ULT

ASSEMBLER

THE ULTIMATE CORP.

TABLE OF CONTENTS

THE ULTIMATE CORP.

Documentation

PROPRIETARY INFORMATION

CHAPTER 1

ASSEMBLY LANGUAGE

1.1 Introduction

The ULTIMATE operating system is written mainly in a high-level assembly language which deals with data in virtual space. Users may also write their own programs in this language. This manual describes the ULTIMATE assembly language, the procedures for creating, assembling, and debugging assembly programs, and guidelines for interfacing with the operating system.

This manual is intended for persons having some familiarity with the ULTIMATE computer system and with programming concepts in general. An introductory manual is available from ULTIMATE which provides an overview of the system, and separate manuals describe the various programming languages.

1.2 Characteristics of Assembly Language

Assembly language programming on any computer requires greater attention to detail, but also provides more control over the machine. 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 code is often shorter and more efficient.

Traditionally, assembly languages deal with data in terms of main memory locations, whereas high-level languages are more abstract. A variable in a BASIC program, for example, may be assigned a value without regard to its memory location. The ULTIMATE assembly language differs from traditional assembly languages in that references are not made to main memory locations, but to virtual memory locations.

1.3 Virtual Memory

"Virtual memory" in the ULTIMATE system refers to a set of locations consecutively numbered from zero to over one billion. With few exceptions, every program and data area in the system has a virtual memory address. This has an important implication in assembly language programming: since virtual memory addresses are used, any assembly program can reference any data in virtual memory. This makes assembly instructions powerful, but also potentially dangerous.

In contrast to programming in BASIC, for example, programming in assembly language must be done with much more care. If a BASIC program works incorrectly, it tends to affect only the terminal on which it is run or the account on which it was compiled. An assembly program, however, could affect several terminals, or destroy data throughout the system. It could even destroy most of the operating system software, which is itself in virtual memory.

Physically, virtual memory is stored on magnetic disc and brought into main memory a section at a time on an as-needed basis. This is discussed in more detail in the next chapter.

1.4 Process Elements

An ULTIMATE computer system is normally configured as a multi-processing system with one process assigned to each terminal port, plus at least one "phantom" process for tasks such as print spooling.

Each process is assigned an area of virtual memory for assembler-related elements such as registers, stacks, and accumulators. When a process executes an assembly instruction which references one of these elements, the reference is always relative to the beginning of the virtual space assigned to that process. This allows several processes to execute the same program simultaneously. The assembly language programmer typically does not need to know the exact virtual memory address of a process element, since it is defined at the same relative offset for whatever process is executing.

Processes are discussed in greater detail in the next chapter.

CHAPTER 2

SYSTEM ARCHITECTURE

2.1 Introduction

The ULTIMATE operating system runs on several different types of central processing units (CPU's). This chapter describes the underlying system architecture, which is identical for all the CPU's. An ULTIMATE computer system consists of a CPU, main memory, secondary memory or disc, asynchronous communication channels to serial devices, and other peripheral devices.

The ULTIMATE operating system software is written in a high-level assembly language that deals with data in the system's virtual memory space. The assembly instructions are typically decoded by high-speed control memory, or firmware. In addition to instruction decoding, the firmware also aids in virtual memory management, resulting in speed and efficiency. The virtual memory scheme is geared heavily towards the data and string handling functions in which the system excels.

2.2 Virtual and Monitor Processes

DEFINITION: A VIRTUAL PROCESS (commonly "process") on the system is an ongoing task that executes a sequence of assembly level instructions. It is identifiable by a PROCESS IDENTIFICATION BLOCK (PIB), which is main memory resident and is uniquely assigned to each process.

DEFINITION: There is one MONITOR PROCESS (commonly "Monitor"). The Monitor executes memory-resident programs called the KERNEL, and is responsible for the following tasks:

a. All I/O scheduling and management.
b. Virtual process scheduling and initiation.
c. Special functions when called via a Monitor Call instruction.

A virtual process is typically attached to one of the asynchronous communication channels available on the system, and is therefore also commonly called a "channel" or "port." This provides the user with the standard interactive interface with the system.

However, a process does not necessarily have to be attached to such a channel. In this case, the process is referred to as a "background" or "phantom" process. The print spooler is an example of such a phantom process.

2.2.1 Activation and Deactivation of Virtual Processes

A process may be ACTIVE or INACTIVE.

The Monitor maintains a schedule of available processes and their relative priority to be activated. When the Monitor turns over

control by selecting a virtual process which is next in line and has no roadblocks to prevent activation, that process is said to be active.

A process is inactive if it has returned control to the Monitor due to one of the following events, which cause a roadblock in its execution:

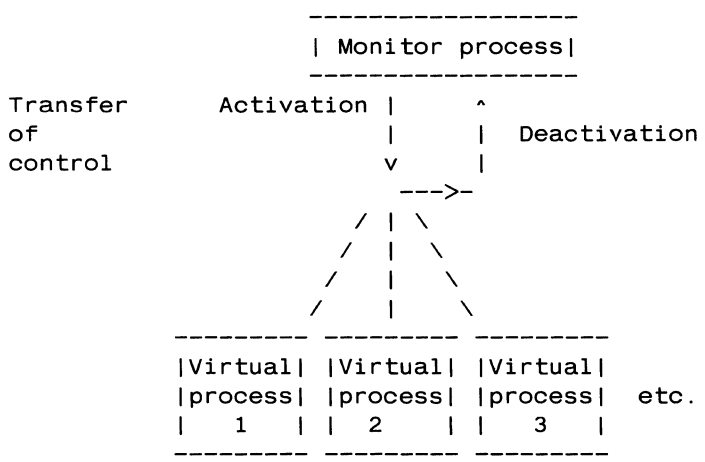
a. When a virtual process makes a reference to data that are not in main memory - a "frame fault" trap to the Monitor.

b. Execution of any Monitor Call instruction. In the case of many such calls, when the Monitor has completed the function that it was called upon to perform, it will reactivate the virtual process immediately.

c. Execution of a READ (asynchronous channel byte) instruction when the terminal input buffer is empty.

d. Execution of a WRITE (asynchronous channel byte) instruction when the terminal output buffer is full.

e. Involuntary termination of execution due to an external interrupt such as a power-failure or time quantum runout.



2.2.2 The Process Identification Block

DEFINITION: A PROCESS IDENTIFICATION BLOCK is a fixed block of main memory that serves to define the status of a process. It is used by the Monitor 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 which are always in main memory. All other information associated with a process is in virtual space, and can remain on disc if the process is not active.

Almost all operations involving PIB's are related either to I/O or to process scheduling. I/O-related PIB fields contain such information as asynchronous channel status flags and buffers. Examples of process scheduling-related fields are the roadblock bits and the PIB links. The Monitor maintains its process activation schedule by linking available PIB's together in order of activation. It attempts to give higher priority to "interactive" processes (those performing terminal I/O) than to non-interactive or batch-type processes, thus ensuring acceptable terminal response time.

Word 0 is the primary PIB status byte; bits in this byte are defined as follows:

Bit Number	Description
0	Reserved and zero
1	Set if process is "sleeping"
2	Set if process is waiting for disc (due to frame fault)
3	Reserved for multiple byte input
4	Reserved and zero
5	Set if process is outputting over asynchronous channel
6	Set if process is inputting over asynchronous channel
7	Set to deactivate process via software ("trap" flag)
8	Set to indicate process is in assembly Debug mode
9	Reserved and zero
10	Set to indicate process is not attached to an asynchronous channel (phantom process)
11	Set to indicate input pending (multiple byte input)
12-15	Used to communicate trap error number to Debugger

If any of the first eight bits are set, the process is said to be "roadblocked" and will not be activated by the Monitor.

Other PIB fields are subject to change from configuration to configuration and from one operating system release to another, and so are not documented here.

2.3 Virtual Memory

The system is accurately defined as a "virtual machine" because all data references are directly to the secondary memory. The secondary memory resides on the DISC SET, which is the set of labeled discs initialized when the system is bootstrapped. There may be other discs attached to the system which are not part of the DISC SET, and therefore not a part of the virtual machine.

Data are read into main memory to perform the actual operations, but the addressing mechanism at the assembly programming level is directly to disc.

The "Address Space" of the machine is the entire available disc space. Every process on the system can address this entire space 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 execption" type of error.

This scheme differs considerably from most other virtual memory mechanisms, in that the assembly programmer does not have a "virtual main memory" to deal with. The addressing mechanism is dealt with in later sections.

Another point to note in this regard is that the disc I/O to the Disc Set is completely under control of the Monitor. A virtual process cannot explicitly perform any I/O to these discs. When a virtual process "writes" data, these are changed in main memory and a flag is set to indicate that a disc write is required. The actual writing of the disc happens at some time later as determined by the state of the memory buffer and the Monitor, and is not easily determinable.

2.4 Memory Map

DEFINITION: A FRAME is a fixed block of data resident on the disc, which can be transferred between disc and main memory. The size of a frame is 512 bytes.

All frames are uniquely identified by a FRAME NUMBER or FRAME IDENTIFIER, also called FID. Frame numbers start at one and continue as high as the available disc space in the Disc Set permits. The physical limit on the frame number is 2**24, or 16,777,215. This gives a total address space of 2**33, or 8 gigabytes. The frame numbers map directly into disc addresses.

Transfer of data between the main memory and disc is one frame at a time. A frame in memory resides in a 512-byte block called a BUFFER.

In main memory, the first few kilobytes are reserved for use by the Monitor 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 paging disc frames.

In order to manage the main memory, there are several tables that contain information regarding the buffers. These tables are accessed by the memory management firmware of the system as well as by the Monitor 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 disc address and a main memory location, the protected areas of memory do not have corresponding disc addresses, and therefore cannot be addressed by a virtual process.

2.5 Monitor Software

The Monitor software, also called the Kernel, is memory-resident and is different from virtual software in the following respects:

1. Virtual software is written in assembly code that is usually decoded by the control memory of the machine, and is tied into the virtual machine's architecture. Monitor software is usually written in the "native" language of the machine (Honeywell Level 6, DEC LSI-11, etc.), and must explicitly follow the conventions of the virtual machine.

2. Monitor software can address any locations in memory directly, and is responsible for all I/O, manangement of memory tables, and virtual process scheduling.

2.5.1 Process Scheduling

The Monitor maintains and uses the PIB links to determine which process can be activated next. The PIB's are searched starting from the highest priority downwards, until a process with no roadblocks is found. The Monitor can then transfer control to this virtual process.

2.5.2 Disc Scheduling

The Monitor keeps a queue of disc addresses, sorted by cylinder number. This table is affected as follows:

1. When a virtual process generates a "frame fault" request to the Monitor, the entry is added to the disc queue.

2. When the Monitor needs to find a buffer in memory to read a frame from disc, and the selected buffer has a "write-required" flag on it, the buffer is added to the disc queue.

3. When a disc I/O completes, the entry is deleted from the disc queue.

Since the disc queue is sorted by cylinder number, the next disc request to be selected by the Monitor is always in ascending cylinder number sequence. When the highest cylinder number in the queue is reached, the table is searched from the lowest entry upwards again.

When an entry is added to the queue that has the same cylinder number as that of the current disc location, the new entry is placed before the current one. This prevents too many requests for the same cylinder to be processed on one pass thorugh the disc, which may result in one virtual process being satisfied at the expense of many others.

2.5.3 Automatic Disc Writes

Whenever the system is idle, the Monitor attempts to "flush" memory by writing buffers to disc which have their write-required flags set. This ensures that updated data will be safely on disc in case of a power failure, which could destroy the contents of main memory.

If uninterrupted, the Monitor will write one write-required buffer at a time to disc and reset its write-required flag, 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 disc will be 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 will be restarted.

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

2.5.4 Monitor I/O

All I/O operations initiated at the virtual level, except those to or from the asynchronous communication channel, are accomplished through special Monitor calls. Since the format and meaning of these Monitor calls depends heavily on the particular CPU and peripheral device, no details are given here. Standard system subroutines are provided, however, for use with common devices such as tape drives and line printers.

CHAPTER 3

DATA ADDRESSING

3.1 Information Formats

The system can address information in the following formats:

1. A bit
2. A byte
3. A byte string of indefinite length
4. A word or 16 bits
5. A double-word or 32 bits
6. A triple-word or 48 bits

At the assembly level, such information fields are called "elements" or "fields," and are given symbolic names just as variables are named in higher level languages.

For the purposes of this documentation, the following conventions apply:

1. All numbering starts at zero, and is incremented left to right. Thus bit 0 in a byte is the high-order bit, and bit 7 the low-order bit.

2. Decimal notation is normally used. When hexadecimal (base sixteen) notation is used, the hexadecimal number is enclosed in single quotes and preceded by an X, e.g. X'1F' = 31. In hexadecimal notation, the letters A through F represent the values of 10 through 15.

3.2 Frame Formats

There are two different formats for a frame: LINKED and UNLINKED. Note that this distinction is purely logical; there is no objective way of determining whether a frame on disc is linked or unlinked. Software conventions determine the usage of any particular frame.

Multiple frames may be physically linked together so as to form a doubly-linked chain of indefinite length. Once the links have been established, traversing the data in such a chain is automatic and transparent since the firmware handles the address resolution as physical frame boundaries are crossed.

A linked frame has 12 bytes of link information, and 500 bytes of data. File data frames are linked, as are the larger buffers or workspaces.

A frame may be used in an unlinked mode, when all 512 bytes are accessible as data. This is the case when the frame stands by itself, and does not logically link to other frames. For example, short buffers and control blocks are unlinked.

3.2.1 Link Field Format

The following describes the format of a linked frame:

```
          ------------------------------------------------------------
Byte no. |-0-|-1-|-2-|-3-|-4-|-5-|-6-|-7-|-8-|-9-|-A-|-B-|-C-|-D...
(Hex)    | x | NN| Forward Link  | Backward Link | NP| x | data....
         |   | CF| Frame number  | Frame number  | CF|   | bytes
          ------------------------------------------------------------
```

Where:

Byte 0 is RESERVED.

Byte 1, "NNCF", is a count that represents the number of sequential frames linked ahead of this one (Number of Next Contiguous Frames).

Bytes 2-5, "FRMN", contain the frame number of the next frame in this logical set; (These are zero if first frame in set).

Bytes 6-9, "FRMP", contain the frame number of the previous frame in this logical set. (These are zero if last frame in set).

Byte X'A' (10), "NPCF", is a count that represents the number of sequential frames linked previous to this one (Number of Previous Contiguous Frames).

Byte X'B' (11) is unused and is referred to as a "dummy data byte."

3.2.2 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, the "forward link" or the

"backward link" is used to continue in the chain.

If the required address is more than 500 bytes ahead or behind the boundary of the current frame, the "contiguous" counts play 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.

3.2.3 Examples of Linked Frames

```
DUMP L,3000

  FID:    3000 :   7   3001   2999 120  (   BB8 :  7    BB9    BB7 78 )
+ FID:    3001 :   6   3002   3000 121  (   BB9 :  6    BBA    BB8 79 )
+ FID:    3002 :   5   3003   3001 122  (   BBA :  5    BBB    BB9 7A )
+ FID:    3003 :   4   3004   3002 123  (   BBB :  4    BBC    BBA 7B )
+ FID:    3004 :   3   3005   3003 124  (   BBC :  3    BBD    BBB 7C )
+ FID:    3005 :   2   3006   3004 125  (   BBD :  2    BBE    BBC 7D )
+ FID:    3006 :   1   3007   3005 126  (   BBE :  1    BBF    BBD 7E )
+ FID:    3007 :   0      0   3006 127  (   BBF :  0      0    BBE 7F )
```

Above is an example of the tail end of a set of 128 contiguously linked frames. The first figure in each line is the FID; the second the NNCF, the third the Forward Link FID and the fourth the NPCF. Figures in parentheses are the same in hexadecimal.

```
DUMP L,12568

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

This frame has no Forward or Backward link fields.

3.3 The Byte Address

DEFINITION: All data are referenced via a BYTE ADDRESS. This byte address consists of a FRAME NUMBER and a DISPLACEMENT within the frame.

The displacement within a frame is relative to the zero'th logical byte of the frame. There are two methods of addressing data in a frame, depending on whether the link fields are to be considered or not.

In UNLINKED mode, physical byte 0 of the frame is addressed by a displacement of 0, and physical byte 511 by a displacement of 511. Therefore, in unlinked addressing mode, the boundaries of the frame cannot be crossed, and all 512 bytes of the frame are addressable.

In LINKED mode, physical byte X'C' (12) of the frame is addressed as byte 1, and physical byte 511 is addressed by a displacement of 500. Addresses with displacements in the range 1-500 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 before use by the firmware. The normalization consists of "chasing the links" in the appropriate direction until the displacement is reduced to the range 1-500.

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 futher details.

3.3.1 Table of Displacements and Addresses

Displacement	Linked mode Address	Unlinked mode Address
Less than 0	Refers to previous frames in logical chain	INVALID
0	Temporary if it refers to the "dummy data byte" in frame at location X'B'. If normalized, reverts to last byte of previous frame in chain.	Physical byte 0 of frame
1-500 Or 1-511	Physical bytes 12-511 -	- Physical bytes 1-511
Greater than above	Refers to forward frames in logical chain	INVALID

3.4 Registers

DEFINITION: There are two types of registers - ADDRESS REGISTERS (AR's) and STORAGE REGISTERS (SR's). These registers are used to store byte addresses ONLY. They cannot be used to store numbers or other data.

An AR or SR contains a frame number (FID) and a displacement, which results in a pointer or reference to a specific byte in a specific frame. At the assembly instruction level, data can be addressed only via an address register. A storage register, as its name implies, is used to save and restore the byte address of an AR. "Address register" is often abbreviated simply "register."

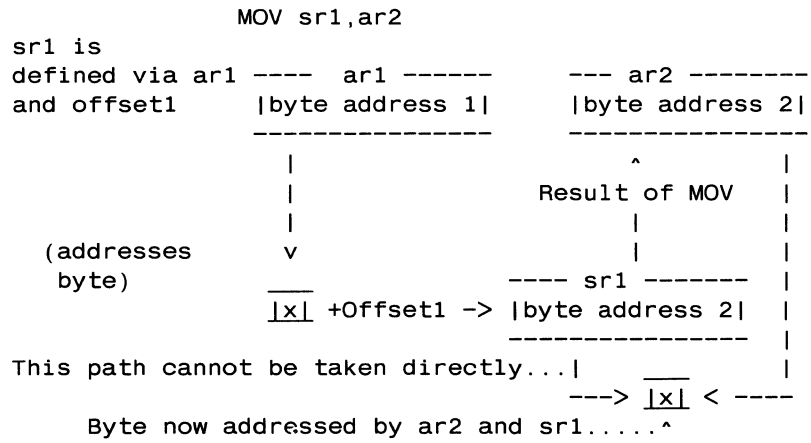
The terminology "(byte) address of a register" will be used to mean "the address of the byte that the register is referencing." This should not be confused with the actual physical location of the register itself, which is largely a matter of indifference, since at the assembly level data references have at least one level of indirection (i.e., data references are made through address registers).

3.4.1 Referencing AR's and SR's

An address register may be referenced directly, that is, by its register number (0-X'F'). In this case it is the contents, or byte address, of the register that is being referenced. When an indirect reference is made to the data pointed at by the register, several methods apply and these are described later under the section "Addressing Modes."

The contents of a storage register are referenced via an AR as a base address, plus an offset to its location. The data that an SR are pointing to cannot be directly referenced; the SR must first be moved to an AR for this to be possible.

Following is an example of an instruction that moves the byte address from an SR into an AR. Both the SR and the AR now point to the same byte.



3.4.2 Attached and Detached AR's

DEFINITIONS: An ATTACHED AR is an address register whose referenced byte (therefore frame) is both resident in main storage, and whose physical address has been resolved by the firmware and saved in an internal hardware register or memory location for use in processing instructions.

A DETACHED AR is one whose referenced byte (therefore frame) may or may not be in main storage, and whose associated hardware register or memory location is in an unresolved state.

Only an attached AR can use its referenced data.

When a process begins execution, all its AR's are in a detached state. As a first step, the firmware attaches Registers 0 and 1 by searching the memory map to determine where in main memory the addressed frames are stored. If the referenced frames are not in main memory, a Frame Fault trap is generated to the Monitor.

If the referenced frames are in main memory, execution begins and the other AR's are attached on a demand basis as needed by the instructions. That is, the memory management routine in the firmware is entered as above whenever a detached AR is used to reference data.

A specific AR is detached when a storage register is loaded into it. It will not be re-attached until a data reference is made by using it in an instruction.

A specific AR also detaches during the execution of an incrementing or decrementing instruction that causes its byte address to cross the logical boundary of the current frame. The memory management routine is entered to re-attach it to the appropriate frame, and if successful, execution of the interrupted instruction continues. Byte string instructions may go through this process repeatedly since strings may be hundreds or thousands of bytes in length.

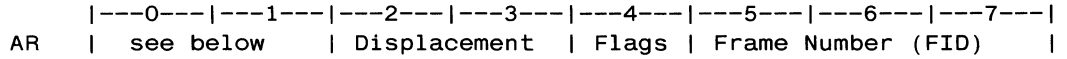
All AR's are detached when the process deactivates and returns control to the Monitor, either voluntarily or involuntarily.

Attachment and detachment of registers is automatic and transparent to the programmer.

An AR should be loaded in only one way: by moving an SR into it, which automatically detaches it. It is possible, though not recommended, to set up an AR by physically affecting its fields. If this is done, it is vital to force detachment of the AR before affecting its FID or displacement. The "DETZ r" or "DETO r" instruction may be used to detach the AR explicitly.

3.4.3 Format of an Address Register

There are sixteen AR's, Registers R0 through R15; each AR is eight bytes in length. An AR contains a byte address as described previously.



Bytes 0 and 1 are used by the firmware in some implementations, such as that for the Honeywell Level 6 WCS. Along with byte 2, these bytes are used to store the main memory address when the register is attached. In other implementations, the main memory address is stored in a hardware register which is inaccessible to the programmer.

Bytes 2 and 3 are the displacement field. Warning - the displacement field contains meaningful data only when the AR is in the detached state, and therefore should not normally be changed directly in any way.

Byte 4 is a flag field that contains specific bits as follows:

Bit 0 is the Unlinked mode flag; if set, the register's address is in unlinked mode; if zero, it is in linked mode.

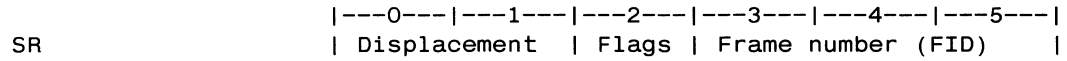
Bit 1 is the Special Attachment flag. If set, and the displacement is zero, the register will be attached to reference the dummy data byte (byte at physical displacement X'B' or 11) in the current frame. If zero, a displacement of 0 causes normalization and attachment to physical byte 511 of the previously linked frame.

The purpose of this bit is to cause the AR to temporarily address physical byte X'B' of the frame when one of the pre-incrementing data movement instructions reaches a frame boundary. It is then pre-incremented to the first data byte in the frame as instruction execution continues.

Bits 2-7 are reserved.

Bytes 5-7 contain the frame number of the register's address.

3.4.4 Format of a Storage Register



As can be seen, the format of an SR is identical to the low six bytes of the AR. All fields have the same meaning as for an AR, except that the special attachement flag is not used. When a SR is moved into an AR, the latter is flagged as detached.

3.5 Registers Zero and One

Address Registers Zero and One have hardware defined meanings.

Register Zero addresses a special frame called the PRIMARY CONTROL BLOCK or PCB (defined next), which is the basis of all data that a particular process can access.

Register One is the process' PROGRAM COUNTER.

3.5.1 The Primary Control Block

DEFINITION: The PRIMARY CONTROL BLOCK, or PCB, is a single frame unique to a particular process, and is the basis for every data reference that the process can make. The PCB contains the AR's themselves, the Subroutine Return Stack, the Accumulator, and various other data variables.

The FID of the PCB is determined when the system is initialized. When the Monitor decides to turn control over to a particular process, its PCB frame number is obtained from the PIB, and the virtual memory table is searched for that FID. If that frame is not in main memory, the process cannot be activated; the Monitor continues on to other tasks.

If the frame is resident, Register Zero is attached to byte zero (unlinked format) of the frame, and this main memory address is saved in a hardware memory register (inaccessible to the programmer). The hardware register is then used to reference all other PCB elements, including the other address registers when they are detached. Register One is attached first, and the other registers are attached as needed by the program.

Note that although Address Register Zero is stored in the process' PCB, it is not actually used at all, since its displacement field is always assumed to be zero and the FID is supplied by the Monitor.

The format of the PCB is described later in the chapter SYSTEM CONVENTIONS.

3.6 Register One

Address Register One has two distinct formats, depending on whether the process is active or inactive. In the inactive state, Register One 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 attached to byte zero of the program frame that the process is currently executing. The real program counter, which actually addresses the next instruction that the process will execute, is stored in a special hardware register and is inaccessible to the programmer.

The purpose of this peculiarity is that since Register One always addresses byte zero of the current program frame, data in that frame

may be referenced relatively using Register One as a base (see the section on Addressing Modes below). This is the mechanism used to address literal text and other data in the program frame.

3.7 Registers Two through Fifteen

Registers Two through Fifteen have no hardware defined meanings, and thus are general purpose registers. But the system software conventions assign Registers Two through Thirteen to specific locations.

Register Two points to another control block, called the Secondary Control Block or 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 format of the SCB and the conventions regarding Registers Three through Fifteen are described later in the chapter SYSTEM CONVENTIONS.

3.8 Addressing Modes

The system has four modes of addressing data using the address registers.

DEFINITIONS:

1. Immediate addressing - The datum is in the instruction itself (literal), and can be only one byte in length. For example, MCC C'A',R4 where the constant character "A" is stored within the instruction.

   Immediate mode operands are either a single byte to be moved, a masking field for logical operations or a byte used as a parameter in a variety of instructions.

2. Direct addressing - Reference is to the AR itself; for example, MOV R14,R15 where R14 is moved to replace the contents of R15, so that the two registers are then identical.

3. Indirect Addressing - Reference is to the byte or byte string addressed indirectly by the AR. There are three sub-modes in this section:

   a. Indirect byte: The addressed byte is located indirectly by using the byte address of the register.

   b. Indirect byte pre-incremented: The addressed byte is located indirectly by first adding one to the byte address of the register. The register remains altered.

   c. Indirect string pre-incremented: The register's byte address is successively incremented by one to generate the locations of a string of bytes. The length of the string is dependent on the exact instruction, which may specify one of several terminating conditions. The register is left addressing the last byte in the string.

4. Relative addressing - The field (variable length, see below) is addressed via a BASE register and an OFFSET (fixed in the instruction) to get the resultant address. That is, a function of the offset is added to the byte address of the AR to get the effective address. The function used is dependent on the actual field being addressed and is described later. Only forward addressing is allowed, and going beyond the boundary of the frame causes a CROSSING FRAME LIMIT abort condition.

3.9 Symbol Types

All symbols or variable names at the assembly level have an associated symbol type code. This code indicates the addressing mode of the variable.

Symbols may be predefined and stored in a file called PSYM. Local symbols which are defined within a program are stored in a file called TSYM for the duration of the assembly.

The table below describes the PSYM or TSYM symbol type codes.

Symbol Type code	Description and length	Addressing Mode	Unit of Offset	Max displacement from AR
B	A single Bit	Relative	Bits	32 bytes=256 bits
C	A character- 1 byte	Relative	Bytes	256 bytes
D	Double Tally- 4 bytes	Relative	Words	512 bytes
F	F-type Tally- 6 bytes	Relative	Words	512 bytes
H	Half Tally- 1 byte	Relative	Bytes	256 bytes
L	Local label	Relative	Bytes	256 bytes *
M	Mode identifier (External label)	-	-	-
N	Literal	Immediate		
R	Address Register	Direct/Indirect		
S	Storage Register- 6 bytes	Relative	Words	512 bytes
T	Tally- 2 bytes	Relative	Words	512 bytes

Table of Symbol Type Codes

* Local labels are subject to this limitation only in the SRA instruction, not when used as targets of a branch, in which case the branch is to an absolute location in the object code.

3.9.1 Computation of Location from Offsets

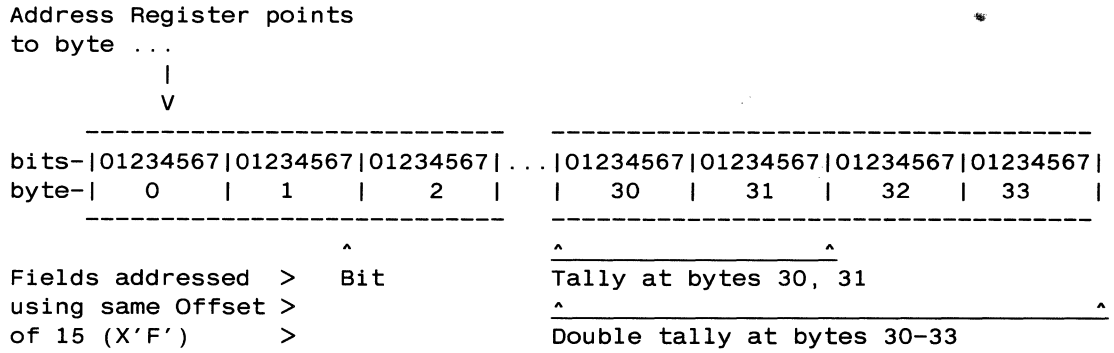
Symbols referenced in a Relative addressing mode are addressed via a base register and an, offset displacement. The resultant address may be offset up to the limit as mentioned in the table above, though it may not cross the boundary of the frame that the register is addressing.

Offsets are fixed in the instruction and are in the range 0-255. The offset value is derived from the symbol's definition in the symbol file.

The column "Unit of Offset" indicates the function used to convert the offset to the effective address.

For example, if the symbol being addressed is a BIT, the offset is also in bits, so that an offset of 15 would address the 7th bit in byte 1 displaced from the address of the AR. If the symbol is of type D, the same offset would address bytes 30 through 33 displaced from the address of the AR.

Graphically,



3.10 Description of Symbol Table Elements

Address registers (type R) and storage registers (type S) have already been described. Local labels (type L) and mode identifiers (type M) are described in the next chapter. The other symbol types are described below.

3.10.1 Bits

Any bit within the first 32 bytes offset from the byte address of a register may be addressed relatively. Bit instructions may set, zero or test a bit.

3.10.2 Characters

A type C element is a single character or byte addressed relatively using a base register and an offset. The difference between addressing a byte as a type C and addressing it indirectly is that in the latter case, the register must point to the byte itself; in the former, it may point up to 255 bytes before it (but in the same frame).

3.10.3 Counters or Tallies

Counters or TALLIES contain a signed (two's complement form) integer which may be used in arithmetic operations.

There are four types of counters: half tallies (Type H), 1 byte; tallies (Type T), 2 bytes; double tallies (Type D), 4 bytes; and F-type tallies (Type F), 6 bytes.

The half tallies are rarely used, since they can only store numbers in the range -128 through +127.

The tallies are used most frequently, since their range is -32,768 through +32,767.

Double tallies have a range of -2**31 through +2**31-1 and are typically used to store FID's (base FID of a file, for instance), and to count items in a file.

F-type tallies are used for any arithmetic that requires the full 48-bit precision of the system, and contain numbers in the range -2**47 through +2**47-1.

CHAPTER 4

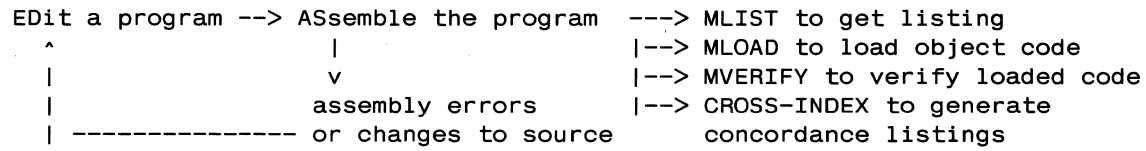
THE ASSEMBLER

4.1 Introduction

The ULTIMATE assembly language is a powerful high-level assembly language which has many instructions designed specifically for data base management.

4.1.1 The Assembler and Related Processors

The ULTIMATE Assembler translates source statements into ULTIMATE CPU machine language equivalents. The source program, or "mode" is an item in any disc file. The program is assembled in place; that is, at the conclusion of the assembly process, the item contains both the original source statements, as well as the generated object code. The same item can then be used to generate a formatted listing (using the MLIST verb) or can be loaded for execution (using the MLOAD verb). The diagram below illustrates the interaction of the various assembly functions:



The assembler 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 during this pass, along with literal definitions that need to be created.

At the end of pass one, the literals are generated and added to the end of the current object code. Pass two then re-assembles all the flagged instructions and concludes the assembly.

The assembly instructions generate object code that is variable in length, from one to six bytes. Each instruction may have zero to three explicitly defined operands. In addition, some instructions implicitly reference Address Register 15 or the accumulator. The section describing each instruction mentions such "side effects" in detail.

4.2 Editing a Source Item

Each line in the source item contains a single assembly statement. The general format of a source line is:

{Label} opcode operand{,operand,...} {comments}

The source item is entered via the system EDITOR, and is free form. Several commands are available in the EDITOR to display the source (and object, if any) in conventional assembly listing format, which makes editing and modification much easier:

Command	.....Description.....
AS	A switch which causes the assembly listing format to be turned ON or OFF. Each entry of "AS" toggles this switch. It may also be called into effect by using the "A" option when calling the EDITOR. When using this option, the format of the source as displayed should not be confused with the format used when inserting or changing lines, which is free form (described next).
M	A switch which causes the display of assembler Macro expansions to be turned ON of OFF. It is initially off.
S	A switch which causes the display of object code to be turned ON of OFF. It is intially on.
Q/locn/	Is used to locate the line with object code location "locn". This command is different from the L/string/ in that it searches only in the object code for the location matching that specified.

When a program is assembled, macro expansions and the generated object code are stored along with the source statement using system delimiters to separate the various components. While editing an already assembled program, any data beyond the source statement may be completely ignored, since the assembler examines only the source data on each line as it performs the assembly; existing object code, etc. are discarded.

4.2.1 Format of a Source Item

1. The assembler expects the assembler directive FRAME to be on the first line of the source item.

2. The next five lines (lines 2-6) are conventionally used for comments concerning the program type, revision level, author, etc. The assembler will automatically write the assembly date (as a comment) on line 3, unless this line is initially a non-comment line.

3. A sequence of unconditional branch (B) statements to the entry points (defined under "Mode-id" below) in the frame follow. This is called the "branch table."

4. There is no need to specify the END statement at the end of the program, though conventionally one is placed there for clarity.

4.2.2 Examples of EDIT Display

With the AS function off:

```
013 SAVEIT 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
```

(Note the spaces in the first column of lines 14 and 15.)

With the AS function ON, S function OFF and M function OFF:

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

With the AS function ON, S function ON, M function OFF:

```
013  SAVEIT   MCC  R4,R5     MOVE THE TERMINATOR
014           MCC  R4,R16    SAVE IT ALSO
015           B    OK
```

(Note that the error message on line 14 was also suppressed.)

With the AS function ON, S function ON, M function ON:

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

4.2.3 Labels

The optional label, if it exists, must start in column one, and must begin with an alphanumeric character. It may be up to 50 characters in length, though for listing formatting purposes, only ten spaces are reserved. Labels should not contain the following characters: *,/,+.

The label is separated from the opcode mnemonic by a space. If there is no label, at least one space must precede the opcode. If the label starts with an asterisk (*), the entire source line will be considered a comment and will be ignored by the assember.

Labels are locally defined symbols that are used to address locations in the program or other symbolic types. They must be used as the target of all instructions that execute a conditional or unconditional branch.

Examples of valid labels are:

```
LOOP
TEST.TOTAL
RESTART-X
RESTART101
```

4.2.4 Opcodes

The opcode is separated from the label and the operand(s) by at least one space. The legal opcodes are defined in the OSYM file, which is described later.

An opcode may be a machine instruction, a macro definition that expands to a set of machine instructions, or an assembler directive.

Examples of opcode mnemonics are:

```
MOV
BCHE
EQU
```

A specific opcode mnemonic does not determine the actual instruction, since the latter is dependent on the operands used with the opcode. The MOV opcode, for example, allows a number of different operands, and each combination produces a different machine instruction or instructions.

4.2.5 Operands

The operands for the instruction follow the opcode and are separated from it by at least one space.

An operand may be one of the following:

1. A literal in one of the following forms:

   a. A single printable character or a text string: C'x' or C'ABCD'. If a single-quote is needed as a literal, two adjacent single quotes must be used; for example, to represent the string JOHN'S, the operand would be C'JOHN''S'.

   b. A decimal integer: n; for example, 12 or -1234.

   c. A hexadecimal constant: X'xxxx'; for example, X'FE' or X'8100FF'. If an odd number of hex characters are used, a leading zero is assumed to fill the left most nybble.

2. A symbol as predefined in the PSYM file, or as defined in the label field of the source program.

3. The "current location" function, *. This function is used to specify the current location or address being assembled. The assembler maintains a byte location counter which is the the location of the first byte of the current instruction being assembled. This location advances as instructions are assembled and can be altered only by the assembler directive ORG (origin). Specific forms of this function are:

   * Returns the current location in bytes.

   *n Returns the current location in units of "n" bits. For example, *1 would return the location in bits, *8 is identical to *, and *16 returns the location in words.

4. A combination of literals or the * function combined with a + or a -. Symbols cannot be used in such combinations.

Multiple operands are separated from each other by a comma; no spaces are allowed within this field except in quoted character literals.

Examples of operand fields are:

```
        MOV   100,COUNTER

        MOV   X'64',COUNTER

LABEL   EQU   *-1

        BCHE  R15,C'A',OK.TO.GO
```

4.2.6 Comments

The optional comment field follows the last operand and is unrestricted in length, though again the listing processor will truncate comments at the end of the defined line length.

4.3 Assembled Object Code

There is no link-editing mechanism in the assembler. A program loads into a specific frame, and must be written with the following limitations in mind:

1. The assembled object code must be less than or equal to 512 bytes in length (one frame).

2. The frame into which the program is to load must be explicitly specified with the FRAME assembler directive.

3. All interframe linkages must be explicitly established and coded. Branches outside the current frame use different opcodes than local branches, though the mnemonic for subroutine calls is identical whether the destination is local or in another frame.

4. The first executable location in a frame is the byte at location one (unlinked format), not zero. The FRAME assembler directive also sets the assembler's location counter to one for this reason. Byte zero can be used for storage (remember Address Register One points there); to do so, the ORG assembler directive must be used to reset the location to zero and store a byte there.

4.3.1 The Mode-id

DEFINITION: A MODE-ID is a sixteen-bit field (therefore a tally can store it) which has a four-bit entry point and a twelve-bit FID. It is an encoded address to which execution control can be transferred via the ENT (external branch) or BSL (external subroutine call) instructions. The actual location addressed is twice the entry point number plus one.

Up to sixteen entries to a frame of object code are allowed; typically there are unconditional branch (B) instructions forming an entry table (called the "branch table") at the beginning of each program. This allows the program body to be changed and reassembled, without affecting the entry points - an important concept.

Strictly speaking, for safety, there should be sixteen branches even if not all of them are used; in practice, only as many branches as are being used need be present.

A mode-id may be defined by the DEFM assembler directive, which defines a symbol, or by the MTLY directive, which defines a symbol and reserves storage in the program for the mode-id. For example,

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

defines the symbol EXT.SUB as a mode-id whose value is entry point four in frame 500, and is therefore byte nine in that frame.

Following are further examples of the relationship between the mode-id and the value of the resultant address:

Mode-id	Entry point	Addressed Location (FID.location)
01FF	0	511.1
11FF	1	511.3
21FF	2	511.5
....		
F1FF	F (15)	511.1E

A typical sequence at the beginning of an assembly program is shown below:

```
Line Object code   Label    Opcode  Operand(s)     Comments

001  0001 7FFF01FF          FRAME 511      Establish frame# for MLOAD
002               * Note: lines 2-6 are comments
003               *21 MAY 1983
004               *
005               * Note: the sequence of unconditional branch
006               * instructions (B), below, must be the first
007               * ones that generate object code.
008  0001 1E34            B     START    Entry point 0 - location 1
009  0003 1F22            B     CONT     Entry point 1 - location 3
010  0005 1E88            B     SUBR1    Entry point 2 - location 5
011               * ---- end of branch table  --------
012  0007 31      CHAR.A  CHR   C'A'
013  0008         START   EQU   *        First inst. For entry 0
etc.
```

4.4 Usable Frames

Since code has to be loaded into a specific frame, the user should be extremely careful to ensure that the selected frame is free. It is almost, but not quite impossible to determine from a DUMP of a frame whether any legal object code exists in it or not; sometimes the disc formatter leaves a readily recognizable pattern in unused frames.

Note that the frame number is only twelve bits, and therefore executable object code can only be loaded in frames 1-4095.

User written code can reside in frames 400 through 599*; but widely used utilities and routines cut down on the available space.

* Specific to release level; check with ULTIMATE to be sure.

4.5 Calling the Assembler

The assembler is called by the statement:

```
AS program-file-name {item-list} {(options)}
```

which will assemble the item(s) in the file specified. Since the AS verb is a TCL-II type, the "item-list" above may be one or more explicit item-ids, an "*" to specify all items in the file, or may be omitted if the AS statement is preceded by a SELECT, SSELECT, GET-LIST or QSELECT statement. See the RECALL manual for a complete explanation of the interface with these statements.

The AS verb requires the three files OSYM, PSYM, and TSYM to be defined for the user's account. OSYM and PSYM are typically Q-pointers to the ULTIMATE-supplied OSYM and PSYM files, but TSYM must be created for each account. These files are discussed in more detail in later sections.

If there are any assembly errors, the assembler will enter the EDITOR so that the program may be conveniently corrected for reassembly.

"options" are enclosed in parentheses and are single alphabetic characters as shown in the table below:

Option	Description
E	When used in conjuction with the L option, specifies that an "errors-only" listing is to be produced.
L	Generates a listing (equivalent to the MLIST verb) during assembly.
N	Inhibits waiting at end-of-page during listing (for multiple-item assembly and/or hardcopy terminals); useful in conjunction with Z option.
P	Routes listing output to the print spooler.
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, and the assembly is not to stop on encountering an error in any item.

As the assembler processes, it outputs an asterisk (*) as every ten source statements are assembled. At the end of pass one a new line is started and an asterisk is printed for each ten statements reassembled.

Examples:

```
AS SM PROG1
******************            < output from assembler
***
[236] NO ERRORS

AS SM PROG1 PROG2 (L,P)
```

4.6 Listing Output

The listing processor may be called by the statement:

```
MLIST program-file-name {item-list} {(options)}
```

where item-list and option formats are as specified for the AS verb.

Options are:

Option	Description
E	Prints error lines only; also enters the EDITOR if any errors are found.
M	Prints macro expansions of source statements.
N	Inhibits stop at end of page if listing is to terminal.
P	Routes output to the print spooler.
S	Suppresses the display of object code.
Z	Inhibits EDIT entry when E option is specified.
n-m	Restricts listing to line numbers n through m inclusive.

The listing is output with a statement number, location counter, object code and source code, with the label, op-code, operand and comment fields aligned. A page heading is output at the top of each new page.

Errors, if any, appear in the location counter/object code area; macro expansions appear as source code if not suppressed, with the operation codes prefixed by a plus sign (+).

Examples:

```
MLIST SM PROG1 (P)
```

```
SELECT SM WITH CLASS "RECALL"
MLIST SM (P,M)
```

4.7 Assembly Errors

Assembly errors are stored along with the source line causing the error. If undefined symbols exist, the last line of source will also have a list of undefined symbols stored as a message. If any assembly errors are found, the Editor is entered as a convenience to correct the source, unless the Z option was used in the assembly.

Error 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.
MUL-DEF	The label is multiply defined.
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 location X'1FF'.
UNDEF: xxx {,xxx..}	List of undefined symbols found.

The following additional error messages :

FRMT. A-FIELD FRMT. B-FIELD OPCD TYP MACRO DEF

are due to errors in the OSYM file definitions.

4.8 Loading a Program Mode

The assembled mode may be loaded into the frame specified by the FRAME opcode by using the statement:

```
MLOAD program-file-name {item-list} {(options)}
```

If the load is successful, the message:

```
[216] MODE 'item-id'      LOADED;   FRAME = nnn  SIZE = sss  CKSUM = cccc
```

is returned, where

nnn	is the three-digit number of the frame into which the mode has been loaded. The number nnn is expressed in decimal.
sss	is the number of bytes of object code loaded into the frame, expressed in hexadecimal (base sixteen) notation.
cccc	is the byte checksum for the object code in the loaded mode.

The mode will not load correctly if its size exceeds 512 bytes, or if a FRAME statement is not the first statement assembled in the mode. In either case, a message will be returned indicating the error.

Options are:

Option	Description
E	Only messages relating to errors will be printed
I	Item-id's will be printed if more than one is MLOADed
N	Load inhibited but message printed.
P	Routes message output to the print spooler.

4.9 Verifying a Loaded Program Mode

The MVERIFY verb may be used to verify previously loaded object code against the assembled source item. Its format is:

```
MVERIFY program-file-name {item-list} {(options)}
```

Options are:

Option	Description
A	See below
E	Only messages relating to errors will be printed
I	Item-id's will be printed if more than one is MVERIFYed
P	Routes message output to the print spooler.

The A option may be used to display All error bytes.

Examples:

```
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
```

The first example verifies, but the second does not. In example 2, the system informs the user that the first byte at location 14 should have a value of 0C, not 18. The other mismatching bytes are not displayed.

If the A option is used, each byte in the source file which mismatches will be listed, followed by the value in the executable frame. For example:

```
MVERIFY SM EXAMPL2 (A)

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

[218] MODE 'EXAMPL2'    FRAME = 511  HAS  78  MISMATCHES
```

4.10 Symbols

A symbol is a named reference to one of the fields that can be addressed by the system. The symbol name is of the same format and has the same restrictions as the previously defined "label" field.

A symbol may be:

1. A globally defined symbol, stored in the Permanent Symbol Table file (PSYM).

2. A locally defined symbol, one that appears in the label field of the current program. A symbol may either be merely defined in the program for local usage, or may also reserve storage in the object code (such as literals).

   For example, the instruction:

   ```
   COUNTER   DEFT  R4,5
   ```

   defines COUNTER as a symbol of type T, with a specific base register of 4 and an offset of 5; whereas the instruction:

   ```
   COUNTER   TLY   1234
   ```

   defines it implicitly at the current location in the object code, and stores a value of 1234 at that location in the object code. This is now a literal in the program.

3. 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.

   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 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.

4. An immediate symbol, defined in a later section.

4.11 The PSYM File

The PSYM is a file that contains the set of permanent or global symbols available to all assembly programs. While symbols in the PSYM may be redefined locally in a program, it is best to treat all the symbols in the PSYM as reserved.

Entries in the PSYM file have two or three attributes of data, and the general format is described below.

The first line in an entry is the SYMBOL TYPE, which is described in the previous chapter. The symbol type is a single alphabetic character which determines the method used by the system to address the field and determines its usage and length, if applicable.

The various symbol types are described in the following table:

Item-id:	symbol-name	symbol-name	symbol-name	symbol-name
Line 1 (type)	B/C/D/H/L/F/S/T	R	M	N
Line 2 *	Offset	Register number	Entry point number	Literal value
Line 3 *	Base register number	(unused)	Frame number	(unused)

* - values are in hexadecimal.

4.12 The TSYM File

The TSYM is a file that contains the set of symbols local to a program. It is always cleared by the assembler before the start of each assembly.

As the assembler finds labels 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.

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.

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

Due to the method that the assembler uses in generating literals, a program loaded and then reassembled with a different TSYM MODULO will not MVERIFY, even though the source statements are identical.

4.13 CROSS-INDEX Verb

The TCL-II verb CROSS-INDEX is used to create a cross-reference of all symbols used in a program or a set of programs. It requires a file called CSYM. The format is:

```
CROSS-INDEX programe-file-name item-list {(options)}
```

The result of the verb is a set of items in the CSYM file. For each item, the item-id is the program name; each of the next ten lines contains multi-valued references to symbols of a specific type.

Attribute#	Contains references to symbols of 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-id's
10	N	Literals or constants

Symbol references are only checked in the PSYM file. To cross-reference local definitions (such as from an INCLUDEed program) as well as the standard global definitions, a temporary PSYM file containing both the global and local definitions must be created. This is best done on an account other than SYSPROG, to avoid destroying the standard PSYM. All items in the regular PSYM file should be copied into the temporary PSYM. Then the INCLUDEd program should be assembled, and all items in the TSYM file copied into the temporary PSYM.

Example of CROSS-INDEX:

```
CROSS-INDEX MODES *
```

This statement will cross-index all items of the MODES file.

Below is an example of an item in the CSYM file. The item-id, DLOAD, is the name of a program. The numbers following the symbol names are the number of times that the symbol is referenced.

```
    DLOAD
001 LISTFLAG 001]RMBIT 002
002 CH8 001
003 NNCF 002
004 CTR1 002]MODULO 007]OBSIZE 001]RSCWA 001]SEPAR 010]T0 001]T4 003
005 BASE 008]DO 001]OVRFLW 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]LINK 001]MBDNSUB 003
010 AM 002
```

4.14 X-REF Verb

The TCL-II X-REF verb uses the CSYM file as updated by the CROSS-INDEX verb for input. X-REF creates a cross-reference listing by symbol name in the XSYM file, with program names as data (as opposed to the CSYM file entries, which are by program name, with symbol names as data). For each XSYM item, the item-id is the symbol name; the only attribute is a multi-valued list of program names. The XSYM may be listed to produce a "where-used" listing of symbols.

The format is:

```
X-REF CSYM-file-name item-list {(options)}
```

Example:

```
X-REF CSYM *
```

would cross reference all items of the CSYM file.

```
SORT XSYM REFERENCES NONCOL (P)
```

would produce an alphabetical non-columnar listing on the line printer.

The following is an example of a partial listing:

```
XSYM : ABIT
REFERENCES EDIT-I   EDIT-II  EDIT-III

XSYM : AF
REFERENCES ASTAT  WRAP-III  EDIT-I  EDIT-III
```

4.15 The OSYM File

The OSYM file contains the set of defined opcode mnemonics. The item-id of an entry in this file has one of two forms:

1. The opcode mnemonic itself; for example, B for Branch.

2. The opcode mnemonic concatenated with the symbol type of each operand. For example:

```
          Source line              Resultant OSYM entry
          -------------------      --------------------

          MOV    R14,R15           MOVRR
Symbol types ->   R   R

          MOV    ISBEG,IS          MOVSR
Symbol types ->   S     R
```

   The purpose of this is both to distinguish between different opcode-operand combinations which may generate completely different machine instructions, as well as to validate the operand list. For example, the MOV opcode with operands of types B and H would result in an OSYM file lookup of MOVBH, which is nonexistent and therefore invalid.

4.15.1 Format of OSYM file entries

Line one of each OSYM file item contains a code. The 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 in the OSYM file to continue processing. This is used to "link" from one item to another to save duplicate definitions.

4.15.2 Argument Field

For the purpose of the next few sections, the terminology "argument field" (AF) is used to refer to the fields in the original source statement being assembled. The label is addressed as AF(0), the opcode as AF(1), and operands as AF(2) through AF(9), if they exist.

For example, in the following source statement:

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

the following AF equivalences are established:

```
AF(0) --  LOOP
AF(1) --  BCE
AF(2) --  C'A'
AF(3) --  STOPIT
```

4.15.3 Primitive Definition Lines

Each line in a Primitive OSYM definition is as follows:

```
G,a1,a2,... b1,b2,...                  (The "a" and "b" fields
   or                                   are separated by one space)
R,a1 b1,b2...
```

The G line performs the actual object code generation. There is a one-to-one correspondence between the "a1", "a2", etc., and the "b1", "b2". 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.

The "b" field expressions may be:

n	A decimal constant.
X'hh'	A hexadecimal constant.
=c	A single byte character constant.
*{n}	Current location counter, where n is as defined previously in the section on Operands.
An;m	References AF(n), and returns the value from the m-th line of the symbol file defintion if it is a symbol, or the value of the literal if it is a constant.
B	Current "base" register (see "Literals" below).
Exxxx	Exit to assembly subroutine whose mode-id is xxxx.
Jn	Returns branch (or jump) address of local label referenced by AF(n).

The R line is used to redefine a temporary symbol file entry. The symbol file item is referenced using AF(a1) (normally, a1 is zero, to reference the label field of the source statement). Successive lines in the symbol file entry are replaced with the data generated by the expressions "b1", "b2", etc.

4.15.4 Macro Definitions

A Macro definition has the character "M" on line one 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:

(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 will create 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.

For example, suppose a new instruction is to be created with the format:

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

where the instruction tests "x" (a signed integer) to see if it is in the range between "low" and "high", and branches to "label" if so. 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
                                (Macro label count = 14 at start)
RANGETTTL
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 this is in the label field because no space
      precedes the "(".
```

4.15.5 Examples of OSYM Entries

Example 1.

```
Original source line:                          MCI  SCO,R11

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

Resultant macro source statements:             INC  R11
                                               MCC  SCO,R11
```

Example 2.

```
Source line:                                   MCC  SCO,R11

PSYM file entries:                                 SCO       R11
                                             001 C     001 R
                                             002 3     002 B    (=11 decimal)
                                             003 0
OSYM file entry:     MCCCR
                 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	SCO	0
8	A2;2	SCO	03
4	1	-	1
4	A3;2	R11	B

```
Final result:                                        D0031B
```

Example 3:

```
Source line:                        NEW   DEFH  4,5

PSYM file entries:                        (none)
OSYM file entry:     DEFHNN
                 001 P
                 002 R,0 =H,A3;2,A2;2
TSYM file entry:
               Before instruction*       After instruction
                    NEW                      NEW
                 001 L                    001 H
                 002 xxxxxxxx             002 5
                 003 1                    003 4
```

* Note - symbol NEW in the TSYM was stored as type L (for label), offset equal to the current location (shown as "xxxxxxxx"), and base register of 1, before the instruction redefined it.

4.16 Literals

The assembler will automatically assemble certain types of literals. Such literals are fields that can be addressed via a base register and an offset displacement. When a program is executing, Address Register One points to byte zero 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.

Tallies of type T and D may be generated. The reason that half tallies are not is that half tallies 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 will cause a truncation error if generated. F-type 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.

The mechanism used to generate literals is as follows:

An instruction that needs a literal must be a macro that references a symbol of the form:

```
=xvalue
```

where "x" is a T or D (symbol type). 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", which actually generate the literal and redefine the symbol to the correct type and location.

Examples of literal generation are on the next page.

```
------------------------ Step 1 --------------------------

Source line:                                MOV 100,COUNTER

PSYM file entries:                          COUNTER
                                        001 D
                                        002 1F
                                        003 0
OSYM file entry       MOVND
                  001 M
                  002  MOV =D(2),(3)

------------------------ Step 2 --------------------------

Resultant macro source statements:          MOV =D100,COUNTER

TSYM file entry:                            =D100
                                        001 U
                                        002 0
                                        003 1

OSYM file entry:     MOVDD
                 001 P
                 002 G,4,4,8,4,4,8 15,A3;3,A3;2,8,A2;2,A2;2

Resultant object code:                      F100801F
[Flagged for reassembly because symbol =D100 is undefined]

------------------------ Step 3 --------------------------

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

Source line:                          =D100 :D 100

OSYM file entry:     :D
                 001 P
                 002 E:101B   <-- Forces Word alignment in object code
                 003 R,0 =D,*16,B
                 004 G,32 A2;2 <-- Generates double tally object code

Resultant object code:                 00000064

TSYM file entry:
               Before instruction        After instruction*
                    =D100                     =D100
                 001 U                     001 D
                 002 0                     002 xxx
                 003 1                     003 1

 * where "xxx" is the offset appropriate to the current location.

------------------------ Step 4 --------------------------

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

4.17 Immediate symbols

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

There is an ability to define an "immediate symbol" as an operand. The purpose of this is when a symbol is only used once and it is simpler than having to define the symbol in a separate line. These symbols are not recommended, however, except to reference bits, since they have a quirk in their syntax that makes them different from the PSYM/TSYM equivalents. They are documented here for compatibility only.

The general form of an immediate symbol is:

```
Rn;xm
```

where Rn is a base register designator (R0 - R15);

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

m is a decimal value that generates the offset displacement:

```
Offset displacement = 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). But R2;D10 addresses the double tally displaced from R2 at bytes 40 through 43 (not 20 through 23).

Following are examples of immediate symbols and their equivalent DEFinition instructions (see the DEFx assembler directive in the chapter on the Instruction Set):

```
Immediate  Displacement from    Equivalent DEF
Symbol     Base Register        Instruction

 R0;B0                 0       HIBIT    DEFB  R0,0
 R15;B7                7       LOBIT    DEFB  R15,7
 R2;C100             100       CHARACT  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       FTALLY   DEFF  R0,45 !!!
```

CHAPTER 5

INSTRUCTION SET

5.1 Introduction

This chapter documents the set of assembly instructions. The instructions are in alphabetical order, and are summarized in the next section by function.

Instruction formats show the opcode mnemonic in capitals, with the operand types in lower-case letters. An instruction is determined only in conjunction with its operand types. The MOV mnemonic, for instance, produces a number of different object code equivalents dependent on the exact operand combination. Operand types correspond to the SYMBOL TYPES that were described in the last chapter. They are summarized below:

Symbol type code	Description
b	A relatively addressed BIT
c	A relatively addressed CHARACTER
d	A relatively addressed DOUBLE TALLY (32 bits)
f	A relatively addressed F-TALLY (48 bits)
h	A relatively addressed HALF TALLY (8 bits)
l	A locally defined label
m	A MODE-ID (16 bits)
n	A constant or literal (variable)
r	An ADDRESS REGISTER
s	A STORAGE REGISTER
t	A relatively addressed TALLY (16 bits)

The symbols SM, AM, VM, and SVM denote the "system delimiters" Segment Mark (X'FF'), Attribute Mark (X'FE'), Value Mark (X'FD'), and Sub-value Mark (X'FC'), respectively.

5.2 Summary of Instructions

1. Bit instructions

BBS		Branch if bit is SET (one)
BBZ		Branch if bit is ZERO
MOV		Move bit operand.1 to bit operand.2
SB		Set bit to one
ZB		Clear bit to zero

2. Single character instructions

AND		Logical AND of character
BCA	BCNA	Branch if operand.1 is (is not) alphabetic (A-Z, a-z)
BCE		Branch if operand.1 is EQUAL to operand.2
BCH	BCHE	Branch if operand.1 is HIGHER than (or EQUAL to) operand.2
BCL	BCLE	Branch if operand.1 is LESS than (or EQUAL to) operand.2
BCN	BCNN	Branch if operand.1 is (is not) numeric (0-9)
BCU		Branch if operand.1 is UNEQUAL to operand.2
BCX	BCNX	Branch if operand.1 is (is not) hexadecimal (0-9, A-F)
MCC		Move operand.1 to operand.2
MCI		Move pre-incremented operand.1 to operand.2
MDB		Convert one ASCII binary character
MIC		Move operand.1 to pre-incremented operand.2
MII		Move pre-incremented operand.1 to pre-incremented operand.2
MXB		Convert one ASCII hexadecimal character
OR		Logical OR of character
READ		Read asynchronous channel buffer
SHIFT		Right shift of character
WRITE		Write asynchronous channel buffer
XCC		Exchange characters
XOR		Exclusive OR of character

3. String character instructions

BSTE		Branch if delimited strings are EQUAL
MBD		Convert binary to decimal ASCII
MBX	MBXN	Convert binary to hexadecimal ASCII
MIID		Move string until DELIMITERs
MIIDC		Move string until DELIMITERs, counting bytes
MIIR		Move string until REGISTER address equivalence
MIIT		Move string until TALLY runout
MIITD		Move string until DELIMITERs or TALLY runout
MFD	MSDB	Convert decimal ASCII string to binary
MFX	MSXB	Convert hexadecimal ASCII string to binary
SICD		Scan string over multiple DELIMITERs
SID		Scan string until DELIMITERs
SIDC		Scan string until DELIMITERs, counting bytes
SIT		Scan string until TALLY runout
SITD		Scan string until DELIMITERs or TALLY runout

4. Register instructions

DEC		Decrement register by ONE or by operand
DETO	DETZ	Detach a register
BE		Branch if register.1 is EQUAL to register.2
BU		Branch if register.1 is UNEQUAL to register.2
FAR		Force attachment of a register
INC		Increment register by ONE or by operand
LAD		Load absolute address difference
MOV		Move register operand.1 to register operand.2
SRA		Set register to address
XRR		Exchange address registers

5. Arithmetic instructions

BDHZ	BDHEZ	Branch if operand.1, decremented by ONE or by operand.2, is HIGHER than or EQUAL to ZERO
BDLZ	BDLEZ	Branch if operand.1, decremented by ONE or by operand.2, is LESS than or EQUAL to ZERO
BDNZ	BDZ	Branch if operand.1, decremented by ONE or by operand.2, is NONZERO or ZERO
BE		Branch if operand.1 is EQUAL to operand.2
BH	BHE	Branch if operand.1 is HIGHER than (or EQUAL to) operand.2
BHZ	BHEZ	Branch if operand is HIGHER than (or EQUAL to) ZERO
BL	BLE	Branch if operand.1 is LESS than (or EQUAL to) operand.2
BLZ	BLEZ	Branch if operand is LESS than (or EQUAL to) ZERO
BNZ	BZ	Branch if operand is NONZERO (or ZERO)
BU		Branch if operand.1 is UNEQUAL to operand.2
ADD	ADDX	Add operand into accumulator
DIV	DIVX	Divide accumulator by operand
DEC		Decrement operand by ONE, or operand.1 by operand.2
INC		Increment operand by ONE, or operand.1 by operand.2
LOAD	LOADX	Load accumulator from operand
MOV		Move operand.1 to operand.2
MUL	MULX	Multiply accumulator by operand
NEG		Negate operand
ONE		Set operand to ONE
STORE		Store accumulator in operand
SUB	SUBX	Subtract operand from accumulator
ZERO		Set operand to ZERO

6. Control instructions

B	Branch to local label
BSL	Subroutine call to local or external label
BSL*	Subroutine call indirect to external label
BSLI	Subroutine call indirect to external location
ENT	Branch to external label
ENT*	Branch to external label indirect
ENTI	Branch to external location indirect
HALT	Halt program
NOP	No operation
RQM	Release process' time quantum
RTN	Return from subroutine
SLEEP	Put process to sleep until specified time

7. Assembler directives

ADDR	Create storage of type S
ALIGN	Align location counter on word boundary
CHR	Create storage of type C
CMNT	Insert comment
DEFx	Define symbol of type x
DTLY	Create storage of type D
EJECT	Eject a page in the MLISTing
END	Indicate end of program
EQU	Equate literal to label, or two symbols
FTLY	Create storage of type F
FRAME	Define Frame number for object code loader
HTLY	Create storage of type H
INCLUDE	Include a program for shared symbol definition
MTLY	Create storage of type M
ORG	Reset assembler's location counter
SR	Create storage of type S
TEXT	Store textual data
TLY	Create storage of type T

8. Miscellaneous

SET.TIME	Set system time and date
TIME	Get system time and date

5.3 ADD ADDX - Add to Accumulator

ADD
ADDX

Function: Accumulator = accumulator + relative.operand

These instructions add the contents of the operand to the accumulator.

Arithmetic overflow or underflow cannot be detected.

The following shows the format of the accumulator and the symbolic names that address various sections of it:

```
Byte no.: |  8 |  9 |  A |  B |  C |  D |  E |  F |
in PCB    |         | <---------- FP0 ---------> | <- 6-byte field
          | <------ D1 -----> | <------ D0 -----> | <- 4-byte fields
          | <- T3 ->| <- T2 ->| <- T1 ->| <- T0 ->| <- 2-byte fields
          |  H7|  H6|  H5|  H4|  H3|  H2|  H1|  H0| <- 1-byte fields
       B63 ..................................... B0  <- bits
```

ADD

The operand is added to the four-byte field D0. One- and two-byte operands are internally sign-extended to form a four-byte field before the operation takes place. Neither the original operand nor other sections of the accumulator are affected.

ADDX

The operand is added to the six-byte field FP0. One-, two-, and four-byte operands are internally sign-extended to form a six-byte field before the operation takes place. Neither the original operand nor other sections of the accumulator are affected.

Formats:

```
ADD  d
ADD  h
ADD  n    *
ADD  t

ADDX d
ADDX f
ADDX h
ADDX n    *
ADDX t
```

* Note: These instructions using a literal normally generate a two-byte field. If the literal is outside the range -32,768 through +32,767, an operand of the form =Dxxxx should be used to generate a four-byte literal (for example, =D40000 or =DX'FC000022'). Six-byte literals must be separately defined using the FTLY instruction.

5.4 ADDR Assembler Directive

ADDR

Function: All symbols or variable names used as operands must have a symbol type-code; this is an assembler directive that reserves storage and sets up the symbol in the label field to be of type S (Storage Register).

ADDR also generates an UNLINKED byte address. The first operand is used to specify the displacement of the generated byte address, and the second the FID or frame number. See the section in the chapter on Data Addressing for a full description of linked and unlinked modes of addressing; also compare to the SR assembler directive.

Format:
symbol ADDR n,n

Example:

```
Instruction                  Generated value

F100U   ADDR  1,100          0001 8000 0064 Address is in unlinked mode;
                                            location is 1 in frame 100.
                                  ^ Note high-order bit (unlinked mode)
                                    flag is set.

        MOV   F100U,R15      This sets R15 to the above address.
```

5.5 ALIGN Assembler Directive

ALIGN

Function: This is an assembler directive that is used to align the assembler's location counter on a two-byte word boundary. If the location counter is not on a word boundary, a single byte of object code with a value of zero is generated. It is typically used before a section of DEFinitions of tallies, double tallies, etc., to ensure word alignment.

Note that the assembler automatically word-aligns literals that it creates itself.

Format: ALIGN

5.6 AND - Logical AND of a Byte

AND

Function: Indirect byte = indirect byte logically AND'ed with operand

The byte referenced by the first operand is logically AND'ed with the byte referenced by the second operand. The byte referenced by the second operand is unchanged.

Formats: AND r,n
AND r,n

5.7 B - Local Branch Unconditionally

B

Function: Tranfers control unconditionally to local label.

The operand of this instruction must be a label that is defined in the current program frame. To transfer control to an external label, see the ENT instruction.

Format: B l

5.8 BBS BBZ - Test a Bit

BBS
BBZ

BBS

Function: If bit = 1, branch.

If the referenced bit is "set" (1), a branch is taken to the local label "l".

Format: BBS b,l

BBZ

Function: If bit = 0, branch.

If the referenced bit is "off" (0), a branch is taken to the local label "l".

Format: BBZ b,l

5.9 BCA BCNA - Test if Character is Alphabetic

BCA
BCNA

Function :
BCA : If indirect.character is alphabetic, branch.
BCNA : If indirect.character is not alphabetic, branch.

The character addressed by the address register is tested for the ASCII character ranges A-Z (X'41'-X'5B'), or a-z (X'61'-X'7B'). If it is (BCA) or is not (BCNA) in either range, a branch is taken to the second operand, which is a local label.

Formats: BCA r,l BCNA r,l

5.10 BCE BCU - Test Characters

Function :

BCE : If character.1 is equal to character.2, branch.
BCU : If character.1 is unequal to character.2, branch.

If the character addressed by the first operand is equal (BCE) or is not equal (BCU) to that addressed by the second, a branch is taken to the third operand, which is a local label.

Formats:

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

Note - a symbol of type C cannot be tested directly against a constant, a literal, or another symbol of type C. Logical equivalents to the instructions "BCE c,n,l", "BCE n,c,l", "BCE c,c,l", "BCU c,n,l", "BCU n,c,l", or "BCU c,c,l" may be coded in one of two forms:

1. Using an SRA instruction to address a C type as an indirect reference; for example:

```
          SRA   R15,SC1      Set R15 to address the C-type symbol
          BCE   R15,C'$',OK
```

2. Using a BE or BU instruction to compare a C type as a half tally, using DEFH and HTLY instructions where necessary to define symbols of type H. For example:

```
HSC1      DEFH  SC1          Define type H equivalent of SC1
HLIT$     HTLY  C'$'         Define a constant of type H
          ...
          ...
          ...
          BE    HSC1,HLIT$,OK
```

5.11 BCH BCHE BCL BCLE - Test Characters

BCH
BCHE
BCL
BCLE

Function :

BCH : If character.1 is higher than character.2, branch.

BCHE : If character.1 is higher than or equal to character.2, branch.

BCL : If character.1 is less than character.2, branch.

BCLE : If character.1 is less than or equal to character.2, branch.

The character addressed by the first operand is tested as an eight-bit 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').

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

Formats:

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

Note - a symbol of type C cannot be tested directly against a constant, a literal, or another symbol of type C. Logical equivalents to the instructions "BCH{E} c,n,l", "BCH{E} n,c,l", "BCH{E} c,c,l", "BCL{E} c,n,l", "BCL{E} n,c,l", or "BCL{E} c,c,l" may be coded in one of two forms:

1. Using an SRA instruction to address a C type as an indirect reference; for example:

```
          SRA   R15,SC1       Set R15 to address the C-type symbol
          BCH   R15,C'$',OK
```

2. Using a BH, BHE, BL, or BLE instruction to compare a C type as a half tally using DEFH and HTLY instructions where necessary to define symbols of type H. For example:

```
HSC1      DEFH  SC1           Define type H equivalent of SC1
HLIT$     HTLY  C'$'          Define a constant of type H
          ...
          ...
          ...
          BH    HSC1,HLIT$,OK
```

Note, however, that form 2 above will perform an arithmetic comparison. In an arithmetic comparison, the lowest half tally is -256 (X'80') and the highest half tally is 255 (X'7F'). In particular, the system delimiters SM, AM, VM, and SVM are logically higher than all regular ASCII characters, but are arithmetically lower.

5.12 BCL BCLE - See BCH

5.13 BCN BCNN - Test if Character is Numeric

BCN
BCNN

Function :
    BCN : If indirect.character is numeric, branch.
    BCNN : If indirect.character is not numeric, branch.

The character addressed by the address register is tested for the ASCII character range 0-9 (X'31'-X'39'). If it is (BCN) or is not (BCNN) in that range, a branch is taken to the second operand, which is a local label.

Formats: BCN r,l BCNN r,l

5.14 BCNA - see BCA

5.15 BCNN - see BCN

5.16 BCNX - see BCX

5.17 BCU - see BCE

5.18 BCX BCNX - Test if Character is Hexadecimal

BCX
BCNX

Function :
    BCX : If indirect.character is hexadecimal, branch.
    BCNX : If indirect.character is not hexadecimal, branch.

The character addressed by the address register is tested for the ASCII character ranges 0-9 (X'31'-X'39') or A-F (X'41'-X'46'). If it is (BCX) or is not (BCNX) in either range, a branch is taken to the second operand, which is a local label. Note that lower case characters "a" through "f" are NOT considered to be hexadecimal, and will fail this test.

Formats: BCX r,l BCNX r,l

5.19 BDHZ BDHEZ BDLZ BDLEZ - Decrement and Compare Against Zero

BDHZ
BDHEZ
BDLZ
BDLEZ

Function:

a. Relative.op.1 = relative.op.1 - 1 {or}
   relative.op.1 - relative.op.2

b. Then:
   BDHZ : If relative.op.1 is higher than zero, branch.
   BDHEZ: If relative.op.1 is higher than or equal to zero, branch.
   BDLZ : If relative.op.1 is less than zero, branch.
   BDLEZ: If relative.op.1 is less than or equal to zero, branch.

These instructions take the place of a DECrement followed by a Branch instruction, and are usually used in loop controls. For convenience, the form without a second relative operand is available, which always decrements by one.

Formats:

BDHZ	d,1	BDHEZ	d,1	BDLZ	d,1	BDLEZ	d,1
BDHZ	d,d,1	BDHEZ	d,d,1	BDLZ	d,d,1	BDLEZ	d,d,1
BDHZ	d,n,1	BDHEZ	d,n,1	BDLZ	d,n,1	BDLEZ	d,n,1
BDHZ	f,1	BDHEZ	f,1	BDLZ	f,1	BDLEZ	f,1
BDHZ	f,f,1	BDHEZ	f,f,1	BDLZ	f,f,1	BDLEZ	f,f,1
BDHZ	h,1	BDHEZ	h,1	BDLZ	h,1	BDLEZ	h,1
BDHZ	h,h,1	BDHEZ	h,h,1	BDLZ	h,h,1	BDLEZ	h,h,1
BDHZ	t,1	BDHEZ	t,1	BDLZ	t,1	BDLEZ	t,1
BDHZ	t,t,1	BDHEZ	t,t,1	BDLZ	t,t,1	BDLEZ	t,t,1
BDHZ	t,n,1	BDHEZ	t,n,1	BDLZ	t,n,1	BDLEZ	t,n,1

Example:

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

```
          MOV   COUNT,CTR1   Set loop counter for iterations
REPEAT    BDLZ  CTR1,QUITLOOP
          ...                |
          ...                | ... Body of loop
          ...                |
          B     REPEAT       Repeat the cycle
QUITLOOP  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 similar example shown for the BDZ and BDNZ instructions.

5.20 BDLZ BDLEZ - see BDHZ

5.21 BDZ BDNZ - Decrement and Compare Against Zero

BDZ
BDNZ

Function:

a. Relative.op.1 = relative.op.1 - 1 {or}
relative.op.1 - relative.op.2

b. Then:
BDZ : If Relative.op.1 is equal to zero, branch.
BDNZ: If Relative.op.1 is unequal to zero, branch.

These instructions take the place of a DECrement followed by a Branch instruction, and are usually used in loop controls. For convenience, the form without a second relative operand is available, which always decrements by one.

Formats:

```
BDZ   d,1          BDNZ  d,1
BDZ   d,d,1        BDNZ  d,d,1
BDZ   d,n,1        BDNZ  d,n,1
BDZ   f,1          BDNZ  f,1
BDZ   f,f,1        BDNZ  f,f,1
BDZ   h,1          BDNZ  h,1
BDZ   h,h,1        BDNZ  h,h,1
BDZ   t,1          BDNZ  t,1
BDZ   t,t,1        BDNZ  t,t,1
BDZ   t,n,1        BDNZ  t,n,1
```

Example:

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

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

Note that the body of the loop executes at least once with this logic; compare this to the example in the section on the BDHZ, BDHEZ, BDLZ, and BDLEZ instructions.

5.22 BE BU - Test Tallies

BE
BU

Function :

BE : If relative.op.1 is equal to relative.op.2, branch.
BU : If relative.op.1 is unequal to relative.op.2, branch.

If the first operand is equal (BE) or is not equal (BU) to the second, a branch is taken to the third operand, which must be a local label.

The two 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). Two- and four-byte literals are automatically generated by the assembler, but one- and six-byte literals are not; the latter must be manually coded using the HTLY or FTLY assembler directives.

Formats:

```
BE    d,d,1          BU    d,d,1
BE    d,n,1          BU    d,n,1
BE    f,f,1          BU    f,f,1
BE    h,h,1          BU    h,h,1
BE    n,d,1          BU    n,d,1
BE    n,t,1          BU    n,t,1
BE    t,n,1          BU    t,n,1
BE    t,t,1          BU    t,t,1
```

Note 1 - A literal or constant of zero should not be used. There are equivalent instructions that are more efficient and clearer that perform the comparison against zero. For example, "BZ CTR1,QUIT" should be used instead of "BE CTR1,0,QUIT".

Note 2 - A symbol of type H cannot be tested directly against a constant or literal. Logical equivalents to the instructions "BE h,n,1", "BE n,h,1", "BU h,n,1", or "BU n,h,1" may be coded in one of two forms:

a. Using the SRA instruction to address the H type as an indirect reference; for example:

```
        SRA   R15,H7       Set R15 to address the H-type symbol
        BCE   R15,10,OK
```

b. Defining an H type as a literal in the program; for example:

```
HLIT    HTLY  X'34'        Define a constant of type H
        ...
        BE    H7,HLIT,OK
```

Note 3 - A symbol of type F also cannot be tested directly against a literal. The FTLY instruction should be used to define a local literal.

5.23 BE BU - Test Registers

Function :

BE : If byte address from register.1 is equal to byte address from register.2, branch.

BU : If byte address from register.1 is unequal to byte address from register.2, branch.

These instructions compare the byte addresses of the two registers and branch on the result of the test, which can only be equal or unequal.

There is no way to determine which register is "less than" or "higher than" the other.

The byte address of a storage register must be normalized before use with these instructions, otherwise the comparison may not work correctly. See the comments under the FAR instruction.

Formats:

```
BE    r,r,l          BU    r,r,l
BE    r,s,l          BU    r,s,l
BE    s,r,l          BU    s,r,l
```

5.24 BH BHE BL BLE - Test Tallies

BH
BHE
BL
BLE

Function :

BH : If relative.op.1 is higher than relative.op.2, branch.

BHE : If relative.op.1 is higher than or equal to relative.op.2, branch.

BL : If relative.op.1 is less than relative.op.2, branch.

BLE : If relative.op.1 is less than or equal to relative.op.2, branch.

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, a branch is taken to the third operand, which must be a local label. The relative operands are compared as two's-complement (signed) integers.

The two 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). Two- and four-byte literals are automatically generated by the assembler, but one- and six-byte literals are not; the latter must be manually coded using the HTLY or FTLY assembler directives.

Formats:

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	BEL	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

Note 1 - A literal or constant of zero should not be used. There are equivalent instructions that are more efficient and clearer that perform the comparison against zero. For example, "BHZ CTR1,QUIT" should be used instead of "BH CTR1,0,QUIT".

Note 2 - A symbol of type H cannot be tested directly against a constant or literal. See the description under the BE instruction for examples of coding the equivalent of a "BL{E} h,n,l" or "BH{E} n,h,l", etc., but use only form (b), since (a) is a logical comparison.

Note 3 - A symbol of type F also cannot be tested directly against a literal. The FTLY instruction should be used to define a local literal.

5.25 BHZ BHEZ BLZ BLEZ - Compare Against Zero

BHZ
BHEZ
BLZ
BLEZ

Function :

BHZ : If relative.op.1 is higher than zero, branch.

BHEZ : If relative.op.1 is higher than or equal to zero, branch.

BLZ : If relative.op.1 is less than zero, branch.

BLEZ : If relative.op.1 is less than or equal to zero, branch.

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.

Formats:

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

5.26 BL BLE - see BH

5.27 BLZ BLEZ - see BHZ

5.28 BNZ - see BZ

5.29 BSL - Call a Subroutine

Function :

a. The return stack pointer is incremented by four, and the location less one of the instruction following the BSL is stored in the next entry in the return stack.
b. Control is transferred to location specified by the operand.

The operand of this instruction must be either a label that is defined as a mode-id (external entry point), or a local label. If it is a mode-id, the label may be predefined in the PSYM table as a symbol with a type code of M, or it may be defined with the DEFM assembler directive, either locally or in an INCLUDEd program.

Note that the same subroutine may be called either locally from the same 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 will execute slightly faster if the local label is used instead.

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. Also see the RTN instruction to return from a subroutine.

Return stack entries are four bytes each; their format is described in the chapter SYSTEM CONVENTIONS. An entry may be deleted from the return stack by the instruction "DEC RSCWA,4". This is mandatory if a subroutine is to be exited without using a RTN instruction. The entire return stack may be reset by the instruction "MOV X'184',RSCWA", 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.

Formats: BSL l
BSL m

Example of defining an external mode-id:

```
EXTS    DEFM  10,500      Define a constant of type M, which
        CMNT  *           is entry point #10 in frame 500.
        ...
        BSL   EXTS        Transfers control to frame 500, entry
        CMNT  *           point #10, which is location 21, or X'15'
        CMNT  *           Control returns when subroutine executes
        CMNT  *           a RTN instruction
```

Example of a local/external subroutine:

```
          FRAME 500
          ...
          B     EXT.S       Entry point for external call
          CMNT  *           part of "Branch table" at beginning
          ...
          ...
          BSL   EXT.S       Local call of same subroutine
          ...
          ...
EXT.S     EQU   *           Subroutine local label
          ...               Body of subroutine
          RTN
```

5.30 BSL* - Indirect Call to a Subroutine

Function:

a. The return stack pointer is incremented by four, and the location less one of the instruction following the BSL is stored in the next entry in the return stack.
b. Control is transferred to the location specified in the operand.

This instruction operates identically to the BSL instruction, except that the subroutine address is variable and is obtained from the operand.

This instruction is a macro that loads the accumulator from the operand, and then executes the BSLI instruction. See BSLI for a complete explanation.

Format: BSL* t

Warning: A side effect of this instruction is that it destroys sections of the accumulator.

5.31 BSLI - Indirect Call to a Subroutine

Function:

a. The return stack pointer is incremented by four, and the location less one of the instruction following the BSLI is stored in the next entry in the return stack.
b. Control is transferred to the location specified in T0.

This 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. See the BSL instruction for details of the subroutine linkage.

T0 must contain a mode-id, which may be loaded into it from a local label, an external label or by converting an ASCII string. Typically, the subroutine address is obtained from a table or from a file.

Format: BSLI

Example:

```
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 executes RTN
```

5.32 BSTE - Compare Delimited Strings

Function: If indirect pre-incremented string.1 is
equal to pre-incremented string.2, branch.

The two address register operands are incremented by one. The character addressed by the first operand is tested as an eight-bit 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:

1. One character is logically higher than or equal to the third operand, but the other is not - the instruction terminates with the strings considered unequal.

2. Both characters are logically higher than or equal to the third operand - the instruction terminates with the strings considered equal. Note that the terminating characters need not be the same, as long as they are both higher than the third operand.

3. The two characters are both less than the third operand, and are not equal - the instruction terminates with the strings considered unequal.

Format: BSTE r,r,n,l

Examples:

Instruction: BSTE R4,R5,X'FE',LABEL

```
Before instruction: R4 --v                    R5 --v
Data:                  |A |B |C |AM| ..          |1 |B |C |SM|5 |6 | ..
After instruction : R4 -----------^           R5 -----------^
```

Strings are considered equal, and a branch is taken to LABEL.

Instruction: BSTE R4,R5,X'FC',LABEL

```
Before instruction: R4 --v                    R5 --v
Data:                  |A |B |C |AM| ..          |1 |B |C |D |5 |6 | ..
After instruction : R4 -----------^           R5 --------------^
```

Strings are considered unequal, and no branch is taken.

Instruction: BSTE R4,R5,X'FC',LABEL

```
Before instruction: R4 --v                    R5 --v
Data:                  |A |B |D |AM| ..          |1 |B |C |D |5 |6 | ...
After instruction : R4 --------^              R5 --------^
```

Strings are considered unequal, and no branch is taken.

5.33 BU - see BE

5.34 BZ BNZ - Compare Against Zero

BZ
BNZ

Function :

BZ : If relative.op.1 is equal to zero, branch.
BNZ : If relative.op.1 is unequal to zero, branch.

These instructions are faster and clearer than the equivalent BE and BU instructions used with a literal of zero as one of the operands.

Formats:

```
BZ    d,l          BNZ   d,l
BZ    f,l          BNZ   f,l
BZ    h,l          BNZ   h,l
BZ    t,l          BNZ   t,l
```

5.35 CHR Assembler Directive

CHR

Function: All symbols or variable names used as operands have a symbol type-code; this assembler directive reserves storage and sets up the symbol in the label field to be of type C (Character). It merely stores the value of the operand if there is no entry in the label field.

Format:

```
symbol    CHR   n
```

5.36 CMNT Assembler Directive

CMNT

Function: Place a comment line in the source program

This 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. This directive may be used to align comments in the MLISTing. It may also be used to define a label as an alternative to the "label EQU *" form.

Format: CMNT comment line

5.37 DEC INC - Decrement or Increment by One

DEC
INC

Function :

DEC : Operand = operand - 1
INC : Operand = operand + 1

These instructions 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.

DEC or INC of symbol types D, F, H, T

If the operand is a tally, either one, two, four or six bytes in length, the contents of the operand are decremented or incremented by one.

Arithmetic overflow or underflow cannot be detected. For example, on a two-byte tally, the value -32768 (X'8000') will wrap around to 327678 (X'7FFF') using a DEC instruction.

Formats:	DEC	d	INC	d
	DEC	f	INC	f
	DEC	h	INC	h
	DEC	t	INC	t

DEC or INC of symbol type R

The byte address of the AR is decremented or incremented by one. If the resultant address crosses a frame boundary, and the register is in the linked mode, it will be normalized and will attach to the frame previous (DEC) or next (INC) in the linked chain.

If the beginning of the linked set is reached during the normalization process, the assembly Debugger will be entered with a trap condition indicating BACKWARD LINK ZERO.

If the end of the linked set is reached during the normalization process, the following action is taken:

a. If the exception mode identifier XMODE is non-zero, a subroutine call is executed to that address, to allow special handling of this condition.

b. 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.

Formats:	DEC	r	INC	r

5.38 DEC INC - Decrement or Increment One Operand by Another

DEC
INC

Function :

DEC : Operand.1 = operand.1 - operand.2
INC : Operand.1 = operand.1 + operand.2

DEC or INC of symbol types D, F, H, T

The contents of the first operand are decremented (DEC) or incremented (INC) by the contents of the second operand. The two operands must be of the same length.

Arithmetic overflow or underflow cannot be detected.

Formats:				
	DEC	d,d	INC	d,d
	DEC	d,n	INC	d,n
	DEC	f,f	INC	f,f
	DEC	h,h	INC	h,h
	DEC	t,n	INC	t,n
	DEC	t,t	INC	t,t

Note - Symbols of type F and H cannot be directly decremented or incremented by a constant or literal. The FTLY or HTLY instruction should be used to define a local constant to use as the second operand.

DEC or INC of symbol type R

The byte address of the AR is decremented (DEC) or incremented (INC) by the second operand. If the resultant address crosses a frame boundary, and the register is in the linked mode, it may become detached, unnormalized.

Formats:				
	DEC	r,t	INC	r,t
	DEC	r,n	INC	r,n

5.39 DEFx Assembler Directives

Function: These are assembler directives that define a local symbol. The type of the symbol is specified by the final character of the DEFx opcode, which may be B, C, D, F, H, M, N, S or T. The DEFM and DEFN directives are described separately in the next sections.

The base register and offset of the symbol's address may be either specified as literal values, or implied in the base register and offset values of a previously defined symbol. A symbol may also be redefined as equivalent to another symbol, but of a different type.

The symbol in the label field of the DEFx directive is created with the specified type.

If two operands are present, the first indicates the base register and the second indicates the offset of the symbol's address. The unit of offset depends on the symbol type: the offset for a bit (type-code B) is in number of bits; the offset for a character or a half tally (type-code C or H) is in number of bytes; the offset for a tally, double tally, F-type tally, or storage register (type-code T, D, F, or S) is in number of words (sixteen bits each). If the second operand is a literal, the offset is the value of the literal. If the second operand is a previously-defined symbol, the offset is the same as the previously-defined symbol's offset.

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.

Formats:

The following formats 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
```

The following formats are used to define symbols in terms of previously-defined symbols:

```
symbol    DEFC  r,h              symbol    DEFH  r,c
symbol    DEFD  r,t              symbol    DEFF  r,t  (Overlay)
symbol    DEFS  r,t                                   (Overlay)
symbol    DEFF  r,d              symbol    DEFS  r,d  (Overlay)
symbol    DEFS  r,f              symbol    DEFF  r,s

symbol    DEFC  h                symbol    DEFH  c
symbol    DEFD  t                symbol    DEFF  t    (Overlay)
symbol    DEFS  t                                     (Overlay)
symbol    DEFF  d                symbol    DEFS  d    (Overlay)
symbol    DEFS  f                symbol    DEFF  s
```

The following special formats are used to define one symbol as a subfield of another:

```
symbol    DEFTL d      (Overlays lower TLY of a DTLY)
symbol    DEFTU d      (Overlays upper TLY of a DTLY)
symbol    DEFTU s      (Overlays upper TLY of an SR)
symbol    DEFDL s      (Overlays lower DTLY of an SR)
symbol    DEFHL t      (Overlays lower HTLY of a TLY)
```

Examples:

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 ddressed 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 displaced from the byte address of the AR
NXTFID	DEFD	R15,7	Defines a double tally with Register 15 as the base register, and an offset of 7, which references bytes 14 through 17 displaced from the byte address of the AR; note this is not the same as a displacement of 7 double tallies, as used for immediate symbols; see the section on Immediate Symbols in the chapter 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
FPOS	DEFS	FP0	Redefines the six-byte accumulator FP0 as a storage register FPOS
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

5.40 DEFM Assembler Directive

Function: This is an assembler directive that defines a local symbol as a modal entry point, or mode-id.

A mode-id consists of a four-bit entry point number and a twelve-bit frame number or FID. The first operand in the instruction is used to specify the entry point number, and must be in the range 0-15 (0-X'F'). The second operand may be a literal or a previously defined mode-id, and is used to specify the frame number. More information on mode-id's can be found in the chapter on the Assembler.

A symbol defined by the DEFM directive may be used in the BSL and ENT instructions to transfer control to the specified location. It may also be used in the MOV and LOAD instructions, when it acts as a literal value (the assembler actually generates a literal at the end of the object code with the value defined in the DEFM instruction).

Formats:

```
symbol    DEFM n,n
symbol    DEFM n,m
```

Examples:

EXT.SYM	DEFM	3,133	Defines the symbol EXT.SYM to be entry point 3 in frame (decimal) 133, which is location 7 in the frame
MYFRAME	DEFM	0,510	Defines MYFRAME as location 0 in frame 510
ENTRY0	DEFM	0,MYFRAME	Defines ENTRY0 as location 0 (entry point 0)
ENTRY1	DEFM	1,MYFRAME	Defines ENTRY1 as location 3 (entry point 1)
ENTRY15	DEFM	15,MYFRAME	Defines ENTRY15 as location 31 (entry point 15)

5.41 DEFN Assembler Directive

Function: This is an assembler directive that defines a local symbol as a constant.

A constant may be used in exactly the same fashion as a literal value. Note that with many instructions, reference to a constant or literal will generate a literal field 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, require explicit four-byte literal generation. See the section on Literals and the comments about literals under LOAD, ADD, STORE, etc.

Format:

```
symbol    DEFN  n
```

Examples:

```
MAXNUM    DEFN  20
XCONST    DEFN  X'8010'
DELIM     DEFN  C'.'
CCONST    DEFN  C'ABCD'

          .
          .
          .

          BH    TO,MAXNUM,ERR This generates a two-byte literal
          MOV   XCONST,CTR30 This also generates two bytes
          MOV   =DCCONST,D1  Must generate four bytes here
          MCC   DELIM,R15    Immediate value
```

5.42 DETO DETZ - Detach Address Register

DETO
DETZ

Function:

DETO: Address register = detached byte address, displacement=1
DETZ: Address register = detached byte address, displacement=0

The purpose of these instructions is to provide a formal method of "detaching" an address register and setting it to a specific location. Normally, an AR is loaded by the "MOV sr,ar" instruction, which detaches it and loads the byte address from the SR. Any other instruction that affects the fields in an AR must be preceded by the DETO or DETZ instruction. For a complete understanding of these instructions, see the section on Attachment and Detachment of an Address Register in the chapter on Data Addressing.

The register is detached and its displacement field is set to one (DETO) or zero (DETZ). The frame number and linked/unlinked flag are unaffected. Since the register is now known to be detached, these fields can be safely changed.

See also the SR and ADDR directives for other means of loading address registers.

Formats: DETO r DETZ r

Examples:

```
DETZ  R14          Detach R14; set its displacement to zero
MOV   X'80000064',R14FID Set it to Frame 100, unlinked mode

DETO  R15          Detach R15; set its displacement to one
CMNT  *            Now addresses byte 12 (X'C') of frame..
CMNT  *            if it was in linked addressing mode
```

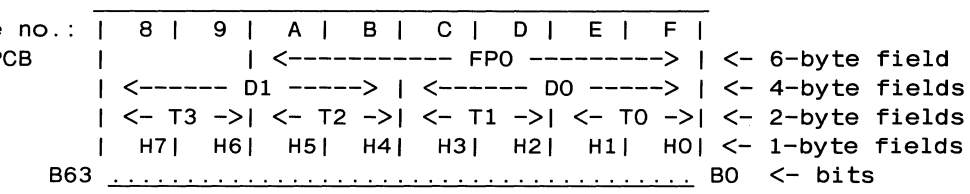
5.43 DIV DIVX - Divide into Accumulator

DIV
DIVX

Function: Accumulator = accumulator / relative.operand

Arithmetic overflow or underflow cannot be detected. If the dividend is zero, the result of the division is undefined.

The following shows the format of the accumulator and the symbolic names that address various sections of it:



DIV

The operand is divided into the four-byte field D0. One- and two-byte operands are internally sign-extended to form a four-byte field before the operation takes place.

The integer result is stored in D0, and the integer remainder in D1. The original operand is unaffected.

DIVX

The operand is divided into the six-byte field FP0. One-, two-, and four-byte operands are internally sign-extended to form a six-byte field before the operation takes place.

The six-byte integer result is stored in FP0, and the six-byte integer remainder is stored in FPY, an F-type tally in the PCB. Neither the original operand nor other sections of the accumulator are affected.

Formats:
```
DIV   d
DIV   h
DIV   n      *
DIV   t

DIVX  d
DIVX  f
DIVX  h
DIVX  n      *
DIVX  t
```

* Note: These instructions using a literal normally generate a two-byte field. If the literal is outside the range -32,768 through +32,767, an operand of the form =Dxxxx should be used to generate a four-byte literal (for example, =D40000 or =DX'FC000022'). Six-byte literals must be separately defined using the FTLY instruction.

5.44 DTLY FTLY HTLY TLY Assembler Directives

DTLY
FTLY
HTLY
TLY

Function: All symbols or variable names used as operands must have a symbol type-code; these are assembler directives that reserve storage and set up the symbol in the label field to be of a specific type. They may also be used to only reserve storage if there is no entry in the label field.

The HTLY directive is used to define a half tally (one byte), and to store a one-byte value. This directive can only be used when the assembler'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 DTLY directive is used to define a double tally (four bytes), and to store a four-byte value.

The FTLY directive is used to define an F-type tally (six bytes), and to store a six-byte value (See the SR directive to define a storage register).

The TLY directive is used to define a tally (two bytes), and to store a two-byte value.

DTLY, FTLY and TLY directives force the location counter to be aligned on a two-byte boundary (word alignment). These directives may appear anywhere in the object code.

Formats:

```
symbol    HTLY  n
symbol    FTLY  n,n     *
symbol    DTLY  n
symbol    TLY   n
```

The label-symbol is optional.

* - Note: The value stored by the FTLY directive must (due to an assembler quirk) be specified as an upper two-byte value and a lower four-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 !

5.45 EJECT Assembler Directive

EJECT

Function: This is an assembler directive that causes the MLIST (listing) processor to eject a page at this point in the listing if the "J" has been specified.

Format: EJECT

5.46 END Assembler Directive

END

Function: Indicate end of source program

This assembler directive may be used to indicate the end of a source program. It has no effect on assembly, and is treated as a comment (see CMNT).

Format: END

5.47 ENT - External Branch Unconditionally

ENT

Function: Transfer control unconditionally to external label.

The operand of this instruction must be a label that is defined as a mode-id, or external entry point. The label may be predefined in the PSYM table as a symbol with a type code of M, or it may be locally defined in the program using the DEFM assembler directive.

To transfer control to a local label, see the B instruction.

Format: ENT m

Example:

```
EXTM    DEFM  10,500      Define a constant of type M, which
        CMNT  *           is entry point #10 in frame 500.
        ...
        ...
        ENT   EXTM        Transfers control to frame 500, entry
        CMNT  *           point #10, which is location 21, or X'15'
```

5.48 ENT* - Indirect External Transfer

Function: Transfer control to external address specified in operand.

This instruction operates identically to the ENT instruction, except that the external address is variable and is obtained from the operand.

ENT* is a macro that loads the accumulator from the operand, and then executes the ENTI instruction. See ENTI for a complete explanation.

Format: ENT* t

Warning: A side effect of this instruction is that it destroys sections of the accumulator.

5.49 ENTI - Indirect External Transfer

Function: Transfer control to external address specified in T0.

This instruction operates identically to the ENT instruction, except that the external address is variable and is obtained from the low-order two bytes of the accumulator, T0, instead of from an operand.

T0 must contain a mode-id, which may be loaded into it from a local label, an external label or by converting an ASCII string. Typically, the subroutine address is obtained from a table or from a file.

Format: ENTI

Example:

```
        R15 points to a hexadecimal ASCII string :
        v
       |x |7 |1 |F |E |AM| ...

BSL   CVXR15       This converts the ASCII string value to a
CMNT  *            binary value in the accumulator FP0
CMNT  *            (therefore T0)
ENTI  *            External branch to specified location;
CMNT  *            Frame 510, location 15 (X'F').
```

5.50 EQU Assembler Directive

Function: This is an assembler directive that is used to set up an equivalence between the symbol in the label field of the statement and the operand.

Formats:

```
label     EQU   n
label     EQU   symbol
```

EQU will create an entry in the symbol table in the following manner:

1. If the operand is a literal or constant, the label-symbol will be stored as type L. See the DEFN assembler directive for information on defining constant values.

2. If the operand is another symbol, the label-symbol will be created as an exact duplicate of the operand-symbol.

Example:

```
LABEL     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.

```
TEXTS     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.

```
INPUT.PTR EQU   R5
```

creates a symbol INPUT.PTR which is equivalent to Address Register 5.

5.51 FAR - Force Attachment of Address Register

Function: Address register = normalized byte address

This instruction attaches an address register, normalizing its byte address. All instructions which reference data via an address register (that is, instructions with indirect or relative operands) attach the register; FAR merely stops at this point. The FAR instruction is typically used before comparing two byte addresses, without regard to the data actually addressed.

Byte 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 byte 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) byte address even though it logically addresses the same location. The FAR instruction may be used to normalize a byte address before MOVing it to a storage register. For more information, see the sections on the Byte Address and Attachment and Detachment of an Address Register in the chapter on Data Addressing.

Another use of the FAR instruction is to set Address Register 15 to the link field of the frame containing the byte address, that is, to byte zero, unlinked. R15 is set up in this manner if bit 5 of the second operand (the "mask" byte) of the FAR instruction is set. The other bits of the mask byte are reserved for future use.

For the register operand, the notes under the "INC Register" instruction apply if the register reaches the boundary of a frame.

Format: FAR r,n

Examples:

```
MOV   ISBEG,IS     Set IS to data start
INC   IS,CTR30     Inc by length
FAR   IS,0         Ensure normalized SR...
MOV   IS,ISEND     ...for future tests


MOV   SR20,R14     Get data pointer
FAR   R14,X'04'    Attach R14, set R15 to links
LOAD  R15;H1       Load nncf
```

5.52 FRAME Assembler Directive

FRAME

Function: This is an assembler directive whose operand specifies the frame into which the object code from the program is to be loaded. It is normally the first statement in the program (see the chapter on the assembler), but in any case must precede any statements that generate object code.

This directive also sets the assembler's location counter to 1, because executable object code begins at location 1, not 0. If it is necesary to use byte zero of the object code, the FRAME statement must be followed by an appropriate ORG assembler directive.

Format: FRAME n

5.53 FTLY - see DTLY

5.54 HALT - Halt Program

HALT

Function: Halt execution and enter Debugger

This instruction halts execution of the current program and transfers control to the assembly Debugger at entry point 11 (HALT). Execution can be resumed only by specifying an address with the Debugger "G" command. Alternatively, execution may be terminated with the "BYE", "END", or "OFF" commands. See the chapter on the Debugger for more information.

The HALT instruction affects only the current process; it does not halt the entire computer.

Format: HALT

5.55 HTLY - see DTLY

5.56 INC - see DEC

5.57 INCLUDE Assembler Directive

Function: This statement is used to "include" another program for the duration of the assembly, in the program being assembled.

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 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. If the INCLUDEd program does generate code, it may be necessary to save and restore the assembler's location counter around the INCLUDE statement, as shown in the example below.

Format: INCLUDE program.name

Example:

```
SAVELOC   EQU   *
          INCLUDE TABLE1
          INCLUDE TABLE2
          ORG   SAVELOC     Reset location counter
```

5.58 LAD - Load Absolute Difference

Function: Accumulator T0 = absolute value of difference in byte addresses of register.op.1 and register.op.2

This instruction computes the difference in the byte 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 following actions are taken:

1. If the byte addresses are in the same frame when normalized, they can be compared directly.

2. If the frame numbers of the byte addresses of the registers are unequal, the following assumptions are made:

   a. That the addresses are in a set of contiguously linked frames, and

   b. That the frame numbers differ by no more than 127.

Limitation on the use of LAD:

The result is therefore undefined under ANY of the following conditions:

a. The byte addresses are in different UNLINKED frames.

b. The byte addresses are in a LINKED set, but the frames are not contiguously linked.

c. The byte addresses are in a contiguous LINKED set, but they are separated by more than 64Kbytes (127 frames).

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, the SIDC or MIIDC type of instructions should be used.

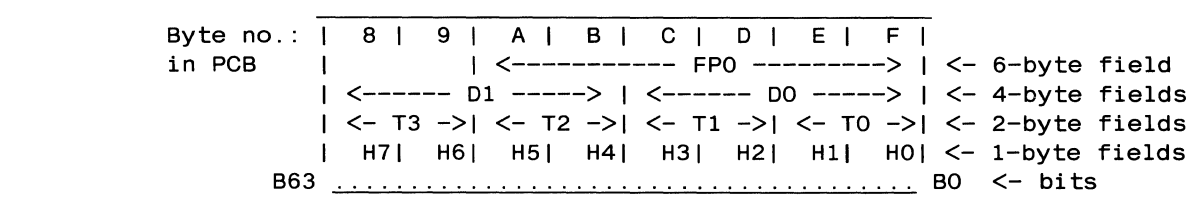
Formats:
```
LAD   r,s
LAD   s,r
```

5.59 LOAD LOADX - Load Accumulator

Function: Accumulator = relative.operand

The LOAD and LOADX instructions load the accumulator with the operand.

The following shows the format of the accumulator and the symbolic names that address various sections of it:



LOAD

The operand is loaded into the four-byte field D0. One- and two-byte operands are internally sign-extended to form a four-byte field before the operation takes place. Neither the original operand nor other sections of the accumulator are affected.

LOADX

The operand is loaded into the six-byte field FP0. One-, two-, and four-byte operands are internally sign-extended to form a six-byte field before the operation takes place. Neither the original operand nor other sections of the accumulator are affected.

Formats:

```
LOAD  d
LOAD  f
LOAD  h
LOAD  m
LOAD  n      *
LOAD  t

LOADX d
LOADX f
LOADX h
LOADX m
LOADX n      *
LOADX t
```

* Note: These instructions using a literal normally generate a two-byte field. If the literal is outside the range -32,768 through +32,767, an operand of the form =Dxxxx should be used to generate a four-byte literal (for example, =D40000 or =DX'FC000022'). Six-byte literals must be separately defined using the FTLY instruction.

5.60 MBD - Convert Binary to Decimal ASCII Byte String

Function:

Pre-incremented string = decimal ASCII equivalent of operand.1

This instruction converts binary numbers to decimal ASCII strings. The register operand is incremented by one, and the next converted byte stored at that location. This operation is repeated until the entire string has been converted, as determined by the following:

1. MBD without a numeric first operand does not create leading zeroes; the field is variable length. MBD, unlike MBX, generates one zero for an operand value of zero.

2. MBD with a numeric first operand stores a fixed length, leading zero filled field. The field is allowed to exceed the specified length if its precision requires this.

Warning - the MBD instruction is actually a macro that generates a subroutine call, and is included here for convenience. For case 1 above, either MBDSUB (for one-, two-, or four-byte numbers) or MBDSUBX (for six-byte numbers) will be called. For case 2 above, either MBDNSUB (for one-, two-, or four-byte operands) or MBDNSUBX (for six-byte operands) will be called. The following elements will be destroyed:

```
BKBIT
T4
D0
D1
R14
R15
FPX     (MBDSUBX and MBDNSUBX only; same as SYSR0)
FPY     (MBDSUBX and MBDNSUBX only; same as SYSR1)
SYSR0   (MBDSUBX and MBDNSUBX only; same as FPX)
SYSR1   (MBDSUBX and MBDNSUBX only; same as FPY)
```

Neither R14 nor R15 should be used as the register operand in the MBD instruction, nor should any section of the accumulator be used as the binary field operand. 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.

For the address register operand, the notes under the "INC Register" instruction apply if the register reaches the boundary of a frame.

Formats:

```
MBD   d,r
MBD   f,r
MBD   h,r
MBD   t,r

MBD   n,d,r
MBD   n,f,r
MBD   n,h,r
MBD   n,t,r
```

Examples:

```
        MBD   CTR1,R9            MBD   4,CTR1,R9
```

Macro expansions:

```
        LOAD  CTR1               LOAD  CTR1
        MOV   R9,R15             MOV   R9,R15
        BSL   MBDSUB             MOV   4,T4
        MOV   R15,R9             BSL   MBDNSUB
                                 MOV   R15,R9
```

5.61 MBX MBXN - Convert Binary to Hex ASCII Byte String

MBX
MBXN

Function:

Pre-incremented string = hex ASCII equivalent of operand.1

These instructions convert binary numbers to hexadecimal ASCII strings. The register operand is incremented by one, and the next converted byte stored at that location. This operation is repeated until the entire string has been converted, as determined by the following:

1. MBX does not create leading zeroes; the field is variable length. MBX, unlike MBD, does not generate a zero for an operand value of zero.

2. MBXN creates a fixed length, leading zero filled field. If the field exceeds the specified length, it is truncated on the right. MBXN is a macro, defined below.

The MBX instruction assumes that the low-order byte of the accumulator, H0, is set up as follows:

Bit 0 - Set if the string is to be padded with leading zeroes

Bits 4-7 - Contain the number of hexadecimal digits to create, (leading zeros will be suppressed if bit 0 is 0).

Warning - the MBXN instruction is actually a macro and is included here for convenience. It uses the accumulator; see the macro expansion below. The contents of H0 are undefined after execution of either the MBX or MBXN instruction.

For the address register operand, the notes under the "INC Register" instruction apply if the register reaches the boundary of a frame.

Formats:

```
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
```

Examples:

```
        LOAD  X'0C'        Output 12 chars; suppress leading zeroes
        MBX   FP2,R14


        MBXN  4,CTR1,R9    Output 4-char leading-zero-filled string
```

Macro expansion:

```
        LOAD  X'84'
        MBX   CTR1,R9
```

5.62 MCC - Move a Character

Functions:

Relative character = relative character
Relative character = indirect character
Indirect character = indirect character
Indirect character = literal
Indirect character = relative character

The character addressed by the first operand is stored at the location addressed by the second operand.

Formats:
```
MCC   c,c
MCC   c,r
MCC   n,r
MCC   r,c
MCC   r,r
```

5.63 MCI - Move a Character

Functions:

Indirect pre-incremented character = indirect character
Indirect pre-incremented character = literal
Indirect pre-incremented character = relative character

The second operand, which is an address register, is incremented by one; the character addressed by the first operand is stored at that location.

For the address register operand, the notes under the "INC Register" instruction apply if the register reaches the boundary of a frame.

Formats:
```
MCI   c,r
MCI   n,r
MCI   r,r
```

5.64 MCI extensions

Function: Indirect pre-incremented string = literal character

This instruction propagates a single character as many times as specified in the third operand. If it is initially zero, 65,536 bytes will be propagated.

The second operand is incremented by one; the literal character from the first operand is stored at that location. This operation is repeated until the terminating condition is met.

These instructions are provided as a convenience in coding. They are both macros that set up the conditions for the appropriate machine instruction that moves a string of bytes.

Note the side effects of the instructions.

In both cases Address Register 15 and the accumulator D0 are used; a MIIT instruction is executed. See the MIIT instruction for details.

Formats:
```
MCI   n,r,t
MCI   n,r,n
```

Macro expansion:

```
MOV   r,R15
MCI   n,r
LOAD  op.3        This may be a tally or a constant
DEC   D0
MIIT  R15,r
```

5.65 MDB MXB - Convert One ASCII Byte to Binary

MDB
Function : MXB

MDB : Operand.2 = operand.2 * 10 + binary equivalent of operand.1
MXB : Operand.2 = operand.2 * 16 + binary equivalent of operand.1

These instructions convert ASCII characters to binary. They are normally used in a loop, with operand.2 (a tally) initially set to zero. Each execution of the MDB or MXB instruction "shifts" the previous value in the tally by the appropriate amount, then adds in the binary equivalent of the character addressed by the first operand. The example below should clarify this.

If the character addressed by the first operand is non-decimal (for MDB) or non-hexadecimal (for MXB), the result of the instruction is undefined.

Note - these instructions have been largely superseded by the equivalent string conversion instructions MSDB, MSXB, MFB and MFX.

Formads:

```
MDB   r,d          MXB   r,d
MDB   r,f          MXB   r,f
MDB   r,t          MXB   r,t
```

Example:

```
        ZERO  FP0          Clear the accumulator
LOOP    INC   R15          Set on next potential numeric character
        BCNN  R15,QUIT     Done if not numeric character
        MDB   R15,FP0      Convert one more character
        B     LOOP
```

5.66 MFD MFX - Convert ASCII String to Binary

MFD
MFX

Function :

MFD : Accumulator FP0 = binary equivalent of pre-incremented decimal ASCII string from operand.1

MFX : Accumulator FP0 = binary equivalent of pre-incremented hexadecimal ASCII string from operand.1

These instructions convert ASCII character strings to binary. The operand, which is an address register, is incremented by one before the instruction starts to convert the string. The string addressed by the operand may optionally contain a leading "+" or "-" sign; it may also contain a decimal point and fractional digits (see below). The result of this instruction is a scaled integer in FP0.

The following shows the format of the accumulator and the symbolic names that address various sections of it:

```
Byte no.: |  8 |  9 |  A |  B |  C |  D |  E |  F |
in PCB    |         | <---------- FP0 ---------> | <- 6-byte field
          | <------ D1 -----> | <------ D0 -----> | <- 4-byte fields
          | <- T3 ->| <- T2 ->| <- T1 ->| <- T0 ->| <- 2-byte fields
          |  H7|  H6|  H5|  H4|  H3|  H2|  H1|  H0| <- 1-byte fields
       B63 ..................................... B0  <- bits
```

Before executing the MFD or MFX instructions, the accumulator must be initialized as follows:

H7 Contains the number of fractional digits expected in the value. This must be in the range 0-15 (0-X'F'). The converted value stored in FP0 will be 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 when converting fixed length strings. A zero is equivalent to 256.

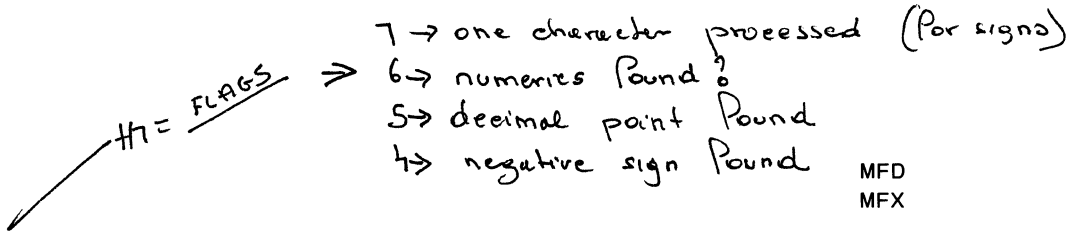
FP0 Initial value is typically zero, though any value in FP0 is "shifted" by multiplying it by 10 (MFD) or by 16 (MFX) as each byte is converted.

The instruction terminates under one of the following conditions:

1. When a non-numeric (for MFD, a character not in the range 0-9) or non-hexadecimal (for MFX, a character not in the range 0-9 or A-F) is found. If the terminating character is a decimal point or is in the range X'FC'-X'FF' (that is, if it is a system delimiter), the flag NUMBIT is set; otherwise, NUMBIT is zeroed. The register addresses the terminating character.

2. When the number of characters specified by H6 have been converted. NUMBIT is zeroed, and the register addresses the last character converted.

3. When the number of fractional digits specified by H7 have been converted, and a system delimiter or decimal point is not found. NUMBIT is zeroed, and the register addresses the terminating (unconverted) character.



MFD
MFX

After execution, H6 will be decremented by one for each digit found to the left of the decimal point. When converting fixed length strings, then, H6 may be compared to zero to determine if an entire string was successfully converted.

Formats: MFD r MFX r

Examples:

```
Instruction:  ZERO  T3
              ZERO  FP0
              MFD   R4

Before instruction: R4 --v                          FP0=0     H7=0, H6=0
Data:                  |A |1 |8 |AM| ..
After instruction : R4 -----------^
                    NUMBIT=1                        FP0=18 (X'000000000012')

Instruction:  MOV   X'0200',T3
              ZERO  FP0
              MFD   R4

Before instruction: R4 --v                          FP0=0     H7=2, H6=0
Data:                  |AM|- |1 |8 |. |7 |5 |SM| ..
After instruction : R4 ------------------------^
                    NUMBIT=1                        FP0=-1875 (X'FFFFFFFFF8AD')
                                                    Note integer is scaled

Instruction:  MOV   X'0200',T3
              ZERO  FP0
              MFD   R4

Before instruction: R4 --v                          FP0=0     H7=2, H6=0
Data:                  |AM|+ |1 |7 |7 |5 |Q |SM| ..
After instruction : R4 --------------------^
                    NUMBIT=0                        FP0=177500 (X'00000002B55C')
                    Non-numeric                     Note integer is scaled even
                    character (Q)                   though there were no
                    found.                          fractional digits present.

Instruction:  MOV   X'0004',T3
              ZERO  FP0
              MFX   R4

Before instruction: R4 --v                          FP0=0     H7=0, H6=4
Data:                  |7 |0 |1 |F |7 |A |2 |3 | ..
After instruction : R4 --------------^
                    NUMBIT=0                        FP0=507 (X'0000000001F7')
                    Maximum string length reached
```

5.67 MIC - Move a Character

MIC

Functions:
Indirect character = indirect pre-incremented character
Relative character = indirect pre-incremented character

The first operand, which is an address register, is incremented by one; the character addressed by the first operand is stored at the location addressed by the second.

MIC is a macro provided for coding convenience. It is equivalent to an "INC r" instruction followed by an "MCC r,c" or "MCC r,r".

For the register operand, the notes under the "INC Register" instruction apply if the register reaches the boundary of a frame.

Formats: MIC r,c
MIC r,r

5.68 MII - Move a Character

MII

Function: Indirect pre-incremented character = indirect pre-incremented character

Both operands, which are address registers, are incremented by one; the character addressed by the first operand is stored at the location addressed by the second.

For both registers, the notes under the "INC Register" instruction apply if the register reaches the boundary of a frame.

Format: MII r,r

5.69 MII Extensions

Function: Indirect pre-incremented string = indirect pre-incremented string

Both operands, which are address registers, are incremented by one; the character addressed by the first operand is stored at the location addressed by the second. This operation is repeated until the terminating condition is met.

These instructions are provided as a convenience in coding. They are both macros that set up the conditions for the appropriate machine instruction that moves a string of bytes.

Note the side effects of the instructions.

In the case of (1), below, Address Register 15 is used; the third operand is moved into it, and a MIIR instruction is executed. See the MIIR instruction for details.

In the case of (2), below, the accumulator D0 is used; the third operand is loaded into it, and a MIIT instruction is executed. See the MIIT instruction for details.

For both registers, the notes under the "INC Register" instruction apply if the register reaches the boundary of a frame.

Formats:

```
MII   r,r,s          (1)
MII   r,r,t          (2)
MII   r,r,n          (2)
```

Macro expansions:

```
   (1)                    (2)
MOV   op.3,R15        LOAD  op.3
MIIR  op.1,op.2       MIIT  op.1,op.2
```

5.70 MIID MIIDC - Move a String

MIID
MIIDC

Function: Indirect pre-incremented string = indirect pre-incremented string

(string terminates when delimiter found)

In addition, for MIIDC:
Accumulator T0 = accumulator T0 - length of string moved

The first two operands, which are address registers, are incremented by one; the character addressed by the first operand is stored at the location addressed by the second. This operation is repeated until a "delimiter," or byte specified by the third operand (the "mask" byte), is encountered. The terminating condition is found by testing each byte after it has been copied.

Note that the byte addresses of the registers will always be incremented by at least one, because the delimiter test is done after the byte copy.

For the MIIDC instruction, as each byte is moved, the low-order two-byte field of the accumulator, T0, is decremented by one. Other sections of the accumulator are unaffected. Normally, T0 is set to either ZERO or ONE before this instruction is executed. If set to zero, the resultant value after the instruction executes is the negative of the length of the string, including the delimiter. If set to one, it is the negative of the string length excluding the delimiter.

The "mask" operand for this instruction is a byte that is used to specify under what conditions the string, and therefore the instruction, terminates. Up to seven different characters can be tested; four of them are fixed, and 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 variable, and may be prestored by the programmer in the scan characters SC0, SC1, and SC2.

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

```
Bits:  ---0-----1-----2------3------4-----5-----6-----7---
Test:          SM    AM     VM     SVM   (SC0) (SC1) (SC2)
```

The parentheses around SC0, SC1 and SC2 are to indicate that it is the contents of these locations that are being compared.

The high-order bit (bit 0) of the byte is used in the following manner: if 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 zero, it indicates that the string terminates on the first non-occurrence of a delimiter as specified by the setting of bits 1-7.

A few examples should make this clear:

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'F8'	1111 1000	Stop on first occurrence of any system delimiter - SM, AM, VM or SVM.
X'C3'	1100 0011	Stop on first occurrence of an SM, or the contents of SC1 or of SC2.
X'01'	0000 0001	Stop on the first NON-occurrence of the contents of SC2 *.

* - For example, if SC2 contains a BLANK, this mask will cause the instruction to terminate when the first NON-BLANK is found.

For both registers, the notes under the "INC Register" instruction apply if the register reaches the boundary of a frame.

Formats: MIID r,r,n
MIIDC r,r,n

Examples:

```
         MIIDC R4,R5,X'C0'      COPY UNTIL SM

                          Before instruction
   R4 --v                        R5 --v                          TO = 1
Data:  |A |B |C |SM| ...             |1 |2 |3 |4 |5 |6 | ...
                 ^                  |1 |B |C |SM|5 |6 | ...
   R4 -----------|               R5 -----------^                TO = -2
                          After instruction


         MCC   C' ',SC1
         MIID  R4,R5,X'82'      COPY UNTIL BLANK

                          Before instruction
   R4 --v                        R5 --v
Data:  |A |B |C |SM|  | ...          |1 |2 |3 |4 |5 |6 | ...
                   ^                |1 |B |C |SM|  |6 | ...
   R4 --------------|            R5 --------------^
                          After instruction
```

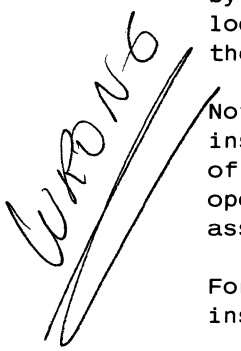
5.71 MIIR - Move a String

Function: Indirect pre-incremented string = indirect pre-incremented string

(string terminates on register match)

If the second operand's address equals that of Address Register 15 at the start of this instruction, no action takes place.

Otherwise, both operands, which are address registers, are incremented by one; the character addressed by the first operand is stored at the location addressed by the second. This operation is repeated until the second operand's address equals that of Address Register 15.



Note that Address Register 15 is not one of the operands in the instruction, though it is implicitly referenced as the ending location of the string. R15, therefore, should not be used as one of the operands. The assembler will not check for this condition, and the assembled instruction will not execute correctly if it arises.

For all three registers, the notes under the "INC Register" instruction apply if the register reaches the boundary of a frame.

Format: MIIR r,r

5.72 MIIT MIITD - Move a String

MIIT
MIITD

Function: Indirect pre-incremented string = indirect pre-incremented string

MIIT: (string terminates on count runout)
MIITD: (string terminates on count runout OR delimiter found)

If the low-order two-byte field of the accumulator, T0, is zero at the start of these instructions, no action takes place.

Otherwise, the first two operands, which are address registers, are incremented by one; the character addressed by the first operand is stored at the location addressed by the second. T0 is then decremented by one. This operation is repeated until the following condition(s) occur(s):

1. For the MIIT instruction, when T0 reaches zero. This instruction is typically used to move a fixed length string.

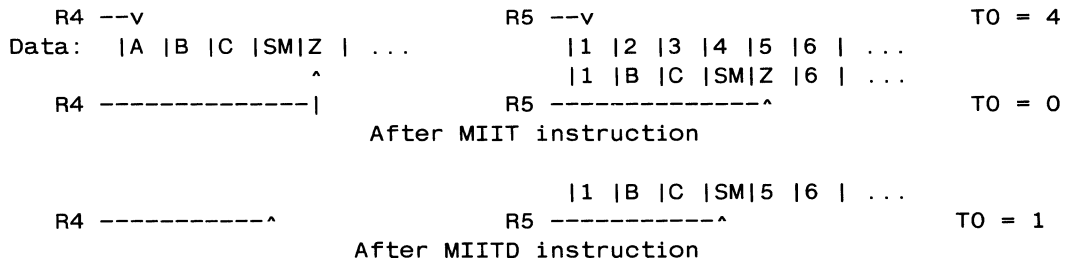
2. For the MIITD instruction, when T0 reaches zero, or when one of the delimiter bytes specified by the third operand (the "mask" byte), is encountered. The terminating condition is found by testing each byte after it has been copied. This instruction is typically used to move a delimited string of unknown length to a location of preset maximum length. If the string is longer than the destination location, the instruction terminates without overlaying subsequent data.

   See the notes under the MIID or SID instruction for a complete description of the "mask" byte.

For both registers, the notes under the "INC Register" instruction apply if the register reaches the boundary of a frame.

Formats:
```
MIIT  r,r
MIITD r,r,n
```

Examples:
```
        LOAD  4                     LOAD  4
        MIIT  R4,R5                 MIITD R4,R5,X'C0'   (Stop on SM)
```



5.73 MOV - Move One Operand to the Other

The MOV instruction is used to move one operand to another.

MOV of symbol types B, D, F, H, S, T

Function: Operand.2 = Operand.1

The contents of the first operand replace the contents of the second operand. The two operands must be of the same type.

Formats:
```
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
```

Note - Symbols of type F and H cannot be directly loaded with a constant or literal. The FTLY or HTLY instructions should be used to define a local constant to move from.

MOV of symbol type R to/from S

Function: Register.2 = "detached" byte address from register.1

These are special instructions which load and store the byte address of the address register. For a complete understanding of these instructions, see the section on Attachment and Detachment of an Address Register in the chapter on Data Addressing.

When an AR is moved to an SR, the byte address of the AR replaces the previous value in the SR. If the AR was attached, the address is converted to the detached form before the move. The AR itself remains unchanged.

When an SR is moved to an AR, the AR is first detached, and then the byte address from the SR replaces the previous value in the AR.

Formats:
```
MOV   r,s
MOV   s,r
```

5.74 MSDB MSXB - Convert ASCII String to Binary

MSDB
MSDB

Function :

MSDB : Accumulator FP0 = binary equivalent of pre-incremented decimal ASCII string from operand.1

MSXB : Accumulator FP0 = binary equivalent of pre-incremented hexadecimal ASCII string from operand.1

These instructions convert ASCII character strings to binary. They are macros provided for coding convenience. They first clear the entire accumulator (T3 and FP0), and then execute either the MFD (if MSDB) or MFX (if MSXB) instruction. See the section on MFD and MFX for more information about these instructions.

Formats:

```
MSDB  r
MSXB  r
```

5.75 MTLY Assembler Directive

MTLY

Function: All symbols or variable names used as operands must have a symbol type-code; this instruction reserves storage and sets up the symbol in the label field to be of type M, which is a mode-id.

A mode-id consists of a four-bit entry point number and a twelve-bit frame number or FID. The first operand in the instruction is used to specify the entry point number, and must be in the range 0-15 (0-X'F'). The second operand may be a literal or a previously defined mode-id, and is used to specify the frame number. More information on mode-id's can be found in the chapter on the Assembler.

MTLY is typically used when creating a table of mode-id's. These may be loaded into the accumulator for use in the BSLI and ENTI instructions to transfer control indirectly. See also the DEFM Assembler Directive, which defines a mode-id without creating storage.

Formats:

```
symbol     MTLY  n,m
symbol     MTLY  n,n
```

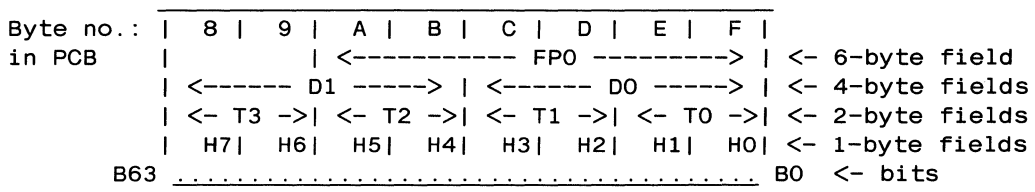
5.76 MUL MULX - Multiply into Accumulator

MUL
MULX

Function: Accumulator = accumulator * relative.operand

Arithmetic overflow or underflow cannot be detected.

The following shows the format of the accumulator and the symbolic names that address various sections of it:



MUL

The operand is multiplied by the four-byte field D0. One- and two-byte operands are internally sign-extended to form a four-byte field before the operation takes place.

The eight-byte result is stored in D1 and D0. The original operand is unaffected.

MULX

The operand is multiplied by the six-byte field FP0. One-, two-, and four-byte operands are internally sign-extended to form a six-byte field before the operation takes place.

The low order eight bytes of the result are stored in D1 and D0. The original operand is unaffected.

Formats:

```
MUL   d
MUL   h
MUL   n      *
MUL   t

MULX  d
MULX  f
MULX  h
MULX  n      *
MULX  t
```

* Note: These instructions using a literal normally generate a two-byte field. If the literal is outside the range -32,768 through +32,767, an operand of the form =Dxxxx should be used to generate a four-byte literal (for example, =D40000 or =DX'FC000022'). Six-byte literals must be separately defined using the FTLY instruction.

5.77 MXB - see MDB

5.78 NEG - Negate Operand

NEG

Function: Operand = -Operand

The contents of the operand are replaced by the negative (two's complement) of the operand.

Formats:
```
NEG   d
NEG   f
NEG   h
NEG   t
```

5.79 NOP - No Operation

NOP

Function: None

This instruction performs no operation; the next instruction in sequence is executed.

Format: NOP

5.80 ONE - Set Operand to One

ONE

Function: Operand = 1

The contents of the operand are replaced by a binary one.

Formats:
```
ONE   d
ONE   f
ONE   h
ONE   t
```

5.81 OR - Logical OR of a Byte

OR

Function: Indirect byte = indirect byte logically OR'ed with operand

The byte referenced by the first operand is logically OR'ed with the byte referenced by the second operand. The byte referenced by the second operand is unchanged.

Formats:
```
OR    r,n
OR    r,r
```

5.82 ORG Assembler Directive

Function: This is an assembler directive that resets the assembler's location counter.

The location counter advances as the object code is generated, and the "Current location function" (*) contains the address of the next byte to be generated. There are several reasons to change the location counter in an explicit manner:

1. A typical example of ORG is to use byte zero of the object code. The FRAME assembler directive sets the location counter to one (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  *         (Now location counter is back to 1)
AM        EQU   R1        This may be used to refernce the byte
          CMNT  *         at location zero symbolically via label AM
```

2. To save and restore the location counter; for example, if a program is INCLUDEd that actually generates code:

```
SAVELOC   EQU   *         Save location counter before INCLUDE
          INCLUDE TABLE1  Include program to get definitions
          ORG   SAVELOC   Reset in case TABLE1 has object code
```

3. To leave "space" in the object code for variables that the program uses. This is not a good feature in general, 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.

5.83 READ READX - Read Byte

Function: Indirect character = next byte in asynchronous byte buffer

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 PIB for the process, until some configuration-dependent maximum number of characters is received. If this condition occurs, no further data are accepted from the channel, which will output a Bell character (X'07') for each attempted input character until the condition is cleared.

The READX instruction never echoes characters on the asynchronous channel.

For the READ instruction, control characters (X'00' through X'1F') are never echoed, while non-control characters are echoed unless bit NOECHO is set. The READ instruction is actually a macro which tests whether a character should be echoed, and executes a WRITE instruction if so.

Formats:
```
READ  r
READX r
```

Example:

```
          READ  R2
```

Macro expansion:

```
          READX R2
          BCL   R2,X'20',=L002
          BBS   NOECHO,=L002
          WRITE R2
=L002     EQU *
```

5.84 RQM - Release Timeslice Quantum

RQM

Function: Release process's timeslice

This 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 RQM instruction is inserted as a means of delaying the process for a while. It is a request to the Monitor to turn over control to the next process in line. The process that executed the RQM will be reactivated after other active processes in the process chain have executed their timeslices.

See the example in the section on the XCC instruction; also see the SLEEP instruction.

Format: RQM

5.85 RTN - Return from a Subroutine

RTN

Function:

a. An address is obtained from the current entry in the return stack, and the stack pointer is decremented by four.
b. Control is transferred to the location so obtained.

This instruction is the correct way to return after a subroutine 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 Debugger is entered with a Return Stack Empty trap condition.

Also see the BSL instruction to call a subroutine.

Return stack entries are four bytes each; their format is described in the chapter SYSTEM CONVENTIONS. An entry may be deleted from the return stack by the instruction "DEC RSCWA,4". This is mandatory if a subroutine is to be exited without using a RTN instruction. The entire return stack may be reset by the instruction "MOV X'184',RSCWA", 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.

Format: RTN

5.86 SB - Set Bit

SB

Function: Bit = 1

The referenced bit is set to an "on" (1 or true) condition.

Format: SB b

5.87 SET.TIME - see TIME

5.88 SHIFT - Logical Right Shift of a Byte

Function: Operand.2 = indirect byte right shifted one bit

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.

Format: SHIFT r,r

5.89 SICD - Scan over Multiple Delimiters

Function: Scan a string until a specific number of delimiters is found

The first operand, which is an address register, is incremented until the terminating condition specified by the accumulator, 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.

This instruction can scan a variable number of delimiters. Its function is to position the register at a specific point within a data structure containing several levels of delimiters.

The low-order tally of the accumulator, T0, contains the count of delimiters.

The "mask" byte is used to specify under what conditions the scan terminates. Note - this "mask" byte is different from the one used in the SID, SIDC, SIT, SITD, MIID, MIIDC, and MIITD instructions.

Three of the possible scan delimiters are fixed, and are the standard system delimiters:

Value mark - VM - X'FD' Attribute mark - AM - X'FE'
Sub-value mark - SVM- X'FC'

Three other delimiters are variable, and may be prestored by the programmer in the scan characters SC0, SC1 and SC2. Six bits in the mask byte are used to determine which of the six above characters are to be compared; if a bit is set (1), the corresponding character is tested; if zero, it is ignored. Only one of these bits may be set for the SICD innstruction.

```
Bits:  ---0-----1-----2------3------4-----5-----6-----7---
Test:          .     AM     VM     SVM   (SC0) (SC1) (SC2)
```

The parentheses around SC0, SC1 and SC2 are to indicate that it is the contents of these locations that are compared.

The high-order bit (bit 0) of the mask, if set, indicates that the accumulator T0 should be DECREMENTED by one BEFORE the scan is started and the terminating condition tested. If zero, this will not take place.

Bit 1 specifies the condition for abnormal termination of the scan. If set, the scan will terminate abnormally if a character is found which is logically higher than the character in SC2. If zero, the scan will terminate abnormally if a character is found which 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.

The scan can terminate either normally or abnormally. It will terminate normally if the number of delimiters specified in T0 (pre-decremented if required) is encountered. In this case, T0 will be decremented to zero, and the register will point to the final

delimiter (or will be unchanged if no scan takes place).

The scan will terminate 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 will be the number of delimiters which must be inserted in the data to create the required data position, and the register will point one byte BEFORE the character which 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 scan.
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.

Format: SICD r,n

Examples are on the next page.

Examples for SICD:

The following data structure is used in the examples:

```
 E0^E11]E12^E2^E31]E321\E322]E33^_
^  ^       ^  ^        ^        ^
a  b       c  d        e        f  < Register locations noted below
```

CASE 1 - Scan to attribute 3; RECALL interface; R15 positioned at "a"

```
    LOAD   3            AMC count
    SICD   R15,X'20'    Scan to AM delimiter
```

At completion, R15 will be positioned at "d," and T0 = 0

CASE 2 - Scan to attribute 6; BASIC interface; R15 positioned at "b"

```
    LOAD   6            AMC count
    SICD   R15,X'A0'    Scan to AM delimiter
```

At completion, R15 will be positioned at "f," and T0 = 2
(Note that R15 has been backed off one byte from the SM).

CASE 3 - Scan to attribute 3, value 2, subvalue 2; RECALL interface;
R15 positioned at "a"

```
     LOAD   3            AMC count
     SICD   R15,X'20'    Scan to AM delimiter
     LOAD   2            value position
     SICD   R15,X'90'    Scan to VM delimiter (DECREMENT T0 BEFORE SCAN)
     LOAD   2            subvalue position
     SICD   R15,X'88'    Scan to SVM delimiter (DECREMENT T0 BEFORE SCAN)
```

At completion, R15 will be positioned at "e," and T0 = 0

5.90 SID SIDC - Scan Over a String

SID
SIDC

Function: Scan a string until a delimiter is found

In addition, for SIDC:
Accumulator T0 = accumulator T0 - length of string scanned

The first operand, which is an address register, is incremented by one. This operation is repeated until the terminating condition specified by the second operand (the "mask" byte), is encountered. The terminating condition is found by testing each byte after it has been addressed.

Note that the byte address of the register will always be incremented by at least one, because it is incremented before the byte test is done.

For the SIDC instruction, as each byte is moved, the low-order two-byte field of the accumulator, T0, is decremented by one. Other sections of the accumulator are unaffected. Normally, T0 is set to either ZERO or ONE before this instruction is executed. If set to zero, the resultant value after the instruction executes is the negative of the length of the string, including the delimiter. If set to one, it is the negative of the string length excluding the delimiter.

The "mask" operand for this instruction is a byte that is used to specify under what conditions the string, and therefore the instruction, terminates. Up to seven different characters can be tested; four of them are fixed, and 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 variable, and may be prestored by the programmer in the scan characters SC0, SC1, and SC2.

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

```
Bits:  ---0-----1-----2------3------4-----5-----6-----7---
Test:          SM    AM     VM     SVM   (SC0) (SC1) (SC2)
```

The parentheses around SC0, SC1 and SC2 are to indicate that it is the contents of these locations that are being compared.

The high-order bit (bit 0) of the byte is used in the following manner: if 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 zero, it indicates that the string terminates on the first non-occurrence of a delimiter as specified by the setting of bits 1-7.

A few examples should make this clear:

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'F8'	1111 1000	Stop on first occurrence of any system delimiter - SM, AM, VM or SVM.
X'C3'	1100 0011	Stop on first occurrence of an SM, or the contents of SC1 or of SC2.
X'01'	0000 0001	Stop on the first NON-occurrence of the contents of SC2 *.

* - For example, if SC2 contains a BLANK, this mask will cause the instruction to terminate when the first NON-BLANK is found.

For the first operand, the notes under the "INC Register" instruction apply if the register reaches the boundary of a frame.

Formats: SID r,n
SIDC r,n

Example:

```
         SIDC  R4,X'C0'

                          Before instruction
   R4 --v                        T0 = 0
Data:  |A |B |C |SM| ...
                ^
   R4 ----------|              T0 = -3
                          After instruction


         MCC   C' ',SC1
         SID   R4,X'02'        SCAN UNTIL NON-BLANK

                          Before instruction
   R4 --v
Data:  |  |  |  |X|  | ...
                ^
   R4 ----------|
                          After instruction
```

5.91 SIT SITD - Scan Over a String

SIT
SITD

Function: Scan indirect pre-incremented string

SIT: (string terminates on count runout)
SITD: (string terminates on count runout OR delimiter found)

If the low-order two-byte field of the accumulator, T0, is zero at the start of these instructions, no action takes place.

Otherwise, the first operand, which is an address register, is incremented by one, and the accumulator T0 is decremented by one. This operation is repeated until the following condition(s) occur(s):

1. For the SIT instruction, when T0 reaches zero. This instruction is typically used to scan over a fixed length string. It is logically equivalent to the "INC r,t" instruction, except that additional frames may be linked on to the end of the linked set by using XMODE.

2. For the SITD instruction, when T0 reaches zero, or when a delimiter byte specified by the third operand (the "mask" byte), is encountered. The terminating condition is found by testing each byte as it is scanned. This instruction is typically used to scan over a delimited string of preset maximum length. Additional frames may be linked on to the end of the linked set by using XMODE.

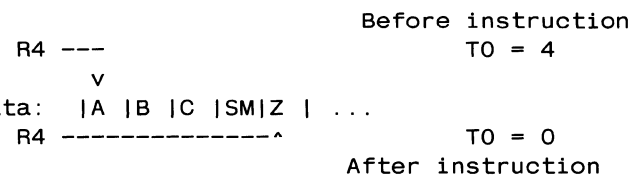
See the notes under the MIID or SID instructions for a complete description of the "mask" byte.

For the first operand, the notes under the "INC Register" instruction apply if the register reaches the boundary of a frame.

Formats:
```
SIT   r
SITD  r,n
```

Example:

```
          LOAD  4
          SIT   R4
```



5.92 SLEEP - Wait

Function: Wait for a specified time

This 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.

The accumulator D0 must be loaded with the "awakening" time of day in internal system format (number of milliseconds past midnight) before the SLEEP instruction is executed. 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.

Format: SLEEP

5.93 SR Assembler Directive

SR

Function: All symbols or variable names used as operands must have a symbol type-code; this is an assembler directive that reserves storage and sets up the symbol in the label field to be of type S (Storage Register). It also generates a byte address.

The first operand is used to specify the displacement of the generated byte address, and the second the FID or Frame number. If the high-order bit of the second operand's value (which is a four-byte field) is set, the byte address is in UNLINKED format; if zero, it is in LINKED format. See the section in the chapter on Data Addressing for a full description of linked and unlinked modes of addressing; also compare to the ADDR assembler directive.

Format:

```
symbol    SR  n,n
```

Examples:

Instruction			Generated value	
F100	SR	1,100	0001 0000 0064	Addresses frame 100 in linked mode, therefore address is location 12 (X'C') in the frame.
F100U	SR	1,X'80000064'	0001 8000 0064	Address is in unlinked mode; location is 1 in frame 100.
	MOV	F100U,R15		Sets R15 to the above address.

5.94 SRA - Set Register to Address

Function: Byte address of operand.1 = address of operand.2

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.

Formats:
```
          SRA   r,c
          SRA   r,d
          SRA   r,f
          SRA   r,h
          SRA   r,l     *
          SRA   r,s
```

* Note - SRA to a local label works only when the location of the label is less than X'100', that is, in the first half 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 in the second half of the frame, one way is to make the label of type T using instructions of the form:

```
          ALIGN *           Need to align location on word boundary!
LABEL     DEFT  R1,*16      Define LABEL as "here" (*16 is to get offset
          CMNT  *           as words, not bytes)
```

Now the SRA r,LABEL would work correctly.

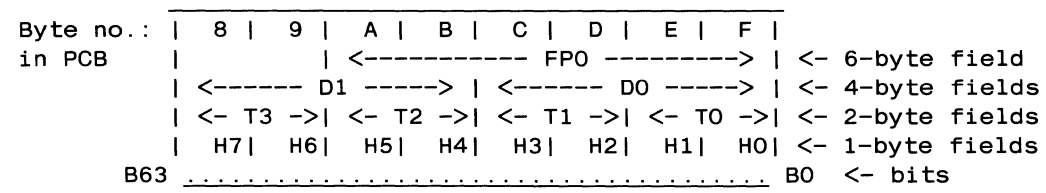
Examples:

```
FILENAME  EQU   *-1
          TEXT  C'INVN',X'FE'
          ...
          ...
          SRA   R15,FILENAME This sets R15 to address one byte BEFORE
          CMNT  *           the byte 'I' in the string 'INV..'. Typically
          CMNT  *           R15 is now used in a MIID instruction to copy
          CMNT  *           the string, until the AM, to another location.


          SRA   R15,D0      This sets R15 to point to the first byte of
          CMNT  *           the accumulator D0.


          SRA   R15,H3      Same as above (see format of Accumulator).
```

5.95 STORE - Store Accumulator in Operand

Function: Relative.operand = accumulator

The following shows the format of the accumulator and the symbolic names that address various sections of it:



The contents of the accumulator (H0, T0, D0 or FP0) replace the contents of the operand. The accumulator is not changed.

Formats: STORE d
STORE f
STORE h
STORE t

5.96 SUB SUBX - Subtract from Accumulator

SUB
SUBX

Function: Accumulator = accumulator - relative.operand

These instructions subtract the contents of the operand from the accumulator.

Arithmetic overflow or underflow cannot be detected.

The following shows the format of the accumulator and the symbolic names that address various sections of it:

```
                 ___________________________________________
Byte no.: |  8 |  9 |  A |  B |  C |  D |  E |  F |
in PCB    |         | <---------- FP0 ---------> | <- 6-byte field
          | <------ D1 -----> | <------ D0 -----> | <- 4-byte fields
          | <- T3 ->| <- T2 ->| <- T1 ->| <- T0 ->| <- 2-byte fields
          |  H7|  H6|  H5|  H4|  H3|  H2|  H1|  H0| <- 1-byte fields
       B63 ..........................................  B0  <- bits
```

SUB

The operand is subtracted from the four-byte field D0. One- and two-byte operands are internally sign-extended to form a four-byte field before the operation takes place. Neither the original operand nor other sections of the accumulator are affected.

SUBX

The operand is subtracted from the six-byte field FP0. One-, two-, and four-byte operands are internally sign-extended to form a six-byte field before the operation takes place. Neither the original operand nor other sections of the accumulator are affected.

Formats:
```
SUB  d
SUB  h
SUB  n    *
SUB  t

SUBX d
SUBX f
SUBX h
SUBX n    *
SUBX t
```

* Note: These instructions using a literal normally generate a two-byte field. If the literal is outside the range -32,768 through +32,767, an operand of the form =Dxxxx should be used to generate a four-byte literal (for example, =D40000 or =DX'FC000022'). Six-byte literals must be separately defined using the FTLY instruction.

5.97 TEXT Assembler Directive

TEXT

Function: This Assembler directive is used to store text strings. It may have any number of operands, each of which may specify a string in either ASCII (character) or hexadecimal formats. It is typically used to store literal strings, messages, tables of values, etc. See the SRA instruction for the method of addressing generated data.

Format: TEXT op1{,op2,... }

Example:

Instruction	Generated value
TEXT C'ABCD',X'07FF'	4142434407FF

5.98 TIME SET.TIME - Get/Set System Time and Date

TIME
SET.TIME

Function :
TIME : Accumulator = system time and system date
SET.TIME : System time and system date = accumulator

The TIME instruction is used to get the system's time and date in internal format. It is a Monitor call that loads the accumulator FP0 as follows:

T3 - (upper two bytes of FP0) contains the date as number of days past DEC 31,1967;

D0 - (lower four bytes of FP0) contains the time as number of milliseconds past midnight.

The SET.TIME instruction is a Monitor call that resets the system's internal time and date. The accumulator FP0 must be set up as described above before SET.TIME is executed.

Formats: TIME SET.TIME

5.99 TLY - see DTLY

5.100 WRITE - Write Byte

Function: Next byte in asynchronous byte buffer = indirect character

The byte 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.

Format: WRITE r

5.101 XCC - Exchange Characters

Function: Indirect character < - > indirect character

The character addressed by the first operand is exchanged with that addressed by the second operand.

Formats: XCC r,r

This instruction allows the "Test and Set" function to be implemented, which may be used to prevent shared usage of sections of code. For example:

```
          SRA   R15,LOCKTBL  Set R15 to the Lock byte, which may contain
          CMNT  *            either a X'00' (unlocked) or X'01' (locked).
LOCKED?   MCC   X'01',R2     Move "Locked" flag to scratch location;
          XCC   R2,R15       Exchange old lock and store "Locked" flag;
          BCE   R2,X'00',OK  If old flag was X'00', we are ok to continue.
          RQM   *            Else wait a while...
          B     LOCKED?      And try again.
OK        EQU   *            Start of non-shared code
          ...
          ...
          MCC  X'00',R15    Unlock the non-shared code Lock byte
```

5.102 XOR - Logical XOR of a Byte

Function: Indirect byte = (indirect byte) logically XOR'ed with (operand)

The byte referenced by the first operand is logically exclusive-OR'ed with the byte referenced by the second operand. The byte referenced by the second operand is unchanged.

Formats: XOR r,n
XOR r,r

5.103 XRR - Exchange Registers

XRR

Function: Address register < - > address register

The first operand is exchanged with the second operand. The "attached" or "detached" state of the address registers is not changed and is not relevant to the operation of this instruction.

Format: XRR r,r

5.104 ZB - Zero Bit

ZB

Function: Bit = 0

The referenced bit is set to an "off" (0 or false) condition.

Format: ZB b

5.105 ZERO - Set Operand to Zero

ZERO

Function: Operand = 0

The contents of the operand are replaced by zero.

Formats:
```
ZERO  d
ZERO  f
ZERO  h
ZERO  t
```

CHAPTER 6

THE DEBUGGER

6.1 The Assembly Debugger

The assembly Debugger is a powerful tool that allows the programmer to control program execution, to display and change variables, and to set breakpoints. The Debugger may be called by the terminal's BREAK key. If the system is executing a BASIC program, the BREAK key calls the BASIC Debugger instead of the assembly Debugger; in this case, the BASIC Debug command "DEBUG" or "DE" will transfer control to the assembly Debugger.

The Debugger signifies its control by typing a message of the form:

I f.l

where "I" is an indication that the system received a BREAK key signal, "f" is the decimal frame number where execution was interrupted, and "l" is the hexadecimal location of the instruction that was interrupted.

The Debugger's prompt character is the exclamation point (!).

The Debugger is also entered when the system encounters an unrecoverable error.

6.1.1 System Privileges and Debug Usage

Users with system privilege levels zero and one have only these Debug commands available: "G", "P", "END" and "OFF". Users with system privilege level two have access to all Debug commands, except "DI" (see next section).

6.1.2 Disabling the Debugger

Access to Debug commands other than "G", "P", "END" and "OFF" for <u>all</u> users may be inhibited by the "DI" (Disable) Debug command from the SYSPROG account. This is a method of improving system integrity by preventing inadvertant or deliberate change of data, etc., via the Debugger.

Once disabled, the Debugger can be enabled only via the same "DI" command (it is a toggle switch).

6.1.3 Inhibiting the Break Key

The TCL BREAK-KEY-OFF verb may be used to inhibit entry to the Debugger via the BREAK key for a particular line. The BREAK-KEY-ON verb reverses the effect of BREAK-KEY-OFF. See the Terminal Control Language for more information on these verbs.

6.2 Debug Context Switching

The Debugger is internally called via a subroutine call to one of the entry points in frame one (1). At this time, if the Debug state is to be entered, a special Monitor call is executed. (The Debug call may be ignored, for example, on a BREAK key entry if the BREAK key is inhibited).

The Monitor sets a flag in the PIB to indicate that the process is in the Debug state. In this state, whenever the process is activated, the special frame at the original PCB FID plus two (the Tertiary Control Block) is used as the effective PCB. This frame is permanently assigned as the Debug state control block, and is sometimes referred to as the Debug Control Block or DCB.

The DCB has its own set of address registers and all functional elements needed by the Debugger. The DCB's Register One (program counter) 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 takes over. Note that, at this time, the DCB Register One 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, and is normally used by the Debug "G" (GO), "END", "BYE" and "OFF" commands.

6.3 Debugger Traps and Error Conditions

When a process is executing, certain conditions can cause it to enter the Debug state. Typically, these are unrecoverable error conditions, although artificial calls to the Debugger can be forced by the Monitor for special processing conditions. The Debugger traps and their related entry points are shown on the next page.

In the case of traps marked with an asterisk (*) the affected register number is stored in the half tally ACF, for use by the Debugger.

In the case of a FORWARD LINK ZERO trap, if the exception subroutine tally XMODE is non-zero, the Debugger will transfer 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. Further details are in the next chapter.

In addition, the Debugger is called by the Monitor under the following conditions:

1. A message has been transmitted to the process by another process (via the TCL MSG verb). The Debugger saves the context via the mechanism described earlier, and transfers control to the message printer. Entry point 13 is used.

2. A disc error has occurred when the process generated a frame-fault. The disc error handler is invoked to log the message in the SYSTEM-ERRORS file. Entry point 9 is used.

3. The BREAK key is pressed on the user's terminal. Entry point 10 is used.

All unrecoverable error conditions cause a message of the form:

```
ABORT @ f.d
```

to be displayed on the terminal attached to the process, where f is the decimal FID and d is the hexadecimal displacement within the program frame where the trap occurred. In addition, for register-related error conditions (traps marked with an asterisk on the next page), the number of the register causing the trap is displayed, for example:

```
FORWARD LINK ZERO; REG = 4
ABORT @ 511.1
```

Debugger Traps (Aborts)

Entry	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 byte 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 (e.g. "INC r" or MIID) has caused the register to go beyond the end of the linked frame set.
6 *	BACKWARD LINK ZERO	A decrementing instruction (e.g. "DEC r") has caused the register to go before the beginning of the linked frame set.
8	REFERENCING ILLEGAL FRAME	A register has a frame number that is beyond the allowable limits for this disc configuration.
11	HALT	A HALT instruction has been executed.

6.4 Summary of Debug Commands

Command format	Description
{f}addr{;window} {f}/symbol{;window}	Direct data display.
{f}*addr{;window} {f}*symbol{;window} Rr R.r	Indirect data display.
ADDD n1 n2	Adds decimal "n1" and "n2".
ADDX n1 n2	Adds hexadecimal "n1" and "n2".
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	Same as END but preserves the Breakpoint and Trace tables.
D	Displays the Breakpoint and Trace tables.
DI	Disables Debugger for all lines.
DTX n	Converts decimal "n" to hexadecimal.
E{n}	Sets the execution Step to "n"; if "n" is 0 or null, clears execution step.
END	Returns unconditionally to TCL. Clears B, E, N, M, T and F commands.
Fn,m	Frame substitution of FID "m" for FID "n" during execution of instructions.
G{addr}	Continues execution at address specified, or at point of interruption if no addr.
line feed or escape	Equivalent to "G" command for convenience.
K{addr}	Kills specific Breakpoint entry, or all entries if "addr" missing.
Lfid	Displays Link fields of frame specified.
M	Toggle to turn on/off Modal execution trace.
MULD n1 n2	Multiplies decimal "n1" by "n2".
MULX n1 n2	Multiplies hexadecimal "n1" by "n2".

Summary of Debug Commands continued

Command format	Description
N{n}	Sets delay counter to "n" or 0 if null; inhibits Debug entry until "n" Breaks, Steps, etc.
OFF	Logs user off system.
P	Toggle to suppress/allow terminal output.
SUBD n1 n2	Subtracts decimal "n2" from "n1".
SUBX n1 n2	Subtracts hexadecimal "n2" from "n1".
T{f}addr{;window} T{f}/symbol{;window} T{f}*symbol{;window}	Traces location specified; up to four direct and four indirect traces can be set, and the data so traced will be displayed on every entry to Debug.
U{addr}	Untrace; clears Trace table entry, or all entries if "addr" is null.
XTD n	Converts hexadecimal "n" to decimal.
Yaddr Y/symbol Y*symbol	Data Breakpoint; interrupts execution if the value contained at the address or in the symbol changes. Up to two traces can be set; "Y" turns them both off.
>statement	Executes TCL statement and returns to Debug.
>>	Steps up one TCL level.
<<	Steps down one TCL level.
<	Returns to current TCL level.

6.5 Symbolic Debugging

One of the powerful features of the assembly Debugger is the ability to specify symbolic variable names for display and data change. To use symbols, the SET-SYM verb must have been used at TCL before entry to the Debugger. It is a good idea for assembly programmers to set up an automatic execution of SET-SYM when logging on to their account (via a LOGON PROC), so as not to forget to do so when debugging a program. The format of SET-SYM is:

SET-SYM filename {(T)}

Two files can be specified. Normally, "SET-SYM PSYM" is used so that all the "global" PSYM symbols can be referenced. Local references may 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. "SET-SYM TSYM (T)" may be used to reference any local symbols. In order for this to work correctly, TSYM must be in the state just after the assembly, so that it has all the local symbols in it. Alternatively, after the assembly, all TSYM symbols may be copied to a more permanant file, and the second SET-SYM made to that file.

Once the SET-SYM(s) has(have) been executed, the Debug symbolic display commands "/" and "*" can be used.

6.6 Address Specification in the Debugger

There are several ways to specify a byte address in a Debug 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. The general formats are:

Notation	Description
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.

For example, "123.7F" refers to frame 123, byte hexadecimal 7F, and is therefore equivalent to "123,127", ".7B.7F" and "7B,127".

For convenience, if the FID is omitted, the user's PCB is assumed. Therefore, for example, the notation ".100" is the byte address of location hex 100 in the user's PCB.

6.7 Indirect Addresses

An indirect address specifies a register to indirectly address the required byte. It has the forms:

Notation	Description
Rr	Register r, where 0 <= r <= 15.
R.r	Register r, where 0 <= r <= F (hexadecimal).

6.8 Windows

When displaying data using one of the Debug display commands, the number of bytes to be displayed may be specified by the window notation. A window is a number or numbers that follow the address notation, separated by a semicolon (;). Its formats are:

Notation	Description
;n	Display n bytes; n is a decimal number.
;m,n	Display as above; start m (decimal) bytes before specified address.
;.m,n	Display as above; start m (hex) bytes before specified address.
;.n	Display n bytes; n is a hexadecimal number.
;m.n	Display as above; start m (decimal) bytes before specified address.
;.m.n	Display as above; start m (hex) bytes before specified address.
;tn	"Immediate symbol" window; see Immediate Symbol in the chapter on the Assembler; t is the symbol type-code; n is the offset; typically used to display bits.

6.9 Bit Addressing

The special form of window:

```
;Bn
```

is used to address the nth bit displaced off the previous address. For example,

```
100.66;B0
```

displays the high-order bit of the 66th hexadecimal location of frame decimal 100.

6.10 Displaying Data

To display data at the "!" prompt, the following notation may be used:

{f}addr{;window}

where "addr" and "window" are as defined previously, and "f" is a single character format code as shown below:

Format	Description
C	Display data in ASCII character format. Non-printable characters display as a period (.); system delimiters SB,SVM,VM,AM and SM display as [,\,],^ and _ (underscore or back-arrow).
X	Display data in hexadecimal format.
I	Display data in integer format; window must be 1,2,4 or 6.

If format and window are unspecified, the previously used values will be reused. For example:

C1234.7F;100

will display 100 bytes, in ASCII format, starting at location hexadecimal 7F (127 decimal) in frame decimal 1234.

6.10.1 Continuing Display

After the Debugger displays data, it will prompt with an "=" rather than a "!". At this point, the user has the option of continuing the display, changing the displayed data, terminating the operation, or a combination of these actions.

1. To terminate the display and return to the "!" prompt, a carriage return should be entered.

2. To continue the display to the next forward window, a control-N or a line feed should be entered. The line feed continues display on the same line; the control-N causes a new line and a new address to be displayed before the data.

3. To continue the display to the previous window, a control-P should be entered. This causes a new line and a new address to be displayed before the data.

6.10.2 Changing Data

Data may be changed at the "=" prompt, before entering one of the above four characters. The replacement value may be entered in bit, character, hexadecimal or integer mode, and does not have to correspond to the format that has been displayed except for bit mode replacement. To change data:

Replacement Mode required	Entry Format
Character	Enter: 'data..... (i.e., a quote followed by the data). Note : 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. Also note there is no trailing quote.
Hexadecimal	Enter: .data..... (i.e., a period followed by the data). Note : 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.
Integer	Enter: n (i.e., the decimal number). Note : The displayed window must be 1,2,4 or 6 only.
Bits	Enter: xxxxx... where x=0 or 1, a string of bits. Note : This is valid only when a bit is displayed.

The data may be terminated by a carriage return (change data; return to "!" prompt), a control-N or line feed (change data; display next window), or a control-P (change data; display previous window).

For example:

Entry at "=" prompt	Action
cr	Terminates display, returns to "!" prompt.
lf	Displays next window on same line.
.1234cr	Replaces two bytes with hex digits 1234, then terminates display and returns to "!".
'Aaron control-N	Replaces six bytes with string "Aaron"; continues display to next window on next line; space shown for clarity only.
0 lf	Replaces window with decimal zero field; displays next window; space shown for clarity only.

6.11 Symbolic Display

There are two symbolic display operators that may be used with symbols: the "/" and the "*". To display data at the "!" prompt, the following notations may be used:

{f}/addr{;window}
{f}*addr{;window}

The "/" display operator is used as a direct reference to the contents of the symbol following. It should therefore be used with symbols of type B (bit), C (character), D (double tally), F (F-type tally), H (half tally) and S (storage register, if it is desired to see the contents of the register, not the data that it addresses).

The "*" display operator is used as an indirect reference to the data addressed by the symbol, which may therefore be only of types R (address register) and S (storage register). For example,

Entry	Action
/ABIT	Displays contents of ABIT (0 or 1).
/DO	Displays contents of DO as an integer.
X/DO	As above, display is in hexadecimal.
/ISBEG	Displays contents of SR 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.

It should be noted that if "format" and "window" are not specified for a symbol, natural 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.

6.12 Debug Traces

The "T" command can be used to set up to four traces. Once set, on every subsequent Debug entry (including system traps), the traced data will be displayed automatically. The Trace command formats are similar in structure to the Display command:

T{f}addr{;window}
T{f}/symbol{;window}
T{f}*symbol{;window}

Note - the window display is limited to 127 bytes, and the "m,n" format window cannot be used.

The U command may be used to Un-trace specific or all traces.

6.13 Execution Control

The "B", "E", "M", "N" and "Y" commands are used to control execution of a program.

6.13.1 Breakpoints

The "B" command sets an execution breakpoint at a location as shown on the MLISTing of a program. If the process reaches that location, the Debugger is entered. Up to four such breakpoints can be set; each one individually, or all can be deleted by the K command. For example, the command:

```
B511.3
```

will cause a breakpoint to be set at Frame 511, location 3. In addition the special form:

```
Bf.0
```

will cause a breakpoint on every location in the specified frame. This form is useful when the user is not sure where in a specific frame execution may begin; this form will break on any entry to the frame.

6.13.2 Execution Step

The "E" command is typically used in the form "E1", which single-steps execution. If any other value is used, for example "E10", then that number of instructions is executed before returning to Debug control. The forms "E" or "E0" turn off the execution step.

6.13.3 Delay Control

The "N" command is used to "delay" entry to Debug for a specific number of breakpoints, execution steps, etc. If "Nn" is used, n entries to Debug are inhibited. For example, if the following commands are used:

```
E10
N9
```

100 instructions are executed before Debug gets control. Every ten instructions, a message is printed (because of the "E10"), but execution continues.

6.13.4 Modal Execution Tracing

The "M" command is a switch that turns modal execution tracing on or off. If 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.

6.13.5 Data Value Tracing

The "Y" command adds an entry to the data trace table; the address (symbolic or direct) specified is monitored and the Debugger entered when the value changes. "Y" by itself turns the data trace off.

6.14 Continuing Execution

The "G" command with no "address" is used to continue execution at the point of interruption. The line feed or escape command is equivalent to this form of the "G" for convenience.

The "G addr" command may be used to unconditionally change the point of execution. If the Debugger was entered via one of the system traps, the "G" command with no address will not be accepted: "END", "OFF", "BYE", or "G addr" must be used, the last only if a location is known where execution can safely resume.

6.15 Terminating Execution and Changing TCL Levels

A process may execute at one of several levels of TCL. Typically, the EXECUTE statement in BASIC steps "up" one level to process a TCL statement, and steps "down" to return to the BASIC program. Stepping up and down may also be done via the Debugger.

The "<", "<<", "END" and "BYE" commands provide a means to terminate execution under different conditions. "END" and "BYE" always terminate execution and return to the TCL state at the lowest (LOGON) level. If the process had been executing at a higher TCL level, all such levels are released. To terminate execution at the current TCL level, the "<" command should be used. "<" at the lowest level of TCL is equivalent to "END".

The ">>" and "<<" commands allow the process to step up or down TCL levels. ">>" by itself will suspend the current level and re-enter TCL at one level higher; the prompt at TCL will change to ">>" as an indication that the process is not at the lowest level of TCL. ">statement...." may be used to execute any short TCL statement from Debug, and return to Debug. A useful example may be to send another user a message from a terminal which is in the middle of processing: Debug is entered via the BREAK key, ">MSG ....." is typed in, and after the message is transmitted, Debug is exited via the "G" or line feed command, returning the process to normal execution.

The "<<" command is used to step back down one TCL level.

Summarizing:

Command	Action
END BYE	Terminate execution; return to TCL at lowest level.
<	Terminates execution; returns to TCL at current level.
<<	Terminates execution; returns to TCL at next lower level.
>>	Suspends current process; goes to TCL at next higher level.
>stmnt	Suspends current process; executes "stmnt" as a TCL statement; returns to Debug at the current level.

6.16 Changing Frame Assignments

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. Its format is:

Fn,m

where "n" is the decimal frame number that is to be changed, and "m" is the decimal frame number temporarily assigned to it. Once this command has been executed, the Debugger will monitor every frame change as the process executes, and any external BSL or ENT instructions to frame "n" will be internally modified to go to frame "m".

In practice, the user modifies an existing program, changes the FRAME statement in the program before reassembly, and MLOADs the object into the temporary frame, before using the "F" command. 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 Debug command:

F420,511

then routes all execution transfers from frame 420 to frame 511.

Note that if a frame is reassigned, breakpoints must be set using the reassigned frame number, not the real one.

Only one frame reassignment at a time be in effect for a process.

Entering "F" alone turns frame reassignment off.

6.17 Arithmetic Commands

The following commands may be used for arithmetic computation at the Debug level, and are identical to their TCL verb equivalents:

```
ADDD n1 n2          Add decimal values n1 and n2.
ADDX x1 x2          Add hexadecimal values x1 and x2.
SUBD n1 n2          Subtract decimal value n2 from n1.
SUBX x1 x2          Subtract hexadecimal value x2 from x1.
DTX  n              Convert decimal value n to hexadecimal.
MULD n1 n2          Multiply decimal values n1 and n2.
MULX x1 x2          Multiply hexadecimal values x1 and x2.
XTD  x              Convert hexadecimal value x to decimal.
```

6.18 Other Debug Commands

The "P" command is a toggle switch that turns the terminal print on or off. It is identical in operation to the TCL "P" verb.

The "L" command may be used to display link fields of a frame. Its formats are:

```
Laddr
L*symbol
```

where "symbol" should be an address register or storage register only. The links fields are displayed in the form:

```
nncf : forward.link   backward.link : npcf
```

where the terms "nncf" and "npcf" are "number of next contiguous frames" and "number of previous contiguous frames." All four fields are displayed in decimal. To display link fields of frame "f" in hexadecimal, use:

```
Xf.1;1        for nncf;
Xf.2;4        for forward link;
Xf.6;4        for backward link;
Xf.A;1        for npcf.
```

6.19 Debug Messages

When the Debugger is entered due to execution interruption, one of the following messages will display:

Message	Interrupt due to
B f.l	BREAKPOINT found at frame f, location l.
E f.l	EXECUTION step at frame f, location l.
I f.l	BREAK KEY at frame f, location l.
M f.l	MODAL entry/External BSL to frame f, location l.
R f.l	External RTN to frame f, location l.

6.20 Address Representation

When the Debugger displays an address, the frame number is always in decimal and the location is always in hexadecimal. If the displayed address is from a register which is in linked mode, a plus sign (+) precedes the frame number just as an indication.

For example,

```
!C*ISBEG +1200.B  .TEST=
```

ISBEG addresses frame (decimal) 1200, displacement (hexadecimal) B (decimal 11); the "+" in front of the 1200 indicates that ISBEG is in linked addressing mode.

```
!C*TSBEG;16 1189. .20 JUN 1946_..=
```

TSBEG addresses frame (decimal) 1189, displacement 0, in unlinked

addressing mode because there is no "+" in front of 1189.

CHAPTER 7

SYSTEM CONVENTIONS

7.1 Introduction

This is the difficult aspect of working with assembly language. The operating system has many conventions that must be adhered to at all times. Generally speaking, these conventions deal with the use of global variables and shared buffer spaces.

7.2 Global Variables

Note that all Permanent Symbol File (PSYM) variables are GLOBAL and can be used by all routines. This is where conventional usage comes in. Each routine uses only a small subset of the available elements. Local variables are not normally defined as in other assembly languages, though this is possible (see Defining an Additional Control Block, later).

Generally, the "lower" the level of a system subroutine, the fewer elements it uses. Thus a "higher" level subroutine may safely call a "lower" level subroutine without losing any data. Also, subroutines that can be grouped together (for example, file I/O routines or terminal I/O routines) tend to share many elements.

Certain elements, however, are considered "totally scratch" in that they may be used by nearly any subroutine. These elements are as follows:

Bits	:	SB60, SB61
Tallies	:	T4, T5
Double Tallies	:	Accumulator (D0, D1), D2
F-type Tallies	:	FPX (overlays SYSR0), FPY (overlays SYSR1)
Registers	:	R14, R15
Storage Registers	:	SYSR0 (overlays FPX), SYSR1 (overlays FPY), SYSR2

The use of these elements is not even mentioned in the documentation for most system subroutines (next chapter).

7.3 Re-entrancy

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, no storage internal to a program is utilized. Instead, each process uses its own storage space.

The storage space most commonly used by a process is that in its Primary and Secondary Control Blocks. The Primary Control Block (PCB) is addressed via Address Register Zero, and the Secondary Control Block (SCB) via Address Register Two. 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 Debug processor, 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.

There are enough PCB and SCB storage areas defined in the PSYM file to accommodate most user programs. If a program must use storage internal to itself, however, it 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 will then wait until the first process "unlocks" it:

```
            ORG   0
            TEXT  X'00'          Initial condition for lock byte
            CMNT  *              (Note usage of storage internal
            CMNT  *              to program)
            ...
            ...
            ...
LOCKED?     MCC   X'01',R2       Move "Locked" flag to scratch location;
            XCC   R2,R15         Exchange old lock and store "Locked" flag;
            BCE   R2,X'00',OK    If old flag was X'00', we are ok to continue.
            RQM   *              Else wait a while...
            B     LOCKED?        And try again.
OK          EQU   *              Start of non-shared code
            ...
            ...
            ...
UNLOCK      MCC   X'00',R1       Unlock the "lock" flag
            BSL   DECINHIB       Conventional way to decrement INHIBITH
```

7.4 Defining an Additional Control Block

If it is necessary to define storage elements or buffer areas that are unique to a process, one of the unused frames PCB+30 or PCB+31 may be used. The following sequence of instructions is one way of setting up an AR to a scratch buffer:

```
.
.
.
MOV  R0,R3
DETZ R3          Set R3 "detached", with displacement of zero
INC  R3FID,30    Set R3 to PCB+30
.
.
.
```

Register Three can now be used to reference buffer areas, 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, exit to TCL via WRAPUP will reset R3 to PCB+3.

7.5 PCB Fields

The Primary Control Block, or PCB, is mapped below. All elements in the PCB are accessed via Address Register Zero, which always addresses, in unlinked mode, byte zero of the PCB.

7.5.1 PCB Fields - The Accumulator

The accumulator and its extension occupy fourteen bytes in the PCB. The accumulator is used:

1. In LOAD and STORE instructions;
2. In arthmetic instructions;
3. In the LAD instruction;
4. In certain string scanning and moving instructions to count the number of bytes scanned or moved;
5. In certain string-to-binary and binary-to-string conversion instructions.

The accumulator consists of two four-byte tallies, labeled D1 and D0. Another six-byte tally, FPY, is used for extended precision division instructions only. D1 and D0 occupy bytes 8 through 15 of the PCB, and six-, four-, two-, or one-byte tallies, as well as individual bits, may be addressed symbolically.

The following shows the format of the accumulator and the symbolic names that address various sections of it:

```
                 ___________________________________________
Byte no.: |  8 |  9 |  A |  B |  C |  D |  E |  F |
in PCB    |         | <---------- FP0 ---------> | <- 6-byte field
          | <------ D1 -----> | <------ D0 -----> | <- 4-byte fields
          | <- T3 ->| <- T2 ->| <- T1 ->| <- T0 ->| <- 2-byte fields
          |  H7|  H6|  H5|  H4|  H3|  H2|  H1|  H0| <- 1-byte fields
       B63 ..........................................  B0  <- bits
```

The symbols used above may be used to address sections of the accumulator. Individual instructions such as LOAD and ADD also address the accumulator differently depending on the operand, so a mental picture of the above is important in understanding how the accumulator functions. Generally speaking:

1. Extended precision arithmetic instructions such as ADDX affect FP0; DIVX also affects FPY (not shown above);
2. Normal precision arithmetic instructions such as ADD affect D0; MUL and DIV also affect D1;
3. Instructions that count string lengths, as well as the LAD instruction, use T0 only;
4. Conversion instructions use FP0 for data and T3 as a parameter.

In the documentation for assembly instructions and system subroutines, the term "accumulator" usually means a section of the accumulator proper - usually T0 or D0. If the precise section is not clear from the context, it will be specified by referring to "the accumulator T0," or "the accumulator D0," for example.

7.5.2 PCB Fields - The Scan Characters

There are three one-byte fields called SC0, SC1 and SC2, which contain the "scan characters" used in string scanning and moving. See the MIID, MIIDC, MIITD, SICD, SID, SIT and SITD instructions for more information on the use of these fields.

7.5.3 PCB Fields - The Subroutine Return Stack

The Assembly subroutine return stack is in the PCB at bytes X'182' through X'1AF'. 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 pushed by four bytes. On executing a subroutine return instruction, the stack pointer is used to get the return address, and the pointer popped by four bytes.

The stack pointer is stored as a two-byte tally at locations X'182' and X'183', and is symbolically referenced as "RSCWA". An empty stack condition is when this tally contains the value X'0184'; a full stack condition is when it contains the value X'01B0'.

Each stack entry is four bytes:

```
|-0-|-1-|-2-|-3-|
| FID   | displ |
|       |acement|
-----------------
```

Note that the FID for executable programs has only twelve significant bits since all executable programs must be in frames 1-4095.

An entry may be deleted from the return stack by the instruction "DEC RSCWA,4". This is mandatory if a subroutine is to be exited without using a RTN instruction. The entire return stack may be reset by the instruction "MOV X'184',RSCWA", 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.

7.5.4 PCB Fields - XMODE

See the section on the XMODE Interface in the next chapter for information on the use of this element.

7.5.5 PCB Fields - RMODE

When the WRAPUP processor is called to store or print messages, a return may be requested by placing a mode-id in the tally RMODE. When WRAPUP completes the requested processing, an ENT* RMODE instruction transfers control. Also see the section on the WRAPUP processor in the next chapter.

7.5.6 PCB Fields - WMODE

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

mode-id has been stored in it. Processors that require special handling to "clean up" may gain contol 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.

7.5.7 PCB Fields - OVRFLCTR

When the system software gets space from the system's overflow space pool, the first frame so obtained is placed in the special double tally OVRFLCTR. This is typically done when a sorting or selecting function such as SORT, SELECT, etc. is being performed. The extra space needed by the processor 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 processor 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. TCL-II initializes OVRFLCTR, however, for "update" verbs used with more than one item; in this case, user code must use some other means of returning space, perhaps via WMODE.

7.5.8 PCB Fields - INHIBIT and INHIBITH

Normally, the terminal's BREAK key will cause the process to enter the Debug state (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.

INHIBITH is used by the system during overflow management, disc writes, etc; 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.

7.6 SCB Fields

The Secondary Control Block, or SCB, is mapped below. All elements in the SCB are accessed via Address Register Two, which always addresses, in unlinked mode, byte zero of the SCB.

7.6.1 SCB Fields - User Available Elements

The following elements in the SCB are unused by the system software, and are thus freely usable by user-written assembly programs:

BITS	:	SB24 through SB35
CHARACTERS	:	None
DOUBLE TALLIES	:	None
HALF TALLIES	:	None
STORAGE REGISTERS	:	SR20 thorugh SR29
TALLIES	:	CTR30 through CTR42

Note that there are no address registers available freely; availability depends on the interface with the system software. Additional elements may be stored by setting up an additional control block (discussed earlier).

7.7 Conventional Register and Buffer Usage

The following illustrates the pre-assigned buffers, or workspaces, available to a process. Note that, unlike in other systems, program space is rarely used to store variables (other than text strings). All programs should be "pure" or re-entrant code.

The buffers as shown in the table should give ample room to store and manipulate string data. Counters, bits, and pointers are stored in PSYM-defined PCB and SCB elements.

Conventionally, buffer beginning pointers (ISBEG, etc.) point one byte before the actual data. This is so that the string scanning and string moving instructions, which always increment an AR before testing or moving the next data byte, work correctly.

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

```
MOV  HSBEG,HS       Set HS Register to start of buffer
MIID R15,HS,X'C0'   Copy a string until a 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 ISINT does the same for the IS, OS and HS buffers, and also calls WSINIT.

The buffer pointers are sometimes changed by processors, but reference is always made symbolically, 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.

In the following table, the various workspace pointers are shown, along with the size of the buffers. "Not a buffer" indicates that there is no permanently assigned space associated with the storage 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.

7.7.1 Table of Buffers and Buffer Pointers

Reg num	PSYM name	Beginning and Ending Pointers	Size of buffer	Description
0	-	-	-	Primary Control Block Pointer
1	-	-	-	Program Counter
2	-	-	-	Secondary Control Block Pointer
3	HS	HSBEG Fixed HSEND Floating; must point to current end of data in the HS buffer	64 Kbytes	Stores messages to be printed at end of processing; area beyond HSEND may be used as scratch; if needed to save data, conventions are: strings separated by SM's; character after SM is an X; string terminated with a SM and a Z; HSEND points to the SM before the Z
4	IS	ISBEG Fixed ISEND Floating; end of current data pointer	64 Kbytes	Stores compiled string for RECALL; data for EDITOR; no conventions
5	OS	OSBEG Fixed OSEND Floating; end of current data pointer	64 Kbytes	Stores compiled string for RECALL; data for EDITOR; in RECALL, area past OSEND is scratch; no conventions
6	IR	No pointers	Not a buffer	Used for file I/O; points to current item in the file if using standard system file I/O subroutines; not to be used for other purposes
7	UPD	UPDBEG No meaning UPDEND No meaning	Not a buffer	No conventionally fixed usage, except on tape I/O; UPD AR is freely usable
8	BMS	BMSBEG Fixed BMSEND Floating on last byte of item-id	50 bytes	Stores item-id when interfacing with system file I/O; item-id's are terminated with an AM
9	AF	AFBEG Fixed AFEND Fixed	50 bytes	Scratch buffer in same frame as BMS; AF AR freely useable
10	IB	IBBEG Fixed IBEND Floating, end of current data pointer	140 bytes	Terminal input buffer; not to be used for other purposes
11	OB	OBBEG Fixed OBEND Fixed	140 bytes	Terminal output buffer; not to be used for other purposes
12	CS	CSBEG Fixed CSEND Fixed	100 bytes	Scratch buffer in same frame as BMS; CS AR may be freely used as a scratch register
13	TS	TSBEG Fixed TSEND Floating; points to current end of data	512 bytes	The TS buffer is used as a scratch area by various languages and processors; particularly useful in the Conversion interface; some processors use the TS buffer itself; most do not; but the area from TSBEG on may be treated as a scratch space in the Conversion interface; TS AR may be used as scratch
14	R14	-	Scratch Register	
15	R15	-	Scratch Register	

7.8 System Control Flow

The diagram below shows the flow of control between the operating system modules. After logging on to the system, a process is at TCL, which is the main interface with the terminal. An initial command word entered at TCL may be:

1. A PROC; the PROC processor gains control; it may call the TCL processor 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.

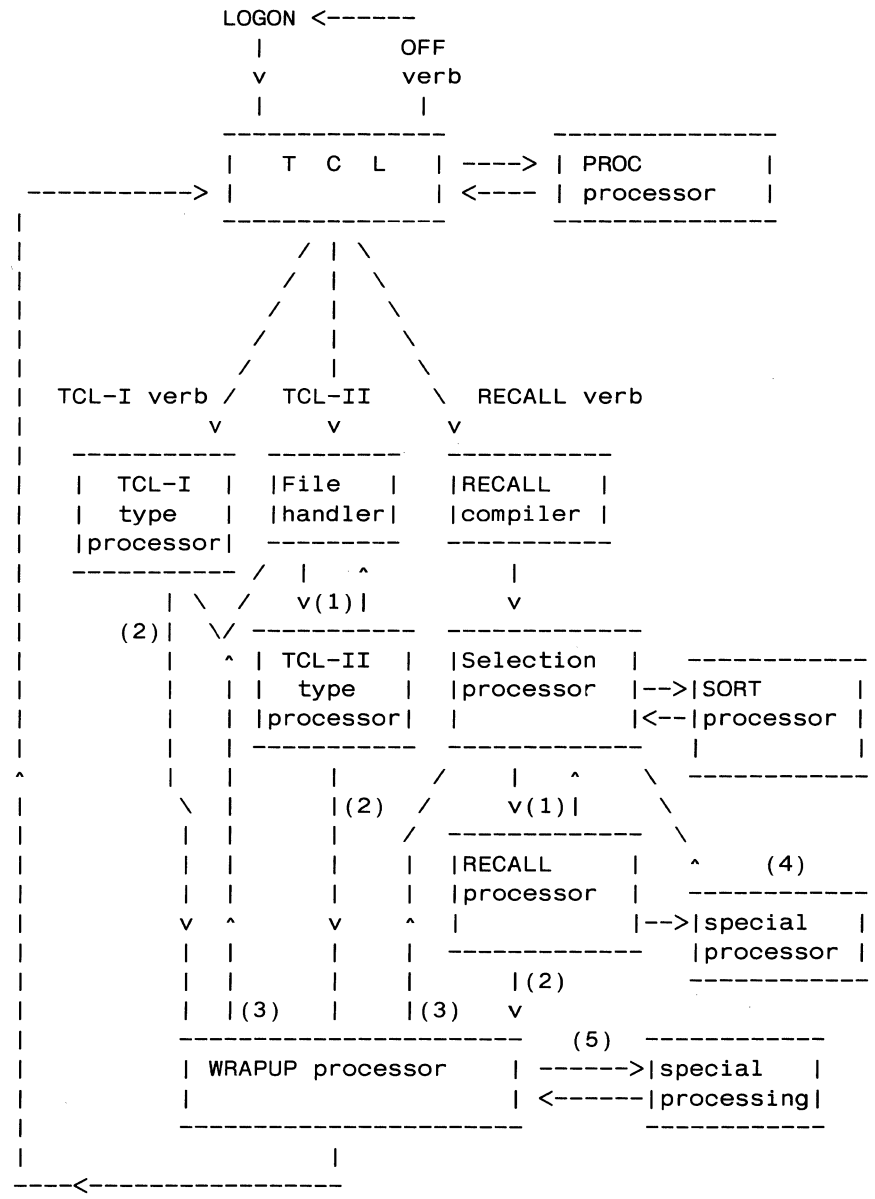
2. A TCL-I verb; typically these do not require file I/O; examples are TIME, SLEEP, and POVF.

3. A 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.

4. A RECALL verb; the RECALL compiler is first called to compile the statement. The resultant compiled string is passed to the Selection processor, which acts as the file-handler for the RECALL run-time. If the verb requires sorting (SORT, SSELECT), the SORT processor is called to sort the selected items on the specified sort-keys. The RECALL run-time processor is called as each item is selected, and it returns control to the selection processor after completion of processing for that item.

All processors return to TCL via the WRAPUP processor, which cleans up, closes spooler files, prints messages, etc. If a PROC was in control, it regains control rather than TCL.

7.8.1 Diagram of System Control Flow



SYSTEM CONTROL FLOW

Notes:
(1) This path is repeated as each item is retrieved by the file handler.
(2) Error message store/print path, and final exit.
(3) Control returns using tally RMODE.
(4) Special processing via MODEID3 in RECALL verb.
(5) Subroutine call using tally WMODE, just before return to TCL.

7.9 TCL Initial Conditions

When a process is at TCL, all workspace or buffer pointers (described earlier) are initialized (though the buffer data are whatever are left over from the last program). Also, all bits (flags) are cleared. Remembering these points is important when first writing assembly programs, since they define initial conditions.

The following defines these initial conditions at TCL:

1. MBASE MMOD MSEP	Base/Mod/Sep of the M/DICT.
2. EBASE EMOD ESEP	Base/Mod/Sep of the ERRMSG file.
3. USER	Contains: 5 = logged on; 3 = logging off.
4. Scan characters SC0/SC1/SC2	Contain X'FB' (SB), blank and blank.
5. ABIT through ZBIT, AFLG through ZFLG, SB0 through SB35	Zeroed.
6. All workspace pointers	Initialized to beginning and end of buffer spaces.
7. Terminal and printer characteristics (such as paper depth and width).	Initialized by the TERM TCL verb.

Note that this means that the process has access to the user's M/DICT (master dictionary) and to the ERRMSG file (that is, they are "open" to use classical terminology).

7.10 Interfacing via a Verb

A program that is to be called via a verb on the system may interface as a TCL-I verb or as a TCL-II verb. The TCL-I interface is adequate if no disc file I/O is to be done. The TCL-II interface is much more convenient for accessing a file and/or items in a file since it relieves the program of responsibility of file opening and item retrieval.

It is also possible to interface with the RECALL processor; in this case, RECALL obtains the data in "output" form (correlated and converted), creates a dummy item in the HS buffer, and then turns control over to the user program. In fact, the standard LIST-LABEL and REFORMAT verbs work in exactly this way. The RECALL interface requires more care than the TCL-I and TCL-II interfaces because the RECALL processor uses most of the available global elements, but it provides the full power of the RECALL Selection, Sort and Correlative/Conversion processing.

All three interfaces are covered in more detail in the next chapter.

7.11 Conversion Processor Interface

The best way of accessing assembly subroutines from the system is via the Conversion processor interface, described in the next chapter. This allows subroutines to be called from BASIC or RECALL, and allows parameters to be passed back and forth.

CHAPTER 8

SYSTEM SOFTWARE

8.1 Introduction

Assembly level programming in the ULTIMATE system is facilitated by a set of standard system routines that allow easy interaction with the disc file structure, terminal I/O, and other system routines. These routines work with a standard set of addressing registers, storage registers, tallies, character registers, bits, and buffer pointers, collectively called "functional elements." In order to use any of these routines, therefore, it is essential that the calling routine set up the appropriate functional elements as required by the called routine's input interface.

The standard set of functional elements is pre-defined in the permanent symbol file (PSYM), and is therefore always available to the programmer. Also included in the PSYM are the mode-id's (program entry points) for the standard system routines documented in this chapter.

The first part of this chapter, "User Program Interfaces," covers the methods of invoking a user-written program from the operating system and returning control to the operating system. The rest of the chapter, "System Subroutines," describes the routines available to the user program once it has been invoked.

8.2 Documentation Conventions

The entry and exit procedures for the user program interfaces are described individually. The system subroutines, unless otherwise specified, are meant to be called using a BSL instruction, and they return to the calling program via a RTN instruction.

A brief description is given for each interface and subroutine. In each case, the Input Interface, Output Interface, and Element Usage sections describe the functional elements used by the routine. The single letter following an element name describes its type: B=bit, C=character, H=half tally, T=tally (word), D=double tally, F=F-type tally, R=address register, S=storage register.

Even if not specified, the following elements may be destroyed by any routine:

Bits	:	SB60, SB61
Tallies	:	T4, T5
Double Tallies	:	Accumulator (D0, D1), D2
F-type Tallies	:	FPX (overlays SYSR0), FPY (overlays SYSR1)
Registers	:	R14, R15
Storage Registers	:	SYSR0 (overlays FPX), SYSR1 (overlays FPY), SYSR2

If no description follows an element name, it indicates that the element is used as a scratch element.

Input interface elements for many routines are divided into two sections: those labeled "User specified" and those labeled "System specified." The 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.

8.3 Summary of System Software Routines

1. User program interfaces

CONV interface
PROC interface
RECALL interface
TCL-I interface
TCL-II interface
WRAPUP interface
XMODE interface

2. Terminal and printer I/O routines

NEWPAGE	Skip to new page, print heading/footing
PCRLF	Print cr/lf sequence
PERIPHREAD1 PERIPHREAD2	Read asynchronous channel
PERIPHSTATUS	Get status of asynchronous channel
PERIPHWRITE	Write asynchronous channel
PRINT CRLFPRINT	Print text from object code to terminal
PRNTHDR	Initialize and print heading/footing
READIB READLIN READLINX	Read a line from the terminal
SETLPTR SETTERM RESETTERM	Set up characteristics of terminal, printer
WRITOB WRTLIN	Write a line to the terminal or printer

3. Disc file I/O routines

GETACBMS	Open the ACC file
GETFILE OPENDD	Open a file or dictionary
GETITM	Get next sequential item from file
GLOCK GUNLOCK	Lock or unlock a file group
GUNLOCK.LINE	Unlock all group locks for a line
HASH	Compute record that item-id hashes to
RETIX RETIXU	Read a specific item from a file
UPDITM	Write a specific item to a file

4. Space management routines

ATTOVF	Attach overflow frame automatically
GETBLK	Get a block of overflow frames
GETOVF	Get a frame of overflow space
NEXTIR NEXTOVF	Attach overflow frame via register
RDLINK	Read link fields of frame
RDREC	Read one frame
RELBLK	Release a block of overflow space
RELCHN	Release a chain of overflow frames
RELOVF	Release a single overflow frame
WTLINK	Write link fields of frame

Summary of System Software Routines continued

5. Tape I/O routines

TPBCK	Backward space tape 1 record
TPRDLBL TPRDLBL1 TPGETLBL	Read tape label
TPREAD TPRDBLK	Read a tape record
TPREW	Rewind the tape
TPWEOF	Write end of file
TPWTLBL TPWTLBL1	Write tape label
TPWRITE	Write a tape record

6. Miscellaneous

ACONV	Convert ASCII character to EBCDIC
CONV	Call Conversion processor
CVDxx subs	Convert ASCII decimal to binary
CVXxx subs	Convert ASCII hexadecimal to binary
DECINHIB	Decrement the INHIBITH counter
ECONV	Convert EBCDIC character to ASCII
HSISOS	Initalize IS, OS and HS buffer pointers
LINESUB	Get user's line number
MBDSUB MBDNSUB MBDSUBX MBDNSUBX	Convert binary to decimal ASCII string
SLEEP SLEEPSUB	Put terminal to sleep
TIME DATE TIMDATE	Get system time and/or date
SORT	Sort a string of keys
WSINIT	Initialize buffer pointers

8.4 User Program Interfaces

This section describes the various means by which the operating system can transfer control to a user-written program, and the methods for returning control to the system from the user program.

8.4.1 TCL-I Interface

The next few sections describe the TCL-I, TCL-II and WRAPUP interfaces. The system control flow should be kept in mind in order to understand the interaction between these processors. The flow diagram appears in the previous chapter. An example of a TCL-I verb and a TCL-II verb appear at the end of this chapter.

To invoke a user program as a TCL-I or TCL-II verb, a verb definition item is created in the master dictionary, with the mode-id of the program specified on line 2 (TCL-I) or on line 3 (TCL-II). Line one consists of a "P", optionally followed by one other character, to identify the item as a verb. The Terminal Control Language (TCL) processor then uses this information to transfer control to the user program. The entry point to the TCL processor is known as MD1 in the PSYM, but this is largely irrelevant to a user program. MD1 is normally entered only from WRAPUP or LOGON.

When MD1 is entered, TCL checks for PROC control, and if this is present, enters the PROC processor. 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 (documented next).

Input Interface

System 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

See MD1B (next) for continuation of the TCL-I interface.

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 master dictionary, and validate it as such. If no errors are found, the rest of the data in the input buffer are edited and copied into the IS work space, and control passes to the processor specified in the primary-mode-id attribute of the verb, or to the PROC processor if the data define a PROC (attribute 1="PQ").

If the TCL statement contains "options," which are an alphabetic character string and/or numerics enclosed in parentheses at the end of the statement, the options are parsed as described below. Examples of options are "(M)" and "(AZ,100-300)".

Input Interface

System specified:

IB	R	Points one character before the input data

Output Interface

BASE	D	|
MODULO	T	| =MBASE, MMODULO, MSEPAR
SEPAR	T	|
IB	R	| Point to the SM at the end of the
IBEND	S	| input line
BMS	R	| Point to the last character in the verb
BMSEND	S	| name (for RETIX)
IR	R	Points to the AM following attribute 4 of the verb item, or to the end-of-data AM in the item, or to the "Q" in attribute one 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 specifications are meaningful only if the first two input characters are not "PQ":

SCP	C	Contains the character immediately following "P" in the verb definition, if present, otherwise contains a blank
CTR0	T	Contains the primary mode-id specified in the verb definition attribute 2
MODEID2	T	Contains the secondary mode-id from the verb attribute 3, if present, otherwise 0
MODEID3	T	Contains the tertiary mode-id from the verb attribute 4, if present, otherwise 0

AFLG thru ZFLG	B	Option flags; AFLG set for "A" option, BFLG for "B" option, etc., thru ZFLG for "Z". Additionally, numeric options of the form "n", "n-m", ".n" or ".n.m" (last two are hex) are stored as shown below
NUMFLG1	B	set if any numeric option was present
NUMFLG2	B	set if second number was present
NUMFLG3	B	set if third number was present
D4	D	contains first number
D5	D	contains second number
D3	D	contains third number
OS	R	=OSBEG
IS ISBEG	R S	Point one character before the beginning of the edited input line; characters are copied from the IB, subject to the following rules:

1. All control characters and system delimiters (SB, SM, AM, VM, SVM) in the input buffer are ignored.

2. Redundant blanks (two or more blanks in sequence) are not copied, except in strings enclosed by single or double quote marks.

3. Strings enclosed in single quote marks are copied as: SM I string SB.

4. Strings enclosed in double quote marks are copied as: SM V string SB.

5. End of data is marked as: SM Z.

Exits

To 0,PROC-I if first verb line contains "PQ", otherwise to the entry point set up in CTR0. If the verb is not found in the master dictionary, or has a bad format, control passes to MD99 in the WRAPUP processor, which prints an error message.

Error number (in REJCTR)	Error type
2	Uneven number of single or double quote marks in the input data
3	Verb cannot be identified in the M/DICT
30	Verb format error (premature end of data or a non-hexadecimal character present in the mode-id)

8.4.3 TCL-II Interface

TCL-II is 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 processor after the verb has been decoded and the primary mode-id has been identified as that of the TCL-II processor (mode-id = 2). TCL-II exits to the processor whose mode-id is specified in MODEID2.

Typically processors such as the EDITOR, ASSEMBLER, LOADER, etc. 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 under these conditions.

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, and are listed below.

Option	Meaning
C	Copy - items retrieved are copied to the IS workspace
F	File access only - file parameters are set up, but any item-list is ignored by TCL-II; if this option is present, any others are ignored
N	New item acceptable - if the item specified is not on file, the secondary processor 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, of if a Select list is in effect
U	Updating sequence flagged - if items are to be updated as retrieved, this option is mandatory
Z	Final entry required - the secondary processor is entered once more after all items have been retrieved (the COPY processor, for instance, uses this option to print a final message)

The input data string to TCL-II consists of the file-name (optionally preceded by the modifier "DICT", which specifies access to the dictionary of the file), 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 preceded the TCL-II statement, the item-list will not be present; item-id's are obtained from the select-list instead of from the statement.

TCL-II Interface continued

Input Interface (from TCL-I; system specified)

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 processor 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-id's are to be copied
ISBEG	S	Standard system buffer where items are to be copied, if the "C" option is present

Output Interface

*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 (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
*FBASE	D	| Contain the base, modulo, and separation
*FMOD	T	| of the file being accessed
*FSEP	T	|

BASE	D	=FBASE, FMOD, FSEP on the first exit
MODULO	T	only
SEPAR	T	
DBASE	D	Contain the base, modulo, and separation
DMOD	T	of the dictionary of the file being
DSEP	T	accessed

The following specifications 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	Points to the SM terminating the item data if the "C" option is present
IR	R	Points to the last AM of the retrieved item copied, if the "C" option is present, otherwise points to the AM following the item-id on file
*RMODE	T	=MD201 if items are left to be processed, otherwise=0
XMODE	T	=0

Flags as set up by TCL-I if the input data contains an option string.

Note - Elements marked with an "*" must not be changed by the next level processor.

Error Conditions

The following conditions cause an exit to the WRAPUP processor with the error number indicated:

Error	Condition
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	File name illegal or incorrectly defined in the M/DICT

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

8.4.4 WRAPUP Interface

The WRAPUP processor prints error messages and returns to TCL if tally RMODE is zero. WRAPUP returns to the location specified in RMODE if it is non-zero.

There are several entry points in WRAPUP, used to print messages under different conditions. In all cases, the messages (and parameters) may be either stored in the HS buffer or may be immediately printed. They are stored if bit VOBIT is set; they are printed if VOBIT is reset, or if RMODE is zero.

Messages are stored in the HS buffer; HSEND is used as the pointer to the next available spot in the buffer. The message string is copied to this location with a SM and an "O" preceding it; the message is terminated with a SM and a "Z":

```
... SM O message ... SM O message ... SM 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 SM-"O" sequences, and the messages are printed. (No messages are printed if HSEND = HSBEG, however.)

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.

Entry point	Description
MD999	Terminates processing; messages previously stored in the HS buffer are printed if needed; closes spooler files, releases overflow, etc., and returns to TCL-I

Note: All entries below eventually enter MD999 if RMODE is non-zero.

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

8.4.5 CONV Interface

There are two distinct interfaces documented here; since they are closely connected, they are documented together. The entire section should be read carefully for a complete understanding of the methods involved.

1. The CONV entry point may be used to call the entire conversion processor as a subroutine, 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 (as documented in the RECALL manual).

2. The other use of this interface is to call a user-written subroutine from BASIC or RECALL. The conversion processor interface is the single most useful interface in the system. This is because a conversion routine can be called by BASIC and RECALL to perform special processing; the interface to both these processors is identical. From BASIC, a user subroutine is called via:

```
CONVERTED.VAL = OCONV(RAW.VAL, 'Uxxxx')
CONVERTED.VAL = ICONV(RAW.VAL, 'Uxxxx')
```

   where the unconverted or raw value in variable RAW.VAL is passed as a parameter to the user assembly code; the latter performs such action as needed, and the result is returned to the BASIC program as a value that (in the above instance) will be 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).

   The value xxxx is the mode-id of the user-written subroutine.

   From RECALL, a Correlative or Conversion code of the following format should be used:

```
Uxxxx
```

   where xxxx is the mode-id of the user-written subroutine. In this case, RECALL passes the unconverted or raw 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 RECALL.

In all 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 processors 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.

See the examples at the end of this chapter.

1. Input Interface to system conversion processor:

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	R	Points to the first character of the conversion code specification string for CONV; for CONVEXIT (see below), 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

2. Input interface for user-written subroutine:

TSBEG	S	Points one before unconverted parameter from BASIC or RECALL processor; value is terminated by any system delimiter
IS	R	Points to non-hexadecimal character in the Uxxxx string
MBIT	B	Set for ICONV function or from Selection processor in RECALL; reset if OCONV function or LIST/output processor in RECALL

Output Interface, either set up by CONV in case 1, or set up by user-written code in case 2

TSBEG	S	Points one before the converted value
TS TSEND	R S	Point to the last character of the converted value; a SM is also placed one past this location; TS=TSEND=TSBEG if a null value is returned
IS	R	Points to the AM/VM terminating the conversion code(s)

Element Usage - scratch

1. On a user-written call to the standard system conversion routines (via CONV), the following elements are considered scratch and may be destroyed.

2. As a corollary, therefore, these same elements are freely usable in user-written subroutines.

SB10	B
SB11	B
SB12	B
SC2	C
T5	T
T6	T
T7	T
CTR1	T
CTR20	T
CTR21	T
CTR22	T
CTR23	T
S4	S
S5	S
S6	S
S7	S

Plus R14, R15, FP0, etc. as defined in Documentation Conventions

Subroutine Usage

The number of additional levels of subroutine linkage required depends on the conversions performed.

Exit convention:

For user-written conversions, one of two methods of exit may be used:

1. 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 may be a VM or an AM, or anywhere before it.

2. 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.

8.4.6 PROC Interface

Summary

A user-written program can gain control during execution of a PROC by using the Uxxxx or Pxxxx command in the PROC, where "xxxx" is the hexadecimal mode-id of the user routine ("user exit"). The routine can perform special processing, and then return control to the PROC processor. Necessarily, certain elements used by the PROC processor must be maintained by the user program; these elements are marked with an asterisk in the table below.

Input Interface

*BASE *MODULO *SEPAR	D T T	Contain the base, modulo, and separation of the master dictionary
*PQBEG	S	Points one prior to the first PROC statement
*PQEND	S	Points to the terminal AM of the PROC
PQCUR IR	S R	Point to the AM following the Uxxxx or Pxxxx statement
*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)
IB	R	Is the current input buffer pointer (may point within either the primary or secondary input buffers)
*SBIT	B	Set if a ST ON command is in effect
*SC2	C	Contains a blank

		SBIT on	SBIT off
IS	R	Points to the last byte moved into the secondary output buffer	Points to the last byte moved into the primary output buffer
UPD	R	Points to the last byte moved into the primary output buffer	Points to the last byte moved into the secondary output buffer

Output Interface

IR	R	Points to the AM preceding the next PROC statement to be executed; may be altered to change PROC execution
IS	R	May be altered as needed to alter data within the input and output buffers, but the formats described above must be maintained
UPD	R	
IB	R	

Exit Convention

The normal method of returning control to the PROC processor 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 documentation).

Note that when a PROC eventually transfers control to TCL (via the "P" operator), certain elements are expected to be in an initial condition. Therefore, if a user routine uses 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 processors. Specifically, the bits ABIT through ZBIT, AFLG through ZFLG and SB0 through SB30 are expected to be reset by the TCL-II and RECALL processors. It is best to avoid usage of these bits in PROC user exits. Also, the scan character registers SC0, SC1, and SC2 must contain a SB, a blank, and a blank, respectively.

8.4.7 RECALL Interface

Summary

It is possible to interface with the RECALL processor at several levels. A typical LIST or SORT statement passes through the Compiler and the Selection processor before entering the LIST processor. All statements must pass through the first two stages, but control can be transferred to user-written programs from that point onward.

General Conventions

The RECALL processors use a compiled string that is stored in the IS work space. String elements are separated by SM's. There is one file-defining element in each string, one element for each attribute specified in the original statement, and special elements pertaining to selection criteria, sort-keys, etc. The formats of various string elements are as follows:

File Defining Element, at ISBEG+1:

```
SM D file-name AM 0 AM conv AM corr AM
  type AM just AM SM
```

Attribute Defining Element:

```
SM c attribute-name AM amc AM conversion AM correlative AM
  type AM just AM SM

     c = A - regular attribute
         Q - D1 attribute
         B - D2 attribute
         Bx- SORT-BY, SORT-BY-DSND, etc.; "x" is from
             attribute one of the connective
```

Explicit Item-id's:

```
SM I item-id SM
```

End-of-string element:

```
SM Z
```

The Selection Processor

This performs the actual retrieval of items which pass the selection criteria, if specified. Every time an item is retrieved, the processor 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 processor passes items to the GOSORT mode, which builds up the sort-keys preparatory to sorting them. After sorting, GOSORT then retrieves the items again, in the requested sorted sequence.

A user program may get control directly from the Selection processor (or GOSORT if a sorted retrieval is required); the formats of the verbs are:

Line number	Non-sorted	Sorted
1	PB	PB
2	35	35
3	xxxx	4E
4		xxxx

where "xxxx" represents the mode-id of the user program, which is loaded into the tally MODEID2 for later use. Note: line one must be a "PB". Note that 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 Processor" 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.

Special Exit From The LIST Processor

A user program may also gain control in place of the normal LIST formatter, to perform special formatting. The advantage here is that all conversions, correlatives, etc. have been processed, and the resultant output data have been stored in the history string (HS area). The formats of the verbs then are:

Line number	Non-sorted	Sorted
1	PA	PA
2	35	35
3	4D	4E
4	xxxx	xxxx

where "xxxx" is the mode-id of the user program, which is loaded into the tally MODEID3 for later use. Note: line one must be a "PA".

Output data are stored in the HS area; data from each attribute are stored in the string, delimited by AM's; multiple values and sub-multiple-values are delimited within an element by VM's and SVM's, respectively. Since the HS may contain data other than the retrieved item, the user program should scan from HSBEG, looking for a segment preceded by an "X"; all segments except the first are preceded by a SM. The format is:

X item-id AM value one AM ... AM value n AM SM Z

The program must reset the history string pointer HSEND as items are taken out of the string. In special cases, data may not be used until, say, four items are retrieved, in which case HSEND is reset on every fourth entry only. 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.

The exit convention for the LIST processor is the same as for the Selection processor (see above).

Example: The following program is an example of one which prints item-id's (only) four at a time across the page.

The format of the RECALL verb is:

```
    LIST4
001 PA
002 35
003 4E  (sorted)  or  4D (unsorted)
004 01FF                (MODEID3 exit to frame 511, entry point zero)
```

```
          FRAME 511
*
*
*
*
*
          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,PRINTIT LAST ENTRY
          BDNZ  CTR32,RETURN  NOT YET 4 ITEMS OBTAINED
          MOV   4,CTR32       RESET
PRINTIT   MOV   HSBEG,R14
LOOP      INC   R14
          BCE   C'X',R14,STOREIT FOUND AN ITEM
          BCE   C'Z',R14,ENDHS END OF HS STRING
SCANSM    SID   R14,X'C0'     SCAN TO NEXT SM
          B     LOOP
STOREIT   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
          CMNT  *             "X" CODE
          DEC   HSEND         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 processors. Only the most important usage is described; elements that have various usages are labeled "scratch." A " " (blank) indicates that the processor does not use the element. Since the LIST processor is called by the Selection processor, any element used for "memory" purposes (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-D8
SR's          : SR20 upwards
```

SB0 and SB1 have a special connotation: they are zeroed by the Selection processor 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 processor 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 processor.

A 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 RECALL verbs.

Bits	Selection Processor	LIST Processor
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-id's 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 being tested	left-justified print limiter test
ZBIT	left-justified item-id	
SB0	unavailable	first entry flag, level one
SB1	unavailable	first entry flag, level two
SB2	reserved; zero	
SB4 through SB17	scratch or reserved	scratch or reseved
VOBIT	set for WRAPUP interface	
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	
RMBIT	set on exit if an item was retrieved; zero on final exit	
WMBIT	FUNC interface	FUNC interface
GMBIT	FUNC intrface	FUNC interface
BKBIT	scratch	scratch
DAF1	set if SCP=B, C, D, E, G, H, I, or J	
DAF8	set if accessing a dictionary	
Tallies	Selection processor	LIST processor
C1;C3-C7	scratch	scratch
C2	contents of MODEID2	
CTR1-CTR4	scratch	scratch
CTR5	scratch	AMC of the current element in the 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 sub-value counter count
CTR13	FUNC interface	current value count
CTR14	reserved	scratch
CTR15	reserved	item size
CTR16	reserved	scratch

CTR17	reserved	reserved
CTR18	reserved	scratch
CTR19	reserved	sequence no 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
Other storage	Selection processor	LIST processor
D9	count of retrieved items	
D7	FUNC interface	FUNC interface
FP1-FP5	FUNC interface	FUNC interface
RMODE	return mode-id	
SIZE	item-size	scratch
FBASE	file base, modulo,	
FMOD	and separation	
FSEP		
SR's	Selection processor	LIST Processor
S1	points to the next explicit item-id	
S2-S9	scratch	scratch
SR0	points one before the 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	GOSORT only: next sort-key	reserved
SR14-SR19	reserved	reserved
PAGHEAD	heading in the HS if HEADING was specified	generated heading in the HS
AR's	Selection Processor	LIST Processor
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
Work Space Usage	Selection Processor	LIST processor
AF	scratch	
BMS	contains the item-id	
CS		
IB		
OB		output line
IS	compiled string	
OS		scratch
HS	heading data	heading data; attribute data for special exits
TS	scratch	current value in process

Additional Notes

1. If a full-file-retrieval is specified, the additional internal elements as used by GETITM will be used. If explicit item-id's are specified, RETIX is used for retrieval of each item.

2. Most elements used by the CONV and FUNC processors have been shown in the table; both may be called either by the Selection processor or the LIST processor.

3. Since the ISTAT and SUM/STAT processors are independently driven by the Selection processor, the element usage of these processors is not shown.

8.4.8 XMODE Interface

As described before, a subroutine mode-id can be placed in XMODE before a pre-incrementing instruction such as MIID, MIIT, etc. is executed. This subroutine gains control if a forward link zero condition is reached, and can perform special processing. On return from the subroutine, the interrupted instruction will continue execution.

There are two uses for this: one is to allow the standard "FORWARD LINK ZERO" system abort message to be replaced with a more formal message; the other is to attach frames automatically when building a table or string of unknown length.

XMODE should be initialized with the mode-id of a subroutine that will handle the forward-link-zero condition. An example:

```
MOV   NEXTOVF,XMODE NEXTOVF is a standard routine
CMNT  *             that can be used to add frames
CMNT  *             to Register 6
....
....
MIID  R15,R6,X'C0'  Copy a string until R15 reaches
CMNT  *             a SM
ZERO  XMODE         Reset XMODE
```

The MIID instruction will automatically generate a subroutine call to NEXTOVF if either register reaches the end of the linked set of frames. If R15 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 on to the last frame in the linked set addressed by R6. The MIID instruction will then continue as if nothing happened.

For user-written subroutines, the Input Interface is:

System specified:

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

The example below uses Register 6; the system subroutine NEXTOVF could have been used in this case also instead of writing a local subroutine.

```
            FRAME 511
      *
      *
      *
      *
      *
            B     ENTRYO         Entry point is    01FF
            B     !TRAPSUB       Entry point is    11FF
      *
      TRAPSUB   DEFM  1,511        Define trap subroutine Mode-id
      *
      ENTRYO    EQU   *
                ...
                ...
                MOV   TRAPSUB,XMODE Initialize XMODE with Mode-id
                ...
                ...
                MIITD R15,R6,X'CO' 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 Register 6!
                STORE D4           Save accumulator, because subs
                CMNT  *            below will destroy it !
                DETZ  R6           Force detached;....
                MOV   500,R6DSP    Set displacement to 500, which is ...
                CMNT  *            last byte of this frame, so on return
                CMNT  *            will increment to 1st byte of next frame
                MOV   R6FID,RECORD Pickup FID from register
                BSL   ATTOVF       Attach another frame from overflow
                LOAD  D4           Restore accumulator
                RTN                This will return to interrupted inst.
      NOT6      ZERO  XMODE        Kill XMODE; when instruction
                CMNT  *            is re-executed, Debug will be
                CMNT  *            entered to print
                RTN   *            FORWARD LINK ZERO message
```

8.5 System Subroutines

This section describes the operating system subroutines available for use by user-written programs. As mentioned before, they are meant to be called with a BSL instruction, and return control via a RTN instruction.

8.5.1 ACONV

This routine translates one character from ASCII to EBCDIC. The high-order bit of the character is always zeroed before translation. The subroutine ECONV (documented separately) may be used to translate a character from EBCDIC to ASCII.

Input Interface

User specified:

IB	R	Points to the character to be translated

Output Interface

IB	R	Points to the converted character; location unchanged

Element Usage

None except standard

8.5.2 ATTOVF

ATTOVF is used to obtain a frame from the overflow space pool and to link it to the frame specified in double tally RECORD. 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.

Input Interface

User specified:

RECORD	D	Contains the FID of the frame to which an overflow frame is to be linked.

Output Interface

OVRFLW	D	Contains the FID of the overflow frame if obtained, or zero if no more frames are available.

Element Usage

None except standard

Subroutine Usage

Two additional levels of subroutine linkage required

8.5.3 CONV - See User Program Interfaces

8.5.4 CRLFPRINT - See PRINT

8.5.5 CVDxxx and CVXxxx Subroutines

These subroutines convert an ASCII numeric character string to its equivalent in binary. The character string starts one past the location addressed by the register used as input interface; it is terminated by any invalid character, typically a system delimiter. The converted value is stored in the accumulator FP0; the register points to the delimiter on exit. Both ASCII decimal to binary and ASCII hexadecimal to binary conversions are available. The register used as the string pointer is dependent on the exact subroutine called.

Input Interface

User specified:

Subroutine Name	Type of Conversion	Input Register
CVDIB	Decimal	IB
CVDIR	Decimal	IR
CVDIS	Decimal	IS
CVDOS	Decimal	OS
CVDR15	Decimal	R15
CVXIB	Hexadecimal	IB
CVXIR	Hexadecimal	IR
CVXIS	Hexadecimal	IS
CVXOS	Hexadecimal	OS
CVXR15	Hexadecimal	R15

Output Interface

FP0	F	Contains the converted binary number
CTR1	T	Contains the low-order two bytes of FP0, except for CVDR15 and CVXR15
NUMBIT	B	Set if string was terminated by a system delimiter or decimal point; zero if any other character
Register	R	Addresses invalid character or system delimiter or decimal point

Element Usage

None except standard

8.5.6 DATE - See TIME

8.5.7 DECINHIB

This 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. The protocol of incrementing and decrementing INHIBITH ensures that several different processors that require BREAK key inhibition may call one another without fear that INHIBITH may accidentally reach zero.

DECINHIB decrements INHIBITH if it is non-zero; if it then reaches zero, and a BREAK key had been previously activated, the Debugger is entered.

Input Interface

None

Output Interface

INHIBITH decremented as described above

Element Usage

None except standard

8.5.8 ECONV

This routine translates one character from EBCDIC to ASCII. Characters without ASCII equivalents are returned untranslated. The subroutine ACONV (documented separately) may be used to translate a character from ASCII to EBCDIC.

Input Interface

User specified:

IB	R	Points to the character to be translated

Output Interface

IB	R	Points to the converted character; location unchanged

Element Usage

None except standard

8.5.9 GETACBMS

This routine retrieves the base, modulo, and separation of the system ACC file.

Input Interface

None

Output Interface

```
BASE      D  | Contain the base, modulo, and separation
MODULO    T  | of the ACC file, if found
SEPAR     T  |
```

Element Usage

Same as GETFILE

Subroutine Usage

Up to seven additional levels of subroutine linkage required

8.5.10 GETFILE and OPENDD

GETFILE is used to set up the base, modulo, and separation parameters of a disc file from the file name. The file name is specified in a string pointed to by register IS.

OPENDD performs a similar function, but in addition the base, modulo and separation of the dictionary of the file are also returned.

GETFILE and OPENDD are the only approved methods of opening a disc file. They perform access code checking as well, and flag the file as being accessible for read-only, or for read-and-update.

GETFILE and OPENDD will exit to WRAPUP if the file cannot be successfully opened, unless bit RTNFLG is set, in which case they will still return to the calling program.

Input Interface

User specified:

IS	R	Points at least one character (any number of blanks) before "{DICT} {dictname,}filename"; the file name cannot contain embedded blanks, and must be followed by a blank, a system delimiter, or character specified in SCO
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 will still be granted unless the update access code test fails

System specified:

BMSBEG	S	Standard system buffer where the file name is to be copied; if IS points to "DICT filename", only "filename" is copied

Output Interface

BASE MODULO SEPAR	D T T	Contain the base, modulo, and separation of the file if found
FBASE FMOD FSEP	D T T	Contain (OPENDD only) the base, modulo, and separation of the file if found, otherwise unchanged
DBASE DMODULO DSEP	D T T	Contain (OPENDD only) the dictionary base, modulo, and separation, if found; if IS specifies "DICT", DBASE, DMOD, and DSEPAR = BASE, MODULO, and SEPAR

IS	R	Points to the first character after the file name
BMS	R	Points to an AM added after the copied file name
RMBIT	B	Set if the file parameters are successfully retrieved
SC2	C	Contains a blank

Element Usage

SC1 C

Plus elements used by RETIX

Subroutine Usage

Up to seven additional levels of subroutine linkage required

Exits

If RTNFLG = 0: To MD99 with message 200 if the input string is null (blank to a SM); to MD995 with message 201 if the string does not refer to a file (item not found or in incorrect format); to MD995 with message 210 if the access code test fails; or to MD99 with message 13 if the data section of a file is not found (no data pointer, or in incorrect format)

8.5.11 GETITM

This routine 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.

If the items 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-id's (per group) in a table, and 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 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).

Note the functional equivalence of the output interface elements with those of RETIX.

Input Interface

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
BASE MOD SEP	D T T	Contain the base, modulo, and separation of the file, required on first entry only

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-id's is stored; if zero, GETOVF is called to obtain space for the table

Output Interface

RMBIT SIZE R14 IR SR4	B T R R S	(See RETIX documentation)

XMODE	T	|
RECORD	D	|
SRO	S	Points one before the count field of the retrieved item

Element Usage

BASE	D	|
MODULO	T	|
SEPAR	T	|
RECORD	D	| Used by GETITM and other subroutines for
NNCF	H	| accessing file data
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 GETITM is used:

DAF1	B	| (See Input Interface)
DAF7	B	|
SBASE	D	Contains the beginning FID of the current group being processed
SMOD	T	Contains the number of groups left to be processed
SSEP	T	Contains the separation of the file
FBASE	D	| Contain the original (saved) base,
FMOD	T	| modulo, and separation of the file
FSEP	T	|
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 overlow space table if DAF1 is set; otherwise unchanged

Subroutine Usage

Four additional levels of subroutine linkage required

Error Conditions

See RETIX documentation ("Exits"); GETITM, however, continues retrieving items until no more are present even after the occurrence of errors

8.5.12 GETOVF and GETBLK

These 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. Note that the link fields of the frame(s) 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 zeroes the link fields of the single frame it returns.

These routines cannot be interrupted until processing is complete.

Input Interface

User specified:

D0	D	Contains the number of frames needed (block size), for GETBLK only

Output Interface

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

Element Usage

None except standard

Subroutine Usage

Two additional levels of subroutine linkage required

8.5.13 GLOCK, GUNLOCK, and GUNLOCK.LINE

These routines are used to ensure that disc files are not updated by more than one process at a time, and are used primarily by routine UPDITM. GLOCK sets a lock on a specified group within a file, preventing other processes from locking the group. If the group is already locked by another process, the second process "hangs" 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.

GLOCK is called at the beginning of UPDITM, before writing an item to a file, and GUNLOCK is called at the end. GLOCK is also called by RETIXU, which retrieves a disc file item and leaves the group containing the item locked (see RETIX/RETIXU).

Input Interface

User specified:

RECORD	D	Contains the beginning FID of the group to be locked (typically set by RETIX or RETIXU)

Output Interface

None

Element Usage

CH9	C	|
R2;C0	C	| Scratch
CTR1	T	|

Plus standard elements

Subroutine Usage

One additional level of subroutine linkage required

8.5.14 HASH

This routine computes the starting FID of the group in which an item in a file would be stored, given the item-id and the base, modulo, and separation of the file. Storage register BMSBEG points to the item-id, which must be terminated by an AM.

Input Interface

User specified:

BMSBEG	S	Points one byte before the beginning of the item-id
BASE	D	Contain the base, modulo, and separation of the file
MODULO	T	
SEPAR	T	

Output Interface

RECORD	D	Contains the frame number to which the item-id hashes

Element Usage

None except standard

8.5.15 HSISOS

This routine initializes the register triads for the HS, IS, and OS work spaces as described below. It does not link frames in the work spaces.

Input Interface

None

Output Interface

HS HSBEG HSEND	R S S	Point to the beginning of the HS work space (PCB+10)
IS ISBEG	R S	Point to the beginning of the IS work space (PCB+16)
ISEND	S	Points to the last data byte in the primary IS work space (3000 bytes past ISBEG)
OS OSBEG	R S	Point to the beginning of the OS work space (PCB+22)
OSEND	S	Points to the last data byte in the primary OS work space (3000 bytes past OSBEG)

Element Usage

None except standard

8.5.16 LINESUB

This routine returns the line number (PIB number) of the calling process in the accumulator.

Input Interface

None

Output Interface

D0	D	Contains the line number associated with the process

Element Usage

None except standard

8.5.17 LINK

LINK initializes the links of a set of contiguous frames (the set may be only one frame). 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).

Input Interface

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 Interface

Frames are linked contiguously backwards and forwards

Element Usage

FRMN	D	|
FRMP	D	| Scratch
NPCF	H	|

Subroutine Usage

One additional level of subroutine linkage required

8.5.18 MBDSUB, MBDNSUB, MBDSUBX, and MBDNSUBX

These routines convert a binary number to the equivalent string of decimal ASCII characters. The number is specified in the accumulator: D0 for MBDSUB and MBDNSUB, and FP0 for MBDSUBX and MBDNSUBX.

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 zeroes 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 instructions MBD (Move Binary to Decimal) and MBDN.

Input Interface

User specified:

D0	D	Contains the number to be converted (for MBDSUB and MBDNSUB only)
FP0	F	Contains the number to be converted (for MBDSUBX and MBDNSUBX only)
T4	T	For MBDNSUB and MBDNSUBX only, contains the minimum string length; leading zeroes or blanks are padded to ensure that the string is at least this length; the string may exceed this length if the value is high enough
BKBIT	B	For MBDNSUB and MBDNSUBX only, set if leading blanks are required as fill; zero if zeroes required as fill
R15	R	Points one prior to the area where the converted string is to be stored; the area must be at least eighteen bytes in length for MBDSUBX and MBDNSUBX; MBDSUB and MBDNSUB require at most eleven bytes

Output Interface

BKBIT	B	=0
R15	R	Points to the last converted character

Element Usage

None except standard

8.5.19 NEWPAGE

This routine is used to skip to a new page on the terminal or line printer and print a heading. No action is performed, however, if bit PAGINATE or tally PAGSIZE is zero. See PRNTHDR for more information on page headings and footings.

Input Interface

User specified:

None

System specified:

As for WRTLIN, except OB is first set equal to OBBEG by this routine

Output Interface

Same as for WRTLIN

Element Usage

Same as for WRTLIN

Subroutine Usage

Additional subroutine linkage required only if WRTLIN is called; see WRTLIN

8.5.20 NEXTIR and NEXTOVF

NEXTIR obtains the forward linked frame of the frame to which register IR (R6) currently points; 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 documentation). 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 (MII, MCI, MIID, MIITD, SIT, SID, etc.). Tally XMODE should be set to the mode-id of NEXTOVF before the instruction is executed by using a "MOV NEXTOVF,XMODE" instruction. If the instruction causes IR to reach an end-of-linked-frame condition (forward link zero), the system will generate a subroutine call to NEXTOVF, which will attempt to obtain and link up an available frame, and then resume execution of the interrupted instruction. Note that the "increment register by tally" instruction cannot be handled in this manner.

Input Interface

User specified:

IR	R	Points into the frame whose forward-linked frame is to be obtained (displacement unimportant)

Output Interface

IR	R	Points to the first data byte of the forward linked frame
RECORD	D	Contains the FID of the frame to which IR 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

Element Usage

Elements used by ATTOVF if a frame is obtained from the overflow space pool

Subroutine Usage

Three additional levels of subroutine linkage required

8.5.21 OPENDD - See GETFILE

8.5.22 PCRLF

PCRLF 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 verb TERM. Note that its use is inconsistent with pagination, headings, footings, etc., which are always handled correctly by WRTLIN.

Input Interface

None

Output Interface

None

Element Usage

None except standard

8.5.23 PERIPHREAD1, PERIPHREAD2, and PERIPHWRITE

These subroutines are used to read and write a string of bytes to another line's asynchronous channel, on configurations which support this feature. They are therefore analogous to the READLIN and WRTLIN subroutines, which read and write to the process' own channel only.

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 verb. 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.

PERIPHREAD2 inputs data just as READLIN does; control is returned on detecting a carriage return or line feed. PERIPHREAD1 inputs data until a byte matching that in SC0, SC1 or SC2 is found. Either subroutine will return when the number of bytes specified in IBSIZE is read. The bytes input are stored starting at the location one past IBBEG (just as in READLIN).

Input Interface - PERIPHWRITE

User specified:

OB	R	Points to the last byte to be output
T0	T	Contains the number of the affected channel

System specified:

OBBEG	S	Standard output buffer pointer

Output Interface - PERIPHWRITE

Same as WRTLIN

Input Interface - PERIPHREAD1 and PERIPHREAD2

User specified:

IBSIZE	T	Maximum number of bytes to be input
T0	T	Contains the number of the affected channel
SC0	C	(PERIPHREAD1 only); contains the
SC1	C	delimiter characters on which to stop
SC2	C	the input

System specified:

IBBEG	S	Standard system input buffer pointer

Output Interface - PERIPHREAD1 and PERIPHREAD2

Same as READLIN

Element Usage

ABIT B

CTR0 T

Plus standard elements

8.5.24 PERIPHSTATUS

PERIPHSTATUS reads the status of the specified asynchronous channel on hardware configurations where this information is available.

Input Interface

User specified:

T0 T Contains the line number whose status is to be retrieved

Output Interface

H8 H Contains the status of the line:

```
Bit# - 0    1    2    3    4    5    6    7
       |    |    |    |... No meaning ....|
       |    |    |........ Carrier Detect
       |    |............. Clear To Send
       |.................. Data Set Ready
```

Element Usage

None except standard

8.5.25 PRINT and CRLFPRINT

These routines send a message to the terminal from textual data in the calling program; CRLFPRINT precedes the text with a carriage return and line feed. These routines are not consistent with conventions regarding the line printer, and pagination. They should therefore be used only for error messages and short terminal prompts. 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.

For example:

```
BSL   PRINT            Call to subroutine
TEXT  C'Hello',X'FDFF' Message as a literal in object code
CMNT  *                Note terminating X'FF' (SM).
```

The above would print the message "Hello there", followed by a blank line. The text following the BSL to these routines may contain the following system delimiters; their meaning is explained below:

Delimiter	Action
SM (X'FF')	End of message; CR/LF printed, and return to calling program
AM (X'FE')	
VM (X'FD')	CR/LF printed, and message processing continues to next character
SVM (X'FC')	End of message; return without printing CR/LF

Input Interface

User specified:

Message text MUST follow BSL instruction

Output Interface

None

Element Usage

None except standard

8.5.26 PRNTHDR and NEWPAGE

These are entry points into the system routine for pagination and heading control of output (also used by WRTLIN and WRITOB when pagination is specified). PRNTHDR must be called once to intialize 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 character(s) 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 4 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; this is described in an example at the end of this chapter.

Input Interface

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

PRNTHDR NEWPAGE Interface continued

Output Interface, Element Usage and Subroutine Usage

Same as for WRTLIN; 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

8.5.27 RDLINK and WTLINK

These routines read or write the link fields from or to a frame, to or from the tallies NNCF, FRMN, FRMP, and NPCF. The FID of the frame is specified in RECORD.

Input Interface

User specified:

RECORD	D	Contains the FID of the frame whose links are to read or written.

User specified for WTLINK; Output Interface from RDLINK:

NNCF	H	Number of next contiguous frames
FRMN	D	Forward link
FRMP	D	Backward link field
NPCF	H	Number of previous contiguous frames

Element Usage

None except standard

8.5.28 RDREC

RDREC is used to set up the IR register to the beginning of the frame defined by the double tally RECORD. The subroutine assumes the frame has the linked format, and therefore IR is set pointing to the eleventh byte of the frame, that is, one 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.

Input Interface

User specified:

RECORD	D	Contains the FID required

Output Interface as described above

Element Usage

None except standard

8.5.29 READLIN, READLINX, and READIB

READLIN, READLINX and READIB are the standard terminal input routines. 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 a 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 will be added at the end of the line.

If READLIN or READLINX is used, a trailing carriage return/line feed sequence is transmitted to the terminal; if READIB is used, it is not.

The entry READLIN also provides 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 the last character in a stacked line is a "<", it is replaced by a SM. This is for processors such as TCL and EDIT that allow for continuation lines, and is equivalent to a control-_ (underscore or back-arrow) input directly from the terminal as a continuation character. Terminal input resumes when the stacked input is exhausted. READLINX does not test for stacked input.

Editing: All three routines perform terminal editing as follows:

Character input	Action
Control-H	Backspace input; echo a backspace-space-backspace unless BSPCH = 0.
Character in BSPCH	As above.
Control-W	Backspace word, to last non-numeric, non-alpha.
Control-X	Cancel line; echo cr/lf or backspaces (see FRMTFLG)
Carriage return or line feed	Terminate input and return control.

READLIN and READIB also perform input tabulation as specified by the TCL verb TABS, 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.

Input Interface

User specified:

CCDEL	B	If set, control characters are deleted; this bit is normally zero
FRMTFLG	B	If set, entering a control-X emits backspaces instead of a cr/lf, to preserve screen forat; this bit is normally zero
PRMPC	C	Terminal prompt character

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 140
LFDLY	T	Contains (in the low-order byte) the number of idle characters to be output after a carriage return/line feed; set by the TCL TERM verb
BSPCH	C	Contains the character to be echoed to the terminal when the back space key is typed, or is zero if no character is to be echoed; set by the TCL TERM verb
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
STKINP	S	Points to the next "stacked" input line; lines are deliminated by AM's, with a SM indicating the end of the stack
ITABFLG	T	Set for input tab stops by the TCL TABS verb

Output Interface

IB	R	=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
STKINP	S	Points to the next line of stacked input (or end of stack) if stacked input is being processed

Element Usage

None except standard

Subroutine Usage

Two additional levels of subroutine linkage required

Error Conditions

If a stacked input line exceeds IBSIZE, the line is truncated at IBSIZE; the remainder of the line is lost.

8.5.30 RELBLK, RELCHN, and RELOVF

These routines 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). A call to RELCHN specifies the first FID of a linked set of frames; the routine will release all frames in the chain until a zero forward link is encountered.

Input Interface

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)
DO	D	Contains the number of frames (block size) to be released, for RELBLK only

Output Interface

None

Element Usage

None except standard

8.5.31 RESETTERM

This routine is used to initialize terminal and line printer characteristics. RESETTERM is called from WRAPUP before reentering TCL.

Input Interface

System specified:

OBSIZE	T	Contains the size of the output (OB) buffer
OBBEG	S	Points to the start of the OB buffer

Output Interface

TOBSIZE	T	|
TPAGSIZE	T	|
POBSIZE	T	| Initialized to default values, as set up
PPAGSIZE	T	| by the TCL verb TERM
PAGSKIP	T	|
LFDLY	T	|
BSPCH	C	|
CCDEL	B	|
FRMTFLG	B	|
STKFLG	B	|
PAGINATE	B	|
NOBLNK	B	|
LPBIT	B	| =0
TPAGNUM	T	|
TLINCTR	T	|
PPAGNUM	T	|
PLINCTR	T	|
PAGNUM	T	|
LINCTR	T	|
PAGHEAD	S	| Contain zero in the frame field
PAGFOOT	S	|
OB	R	=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.

Element Usage

None except standard

8.5.32 RETIX and RETIXU

These 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 are the file parameters base, modulo, and separation.

The routine performs a "hashing" algorithm to determine the group (see HASH documentation). The group is then searched sequentially for a matching item-id. If the routine finds a match, it returns pointers to the beginning and end of the item, and the item size (from the item count field). If entry 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 item-id is specified in the system-standard buffer defined by storage register BMSBEG; it must be terminated with an AM.

Input Interface

User specified:

BMSBEG	S	Points one byte before the item-id
BASE	D	Contain the base, modulo, and separation of the file to be searched
MODULO	T	
SEPAR	T	

Output Interface

BMS	R	Point to the last character of the item-id
BMSEND	S	
RECORD	D	Contains the beginning FID of the group to which the item-id hashes (set by HASH)
NNCF	H	Contain the link fields of the frame specified in RECORD; set by RDREC
FRMN	D	
FRMP	D	
NPCF	H	
XMODE	T	=0

		Item Found:	Item Not Found:
RMBIT	B	=1	=0
SIZE	T	=item size (one less than value of count field)	=0
R14	R	Points one prior to the item count field	Points to the SM after the last item in the group

IR	R	Points to the first AM of the item	Points to the SM indicating end of group data (=R14+1)
SR4	S	Points to the last AM of the item	=IR

Element Usage

None except standard

Subroutine Usage

RDREC, HASH, GLOCK (RETIXU only)

Three additional levels of subroutine linkage required

Exits

If the data in the group are bad - premature end of linked frames, or non-hexadecimal character encountered in the count field - the message

GFE HANDLER INVOKED - RECORD/GFE x/f.d

is returned, where "x" is the starting FID of the group to which the item hashes, and "f.d" is the approximate frame and displacement where the error was detected. RETIX and RETIXU then return with an "item not found" condition under these circumstances. Data are not destroyed, and the group format error will remain.

8.5.33 SETLPTR and SETTERM

These routines are used to set output characteristics such as line width, page depth, etc. to the previously-specified values for either the terminal or the line printer. In addition, the current line number and page number are saved so that when switching from terminal to line printer output, say, and then switching back, pagination will continue automatically from the previous values.

Input Interface

User specified:

None

System specified:

LINCTR	T		Contains the current line number
PAGNUM	T		Contains the current page number
TPAGSIZE	T	|	Contains the number of printable lines
or PPAGSIZE	T	|	per page for the terminal or line printer
TOBSIZE	T	|	Contains the size of the output (OB)
or POBSIZE	T	|	buffer for the terminal or line printer
TLINCTR	T	|	Contains the current line number for the
or PLINCTR	T	|	terminal or lineprinter
TPAGNUM	T	|	Contains the current page number for the
or PPAGNUM	T	|	terminal or line printer

Note: TPAGSIZE, TOBSIZE, TLINCTR, and TPAGNUM are required only by SETTERM; PPAGSIZE, POBSIZE, PLINCTR, and PPAGNUM are required only by SETLPTR

Output Interface

LPBIT	B		Reset by SETTERM; set by SETLPTR
PAGSIZE	T	|	
OBSIZE	T	|	Set to the appropriate characteristics
LINCTR	T	|	for terminal or line printer output
PAGNUM	T	|	
TLINCTR	T	|	=LINCTR; TLINCTR set by SETLPTR, PLINCTR
or PLINCTR	T	|	set by SETTERM
OBEND	S		=OBBEG+OBSIZE

The area from the location addressed by OBBEG to that pointed to by OBEND is filled with blanks.

Element Usage

None except standard

8.5.34 SLEEP and SLEEPSUB

These routines 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 may be specified.

Input Interface

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 Interface

None

Element Usage

None except standard

8.5.35 SORT

This routine sorts an arbitrarily long string of keys in ascending sequence only; the calling program must complement the keys if a descending sort is required. The keys are separated by SM's when presented to SORT; they are returned separated by SB's. Any character, including system delimiters other than the SM (X'FF'), SB (X'FB'), X'F0' and X'F1', which have a special meaning, 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 the characters X'F0' and X'F1' are present, which have a special meaning: X'F0' indicates the start of a numeric string, terminated by a SVM; X'F1' indicates the start of a numeric string, terminated by a SVM, that is to be compared negatively (for example, this may be set up by the 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, the sort keys "ABC/100X" and "ABC/9Y" may be presented to the SORT subroutine as:

ABC/[F0]100[FC]X and ABC/[F0]9[FC]Y

which will result in correct sequencing of "ABC/9Y" before "ABC/100X."

A six-way polyphase sort-merge sorting algorithm is used. 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.

Alternately, the entire set of buffers may be in the IS or OS workspace if they are large enough.

Input Interface

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 Interface

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 SB's) is marked by a SM

Element Usage

SB1	B	|
SC2	C	|
XMODE	T	|
IS	R	|
OS	R	|
BMS	R	|
TS	R	| Utility
CS	R	|
S1 thru	S	|
S9	S	|

Elements used by ATTOVF

BMS and AF workspaces

8.5.36 TIME, DATE, and TIMDATE

These routines return the system time and/or the system date, and store it in the buffer area specified by register R15. 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

Input Interface

User specified:

R15	R	Points one prior to the buffer area

Output Interface

R15	R	Points to the last byte of the data stored

Element Usage

D2	D	| Used by TIME and TIMDATE only
D3	D	|

Subroutine Usage

TIME used by TIMDATE; MBDSUB used by TIME

Two additional levels of subroutine linkage required by TIMDATE, one level required by TIME, none by DATE

8.5.37 TPBCK

This routine backspaces the tape one physical record, or block. The tape must be attached to the process via the TCL T-ATT verb.

No input or output interface; no element usage except standard

RDLABEL WTLABEL

8.5.38 TPRDLBL, TPRDLBL1, TPWTLBL, TPWTLBL1, and TPGETLBL

These routines are used to read and write standard ULTIMATE tape labels.

The tape I/O routines TPREAD and TPWRITE are capable of processing both labeled and unlabeled tapes. By default, if a process begins tape operation via TPREAD or TPWRITE without first calling one of the tape label routines, the tape is considered unlabeled. If the tape operation spans multiple reels of tape before entering WRAPUP, each reel will be unlabeled.

For labeled tapes, the label is the first physical record, or block, on the tape. In the case of multi-file tapes, such as FILE-SAVE tapes, each tape file may be preceded by a label, which would follow immediately after the EOF after the preceding tape file. (Each account on a FILE-SAVE tape is a separate tape file.)

To read a labeled tape, TPRDLBL or TPRDLBL1 should be called once at the beginning of processing to read the label and to set the labeled-tape flag for the tape routines. After that, if tape operation spans multiple reels, TPREAD will make sure that each reel has the same label and that the reels are numbered consecutively.

TPRDLBL1 makes sure that the current reel is reel number one. TPRDLBL accepts any reel number (but still forces subsequent reels to be numbered consecutively from this point). If the tape record read by these routines is not a recognizable label, the tape is considered unlabeled. The tape is then backspaced so that the next call to TPREAD will read the first record as a data record. TPRDLBL or TPRDLBL1 may therefore be called if it is not known whether a tape is labeled or not.

To write a labeled tape, TPWTLBL or TPWTLBL1 should be called once at the beginning of processing to write the label and to set the labeled-tape flag for the tape routines. After that, if tape operation spans multiple reels, TPWRITE will make sure that each reel has the same label and that the reels are numbered consecutively.

TPWTLBL1 sets the current reel number to one. TPWTLBL does not change the current reel number. Note that at the beginning of tape operation, the reel number will be zero, which is an invalid reel number - TPREAD would consider the tape unlabeled.

TPGETLBL returns the status of the labeled-tape flag, and if it is set, returns the label data.

ULTIMATE tape labels are 80-byte records having the following format:

```
_L bksz hh:mm:ss  dd mmm yyyy ... variable label data ... ^rr
```

where _ = Segment Mark
^ = Attribute Mark
bksz = block size (four hex digits), preceded by a blank
hh:mm:ss dd mmm yyyy = time and date, preceded by a blank
variable label data = string passed to TPWTLBL or TPWTLBL1, truncated or padded with trailing blanks if necessary, and preceded by a blank
rr = reel number (two hex digits)

Input Interface

User specified:

IS	R	For TPWTLBL and TPWTLBL1 only, points one byte before the beginning of a text string to be included in the label; the string may be up to 47 bytes in length and must be terminated by a SM; if the first byte is a SM, no label will be written

Output Interface

RMBIT	B	For TPGETLBL only, set if a labeled tape is being processed, otherwise zero
R15	R	For TPGETLBL only, points to the initial SM (not one before) of the 80-byte tape label in the tape routine label buffer if RMBIT is set

Element Usage

D3	D	Scratch

Plus standard elements

Subroutine Usage

Up to seven additional levels of subroutine linkage required

8.5.39 TPREAD, TPWRITE, and TPRDBLK

TPREAD reads a specified number of bytes from the tape into a buffer area pointed to by R15 at entry to the routine.

TPWRITE writes a specified number of bytes from the area pointed to by R15 to the tape.

TPRDBLK reads one physical tape record, or block, into an internal tape buffer (see below), and returns a pointer to the data along with the number of bytes read.

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 will be reset on each call.

Bit EOFBIT will be 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, then TPWRITE will fill the rest of the tape buffer with the character pointed to by R15, which will cause 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 will exit to WRAPUP with an error message. The TCL T-ATT verb 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, TPRDLB1, TPWTLBL, and TPWTLBL1 may be used to read and write the labels. See the documentation on these routines for more information.

Input Interface

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 Interface

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

Element Usage

The tape handler will stack and restore most of the elements which it uses. The following elements are modified, however:

T5	T	Scratch
T6	T	
T7	T	
D2	D	
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
OVRFLW	D	Used by routine GETBLK
RECORD	D	
FRMN	D	
FRMP	D	
NNCF	H	
NPCF	H	

Subroutine Usage

TPREAD and TPWRITE use an extensive set of internal subroutines in such a way that element usage is transparent outside of the above set. Both may go to seven levels of subroutine usage if either encounters a parity error while handling a label on the second and following reels in a set of tapes. TPRDBLK, which calls TPREAD, may require eight levels.

8.5.40 TPREW

This routine is used to rewind the tape. The tape must be attached to the process via the TCL T-ATT verb.

No input or output interface; no element usage except standard

8.5.41 TPWEOF

This routine is used to write a tape mark on the tape. The tape must be attached to the process via the TCL T-ATT verb.

No input or output interface; no element usage except standard

8.5.42 UPDITM

UPDITM performs updates to a disc file defined by its base FID, modulo, and separation. If the item is to be deleted, the routines compress 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-and-add" function takes place.

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.

Input Interface

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	R	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
BASE	D	Contain the base, modulo, and separation of the file being updated
MODULO	T	
SEPAR	T	

Output Interface

None

Element Usage

NNCF	H	|
NPCF	H	|
XMODE	T	|
D3	D	|
D4	D	|
RECORD	D	| Scratch
FRMN	D	|
FRMP	D	|
IR	R	|
BMS	R	|
UPD	R	|

Subroutine Usage

Four additional levels of subroutine linkage required

Error Conditions

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.

8.5.43 WRTLIN and WRITOB

These are the standard routines for outputting data to the terminal or line printer. Entry WRTLIN deletes trailing blanks from the data, and increments the internal line counter LINCTR; WRITOB does neither.

The data to be output are pointed to by OBBEG, and continue through the address pointed to by OB. This is for convenience in calling the subroutines; normally they are called just after data have been transferred to the OB buffer, in which case the OB address register will be on the last byte copied. On return, the OB AR is set back to OBBEG, again for convenience; the output buffer area is blanked unless bit NOBLNK is set. The data are transmitted to the terminal if bit LPBIT is off, otherwise they are stored in the printer spooling area.

Pagination and page heading and page footing routines are automatic, 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 documentation for PRNTHDR.

If PAGINATE is set, the end of page is checked for, and action is taken automatically to print the page footing (if it exists), to skip to the next page, and to 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).

A value of zero in PAGSIZE suppresses pagination, however, regardless of the seting of PAGINATE.

WRTLIN and WRITOB also perform output tabulation as specified by the TCL verb TABS, 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.

Input Interface

User specified:

OB	R	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

System specified:

OBBEG	S	Standard system buffer used to store data for terminal or line printer output

LISTFLAG	B	If set, all output to the terminal is suppressed; set and reset by the TCL verb P, 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 verb TERM
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 verb TABS)

The following specifications are meaningful only if PAGINATE is set:

User specified:

PAGHEAD	S	Points one byte before the beginning of the page heading message; 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; if the FID field of this storage register is zero, no footing is printed; this is the default condition

System specified:

PAGSIZE	T	Contains the number of printable lines per page; set by the TCL verb TERM
PAGSKIP	T	Contains the number of lines to be skipped at the bottom of each page; set by the TCL verb TERM
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 control-X will return the process directly to TCL; normally set, this bit is zeroed by the "N" option at TCL for most verbs, or by the NOPAGE connective in 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 Interface

OB	R	=OBBEG

Element Usage

T4	T	|
T5	T	|
D2	D	|
D3	D	| Scratch
SYSR0	S	|
SYSR1	S	|
SYSR2	S	|
BMS	R	|
ATTOVF		| Used if LPBIT set
elements		

8.5.44 WSINIT

This routine initializes the following process workspace pointer triads: BMS, BMSBEG, BMSEND; CS, CSBEG, CSEND; AF, AFBEG, AFEND; TS, TSBEG, TSEND; IB, IBBEG, IBEND; OB, OBBEG, OBEND; also PBUFBEG and PBUFEND. In each case, 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.

WORK SPACE	SIZE (BYTES)
BMSBEG-BMSEND	50
AFBEG-AFEND	50
CSBEG-CSEND	100
IBBEG-IBEND	Contents of IBSIZE; max. 140
OBBEG-OBEND	Contents of OBSIZE; max. 140
TSBEG-TSEND	511
PBUFBEG-PBUFEND	2000 (4 linked frames)

Input Interface

None

Output Interface

Registers are set up as described above. 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 140 if initially greater.

Element Usage

None except standard

8.5.45 WTLINK - See RDLINK

8.6 Example of a Simple TCL-I Verb

Here is an example of a simple routine, called via a TCL-I verb, that performs the equivalent of the BASIC 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)
REPEAT
```

```
          FRAME 511
*
*
*     This program is called via a verb of the form:
*         COPYIT
*     001 P
*     002 01FF
*
*    At TCL, enter   COPYIT  {(P)}
*
          B     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 there was 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 to output buffer, through SM
          DEC   OB           Backoff SM to setup interface to WRTLIN
          BSL   WRTLIN       PRINT
          CMNT  *            Note there was no initialization for above
          MOV   IBBEG,IB     Set back to one before first byte
          BSL   CVDIB        Convert numeric from IB to binary in accumulator
          BZ    T0,LOOP      If zero or non-numeric, nothing to do
          BLE   T0,140,OK    This test needed to ensure number < 140
          LOAD  140          Else setup to limit to 140 bytes
OK        MOV   OB,R15       OBBEG=OB=R15 now (WRTLIN resets OB)
          MCI   C'+',OB      Store first '+', pre-incrementing OB
          DEC   T0           Adjust for above move
*MAP:
*                +
* OBBEG & R15...^^......OB
*
          MIIT  R15,OB       Propagate '+' as many times as value in T0
          CMNT  *            Note that R15 always pre-increments to a '+'
          CMNT  *            and that OB always is one ahead of R15.
          BSL   WRTLIN       PRINT
          B     LOOP         REPEAT
STOP      ENT   MD999        Conventional return to TCL via WRAPUP
```

8.7 Example of a Simple TCL-II Verb

This is an example of a routine called via a TCL-II verb that strips comments from BASIC source file item(s). The stripped source is written back to the same file, with an item-id of "STRIP-" concatenated with the original item-id.

```
      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
      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
```

```
          FRAME 511
*
*
*     This program is called via a verb of the form:
*         STRIPIT
*     001 P
*     002 2          .... TCL-II verb
*     003 01FF
*     004 0
*     005 CU         .... Copy item to IS buffer; verb may update file
*   At TCL, enter:   STRIPIT filename {itemlist}
*   This routine will be called once as each item is read from file.
*
          B     ENTRY0       Entry point is 01FF
STRIPX    EQU   *-1          Note the '*-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
ITMEND    EQU   *            End of item body reached
          MCI   SM,OS        Interface to UPDITM; end of new item body
          MCC   UCHAR,CH8    Interface to UPDITM; update flag
          MOV   OSBEG,TS     Interface to UPDITM; start of new item body
          BSL   UPDITM       WRITE
          ENT   MD999        Rtn via WRAPUP to TCL-II for next item, if any
```

8.8 Example of a User Conversion Subroutine

Here is an example of a conversion subroutine that 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; since 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')      "    "     "    "
*
          B     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
          INC   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
```

8.9 Example Using Heading and Footing

This example shows how to set up a heading and footing area, using the HS buffer.

```
          FRAME 511
*
*
*   This is an example of setting up a heading and footing
*   It may be added to the program shown in the first example
*
*
*
*
          B     ENTRYO        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  *             Note FC P=page#; FD=newline; FF=end of data
          TEXT  X'FD',C'ULTIMATE ASSEMBLY MANUAL'
          TEXT  X'FF'         To stop MIID !
*---
ENTRYO    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 one before heading
          MIID  R14,R15,X'CO' 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 one before heading
          MIID  R14,R15,X'CO' 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       Intialize and print first heading
          *
LOOP      BSL   READLIN       INPUT x
          etc.
```

8.10 Example of a PROC User Exit

Here 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 RECALL Conversion. The PROC exit format is:

General Format	Examples	
Uxxxx	U01FF	U01FF
x conversion.code	;D2/	:TINV;C;;2

where "x" is a ":" for Output Conversion (similar to OCONV) and is a ";" for Input Conversion (similar to ICONV). The parameter is taken from the current Input Buffer Pointer (IB), which is assumed for simplicity to be the last parameter in the buffer.

```
          FRAME 511
*
*
*
*
*
          B     ENTRY0       U01FF
*
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 for : here!)
EP10      INC   IR           Set on first byte of conversion code
          XRR   IR,IS        Conversion processor 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, thanks
          MOV   SR20,TSBEG   Restore
          ZB    MBIT         Later processors may expect it zero
          ENT   1,PROC-I     Return to PROC
```

CHAPTER 9

LIST OF ASCII CODES

Note: characters shown under notes in the form "cx" represent control character using the "x" key, e.g., cA is a control-A.

ASCII hex	dec	EBCDIC hex	ASCII character	Notes	ASCII hex	dec	EBCDIC hex	ASCII character	Notes
00	0	00	NUL	c@	30	48	F0	0	
01	1	01	SOH	cA	31	49	F1	1	
02	2	02	STX	cB	32	50	F2	2	
03	3	03	ETX	cC	33	51	F3	3	
04	4	37	EOT	cD	34	52	F4	4	
05	5	2D	ENQ	cE	35	53	F5	5	
06	6	2E	ACK	cF	36	54	F6	6	
07	7	2F	BEL	cG	37	55	F7	7	bell
08	8	16	BS	cH	38	56	F8	8	backspace
09	9	05	HT	cI	39	57	F9	9	horiz. tab
0A	10	25	LF	cJ	3A	58	7A	:	linefeed
0B	11	0B	VT	cK	3B	59	5E	;	
0C	12	0C	FF	cL	3C	60	4C	<	formfeed
0D	13	0D	CR	cM	3D	61	7E	=	cr/newline
0E	14	0E	SO	cN	3E	62	6E	>	
0F	15	0F	SI	cO	3F	63	6F	?	
10	16	10	DLE	cP	40	64	7C	@	
11	17	11	DC1	cQ	41	65	C1	A	X-ON
12	18	12	DC2	cR	42	66	C2	B	retype
13	19	3A	DC3	cS	43	67	C3	C	X-OFF
14	20	3C	DC4	cT	44	68	C4	D	
15	21	3D	NAK	cU	45	69	C5	E	
16	22	32	SYN	cV	46	70	C6	F	
17	23	26	ETB	cW	47	71	C7	G	back word
18	24	18	CAN	cX	48	72	C8	H	cancel
19	25	19	EM	cY	49	73	C9	I	
1A	26	3F	SUB	cZ	4A	74	D1	J	
1B	27	27	ESC	c[	4B	75	D2	K	
1C	28	1C	FS	c\	4C	76	D3	L	
1D	29	1D	GS	c]	4D	77	D4	M	
1E	30	1E	RS	c^	4E	78	D5	N	
1F	31	1F	US	c_	4F	79	D6	O	
20	32	40	blank		50	80	D7	P	
21	33	5A	!		51	81	D8	Q	
22	34	7F	"		52	82	D9	R	
23	35	7B	#		53	83	E2	S	
24	36	5B	$		54	84	E3	T	
25	37	6C	%		55	85	E4	U	
26	38	50	&		56	86	E5	V	
27	39	7D	'		57	87	E6	W	
28	40	4D	(		58	88	E7	X	
29	41	5D	)		59	89	E8	Y	
2A	42	5C	*		5A	90	E9	Z	
2B	43	4E	+		5B	91	80	[	
2C	44	6B	,		5C	92	E0	\	
2D	45	60	-		5D	93	90	]	
2E	46	4B	.		5E	94	5F	^	
2F	47	61	/		5F	95	6D	_	

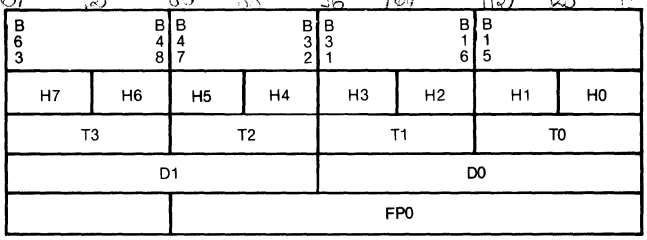
ASCII hex	dec	EBCDIC hex	ASCII character	Notes
60	96	79	`	
61	97	81	a	
62	98	82	b	
63	99	83	c	
64	100	84	d	
65	101	85	e	
66	102	86	f	
67	103	87	g	
68	104	88	h	
69	105	89	i	
6A	106	91	j	
6B	107	92	k	
6C	108	93	l	
6D	109	94	m	
6E	110	95	n	
6F	111	96	o	
70	112	97	p	
71	113	98	q	
72	114	99	r	
73	115	A2	s	
74	116	A3	t	
75	117	A4	u	
76	118	A5	v	
77	119	A6	w	
78	120	A7	x	
79	121	A8	y	
7A	122	A9	z	
7B	123	C0	{	
7C	124	6A	|	
7D	125	D0	}	
7E	126	A1	~	
7F	127	07	DEL	

80 128 through FA 250 Undefined characters

System Delimiters

ASCII hex	dec	EBCDIC hex	ASCII character	Notes
FB	251		SB	Start Buffer Display: [
FC	252		SVM	Sub-value Mark Display: \ Entered via READLIN: c\
FD	253		VM	Value Mark Display:] Entered via READLIN: c]
FE	254		AM	Attribute Mark Display: ^ Entered via READLIN: c^
FF	255		SM	Segment Mark Display: _ Entered via READLIN: c_

ACCUMULATOR OVERLAY DESCRIPTION

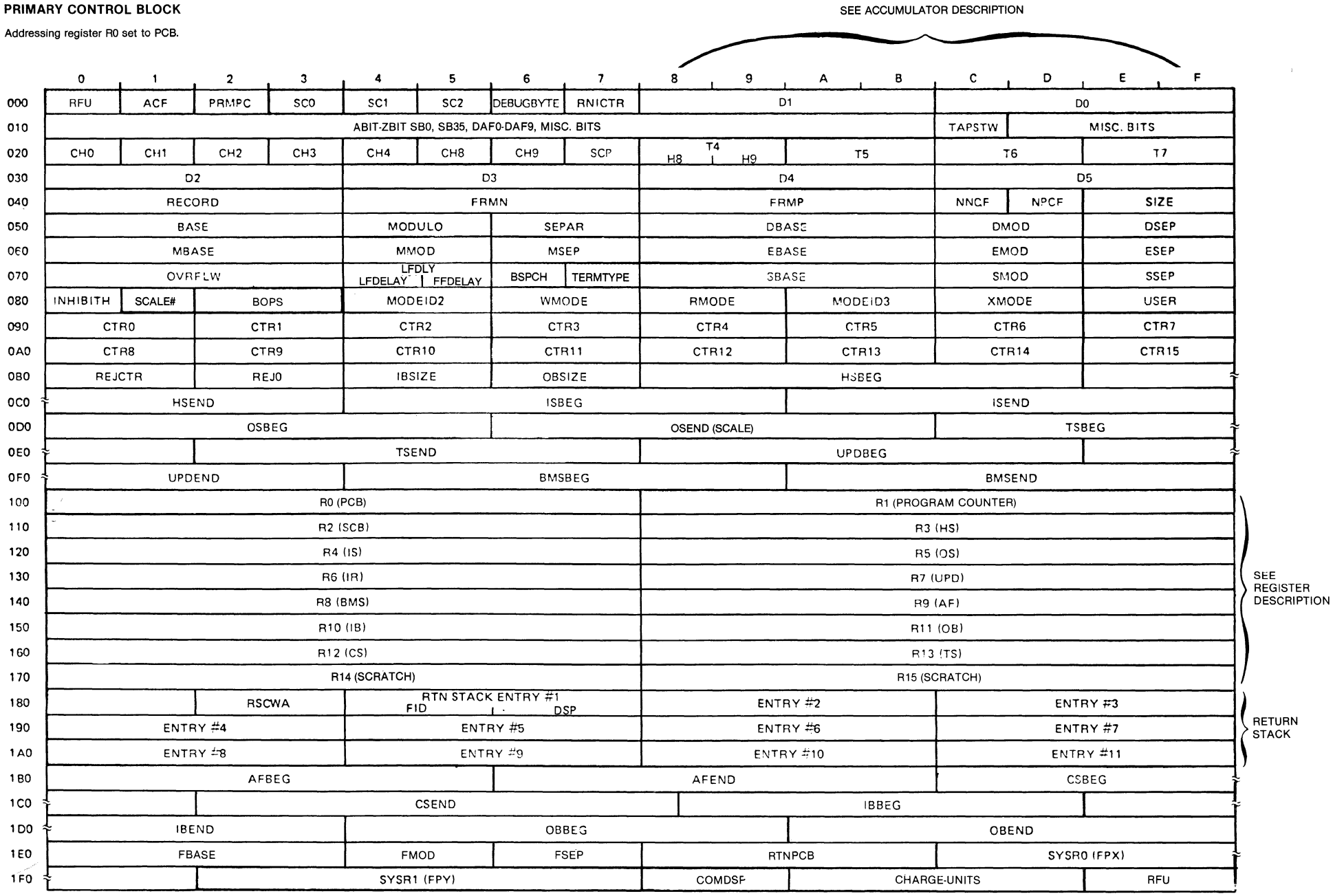


REGISTER FORMAT DESCRIPTION

RFU	Rx DSP	Rx FID

PRIMARY CONTROL BLOCK

Addressing register R0 set to PCB.



Primary Control Block

SECONDARY CONTROL BLOCK

Addressing register R2 set to SCB. SCB = PCB + 1.

	0 1	2 3	4 5	6 7	8 9	A B	C D	E F
000	(SCRATCH) BSPCH	C1	C2	C3	C4	C5	C6	C7
010	AFLG-ZFLG, NUMFLG1, NUMFLG2, MISC. BITS		CTR16	CTR17	CTR18	CTR19	CTR20	CTR21
020	CTR22	CTR23	CTR24	CTR25	CTR26	CTR27	CTR28	CTR29
030	CTR30	CTR31	CTR32	CTR33	CTR34	CTR35	CTR36	CTR37
040	CTR38	CTR39	CTR40	CTR41	CTR42	FP5		
050	PFILE	YMODE	FP1			FP2		
060	FP3			D6		D7		D8
070		D9		REJ1	REJ2	FP4		
080	SYSR2			NXTITM			S0	
090		S1			S2			
0A0	S3		S4			S5		
0B0	S6			S7			S8	
0C0		S9			SR0			
0D0	SR1		SR2			SR3		
0E0	SR4			SR5			SR6	
0F0		SR7			SR8			
100	SR9		SR10			SR11		
110	SR12			SR13			SR14	
120		SR15			SR16			
130	SR17		SR18			SR19		
140	SR20			SR21			SR22	
150		SR23			SR24			
160	SR25		SR26			SR27		
170	SR28			SR29			PQBEG	
180		PQCUR			PQEND			
190	STKEND		STKBEG			SR35		
1A0	LOCKSR			QCBSR			BDESCTBL	
1B0		IOFID		TPRECL	SPSAVE			PAGPRINT
1C0	TPSIZE	FOOTCTR	PAGFOOT			PBUFBEG		
1D0	PBUF			PBUFEND			OVRFLCTR	
1E0	POBSIZE	PPAGSIZE	PLINCTR	PPAGNUM	TOBSIZE	TPAGSIZE	TLINCTR	TPAGNUM
1F0	PAGNUM	PAGHEAD			LINCTR	PAGSIZE	PAGSKIP	LFCTR

Secondary Control Block

NOTES

NOTES

ULT
THE ULTIMATE CORP.
77 BRANT AVENUE, CLARK, NEW JERSEY 07066
(201) 388-8800 TWX 710-996-5862 Telecopier (201) 388-1377