT 'NOT ON FILE'; GO 100
      END
      LOOP I=I+1; LINE=ITEM<I> UNTIL LINE='' DO
         IF LINE[1,1 = '*' THEN
            ITEM = DELETE(ITEM,I,0,0) ;  *DELETE COMMENTS
         END
      REPEAT
      WRITE ITEM ON FILE,'STRIP-':ID
      GO 100
```

Conversion Subroutine

This example of a conversion subroutine converts a nine-digit stored number to nnn-nn-nnnn Social Security Number format and vice-versa; this routine assumes that the value on entry is valid. Because only R14, R15 and TS are used, no elements are saved.

```
          FRAME 511
*
*       BASIC             RECALL
*Input Conv:  RAW.VAL=ICONV(VAL,'U01FF')   U01FF in V/CONV field
*Output Conv: OUT.VAL=OCONV(VAL,'U01FF')    "  "    "  "
*
          EP      ENTRY0          Entry point is 01FF
ENTRY0    EQU     *
          MOV     TSBEG,TS        Locate start of data
          BBS     MBIT,INPUTC     Process input conversion
*---------------Output conversion-------------------------*
          MOV     TS,R14          Save start
          SID     TS,X'F8'        Scan to any delimiter
          MOV     TS,TSEND        Save this location (TSEND is SCRATCH)
*MAP:
  *       nnnnnnnnnD...scratch space ...   D is Delimiter
  * TSBEG & R14..^     ^....TSEND & TS
  *
          MII     R14,TS,3        Copy 3 numbers;
          MCI     C'-',TS         Add a dash;
          MII     R14,TS,2        Copy 2 numbers;
          MCI     C'-',TS         Add a dash;
          MII     R14,TS,4        Copy 4 numbers;
          MCI     SM,TS
  *       nnnnnnnnnDnnn-nn-nnnnS    D is Delimiter; S is SM
  * TSBEG........^     ^...TSEND  ^...  TS
  *
          MOV     TSBEG,TS        Reset to start
          MOV     TSEND,R14       Start of CONVERTED data
          MIID    R14,TS,X'C0'    Copy back thru SM
  QUIT    DEC     TS              Now on last byte of data
          MOV     TS,TSEND        Correct EXIT interface
*MAP (for output conversion only)
*       nnn-nn-nnnnSn-nn-nnnnS
* TSBEG........^     ^...TS & TSEND
          ENT     CONVEXIT        Conventional exit
```

```
*---------------Input conversion-------------------------*
  INPUTC   EQU     *              Input side; convert nnn-nn-nnnn to 9n
           INC     TS,3           Set one before first "-"
           MOV     TS,R14
           NC      R14            Set on first "-"
*MAP:
*         nnn-nn-nnnnD         D is Delimiter
* TSBEG........^ ^^
*      TS..../ \.....R14
           MII     R14,TS,2       Note 2 bytes copied back "in place"
           INC     R14            Skip over second "-"
           MIID    R14,TS,X'F8'   Copy rest of data to any delimiter
           MCC     SM,TS          Ensure that delimiter is a SM
           B       QUIT
```

Setting Up Heading and Footing Area

This example shows how to set up a heading and footing area, using the HS buffer. It can be added to the program shown in the first example.

```
          FRAME 511
*  This is an example of setting up a heading and footing
*
*
          EP ENTRY0 Entry 01FF
*
HEAD EQU *-1 Heading text
          TEXT   C'THIS IS AN EXAMPLE'
          TEXT   C' OF A HEADING   '
          TEXT   C'      PAGE'
          TEXT   X'FC',C'P',X'FDFF'
          CMNT   *              FC P=page#; FD=newline; FF=end of data
          TEXT   X'FD',C'ULTIMATE ASSEMBLY MANUAL'
          TEXT   X'FF'          To stop MIID !
*---
ENTRY0    EQU    *
          MOV    HSEND,R15      Note use of HSEND, not HSBEG!
          SRA    R14,HEAD       Set R14 one before heading data
          MCI    C'X',R15       Conventional X in HS area
          MOV    R15,PAGHEAD    Initialize PAGHEAD to 1 before heading
          MIID   R14,R15,X'C0'  Copy heading data thru SM
          CMNT   *              Note R14 is on SM in object, above
          MCI    C'X',R15       Conventional X in HS area
          MOV    R15,PAGFOOT    Initialize PAGFOOT to 1 before heading
          MIID   R14,R15,X'C0'  Copy footing data thru SM
          MOV    R15,HSEND      Update ending pointer
          MCI    C'Z',R15       Mark new HS end
*
          SRA    R15,PRMPC
          MCC    C'+',R15 PROMPT '+'
          BBZ    PFLG,NOTLP "P" option not used
          BSL    SETLPTR        PRINTER ON
NOTLP     BSL    PRNTHDR        Initialize and print first heading
*
LOOP      BSL    READLIN        INPUT x
   etc.
```

PROC User Exit

This is an example of a PROC user exit that can be used to perform simple conversions such as Date or Time. In fact, this is a general exit that can call any Ultimate RECALL Conversion.

The PROC exit format is:

```
Uxxxx                   U01FF       U01FF
xconversion.code        ;D2/        :TINV;C;;2
```

where

x : - for output Conversion (similar to OCONV)
; - for input Conversion (similar to ICONV)

The parameter is taken from the current Input Buffer Pointer (IB); in this program it is assumed for simplicity to be the last parameter in the buffer .

```
          FRAME 511
*
*
*
          EP      ENTRY0
*
ENTRY0    EQU     *
          INC     IR              set on : or ; on next line of PROC
          SB      MBIT            for input conversion
          BCE     C';',IR,EP10    yes
          ZB      MBIT            for output conversion (should check
          CMNT    *               for : here!)
EP10      INC     IR              set on first byte of conversion code
          XRR     IR,IS CONV      software requires IS on code
          MOV     TSBEG,SR20      save this
          DEC     IB
          MOV     IB,TSBEG        interface to CONV
          BSL     CONV            process conversion
          XRR     IR,IS           restore registers; CONV has kindly
          CMNT    *               scanned IS (really IR) to an AM
          CMNT    *               thanks
          MOV     SR20,TSBEG      Restore
          ZB      MBIT            later software may expect it zero
          ENT     1,PROC-I        return to PROC
```

Cursor and Printer Control

Users can write their own cursor and printer control routines in assembly language in order to support terminals and printers that are not standard on an Ultimate system. To interface the routine with the Ultimate system, the user then needs to load an item into the TERMDEF file which will define the terminal type code and assembly program mode-id/entry point to the Ultimate system.

To define a cursor or printer control routine, the user may write an assembly routine and then reference it in an item in the TERMDEF file. This item is called a TERMDEF XY mode-id.

After loading the TERMDEF item, the specified terminal or printer code (one character) is valid for use in the TERM and TERMINAL commands, or in a PRINTER command.

The assembly routine will be called by SYSTEM-CURSOR whenever cursor control is specified for that terminal type or whenever printer control is specified for that printer type.

Cursor and printer control routines must conform to the following system interface requirements:

Inputs (user specified):

CTR10	T	contains the screen row number for positioning the cursor; rows are numbered from top to bottom, starting with zero; a value less than zero indicates no row specification.
CTR11	T	contains the screen column number for positioning the cursor; columns are numbered from left to right, starting with zero; a value less than zero indicates no column specification.
R15	R	points one byte before the output area to be used
T4	T	contains relative function number for printer routines, equal to -(n-100) for @(-n); so T4=1 for @(-101), T4=2 for @(-102), and so on.

Inputs (system specified):

For terminal control routines:

TERMTYPE	C	specifies the terminal type

For printer control routines:

Byte 0,QCB	C	character that specifies the printer type

Outputs:

R15	R	Points to the last byte of data generated; or unchanged if the specified function is not defined for this terminal type

A user may place an XY mode-id specification, conforming to the above interface, into any USER-MODE frame. An example of such an XY cursor mode is the following:

```
'ADD-CURSOR  EQU *
             BLZ     CTR10,AC100  BRANCH IF NO LINE ADDRESS SPECIFIED
             MCI     X'0B',R15    SET UP 'VT' FOR ROW ADDRESS
             LOAD    CTR10        GET LINE NUMBER
             DIV     24           REMAINDER (T2)=LINE ADDRESS, 0-23
             INC     R15
             MOV     H4,R15;H0    SET UP LINE ADDRESS CODE
             SB      R15;B1       MAKE CHAR NON-CONTROL CODE
AC100        LOAD    CTR11        GET COLUMN NUMBER
             BLZ     T0,XIT
             MCI     X'10',R15    SET UP 'DLE' FOR COLUMN ADDRESS
             DIV     80           REMAINDER (T2)=COL ADDR, 0-79
             MOV     T2,CTR11     SAVE VALUE
             LOAD    T2
             DIV     10
             MUL     6
             ADD     CTR11        VALUE=X/10*6+X THIS IS CORRECT BCD CODE
             INC     R15
             STORE   R15;H0       SET UP COLUMN ADDRESS CODE
             MOV     R15,R14      PRESERVE THIS FRAME
XIT          RTN
RTN          END
```

The TERMDEF item must have an item-ID and at least two attributes. Comment lines are normally used to describe the terminal or printer type being defined. The general item format is:

```
     item-ID
001 *
002 * description comment lines
003 *
004 REV xx
005 *
006 TERM x    (or PRINTER x)
007 XY Uxxxx
```

The REV number refers to the current revision number of the (Ultimate-supplied) TERMDEF definition format (starting with 1). The TERM or PRINTER code is a 1-character letter code that has not previously been assigned. The 'Uxxxx' in the XY line specifies the mode-id of the cursor control or printer control routine. (See the TERM and PRINTER commands in the System Commands Guide for the standard Ultimate terminal and printer codes).

The following example show a definition item for a terminal (TERM.H) and a printer (PRINTER.H).

```
     TERM.H                      PRINTER.H
001 *                            *
002 * HONEYWELL VIP-7200         * HONEYWELL (NEC) PRINTER
003 *                            *
004 REV 1                        REV 1
005 *                            *
006 TERM H                       PRINTER H
007 XY U0187                     XY U6187
      .
      .
      .
```

To load a TERMDEF item, run the LOAD-TERMDEF program on the SYSPROG account. This makes the terminal or printer type code and its associated control routine available to the Ultimate system and the appropriate commands.

Returning a Port's Logon PCB Frame

The routine below may be used to return the Logon PCB frame of a specified port.

```
          FRAME 534
* Returns the Logon PCB Frame for a given Port
*
*
*
*
          EP      START
          NEP
          NEP
          NEP
          NEP
          NEP
          NEP
          NEP
          NEP
          NEP
          NEP
          NEP
          NEP
          NEP
          NEP
START     MOV     TSBEG,TS        Move raw value
          MSDB    TS              Move to accumulator
          MUL     64              Workspace size is 64
          ADD     2048            Base FID of port 0
          MOV     TSBEG,R15       Reset transient R15
          BSL     MBDSUB          Convert Binary to Decimal
          MOV     R15,TS          R15 points to end of TSBEG
          MOV     TS,TSEND        Conversion demands TS=TSEND
          MCI     SM,R15          Segment mark must end TSBEG
          RTN
```

Returning Time in Milliseconds

The routine below may be used to return the current system time in milliseconds past midnight.

```
          FRAME 540
* Returns Time in Milliseconds
*
*
*
          EP ENTRY0
          NEP
          NEP
          NEP
          NEP
          NEP
          NEP
          NEP
          NEP
          NEP
          NEP
          NEP
          NEP
          NEP
          NEP
SPOT      DEFD   CTR30          Make CTR30 & CTR31 variables
ENTRY0    EQU *
          TIME * Put Time in D0
          DIV    1000           Divide out the milliseconds
          MOV    D1,SPOT        Save the milliseconds
          MOV    TSBEG,R15      Setting up for MBDSUB
          BSL    MBDSUB         Convert Binary Accumulator
          MCI    C'.',R15       Put in a Period
          LOAD   SPOT           Put milliseconds in D0
          BSL    MBDSUB         Convert Binary to millisec
          MOV    R15,TSEND      Marks the last character
          MOV    R15,TS         Place at the End of Data
          MCI    SM,R15         An End-of-String Mark
          RTN
```

Handling BREAK Key Activity

The routine below may be used to take the correct action when a user presses the BREAK key.

```
          FRAME 524
* Deals with Break Key Activity
*
*
*
          EP      ENTRY0
          NEP
          NEP
          NEP
          NEP
          NEP
          NEP
          NEP
          NEP
          NEP
          NEP
          NEP
          NEP
          NEP
          NEP
ENTRY0    EQU     *
          BBS     BREAKKEY,BRK.SET      Was BREAK key hit?
          MOV     TSBEG,TS              Set up to move flag
          MCI     C'0',TS               User didn't hit BREAK
          MOV     TS,TSEND              Set the End Point
          MCI     SM,TS                 Put on the End Point
          RTN
BRK.SET   EQU     *
          MOV     TSBEG,TS              Set up to move Flag
          MCI     C'1',TS               User did hit BREAK
          MOV     TS,TSEND              Set the End Point
          MCI     SM,TS                 Put on the End Point
          ZB      BREAKKEY              Don't let it Break
          RTN
```

Changing Width on Wyse Terminals

The routine below may be used to change the TERM width of a Wyse terminal to 132 or 80 columns.

```
          FRAME 546
* Changes Wyse terminal to 132 or 80 columns
*
*
*
          EP      ENTRY0
          EP      ENTRY1
          EP      ENTRY2
          EP      ENTRY3
          EP      ENTRY4
          NEP
          NEP
          NEP
          NEP
          NEP
          NEP
          NEP
          NEP
          NEP
          NEP
          NEP
          NEP
          NEP
          NEP
W80       EQU     *
          TEXT    X'1B603AFF'
W132      EQU     *
          TEXT    X'1B603AFF'
ENTRY0    EQU     *
          LOAD    79              Term Width
          STORE   TOBSIZE
          BSL     SETTERM
          SRA     R15,W80
          B       WYSE
ENTRY1 EQU *
          LOAD    132             Term Width
```

```
          STORE   TOBSIZE
          BSL     SETTERM
          SRA     R15,W132
WYSE      EQU     *
          DEC     R15
          MIID    R15,OB,X'F8'   Move the Command
          BSL     WRTLIN
          RTN
*
* Find out if Terminal is a Wyse
ENTRY2    EQU     *
          MCI     ESC,OB
          MCI     C' ',OB
          MCI     FF,OB
          BSL     WRTLIN
          RTN
*
* W80 from TCL
ENTRY3    EQU     *
          LOAD    80             Term Width
          STORE   TOBSIZE
          BSL     SETTERM
          SRA     R15,W80
          BSL     WYSE
          ENT     MD999
*
* W132 from TCL
ENTRY4    EQU     *
          LOAD    132            Term Width
          STORE   TOBSIZE
          BSL     SETTERM
          SRA     R15,W132
          BSL     WYSE
          ENT     MD999
```

8 The System (Assembly Language) Debugger

The system (assembly language) debugger allows the programmer to control program execution, to display and change variables, and to set breakpoints.

To call the system debugger, press the terminal's BREAK key.when a process is executing an assembly language program. The debugger is also entered when the system encounters an unrecoverable error.

Note: If the system is executing a BASIC program, the BREAK key calls the BASIC debugger instead of the system debugger. To call the system debugger from the BASIC debugger, use the BASIC debugger command DEBUG:

The system debugger signifies its control by displaying a message of the following form:

```
c f.l
!
```

c code that indicates why the debugger was entered:

- B breakpoint
- E execution step
- I BREAK key pressed
- M modal entry - external BSL
- R modal entry - external RTN

f frame number (FID) in decimal where execution was interrupted

l location; displacement (offset in the FID) in hexadecimal of the instruction that was interrupted

! debugger's prompt character

Entering the Debugger

When a process enters the debugger, for whatever reason, the debugger program is called via a subroutine call to one of the entry points in frame one (FID=1). At this time, if the debug state is to be entered, the system executes a Monitor call to DB.INT. (The debug will not be entered, for example, on a BREAK key entry if the BREAK key is inhibited.)

The kernel sets a flag in the PIB to indicate that the process is in the debug state. In the debug state, whenever the process is activated, the frame at the original PCB FID plus two (that is, the Debug Control Block or DCB) is used as the effective PCB. This frame is permanently assigned as the control block for the debug state.

The DCB has its own set of address registers and all functional elements needed by the debugger. Register 1 (R1, or program counter) in the DCB is always set up to start execution at a specific location in the debugger's software. By switching PCB context, then, the state of the virtual machine is preserved, as the original PCB is saved.

When the debug state is to be exited, another Monitor call is executed to reset the flag in the PIB, and the normal PCB is reinstated. Note that, at this time, the DCB Register 1 (R1) is left pointing to the instruction immediately following the Monitor call, which is the "re-entry" point when the debug state is next entered.

Prior to this, the debugger may have changed the last entry in the real PCB's return stack. This has the effect of unconditionally changing the execution address upon re-entry to regular system processing. The return stack address is normally used by the following debugger commands, which exit the debugger:

G{O}	resumes program execution at the same point as prior to entering the debug state
END or BYE	exits to the system (TCL) level
OFF	logs off the current user account

System Privileges

Users with system privilege levels 0 and 1 (SYS0 and SYS1) have only the following debugger commands available:

```
G       P       END       OFF
```

Users with system privilege level 2 (SYS2) have access to all debugger commands, except DI (disable). The DI (disable) debugger command is used prevent users with SYS2 privileges from using the system debugger commands and is invoked from the SECURITY or SYSPROG account (if set equal to SECURITY).

Inhibiting the BREAK Key

Normally, when the BREAK key is pressed, execution of the current program is terminated and the process enters the debugger. However, the BREAK key on a particular line can be inhibited by invoking the following system command:

```
BREAK-KEY-OFF
```

When the BREAK key is inhibited, that line is prevented from entering the debugger via the BREAK key. To enable the BREAK key, use the following system command

```
BREAK-KEY-ON
```

For more information on these commands, please refer to the *Ultimate System Commands Guide*.

Program Aborts

When a process is executing, certain conditions can cause it to enter the debug state as a program abort, not a user request. Typically, these are unrecoverable error conditions, although artificial calls to the debugger can be forced by the kernel for special processing conditions. All unrecoverable error conditions cause a message similar to the following to be displayed on the terminal attached to the process:

```
message
Abort @ f.l
```

- f — frame number (FID) in decimal where trap occurred
- d — location; displacement (offset in the FID) in hexadecimal of the instruction where the trap occurred

In addition, for register-related error conditions (traps marked with an asterisk in Table 8-1), the number of the register causing the trap is displayed, for example:

```
Forward Link Zero; Reg = 4
Abort @ 511.1
```

On entry to the debugger, the error number associated with the abort condition is stored in byte 0 of the user's PCB (that is, R0;H0).

In the debug state, the debugger often displays the current program counter location as an address.

Table 8-1 lists the debugger traps (abort conditions) invoked by virtual and their related entry points. Table 8-2 lists the debugger traps (abort conditions) invoked by the kernel and their related entry points.

Note: *In the case of a Forward Link Zero trap, if the exception subroutine tally XMODE is non-zero, the debugger transfers control to the subroutine whose mode-id is stored in XMODE. The subroutine can perform such error handling as necessary, and when it executes a RTN instruction, control returns to the instruction which originally caused the trap condition. See Chapter 6 for more information about the XMODE interface.*

Table 8-1. Traps (Aborts) (1 of 2)

Entry Point	Message	Description
0	Illegal Opcode	An undefined assembly instruction has been found.
1	Rtn Stack Empty	A RTN (return) instruction has been executed when there were no entries in the subroutine return stack.
2	Rtn Stack Full	A BSL or BSLI instruction (subroutine call) has been executed when there were already ten entries in the stack.
3	Referencing Frame Zero	An address register has a zero FID.
4*	Crossing Frame Limit	Either an address register with a virtual address in the unlinked mode has been incremented or decremented beyond the boundaries of the frame, or a relative address computation (base+offset) resulted in an address that was beyond the boundary of the frame addressed by the register.
5*	Forward Link Zero	An incrementing instruction (for example, INC r) has caused the register to go beyond the end of the linked frame set.
6*	Backward Link Zero	A decrementing instruction (for example, DEC r) has caused the register to point to a frame before the beginning of the linked set.

* the affected register number is stored in the half tally ACF for use by the debugger.

Table 8-1. Traps (Aborts) (2 of 2)

Entry Point	Message	Description
8	Referencing Illegal Frame	A register has a frame number that is beyond the allowable limits for this disk configuration.
11	Halt	A HALT instruction has been executed.

Table 8-2. Kernel Traps

Entry Point	Description
9	A disk error has occurred when the process generated a frame-fault. The disk error handler is invoked to log the message in the SYSTEM-ERRORS file
10	The BREAK key is pressed on the user's terminal
13	A message has been transmitted to the process by another process (via the TCL MSG command). The debugger saves the context via the mechanism described earlier, and transfers control to the message printer

Summary of Debugger Commands

The following is a summary of the system debugger commands.

Command	Description
{f}addr{;window} {f}/symbol{;window}	Direct data display, where f — format code addr — location window — number of bytes to be displayed symbol — name of symbol from PSYM
{f}*addr{;window} {f}*symbol{;window} Rr R.r	Indirect data display, where f — format code addr — location window — number of bytes to be displayed symbol — symbolic name of element to display r — register number
ADDD n1 n2	Adds decimal numbers n1 and n2.
ADDX x1 x2	Adds hexadecimal numbers x1 and x2.
A/symbol	Displays the address of a symbol.
Baddr	Adds the address to the execution breakpoint table; up to four addresses can be set.
BYE	Returns to TCL; same as END but preserves the breakpoint and trace tables. (set by B, E, N, M, T, and F commands)
D	Displays the breakpoint and trace tables.
DI	Disables debugger for all lines.
DIVD n1 n2	Divides decimal number n1 by n2.
DIVX x1 x2	Divides hexadecimal number x1 by x2.
DTX n	Converts decimal n to hexadecimal.

Command	Description
E{n}	Sets the execution step to n; if n is 0 or omitted, clears execution step.
END	Returns to TCL; same as BYE, but clears breakpoint and trace commands (set by B, E, N, M, T, and F commands)
Fn,m	Uses frame with FID m in place of FID n during execution of instructions.
G{addr}	Continues execution at address specified, or at point of interruption if addr is omitted
K{addr}	Kills specific breakpoint entry, or all entries if addr is omitted.
line feed or escape	Continue execution; equivalent to G command; included for convenience.
Lfid	Displays link fields of frame specified.
M	Toggle to turn on and off modal execution trace.
MULD n1 n2	Multiplies decimal number n1 by n2.
MULX x1 x2	Multiplies hexadecimal number x1 by x2.
N{n}	Sets delay counter to n; if n is omitted, sets counter to 0; inhibits debug entry until after n breaks or steps have occurred
OFF	Logs user off system.
P	Toggle to suppress/allow terminal output.
SUBD n1 n2	Subtracts decimal number n2 from n1.
SUBX x1 x2	Subtracts hexadecimal number x2 from x1.

Command	Description
T{f}addr{;window} T{f}*symbol{;window} T{f}/symbol{;window}	Traces location specified, where f — format code addr — location window — number of bytes to be displayed symbol — symbolic name of element to display
U{{f}addr{;window}} U{f}*symbol{;window} U{f}/symbol{;window	Untrace; clears trace table entry, or all entries if addr is omitted.
XTD x	Converts hexadecimal x to decimal.
Yaddr;window Y/symbol Y*symbol Y	Sets a data breakpoint, where addr — location window — number of bytes to be displayed symbol — symbolic name of element to display
>TCL.stmt	Executes TCL statement and returns to debug.
>>	Invokes new TCL level.
<<	Returns to previous TCL level.
<	Returns to current TCL level.

Address Specification and Representation

When the debugger displays an address, the frame number (FID) is always in decimal and the location (displacement) is always in hexadecimal. If the displayed address is from a register which is in linked mode, a plus sign (+) precedes the frame number as an indicator.

`!C*ISBEG +1200.B  .TEST=`	ISBEG addresses frame 1200 (decimal), displacement B (hexadec imal; decimal 11). The plus sign (+) preceding 1200 indicates that ISBEG is in linked addressing mode.
`!C*TSBEG;16 1189. .20 JUN 1990_..=`	TSBEG addresses frame (decimal) 1189, displacement 0, in unlinked addressing mode as indicated by the lack of a plus sign (+) preceding 1189.

There are several ways to specify a virtual address in a debugger command. Typically, a frame number (FID) and a displacement or location are required. Each number may be entered either in decimal or in hexadecimal notation for convenience.

Displaying Data in the Debugger

To display data, specify the location using one of the following forms at the debugger prompt (!):

Syntax

{f}addr{;window}
{f}/symbol{;window}
{f}*addr{;window}
{f}*symbol{;window}
Rr
R.r

f — format code; may be one of the following:

- C — display data in ASCII character format; non-printable characters are displayed as a period (.); System delimiters are displayed as follows:

SB	[
SVM	\
VM	]
AM	^
SM	_

- X — display data in hexadecimal format.
- I — display data in integer format; window must be 1, 2, 4, or 6.

addr — direct address; may be in one of the following forms:

- f.l — FID f in decimal; location l in hexadecimal.
- f,l — FID f in decimal; location l in decimal.
- .f.l — FID f in hexadecimal; location l in hexadecimal.
- .f,l — FID f in hexadecimal; location l in decimal.

A period (.) preceding a value indicates a hexadecimal (base 16) number. A comma (,) is used preceding a decimal location, but not preceding a decimal FID.

indirect address; may be in one of the following forms:

- Rr — register **r**, in the range R0-R15.
- R.r — register **r**, in the range R.0-R.F (hexadecimal).

window number of bytes to be displayed; the window is separated from the address by a semicolon (;); may be in one of the following forms:

- ;n display n bytes in decimal.
- ;m,n display n bytes starting m (decimal) bytes before specified address.
- ;.z,n display n bytes starting z (hexadecimal) bytes before specified address.
- ;.x display x bytes; x is a hexadecimal number.
- ;m.x display x bytes starting m (decimal) bytes before specified address.
- ;.z.x display x bytes starting z (hexadecimal) bytes before specified address.
- ;tn immediate symbol window; references immediate data at the specified address. t is the symbol type-code; n is the offset; typically used to display bits; for example,

  ```
  ;Bn
  ```

  addresses the nth bit displaced off the previous address.

/ specifies direct reference to the contents of the symbol following.

* specifies an indirect reference to the data addressed by the symbol, which may be only of types R (address register) or S (storage register).

symbol symbolic name of element to be displayed

R specifies an indirect reference to data using a register

r register number; if preceded by . (period), specifies number is in hexadecimal format

If format and window are not specified for a symbol, the defaults are assumed by the debugger. For example, if the symbol is of type D (double tally), the format defaults to I (integer) and window to 4.

If the FID is omitted, the user's PCB is assumed. For example, the notation 100 is the virtual address of location X'100' in the user's PCB. If format and window are unspecified, the last previously used values are reused

`100.66;B0`	displays the high-order bit of location 66(hexadecimal) of frame 100 (decimal).
`C1234.7F;100`	displays 100 bytes in ASCII format, starting at location x'7F' (127 decimal) in frame 1234 (decimal).
`/ABIT`	displays contents of ABIT (0 or 1).
`/D0`	displays contents of D0 as an integer.
`X/D0`	displays contents of D0 in hexadecimal.
`/ISBEG`	displays contents of ISBEG.
`C*ISBEG;300`	displays 300-byte indirect character string starting at location addressed by ISBEG.
`X*R15;10,20`	displays 20 bytes in hex, starting 10 bytes *before* the location addressed by R15.

Changing Data in the Debugger

After the debugger displays data, it prompts with an equals sign (=) rather than a !.

When = is displayed, you have the option of terminating the operation, continuing the display, changing the displayed data, or a combination of these actions.

- To terminate the display and return to the ! prompt, press RETURN.
- To continue the display to the next forward window, press the <CTRL-N> keys or <LINEFEED>. <CTRL-N> causes a new line and a new address to be displayed before the data; <LINEFEED> continues display on the same line

  To continue the display to the previous window, press <CTRL-P>. This causes a new line and a new address to be displayed before the data.
- To change data, enter the new data. The replacement value may be entered in bit, character, hexadecimal or integer mode, and except for bit replacement mode, does not have to correspond to the format that has been displayed.

  The replacement value can be entered in one of the following modes:

Entry Format	Mode	Notes
'data....	character	A single quote mark followed by the data; there is no trailing quote. The character string entered cannot contain control characters and its length need not correspond to the size of the window. Up to 100 characters may be entered.
.data....	hexadecimal	A period followed by the data. The hexadecimal string entered must be an even number of hex digits and its length need not correspond to the size of the window. Up to 100 digits may be entered.

n	integer	A decimal number. The displayed window must be 1, 2, 4, or 6 only.
xxxxx...	bits	x=0 or 1, that, is a string of bits. This form is valid only when a bit is displayed.

The replacement mode entry is terminated by pressing one of the following:

RETURN (carriage return) changes data and returns to ! prompt.

<CTRL-N> changes data and displays next window on new line.

<LINEFEED> changes data and displays next window on current line.

<CTRL-P> changes data and displays previous window.

A Command - Display Address

The A command displays the address of a specified symbol.

Syntax

A/symbol

Description

!A/CTR24 2625.24	Displays address of symbol CTR24.

Arithmetic Commands

Several arithmetic commands may be specified to perform computations at the debugger level.

Syntax

ADDD n1 n2
ADDX x1 x2
DIVD n1 n2
DIVX x1 x2
DTX n1
MULD n1 n2
MULX x1 x2
SUBD n1 n2
SUBX x1 x2
XTD x1

n1 decimal number

n2 decimal number

x1 hexadecimal number

x2 hexadecimal number

Description

These commands may be used for arithmetic computation at the debug level and are identical to their TCL verb equivalents:

ADDD adds decimal values n1 and n2.

ADDX adds hexadecimal values x1 and x2.

DIVD divides decimal value n1 by n2.

DIVX divides hexadecimal value x1 by x2.

DTX converts decimal value n1 to hexadecimal.

MULD multiplies decimal values n1 and n2.

MULX multiplies hexadecimal values x1 and x2.

SUBD subtracts decimal value n2 from n1.

SUBX subtracts hexadecimal value x2 from x1.

XTD converts hexadecimal value x1 to decimal.

For more information, refer to the *Ultimate System Commands Guide.*

B Command - Breakpoint Specification

The B command sets an execution breakpoint at a specified location. If the process reaches the location specified as a breakpoint, the debugger is entered.

Syntax

Baddr

addr direct address; may be in one of the following forms:

- f.l FID f in decimal; location l in hexadecimal.
- f,l FID f in decimal; location l in decimal.
- .f.l FID f in hexadecimal; location l in hexadecimal.
- .f,l FID f in hexadecimal; location l in decimal.

A period (.) preceding a value indicates a hexadecimal (base 16) number. A comma (,) is used preceding a decimal location, but not preceding a decimal FID.

Description

Up to four breakpoints can be set.

If zero is specified as the location, a breakpoint is set for every location in the specified frame. The causes a break on any entry to the frame. This can be useful if you are not sure where in a specific frame execution may begin.

To clear the breakpoints, see the K command.

Note: On Ultimate software implementations, the B commands may not work except on instructions assembled with a label on the source code line. For more information, see the section in Chapter 2, Assembly on Software Machines.

`!B511.3`	causes a breakpoint to be set in frame 511, location 3
`!B511.0`	causes a breakpoint on every location in frame 511; any entry to the frame causes a break

Bye Command - Exiting the Debugger

The BYE command is used to terminate debugger execution and return to the primary TCL level.

Syntax

BYE

Description

The BYE command terminates execution and returns to TCL at the lowest (LOGON) level. If the process had been executing at a higher TCL level, all such levels are released.

This command is equivalent to the END command.

D Command - Display Tables

The D command is used to display tables set by the B, F, and T commands.

Syntax

D

Description

The tables are displayed, similar to the following:

```
 Brk tbl n.  n.  n.  n.
 Trc tbl n.  n.  n.  n.
*Trc tbl n.  n.  n.  n.
 Frm tbl n. = n.
 Chg tbl n.
```

n location

DI Command - Disabling the Debugger

The DI (disable) debugger command is used to prevent access to the system debugger commands.

Syntax

DI

Description

The DI (disable) debugger command is invoked from the SECURITY or SYSPROG account. It prevents users with SYS2 privileges from using the debugger commands normally available to them.

Invoking the DI command is known as *disabling the debugger*, and is a method of improving system integrity by preventing accidental or deliberate change of data, via the debugger.

Once disabled, the debugger can be enabled via the same DI command. An alternative is to use the SECURITY-STATUS command. For more information on the SECURITY-STATUS command, please refer to the *Ultimate System Commands Guide*.

E Command - Execution Step.

The E command specifies the number of program lines to be executed when the program is resumed. After the specified number of lines is executed, an execution break occurs.

Syntax

```
E{n}
```

n number of lines to execute before returning to debugger control; if zero or not specified, execution stepping is turned off

Description

The E command is typically used in the form E1, which single-steps execution. The forms E or E0 turn off the execution step.

When an execution break occurs, the current location is displayed, similar to the following:

```
E f.l
```

E indicates an execution break has occurred

f frame number (FID) in decimal where execution was interrupted

l location; displacement (offset in the FID) in hexadecimal of the instruction that was interrupted

Note: On Ultimate software implementations, the E commands may not work except on instructions assembled with a label on the source code line. For more information, see the section in Chapter 2, Assembly on Software Machines.

`!E1`	single steps execution
`!E10`	Ten lines are executed before an execution break occurs

END Command - Exiting the Debugger

The END command is used to terminate debugger execution and return to the primary TCL level.

Syntax

END

Description

The END command terminates execution and returns to TCL at the lowest (LOGON) level. If the process had been executing at a higher TCL level, all such levels are released.

This command is equivalent to the BYE command.

F Command - Changing Frame Assignments

The F command is used specify to a frame that is to be temporarily substituted for another frame.

Syntax

F{n,m}

n frame number, in decimal, that is to be reassigned

m frame number, in decimal, to be used whenever frame **n** is referenced

Description

Once this command has been executed, the debugger monitors every frame change as the process executes, and any external BSL or ENT instruction to frame **n** is internally modified to go to frame **m**.

Only one frame reassignment at a time can be in effect for a process.

If no parameters are specified with the F command. frame reassignment is turned off.

The F command is very useful when debugging a program because it can be used to temporarily reassign an executable frame number for the user's process only.

In practice, the user can modify an existing program, changing the FRAME statement in the program before reassembly. After the program has been reassembled, the object can be MLOADed into the temporary frame. The F command can then be used to redirect the process to the modified frame.

For example, when debugging a program normally assigned to frame 420, the user changes the FRAME statement in the source program to 511 (a temporary location), assembles and MLOADs it. The following debugger command then routes all execution transfers from frame 420 to frame 511:

```
!F420,511
```

Note: If a frame is reassigned, breakpoints must be set using the reassigned frame number, not the original one.

G Command - Resume Execution

The G command may be used to resume execution of the process in effect when the debugger was entered, if possible.

Syntax

G{addr}

addr location of instruction

Description

The G command with no address is used to continue execution at the point of interruption.

The G command with an address unconditionally changes the point of execution.

Pressing the LINEFEED (down arrow) or ESC key is equivalent to entering G with no address.

If the debugger was entered via one of the system traps (that is, a program abort), the G command with no address is not valid. In that case, one of the following commands must be used:

END

OFF

BYE

G addr

Caution: *The G command should be used with an address only if the programmer knows a location where execution can safely resume. The debugger is not aware of which addresses are valid for restarting execution.*

K Command - Clear Breakpoints

The K (kill) command is used to clear breakpoints.

Syntax

K{addr}

addr address of breakpoint to be cleared

Description

If no address is specified, all breakpoints are cleared.

L Command - Display Link Fields

The L (link) command is used to display link fields.

Syntax

Laddr
L*symbol

addr frame number (FID) whose links are to be displayed; to specify the FID in hexadecimal, precede the number with a period

***symbol** should be an address register or storage register only

Description

The link fields are displayed in the following form:

```
nncf : forward.link   backward.link : npcf
```

nncf number of next contiguous frames
npcf number of previous contiguous frames

All four fields are displayed in decimal.

To display link fields of frame f in hexadecimal, use the display data command with the X format code as follows:

nncf	Xf.1;1
forward link	Xf.2;4
backward link	Xf.6;4
npcf	Xf.A;1

M Command - Modal Execution Trace.

The M command is a switch that turns modal execution tracing on or off.

Syntax

M

Description

If the modal trace is on, the debugger is entered whenever an ENT, ENTI, external BSL, external BSLI, or RTN from external subroutine instruction is executed. That is, execution is interrupted whenever the program frame changes.

Local subroutine calls and RTNs cannot be traced.

N Command - Delay Entry to Debugger.

The N command is used to delay entry to debugger for a specific number of breakpoints or steps.

Syntax

```
N{n}
```

n number of execution breaks or steps to bypass; if not specified, no breakpoints or steps are skipped

Description

Values that are being traced and location of execution breakpoints are displayed at every breakpoint, although no break actually occurs.

`!E10`	Execute ten instructions, then break.
`!N9`	Skip first nine breaks.
`!G`	100 instructions are executed before debugger gets control. Every ten instructions, a message is printed (because of the E10), but execution continues.

P Command - Toggle Terminal Display

The P command is used to toggle the terminal display on and off.

Syntax

P

Description

The P command is a toggle switch that turns the terminal display on or off. It is identical to the TCL P command.

For more information, refer to the *Ultimate System Commands Guide.*

T Command - Trace Data

The T (trace) command is used to set up traces for direct and indirect addresses.

Syntax

T{f}addr{;n}
T{f}/symbol{;n}
T{f}*symbol{;n}

f	format code
addr	location
n	number of bytes to be displayed; display is limited to 127 bytes
symbol	symbolic name of element to display
/	direct address
*	indirect address; if specified, symbol must be a register or storage register

Description

The T (trace) command can be used to specify up to four traces using a direct address and four traces using an indirect address.

Once the trace is set, on every subsequent debugger entry (including system traps), the traced data is displayed automatically.

The U command can be used to cancel traces.

U Command - Delete Traces

The U (untrace) command is used to delete traces.

Syntax

U{{f}addr{;n}}
U{f}/symbol{;n}
U{f}*symbol{;n}

f	format code
addr	location
n	number of bytes to be displayed; display is limited to 127 bytes
symbol	symbolic name of element to display
/	direct address
*	indirect address; if specified, symbol must be a register or storage register

Description

If no parameters are specified, all traces are deleted.

Y Command - Data Breakpoint

The Y command sets a data breakpoint. Up to two breakpoints can be set.

Syntax

Y{addr;n}
Y/symbol
Y*symbol

addr	location
n	number of bytes to be displayed; display is limited to 127 bytes
symbol	symbolic name of element to display
/	direct address
*	indirect address; if specified, symbol must be a register or storage register

Description

If no parameters are specified, all breakpoints are deleted.

If the address is not a symbol, the number of bytes to be displayed is required.

The Y command adds an entry to the data trace table. The address (symbolic or direct) specified is monitored and the debugger is entered when the value changes.

>>, <<, >, < Commands - Changing TCL Levels

There are four commands that can be used to change TCL levels:

>>	suspends the current level and creates a new TCL level. If you are using the default TCL prompt, it changes to indicate the TCL level.
<<	pops one TCL level
>	exits the debugger and returns to TCL at the current level.
<cmd	suspends current program; executes "command" as a system command; returns to debugger at current level

At any given time, a process executes at one of several levels of TCL. As needed, it can change levels. For example, the EXECUTE statement in BASIC sets up a new level (pushes a level) to process a specified system (TCL) command, then returns to the previous level (pops a level) where the BASIC program is executing.

Pushing and popping levels can be done from the debugger by using the TCL level pushing features, which are described in the *Ultimate System Commands Manual*, or by using the >> and << commands at the debugger prompt.

If END or BYE is typed in the debugger, all TCL levels are popped and you are returned to the primary TCL level.

9 Monitor Calls (MCALs)

Monitor calls (MCALs) are used to perform functions that are hardware dependent or architecture dependent. These instructions call routines in the kernel.

The Monitor calls documented here primarily fall into the categories below.

Monitor calls that affect the PIB

PIB.AND
PIB.ATL
PIB.OR
PIB.PEEK
PIB.POKE

Buffer Management Calls

FAKE.RD
FAKE.WT
FORCE.WT
LOCK.FRAME
UNLOCK.FRAME

Control Calls

ALARM.CLOCK
CLOCK.CANCEL
INT.CANCEL
LOCK
RQM
SLEEP
WAIT

Asynchronous I/O Calls

PERIPH.READ
PERIPH.WRITE
CLEAR.INP
TEST.INP

Normally asynchronous channel I/O to the process' own channel is handled by the READ and WRITE instructions, which handle single byte transfers. PERIPH.READ and PERIPH.WRITE may also be used to transfer data via the process' own channel or via a different one, and give a greater degree of control than READ and WRITE. (See READ and WRITE in the external Assembly manual; see READN in Section 1).

MCAL 1	COM.CTL
MCAL 2	MTBF
MCAL 3	LINK.CNT
MCAL 4	MTB
MCAL 5	CMD.STAT
MCAL 6	CMD.MAXFID
MCAL 7	CMD.FAKE.WT
MCAL 8	CLEAR.REDUN
MCAL 9	GET.ID
MCAL C	TL.READ
MCAL D	PANEL
MCAL E	START.IO.PIB
MCAL F	WARM.DUMP
MCAL 10	DB.ENT
MCAL 11	DB.LV
MCAL 12	PIB.AND
MCAL 13	PIB.OR
MCAL 14	FAKE.RD
MCAL 15	FAKE.WT
MCAL 16	WAIT
MCAL 17	QUERY
MCAL 18	PIB.PEEK
MCAL 19	PIB.POKE
MCAL 1A	N.GET.ID
MCAL 1C	ALARM.CLOCK
MCAL 1D	CLOCK.CANCEL
MCAL 1E	INT.CANCEL
MCAL 1F	VMCAL
MCAL 20	FRM.UNLOCK
MCAL 21	FRM.LOCK
MCAL 22	SLEEP or SLEEP:
MCAL 24	DISK.ERR
MCAL 25	FORCE.WRITE
MCAL 26	SET.TIME
MCAL 27	TIME or GET.TIME
MCAL 28	RQM
MCAL 29	LOCK
MCAL 2A	LOCK
MCAL 2B	PIB.ATL
MCAL 2C	DSABL.DSK
MCAL 2D	QUEUE.READ
MCAL 2E	VMB.INPUT
MCAL 2F	RTC.CALIB
MCAL 30	TEST.INP
MCAL 31	VOPT.OR
MCAL 32	VOPT.AND
MCAL 33	CLEAR.INP
MCAL 34	PERIPH.WRT.ONE
MCAL 35	PERIPH.RD.ONE
MCAL 36	CLR.OUT
MCAL 37	PIB.XPCB
MCAL 38	DISK.STAT
MCAL 39	WRITE.WAIT
MCAL 3A	RCV.LEN
MCAL 3D	SET.FL.DEN
MCAL 3E	XFER.CLOCK
MCAL 3F	SET.BATCH.TM
MCAL 40	PERIPH.RD
MCAL 41	PERIPH.WRT
MCAL 44	VMS.SPOOL
MCAL 45	VMS.TAPE
MCAL 46	VMS.OFF
MCAL 47	VMS.MSG
MCAL 48	PC.MSG
MCAL 49	FAKE.READ
MCAL 4A	RFLAGS.CLR
MCAL 4B	FLAGS.SET
MCAL 4C	PIBNO.POKE
MCAL 4D	PIBNO.PEEK
MCAL 4E	SYSTEM
MCAL 4F	TIME.OUT
MCAL 50	VT.KILL
MCAL 53	BREAK.PORT
MCAL 54	MCK.STAT

VTERM.CTL:R	X'01'	VOPT.OR	X'31'
MTBF	X'02'	VOPT.AND	X'32'
LINK.CNT	X'03'	CLEAR.INP	X'33'
MTB	X'04'	PERIPH.WRT1:R	X'34'
CLEAR.REDUN	X'08'	PERIPH.RD1:R	X'35'
GET.ID	X'09'	CLR.OUT	X'36'
BISYNC.IOR	X'0A'	PIB.XPCB.DWN:R	X'37'
TL.READR	X'0C'	PIB.XPCB.UP:R	X'37'
PANEL	X'0D'	PIB.XPCBR	X'37'
START.IO.PIB	X'0E	DISK.STATR	X'38'
WARM.DUMP	X'0F'	WRITE.WAIT	X'39'
DB.ENT	X'10'	RCV.LEN	X'3A'
DB.LVR	X'11'	SET.FL.DEN	X'3D'
PIB.AND	X'12'	XFER.CLOCKR	X'3E'
PIB.OR	X'13'	SET.BATCH.TM	X'3F'
FAKE.WT	X'15'	PERIPH.RDR	X'40'
WAIT	X'16'	PERIPH.WRTR	X'41'
QUERY	X'17'	NDISK.STATR	X'42'
PIB.PEEK	X'18'	VMS.SPOOL	X'44'
PIB.POKE	X'19'	VMS.TAPE	X'45'
N.GET.ID	X'1A'	VMS.OFF	X'46'
ALARM.CLOCK	X'1C'	KERNEL.MSGR	X'47'
CLOCK.CANCEL	X'1D	VMS.MSGR	X'47'
INT.CANCEL	X'1E'	PC.MSG	X'48'
VMCALR	X'1F'	FAKE.READ	X'49'
FRM.UNLOCKR	X'20'	RFLAGS.CLR	X'4A'
FRM.LOCKR	X'21'	RFLAGS.SET	X'4B'
SLEEPXR	X'22'	PIBNO.POKE	X'4C'
DISK.ERRR	X'24'	PIBNO.PEEK	X'4D'
FORCE.WRITE	X'25'	SYSTEMR	X'4E'
SET.TIME	X'26'	TIME.OUT	X'4F'
GET.TIME	X'27'	VT.KILL	X'50'
TIME	X'27'	BREAK.PORT	X'53'
RQM	X'28'	INIT.TR	X'56'
PIB.ATL	X'2B'	INIT.FKR	X'5B'
DSABL.DSK	X'2C'	INIT.RESYNC	X'5C'
QUEUE.READ	X'2D'	MCK.STAT	X'54'
MB.INPUTR	X'2E'	TD.READR	X'55'
RTC.CALIB	X'2F'	DISKETTE:R	X'5A'
TEST.INP	X'30'		

How to Use MCAL Information

This section contains information which is strictly confidential to the Ultimate Corp. It should be read only by Ultimate Research and Development personnel who will be working with the Assembler and/or an Ultimate Kernel, and should not be divulged outside the R & D Department.

An MCAL source statement has the following general syntax:

Syntax

MCAL Rr,nn,m

Rr register number (for example, R0, R8). In many MCALs this parameter is not used, but still must be specified . May be expressed in decimal (2, 15, etc.) or hex (X'2', X'F', etc.).

nn sequential number of this MCAL (for example, 2, 14, 26). May be expressed in decimal (4, 26, etc.) or hex (X'4', X'1A', etc.).

m class number, which must be 11 (X'B'). The 'm' value is assembled into the opcode's third byte, second nibble (e.g., AB); the first nibble is the sub-opcode identifier, which is typically 'A', but may also be 8 or 9.*

Note: All MCALs have been assigned names. Writers of virtual code are hereby requested to use the named form. The numbered form may be used in the interim, while you are developing a name.

What is syntax if I use name, not number??

Where did the following come from?? I found it in the document on the page with RFLAGS.SET.

A new monitor call was introduced in TERMIO7 to do fast multibyte terminal output. The format is:

```
OUTSTR Reg, StorReg, ByteCount
```

Reg	points one byte before the first character to print
StorReg	points to the last character to print
ByteCount	contains the number of bytes output by the routine (output interface only).

Hopefully other implementations will be able to make use of this monitor call.

ALARM.CLOCK - MCAL 1C

The ALARM.CLOCK monitor call enables alarm clock request for specified time.

Input:

D0 amount of time until expiration, in milliseconds

Output:

none

Data Structure:

modifies clock request block and links

Description

A timer is initialized and started. This may later be used by the WAIT and QUERY MCALs.

CLEAR.INP - MCAL 33

The CLEAR.INP monitor call resets (clears) terminal input buffer.

Input:

none

Output:

none

Data Structure:

modifies terminal input buffer pointers.

Description

None needed.

CLOCK.CANCEL - MCAL 1D

The CLOCK.CANCEL monitor call resets an alarm clock request.

Input:

none

Output:

none

Data Structure:

modifies clock request block and links.

Description

A previously set timer, if present, is cleared.

CLR.OUT - MCAL 36

The CLR.OUT monitor call clears terminal output.

Input:

T1 PIB number (negative means self)

Output:

none

Data Structure:

modifies the terminal output buffer.

Description

This cancels any terminal output and clears the output roadblock.

DB.ENT - MCAL 10

The DB.ENT monitor call enters the software debugger.

Input:

none

Output:

debug bit (X'0080') in PIB word zero set

debug bit (X'0080') in PCB ACF field set

Data Structure:

no data structures modified

Description

The two debug bits are set and the process is detached. When the process is next activated, the firmware will use the DCB instead of the PCB. (only for firmware??)

DB.LV - MCAL 11

The DB.LV monitor call exits software debugger.

Input:

Rr	byte zero of the PCB

Output:

debug bit (X'0080') in PIB word zero	cleared
debug bit (X'0080') in PCB ACF field	cleared

Data Structure:

BT entry for Rr (PCB)	set write-required

Description

The two debug bits are cleared and the process is detached. When the process is next activated, the firmware will use the PCB instead of the DCB. (only for firmware??)

DISK.ERR - MCAL 24

The DISK.ERR monitor call reports disk error from 'stack' to virtual process.

Input:

Rr	address where disk error information should be put; this must be at least 32 bytes before the end of a frame.

Output:

Rr	32 bytes of disk error information are copied

Data Structure:

modifies the buffer table (write-required bit set).

Description

This copies the kernel's disk error table to a virtual frame and clears the table. If the table was empty, zeroes are copied to the virtual frame.

DISK.STAT - MCAL 38

The DISK.STAT monitor call reports disk I/O statistics.

Input:

Rr address of buffer to copy statistics to

Output:

none

Data Structure:

no data structures modified

Description

Copies disk statistical counts to Rr.

**More to be supplied??

DSABL.DSK - MCAL 2C

The DSABL.DSK monitor call disables the disk set.

Input:

none

Output: :

none

Data Structure:

no data structures are modified

Description

This is only used by the Ultimate 6000/7000 offline monitor. It switches the interrupt level of each disk from the disk level to the virtual device level.

FAKE.RD - MCAL 14

The FAKE.RD monitor call does a fake disk read (as if R15 has been frame faulted); that is, it assigns the buffer, but does not do the read.

Input:

R15FID frame number to be fake-read.

Output:

none

Data Structure - the following are modified:

buffer table

age links

disk I/O queues (on some systems)

Description

The memory map is modified so that the FID of an available buffer in main memory is changed to the FID to be fake-read; but the frame is not read from disk. This may be used when the virtual process knows the frame number of a needed frame, but the contents of the frame are irrelevant. For example, the overflow space manager may get an available frame number from a table of FIDs. Since the frame will be initialized after this, the data in the disk frame will not be used.

If the frame is already in main memory, there is no operation. If there are no available buffers in main memory (an unlikely occurrence), the frame may actually be read in from disk as usual, at the discretion of the kernel programmer.

Caution: *The FAKE.RD monitor call is obsolete. In most cases, the* FAKE.READ *monitor call (MCAL 49) should be used instead. In order for the FAKE.RD MCAL to actually save any processing time, R15 must be set up in such a fashion that it does not attach (which would cause an automatic frame fault and would read the disk frame into main memory). One way this can be done is by setting up an SR with the required FID and a displacement of 1, then moving the SR to R15 just before the FAKE.RD. Also, some software systems force all registers to be always attached. Therefore, R15 would be attached, and the frame would be automatically read before the MCAL is executed by the kernel.*

FAKE.READ - MCAL 49

The FAKE.READ monitor call does a fake read. The FID is in D0.

Input:

D0 frame to fake read

Output:

none

Data Structure - the following are modified:

buffer table

age links

disk I/O queues (on some systems)

Description

The memory map is modified so that the referenced frame is in memory, but it is not read from disk. It is assumed that the frame will be initialized after this.

FAKE.WT - MCAL 15

The FAKE.WT monitor call does a fake write by zeroing the buffer table write-required bit of the buffer pointed to by R15, thus making the buffer available.

Input:

R15FID he frame number to be fake-written.

Output:

none

Data Structure - the following are modified:

buffer table

age links

Description

Normally, when data in a frame stored on disk has been modified, the write-required flag in the status byte of the buffer is set to indicate that the frame needs to be written back to disk at some time. If the virtual process knows that the frame does not need to be written to disk, a FAKE.WT can be used to clear the write-required flag. For example, a frame returned to the overflow pool does not need to be saved.

If the frame is not in main memory, no operation is performed.

FORCE.WRITE - MCAL 25

The FORCE.WRITE monitor call forces write of designated frame by enqueueing.

Input:

Rr address of frame to write (should r be 15??)

Output:

none

Data Structure:

modifies disk queues.

Description

The purpose of this MCAL is to checkpoint a particular frame, and schedule it to be written to disk. Normally, when data in a frame has been modified, the write-required flag in the status byte of the buffer is set to indicate that the frame needs to be written to disk at some time (that is, when the buffer has aged or the system is quiescent). But, if the frame is used frequently, it may tend to stay at the top of the Buffer Age Queue, which may delay writing it to disk.

When data in a frame is particularly sensitive, the FORCE.WRITE ensures that the frame whose FID is in R15 will be written as soon as possible.

If the frame is not write-required, this is a NOP.

If the frame is being written, the program counter is backed up to the beginning of the MCAL and an RQM is executed. This makes the MCAL wait for previously started writes to complete.

Caution: *Not all kernels support multiple force-writes being active concurrently, due to a limited number of disk queue entries.*

FRM.LOCK - MCAL 21

The FRM.LOCK monitor call locks designated frame in memory.

Input:

Rr	address of any byte within the frame

Output:

H4	high byte of 24-bit byte address of byte 0 of the frame
H5	middle byte of 24-bit byte address of byte 0 of the frame

Data Structure:

modifies the buffer table.

Description

This locks the frame in main memory whose FID is in R15, and returns the memory address in an encoded format where the frame was locked.

While locked, the frame is not considered for replacement. This may later be used by an VM or MV instruction. The frame will stay in that memory location until one of the following:

- the FRM.UNLOCK MCAL is used
- the system is restarted

If the frame is modified by virtual software, the modification will be reflected on disk.

FRM.UNLOCK - MCAL 20

The FRM.UNLOCK monitor call unlocks designated frame.

Input:

Rr address of any byte within the frame

Output:

none

Data Structure:

modifies buffer table - the corelock bit is cleared

Description

None needed.

GET.ID - MCAL 9

The GET.ID monitor call gets device-id for device number in T0.

Input:

T0	virtual device number (used as index into Virtual Device Table)

Output:

T0	0 if no device is configured on that channel; otherwise, the device ID number is returned.
T1	channel address

Data Structure:

no data structures are modified

Description

If no device exists for that virtual device number, T0 is zeroed.

The virtual device number is assumed to be in Ultimate 6000/7000 format; that is, there is one number for the input function of a device and another number for the output function. On 6000/7000 systems, one of the channel address bits shows whether the channel is for reception or transmission. Virtual code uses this bit to select only one of the pair of entries associated with a given device.

INT.CANCEL - MCAL 1E

The INT.CANCEL monitor call resets virtual interrupt request.

Input:

T0 virtual device number; if X'FFFF', all virtual devices for the process.

Output:

none

Data Structure:

modifies the virtual device table.

Description

The interrupts from any I/O operations that the process had started (on the specified device or on any device) are cancelled. Interrupts from the devices may still come to the kernel, but the virtual device table is marked in a way that causes the interrupts to be ignored.

The virtual device number is assumed to be in Ultimate 6000/7000 format; that is, there is one number for the input function of a device and another number for the output function. On 6000/7000 systems, one of the channel address bits shows whether the channel is for reception or transmission. Virtual code uses this bit to select only one of the pair of entries associated with a given device.

LINK.CNT - MCAL 3

The LINK.CNTO monitor call counts forward and backward age links.

Input:

none

Output:

T0	number of buffers in age links, counting in the forward direction.
T1	number of buffers in age links, counting in the backward direction.

Data Structure:

no data structures are modified

Description

This may be used by virtual programs to test the integrity of the age links. The count should be the same from each direction. The count is the number of buffers available for paging.

LOCK - MCAL 29

The LOCK monitor call locks a system resource with an ELSE clause.

Input:

Rr	address of a tally to be used as a lock.

Output:

tally at Rr	may contain PIB number + 1, with the bytes in swapped order
INHIBITH	incremented if the resource is obtained or already owned by the process.

Data Structure (the following are modified):

clock request block and links

SNU links

buffer table write-required flag is set

Description

This is used to try to lock a system resource. If the resource is already locked by another process, a branch instruction that is assembled immediately following the MCAL is taken.

In assembly language, this is coded as follows:

```
LOCK REGISTER,LABEL
```

The assembler uses the label to construct the branch instruction.

The tally pointed to by the register is used as a lock. Zero is the unlocked condition. When the tally is locked, it contains the PIB number plus one of the process that set the lock (owns the resource).

The PIB number + 1 is stored with the high and low order bytes reversed as follows, so that the LSI systems run more efficiently:

Rr

low order byte of PIB number + 1	high order byte of PIB number + 1

The firmware in the LSI systems enables the kernel to efficiently process the first byte pointed to by Rr, but not the second byte. On the LSI systems, or any systems that cannot support more than 254 processes, the high order byte is always zero, and the lock may be treated as a byte by the kernel. Virtual software should always consider the lock to be a tally whenever it reads it or initializes it.

If the tally contains a zero, the PIB number + 1 is put into the tally in byte-swapped order and execution resumes after the branch instruction that follows the MCAL.

If the tally is not zero, the lock is owned by the PIB represented by the tally. The execution path depends on the following:

- If the current process already owns the lock, execution is the same as if the tally were zero.
- If the PIB that owns the lock is roadblocked by disk, terminal I/O, or a trap, or it is active in the other processor of a dual processor system, the LOCK MCAL is treated as if it were an RQM. In this case, the virtual program counter is left pointing to the branch instruction, so the ELSE clause will be taken after the RQM.
- If the PIB that owns the lock is able to be activated, an attempt is made to activate it by doing what the PIB.ATL MCAL does. The virtual program counter is left pointing at the branch instruction that follows the MCAL.

LOCK - MCAL 2A

The LOCK monitor call locks a system resource.

Input:

Rr	address of tally to be used as a lock

Output:

tally at Rr	contains process number + 1, with the bytes in swapped order
INHIBITH	incremented if the resource is obtained or already owned by the process.

Data Structure (the following are modified):

clock request block and links

SNU links

buffer table write-required flag is set

Description

This is used to try to lock a system resource. If the resource is already locked by another process, a branch instruction that is assembled immediately following the MCAL is taken.

In assembly language, this is coded as follows:

```
LOCK REGISTER
```

The tally pointed to by the register is used as a lock. Zero is the unlocked condition. When the tally is locked, it contains the PIB number plus one of the process that set the lock (owns the resource).

The PIB number + 1 is stored with the high and low order bytes reversed as follows, so that the LSI systems run more efficiently:

Rr	low order byte of PIB number + 1	high order byte of PIB number + 1

The firmware in the LSI systems enables the kernel to efficiently process the first byte pointed to by Rr, but not the second byte. On the LSI

systems, or any systems that cannot support more than 254 processes, the high order byte is always zero, and the lock may be treated as a byte by the kernel. Virtual software should always consider the lock to be a tally whenever it reads it or initializes it.

If the tally contains a zero, the PIB number + 1 is put into the tally in byte-swapped order and execution resumes after the branch instruction that follows the MCAL.

If the tally is not zero, the lock is owned by the PIB represented by the tally. The execution path depends on the following:

- If the current process already owned the lock, execution is the same as if the tally were zero.
- If the PIB that owns the lock is roadblocked by disk, terminal I/O, or a trap, or it is active in the other processor of a dual processor system, the LOCK MCAL is treated as if it were an RQM and the virtual program counter is backed up to the beginning of the MCAL.
- If the PIB that owns the lock is able to be activated, an attempt is made to activate it by doing what the PIB.ATL MCAL does. The virtual program counter is backed up to the beginning of the MCAL instruction.

MTB - MCAL 4

The MTB monitor call moves a frame (FID) to bottom of age links.

Input:

Rr points to the buffer to move

Output:

none

Data Structure:

modifies age links

Description

This MCAL makes the specified buffer the first one to be used to satisfy frame faults or fake reads.

MTBF - MCAL 2

The MTBF monitor call moves a buffer to the bottom of the age links.

Inputs (user specified):

T0 T buffer number

Output:

none

Data Structure:

modifies age links

Description

This MCAL makes the specified buffer the first one to be used to satisfy frame faults or fake reads.

N.GET.ID - MCAL 1A

The N.GET.ID monitor call gets the device ID.

Input:

T3	virtual device number

Output:

T0	device ID or zero if device number is too big
T1	channel address
T2	buffer number, in a format appropriate for VIOLD instructions

Data Structure:

no data structures are modified.

Description

The virtual device number is assumed to be in Ultimate 6000/7000 format; that is, there is one number for the input function of a device and another number for the output function. On 6000/7000 systems, one of the channel address bits shows whether the channel is for reception or transmission. Virtual code uses this bit to select only one of the pair of entries associated with a given device.

PANEL - MCAL D

The PANEL monitor call invokes the remote panel processor Ultimate 6000/7000 systems.

Input:

T0 port number to use

Output:

none

Data Structure:

no data structures are modified

Description

On Ultimate 6000/7000 systems, this starts the PANEL debugger program running on the port specified by T0.

PC.MSG - MCAL 48

On PC implementations, the PC.MSG monitor call performs a file transfer.

Input:

read or write flag, drive number, filename, directory

Output:

none

Data Structure:

no data structures are modified

Description

On a PC, the PC.MSG monitor call performs a read or write (as specified by the flag). On all systems other than a PC, this is an illegal opcode.

On PC implementations, the disk space is partitioned for DOS and Ultimate operating systems. This MCAL calls a DOS routine to cross the partition and move the file data in or out of virtual memory. One item at a time is transferred, in 1024-byte increments.

PERIPH.RD - MCAL 40

The PERIPH.RD monitor call performs a multi-byte peripheral read.

Input:

T0	PIB number
H2	count
H3	flags
Rr	address to start copying bytes to

Output:

T1	number of bytes copied

Data Structure:

modifies the terminal input buffer.

Description

This copies multiple bytes from the designated PIB's terminal input buffer to the location pointed to by the register.

If T0 refers to a PIB other than the one making the monitor call, the TRAP roadblock is set.

If the input buffer is empty, zero is put in T1 and execution resumes at the instruction after the MCAL.

Characters are removed from the terminal input buffer to the locations pointed to by Rr until one of the following occurs:

- the input buffer is emptied
- the end of the frame that Rr points to is reached
- the number of characters specified by the count in H2 have been moved
- if virtual B24 (the low bit of H3) is set, a control character is moved (control characters have an ASCII value less than X'20')

PERIPH.RD.ONE - MCAL 35

The PERIPH.RD.ON monitor call reads data byte from another line's port.

Input:

T0	PIB number
Rr	address to copy byte to

Output:

byte at Rr	copied from designated input buffer

Data Structure (the following are modified):

terminal input buffer

clock request block and links

Description

This removes one byte from the designated PIB's terminal input buffer and copies it to virtual space. If T0 refers to a PIB other than the one making the monitor call, the TRAP roadblock is set.

If the PIB's terminal input buffer is not empty, the first byte in it is removed and copied to where Rr points.

If the input buffer is empty, the program counter is set back to the MCAL instruction and an RQM is done; this causes the process to wait for about 100 ms. before re-executing the MCAL.

PERIPH.WRT - MCAL 41

The PERIPH.WRT monitor call performs a multi-byte peripheral write.

Input:

T0	PIB number
T1	input flags
Rr	address of string to write

Output:

T1	number of characters actually written

Data Structure:

modifies the terminal output buffer

Description

If T0 refers to a PIB other than the one making the monitor call, T1 is checked. If bit X'0100' is *not* set, the TRAP roadblock is set; it the bit is set, the process is not trapped.

The string of bytes pointed to by Rr is output (or buffered for output) on the port specified by T0, as if the WRITE instruction had been executed for each byte. The end of the string is marked by the first of the following conditions that is encountered:

- no more characters can be buffered by the terminal I/O interface; typically, this occurs because the output buffer is full
- a segment mark is found (the segment mark is not transmitted)
- the string reaches the end of its frame

The number of characters sent is stored in T1. If this is not the entire string, it is the responsibility of the virtual code to send more later.

Execution resumes with the next instruction.

PERIPH.WRT.ONE - MCAL 34

The PERIPH.WRT.ONE monitor call writes data byte to another line's port.

Input:

T0	PIB number
T1	input flags
Rr	address of byte to write

Output:

none

Data Structure (the following are modified):

terminal output buffer

clock request block and links

Description

If T0 refers to a PIB other than the one making the monitor call, T1 is checked. If bit X'0100' is *not* set, the TRAP roadblock is set; it the bit is set, the process is not trapped.

The specified byte is output as if the specified process had executed a WRITE instruction. Control returns immediately (that is, there is no wait for the output to complete). If the byte could not be written, the program counter is backed up to the MCAL instruction and an RQM is done; this causes the process to wait about 100 ms. before re-executing the MCAL.

PIB.AND - MCAL 12

The PIB.AND monitor call is used to AND bits in PIB word zero.

Input:

T0	tally with mask containing bits to be ANDed with word zero of the PIB
T1	PIB line number; if negative, specifies caller's own PIB

Output:

T0	contains resulting PIB word zero

Data Structure:

no data structures are modified

Description

This is typically used to clear roadblocks.

PIB.ATL - MCAL 2B

The PIB.ATL monitor call activates a process by adding it to the top of PIB links.

Input:

T0 PIB number to be activated

Output:

none

Data Structure (the following are modified):

clock request block and links

SNU links.

Description

The SLEEP roadblock is cleared. If the CRB was in the links (meaning an alarm clock or sleep or RQM had not yet expired), the CRB is removed from the links and the semaphore flag is set (this causes a -3 to be returned by WAIT or QUERY). The PIB is moved to the top of the SNU links. The current process is detached and the target PIB is activated is there are no roadblocks.

The R0,00,15 form is an obsolete synonym. Support for it will be withdrawn in the future. (Has it??)

PIB.OR - MCAL 13

The PIB.OR monitor call is used to OR bits in the PIB word zero.

Input:

T0	tally with mask containing bits to be ORed with word zero of the PIB
T1	PIB line number; if negative, specifies caller's own PIB

Output:

T0	Contains resulting PIB word zero

Data Structure:

no data structures are modified

Description

The PIB.OR monitor call is typically used to set roadblocks.

PIB.PEEK - MCAL 18

The PIB.PEEK monitor call returns the value of a specified word in a PIB.

Input:

T1	PIB line number; if negative, specifies caller's own PIB
H4	PIB word number

Output:

T0	value of the specified word

Data Structure:

no data structures are modified

Description

Note: Be aware that the kernel or firmware may change the values of certain PIB words at any time.

PIB.POKE - MCAL 19

The PIB.POKE monitor call replaces a specified word in a PIB.

Input:

T1	PIB line number; if negative, specifies caller's own PIB
H4	PIB word number
T0	value to poke into the PIB

Output:

none

Data Structure:

depends on which word is modified

Description

Note: Be aware that the kernel or firmware may change the values of certain PIB words at any time. Care must be taken to not damage the state of the operating system by inappropriate use of PIB.POKE.

PIB.XPCB - MCAL 37

The PIB.XPCB monitor call changes the PCB FID in PIB.

Input:

H2:H1:H0	the new PCB FID
H3	flags
Rr	some address in the new PCB

Output:

none

Data Structure:

no data structures are modified

Description

The PIB.XPCB monitor call does the following:

- sets bit in position X'20' of ACF of current PCB
- moves B30 of old PCB (bit position X'40' of H3) to PIB debug bit
- moves new PCB FID from the accumulator to the PIB
- zeros bit in position X'20' of ACF of new PCB (for warmstart)
- deactivates the process

The next time the process is activated, the new PCB is used.

QUERY - MCAL 17

The QUERY monitor call processes a Query or Query virtual interrupt.

Input:

none

Output:

T0	0 or greater	device number
	-1	no entry for this device
	-2	clock timeout)
	-3	semaphore timeout
	-4	no clock timeout and outstanding I/O
T1	if a virtual device interrupted, this is the number of interrupts received; otherwise, zero (0).	

Data Structure (the following are modified):

SNU links

virtual device table

clock request block and links

Description

The QUERY monitor call is the same as WAIT (MCAL 16) except that QUERY never suspends the process.

If any interrupts have been received from any device that had an I/O operation started by this process, the following occurs:

- the virtual device number of the interrupting device is placed in T0
- the number of interrupts received in placed in T1
- if the device has any I/Os still outstanding, the count of outstanding I/Os is reduced by the number received; otherwise, the device is marked inactive.
- the virtual process is resumed.

Note: If the process has interrupts outstanding for multiple devices, only the interrupts for one device will be returned and there is no priority ordering of devices.

If no device interrupts have been received, the following occurs:

- if an alarm clock that the process had armed has expired,
  -2 is returned in T0
  the clock request block is marked inactive
  the process is resumed
- if an alarm clock that the process had armed was cancelled because another process attempted to activate this process
  -3 is returned in T0
  the clock request block is marked inactive
  the process is resumed
- if no alarm clock has expired or was cancelled and there was at least one outstanding I/O operation
  -4 is returned in T0
  the process is detached
- if no alarm clock had expired or was cancelled, and there were no outstanding I/O operations
  -1 is returned in T0
  the process is detached

QUEUE.READ - MCAL 2D

The QUEUE.READ monitor call queues a read (frame fault).

Input:

D0 frame to read

Output:

none

Data Structure:

modifies disk queues.

Description

If the frame is in memory, this is a NOP; otherwise, a disk read is started, but the PIB is not marked disk roadblocked.

RCV.LEN - MCAL 3A

The RCV.LEN monitor call gets residual UltiNet range.

Input:

none

Output:

IO.BIT	set to indicate success; zero to indicate failure
T0	if successful, contains length if unsuccessful, specifies which I/O code to the controller (1-6) failed

Data Structure:

no data structures are modified

Description

This is used on Ultimate 6000/7000 systems to get the residual range from the UltiNet controller. It is equivalent to a series of six VIO instructions, but faster.

RFLAGS.CLR - MCAL 4A

The RFLAGS.CLR monitor call clears bits in RFLAGS.

Input:

T0	mask containing bits to clear
T1	PIB line number; if negative, specifies caller's own PIB

Output:

T0	new value of RFLAGS

Data Structure:

no data structures are modified

Description

This is the method of enabling and disabling XON/XOFF, typeahead, etc. Each bit that is set in the mask is cleared in RFLAGS.

If the PIB number is negative, the currently executing PIB is used.

(where are the RFLAGS bits documented??)

RFLAGS.SET - MCAL 4B

The RFLAGS.SET monitor call sets bits in RFLAGS.

Input:

T0	mask containing bits to set
T1	PIB line number; if negative, specifies caller's own PIB

Output:

T0	new value of RFLAGS

Data Structure:

no data structures are modified

Description

This is the method of enabling and disabling XON/XOFF, typeahead, etc. Each bit that is set in the mask is set in RFLAGS.

If the PIB number is negative, the currently executing PIB is used.

RQM - MCAL 28

The RQM monitor call releases time quantum.

Input:

none

Output:

none

Data Structure:

modifies clock request block and links.

Description

This deactivates a process for approximately 100 milliseconds. Any alarm clock that was set is not disturbed.

??Opcode 4000A9 is an obsolete synonym. Support for this opcode will be withdrawn in the future.

RTC.CALIB - MCAL 2F

The RTC.CALIB monitor call changes the RTC (real-time clock) tick count from 6 to 5 to accommodate systems that operate on 50 hertz power.

Input:

none

Output:

none

Data Structure:

no data structures are modified

Description

The RTC.CALIB monitor call is used only on LSI-based systems. It changes the RTC refresh value in the kernel to 5. (Five ticks equal 100 ms.)

SET.BATCH.TM - MCAL 3F

The SET.BATCH.TM monitor call sets the batch time limit.

Input:

T0	if non-zero, new time limit if zero, current value is not changed

Output:

T0	current time limit

Data Structure:

no data structures are modified

Description

This sets or returns the maximum number of seconds that must pass with no interactive jobs running before batch jobs are allowed to use all of memory.

If T0 is non-zero, it contains the new value, in seconds. If T0 is zero, the current value is not changed. In either case, the (new) current value is returned in T0.

SET.FL.DEN - MCAL 3D

The SET.FL.DEN monitor call sets the density of floppy disk; this is available on PC implementations only.

Input:

none

Output:

none

Data Structure:

no data structures are modified

Description

On a PC, the SET.FL.DEN monitor call causes floppy diskette parameters to be set properly for the type of diskette inserted into the drive.

The SET.FL.DEN monitor call is a NOP on any system other than a PC.

SET.TIME - MCAL 26

The SET.TIME monitor call sets the system time and date.

Input:

D0	new system time, in milliseconds since midnight
T2	new system date

Output:

none

Data Structure:

no data structures are modified

Description

Note: This is a NOP on the VAX because the timekeeping is done by VMS.

??Opcode 40FFAA is an obsolete synonym. Support for this opcode will be withdrawn in the future.

SLEEP - MCAL 22

The SLEEP monitor call puts process to sleep for specified time.

Input:

D0	time to wake up (in milliseconds after midnight)
Rr	address of byte to clear

Output:

Rr	tally pointed to by the register is zeroed

Data Structure:

modifies the clock request block and links

Description

The tally pointed to by Rr is zeroed. This is in case the spooler, which must clear a lock in synchronization with being deactivated, is executing the sleep. Ordinarily, the register is set to a scratch tally.

If the SLEEP opcode is used, the register defaults to R0 (the first tally of the PCB is scratch). To specify another register, use the SLEEP: opcode. ??

The process is deactivated until one of these occurs:

- the wakeup time is reached.
- the break key is pressed.
- another process wakes it up, either by the PIB.ATL or PIB.AND (not recommended) MCAL, or by trying to set a lock that the process has set.

START.IO.PIB - MCAL E

The START.IO.PIB monitor call starts MLCP I/O on the line (PIB port).

Input:

T0 line number of PIB port to start I/O on.

Output:

none

Data Structure:

terminal input buffer initialized

Description

This initializes and starts input on the terminal attached to the specified PIB.

TEST.INP - MCAL 30

The TEST.INP monitor call tests for characters in terminal input buffer.

Input:

T1	PIB number

Output:

T0	number of characters in input buffer

Data Structure:

no data structures are modified

Description

This may send an XON if the buffer is empty and typeahead is enabled. (when and why would it??)

TIME - MCAL 27

The TIME monitor call gets the system time and date.

Input:

none

Output:

T2	system date
D0	system time in milliseconds since midnight

Data Structure:

no data structures are modified

Description

??Opcode 4000AA is an obsolete synonym. Support for this opcode will be withdrawn in the future.

TL.READ - MCAL C

The TL.READ monitor call is used by the transaction logger for special READs.

Input:

Rr specifies the address of the byte (tally) to clear.

Output:

Rr byte pointed to by the register is zeroed

Data Structure:

PIB may be removed from SNU links

BT entry for frame pointed to by RA is marked write-required

Description

The Transaction logger uses this. If there are any characters in the terminal input buffer (typeahead buffer), the virtual process is allowed to resume. Otherwise, the process is detached and removed from the SNU links.

does the register point to a byte or a tally?? does it matter??

VMCAL - MCAL 1F

The VMCAL monitor call executes kernel code.

Input:

Rr address of kernel code

Output:

none

Data Structure:

modification depends on the kernel code.

Description

This is used to execute kernel code (native CPU instructions) that resides in a virtual frame. This is presently only supported on Ultimate 6000/7000 systems and is intended to be used for special purpose patches.

VMS.MSG - MCAL 47

The VMS.MSG monitor call performs a file transfer.

Input:

Rr	address of byte zero of the frame to be copied

Output:

T0	zero (0) to indicate success

Data Structure:

no data structures are modified

Description

On a VAX, this copies data between Ultimate memory and VAX memory. On all systems other than a VAX, this is an illegal opcode.

To use this MCAL, the virtual process fills a frame with the following information:

Bytes	Use
0-1	number of bytes of meaningful data in frame
2	function code
3	zero (0); this is a first-time flag that is altered by the monitor
4-511	data to transfer

If byte 3 of the frame contains a zero (0), the following occurs:

- as many words as indicated by bytes 0-1 are copied from the frame to a corresponding I/O buffer in VMS memory
- the SLEEP roadbloack for the Ultimate process is set
- the program counter is backed up to the MCAL instruction
- the process is detached
- event flag 2 (file transfer attention) is set for the corresponding VMS process
- if byte 2 contains a 4 or 5, the VMS process is awakened

- a one (1) is written into byte 3 of the frame and the frame's write-required flag is set

If byte 3 of the frame does not contain a zero and the function code is negative, the VMS process has finished and the following occurs:

- As many words as indicated by the byte count in bytes 0-1 of the corresponding VMS I/O buffer are copied into the frame.
- The frame's write-required flag is set.
- T0 in the accumulator is zeroed to indicate success.
- The Ultimate process is resumed.

If byte 3 of the frame does not contain a zero and the function code is not negative, this means the operator pressed the BREAK key, etc., and the following occurs: (what else besides break key??)

- The PC is backed up and the process is put to sleep again

(what is the PC?? what does backed up mean??)

VMS.OFF - MCAL 46

The VMS.OFF monitor call is a VMS Logoff.

Input:

none

Output:

none

Data Structure (the following are modified):

terminal I/O table

SNU links

clock request block and links

Description

On a VAX, this returns to VMS the control of the terminal attached to the PIB making the MCAL. On all systems other than VAX, this is a NOP.

VMS.SPOOL - MCAL 44

The VMS.SPOOL monitor call passes a spool file to VMS for printing.

Input:

Rr first FID of spool file

Output:

none

Data Structure:

no data structures are modified

Description

On a VAX, the VMS.SPOOL monitor call does the following:

- causes the VMS detached process to create a VMS file
- moves Ultimate spool file data into the VMS file
- then sends it to the VMS job controller for printing

This call is a NOP on all systems other than VAX.

VMS.TAPE - MCAL 45

The VMS.TAPE monitor call issues tape commands to various tape drivers in the VAX.

Input:

For label I/O:

Rr	points to label buffer in tape control block

For non-label I/O:

Rr	R0

Output:

none

Data Structure:

no data structures are modified

Description

On a VAX, the VMS.TAPE monitor call gives information on the type of I/O, size of the transfer, which device to use, etc. This information is used to build I/O packets to issue to VMS tape drivers.

This call is a NOP on all systems other than VAX.

VOPT.AND - MCAL 32

The VOPT.AND monitor call is used to AND the virtual option flag.

Input:

T0	mask containing bits to be ANDed with virtual option flag

Output:

T0	new options value

Data Structure:

no data structures are modified

Description

This is used to change certain global system parameters.

what are the virtual option flags?? where are they documented??

VOPT.OR - MCAL 31

The VOPT.OR monitor call is used to OR the virtual option flag.

Input:

T0	mask containing bits to be ORed with virtual option flag

Output:

T0	new options value

Data Structure:

no data structures are modified

Description

This is used to change certain global system parameters.

WAIT - MCAL 16

The WAIT monitor call suspends the process until a virtual interrupt occurs.

Input:

none

Output:

T0	0 or greater	device number
	-1	no entry for this device
	-2	clock timeout)
	-3	semaphore timeout
T1	if a virtual device interrupted, this is the number of interrupts received; otherwise, zero (0).	

Data Structure (the following are modified):

SNU links

virtual device table

clock request block and links

Description

If any interrupts have been received from any device that had an I/O operation started by this process, the following occurs:

- the virtual device number of the interrupting device is placed in T0
- the number of interrupts received in placed in T1
- if the device has any I/Os still outstanding, the count of outstanding I/Os is reduced by the number received; otherwise, the device is marked inactive
- the virtual process is resumed

Note: *If the process has interrupts outstanding for multiple devices, only the interrupts for one device will be returned and there is no priority ordering of devices.*

If no device interrupts have been received, the following occurs:

- if an alarm clock that the process had armed has expired,
  -2 is returned in T0
  the clock request block is marked inactive
  the process is resumed
- if an alarm clock that the process had armed was cancelled because another process attempted to activate this process,
  -3 is returned in T0
  the clock request block is marked inactive
  the process is resumed
- if no alarm clock had expired or was cancelled, and there were no outstanding I/O operations,
  -1 is returned in T0
  the process is detached
- if no alarm clock had expired or was cancelled and there is at least one outstanding I/O operation, and no interrupts have been received (a -4 condition in MCAL QUERY),
  the process is removed from the SNU links.
  the program counter is backed up to the WAIT MCAL
  the process is detached

  This prevents activation of the process until an interrupt occurs.

WARM.DUMP - MCAL F

The WARM.DUMP monitor call can both warmstart the system and dump memory to tape..

Input:

T0 code for desired action; valid codes are:

- X'F511' flush memory to tape and warmstart system
- X'DEAD' flush memory to disk and halt system
- X'DC10' flush memory to disk and halt system

Output:

none

Data Structure:

As part of flushing memory to disk, the following occurs:

buffer table is rebuilt

age links are initialized and rebuilt

all disk I/O data structures ??

Description

X'F511' is used by the system command :WARMSTART.

X'DEAD' is used by the system command :MDUMP.

X'DC10', is used by the system command :WARMSTOP.

If the system is restarted, all data structures are reinitialized, including those mentioned above.

WRITE.WAIT - MCAL 39

The WRITE.WAIT monitor call waits for a frame to be written to disk.

Input:

R15FID frame being written

Output:

none

Data Structure:

modifies clock request block and links

Description

This is used to synchronize writes to disk.

If the frame has been written to disk since the last FORCE.WRITE MCAL for the frame, this is a NOP. (The frame not being in memory or not being write-required satisfies this condition.) Otherwise, the program counter is backed up to the beginning of the MCAL and an RQM is performed.

XFER.CLOCK - MCAL 3E

On PC implementations, the XFER.CLOCK monitor call sets time and date from the internal clock.

Input:

none

Output:

none

Data Structure:

no data structures are modified

Description

On a PC, this is called at system initialization time on processors having a hardware time of day clock.

The XFER.CLOCK monitor call is a NOP on all systems other than a PC.

10 Instruction Set for Internal Use

This chapter is intended as a reference for Ultimate system programmers, for internal use only. It details each instruction in alphabetical order.

The internal instruction set contains descriptions of OSYM entries not documented in Chapter 4.

Summary of the Instructions and Directives

:D, :F, :Q, :T	generates literal values
:INIT	enforces standard TSYM modulo
BISYNC.IO	handles bisynchronous data communications
BNREADN	branches around READN code if necessary
CRC	computes the crc for data integrity checks
DCD	interprets BASIC object code
FRM:	determines the assembled item's frame number
HLT	DELETE?
IBM.DB.TRAP	enters the debugger and passes error info.
LOCK	performs a lock at the virtual level
MCAL	calls an MCAL in the kernel as a subroutine
MCODE	DELETE?
MODEM	defines a mode-id for BASIC opcode table
MP	moves a string which may be write-protected
MSG	references a system message in another frame
MTEXT	inserts the text of a system message
MV	moves data referenced at a memory address
MVER.OFF	turns verifying off
MVER.ON	turns verifying on
POPN	pops an integer from the BASIC stack
POPS	pops a string from the BASIC stack
PUSHD	pushes a descriptor onto the BASIC stack
PUSHN	pushes an integer onto the BASIC stack
PUSHS	pushes a string onto the BASIC stack

PUSH0	pushes a zero (0) onto the BASIC stack
PUSH1	pushes a scaled one (1) onto the BASIC stack
R1EQU	equates a symbol
REV	returns the current firmware revision level
READN/READT	reads multiple-character input from terminal
RPLDCD	same as DCD, but RPL source code used
RTNX	returns from a subroutine (in Debugger code)
SCHR	defines a character symbol
SETAR	DELETE?
SETDO	sets register to address BASIC descriptor via R6
SETDD	sets register to address BASIC descriptor via R3
SHTLY	defines a half tally symbol
SLEEPX	puts a process to sleep and unlocks a tally
SMOD	identifies system module items
TIIDC	moves a string with incrementing and count
VIO	transfers word from external device to CPU
VIO*	transfers word as VIO and sets interrupt
VIOLD	transfers data from external device to CPU
VIOLD*	transfers data as VIOLD and sets interrupt
VM	moves data to a specified memory address
XBCA	translates and tests for alpha character
XBCNA	translates and tests for non-alpha character

:D
:F
:Q
:T

The :D, :F, :Q, and :T instructions are used by the assembler for generating literals.

Syntax

:Dn :Fn :Qm :Tn

m mode-id

n constant or literal values

Description

The :D, :F, :Q, and :T instructions should not be used by programmers. Instead, use DTLY, FTLY, MTLY, and TLY if constants must be explicitly defined (such as in a table).

The assembler generates these instructions in statements at the end of Pass 1 to put literals at the end of object code. The correspondence is as follows:

- :D generates double tally, symbol type D
- :F generates triple tally, symbol type triple tally F
- :Q generates mode-id, symbol type M
- :T generates tally, symbol type T

Each reference to a new literal value (n) causes one of these statements to be generated at the end of the program.

:INIT

The :INIT instruction is used by the internal firmware assembler OSYM to enforce a standard TSYM modulo so that literals always appear in the same order at the end of the object code. :INIT applies only to implementations that generate literals, such as firmware machines.

Syntax

:INIT

Description

The :INIT instruction should not be used by programmers.

When the AS command is executed, the first thing the assembler attempts to do, prior to assembling any statements in the user's programs, is to assemble a statement of the form:

```
:INIT
```

If there is no :INIT opcode defined in the OSYM file (normally the case), this is not considered an error, and assembly proceeds with the first line of user code.

If :INIT exists, however, the :INIT instruction is assembled. The standard internal firmware OSYM contains an :INIT item which is a primitive (that is, it has a "P" on line 1). This primitive specifies (via an E line) a subroutine and a decimal parameter. When the subroutine is called, it ensures that the modulo of the TSYM file has the value of the decimal parameter; otherwise, it aborts the assembly with an error message [225], indicating that the TSYM modulo must be changed.

BISYNC.IO

The BISYNC.IO instruction is a monitor call that performs bisync-related tasks on LSI-based systems with DPV11 bisync boards or Ultimate 1400 systems with LP0 controllers.

Syntax

BISYNC.IO r

r address register; points to the transmit or receive data buffer address for codes 3 and 7, respectively. The data buffer address must be word-aligned and must point to the location of the first byte to be sent or stored. For the other codes, R0 should be specified.

Description

The BISYNC.IO monitor call is used in data communications and is applicable only to LSI-based systems with DPV11 bisync boards or Ultimate 1400 systems with LP0 controllers.

The LSI and 1400 kernels contain special monitor call codes to handle driving these boards. The call codes were initially designed to support the bisync 2780 and 3780 data communications protocols. However, since the actual protocol handling is done in virtual programs, these I/O commands have been generalized so that a number of functions are available, which can be used with various protocols (see codes below).

Bisync, which is a half-duplex protocol, requires an interrupt from the kernel indicating that the function has been completed before the virtual program can proceed. To accomplish this, when using codes 1-4, the program must execute a WAIT instruction until the interrupt occurs.

BISYNC.IO expects that the low-order byte of the accumulator, H0, has been set up to contain a command code number which indicates which function is to be performed. Depending on the function, other parameters may be passed from other parts of the accumulator.

BISYNC.IO calls the kernel to process the function identified by the command code in H0. The valid codes are as follows:

Code	Requested Action

1 SET DTR; initializes the controller and brings up DSR. The kernel interrupts when ready (WAIT instruction required).

T1 must contain the virtual device number of the bisync controller.

2 SEND CONTROL CHARACTERS; sends one or two control characters (depending on the control word) and brings up CTS. The kernel interrupts when ready (WAIT instruction required).

T1 may contain one of the following control words when using the standard 2780/3780 protocol. Note that 'FF' is used as a pad character. The two bytes are reversed because LSI machines process the low-order byte of a word before the high-order. When using other protocols, T1 may contain any valid control word for that protocol.

2780-3780 protocol control characters

```
X'FF2D'  ENQ      X'3710'  EOT-DLE
X'FF37'  EOT      X'FF3D'  NAK
X'7010'  ACK0     X'6110'  ACK1
```

3 TRANSMIT BUFFER; transmits a buffer of data. The kernel interrupts when transmission is completed (WAIT instruction required).

T1 must contain the number of bytes to transmit, and the register operand points to the first byte of data.

4 ENABLE RECEIVER; initializes the internal receive buffer and enables the DPV11 or LP0 receiver. The kernel interrupts when something is received (WAIT instruction required).

5 DISCONNECT; disconnects and resets the DPV11 or 1400 board.

6 READ STATUS WORD; returns into T0 the current status word indicated by the value in H1. H1 must contain one of the following subcodes (the applicable status bits are shown for each type):

Sub Code **Request Type**

1 requests transmit status word:

```
----------------------
| x000 0000 0000 0000 |
----------------------
```

applicable status word bit:

X'8000' illegal character in xmit buffer

2 requests receive status word:

```
----------------------
| xxxx xxxx xxxx xxxx |
----------------------
```

applicable status word bits:

X'8000' ACK0 bit set
X'4000' ACK1 bit set
X'2000' NAK (no acknowledge)
X'1000' WACK
X'0800' reverse interrupt
X'0400' end of transmission
X'0200' xmit abort condition
X'0100' ENQ
X'0080' good data
X'0040' bad data
X'0020' buffer overflow
X'0010' transmit aborted
X'0008' emporary text delay
X'0004' end of text
X'0002' bad buffer
X'0001' bad crc

3 requests DSR status word:

```
----------------------
| x000 0000 0000 0000 |
----------------------
```

applicable status word bits:

X'8000' data set change bit

4 requests bisync revision level

Code	Requested Action
7	ACK & SET UP RECEIVE BUFFER; sets up the next receive buffer and acknowledges receipt of the previously sent data block. The kernel interrupts when the operation is completed (WAIT instruction required). T1 must contain the value of ACK0 or ACK1 (see code 2, above). The register operand points to the first byte of the buffer storage area.

Sequence for Data Transmission

A complete I/O data transmission normally involves a sequence of operation similar to the following:

1. Before any bisync I/O can take place, the program must execute BISYNC.IO instructions with the following instructions:
   - code 1 (initializes controller)
   - code 4 (enables receiver).
2. To send data, the program uses instructions with:
   - code 2 (to send a protocol-dependent control character sequence)
   - code 3 (to send a block of data).
3. To receive data, the program uses instructions with:
   - code 7 (to acknowledge and set up for the next)
4. At any time, transmission problems may be detected and dealt with in the program via instructions with:
   - code 6 (and the subcode for the status request)
   - code 2 (if NACK or another character should be sent)
5. When all data communication is finished, the program uses instructions with:
   - code 2 (if the protocol requires an EOT character)
   - code 5 (to disconnect from the controller)

Processing Interrupts

When the kernel processes a BISYNC.IO instruction with a code that requires an interrupt, the kernel puts an entry in the Virtual Devices table for that process. Then the kernel performs the specified function, returning to the program.

Meanwhile, the program must execute a WAIT instruction (subsequent to the BISYNC.IO). WAIT is a monitor call (MCAL 16) that causes the kernel to scan the Virtual Devices table for that process. If there is an outstanding interrupt, the kernel deactivates the process and sets the program counter back to the WAIT instruction.

The process remains deactivated until the operation is completed. Then, the interrupt handler sets the interrupt flag in the Virtual Devices table and reactivates the process. The program re-executes the WAIT; the kernel checks for the interrupt flag and exits back to the program, which can then proceed in sequence. When the process is interrupted from a WAIT, the device (a positive number) or status of the interrupt (a negative number) is returned in T0:

- -1 indicates that no interrupts are queued for this port
- -2 indicates that the timer (ALARM.CLOCK) has expired.

BNREADN
READN
READT

The BNREADN instruction executes a branch if the READN instruction is not supported on the system for which the code is being assembled.

The READN and READT instructions read several characters of input from the terminal.

Syntax

BNREADN l READN r READT r

l label to branch to

r address register which points to the area where the input characters are to be placed; R0 may not be used for READN

Description

BNREADN is typically used to branch conditionally around code containing a READN instruction.

READN is used to read multiple-character groups of input from the asynchronous channel input buffer (terminal) into virtual memory, using a control character or count runout as the terminator of the read operation. READT is identical to READN except that READT terminates only on count runout.

The READN instruction expects that T0 has been initially set to zero and that T1 has been set to the size of the virtual memory buffer (for example, IBSIZE) where the input will be stored.

The READT instruction expects that T0 has been initially set to zero and that T1 has been set to the desired input size.

BNREADN is used to detect whether the READN instruction is supported on the current system. It assembles as a macro composed of a special FAR instruction and a B (branch) instruction.

```
FAR   R0,X'13'
B     label
```

If the special form of the FAR instruction is executed on a system that supports READN, the branch instruction is skipped and execution continues with the instruction after the branch instruction. However, if the special FAR instruction is executed on a system that does *not* support READN, that special form of FAR is essentially a NOP. Therefore, the branch is executed, allowing the program to branch around a subsequent READN (which would cause an Illegal Opcode abort).

The READN instruction inputs data from an asynchronous channel in multiple-character groups and stores the characters in a virtual memory buffer, starting at the byte address of the register.

If no characters are waiting to be input, the process is suspended until a group of characters is received from the asynchronous channel.

The READN terminates input when a control character is encountered (X'00'-X'1F', X'7F', X'80'-X'9F', and X'FF') or count runout. READT terminates input only on count runout.

The control characters typically input from a terminal are:

Hex	Character	Terminal Keys
08	BS	BACKSPACE or <CTRL-H>
09	HT	TAB or <CTRL-I>
0A	LF	LINEFEED or <CTRL-J>
0D	CR	RETURN
12	DC2	<CTRL-R> (to retype entire line)
17	ETB	<CTRL-W> (to delete a word)
18	CAN	<CTRL-X> (to cancel line on terminal)
1B	ESC	ESCAPE
7F	DEL	DEL

If a control character terminates input, the multi-character group includes the control character; if a count runout occurs, the group contains only the non-control input characters. (even on READT??)

A count runout occurs when the virtual memory buffer does not have enough storage available to handle any more input characters in the group. (even on READT??)

Before starting to read in the character group, the READN or READT instruction determines the maximum number of remaining characters to input by subtracting the value in T0 from the value in T1. The T1 field contains the maximum size of the virtual memory buffer and the T0 field contains the number of characters already read in. If the result of the subtraction is 0 (no remaining characters), the instruction terminates, and execution continues at the next instruction. If the result of the subtraction is negative, an Illegal Opcode abort occurs.

If there are characters remaining, the READN or READT instruction then begins to read in characters, and increments T0 for each non-control character (READN) or each character (READT) that it processes (inputs). READN handles all echoing of non-control characters, READT of all characters. At termination, the READN/READT instruction stops with the register pointing to the last character processed and T0 contains the number of characters read in.

The program using the READN is responsible for handling the control characters themselves. If the current control character is a valid signal of the end of input (for example, RETURN), the program would not need to loop back to the READN. But if the control character is, for example, BACKSPACE, the program could continue the READN. If the program adjusts the T0 field in its control character handling (for example, BACKSPACE would subtract 1 from T0), the T0 field can be used to determine the current position in the virtual memory buffer (for example, the point in the buffer when the BACKSPACE occurred). The program can also test whether the value in T0 is currently 0 (for example, the user has backspaced to the start-of-buffer), which would indicate that any more control characters (such as BACKSPACE) should be ignored.

If count runout has occurred (T0 = T1, and for READN, the last character is a non-control character), the program can either accept it as a completed input group or take corrective action, such as increasing the buffer size.

Note: *If the virtual memory buffer is a linked set of frames, the READN and READT instructions handles the crossing of frame boundaries without error or loss of data.*

CRC

The CRC (cyclic redundancy check) instruction is used to check the integrity of data by scanning a string and performing a checksum calculation on it. The check is terminated after a specified delimiter has been reached.

Syntax

CRC r,n

r address register that points one byte before the string to be checked

n constant or literal that specifies the mask of delimiter criteria to use for the string being scanned.

Description

CRC is not available in firmware on all machines (that is, some systems must compute a CRC via software).

The CRC instruction can be used to provide a more reliable data check than a parity bit or LRC. For example, the CRC could be used in data transmission or validation, or other operations where data integrity must be maintained.

The instruction scans the string, building a CRC value in D0, based on the ASCII value of each character encountered. The instruction stops with the register pointing to the delimiter, and the CRC in D0. The CRC instruction expects that D0, the low-order double tally of the accumulator, has been set to zero; then, on termination, D0 will contain the CRC value for the string.

The CRC calculation is a binary division operation, using the following polynomial as the divisor and the string as the dividend:

```
 x**32 + x**26 + x**23 + x**22 + x**16 + x**12 + x**11
+ x**10 + x**8 + x**7 + x**5 + x**4 + x**2 + x + 1
```

This polynomial is used to build an XOR mask (the divisor). The 1 value is assigned to bit 0 (high order), causing the x**32 bit (bit 33) to be dropped at the low order end. The resulting 32-bit XOR mask has the following hex value:

```
X'ED888320'
```

After the binary division, the remainder is the CRC value in D0. The CRC can then be stored in a checksum or history file or used in data transmission, as needed.

Multiple strings can be used to build one combined CRC until D0 is cleared. Several strings can be checked by using a coding sequence such as the following:

```
ZERO D0
SRA  R15,STRING1
CRC  R15,x'C0'
SRA  R15,STRING2
CRC  R15,x'C0'
```

In this example, R15 is initially set to point to STRING1, which is the byte before the first string. It is then set to point to STRING2, which is the byte before the second string. At the end of the routine, D0 will contain the combined CRC for both strings.

For the register operand, the incrementing process could generate an address that crosses a frame boundary. If the register is in the linked mode, it is normalized and attached to the next frame in the linked chain. If the end of the linked set is reached during the normalization process, the following action is taken:

- if the exception mode identifier XMODE is non-zero, a subroutine call is executed to that address, to allow special handling of this condition
- if XMODE is zero, the assembly debugger is entered with a trap condition indicating Forward Link Zero.

If the register is in the unlinked mode, and the frame boundary is reached, the debugger is entered with a trap condition indicating Crossing Frame Limit.

Mask Byte

The mask byte indicates the terminating condition for the string scan. Each byte is tested after it has been scanned to see if it satisfies the terminating condition. The delimiter itself is not used in the CRC calculation.

Note: *Because the delimiter test is done after the byte scan, the byte address of the register is always incremented by at least one.*

The mask byte can specify up to seven different characters to be tested; four of them are the standard system delimiters:

segment mark	SM	X'FF'
attribute mark	AM	X'FE'
value mark	VM	X'FD'
sub-value mark	SVM	X'FC'

The other three characters are taken from the scan character symbols SC0, SC1, and SC2. The contents of these symbols are specified by the programmer.

The low order seven bits in the mask byte are used to determine which of the seven characters are to be compared; if any bit is set (1), the corresponding character is tested; if zero (0), it is ignored.

If the high-order bit (bit 0) of the byte is set (1), it indicates that the string terminates on the first *occurrence* of a delimiter as specified by the setting of bits 1-7. If it is zero (0), it indicates that the string terminates on the first *non-occurrence* of a delimiter as specified by the setting of bits 1-7.

For more information on the use of the mask byte, see the description of SC0, SC1, and SC2 in Chapter 3.

DCD

The DCD (Decode) instruction is used to interpret BASIC object code, or to enter an appropriate routine based on the value of an opcode byte.

Syntax

DCD r,r

r address registers; the first is R6, BASIC instruction register; the second is R3, the BASIC stack register

Description

DCD uses a table of 256 mode-ids. This table is coded as an assembly language program which simply consists of 256 MODEM directives. Depending on the implementation, MODEM may generate something other than a standard mode-id (a 4-bit entry point and a 12-bit FID). The value generated, however, will be compatible with the DCD instruction for the particular implementation.

The mode-id table *must* be locked in main memory before the first DCD is executed. The following sequence of instructions must be used to lock in the table (R15 must be pointing to the frame with the table):

```
FRM.LOCK R15        Lock frame in memory; get
CMNT     *          encoded memory address in
CMNT     *          T2 via monitor call
MOV      T2,BOPS    Store memory address for DCD
```

This sequence is performed by the BASIC initialization software, as well as by Ultimate RECALL (the F-correlatives and A-correlatives use some BASIC instructions).

Each entry in the mode-id table corresponds to a B-primitive:

Entry	Byte Location (on firmware machines) and B-primitive
0	bytes 0 and 1; unused
1	bytes 2 and 3; mode-id for B-primitive X'01'
2	bytes 4 and 5; mode-id for B-primitive X'02'
.	
.	
255	bytes X'1FE' & X'1FF'; mode-id for B-primitive X'FF'

(The MODEM opcode may generate more than two bytes when assembled for a software machine.)

The BASIC compiler generates a pseudo-object code that contains BASIC primitive opcodes called B-primitives and references to variables in the BASIC variable storage area or the BASIC stack. The DCD instruction is used to execute the B-primitives in BASIC compiled code.

The DCD instruction performs the following functions:

- increments the instruction register (R6 for BASIC)
- if the byte addressed by the instruction register is within a configuration-dependent range, usually 0-11 (X'0'-X'B'), the B-primitive is executed directly by the CPU. If the byte is higher than 11, it is usually used as an index into the mode-id table to determine the address of the software that is to be entered (not a subroutine call).

These two steps execute one B-code primitive. If the B-primitive is executed directly by the CPU, it continues at the next primitive after completion of processing. If a software routine is entered, then after it has completed its processing, the routine must terminate with another DCD instruction to continue to the next B-primitive.

FRM:

The FRM: instruction is used in the FRAME macro to set the frame number.

Syntax

FRM: m
FRM: n

m mode-id

n literal value

Description

The FRM: instruction is used only in the FRAME macro, and should never appear in a virtual program.

The FRM: instruction generates the object code that MLOAD looks for to determine where to load the rest of the code for the assembled item. However, the object code from FRM: is never loaded.

HLT

One byte HALT on firmware systems. Should probably be deleted to encourage compatibility (Scott).

IBM.DB.TRAP

The IBM.DB.TRAP instruction is used on IBM systems to trap to the debugger and inform the kernel of the error type.

Syntax

IBM.DB.TRAP n

n literal value that specifies the error number

Description

IBM.DB.TRAP is an IBM-only command. It is currently used when entering the virtual (system) debugger to pass the error type to the kernel.

The error information passed to the kernel allows the kernel to store the information and optimally halt the system for debugging purposes.

LOCK

The LOCK instruction may be performed at the virtual code level on Ultimate 7000 systems, and possibly on other systems as well. In all cases where the LOCK instruction cannot be performed by firmware, it automatically invokes the LOCK monitor call.

Syntax

LOCK r
LOCK r,l

r address register that points to a lock tally

l local label to branch to if the tally is already locked by another process.

Description

LOCK is used at the beginning of a section of code which inspects or manipulates data global to the process (such as files, tables, etc.). Locking the data ensures that other processes do not change the data while it is being used by the original process.

The LOCK instruction checks the tally pointed to by the register. If the tally is currently unlocked, it contains a zero (0); if it is currently locked, it contains a process number (port number + 1). If the tally is unlocked, the LOCK instruction sets the lock, increments INHIBITH, and continues. If the lock has already been set *by this process*, the LOCK instruction simply increments INHIBITH and continues. If the tally has been locked by another process, the LOCK instruction waits until the tally is unlocked. If the tally is currently locked and the label form is used, the program branches to the specified label instead of waiting.

MCAL

The MCAL instruction executes a subroutine call to a specified monitor call routine residing in the kernel.

Probably should be deleted (Scott)

Syntax

MCAL r,n,11

r address register; must be specified even if it is not actually used by the MCAL

n literal value that specifies the sequential number of this MCAL within its class

11 literal value that specifies the class; this value must be 11 (X'B')

Description

MCODE

Probably should be deleted (Scott)

MODEM

The MODEM directive defines a mode-id. It is similar to the MTLY directive, but is used for the BASIC OPCODES frame for the implementation-dependent format of the object code.

Syntax

MODEM m
MODEM n,m
MODEM n,n

m mode-id

n constant or literal value; as first operand, n specifies the entry point number and must be in the range 0-15 (X'0'-'F')

Description

MODEM may be used whenever coding a table of entry points for use with the DCD instruction.

Each MODEM directive defines a mode-id value in the OPCODES frame. This frame contains only the mode-id table used by DCD to interpret BASIC instructions, not executable instructions. Unlike mode-ids generated by MTLY, these mode-ids are never referenced in an instruction in a program.

MP

The MP instruction is used on IBM systems to move a string into memory which may be write-protected.

Syntax

MP r,r

r address register; the first points to the first byte of the source string; the second points to the first byte of the destination area

Description

The MP instruction expects that T0 contains the number of bytes to move. If T0 is zero, no action takes place. If T0 is negative, an error occurs.

The MP instruction uses the SYSTEM MCAL (4E) with a system type of IBM (0) and subcommand 2.

The instruction can handle cases where the source register points to a string that crosses a frame boundary, but the destination area referenced by the second register is assumed to be in one frame.

MSG
MTEXT

MSG and MTEXT are used to retrieve system messages. The MSG directive generates a reference to a system message previously loaded in another frame. The MTEXT directive generates the object code form of a system message.

Syntax

MSG e
MTEXT e

e system message stored in PSYM; by convention, messages are named MSG.nnnn where 'nnnn' is a decimal number.

Description

Certain virtual processes such as the debugger and file-restore are not able to retrieve and display messages from the ERRMSG file. These routines must either have the text of their messages in-line in the ABS frame, or use the system message (SYSMSG) facilities, which are more convenient for foreign language translations. The actual message text is stored in special E-type items in the PSYM file, and is retrieved via MSG or MTEXT.

Operating system routines that generate messages to the user nearly always refer to the messages symbolically. By defining messages in the PSYM file, the assembly code may be changed more easily, such as when making non-English language versions of the system.

System messages in PSYM have the following format:

	MSG.nnnn	item-ID
001	E	PSYM type-code
002	msg.no	within the frame referred to by block.no
003	block.no	within the message module
004	text-1	text element 1
...		
nnn	text-n	text element n

The msg.no and block.no are hexadecimal numbers, starting from 0, that indicate where the message is located in the SMODULE; these values are used to produce object code for the MSG instruction.

The text element is in the form C'xxxxx' or X'xxxx' as used in the TEXT directive. Each element must be on a separate line; no commas or comments are allowed.

The last (text-n) entry for message items that are to referenced by MTEXT must end with one of the following terminators expected by the PRINT or CRLFPRINT subroutines:

SM (X'FF') AM (X'FE') SVM (X'FC') SB (X'FB')

However, a message referenced by MSG must end with a segment mark (X'FF'); that is, the last (text-n) element in the PSYM item must end with a SM and it must be the only SM in the entire body.

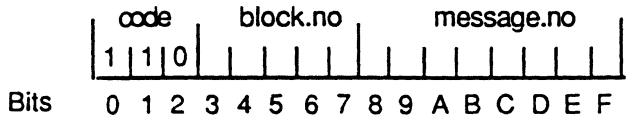
For any symbolic message, then, attribute 3 of the PSYM item should indicate the SYSMSG item that contains the message; attribute 2 of the PSYM item should give the relative position of the message within the item.

The following is sample PSYM system message:

```
     MSG.0109
001  E
002  10
003  3
004  C'Linking workspace--wait'
005  X'FF'
```

The MSG or MTEXT directive can be used immediately after a call (BSL) to PRINT, CRLFPRINT, or other routines which expect message text (or an indirect message from MSG) after the BSL. GET-MSG expects an MSG instruction to follow the BSL.

Message names can be used with both the MTEXT and the MSG directive. If a message name is used as the operand of an MTEXT directive, the assembler copies the text of the message from the PSYM item into the ABS frame as object code, just as it does for TEXT. If a message name is used as the operand of an MSG directive, the assembler generates an indirect reference (message-id) into the assembled object, using the values from lines 2 and 3 of the PSYM item defining the message. The message-id has the following format:



The message-id code causes the message text to be retrieved from the SYSMSG facility at run time.

The system message module is SMOD 125, also known as RMOD 125. This is a block of linked frames pointed to by entry 125 in frame 223 (SYS-TABLES4). The 125th word (bytes 500-503) consists of the frame count (first half tally) and the FID (second three bytes). These frames are always copied to tape as part of a SYS-GEN, and are loaded during a coldstart or a file-restore.

The SYSMSG file on the R10 account contains the source for the system message module. Each item in the file contains the MTEXT instructions for one block of messages (the block.no on line 3 of the PSYM item). The order of the messages within each item is the msg.no on line 2 of the PSYM item.

Item-IDs in the SYSMSG have the format MSGnn; nn corresponds to message blocks. For example, MSG00 corresponds to block 0 (zero).

The first line of each of these items contains an SMOD directive instead of a FRAME directive. The object code for the message is not loaded into an ABS frame, but into the module specified by the SMOD directive, using the SLOAD command).

After the SMOD directive on line 1, each SYSMSG item contains an ORG directive to the first byte of the block that the subsequent messages belong in. This ensures that the object code for that item will begin at the proper number of bytes from the beginning of the module. That is, the item for message block zero contains an ORG 0 directive, the item for message block 1 contains an ORG 500 directive, and so on.

The remainder of each item consists of MTEXT directives, one for each message in that block, in the proper order.

An ORG to the last byte of the block immediately precedes the item's END instruction.

These items are assembled just like an ordinary assembly language program. Module 125 consists simply of the object code of all the assembled items in the SYSMSG file.

If a symbolic message is always used with MTEXT, the values of attributes 2 and 3 of the PSYM item can be left as zero (0), since they are not used. But if a message is to be referenced indirectly, it must be assigned a location in the system message module and loaded (via the SLOAD command), and the location must be indicated on lines 2 and 3 of the PSYM item.

When changes are necessary, they are made to the appropriate items in the SYSMSG file, which are then assembled and loaded using the SLOAD command.

Note: Although each block can contain 500 bytes of message text, Ultimate recommends leaving space between messages to as a pad for translation into non-English language versions.

It is the system programmer's responsibility to see that this is so. It is also the system programmer's responsibility to see that the total object code for any SYSMSG item does not exceed 500 bytes.

The following is a sample listing of an item in the SYSMSG file:

```
          SMOD  125
* SYSTEM *MSGS
* SYSTEM MESSAGE FRAME
*
* SYSTEM MESSAGES IN PSYM HAVE THE FOLLOWING FORMAT:
*
*     001 E
*     002 MSG# WITHIN 500-BYTE BLOCK
*     003 BLOCK# WITHIN MODULE
*     004 TEXT ELEMENT 1
*          .
*          .
*     NNN TEXT ELEMENT N
*
  .
  .
          ORG   5000              FRAME X'A'
*
* WASTE ONE BYTE TO AVOID ATTACHING REGISTERS TO THE LAST
* BYTE OF THE PREVIOUS FRAME BEFORE MII-TYPE INSTRUCTIONS.
*
          ORG    *+1
*
X'0'      CMNT  C'
          CMNT  X'27'
          CMNT  C's   Mult-'
          CMNT  X'FD'
          CMNT  C'Attr#  Attr.name   Correlative
          CMNT  X'FD'
          CMNT  C'-----  ---------   ----------------------
          CMNT  X'FF'
*
X'1'      CMNT  C'Attr#  Index-L  Write  Read-ctr.'
          CMNT  X'FD'
          CMNT  C'-----  -------  -----  ---------'
          CMNT  X'FF'
          MTEXT MSG.0014
 *
X'2'      CMNT  C' This hold file has not been printed. OK
          CMNT  X'FCFF'
          MTEXT MSG.0526
*
* FOLLOWING IS THE LOCATION OF THE LAST BYTE IN THIS
* FRAME OF THE MESSAGE MODULE; TEXT IN THIS FRAME MUST
* NOT EXCEED THIS LOCATION.
*
          ORG   5499
*
          END
*
```

What is significance of the following??

PRINT and CRLFPRINT inspect the first byte of object code after the BSL instruction. If the two high order bits are on and the next bit is off (B'110'), it is assumed to be the first byte of an indirect reference. Otherwise, it is treated as the actual first byte of the message.

With the addition of the MTEXT and MSG instructions and the SB (X'FB') terminator, the PRINT and CRLFPRINT subroutines operate normally (as documented in the external version of the Assembly Language Manual). The SB terminator causes the subroutines to discard the top entry on the stack after using it to display the message, and then return to the next address on the stack. GET-MESSAGE returns with the address of the message text in R14 and R15.

MV

The MV instruction transfers a specified number of words (tallies) from an absolute memory address to a virtual memory location.

Syntax

MV r

r address register operand that points to the first byte of the storage area (if word-aligned), or one byte before the first byte (if an odd byte).

Description

MV expects that T2 and T0 have been set up to contain the memory address (word address) where the first word (tally) of data is to be found. T2 holds the high-order 2 bytes of the memory address, and T0 holds the low-order 2 bytes.

MV also expects that T1 has been set up to contain the number of words (tallies) to move.

The MV instruction can be used to move strings that cross frame boundaries, since it will follow the links.

Both source and destination addresses must be word-aligned since the move is one tally at a time. The kernel ensures that the byte address of the destination location is word-aligned. If the register initially points to an odd byte, the kernel automatically increment sthe register by 1 byte and zeros the low address bit (thus converting the byte address in the register to a word address).

Then the number of words indicated by T1 are moved. (The memory address is not pre-incremented for the first word move.) The instruction terminates with the register pointing to the high-order byte of the last word moved.

If no frame boundary is crossed in the move, the contents of the symbols (T1, T2, and T0) are not touched; otherwise, they are all altered.

MVER.OFF
MVER.ON

The MVER.OFF directive is used to exclude certain code from being verified; this applies to the definition of variables that are initialized by the ABS mode. The MVER.ON directive is used to turn verifying back on after it has been turned off via MVER.OFF and the variables have been declared.

Syntax

MVER.OFF
MVER.ON

Description

The MVER.OFF and MVER.ON directives are designed to ensure that variable data will not be included in the verification by either the MVERIFY or VERIFY-SYSTEM commands. MVER.OFF generates special object code that turns off verifying for the code that follows this directive. MVER.ON turns on verifying, which is the normal condition for program code. (MLOAD recognizes and ignores the special object code.)

These directives should be used in any mode that has variables (flags, tables, lock-tallies, etc.) that initialize a part of the ABS frame that will be modified at execution time. MVER.OFF should be used before declaration of the variables and MVER.ON afterward to resume verification.

The BUILD.CHECK-SUM program, which is used to create a CHECK-SUM type item from the object code of assembled modes in a file (for example, SM or SYSTEM-OBJECT), uses the special object code to create a CHECK-SUM item for the proper ranges of program lines, excluding lines with MVER.OFF. The locations skipped by ORG statements are not included in the checksum and do not require MVER.OFF.

The following example shows how MVER.OFF and MVER.ON are used in the LOCKS mode (frame 170).

```
 FRAME 170
  .
  .
 MVER.OFF                        *Don't verify NEP
0        ORG    *+ID.EP.SIZE     Leave room for one EP
 MVER.ON                         * Turn verify back on.
1        EP       !QUNLOCK
2        EP       !START
  .
  .
END.ENTRIES EQU *
 ORG    0
 MVER.OFF                        * Don't verify lock
QLOK     TLY    0
 MVER.ON                         * Turn verify back on.
 ORG    END.ENTRIES
*
 ALIGN   * THIS TABLE IS REFERENCED...
 MVER.OFF *                      * Don't verify table
PEQLOK   TLY    0
IQLOK    TLY    0
FQLOK    TLY    0
SPLRLINE DEFT   R1,*16
 ORG    *+2
 MVER.ON *                       * Turn verify back on.
  .
  .
```

The following discussion is based on an older version of the LOCK mode example. The sense of it is probably OK, but references to the NEP and to the specific locations are not accurate.

One objective of the above is to allow R1 to address a system lock (now a tally, which means that the first entry point cannot be used since it would be overwritten by the lock). So, we use MVER.OFF before the NEP and MVER.ON after it.

Second, after ORGing back to zero to name and initialize the lock tally, we MVER.OFF, declare the QLOK tally, and MVER.ON again.

Since the MVERs are directives, their order in the source is significant, while the actual location counter at the time the MVER is encountered is not. Thus, the MVER.OFF may either precede or follow the ORG to location zero (0), and the MVER.ON may precede or follow the ORG to END.ENTRIES (because there is no other object code generated between the ORGs).

The same is true later with the MVER.ON at location 0028; it may either precede or follow the ORG to *+2 (or the DEFT). Actually, if the MVER.ON at 0002 and the MVER.OFF at 0020 were simply omitted, the only consequence would be that the ALIGN byte at 001F would not be verified.

The result of the MVER directives in the source shown above yields a CHECK-SUM entry with ranges 3-1F, 28-1F5 (1F5 is the last byte of generated object), and MVERIFY will only verify the same ranges. (Also, the second HALT of the NEP--the 01 at location 0002--is not part of the lock and won't be verified either.)

POPN
POPS

The POPN and POPS instructions are BASIC instructions. The POPN instruction pops a direct integer off the BASIC stack. The POPS instruction pops a string (either direct or indirect) off the BASIC stack.

Syntax

POPN r3 POPS r3,r

r address registers

Description

The register for POPN should normally be R3 (HS) since that is the BASIC stack register, which points to the next stack entry on the BASIC stack of descriptors. The first register should *always* be R3 for POPS. The second register will be set to point to the string referenced by the descriptor popped off the stack.

The POPN and POPS instructions are used whenever a BASIC program needs to retrieve data from the stack. The stack consists of descriptors; in the case of long strings, the actual data is not on the stack. POPN is used to retrieve the numeric value of the data referenced by the top-of-stack descriptor; POPS is used to retrieve a string value.

The BASIC stack is referenced by the HS buffer beginning pointer HSBEG. HSBEG always points to the beginning of the stack. The stack itself, however, is built in frames obtained from overflow, which are released when the program exits. Since the address register R3 (HS) points within the stack to the next stack entry, the actual top-of-stack is ten bytes before the current location of R3.

Popping an entry off the stack consists of decrementing R3 by ten bytes, and retrieving the data via the descriptor then addressed by R3.

If the descriptor code after the stack is popped does not match the requirement of the instruction (that is, a direct integer for POPN or a direct or indirect string for POPS), or if the stack register crosses a frame boundary (on most implementations), an appropriate subroutine is called to perform the conversion or exception processing.

The subroutine's address is obtained from the mode-id table entry 2 for POPN, or entry 4 for POPS. If a subroutine is called, the stack register is left unchanged before the routine is entered .

POPN or POPS should never be used with operands that could produce different results when the subroutine is invoked. In particular, POPS should always be used with R3 and R10 in BASIC run-time code since that is what is used by the software routines referenced by the BASIC OPCODES (mode-id) table.

For example, the following sets R10 to point to the top-of-stack string data:

```
        POPS    R3,R10
```

The functions of each POP instruction is best described by showing their software equivalents (for illustration only; the code may not do these exact steps to get the correct value):

```
POPN    (Value will be returned in FP0)

        DEC  R3,10          Pop the stack
        BCE  R3,X'01',IN    OK if value is integer
        INC  R3,10          Reset stack pointer
    ...
 [Enter software routine]
 ...
 IN     INC  R3,2           Set stack pointer to value
        LOAD R3;F0          Load value from descriptor
        DEC  R3,2           Reset stack pointer
        RTN
```

```
POPS     (Pointer will be returned in 'r' below)

         DEC R3,10           Pop the stack
         BCE R3,X'02',SS     Test for Short String code*
         BCE R3,X'82',LS     Test for Long String code*
         INC  R3,10           Reset stack pointer
...
[Enter software routine]
...
 SS      MOV  R3,r           Pointer to short string
         RTN
 LS      INC  R3,2           Set stack pointer to
         CMNT *               storage register
         MOV R3;S0,r         Long string pointer from
         CMNT *              descriptor.
         DEC  R3,2           Reset stack pointer
         RTN
```

PUSHx

The PUSH instructions are BASIC instructions that are used to push data onto the BASIC stack.

Syntax

```
PUSH0  r3        PUSH1  r3
PUSHN  r3        PUSHS  r,r3      PUSHD  r,r3
```

r3 The PUSH0, PUSH1, and PUSHN forms have one address register (r) operand, which should normally be R3 (HS) since that is the BASIC stack register (points to the next stack entry on the BASIC stack of descriptors).

r In the PUSHS and PUSHD commands, the first register points to the location of the string or descriptor; the second register should always be the BASIC stack register (R3).

Description

The instructions push values onto the BASIC stack as follows:

PUSH0 pushes a zero

PUSH1 pushes a (scaled) one

PUSHN pushes an integer

PUSHS pushes an indirect string

PUSHD duplicates a descriptor, or enters a software routine, depending on ??

The BASIC stack is referenced by the HS buffer beginning pointer, HSBEG. HSBEG always points to the beginning of the stack. The stack itself, however, is built in frames obtained from overflow, which are released when the program exits. Since the address register R3 (HS) points within the stack to the next stack entry, the actual top-of-stack is ten bytes before the current location of R3.

Pushing an entry onto the stack consists of storing a ten-byte descriptor starting at the current location addressed by R3 (not at the location +1). R3 is then incremented ten bytes.

The only exception to the above is in the case of PUSHD when the descriptor being copied has a type code of X'00' in byte 0 of the descriptor. When this happens, the copy is not made, and the routine referenced by entry 3 (PUSH ADDRESS) in the mode-id table is entered.

(The routine is entered as if an ENT instruction had been executed, not a BSL as is the case for POPN and POPS software routine entries.)

The reason for this is that a descriptor of type X'00' (Undefined variable) does not reference data, and it is assumed that the PUSHD instruction is being used to push data (or a pointer to data) onto the stack. Entry 3 of the mode-id table is used to reference an error routine, which traditionally prints the message:

```
Variable has not been assigned a value--zero used.
```

If a software routine is called, the stack register is left unchanged before the routine is entered .

The functions of each PUSH instruction is best described by showing their software equivalents:

```
PUSH0
        MCC    X'01',R3          Direct Integer
        INC    R3,2              On byte 2
        ZERO   R3;F0             Store zero
        INC    R3,8              Complete push


PUSH1
        MCC    X'01',R3          Direct Integer
        INC    R3,2              On byte 2
        MOV    SCALE,R3;F0       Store scaled 1
        INC    R3,8              Complete push


PUSHN (value is in FP0)
        MCC    X'01',R3          Direct integer
        INC    R3,2              On byte 2
        STORE  R3;F0             Store value
        INC    R3,8              Complete push


PUSHS (Register addressing string is r below)
        MOV    X'8200',R3;T0     Indirect string
        INC    R3,2              On byte 2
        MOV    r,R3;S0           Store pointer
        INC    R3,8              Complete push


PUSHD (Descriptor to be copied is addressed by r below)
        BZ     r;H0,ENT          Branch if descriptor 0
        MOV    r;F0,R3;F0        Copy next 6 bytes
        MOV    r;T3,R3;T3        Copy next 2 bytes
        MOV    r;T4,R3;T4        Copy next 2 bytes
        INC    R3,10             Complete push
*                                Done

ENT     ENT    2,MODEID          BOPS
```

R1EQU

The R1EQU directive is used to improve efficiency on software implementations and is an alternative to the EQU directive, for internal use only.

Syntax

R1EQU symbol,n

symbol name of the symbol

n constant or literal value to assign to the symbol.

Description

The R1EQU directive may be used once in any frame in place of the following sequence, where the symbol is not used in any instruction that requires the address of the symbol:

```
          FRAME nnn
          ...
          ORG   0
          CHR   xxx
symbol    EQU   R1
```

In fact, this is exactly the macro sequence generated on firmware implementations.

Use of R1EQU improves efficiency on software implementations because immediate values within instructions execute faster than references to locations within virtual memory.

On software implementations, R1EQU equates a symbol to a value. On firmware implementations, R1EQU places a value at location X'000' in a frame and EQUATEs R1 to a symbol that refers to it.

In the following example, the equated symbol COLON can be used in any instruction that refers to the symbol's value. However, it would not be valid in an instruction that uses the address of the symbol, such as a WRITE.

```
        FRAME   511
        R1EQU   COLON,C':'
        ...
        BCE     R5,COLON,L1
```

The R1EQU shown in the example is expanded to the following code on firmware implementations:

```
        ORG     0
        CHR     C':'
COLON   EQU     R1
```

On software implementations, it will be expanded as follows:

```
COLON   EQU     C':'
```

READN

The READN instruction is used to read a group of characters of terminal input.

For details, see the BNREADN/READN/READT instruction.

READT

The READT instruction is used to read a group of characters of terminal input.

For details, see the BNREADN/READN/READT instruction.

REV

The REV instruction returns the revision number of the current firmware revision level. Its use is dependent on the particular implementation.

Syntax

REV r

r address register

Description

REV returns a 1-byte number that identifies the current firmware revision level. The number is stored at the byte address of the specified register.

RPLDCD

The RPLDCD instruction is used on some implementations instead of the DCD instruction. It uses the source code for RPL as a replacement for the DCD code.

Syntax

RPLDCD r,r

r address register; the first is the RPL instruction register (typically R5); the second is the RPL stack register (typically R2)

Description

On some implementations, the location of the opcodes table is derived from the BOPS tally in the PCB. When BASIC is running, BOPS contains the FRM.LOCK-provided memory location for the BASIC opcodes frame. When RPL is running, BOPS contains the FRM.LOCK-provided memory location for the RPL opcodes frame. For performance reasons, some implementations do not decode BOPS to get the memory address, but already know where the BASIC and RPL opcodes tables are. For these implementations, different decode instructions are needed for RPL and BASIC so that the appropriate table can be accessed.

RPLDCD functions identically to the DCD instruction except that it uses the RPL opcodes table instead of the BASIC opcodes table.

RTNX

The RTNX instruction is a form of RTN for internal use only. This form of RTN is not noticed by the assembly debugger when tracing modal entries (M command). It is used only in debugger system code.

Syntax

RTNX

Description

The RTNX instruction should be used only after the DB.ENT monitor call in System Mode DB01.

The assembly debugger is entered as if a BSL instruction to one of the entry points in frame 1 (DB01) were executed. The code in frame 1 executes a DB.ENT monitor call to begin executing debugger code using the DCB (debug control block) in place of the PCB. When the debugger is through executing, a DB.LV monitor call returns control to the instruction after the DB.ENT in DB01; this instruction is a RTNX. If an ordinary RTN were executed here, and if modal tracing were enabled, the RTN would immediately cause another entry to DB01, producing an endless loop.

SCHR

The SCHR directive is used to improve efficiency on software implementations and is an alternative to the CHR directive, for internal use only.

Syntax

{symbol} SCHR n

symbol symbol name of the character

n constant or literal value to be assigned to the character symbol.

Description

SCHR can be used when only the value of a symbol is desired. It cannot be used where the CHR values must be in virtual memory.

Use of SCHR improves efficiency on software implementations because immediate values within instructions execute faster than references to locations within virtual memory.

On software implementations, SCHR equates a symbol to a value. On firmware implementations, SCHR reserves one byte of storage and defines the symbol in the label field to be of type C (character).

In the following example, the character symbol PERIOD can be used in any instruction that refers to the symbol's value. However, it would not be valid in an instruction that expects to find the value in virtual memory, such as SRA or LOAD.

```
        FRAME   511
        ...
PERIOD  SCHR    C'.'
...
        BCE     R12,PERIOD,L1
```

SETAR

?? Fate unknown since not currently used. It changes the number of the base register of literals (which is always R1 as far as virtual programmers are concerned). (Scott)

SETDD
SETDO

The SETDD and SETDO instructions are BASIC instructions used to set up a register to address a specific descriptor in the BASIC variable space. The variable's descriptor location is encoded as an offset into the variable space.

Syntax

SETDD r,r **SETDO r,r**

r address register operands; the first register points to the register to set up to address the descriptor. For SETDD, the second register is the BASIC stack register (R3). For SETDO, the second register is the BASIC instruction register (R6).

Description

SETDD is used when the offset to the descriptor is to be obtained from the BASIC stack. SETDO is used when the offset is to be obtained from the next two bytes of the BASIC object code in the current program.

For SETDD instructions, the offset to the descriptor is obtained from the BASIC stack, via the stack register. The PUSH ADDRESS B primitive creates a stack entry with the offset stored in bytes 2 and 3. This value is picked up by SETDD and used to reference the variable.

For SETDO instructions, the offset to the descriptor is obtained from the object code, via the instruction register. The offset is the next two bytes of BASIC object code.

SETDD does not alter its second (stack) register; however, SETDO increments the second (instruction) register.

The functions of the SETDD and SETDO instructions are best described by showing their software equivalents:

```
 SETDD (Pointer returned is 'r' below)
         LOAD  R3;T1           Pick up offset*
         MOV   ISBEG,r         Set to beginning of
         CMNT  *               variable space
         BHEZ  T0,NC           Not in COMMON area if
         CMNT  *               high-order bit 0
         ADD COMDSP            Adjust for COMMON
 NC      INC r,T0
         RTN

 The accumulator is used here only for illustration;
 SETDD does not destroy it.

SETDO (Pointer returned is 'r' below)
         INC R6                To next byte in object code
         LOAD R6;T0            Pick up offset*
         MOV ISBEG,r           Set to beginning of
         CMNT *                variable space
         BBZ R6;B0,NC          Not in COMMON area if
         CMNT *                high order bit 0
         ADD COMDSP            Adjust for COMMON
 NC      INC r,T0
         INC R6                Update instruction register
         RTN

* The accumulator is used here only for illustration;
 SETDO does not destroy it.
```

SHTLY

The SHTLY directive is used to improve efficiency on software implementations and is an alternative to the HTLY directive, for internal use only.

Syntax

symbol SHTLY n

symbol specifies the symbol name of the character

n specifies the value of the half tally symbol as an alphabetic, numeric, or alphanumeric value

Description

SHTLY can be used when only the value of a symbol is desired. It cannot be used where the SHTLY values must be in virtual memory.

Use of SHTLY improves efficiency on software implementations becuase immediate values within instructions execute faster than references to locations within virtual memory.

On software implementations, SHTLY equates a symbol to a value. On firmware implementations, SHTLY reserves one byte of storage and defines the symbol in the label field to be of type H (Half tally).

In the following example, the half tally symbol FIVE can be used in any instruction that refers to the symbol's value. However, it would not be valid in an instruction that expects to find the value in virtual memory, such as SRA or LOAD.

```
        FRAME   511
        ...
FIVE    SHTLY   5
        ...
        BU      ACFH,FIVE,L1
```

SLEEPX

The SLEEPX instruction performs the same function as SLEEP, except that the lock tally addressed by the address register is zeroed. SLEEPX causes the process to be deactivated and put in a wait state until a specified time of day.

Syntax

SLEEPX r

r address register; points to a lock tally on an even-byte boundary (word-aligned).

Description

The SLEEPX instruction is used when a process is to be put to sleep and a lock which the process has is to be unlocked (zeroed).

SLEEPX is used to avoid the following problem:

Suppose that a LOCK monitor call instruction has been used to gain exclusive access to a table or other data. The code to manipulate the table has been executed. The process now wants to unlock the table and go to sleep until another process wakes it up with a PIB.ATL monitor call.

If the first process were to use two separate instructions to unlock and go to sleep, there is a chance that it would be deactivated after the first instruction, the other process would execute the PIB.ATL, the first process would re-activate and execute the SLEEP, and the signal from the second process to wake up from the SLEEP would be completely missed.

The solution is for the first process to point an address register at the lock tally and execute a SLEEPX. Then the second process will not run until the first is asleep, since unlocking and going to sleep are done by the kernel in effectively one step.

SMOD

The SMOD directive is used instead of a FRAME directive to identify system module items.

Syntax

SMOD n

n system module number

Description

The SMOD directive is used instead of a FRAME directive when assembling system module items, such as SYSMSG messages and keywords. System module items provide a convenient method of storing text data together for foreign language translation, and also allow the system to store certain data structures in binary format that do not conform to the standard Ultimate file structure.

The SMOD items are loaded via the SLOAD command. Before a new SMOD item can be loaded, it must be added as an entry to the SLOAD2 table (frame 155) so that it will be recognized and processed by SLOAD.

Frame 223 is a map called SYS-TABLES-IV that identifies the system code stored as system module items; system module items are stored in binary format rather than in standard Ultimate file structure. The kernels for firmware implementations as well as certain data structures used by virtual code are stored in this way. The modules containing no executable code can be referred to as either SMODs or RMODs.

One such module is the system message module, SMOD 125 (or RMOD 125). This is a block of linked frames pointed to by entry 125 in frame 223 (SYS-TABLES-IV). These SMOD frames are always copied to tape as part of a SYS-GEN, and are loaded during a coldstart or file-restore. They are initialized with the SLOAD command.

The source for SMOD 125 is in the SYSMSG file on the R10 account. Each item in the file corresponds to one 500-byte block of messages, and is assembled just like an ordinary assembly language program. However, the first line of each of these items consists of an SMOD directive instead of a FRAME directive. This is because the object code is not loaded into a specified ABS frame via the MLOAD command, but into a specified module via the SLOAD command. Module 125 consists simply of the object code of all the assembled items in the SYSMSG file.

TIIDC

The TIIDC (translate incrementing string to incrementing string, delimiter) instruction is the same as the MIIDC instruction except that each byte is translated before it is moved. The number of characters moved is counted.

Syntax

TIIDC r,r,n

r address register in the range of R3-R14

n constant or literal value that specifies the mask of delimiters to use as terminators for the string being moved.

Description

The first address register's byte address +1 references the starting character of the string to be moved. The second address register's byte address +1 is the location at which the starting character of the string is to be stored.

TIIDC increments the register operands. The character addressed by the first register is translated and stored at the location addressed by the second. If the specified delimiter is not encountered, the operation is repeated.

The TIIDC instruction expects that the accumulator T0 has been set up (usually to ZERO or ONE), so that on termination T0 will indicate the number of characters moved.

TIIDC also expects that R15 has been set up to point to a 256-byte translation table. Byte 0 of the table contains the value to be put in the destination string whenever a X'00' appears in the source string. Byte 1 of the table contains the translated value of X'01', etc. The translation table must be set up prior to executing the instruction.

The TIIDC instruction can be used whenever a string needs to be translated into another format (such as from European to English) before it is moved to another location and the string delimiters can be specified in a mask byte.

The third operand, called a mask byte indicates the terminating condition for the string move. Each byte of the untranslated (source) string is

tested after it has been copied, to see if it satisfies the terminating condition.

For the address register operand, the incrementing process could generate an address that crosses a frame boundary. If the register is in unlinked mode, and the frame boundary is reached, the assembly debugger is entered with a trap condition indicating Crossing Frame Limit. If the register is in linked mode, it is normalized and attaches to the next frame in the linked chain. If the end of the linked set is reached during the normalization process, the following action is taken:

- If the exception mode identifier XMODE is non-zero, a subroutine call is executed to that address, to allow special handling of this condition.
- If XMODE is zero, the assembly debugger is entered with a trap condition indicating Forward Link Zero.

Mask Bytes

The mask byte can specify up to seven different characters to be tested; four of them are the standard system delimiters:

segment mark	SM	X'FF'
attribute mark	AM	X'FE'
value mark	VM	X'FD'
sub-value mark	SVM	X'FC'

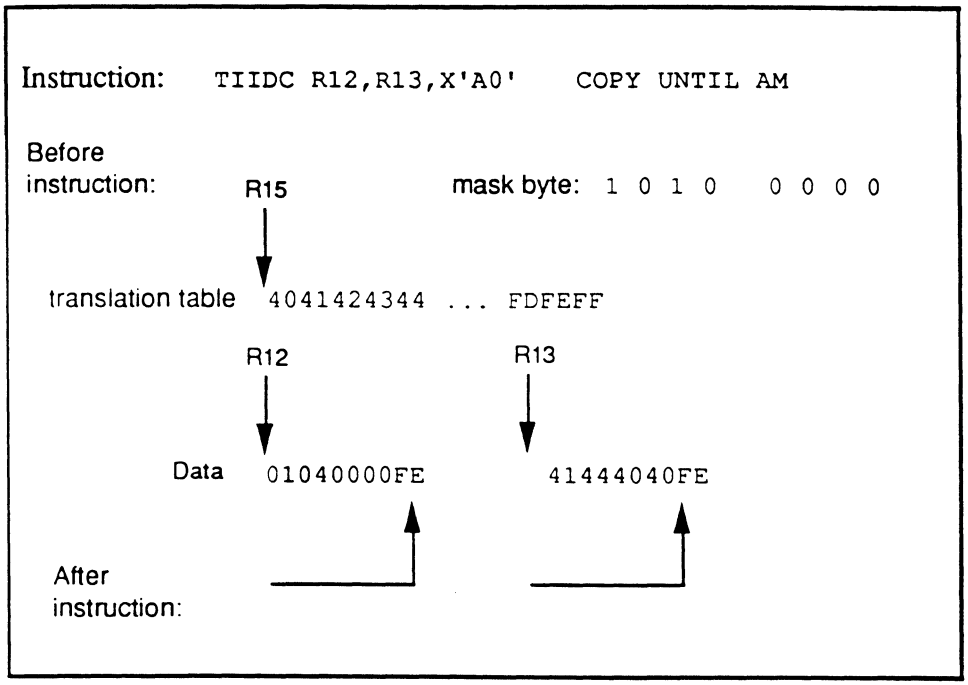
The other three characters are taken from the scan character symbols SC0, SC1, and SC2. The contents of these symbols are specified by the programmer.

The low order seven bits in the mask byte are used to determine which of the seven characters are to be compared; if any bit is set (1), the corresponding character is tested; if zero (0), it is ignored.

If the high-order bit (bit 0) of the byte is set (1), it indicates that the string terminates on the first *occurrence* of a delimiter as specified by the setting of bits 1-7. If it is zero (0), it indicates that the string terminates on the first *non-occurrence* of a delimiter as specified by the setting of bits 1-7.

For more information on the use of the mask byte, see the description of SC0, SC1, and SC2 in Chapter 3.

In the following example, note that X'FE' was simply translated to itself because that is how the translation table is built.



VIO
VIOLD

The VIO and VIOLD (virtual input/output) instructions are used internally to transfer data between external devices and the CPU. The VIO instruction transfers one word or tally (16 bits) to or from the specified device. The VIOLD instruction may transfer one word or a block of words, and an address.

Syntax

VIO t,t,t **VIO* t,t,t**
VIOLD r,t,t,t **VIOLD* r,t,t,t**

t tally symbol or a literal (symbol type n)

r address register

* indicates that an interrupt request is to be set in the kernel's interrupt table

Description

For VIO{*}, the first operand is the transfer.loc; the second is the device.addr ; the third is the funct.code.

For VIOLD{*}, the first operand is a register that references the first byte of the transfer.loc; the second is the dev.addr; the third is the funct.code. The fourth operand specifies a range .

transfer.loc transfer location; for VIO, this is where the word transferred is stored (if input) or obtained (if output); for VIOLD, this is the starting location of the transfer.

device.addr virtual device address; must be addressed by R0 or R1. This identifies a peripheral device on the system that can be accessed by virtual code. Virtual device numbers begin at zero, and come in pairs. Even device numbers refer to the input channel of a device, and odd numbers refer to the output channel. Two channels are always reserved, even for devices normally associated with output only (printers, for example), or input only. Which channel to use in an instruction depends on the function being performed. Details must be obtained from the appropriate hardware manual.

funct.code — function code; contains the function to be performed, and must be addressed by address registers R0 or R1. The actual function code format is dependent on the hardware.

range — range; used only in VIOLD instructions to specify the number of bytes to be transferred, and must be addressed by address registers R0 or R1.

Note that, unlike all other instructions, VIO and VIOLD *require* operands to be either in the PCB (addressed via R0) or in the current program frame (addressed via R1).

The usage of these instructions is completely dependent on the particular hardware implementation of Ultimate. For example, on LSI-based systems, all VIO/VIOLD instructions automatically set interrupts for the appropriate function codes (and ignore an asterisk, if one is present). The details about virtual device address and function code formats must be obtained from the appropriate hardware manual or the source code.

Also associated with each virtual device is a device-id. This is obtained by executing the GET.ID or N.GET.ID monitor call instruction with a specified virtual device number as a parameter. If GET.ID or N.GET.ID returns zero, there is no device with that virtual device number. Otherwise, the device type can be determined from the device-id.

Device-ids are two-byte numbers. The following are some examples of device-IDs.

Device-id	Description
X'2003'	line printer with 96-character set and VFU
X'2017'	5 1/4" diskette
X'204E'	9-track 75 IPS 800/1600 BPI tape
X'207F'	1/4" QIC-02 Streamer tape drive
X'2118'	50baud-19.2kb async comm port
X'2158'	50baud-19.2kb sync comm port
X'2305'	413mb FSD disk
X'2363'	256mb mass storage unit
X'2949'	HDLC broadband comm port
X'3311'	1/4" streamer tape drive on FSD controller

When a system has several virtual devices of the same type (such as tape drives), the devices of that type are always numbered in the order of their virtual device numbers, usually corresponding to hardware channel numbers or something similar.

Execution continues as usual after the VIO or VIOLD has initiated the I/O request. VIO* and VIOLD* initiate data transfer but allow the program to continue.

If an interrupt has been requested (VIO* or VIOLD* forms), a WAIT monitor call instruction should be used after the VIO* or VIOLD* instruction. WAIT will suspend (deactivate) the process until the completion interrupt occurs. If more than one interrupt can be outstanding at a time (such as by issuing VIO* instructions to two different devices), the accumulator should be inspected after execution of the WAIT instruction to see if it matches the device number of the desired device. The VIOLD* form also keeps the data buffer in main memory until the interrupt is received.

VM

The VM instruction transfers a specified number of words (tallies) from a relatively addressed location to an absolute memory address.

Syntax

VM r

r address register; points to the first byte to be moved (if word-aligned), or one byte before the first byte to be moved (if an odd byte).

Description

The VM instruction can be used to move strings that cross frame boundaries, since it follows the links.

VM expects that T2 and T0 have been set up to contain the memory address (word address) where the first word (tally) of data is to be stored. T2 contains the high-order 2 bytes of the memory address, and T0 contains the low-order 2 bytes.

VM also expects that T1 has been set up to contain the number of words (tallies) to move.

Both source and destination addresses must be word-aligned since the move is one tally at a time. The kernel ensures that the byte address of the source data is word-aligned. If the register initially points to an odd byte, the kernel automatically increments the register by 1 byte and zeros the low address bit (thus converting the byte address in the register to a word address).

Then the number of words indicated by T1 are moved. (The memory address is *not* pre-incremented for the first word move.) The instruction terminates with the register pointing to the high-order byte of the last word moved.

If no frame boundary is crossed in the move, the contents of the symbols (T1, T2, and T0) are not touched; otherwise, they are all altered.

XBCA
XBCNA

The XBCA and XBCNA instructions are extended versions of BCA and BCNA, and are available for internal use only.

Syntax

XBCA r,l **XBCA r,l,l**
XBCNA r,l

l local label

r address register

Description

These instructions are used instead of the BCA and BCNA instructions when the French version of the operating system must test an extended alpha character set. The extended set contains characters such as the acute (/) and grave (\) accent marks used with vowels, which are not in the normal U.S. English range.

The U.S. OSYM entries for XBCA and XBCNA are the same as the BCA and BCNA versions. The French versions in OSYM are documented (??) in the OSYM-FRENCH file on the R10 account.

Both of these instructions expect that a special subroutine has been defined in the system mode ALPHA-ENTRY-DEFS (??); this item should be INCLUDEd in any mode where the XBCA or XBCNA instruction is used. The subroutine name must contain the address register (R0-R15) used in the instruction; the valid names are ALPHA.Rx (for XBCA:RL and XBCNA:RL) and XALPHA.Rx (for XBCA:RLL), where Rx is the name of an address register.

The XBCA and XBCNA instructions are macros that generate calls to character testing subroutines, followed by one or more branch instructions. The subroutines test the indicated character, and then adjust the return stack as necessary to cause the correct instruction to be executed on return, based on the value of the character.

XBCARL Usage

The macro expansion format of the XBCA:RL form is:

```
BSL ALPHA.(2)
B (3)
```

This causes the program to do the following:

- branch if the character is alphabetic (C'A'-C'Z' or C'a'-C'z'), or if the character is in the table CHAR.TABLE below
- fall through the code in any other case

The following shows the results of compiling an XBCA instruction:

```
XBCA  R13,MYLABEL
+BSL  ALPHA.R13
+B    MYLABEL
```

The subroutine ALPHA.R13 inspects the character pointed to by address register R13. If the character is in the extended alphabetic range (see Table 10-1), execution continues with the Branch instruction after the return from ALPHA.R13. But if the character referenced by R13 is not in the extended alpha range, the return stack is adjusted so that the Branch instruction is skipped, causing the program to fall through to the next instruction after the XBCA instruction.

XBCARLL Usage

The macro expansion format of the XBCA:RLL form is:

```
BSL XALPHA.(2)
B (3)
B (4)
```

This causes the program to do the following:

- branch to the first label if the character is alphabetic C'A'-C'Z' or C'a'-C'z'
- branch to the second label if the character is in the table CHAR.TABLE below
- fall through the code in any other case

The following shows the results of compiling an XBCA instruction:

```
XBCA R13,LABEL1,LABEL2
+BSL XALPHA.R13
+B LABEL1
+B LABEL2
```

The subroutine XALPHA.R13 inspects the character referenced by R13, and returns to the first Branch instruction if the character is in the normal alpha range, or to the second Branch instruction if the character is in the special range, or to the instruction after the XBCA if the character is not in either part of the extended alphabetic range.

XBCNARL Usage

The macro expansion format of the XBCNA:RL form is:

```
        XBCA (2),(L+1)
        B (3)
(L)     EQU *
```

The meaning of this form is simply the opposite of XBCA:RL. There is no XBCNA:RLL.

Table 10-1. CHAR.TABLE

X'AC'	à	accent grave
X'AD'	â	circonflexe
X'AF'	ä	trema
X'B1'	ç	with cedilla
X'B2'	é	accent aigue
X'B3'	è	accent grave
X'B4'	ê	circonflexe
X'B5'	ë	trema
X'B8'	î	circonflexe
X'B9'	ï	trema
X'BE'	ô	circonflexe
X'BF'	ö	trema
X'C3'	ù	accent grave
X'C4'	û	circonflexe
X'C5'	ü	trema

French terminals do not actually generate the values shown for these letters. Instead, a multi-byte sequence is generated for each special character, which Ultimate system software translates into the corresponding one-byte values in CHAR.TABLE. These standard logical values can then be operated on more easily by software such as that found in word processing routines. In particular, these one-byte values are easily tested by using the XBCA and XBCNA instructions.

Notes