

PREVIEW



Notice

The information contained in this document is subject to change without notice.

HEWLETT-PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

Hewlett-Packard assumes no responsibility for the use or reliability of its software on equipment that is not furnished by Hewlett-Packard.

This document contains proprietary information which is protected by copyright. All rights are reserved. No part of this document may be photocopied, reproduced or translated to another program language without the prior written consent of Hewlett-Packard Company.

Hewlett-Packard assumes no responsibility for the use or reliability of its software on equipment that is not furnished by Hewlett-Packard.

This document contains proprietary information which is protected by copyright. All rights are reserved. No part of this document may be photocopied, reproduced or translated to another program language without the prior written consent of Hewlett-Packard Company.

Copyright 1978 by Hewlett-Packard Company

Hewlett-Packard Company
5303 Stevens Creek Blvd., Santa Clara, California, U.S.A.

Part No. 31000-90004
Printed in USA May 1978

	Page
<u>1. General Overview</u>	
Resident Operations	1-2
Disc Operations	1-2
<u>2. Job Execution</u>	
The Job	2-1
Program Hierarchy	2-1
Program's Code and Data	2-2
Segmentation	2-2
Segment Tables	2-3
ID Translation Table	2-3
Program Environment	2-4
The Need for Tasks	2-4
Task Definition	2-5
Task Control Block	2-6
Procedure Calls	2-7
Procedure Exits	2-8
Data Space Allocations	2-9
Large Arrays	2-10
Data Manipulation	2-11
Review No. 1	2-13
System Tasks	2-14
The Current Task	2-15
Scheduling Tasks	2-16
Task Priority Schemes	2-17
Scheduling Options	2-19
Review No. 2	2-20
<u>3. Disc Storage</u>	
Permanent Data Storage	3-1
Basic File Transfer Mechanism	3-2
Basic File Management	3-2
Disc Organization	3-3
Record Blocking	3-3
File Information in Memory	3-4
Example of File Operations	3-5
Primitive Access	3-6
Storage of Programs	3-7
Large Programs	3-8
workspace Concept	3-9
Program Loading	3-11
Task Start	3-13
Calling the User's Task	3-14
Swapping Data Segments	3-16
Swapping File Blocks	3-18
Review No. 3	3-20



4. <u>Hardware Operation</u>	
Operating Environment	4-1
Central Processor Registers	4-1
Instruction Formats	4-3
Memory Reference	4-3
Stack and Control Instructions	4-4
Shifts, Bit Tests, Conditional Branches	4-5
Immediate	4-5
Linkage and Control Instructions	4-5
Special Opcodes	4-5
Move Opcodes	4-5
Mini Opcodes	4-5
Addressing Conventions	4-6
Virtual Memory	4-9
Code Segmentation	4-9
Data Segmentation	4-14
Formats	4-17
Data Formats	4-17
Stack Marker Format	4-22
Tables	4-23
System Code Segment Table	4-24
User Code Segment Table	4-24
User Data Segment Table	4-24
System Data Segment Table	4-24
Task Control Block	4-24
Device Reference Table	4-26
Interrupt Stack Base and Limit	4-27
Interrupt Mask	4-27
I/O System	4-27
General I/O Channel	4-28
Program Interpretive Channel	4-28
SIO Programming	4-28
Interrupt and Trap Processing	4-29
External Interrupts	4-29
Interrupt Control Stack	4-31
Internal Interrupts and Traps	4-32
Special System Functions	4-39
Power-Fail/Power-On	4-39
Cold-Load	4-40
System Dump	4-40
System Clock	4-40
System Halt	4-40
5. <u>Central Processor Instructions</u>	
Memory Address Computation	5-2
Load Instructions	5-4
Store Instructions	5-8
Memory Reference Integer Instructions	5-9
Branch Instructions	5-10
Conditional Branches	5-11
Integer Instructions	5-14
Logical Instructions	5-16
Double Integer Instructions	5-17

Floating Point Instructions	5-19
Extended Floating Point Instructions	5-21
Boolean Instructions	5-23
Test Instructions	5-24
Zero Instructions	5-25
Duplicate and Delete Instructions	5-25
Exchange Instructions	5-26
Index Transfer Instructions	5-28
Index Arithmetic Instructions	5-28
Control Instruction	5-29
Shift Instructions	5-29
Single Word Shift Instructions	5-30
Double Word Shift Instructions	5-31
Four Word Shift Instructions	5-32
Bit Instructions	5-33
Field Instructions	5-34
Immediate Instructions	5-34
Immediate Index Instructions	5-36
Program Control Instructions	5-37
Move Instructions	5-44
I/O and Interrupt Instructions	5-49
Register Control Instructions	5-55
Special Control Instructions	5-58
Semaphore Instruction	5-60
List Search Instruction	5-61
Label Instructions	5-61
Extended Instruction Set	5-65
Arithmetic Decimal Instructions	5-68
Conversion Decimal Instructions	5-69
Special Decimal Instructions	5-72
Decimal Edit Instructions	5-74

Appendix

<u>A. Alphabetical Listing of Instructions</u>	<u>A-1</u>
---	------------

Index

n.a.

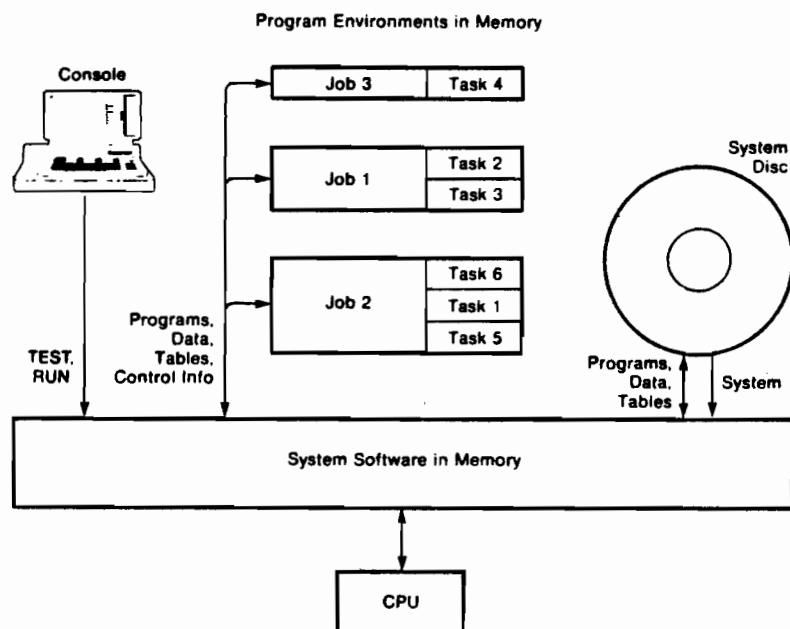


Section 1 General Overview

This manual presents a simplified introduction to the internals of the HP 300 Computer System. It is intended to be followed up with intensive classroom training to explain in greater depth the operations introduced here.

To lend coherence to the presentation, the manual is organized so as to follow the general flow of work through the system--from job initiation to job termination--rather than to give separate "theory-of-operations" of the various modules of the system.

The following diagram will be used as a reference for the description of the overall plan, which immediately follows. The layout of this diagram may be compared with the layout of the foldout diagram at the end of Section 3.



HP Computer Museum
www.hpmuseum.net

For research and education purposes only.

Resident Operations

Section 2 of this manual, entitled "Job Execution", introduces the internal workings of the system as if it were resident in memory at all times.

That section shows that a "job" is created by a RUN or TEST command from the console. A job consists of the total "program environment", including the program code, data, various tables, and one or more "tasks".

Thus it will be shown that not only can there be many jobs in the system (multiprogramming), there can also be multiple tasks within a job (multitasking). The basic method of scheduling tasks for execution will also be shown.

Both code and data will be seen to exist in "segments". Code within a code segment is organized as a number of "procedures". Data may be found in a "global data segment", "local data segments", or a "stack segment"; the first is global to all tasks in the program, and the latter two are local to each task.

Disc Operations

Section 3, entitled "Disc Storage", introduces the real world of operation with the system disc.

The discussions show that all information on the disc (with two minor exceptions) is stored in "files", and in "blocks". Blocks are subdivisions of files, and fill (or become) segments when loaded into memory. The complete collection of special files that make up a given program is a "workspace".

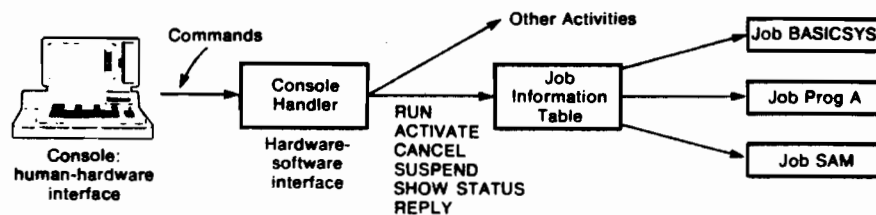
Specific operations covered in this section include: program loading, starting a task, array access, and record access. Segment "swapping", the fundamental operation of "virtual memory", is shown in connection with array and record access.

A review covering all operations described in this manual, given in a sequence of 43 steps, begins on page 3-20.

Section 2 Job Execution



The Job



A program launched into execution runs as a "job".

The job is the major unit of work in the system.

Subsystems (such as BASIC) run as jobs, just like user programs.

Many jobs can run concurrently.

A job run without a specified job name assumes the program name for the job.

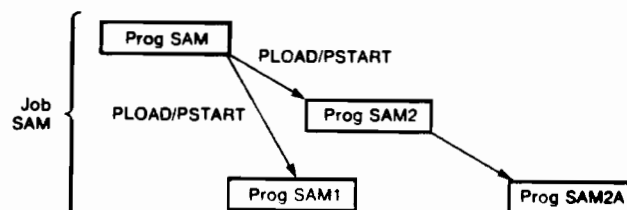
One of the functions of the Console Handler is to maintain a list of current jobs in a Job Information Table (JIT).

Program Hierarchy

A job is the execution of one or more programs under one job name.

Subordinate programs are launched programmatically by PLOAD and PSTART calls.

The Console Handler, however, is aware of only one program--the one that was launched with the RUN command.



For now, we'll concentrate on just the top-level program.

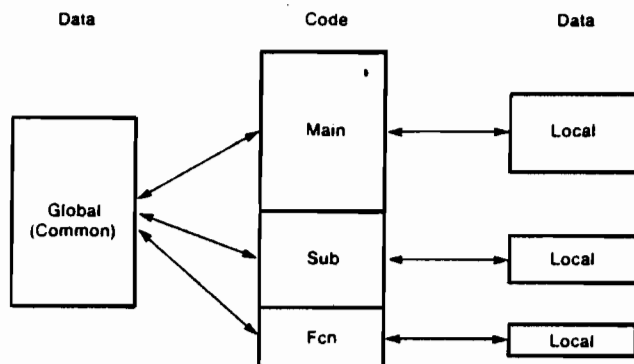
Program's Code and Data

Three major elements of a program's environment are: code, data, files.

Files will be considered under a later major heading.

A program's code usually includes: main program, one or more subprograms, and (in Basic) one or more function routines.

A program's data usually includes: global data, local data.



Separation of code and data permits an architectural rule that code cannot be modified during execution. This allows code to be re-entrant.

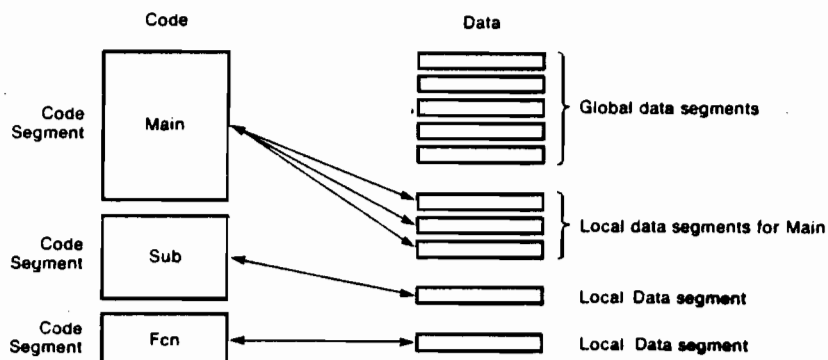
Segmentation

Large programs and data areas are clumsy to manage.

The HP 300 can segment both code and data for efficient memory management.

Code is segmented on natural code boundaries (e.g., subprogram).

Data is segmented at a fixed, system-determined number of bytes.



Segments need not be contiguous in memory, though to the user an entire data array is logically contiguous.

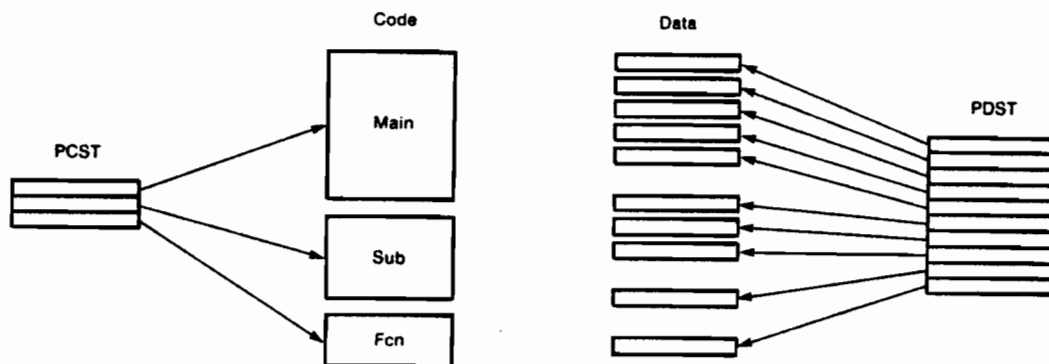
Segment Tables

If segments are not contiguous in memory, how does the system keep track of where all the segments are for a particular program?

For each program there is a:

Program Code Segment Table (PCST), and a
Program Data Segment Table (PDST).

These give absolute memory addresses for each of the program's segments.

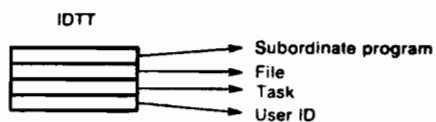


ID Translation Table

Besides code and data segments, the system needs to keep track of miscellaneous elements that are created dynamically by the program as it runs.

This is done by a set of identification numbers (ID numbers) that are assigned and released as needed.

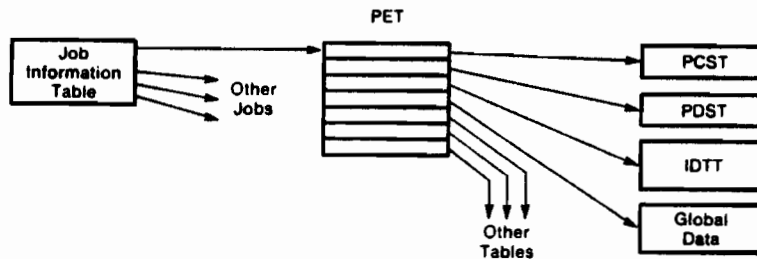
Each ID number is an index into an ID Translation Table (IDTT) which points into other table-structures in order to locate: subordinate programs (see page 2-1), files (page 3-4), tasks (next subject, and "user ID's").



Program Environment

Thus, so far, there are at least 3 major tables defining the program's environment: PCST, PDST, IDTT.

The location of these three tables, and the location of the global data, are made known to the system for job execution (see page 2-1) by a Program Environment Table (PET).



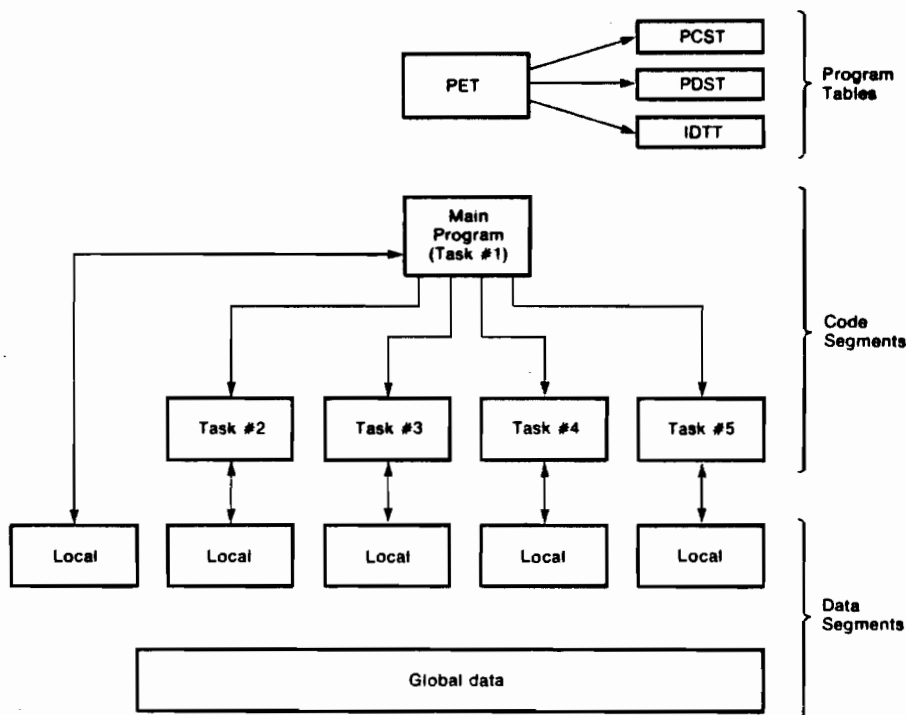
The Need for Tasks

The foregoing structure exists for every job. If concurrent execution were possible only at the job level and many concurrent jobs are needed, many such structures would be needed. System overhead would be high, response poor.

Therefore the HP 300 allows a main program and its subprograms to execute concurrently as separate "tasks".

All tasks within a job share one: PET, PCST, PDST, IDTT, global.

Each task has its own local data storage.



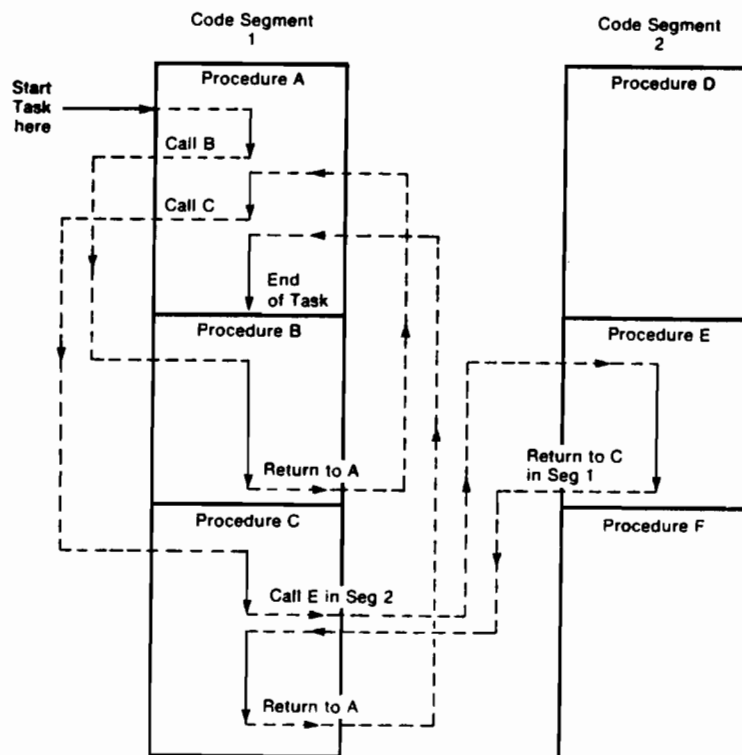
But note: It would be too restrictive to say that a task is the execution of a subprogram. Subprograms must be able to call other subprograms, whether in the same code segment or some other code segment. Also, since code is re-entrant, two or more tasks should be able to concurrently execute the same subprogram.

Thus a task is not a subprogram, not is it a code segment. What, then, is a task?

Task Definition

A task is an execution path (data dependent) through a series of procedures. [Note: since we are now down to the machine level, we will use "procedure" instead of "subprogram"; these terms are equivalent.]

The task starts executing at the start of a specified procedure. Procedure calls in the code cause branches to, and returns from, other procedures--both within and between segments. Branching decisions are based on data processed by the code.



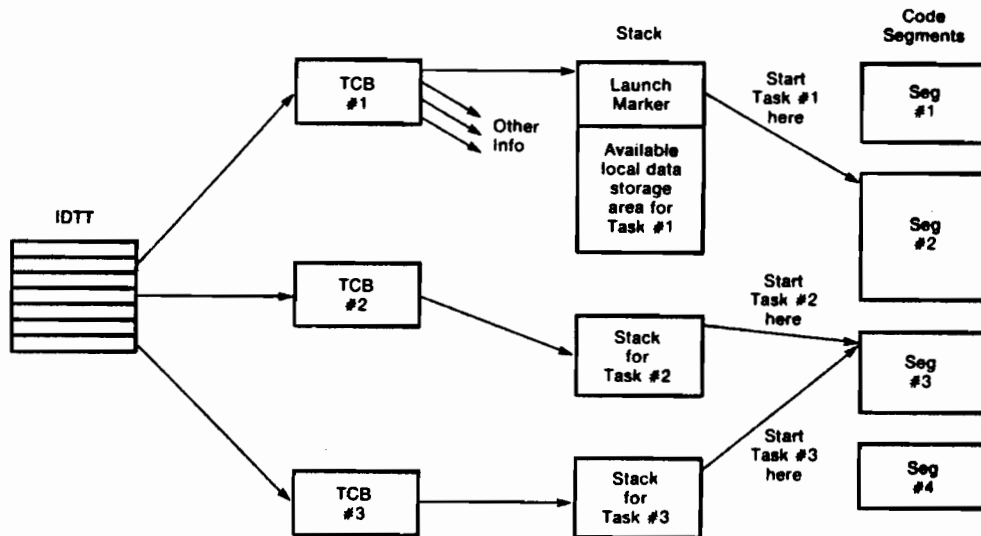
Task Control Block

The two main things to specify for a task before it is started:

- Location of local data area (a data segment, called the task's "stack").
- Starting point for code execution.

The location of the stack is specified in a Task Control Block (TCB), of which there is one for every task. [The TCB also contains other information relating to task execution, to be covered later.]

The starting point is specified in a reserved space of the stack called the "launch marker". Details concerning task start are given in Section 3 (page 3-13).



Each task has an ID number, maintained in the IDTT (see page 2-3). TCB's are dynamic elements, assigned and used as needed while the program is in execution.

Knowing the ID number, the system can locate the TCB via IDTT.

In this example, Task #1 will begin its execution in code segment #2. Tasks #2 and #3 will begin in code segment #3. Execution paths may run in parallel, criss-cross, jump from one code segment to another, etc.

How does each task keep track of its procedure calls?

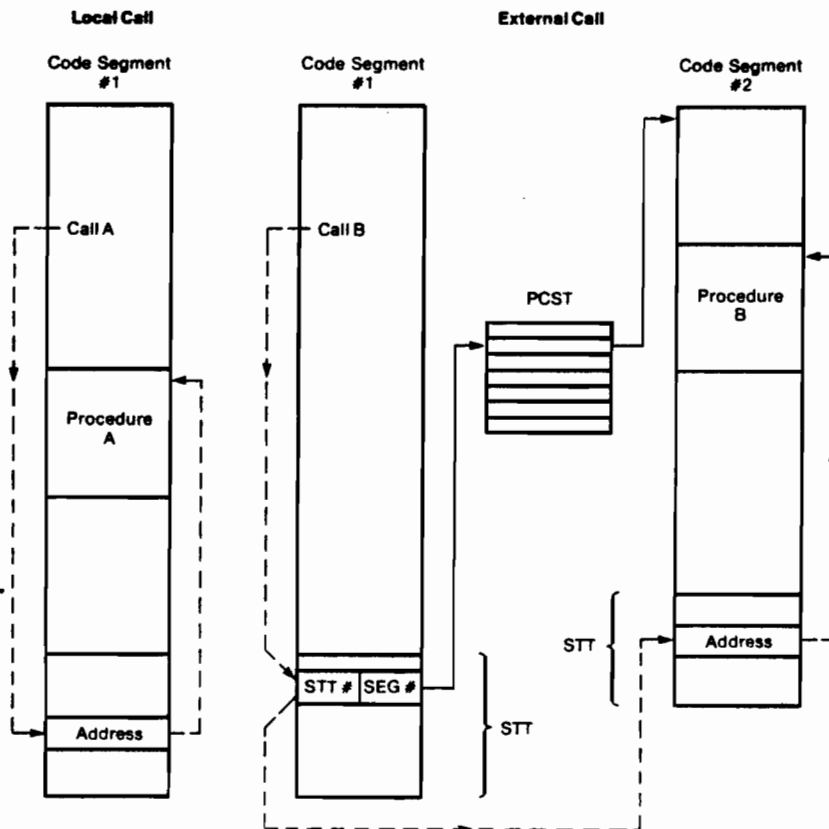
Procedure Calls

At the end of each code segment is a Segment Transfer Table (STT).

The STT contains local labels, pointing to all procedures within the segment, and external labels, pointing to procedures in other segments.

For a local call, control passes to the address specified in the STT entry.

For an external call, the code segment specified in the STT entry is made the current (active) segment. Then a second STT number, given in the external label, is used to access an entry in the called segment's STT; control is passed to the address specified in that entry.



Procedure Exits

Return information for each procedure call is automatically placed on the task's data stack as a "stack marker".

The marker is placed at the time the call is made, and secondarily serves to delimit or "cap" the data which is local to the calling procedure.

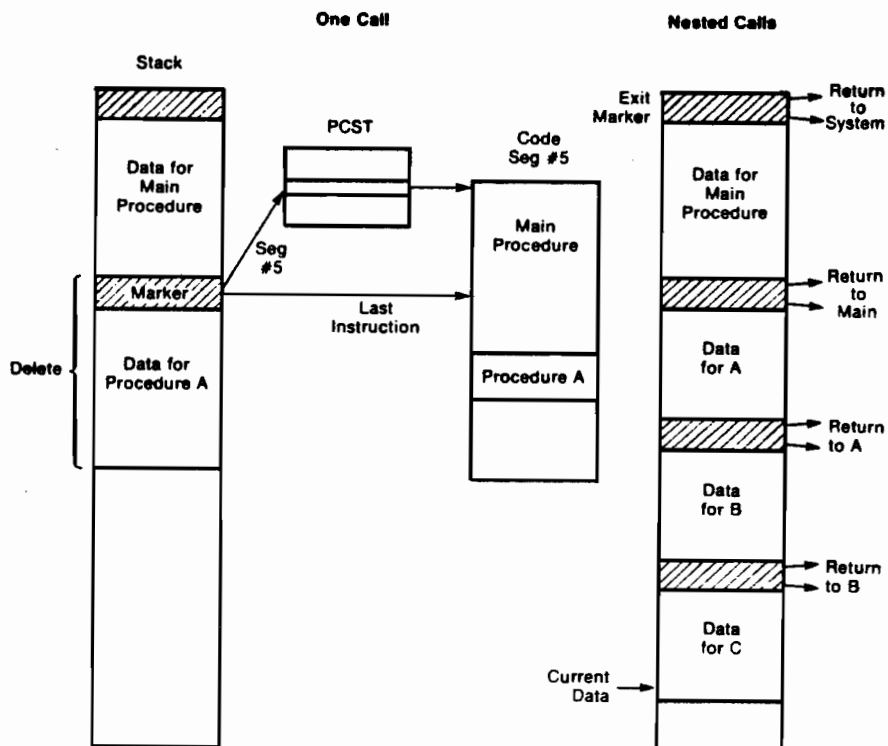
The return information includes (among other things):

- a. Segment number of the calling procedure.
- b. Location of the next instruction to be executed.

When a procedure exit is executed, return from the called to the calling procedure is accomplished by making the specified segment (a) the active code segment, pointing to the next instruction (b), and deleting the stack data that was local to the called procedure (along with the stack marker).

Thus it does not matter whether the call was local or external; the return information always includes segment number. (The PCST is accessed to get the segment's address.)

For nested calls, there will be a series of local data areas, one for each called procedure, separated by stack markers giving return information into the code segments for each call.



Data Space Allocations

The stack diagrams on the preceding page showed two kinds of information on the stack: markers (to remember return points in code), and data areas for procedures.

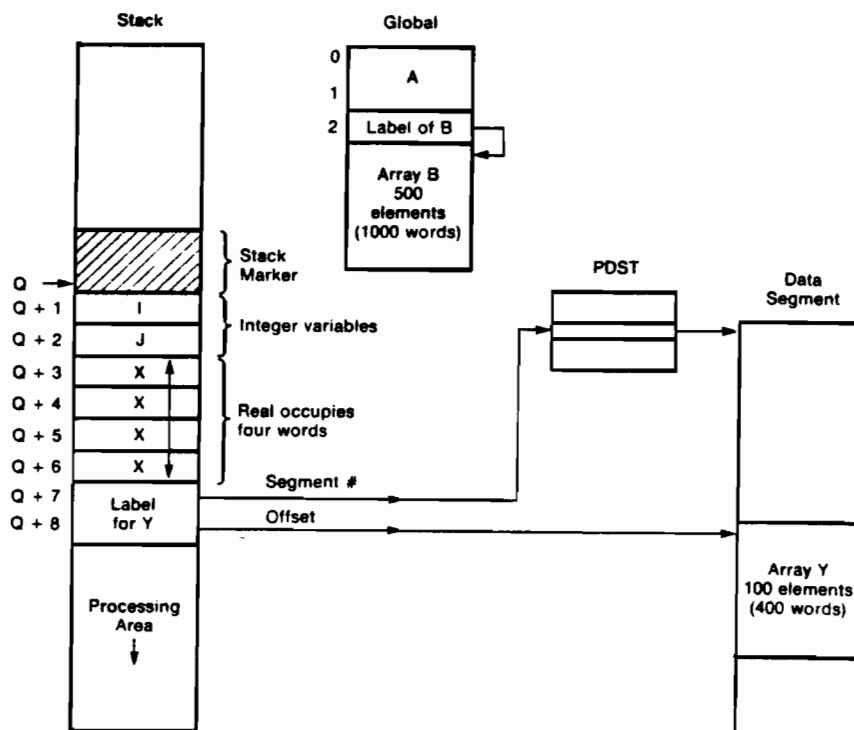
We will now take a closer look at how data space is allocated for a procedure.

In the compilation process:

<pre> 5 COM SHORT A,B(500) 10 INTEGER I,J 20 REAL X 30 DIM Y(100) 40 I = 1 50 A = I+1000 60 END </pre>	↓ This program.....translates into ↓ In global, reserve 2 words for A, 1000 for B. On stack, reserve 1 word each for I,J. On stack, reserve 4 words for X. In data segment, reserve 400 words for Y. Code this as a procedure (see page 2-11)
---	--

Assume a maximum data segment size of 2K words (actually varies from about 1K to 16K, depending on system memory size). In this case, the arrays B and Y are small enough to fit within one segment. B, being declared as COM, is assigned to the global data segment; Y requires a separate data segment to be created to contain the array, and a data label (PDST pointer and offset value) to be created and loaded on the stack.

The ending location of the stack marker is known as "Q", and marks the beginning of data allocations local to the current procedure.

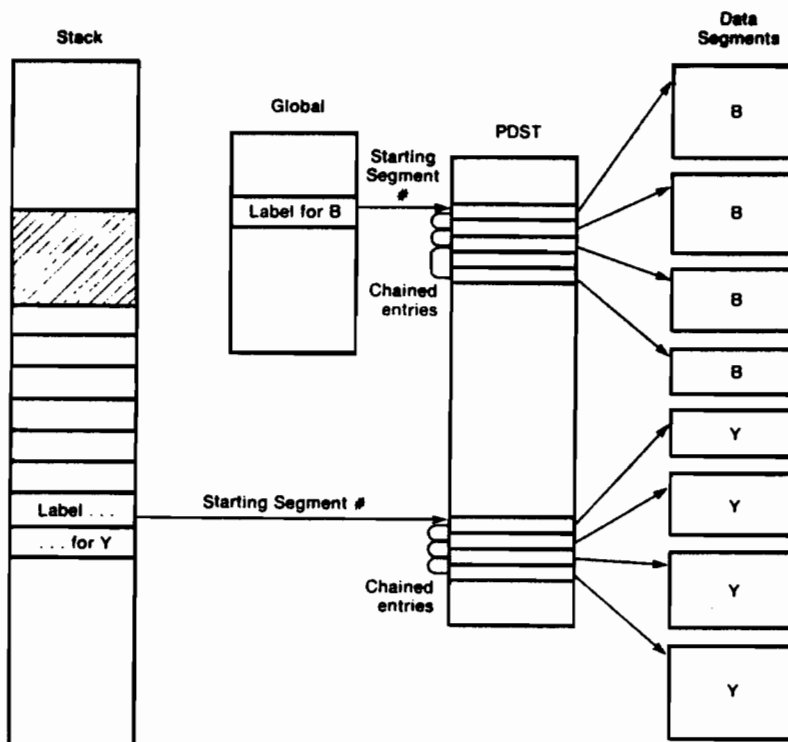


Large Arrays

What if the arrays do not fit into the maximum data segment size?

For a local array: a chained sequence of PDST entries, each pointing to a segment which contains part of the array.

For a global array, a label which points to a chained sequence of PDST entries, each pointing to a segment which contains part of the array.



The chained entries are not necessarily consecutive in the PDST.

Data Manipulation

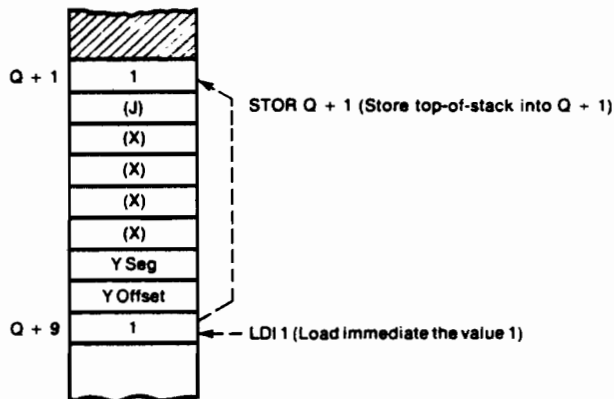
It was shown (page 2-9) where various kinds of storage space are allocated for a simple program. How are those spaces filled with actual data during procedure execution?

In machine language:

This procedure..... Translates into this code.

<pre> I = 1 A = I+1000 END </pre>	<pre> { LDI 1 STOR Q+1 LOAD Q+1 LOAD P+2 BR P+2 1000 ADD,FLT STD DB+0 EXIT 0 } </pre>
-------------------------------------	---

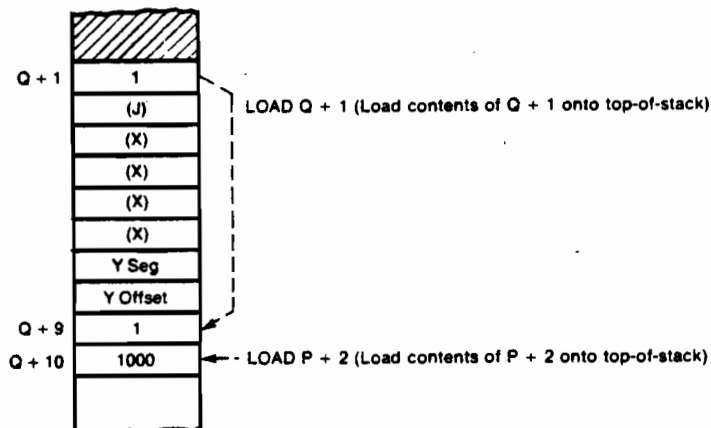
The last used stack location (see page 2-9) was Q+8; processing begins at Q+9. Remember that Q+1 was reserved for I.



The LDI instruction, as emitted by the compiler, loads the value "1" onto the top of the stack.

The STOR instruction takes whatever is on the top of the stack and stores it, as specified, into Q+1. Thus I now has a value, the integer "1". The content of Q+9 is no longer needed, so it is deleted (i.e., Q+9 will be re-used).

Now, to execute $A=I+1000$, we need to get the value of I and the value 1000 into two consecutive locations on the top of the stack, so they can be processed.



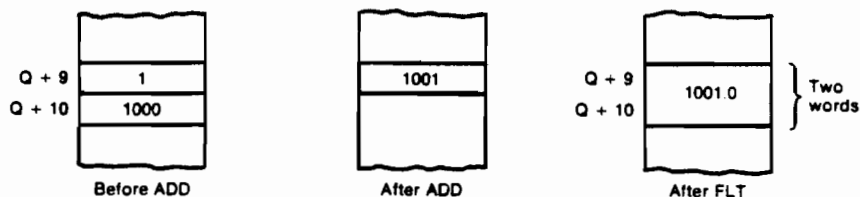
The LCAD Q+1 instruction loads the assigned value of I (now in Q+1) into Q+9, which is then the top-of-stack.

The LCAD P+2 loads a constant from the program code (see sixth line of code on preceding page) into Q+10, which then becomes the top-of-stack. The BR P+2 (branch) instruction (seventh line) merely skips over this constant.

The second step, after loading these two values onto the stack, is to add them together. ADD does this.

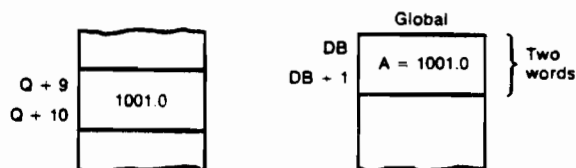
Both values are integers. However, A is a SHORT real number (see first line of Basic program on page 2-9). The FLT (float) instruction converts the integer result to real.

Since the operands are no longer needed, they are deleted in the process, leaving the answer in Q+9 and Q+10.



The third step is to put this value into the space allocated for A . This was assigned to be the first two words of global.

Since the start of a global segment is known as DB (data base), a STD DB+0 will accomplish this step. (EXIT 0 returns control to the system.)

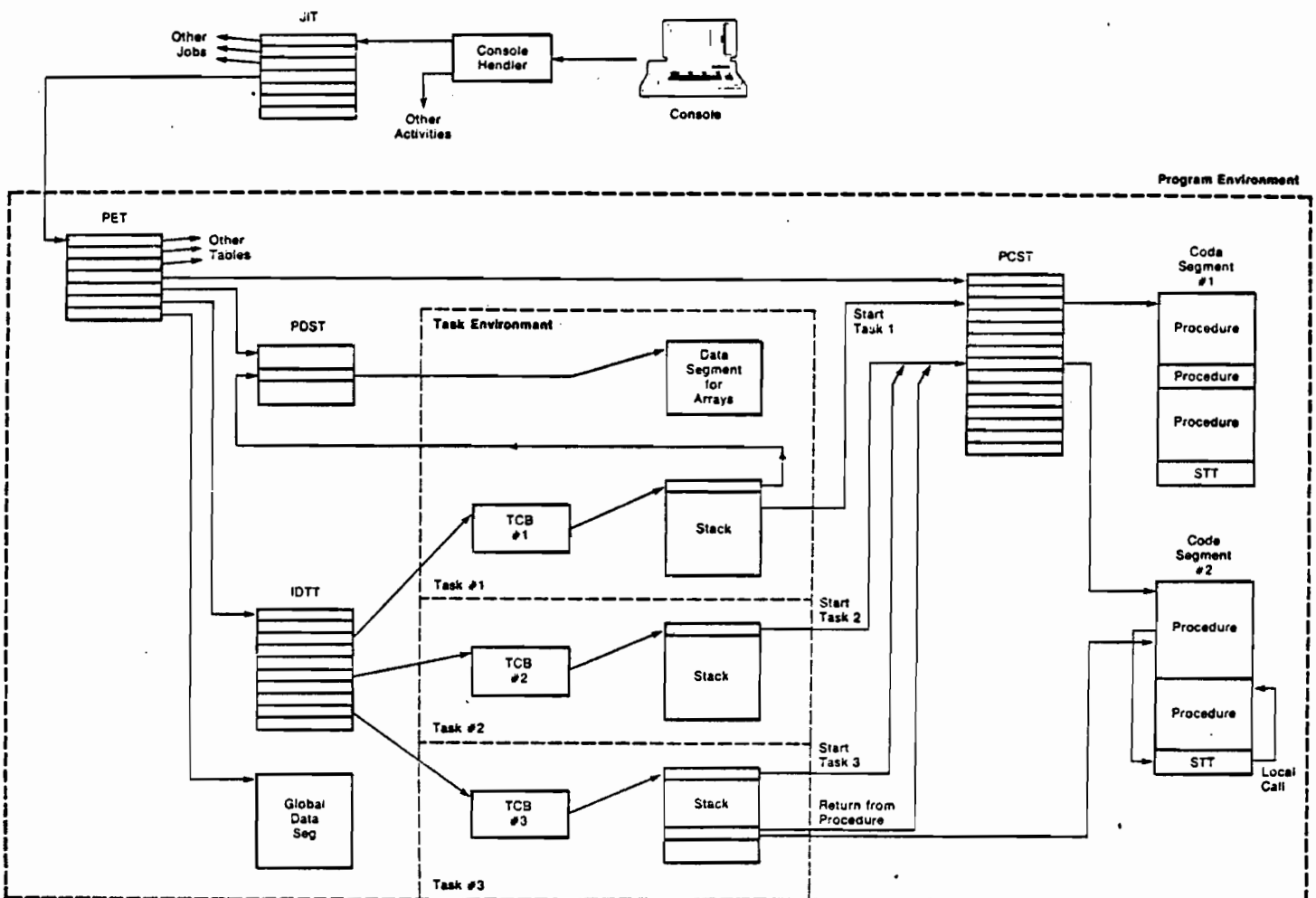


Review No. 1

The diagram below combines most elements discussed so far. Not shown (for simplicity): external procedure calls (see page 2-7), chained PDST entries for large arrays (page 2-10).

Note particularly:

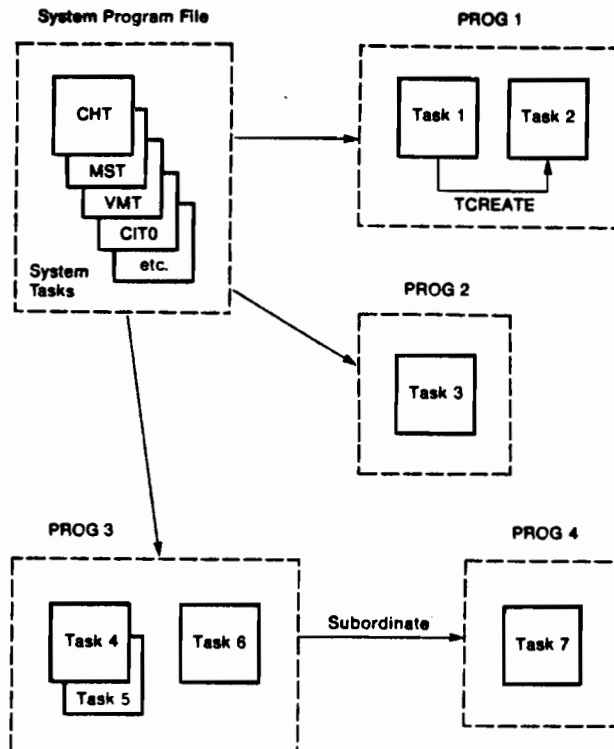
1. The task environment includes a Task Control Block, a stack, and possibly one or more local data segments for arrays.
2. The program environment includes all the program's task environments, plus the following, which are sharable by tasks:
Tables (PCST, PDST, IDTT).
Global data segment.
Code segments.
3. The task is the only executable entity; it executes procedures in code segments.



System Tasks

Like user programs, the system program (the System Program File) runs as a set of tasks.

Although programs can be subordinate to others (e.g., all are subordinate to the system program), tasks compete with one another for execution on the cpu independently of the program hierarchy.



Although tasks are said to execute "concurrently", CPU time is actually multiplexed among them ("apparent parallelism").

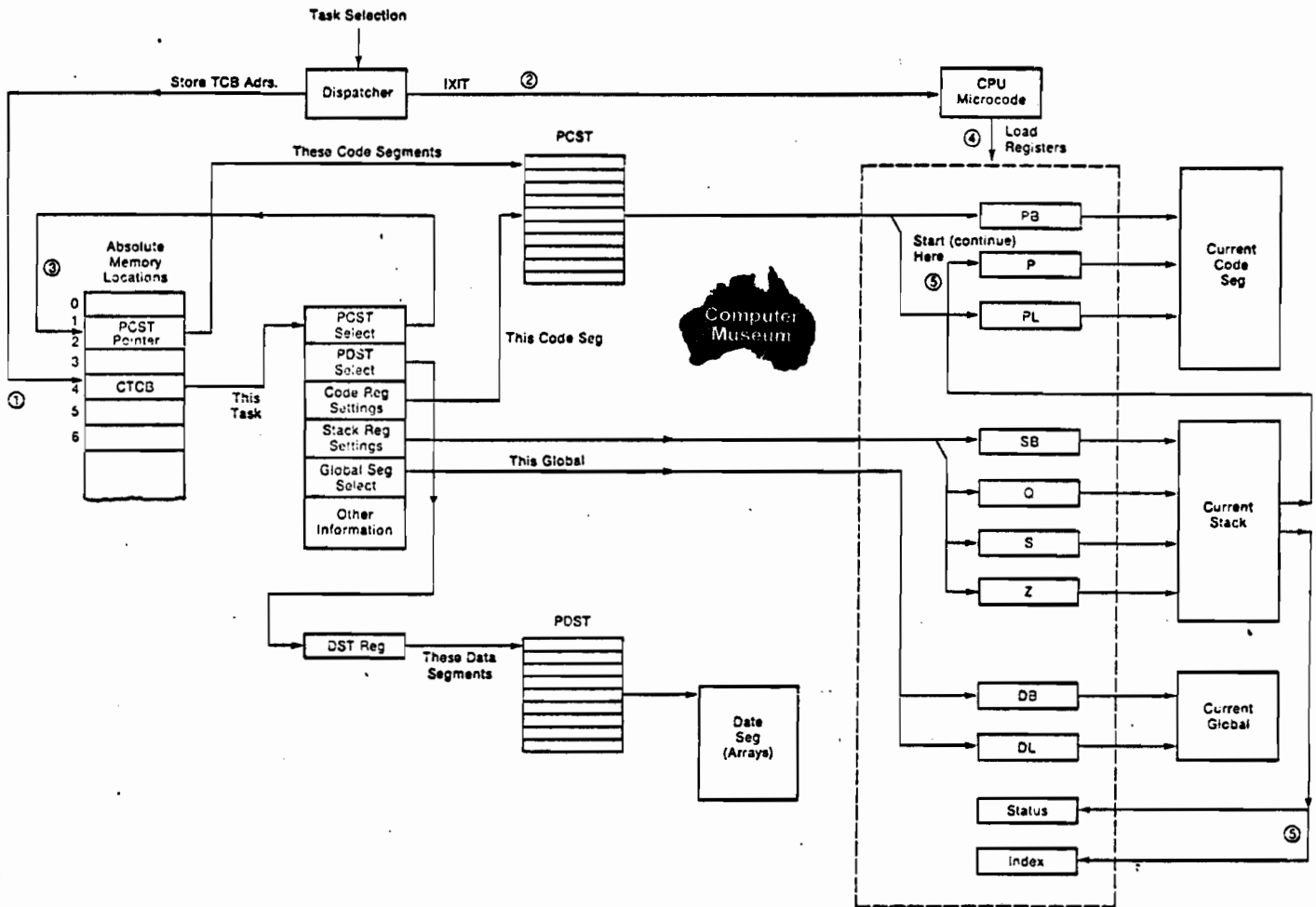
What determines that a particular task is physically executing its code?

Only two main operations need to be done:

- a. Set Absolute Memory. The absolute memory address for pointing at the "current" TCB must be stored into absolute memory location 4, and the address for the current PCST must be stored into locations 1 and 2.
- b. Set Registers. All necessary hardware registers (12) must be set to address the appropriate points in the task's code and data. This is done by the cpu microcode, using the addresses stored in absolute locations 4, 1, and 2. At this point, the task is running.

See next page for these operations.

The Current Task



Set Absolute Memory

1. Once a particular task is selected for execution, the Dispatcher stores the address of this "current" TCB (CTCB) into location 4.
2. The Dispatcher issues an EXIT (Interrupt Exit) instruction to the CPU. The EXIT microcode does the remaining steps.
3. Using the CTCB address, the microcode obtains the PCST and PDST pointers from the TCB, and stores the PCST pointer into locations 1 and 2, and the PDST pointer into the DST register.

Set Registers

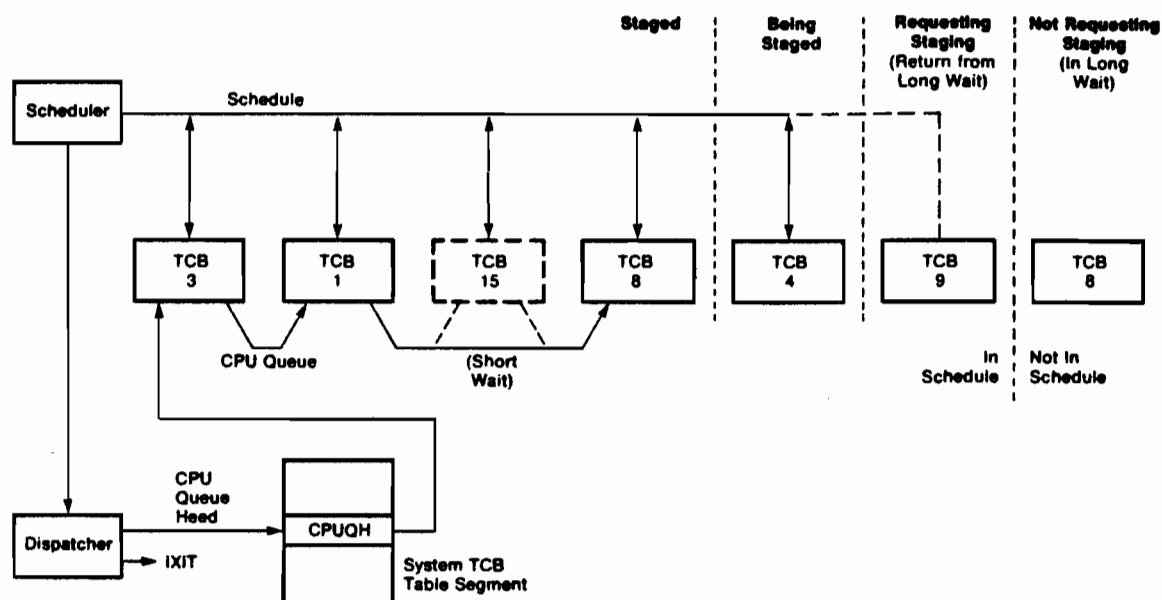
4. The CPU microcode uses the addresses in locations 1, 2, and 4, and the values in the TCB to set:
 - PB and PL code registers
 - SB, Q, S, and Z stack registers
 - DB and DL global data registers
5. The P, Index, and Status registers are set from the current stack marker.

Task execution begins (or resumes) at the new P-register setting.

Scheduling Tasks

How does the Dispatcher know which task to run?

There are two separate "linked lists", each consisting of forward and backward pointers, threading through the TCB's. The position of a task in these lists define its "precedence"; the task which has highest precedence is the one that is currently dispatched (has possession of the CPU).



The first list is the Schedule, maintained by the Scheduler. This is a list of tasks which are "staged" or being staged. Unstaged tasks lack memory resources--i.e., some segments may be absent from memory. [The subject of absent segments is discussed later; see page 3-14.] Task 9, above, is "ready to run" but is unstaged. Task 6, also unstaged, is not ready to run. Tasks 3, 1, 15, and 8 are staged and so are passed to the second list, the CPU Queue. The Scheduler assumes these are running.

The CPU Queue, maintained by the Dispatcher, is arranged in order of task priority, and first-come-first-served within priority levels (see next two pages).

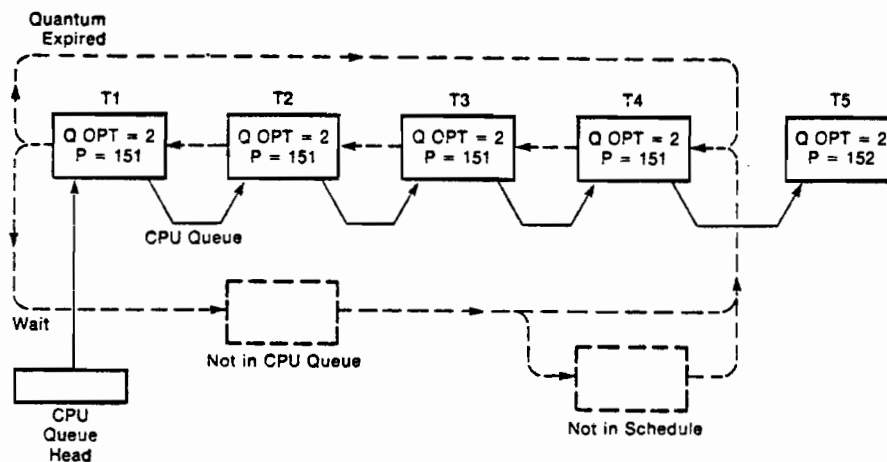
The address of the TCB at the head of this queue is always stored in a known location of the System TCB Table Segment, where it is accessible by the Dispatcher for storing in memory location 4.

A task (e.g., TCB 15) which encounters a short wait, such as disc I/O, temporarily drops out of the CPU Queue, but retains its position in the Schedule.

Task Priority Schemes

Round-Robin

This is specified by a Quantum Option of 2 (i.e., QOPT=2). Assume 5 tasks, four with Priority = 151, one with 152.



Task T1, being at the head of the CPU Queue, executes until its time quantum expires or it goes into a long or short wait.

If the quantum expires, the CPU Queue is rearranged so that T1 is positioned after T4 (the last task in the list with the same priority). T2 is now at the head of the queue.

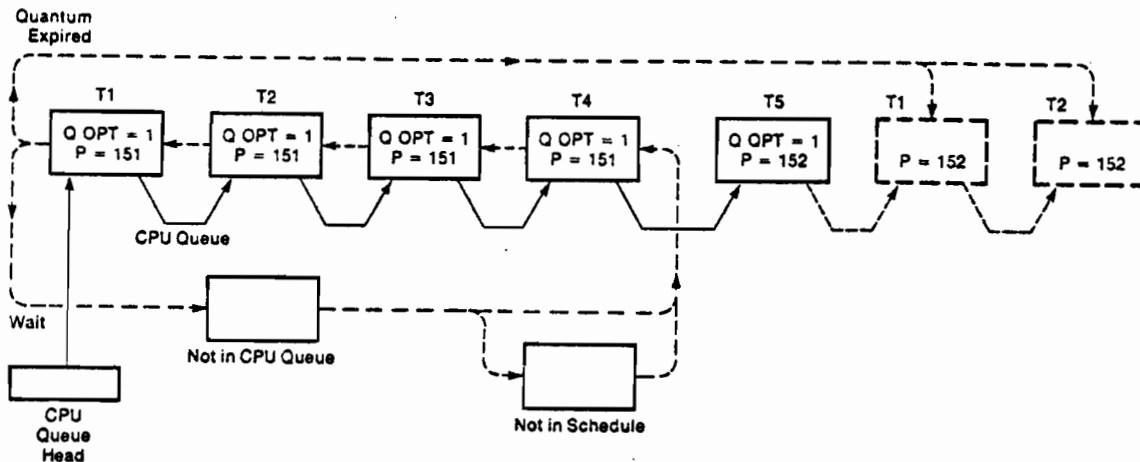
Assuming tasks T1 thru T4 each execute till their quanta expire (compute-bound), they will continue to take turns in execution in a circular "round-robin" fashion (upper dotted loop).

When the currently executing task begins a wait, it is removed from the CPU Queue. If it is a short wait, the task is simply inserted at the tail end of its priority level on return from the wait; this is the lower dotted-line loop. If it is a long wait, the task is removed from the schedule as well. In this case, if a higher precedence task demands its memory, the task may become unstaged and then, on its return from the wait, would have to be staged again by the Scheduler. Like the former case, this task returns to the tail end of its priority level.

Task T5 will not receive an opportunity to execute until tasks T1 through T4 are all out of the CPU Queue, performing waits, since it has lower priority (152). Its execution may be interrupted by any of the four higher priority tasks becoming ready to run (the third eventuality mentioned on the preceding page). It would typically be a background task.

Priority-Degradation

This is the default scheme; may be specified as QOPT=1. Assume the same five tasks.



If T1's quantum expires, its priority is "degraded" by increasing its value (to the next lower occupied priority level, or increment by one if there are no lower levels). Then the CPU Queue is rearranged to position T1 according to its new priority value--in this case after T5.

If T2's quantum expires, its priority is likewise degraded.

But if any executing task begins a long wait, it is removed from the CPU Queue and the Schedule. When the long wait completes, the task is rescheduled at its undegraded priority. If the task is still staged (or when it is), it is inserted back in the CPU Queue at the tail end of its priority level. Since its priority has not been degraded, this task will have a higher position in the CPU Queue on its return.

The more frequently a task runs into quantum expiration, the lower will become its priority; it is assumed to be compute bound.

The more frequently a task goes into long waits, the more it will tend to maintain its priority level, and thus be assured of faster response. Typically this would be a terminal task, for which fast response is desirable.

When the lowest allowed priority level is reached (255), round-robin scheduling occurs. Long waits always restore priority to the original (or nominal) undegraded value.

Scheduling Options

Users can influence system performance by managing three scheduling options. This is done by assigning non-default values to parameters of TCREATE when each task is created.

The three options are:

	<u>Default</u>	<u>Range</u>
Task Priority	151	0-255
Time Quantum	2 sec	0-6 sec
Queue Option	1	0-2

Since a task is automatically created for a user's main program, without an explicit TCREATE call, the main program initially assumes the above defaults. However, they may be changed by use of the TALTER service.

The highest priority values, from 1 through 50 are reserved for system use.

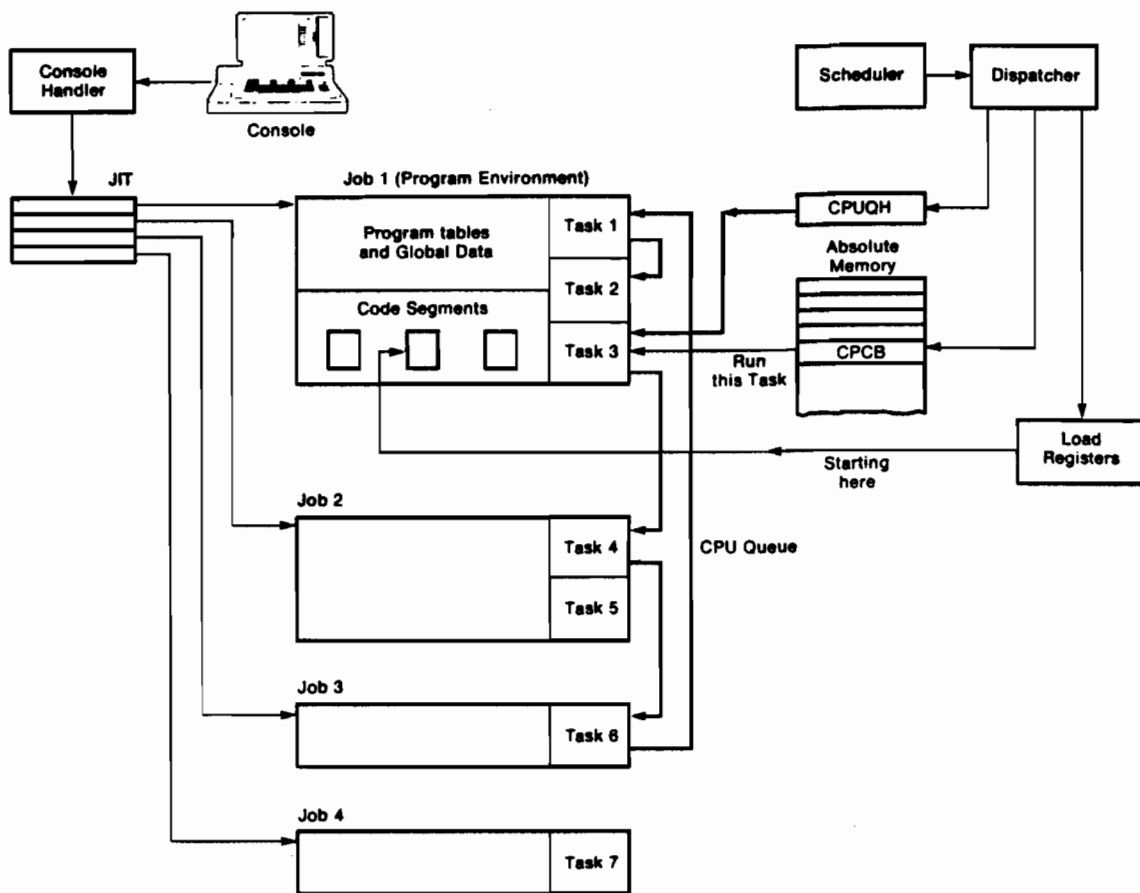
Guidelines:

Use round-robin scheduling among a group of tasks for best overall throughput, where response is less important.

Use priority-degradation for a group of tasks that do not need best response, so that certain other tasks needing best response will maintain higher priority. Those other tasks, needing good response, may be assigned either queue option (next two items).

If using priority degradation, you can increase the time quantum of the fast-response task (e.g., 3 sec), so that it will go into priority degradation less often.

If using round-robin for the fast-response task, assign it a higher priority (e.g., 100); it can be a round-robin "group" of one.

Review No. 2

A request at the console to execute a program causes the Console Handler to activate the program's complete environment as a job. (See Review No. 1 for details of the program environment.)

Internally, each program has one or more tasks which are either ready to run or not ready to run. The Scheduler informs the Dispatcher which ones are ready.

The Dispatcher arranges tasks in a CPU queue in order of priority. In the above example: Tasks 3,4,7,1,2. Task 5 and 6 are presumably not ready for execution at the moment, so are not in the CPU queue.

The Dispatcher stores the identity of highest priority task in a location known as the CPU Queue Head in the System TCB Table Segment.

The Dispatcher launches the highest priority task by storing its address in absolute memory location 4 and issuing an IXIT instruction to load the CPU hardware registers.

One of the CPU registers is the R-register, which points to the next instruction to execute in the processor. Since this register now points to an instruction in a code segment relating to Task 3, Task 3 is now running.

Task 3 will run until some other task is assigned the CPU Queue Head.

Section 3 Disc Storage



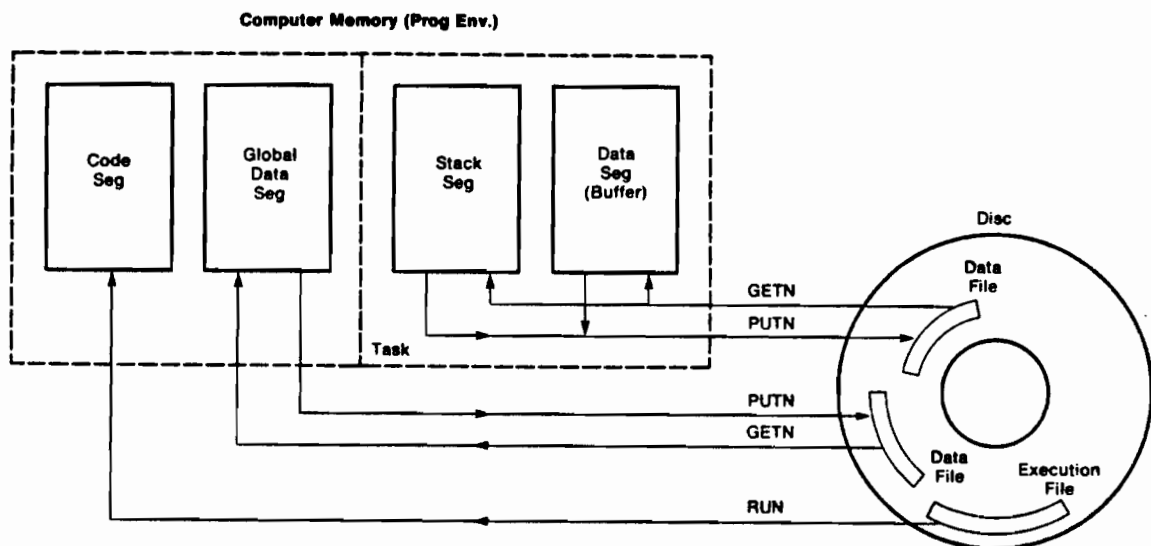
Permanent Data Storage

In the preceding section, code segments and data segments were shown to be part of the program environment for a job. What happens when the job ends?

At end of job, all code segments, data segments, and associated tables (PET, IDTT, PCST, PDST) disappear, are "deallocated".

A permanent copy of the code is automatically retained on disc as a workspace file, discussed later. The copy in memory is only a working copy of the execution file in the workspace.

The permanent copy of the data, as updated by PUTN and GETN operations, is maintained in data files on disc. The data segments in memory are merely temporary buffers.

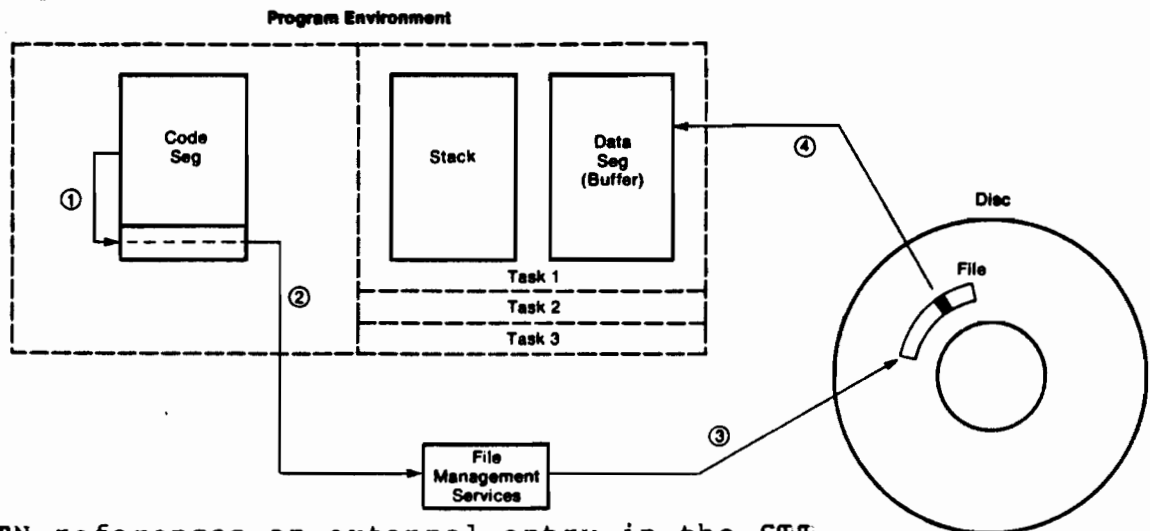


We will now consider the mechanism by which data files are kept up-to-date by file accesses (GET's and PUT's).

(Refer back to page 2-9 in Section 2 for orientation.)

Basic File Transfer Mechanism

Simplistically, a Get-Next transfer would operate as follows.



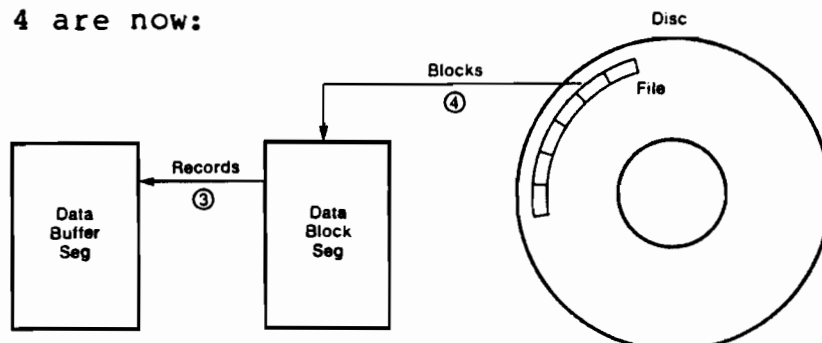
1. PCAL GETN references an external entry in the STI.
2. This invokes the Get-Next file management service.
3. The service accesses the appropriate data file.
4. Moves the record from the disc file into the data segment.

Basic File Management

However, it would be inefficient and grossly time consuming to have a disc access for every record transferred.

Thus file management transfers a block of records to/from the disc, and dispenses the records one at a time to/from the buffer segment.

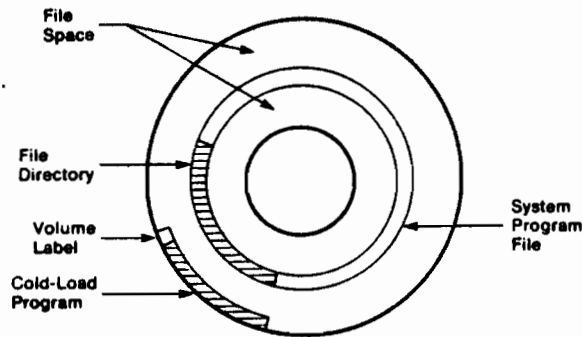
Steps 3 and 4 are now:



3. Move a record from an intermediate "data block segment" (transparently managed by File Management) into the buffer segment.
4. Get another block of records from the file when necessary.

Note: The data block segment is located in the system environment (pointed to by an SDST entry) for shared-access files, and in the program environment (pointed to by a PDST entry) for exclusive-access files. We will assume shared-access.

Disc Organization



The volume label identifies this particular disc volume.

The cold-load program (used by hardware) is the only other data on the disc not organized as a file.

A complete copy of the system code is kept in the middle of the disc, in the System Program File.

The file directory also is kept in the middle of the disc. For each file on the disc it describes: who created it, when it was created, how many times it has been accessed, when it was last accessed, how big it is, how much space is left, whether it can be extended, etc.

All information on the disc, with the exception of the volume label and the cold-load program, exists as files. This includes the directory itself.

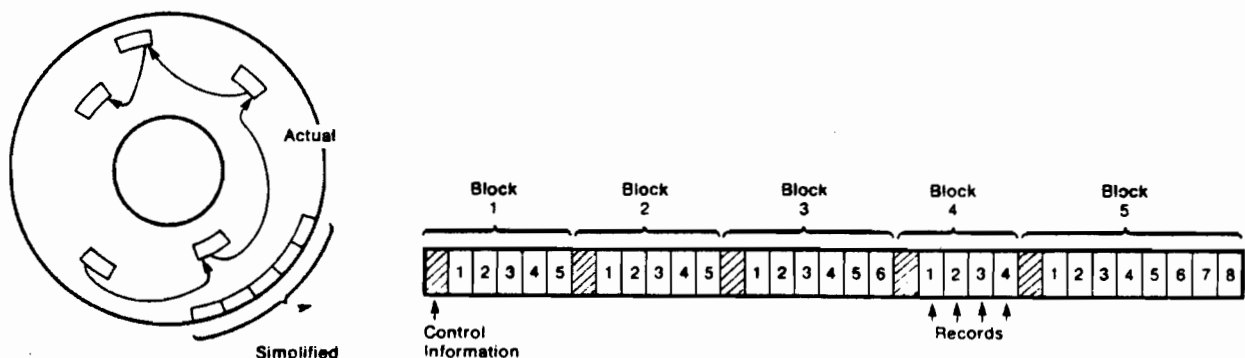
Record Blocking

Topic #3 showed that disc transfers are made in blocks of records.

Data segments are limited in size to 2K bytes (2048); it is therefore convenient to make the block size 2K also. Since records can be variable length, the number of records in a block is unfixed.

The blocks of a file need not be contiguous on the disc; control information in each block tells where the next block is located, how many records are in this block, etc. This takes ten bytes, leaving 2038 bytes maximum for data records.

For simplicity, files are shown as having contiguous blocks.



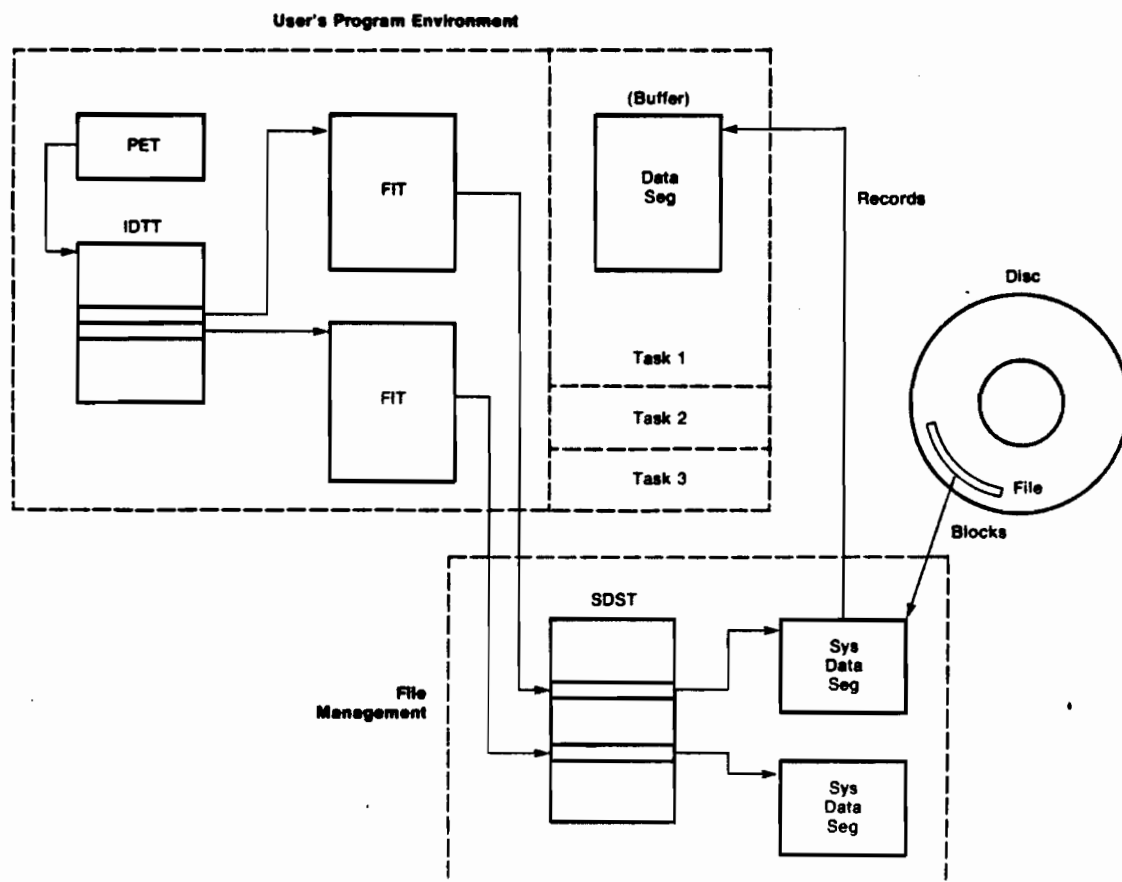
File Information in Memory

The preceding page showed that all information pertaining to all files can be found on the disc. However, just as it is inefficient to transfer data records individually between disc and memory, it would be inefficient to access the directory repeatedly for the file information for each record transferred.

Thus there is a File Information Table (FIT) in the user's program environment (in memory) that provides ready access to the essential information pertaining to the file.

There is one FIT for each file currently opened by the program. Like TCB's, FIT's are dynamically acquired, exist only as long as the file is open, and are assigned ID numbers for identification and location by the system.

One function of the FIT is to identify a particular system data segment (in the File Management environment) which is to be used for block transfers of data between disc and memory. Thus it contains a pointer into the System Data Segment Table (SDST).



In this GETN operation, the user's task simply asks for records from his file, one at a time, unaware that they are actually coming from a data segment, and that a disc transfer refills that segment when empty by transferring in a whole new block.

Example of File Operations

These statements in an SPL-II Program....Produce these results:

LOGICAL FILE3ID

Reserve stack space for ID number.

BYTE ARRAY DATABUF(0:131);

Reserve 132 bytes for buffer space in data segment.

OPEN (FILE3,5, FILE3ID);

Open FILE3, get an ID number for it, set up FIT and a system data segment, with an SDST link.

GETN (FILE3ID, DATABUF, 132);

Process record

Repeat

.

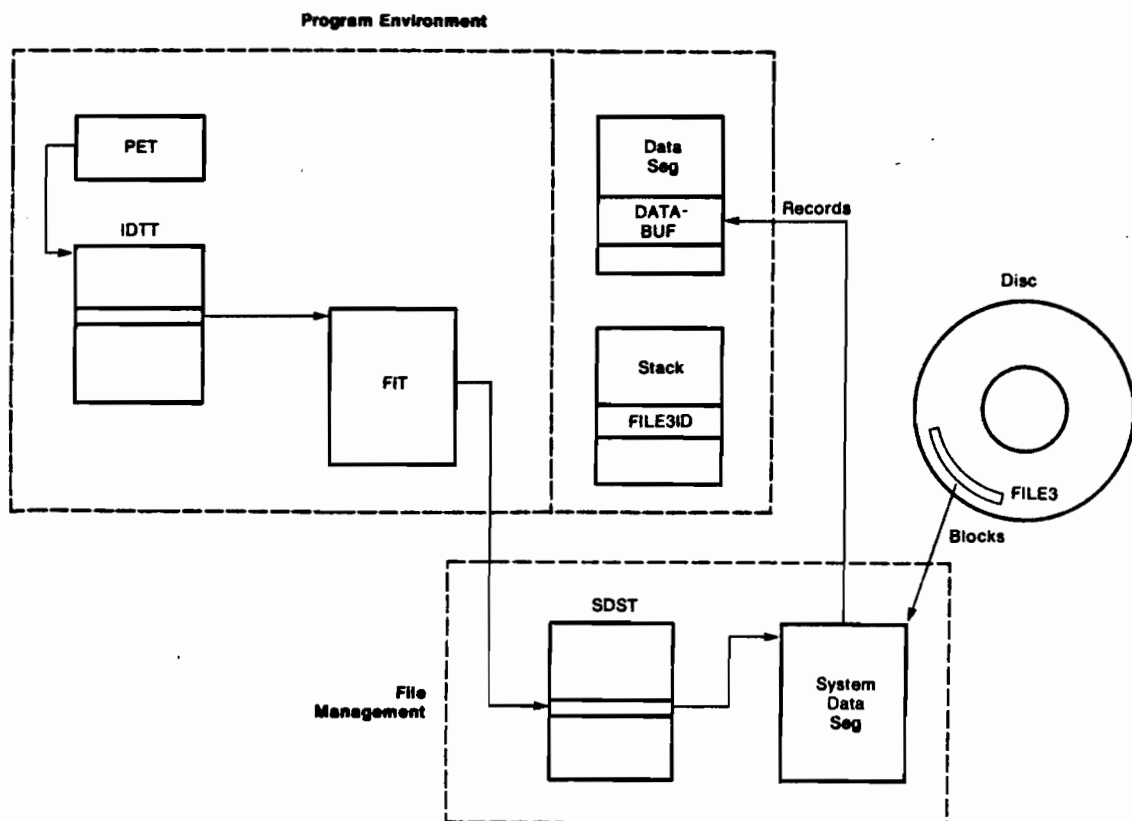
.

.

Get a record from the file (identified by its ID#) and put it in my data buffer; maximum acceptable record length is 132 bytes.

CLOSE (FILE3ID);

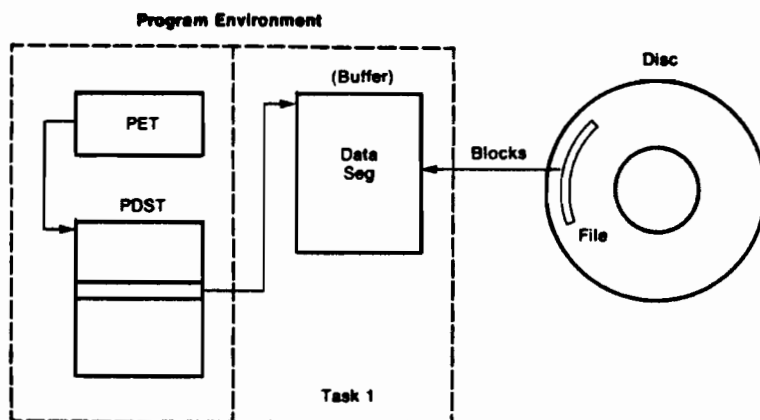
Close the file and deallocate: FIT, system data segment, SDST entry.



Primitive Access

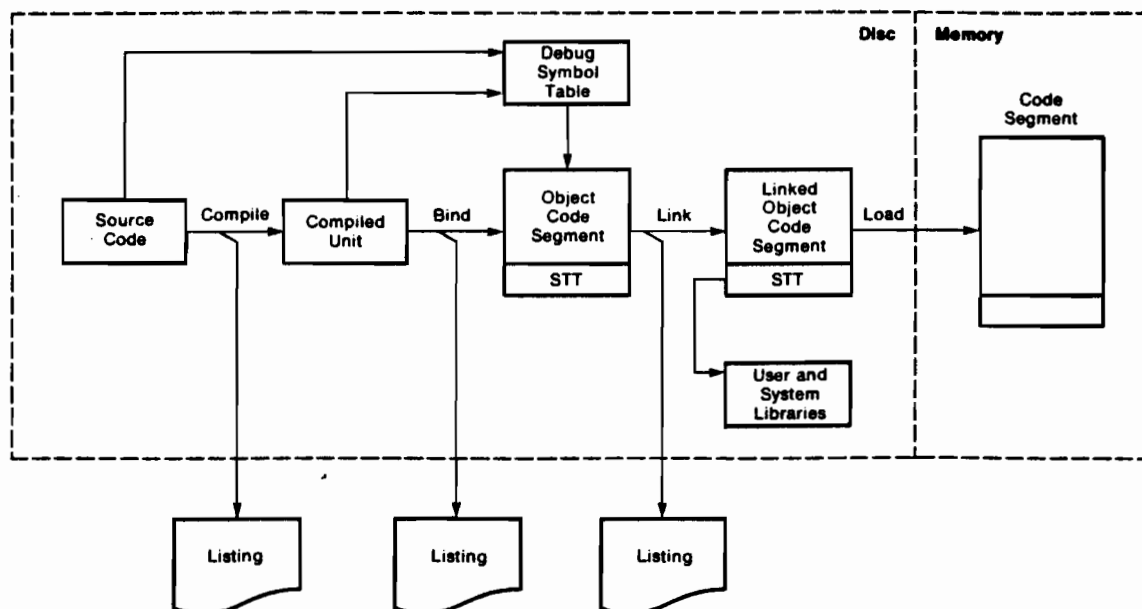
The one exception to the foregoing structure is "primitive access" files. In this case, the user provides an entire data segment for a data buffer, and prefers to handle entire blocks instead of records. Since control information is not needed within the blocks, the full 2048-byte block size is available for storing data.

The system data segment is not needed, so the PDST points directly to the buffer segment.



Storage of Programs

The preceding pages showed how data is stored on the disc. To see how programs are stored on disc, we need to review the components that normally exist during the various phases of program development.



For a simple program, the main components are as follows:

Source code consists of the human-readable source statements.

The source code is compiled into the form of a compiled unit and a compiler output listing is produced.

The compiled unit is "bound" or "prepared" into the form of an object code segment, with all internal procedure references resolved by entries in the STT (see page 2-7 in Section 2), and a Segmenter output listing is produced.

The object code segment is linked to any referenced user and external system libraries (by further entries in the STT) and then exists as a linked object code segment; a linker output listing is produced. The program, on disc, is now runnable.

As a final step, the linked object code segment is loaded from the disc into memory for execution.

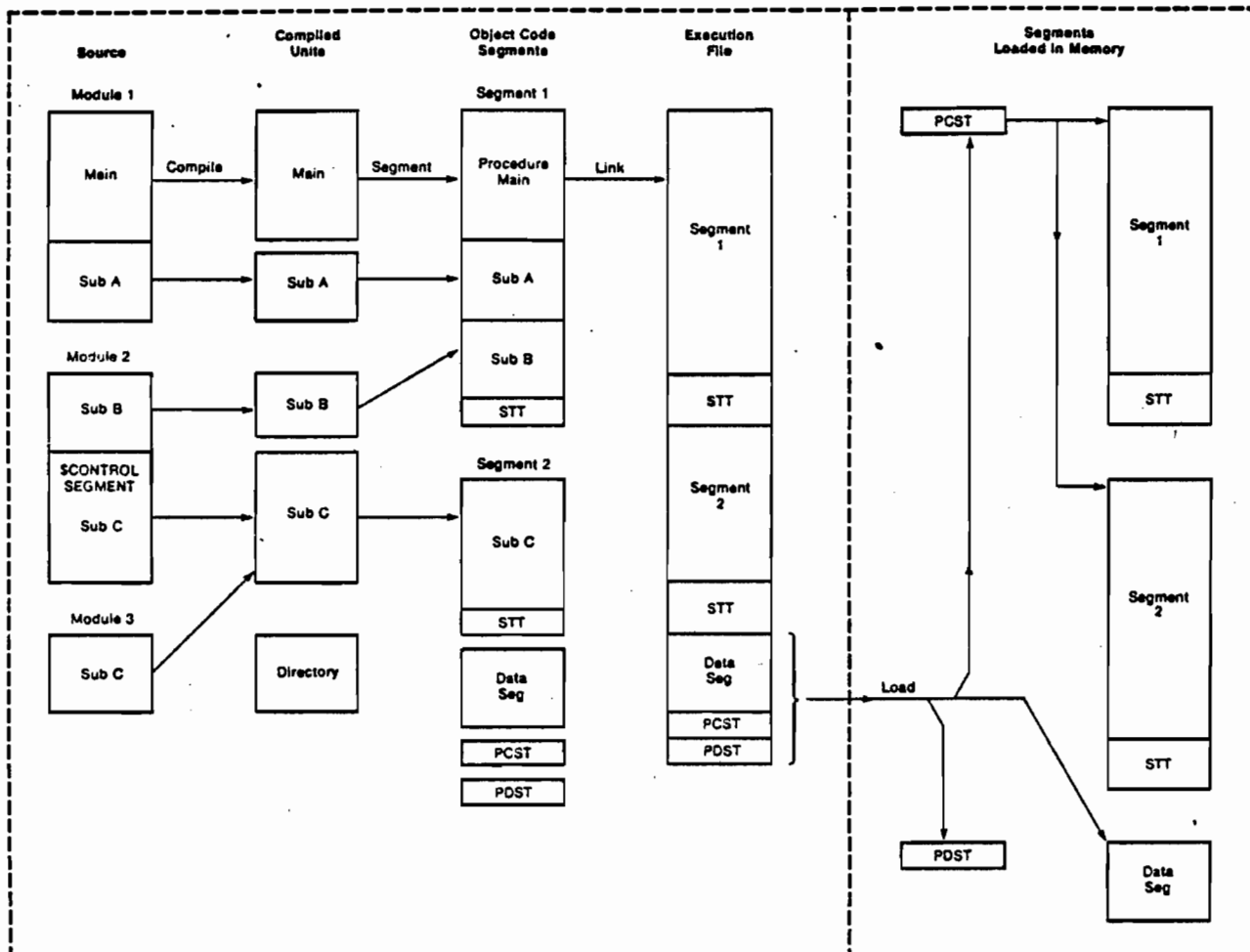
Since Debug uses symbolic references and must be able to relate these symbols to places in the object code, a Symbolic Debug Symbol Table must also exist.

For any runnable program, all of the above components typically exist together on the disc, in files created by a Language Subsystem.

Large Programs

The preceding picture was for a simple program, having one source module, one compiled unit, and one object code segment.

A large program may have several source modules, compiled units, and segments, and they need not have a one-for-one correspondence among them. Thus the arrangement of these components is a little more complicated.



In this example, the programmer has (for his own working convenience) divided the source program into three source modules. The program has a main procedure and three subprograms (A, B, C). Since C is large, he has decided to begin it with a \$CONTROL SEGMENT statement, so it will become a separate segment.

In compilation, the main program and each subprogram become separate compiled units, disregarding module boundaries.

Note that if Module 3 is revised, the entire Sub C compiled unit must be recompiled. Thus there needs to be a Directory of compiled units to define the contents of each.

In segmentation, code is divided into two segments at the \$CONTROL SEGMENT statement. References between Main, A, B, and C are resolved in the two STT's. Declared arrays are set up in data segments, and PCST and PDST are created.

In linking, external references are resolved (note larger STT's in the code segments). All segments and tables are combined into an execution file (though possibly separated in blocks on the disc).

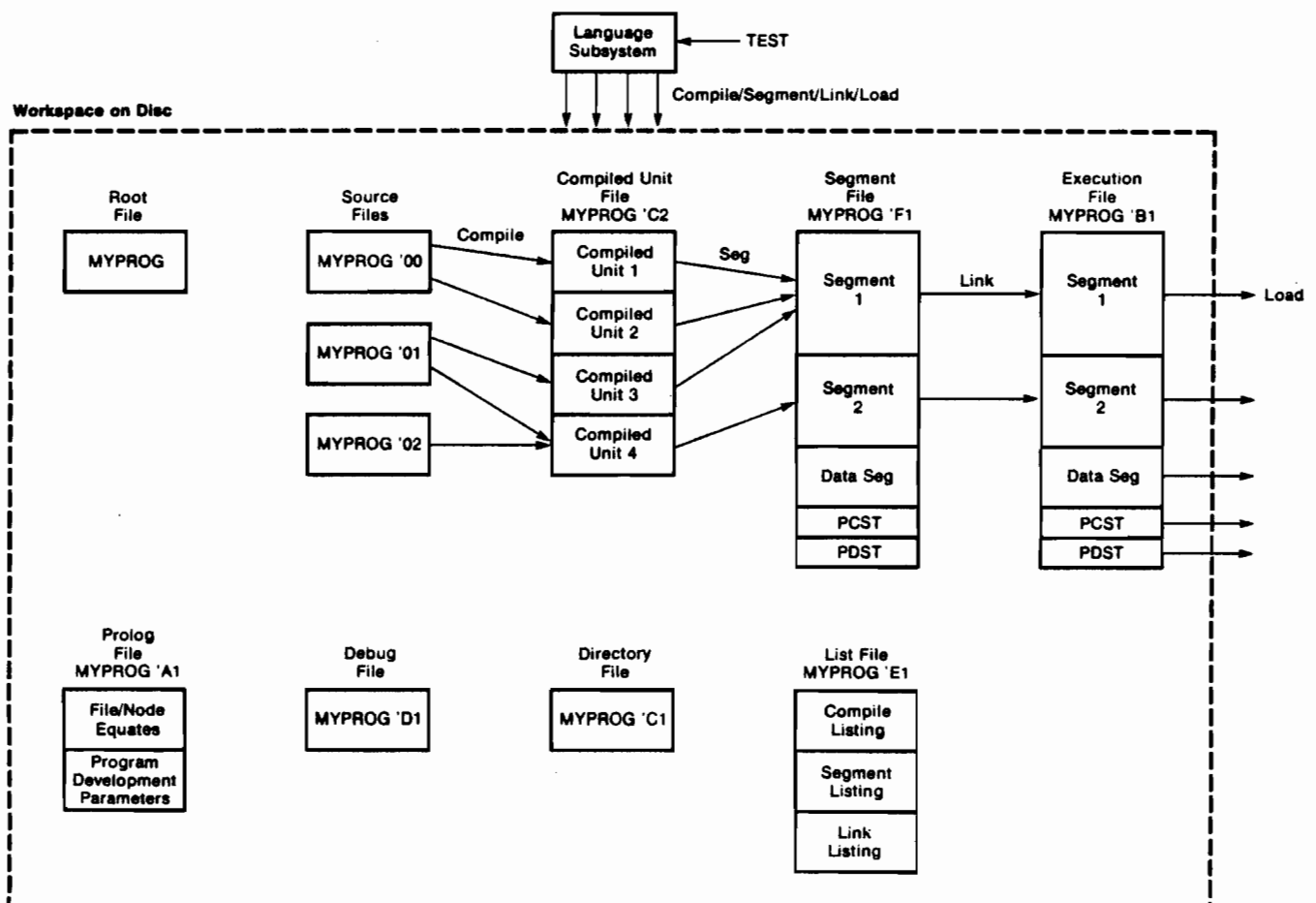
In loading, the PDST, PCST, and data segments are brought into main memory in block transfers from the disc. As execution begins, code segments are also brought in, as needed, in block transfers.

Workspace Concept

The user could create all of these components individually, saving each with a unique file name as he goes. Under the workspace concept, however, a TEST command will cause all necessary components to be saved, with unique file names, all automatically.

The entire group of files can then be referenced by program name.

The following diagram shows the files for a workspace created for a program named MYPROG.



Eleven uniquely named, permanent files are created for this program by the Language Subsystem. Several are, or can be, created automatically when the TEST command is given.

Only the root file bears the unmodified program name. The root file identifies all other files contained in the workspace.

The prolog file is identified by the program name with a suffix of 'A1. It contains all file equates and node equates and other parameters (SET) currently in effect for this program.

There is one source file for each source module, identified by suffixes '00 through '99. Thus the maximum number of source modules for a program is 100.

All compiled units go into one file, identified by the suffix 'C2.

The directory file for compiled units has the suffix 'C1.

All segments go into the segment file, identified by the suffix 'F1.

The execution file is identified by the suffix 'E1.

The list file contains all the listings produced by each of the operations performed on the workspace. Its suffix is 'L1.

The debug file has the suffix 'D1.

Each of the above files has its own requirements for file structure.

<u>File</u>	<u>Organization</u>
Root	Sequential
Prolog	Library
Source	Keyed sequential
Compiled Unit	Library
Directory	Relative
Segment	Library
Execution	Primitive
List	Keyed sequential
Debug	Relative

Note all of the above files need to be opened simultaneously. The Language Subsystem creates and opens whatever files are necessary.

Commands exist for the user to compile, segment, and link various components of his workspace if he so desires.

Program Loading

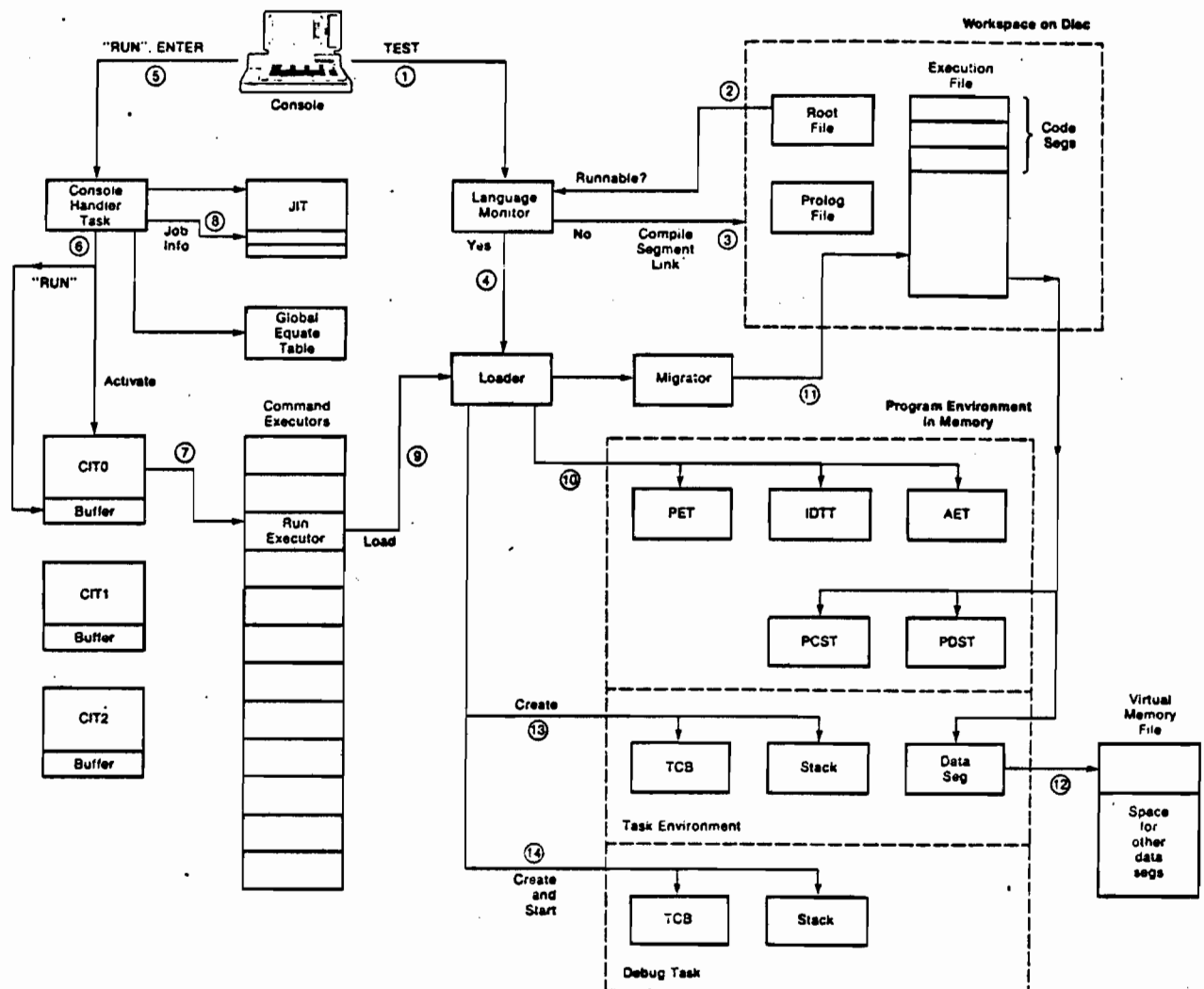
Program loading was shown as the final step in the preceding three diagrams. We will now examine the load mechanism.

Loading essentially consists of setting up the static environment for a job (refer back to page 2-13 in Section 2), and the primary functions are:

- a. Create the PET, IDTT, and AET (Active Equate Table).
- b. Create a virtual memory file for data segment swapping.
- c. Create the TCB and stack for the main procedure task.
- d. Load the PCST, PDST, and global and local data segments into memory.

Note: Code segments are not loaded until a task is started (next topic).

Loading is accomplished as follows:



Two principal ways of invoking the Loader will be considered:

- a) as a result of TEST (steps 1 through 4);
- b) as a result of a RUN command (steps 5 through 9).

- a {
 - 1. A TEST command/softkey starts the test sequence for the active workspace.
 - 2. The Language Monitor checks the root file to see if the program is runnable (i.e., has a valid execution file).
 - 3. If not, the workspace files are compiled, segmented, or linked, as necessary.
 - 4. If yes, the Loader is directed to load the program. (No JIT entry is made for a TEST-launched program.)
- b {
 - 5. A RUN command entered at one console is received by the Console Handler Task. The ENTER key activates the task.
 - 6. The Console Handler selects one of three Command Interpreter Tasks (CIT) to process the command; the CHT stores the command image in the CIT's buffer and then activates the CIT.
 - 7. The CIT determines that this is a RUN command, and so invokes the RUN Executor. (There is an executor for every command.)
 - 8. A JIT entry is created, and job information is stored in it.
 - 9. The Loader is directed to load the program named in the RUN command. (Relinking takes place if the execution file is not linked to the current system.)

Loading

- 10. The Loader causes memory space to be allocated for the program and in that space creates the Program Environment Table, the ID Translation Table, and the Active Equate Table. The AET is derived from equates in the Global Equate Table (maintained by the Console Handler) and in the program's Prolog File.
- 11. The Loader transfers from disc to memory the PCST, PDST, and all data segments that exist in the Execution File.
- 12. The Loader also creates a Virtual Memory File on disc; if the system needs the memory space occupied by this program's data segments, they are swapped out to this file. The file is expandable for dynamically created data segments.
- 13. The Loader creates the main procedure's task (TCB and stack).
- 14. The Loader creates and starts the Debug Task (only if this is a TEST operation).

Task Start

Program loading, as discussed on the preceding two pages, loaded data segments and various tables into memory, but no code segments.

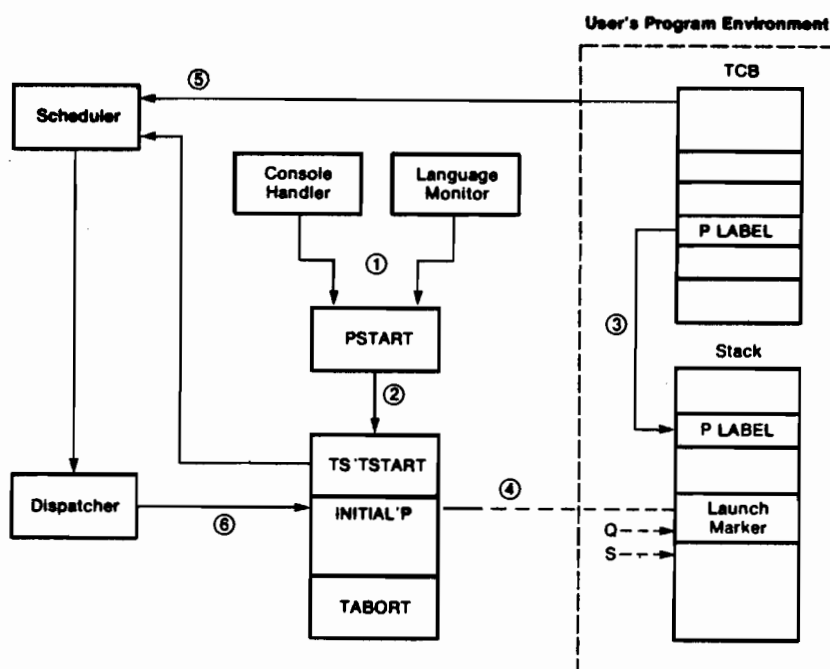
Code segments are brought into memory only when needed. The first is needed when the first task of a program is started.

Either the Console Handler or the Language Monitor initiates the first task start.

Starting a task occurs in two phases:

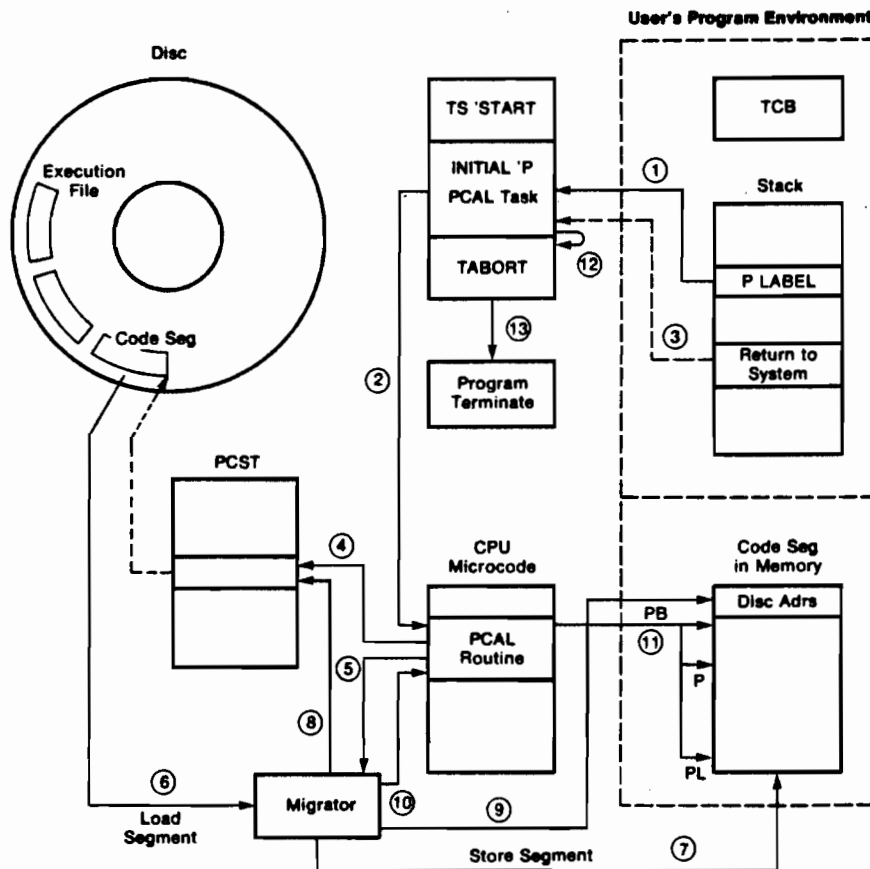
1. Set up a system subroutine (INITIAL 'P) to run on the user's stack in order to put a return-to-system marker on the stack.
2. Have INITIAL 'P call the main program task as a procedure call.

The first phase is shown below.



1. The Console Handler (on receipt of RUN-executor completion status) or the Language Monitor calls the PSTART procedure.
2. PSTART calls a task management procedure (TS 'TSTART) to initialize the user's TCB and stack, and request scheduling (steps 3-6).
3. The procedure label (starting point address) of the main program task is loaded onto the stack for future PCAL (next page).
4. TS 'TSTART also puts the launch marker on the stack, pointing to the INITIAL 'P subroutine. (Also increments a task counter.)
5. And requests scheduling of this task (link TCB into Schedule).
6. Dispatcher launches task, starting in INITIAL 'P (See page 2-15 for details.)

Calling the User's Task



1. As INITIAL 'P begins to run, the launch marker (previous page) is deleted from the stack. Also the quantum timer begins. Then INITIAL 'P gets the main program's task procedure label from the stack; this gives the code segment number and the starting procedure within that segment.
2. INITIAL 'P issues a hardware PCAL instruction to call the specified user procedure.
3. PCAL puts a new marker on the stack pointing back to INITIAL 'P for eventual return to the system (see step 12).
4. PCAL discovers that the "absence" bit in the PCST entry is set (segment is not in memory); the PCST entry contains an absolute disc address that points to the desired segment on disc.
5. PCAL invokes an absence trap which calls the Migrator, and passes the code segment number to it.
6. Using the disc address in the PCST, the Migrator loads the code segment from the disc, and:
7. Stores it in available space in memory.

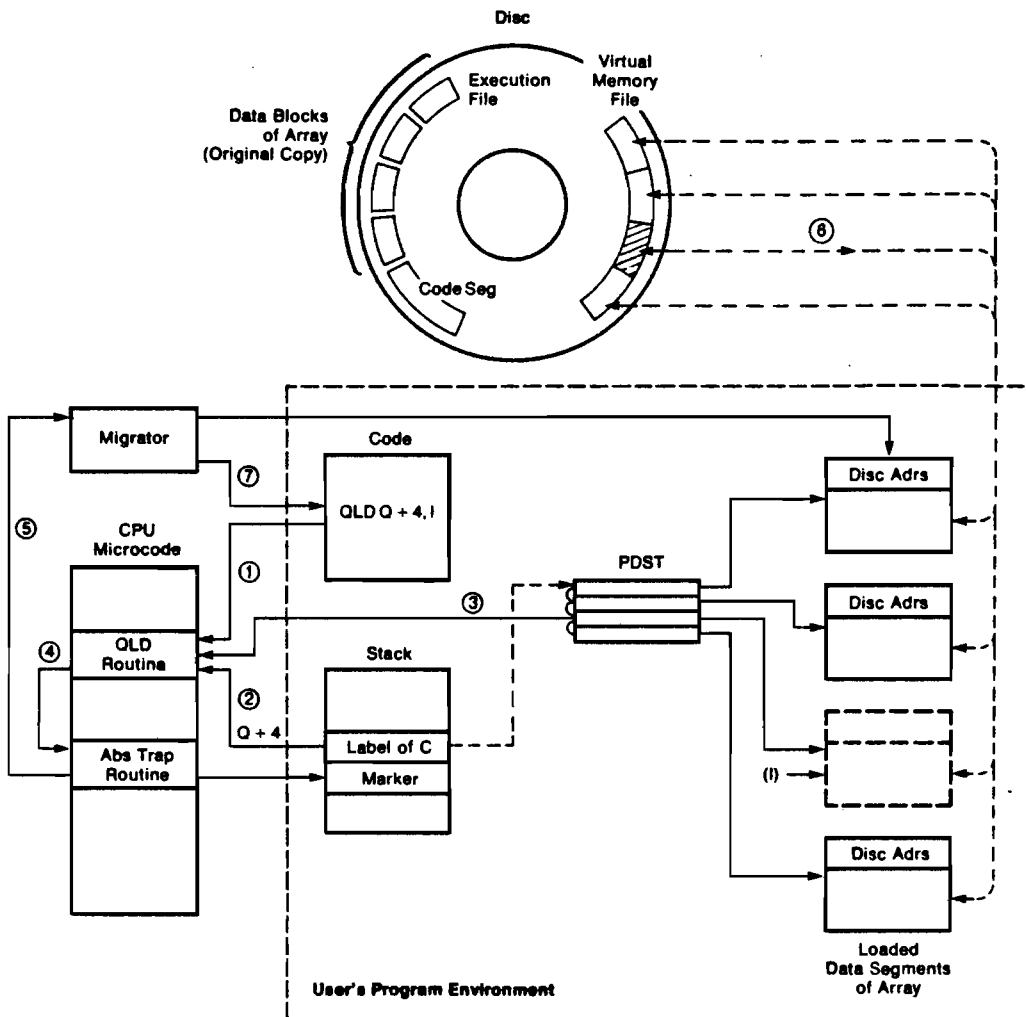
8. The disc address in the PCST entry is replaced with a memory address, and the absent bit is cleared.
9. The disc address is stored in an area of the code segment that is not addressable by the user. This is done so that in case the Migrator needs to temporarily overlay this code segment to make room for another, it will be able to bring a new copy in from the disc at a later time.
10. Since the code segment is now present, control returns to the PCAL routine in microcode.
11. The PCAL routine now completes its action of setting up the PB, PL, and P registers to point to the appropriate addresses in the code segment. The information for this is obtained from the segment's PCST entry. The first task of the program now begins to run at the established P-register setting.
12. When the task completes its execution, control is passed back to the INITIAL'P subroutine, which now calls the TABORT procedure. (TABORT handles both normal and abnormal task terminations.) This procedure decrements the program's task counter.
13. At the end of the last task (task count = 0), TABORT calls the Program Terminate procedure. This deallocates all memory resources that were allocated in the load and start operations.

Note: Parts of the above action sequence apply in other situations. Any time a PCAL references a procedure in an absent code segment, steps 5 through 10 describe the way the segment is made present.

If a PCAL references a procedure in a code segment that is already present, the "make-present" steps are skipped; action proceeds from step 2 directly to step 11.

Swapping Data Segments

Under the topic of "Program Loading" (step 13 on page 3-12) a Virtual Memory File was created for the purpose of data segment swapping. The mechanism for doing this is shown below.



Suppose the user's program has a large array which requires four blocks in the execution file. When loaded into memory, the array occupies four data segments, located by four chained PDST entries, and there is a virtual memory file on disc having reserved space for receiving each of the data segments.

During execution, the program refers to the "Ith" element of the array. In BASIC:

```
10 DIM C(100,10)           [Allocates array at compile time]
140 X = C(I)                [Machine instruction: QLD Q+4,I,X
                             assuming label for C is at Q+4]
```

Suppose, now, the Ith element is in a data segment that is absent, i.e., the Migrator has swapped it out to the virtual memory file.

1. The QLD instruction in the program causes the CPU microcode to execute its QLD routine.
2. The QLD microcode uses the data label at Q+4 in the stack to determine which data segment contains the Ith element.
3. It discovers that the PDST entry for that segment has its absence bit set, and the entry contains a disc address rather than a memory address.
4. The microcode generates an interrupt which transfers control to the Absence Trap Routine.
5. The Absence Trap Routine request the Migrator to load the missing data segment. It also puts a stack marker on the stack to enable later return to the QLD instruction (7).
6. The Migrator finds space in memory for the data segment (likely a different space than it formerly occupied) and loads the segment into memory. It updates the PDST entry to contain a memory address and clear the absence bit.
7. The Migrator now exits (via the stack marker) to the user's code to re-execute the QLD instruction. This time, since the needed data segment is present, the QLD routine in microcode is able to access element I from the array and load it on the stack. The element can then be processed by succeeding instructions.



Swapping File Blocks

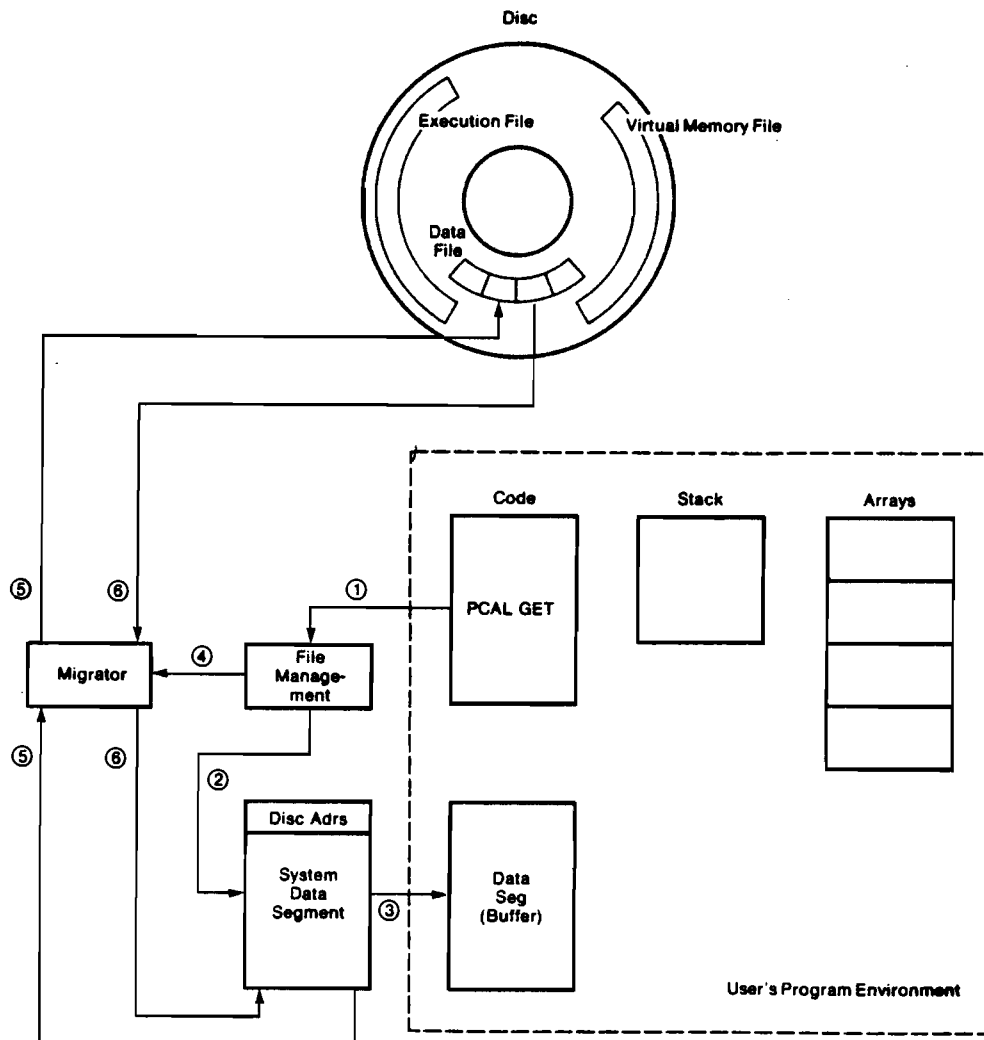
An arrangement similar to that shown in the preceding topic is used when file procedures (GET, PUT) reference a record that is not in the currently present data block.

The main difference is that since all transfers for a file are funneled through one single data segment, a two-way swap is necessary, in contrast to the one-way "make-present" operation shown previously. Prior PUT's have likely modified the current block, so it must be stored back on the disc before the new block can be brought in.

A second difference is that no temporary "virtual memory file" is involved; the blocks are taken from and restored to the permanent file on the disc.

A third difference is that File Management (not trap-processing software) calls the Migrator.

The sequence of operations is as follows.



1. As shown earlier (page 3-2), a request to access a record in a file invokes one of the file management services.
2. In the normal case, file management accesses the record from its own system data segment, and
3. Transfers the record between its data segment and the user's data buffer.
4. However, if the request references a record that is not in the system data segment, file management requests the Migrator to swap data segments.
5. Using the disc address in the header part of the current data block, the Migrator transfers that block to the data file on disc, replacing the obsolete block occupying the space.
6. The required new block is loaded by the Migrator into the system data segment in memory. The requested record can now be accessed (steps 2 and 3).

Review No. 3

The fold-out diagram at the end of this text summarizes nearly all of the operations previously described in this manual.

First, note the dotted-line "environment" boundaries. At the top left is the console. The lower half of the diagram is System Software in Memory. The center portion, upper half, shows three jobs, of which "Job 2" is shown in detail. Job 2 has three tasks, of which "Task 5" is shown in detail. The system disc is at upper right.

The operations by which work gets done in the system, from RUN command to program termination, are shown in 43 steps. These are broken down into 4 major operations: Job start, Record Access, Array Access, and Termination.

A flowchart of these operations is given on the page preceding the foldout. The 43 step numbers from the foldout are given adjacent to each block for correlating the two diagrams. ("Procedure Calls", a subset of the Job Start operations, is shown in the flowchart as a distinct major operation.)

Some operations and pointers are shown in the foldout diagram but not specifically mentioned in the 43 steps. The loading of system software (and subsequent swapping of system code segments) is indicated by the broken line at the extreme right. The Console Handler's management of the Global Equate Table and Job Information Table is indicated by arrows at the left side of the diagram. A File Information Table is shown, pointing to an entry in the SDST, which in turn points to the System Data Segment for file transfers to/from the task's data buffer segment. The other buffer segment (for primitive files) is pointed to only by a PDST entry.

Job Start

1. a) A TEST command or softkey operation is processed by the Language Subsystem. The Language Subsystem checks the root file of the program's workspace on disc to see if the current execution file incorporates all changes which may have been made in the source, compiled unit, or segment files. If not, the Language Subsystem compiles, binds, and links these files as necessary to obtain a valid execution file. [Details: page 3-9] Skip to step 4.

- h) A RUN command entered at the console is received by the Console Handler. [Details: 3-11.] Proceed to steps 2,3, and 4.
2. The RUN command is passed to one of three Console Interpreter Tasks for interpretation and validation. [3-11]
 3. The CIT invokes the Run Executor. [3-11]
 4. The Console Handler causes a new entry to be created in the Job Information Table, to point to the program's PET. [3-11]
 5. Loading begins (through step 9). [3-11]
 6. The Loader allocates memory space for the program environment and creates the Program Environment Table, the ID Translation Table, and the Active Equate Table. [3-11]
 7. The Loader gets the Program Code Segment Table, the Program Data Segment Table, and the global data segment, and all local data segments from the execution file on disc. [3-11]
 8. These are loaded into their allocated memory space. [3-11]
 9. The Loader also creates a Task Control Block and a data stack for the main procedure's task, and a Virtual Memory File on disc. (These steps are not identified on the diagram.) The task environment now exists and loading is complete. [3-11]
 10. Program start begins (through step 28). [3-13]
 11. The Program Start procedure calls the Task Start procedure to start the first task of the program. TS'TSTART fills entries in the TCB and puts starting addresses on the stack (launch marker to start INITIAL'P subroutine, and a procedure label to start the first actual procedure of the program). [3-13]
 12. TS'TSTART requests scheduling of this task. [3-13]
 13. The Scheduler links this task into the schedule. [2-16]
 14. Since the task has all resources needed to run, the Scheduler instructs the Dispatcher to put the task in the CPU Queue. [2-16]
 15. The Dispatcher links the task's TCB into the CPU Queue; and when this task gains precedence among all tasks, the Dispatcher places it at the head of the CPU Queue. The TCB address is stored in the CPUQH location of the System TCB Table Segment. [2-16]
 16. The Dispatcher stores the TCB address into absolute location 4 of memory, which now points to the TCB of this task. This is so both microcode and software can access the TCB via a known location. [2-15]

17. The Dispatcher issues an IXIT instruction to the CPU. [2-15]
18. Using the Current TCB address in location 4, the IXIT microcode obtains the addresses of the PCST and PDST of this program, and stores the PCST pointer in absolute memory locations 1 and 2, and the PDST pointer in the DST register. [2-15]
19. The IXIT microcode uses the address information in absolute memory locations 1, 2, and 4, in the stack, and in the TCB to set the remaining eleven main CPU registers. [2-15]
20. Since the value for the P register came from the launch marker (see step 11 above), execution of the INITIAL'P subroutine begins. [2-15]
21. INITIAL'P issues a Procedure Call instruction to the CPU, supplying the address of the starting procedure as given in the procedure label on the stack. [3-14]
22. PCAL microcode puts a system-return marker on the stack (actually replaces the launch marker) for eventual return to INITIAL'P (see step 40). [3-14]
23. Discovering that the starting procedure is in an absent code segment, PCAL invokes the Absence Trap microcode to make the segment present (through step 26). [3-14]
24. The Absence Trap calls the Migrator, passing the code segment number. [3-14]
25. Using the code segment number, the Migrator accesses the disc address of the segment (in the PCST) and gets the segment from the disc. [3-14]
26. The Migrator loads the code segment into memory. [3-14]
27. The Migrator returns control to PCAL, which completes its action of setting up the P register. [3-14]
28. The first task begins executing the specified procedure in the ncw-present code segment. [3-14]

Record Access

29. An instruction (e.g., PCAL GETN) in the executing code segment requests retrieval of a record from a certain data file. This generates a call to File Management. If the record is present in the system data segment reserved for transfers to and from this file, skip steps 30 through 32. [3-18]

30. If the record is not in the system data segment, File Management requests the Migrator to swap data segments. [3-18]
31. The Migrator stores the currently present data segment into its place in the data file on disc, and gets the necessary new data block from the disc. [3-18]
32. The data block is loaded into the system data segment. Control is now passed back to File Management. [3-18]
33. File Management transfers the record from its system data segment to the buffer segment in the task environment, where it is accessible for processing. [3-18]

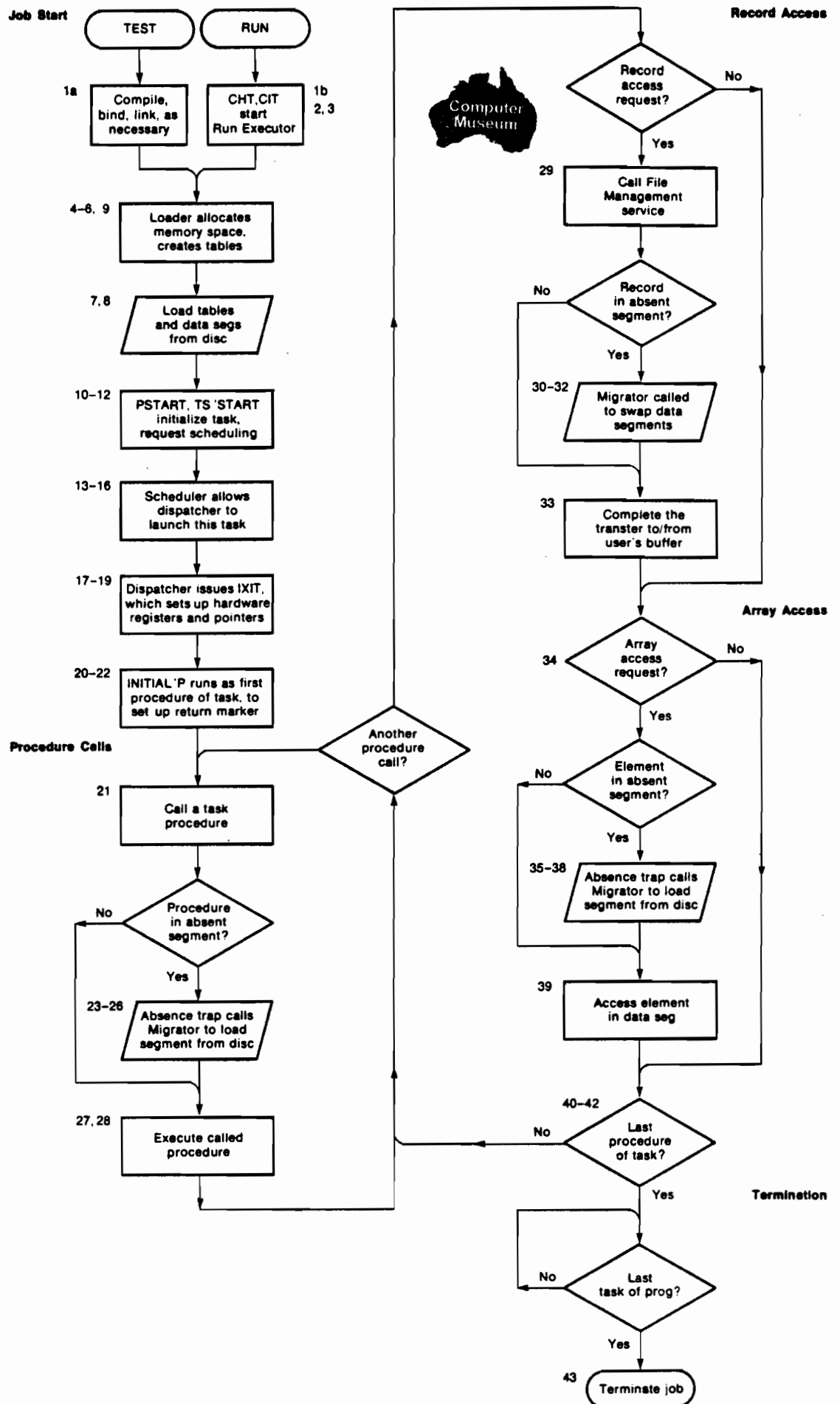
Array Access

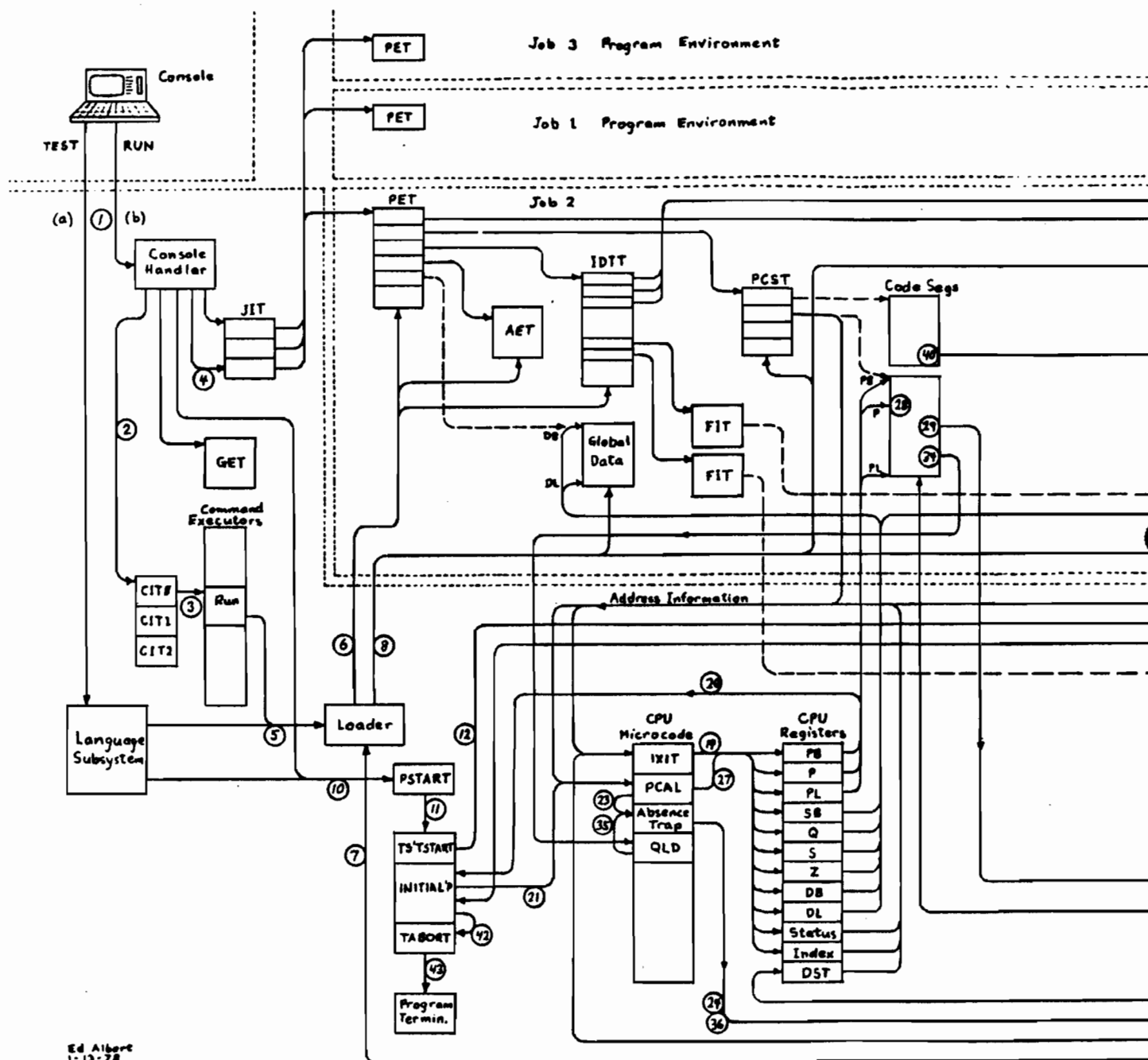
34. An instruction (e.g., QLD) in the executing code segment requests retrieval of a specific array element. This starts execution of the QLD microcode. If the requested element is in a data segment that is present in memory, skip steps 35 through 38. [3-16]
35. Discovering from the PDST entry for the data segment that the segment containing the element is absent, QLD invokes the Absence Trap microcode to make the segment present. [3-16]
36. The Absence Trap calls the Migrator, passing the disc address obtained from the DST register. [3-16]
37. The Migrator gets the necessary data block from the Virtual Memory File on disc. [3-16]
38. The data block is loaded as a data segment in the task environment. Control returns to the QLD microcode. [3-16]
39. The array element is now accessed via the data label on the stack, and is available for processing. [3-16]

Termination

40. When the final procedure of the task has been executed, the code exits to the address contained in the system-return marker on the stack. (See step 22.) [3-15]
41. The system-return marker passes control back to the INITIAL'P subroutine. [3-15]

42. INITIAL 'P calls the TABORT procedure, which terminates the task.
[3-15]
43. If this was the last or only task executing for this program, TABORT calls the Program Terminate procedure, which deallocates all memory resources allocated for this job. The JIT entry is deleted, and the job is thus terminated. [3-15]





Section 4 Hardware Operation

Operating Environment

The addressable space of an executing program consists of a code segment for the executing code, a data stack for local variables, arithmetic processing and program control, a global data area and a number of data segments for array data. Access to other code segments is done by a call mechanism thru a system code segment table which contains entries with location information for all of the code segments of the program. The code segments need not be present in memory and are brought in if called.

Most arithmetic and logical processing is of the zero address structure of the stack machine where the operands on the top of the stack are used. The executing code segment, data stack and global data areas are defined by registers for direct or implied stack address access. Areas of direct access are shown by the shaded areas in the figure. Other areas are accessed by means of indirect pointers or labels. Data segments are accessed in the indirect mode using a data label pointer through a data segment table which contains entries with location information for the data segments of the executing program.

Central Processor Registers

Note: The 16 bit address registers in the following list are concatenated with the high order bits of the associated base register to form the absolute address. For possible future needs the intermodule bus and I/O system have provided for memory addressing of up to 24 bits. The present CPU and memory, however, are implementing only 20 bits as indicated by the base register lengths is the following list. Bit positions in the address base registers are numbered as follows:

E3,E2,E1,E0,0,1,2,.....14,15

16 bit registers number from 0.

DB : Data base register - 20 bits

Contains the address of the first location of the directly addressable global data area.

DL : Data limit register - 16 bits

Contains the address of the last word of memory available to the global data space.

SB : Stack base register - 20 bits

Contains the address of the first location of the stack data area.

- Q** : Stack marker pointer - 16 bits
Contains the address of the current stack marker being used within the stack.
- S** : Memory stack pointer register - 16 bits
Contains the address in memory of the top of the stack (TOS).
- Z** : Stack limit register - 16 bits
Contains the address of the last word of memory available for use in the stack.
- PB** : Program base register - 20 bits
Contains the address of the first location of the code segment being executed.
- P** : Program counter - 16 bits
Contains the address of the instruction being executed.
- PL** : Program limit register - 16 bits
Contains the address of the last location of the current code segment.
- X** : Index Register - 16 bits
Contains the index to be used when indexing is involved.
- DST** : Data Segment Table Pointer - 16 bits
Contains the address of the user portion of the Data Segment Table. (Instructions allow for 24 bits in later hardware).
- STATUS** : Status register - 16 bits including:

```

-----
M * I * T * R * O * C * CC *   SEGMENT #
-----
0 , 1 , 2 , 3 , 4 , 5 , 6-7 , 8-----15

```

where...

- M** = Privileged/User (I/O) Mode Bit
I = Enable/Disable (I/O) External Interrupts bit
T = Enable/Disable (I/O) User traps bit
R = Right Stack Op pending bit -- set to 1 execution will proceed with Stack Op B. It is not set if Stack Op B is a NOP.
O = Overflow bit - Set but not cleared by arithmetic operations.
C = Carry bit
CC = Condition code (see note)
 CCG = 00, CCE = 10, CCL = 01
SEGMENT # = The currently executing code segment entry no. in the Code Segment Table (CST). The index into the CST for this entry is 4*STA(8:8).

note: Condition code definitions

CCA sets CC = CCL if operand < 0
 CC = CCE if operand = 0
 CC = CCG if operand > 0

CCB sets CC = CCL if special ASCII character
 CC = CCE if alphabetic ASCII character,
 upper & lower case
 CC = CCG if numerical ASCII character

CCC sets CC = CCL if opnd1 < opnd2
 CC = CCE if opnd1 = opnd2
 CC = CCG if opnd1 > opnd2

CIR: Current instruction register - 16 bits
 contains the instruction currently being executed (this
 register is not available to software).

CLK: Clock register - 16 bits
 Contains current free running clock time.

CCR: Clock comparison register - 16 bits
 Contains number to be compared with free running clock.

Instruction Formats

Instructions may take one of eight formats. Please see Section 5
 for instruction descriptions.

note: In this section and throughout this document, bit
 locations within a 16 bit register or within a data word
 are specified by NAME(L:N) where N is the number of bits
 starting at the bit location L in the word (numbered from 0
 on the left). For example, CIR(6:2) specifies bits 6 and 7
 of the CIR register.

Memory reference

7

```

-----
OP CODE * X * I * ADDRESS
-----
0-----3, 4, 5, 6-----15

```

I Indirect bit
 X Indexing to be used

If both X and I = 1 post indexing will occur for operands and
 pre-indexing will occur for branches. The displacement is a
 positive word quantity, giving sign-magnitude addressing modes.

Address decoding:

P+ CIR(6:2) = 00
 CIR (8:8) = Displacement
 Range (P,P+255)

 P- CIR(6:2) = 01
 CIR(8:8) = Displacement
 Range(P-255,P)

 DB+ CIR(6:2) = 10
 CIR(8:8) = Displacement
 Range(DB,DB+255)

 Q+ CIR(6:3) = 110
 CIR(9:7) = Displacement
 Range(Q,Q+127)

 Q- CIR(6:4) = 1110
 CIR(10:6) = Displacement
 Range(Q-63,Q)

 S- CIR(6:4) = 1111
 CIR(10:6) = Displacement
 Range(S-63,S)

		CIR														
		6	7	8	9	10	11	12	13	14	15					
P+	relative	0	0	.	.	.	.	.	.	.	.	Displace.	(0:255)			
P-	relative	0	1	.	.	.	.	.	.	.	.	Displace.	(0:255)			
DB+	relative	1	0	.	.	.	.	.	.	.	.	Displace.	(0:255)			
Q+	relative	1	1	0	.	.	.	.	.	.	.	Displace.	(0:127)			
Q-	relative	1	1	1	0	.	.	.	.	.	.	Displace.	(0: 63)			
S-	relative	1	1	1	1	.	.	.	.	.	.	Displace.	(0: 63)			

Stack and Control Instructions

```

-----
*0000 * STACKOP * STACKOP *
          A          B
-----
0----3,4-----9,10-----15
  
```

Execution is from left to right (A first, then B). Interrupts may occur between execution A and B.

Shifts, Bit Tests, Conditional Branches

```

-----
0001 * Y * SUBOP * ARGUMENT
-----
0---3, 4, 5-----9,10-----15

```

Note that in this format the Y bit, CIR(4), is a true index bit on the shift and bit test instructions, and is the indirect addressing bit for the conditional branches.

Immediate

```

-----
0010 * SUBOP 2 * OPERAND
-----
0---3,4-----7,8-----15

```

Linkage and Control Instructions

```

-----
0011 * SUBOP 3 * ARGUMENT
-----
0---3,4-----7,8-----15

```

Special Opcodes

```

-----
0011 0000 * SPECIAL * R-FIELD
-----
0---3,4---7,8-----11,12-----15

```

Move Opcodes

```

-----
0010 * 0000 * MOVE OP * CCF * SDEC
-----
0---3,4---7,8-----10,11-13,14--15

```

Mini Opcodes

```

-----
0010 * 0000 * MINI OP * RESERVED
-----
0---3,4---7,8-----11,12-----15

```

Addressing Conventions

The following are definitions used throughout in discussing address computation:

X_c = If the X-bit, $CIR(4)$, = 0 then 0 else the content of the X register.

S = The CPU register defined previously.

D = Displacement field of instruction word (see 3.3 of instruction formats).

IX_c = If word instruction then X_c
 else if byte instruction then $X_c/2$
 else if double word instruction then $2*X_c$
 else if quad word instruction then $4*X_c$.

Top of Stack

Many operations use the top elements of the stack as implicit operand addresses. The operands involved are deleted from the stack and the results of the operation are placed back on the stack.

Direct Addressing - The location of operands and branch targets in memory may be determined by a direct addressing convention that uses the displacement field of the instruction as a pointer relative to a base register selected by the instruction format. Indexing also may be used in this mode. Thus the absolute Address E (or effective address) is computed by:

$$E = \text{BASE} + \text{or} - \text{Displacement} + IX_c.$$

The base registers available to this mode are DB, Q, S and P. DB is used for global variable addressing, Q is used for parameter passing (-Disp) and local variable storage in procedures (+Disp), S is typically used for accessing temporary variables in subroutines and P is used for branches and literals.

Indirect Addressing - The location of operands and branch targets also may be determined through a pointer structure where the content of the direct address, as computed, less indexing if an operand address, is assumed to be a label. These are either Program Labels that point into code segment space, or they are Data Labels that point into the stack, the global data area or into data segments.

Program Labels - Program Labels that are found in code segments contain only offsets and can be identified only by their usage in the instruction, such as branches or P-relative loads, etc. These labels are self-relative, i.e. they contain an offset relative to their present location. Thus the absolute address of a load is:

$$E = P +/- D + (P +/- D) + fxc.$$

Pre-indexing is used for branches so the absolute address for a branch is:

$$E = P +/- D + xc + (P +/- D + xc).$$

Labels used by the call or branch instructions are different and are either logical external labels that point to other code segments or are local labels that contain PC relative offsets.

Labels that point into data areas are of four different types that are described later under the heading "Data Labels". These are differentiated by a Huffman coding of the first three bits of the label. The stack and global data labels contain offsets relative to their respective base registers.

The absolute address of a Stack Base label is:

$$E = SB + (Base +/- D)(2:14) + fxc.$$

The address for a Global Data label is:

$$E = DB + (Base +/- D)(1:15) + fxc.$$

Labels that point into data segments contain logical pointers into a data segment table in which the location of the segment is entered.

The one word DST label effectively points at the base of a data segment. The absolute address in this case is:

$$E = \text{Segment base} + fxc.$$

where:

$$\begin{aligned} \text{Segment Base} &= (\text{DST PTR} + 4 * \text{DSEG} + 2)(8:8), \\ &\quad (\text{DST PTR} + 4 * \text{DSEG} + 3). \end{aligned}$$

here:

$$\begin{aligned} \text{DST PTR} &= \%1000 \text{ for } \text{DSEG} < \%10000 \\ &= (\text{DST}) \text{ for } \text{DSEG} > \%10000 \end{aligned}$$

and

$$\text{DSEG} = (Base +/- D)(3:13).$$

As defined, up to 24 bits of segment base address have been provided for in the data segment table.

To point to a location within a segment the two word Data Segment label form is used in which the second word contains an offset from the segment base. Therefore the absolute address for the Data Segment label is:

$$E = \text{Segment Base} + \text{OFFSET} + \text{fxc}$$

where:

$$\text{OFFSET} = (\text{base} + \text{or} - D + 1).$$

The segment base definition is the same as described above.

Two types of linked segment structure have been provided to accommodate arrays that are too large to fit in one data segment. This process is invoked when the base relative address, offset plus index, is larger than the segment length. The first type uses linkages to the next segment and adjustments are made to the relative address through each linked segment to arrive at the desired address. The second type uses consecutively linked segments and through a computation goes directly to the desired segment. Linked through i consecutive segments the address now is:

$$E = \text{Seg.base}(i) + \text{OFFSET} + \text{fxc} - i * \text{Seg.lengthn}.$$

Bounds check - If E lies outside of the ranges defined in the following table, a trap to STT1 of Segment 1 will occur.

Code, direct or indirect

$$PB < = E < = PL$$

Data, direct

$$DB < = E < = DL \text{ (global data)}$$

$$\text{or } SB < = E < = S \text{ (stack)}$$

Data, indirect

$$DB < = E < = DL \text{ (global data)}$$

$$\text{or } SB < = E < = S \text{ (stack)}$$

If, when accessing Data Segments, the label does not comply with the range

$$0 < = \text{data label offset} < = \text{data segment length} - 1$$

a trap to STT13 Segment 1 will occur.

Bounds checking is applied in both USER and PRIVILEGED mode.

byte Addressing - The content of the index register in all byte instructions is a byte index that is arithmetically shifted right one bit when used in computing the word address.

In Load and Store byte instructions the displacement field is a word displacement.

For indirect addressing the label offsets are byte pointers and are added to the contents of the index register before the arithmetic right shift is performed.

Double word Addressing - The content of the index register in a double word instruction is an entry index that is multiplied by two when used in computing the word address. The instruction displacement field is a word offset. All indirect label offsets are word offset entities that are added after the contents of the index register has been multiplied by two.

Quad word Addressing - The content of the index register in quad word instructions is an entry index that is multiplied by four when used in computing the word address. The instruction displacement field and all indirect label offsets are word entities and are added after the index multiplication.

virtual memory

Code Segmentation

Functional Description - Access to a segment is done through an external Program Label used by a branch instruction or by a PCAL instruction whose N field specifies the location of a program label. This label specifies the program segment thru a CST seg #, and entry point desired (and STT # in called segment). A sequence of accesses thru system or user CST tables takes place to locate the new code and set the program registers to its location. Information about the calling code is placed in a stack marker so that program control can be returned to the caller at the end of the procedure with the EXIT instruction.

Segment Transfer Table (STT) - A Segment Transfer Table is appended to each generated program segment. Its entries are used to access procedures and subroutines local to the segment, or to procedures contained in external segments. PL- addressing is used by the PCAL and LLEL instructions to access the Table. It contains 1 word entries of two basic types, which are the program labels having the formats described under the next heading, "Program Labels".

PL entry (the last word of any program segment)

```

-----
0 * 0 * 000000 * STTL
-----
0, 1, 2-----7,8---15

```

STTL = # of entries in this Segment Transfer Table (STT).

Note: If the u-bit (uncallable bit) is set, and the calling process is in USER mode, an STT violation will occur.

A transfer through STT entry # 0 implies that execution will begin at P = PL.

Note: Any call outside the current program segment may only reference the first 128 entries in any other segment transfer table (PL, PL-127), due to only 7 bits available in the External Program Label. Calls internal to the current program segment may reference any of the 255 entries.

Program Labels

External Program Label

Program calls or branches to external program segments use an external program label.

```

-----
1 * STT ENTRY # * SEGMENT #
-----
0, 1-----7,8-----15

```

STT ENTRY # = Number to index from PL to obtain entry point into segment.

SEGMENT # = Logical entry number in CST.

Local STT Program Label

Program calls local to a segment or calls through external labels to a segment (which may be the same segment) go through a Local Program Label in the target segment.

```

-----
U * U * PB Relative Address
-----
0, 1, 2-----15

```

U = Uncallable bit prevents calls from unprivileged external segments to this section of code. An STT violation will occur in this case.

Code Program Labels

branching or literal pointers occurring in line in code are self relative (i.e. they contain the offset to the pointer target from the location of the pointer).

```

-----
Pointer Relative Offset
-----
0-----15

```

Reference Program Labels

Pointers passed as reference parameters on the stack by the branch or PCAL instruction that point into the current code segment contain offsets relative to the code segment base PB.

Pointers passed as reference parameters on the stack that point to external code segments are external program labels.

Code Segment Table & Code Segment Table Pointer

The Code Segment Table contains a list of code segments that have been allocated by the system and assigned virtual memory space (note that this also includes system code segments). Its length is determined at system generation time, and consists of a number of 4-word segment descriptor entries. The actual number of entries in use at any point in time is a variable, limited only by the length of the table. Entries are dynamically allocated by the operating system as programs are loaded and unloaded. Each entry contains control information about the segment, and gives its length and starting address.

Segment 1 contains the routines needed to service internal interrupts. These are described further in the section covering interrupts. Segment 2:N ($N \leq 191$) contain such things as system code, shareable code, I/O drivers, service routines for external interrupts, etc.

The remainder of the table (beyond %300) consists of user program segments. Since the first entry must contain the length - 1 of this table (identical to the segment 0 entry), this part of the table is limited to 63 segments. The User CST pointer in mem. addr.' 1,2 is set to the address pointed at by the UCST data label in the Current Task Control block. (In the figure above, process # 2 is currently executing). Accessing the table is done via the PCAL, EXIT, Branch external, IXIT and DISP instructions and is completely transparent to the user. E.G. if in PCAL an external segment number lies in the range $0 < \text{Seg. No.} \leq \%277$, the entry is accessed via the SCSTP. If the seg. no. is in the range $\%300 < \text{Seg. No.} \leq \%377$, then %300 is subtracted from the segment number and the entry is accessed via the UCSTP.

The entries of the CST have the following form:

	0	1	2	3	4	-----	15
CST0	* 0	* 0	*			Length - 1	*
CST1	*					Reserved	*
CST2	* A	* M	* T	* R	* res	* VATBI(8:4)/Addr(8:8)	*
CST3	*					Address	*

A = Absence bit, set to 1 if the code segment is absent from main memory.

M = Mode bit, set to 1 if segment is to be executed in privileged mode.

T = Trace bit, set to 1 if trace feature is used. This is checked by the PCAL instruction.

R = Reference bit, set to 1 when this segment is accessed.

Length = The length of the code segment. Note: 14 bits are allowed for Length - 1. The maximum segment size, therefore, is 16,384 words.

VATBI = Disc volume table index.

Address(CST2) = Hi order bits of absolute memory address of PE if A = 0, or hi order bits of disc address if A = 1.

Address(CST3) = Lo order bits of absolute memory address of PB if A = 0, or lo order bits of disc address if A = 1.

Data Segmentation

Functional Description

The indirect cell of a memory reference instruction or the pointer in other instructions contains a DATA LABEL in one of the defined formats. This label specifies in which data area the data resides and the relative location within that data area. If the label points to a data segment, the data segment number is used to index into the data segment table to get the location of the segment. The offset and index register then are used to compute the relative address of the location of the data within the segment. If this relative address is greater than the segment length, a check is made to determine if the segment belongs to a set of linked segments. If so, sequencing through the set of linked segments is pursued according to the type of linking until the desired address is found.

Data Labels

Stack Data Label

References to data in the stack (SB to S) use the stack relative data label.

```

-----
* 0 0          OFFSET          *
-----
0, 1, 2-----15

```

OFFSET = word or Byte Displacement in the stack relative to the base of stack (SB register). The fourteen bit field allows a maximum stack size of 16K bytes (8K words).

Global Data Label

References to data residing in the global data area (DB to DL) use the global data label.

```

-----
* 1          OFFSET          *
-----
0,1-----15

```

OFFSET = word or Byte Displacement from the base of the global data area (DB register). The fifteen bit field allows a maximum size of 32K bytes (16K words).

Labels for Data Segments

Reference to an array residing in a separate data segment is made with one of two array data labels.



DST Label

If the segment contains only one array, reference to the base of the array may be made with this one-word DST label.

```

-----
* 0 1 0           DSEG          *
-----
0,1,2,3-----15
  
```

DSEG = Entry number of the data segment in the data segment table. Indexing is relative to the base of the array.

Data Segment Label

References to specific elements in an array or to segments containing multiple arrays use this two word label.

```

-----
* 0 1 1           DSEG          *
-----
0,1,2,3-----15

-----
*           OFFSET           *
-----
0 -----15
  
```

DSEG = Entry number of the data segment in the data segment table.

OFFSET = word or Byte Displacement from the base of the data segment.

Indexing is relative to the offset field.

Data Segment Table & Data Segment Table Pointer

The Data Segment Table contains a list of data "spaces" used by the system and user programs, which are currently resident in virtual memory. These spaces include such things as I/O buffers, system & user stacks, and user data segments. It consists of a number of 4-word segment descriptor entries where the actual number of entries in use at any point in time is a variable, limited only by the maximum length of the Table. Entries are dynamically allocated by the system as jobs are initiated and terminated, or as certain classes of active users request or release additional data segments. Each entry contains control information about the segment, and gives its length and starting address.

For the purpose of data management the DST is divided into two sections much the same as has been done with the CST. The first section is reserved for system data segments. The first word of the first entry in this table is located at the absolute memory address %1000.

segment 0 is unused by the system, and hence its 4-word entry may contain other information. The first word of this entry contains one less than $4 * \text{the count of the current maximum number of entries (excluding seg. 0) in the first section of DST}$. The remaining three words of this entry are reserved. NOTE: Since microcode does not make a special test for Segment 0, the absent bit for this entry must be set by software so that an illegal access to this segment will be caught by the absence trap software.

The second section of the DST, starting at segment number %10000 (4096), is used for user data segments. The first word of the first entry in this section of the table is located at the memory address contained in the DST register. As in the first section the first word of this section contains one less than $4 * \text{the count of the maximum number of segment pointer entries in the second section (excluding segment \%10000)}$. The rest of the first entry is reserved.

If the PRIV mode bit in the DST entry is set, access to that segment is allowed only in privileged mode.

The entries of the DST have the following form:

	0	1	2	3	4	-----	15
DST0	*				Length - 1		*
DST1	* C *	0	0	0 *	Link DST#		*
DST2	* A *	M	*res*	R	* D *	res* VTABI(8:4)/Addr(8:8)	*
DST3	*				Address		*

Length = Maximum length of this segment. Note: the maximum length allowed is 65K bytes or 32K words.

C = Consecutive Link bit, set=1 when this descriptor is part of a set of consecutively linked descriptors pointing to a single multiple segment array. Consecutive linked segments must all be of equal size and of length $N \times 2$ where N is an integer from 2 to 16.

Link DSI# = Linked DSI entry, set non-zero to point to additional descriptors in the set of multiple segments assigned to a single logical data label. If C=0, the field contains the DSI number of the next link. If C=1, the field contains the DSI number of the LAST DSI entry in the family of consecutively linked DSI's. A zero field indicates no further linked entries.

A = Absence bit, set=1 if the data segment is absent from main memory.

P = Privileged mode bit, set=1 if access to this segment is allowed only in privileged mode.

D = Dirty bit, set=1 when the data segment has been altered.

R = Reference bit, set=1 when this data segment has been accessed.

res = Reserved for software.

VTA3I = Disc volume table index.

Address(DSI2) = hi order bits of absolute memory address of the base of the data segment if A=0, or the hi order bits of the disc address if A = 1.

Address(DSI3) = Lo order bits of absolute memory address of the base of the data segment if A=0, or the lo order bits of the disc address if A=1.

formats

Data formats

fixed point format: 16 bit, two's complement

```

-----
* S *                               *
-----
0 , 1-----15

```

(For double-word fixed point format, 16 magnitude bits are added on the right).

Logical format, 16 bit positive integer

```

-----
*                               *
*                               *
-----
0-----15

```

Byte string

*0	7*8	15*	Amigo
*7	0*7	0*	ASCII

Byte 0	Byte 1		Increasing
-----*			Memory
Byte 2	Byte 3		Address

Note: The bit #'s within an ASCII byte definition are in the reverse order to the normal Amigo bit #'s (i.e. they consider bit 0 to be the LSB of the byte).

Floating point format, sign + magnitude representation

```

-----
* S *   E   *           F           *
-----
0 ,1-----9,10-----31

```

S = Fraction sign bit (0 for positive, 1 for negative)

E = Exponent+256 (Range 0 to 511)

F = Fraction (Range 0 to $(2^{22}) - 1$)

Note: In sign + magnitude representation, the fraction is always positive with the S bit containing the sign of the number. The binary point is assumed to the left of bit 10 with an implied leading 1 to the left of the binary point. Thus all floating point numbers are stored in normalized form, but no bit is wasted on the leading 1, making all fraction bits significant. The exception is that 0 is a word containing all 0's.

$$\begin{aligned}\text{Decimal value} &= (-1)^S * 2^{(E-256)} * (1 + F * 2^{(-22)}) \\ &= 0 \text{ when } S = E = F = 0\end{aligned}$$

Note the following special cases:

$$\begin{aligned}0 \text{ 000000000 000000000000000000000000} &= 0 \\ 0 \text{ 100000000 000000000000000000000000} &= 1\end{aligned}$$

Double floating point format, sign + magnitude representation

$$\begin{array}{c} \text{-----} \\ * \text{ S } * \quad \text{E} \quad * \quad \text{F (n1)} \quad * \\ \text{-----} \\ 0, 1 \text{-----} 9, 10 \text{-----} 31, \\ \text{-----} \\ * \quad \text{F (l0)} \quad * \\ \text{-----} \\ 32 \text{-----} 63 \end{array}$$

S = Fraction sign bit (0 for positive, 1 for negative)

E = Exponent + 256 (Range 0 to 511)

F = Fraction (Range 0 to $(2^{54}) - 1$)

Note: The same hidden 1 leading bit in the fraction F is used as in the single floating point format.

$$\begin{aligned}\text{Decimal value} &= (-1)^S * 2^{(E-256)} * (1 + F * 2^{(-54)}) \\ &= 0 \text{ when } S = E = F = 0\end{aligned}$$

Definitions

$\text{rfloor}(x)$ = largest integer $\leq x$
 $\text{sign}(x)$ = 1, if $x > 0$
 0, if $x = 0$
 -1, if $x < 0$
 $\text{exp}(x)$ = $\text{rfloor}(\log_{\text{base}2} x)$, if $x \neq 0$
 undefined, if $x = 0$
 $\text{trunc}(x)$ = $\text{sign}(x) * \text{rfloor}(x)$
 $\text{round}(x, N)$ = 0, if $x = 0$
 $2^{**}(\text{exp}(x) - N) * \text{sign}(x) * \text{rfloor}(2^{**}(N - \text{exp}(x)) * x + 0.5)$, if $x \neq 0$
 where $N+1$ is the number of significant bits.

Packed Decimal Data

	0	Bytes	7

Low Byte Memory Address	Dn	:	Dn-1
(Most Sig Digit)	-----		
	Dn-2	:	Dn-3

	:		
	:		

	D3	:	D2
(Least Sig Digit)	-----		
High Byte Memory Address	D1	:	+-

The above is a representation of a decimal numeric number showing n digits and the position of the sign. Each D_i must be in the range !0 to !9. All other values are illegal. The values for the sign digit +- are as follows. !0 - !C are positive. !D is negative. !E is positive and !F is absolute. n may be 0 to 28, and is specified by a parameter on the stack. The sign MUST be in the right nibble of a byte (nibble = 4 bits).

Zoned Decimal Data

	0	Bytes	7

Low Byte Memory Address		Cn	
(Most Significant Byte)		-----	
		Cn-1	

		Cn-2	

		:	
		:	
		:	

		C2	
(Least Significant Byte)		-----	
High Byte Memory Address		C1	

	0	Bytes	7

Zoned Decimal Data is represented as a byte string of characters with the following properties:

- (1) Blanks may be present at the leading end of the string only.
- (2) Following the first non-blank character, all characters including this non-blank character must be the characters "0" thru "9", except for C1 (the last character).
- (3) the character C1 may be any one of the characters in the following table. This character specifies the value of the least significant digit and the sign of the number.

Internal Representation ASCII			
Units digit	Positive	Negative	Absolute
0	{ %173 !7B : }	%175 !7D	0 %060 !30
1	A %101 !41	J %112 !4A	1 %061 !31
2	B %102 !42	K %113 !4B	2 %062 !32
3	C %103 !43	L %114 !4C	3 %063 !33
4	D %104 !44	M %115 !4D	4 %064 !34
5	E %105 !45	N %116 !4E	5 %065 !35
6	F %106 !46	O %117 !4F	6 %066 !36
7	G %107 !47	P %120 !50	7 %067 !37
8	H %110 !48	Q %121 !51	8 %070 !38
9	I %111 !49	R %122 !52	9 %071 !39

ASCII Sign Digit Representation

note: Zoned Decimal Data is used only by CVAP,CVPA,EDIT instructions.

Stack Marker Format

```

-----
*           X           *
-----
* T *  RELATIVE P      *
-----
*           STATUS      *
-----
*           DELTA Q      *
-----

```

X = Contents of the Index Register
 RELATIVE P = P+1 - PB (The instruction to be returned to)
 STATUS = The Content of the Status Register.
 DELTA Q = Value to be subtracted from the Q register to obtain the Q value of the caller.
 T = Trace bit for exit instruction (Set by software).

The stack marker is pushed on the stack and Q is moved to the delta Q entry of the stack marker by the procedure call instruction. This saves the calling environment and establishes a new addressing base for the called procedure. Thus the stack marker provides the means for restoring the calling environment on return from the called procedure.

Tables

There are several classes of tables resident in the Amigo operating system. They are classified below:

NAME	MNEMONIC	LENGTH	WHERE
System Code Seg. Table Pntr.	SCSTP	1	Fixed location 0
User Code Seg. Table bank pntr.	UCSTP1	1	Fixed location 1
User Code Seg. Table Pntr.	UCSTP2	1	Fixed location 2
Reserved		1	Fixed location 3
Current TCB Pointer	CTCBP	1	Fixed location 4
Interrupt Stack Base	QI	1	Fixed location 5
interrupt Stack Limit	ZI	1	Fixed location 6
Interrupt Mask	MASK	1	Fixed location 7
Device Reference Table	DRT	<=120	Fixed loc. %40-%777 4 word entries
System Code Segment Table	SCST	<=192	Loc. at (SCSTP) 4 word entries.
User code Segment Table	UCST	<=64	Loc at (UCSTP) 4 word entries.
System Data Segment Table	SDST	var	Fixed loc. %1000 4 word entries.
User Data Segment Table	UDST	var	Loc. at (DST reg.) 4 word entries.
Segment Transfer Table	STT	<=256	One table per code segment. # of entries & length fixed at load time.
Task Control Block	TCB	var	Dynamically allocated

The entries in these tables are described in the following paragraphs.

System Code Segment Table Pointer and System Code Segment Table.

The system CST pointer is a 16 bit word at memory location zero that points to the base of the system portion of the CST table. This table contains up to 191 four word entries, one for each active system code segment.

User Code Segment Table Pointer and User Code Segment Table.

The User CST pointer is a 24 bit word at memory locations 1 and 2 that points to the base of the user portion of the CST table. This table contains up to 63 four word entries, one for each active user code segment.

User Data Segment Table Pointer and User Data Segment Table.

The user DST pointer is the DST register that points to the base of the user Data Segment Table. Software treats this as a 24 bit register to provide for expansion in later hardware versions. This table contains up to 4095 four word entries, one for each user data segment assigned space in virtual memory.

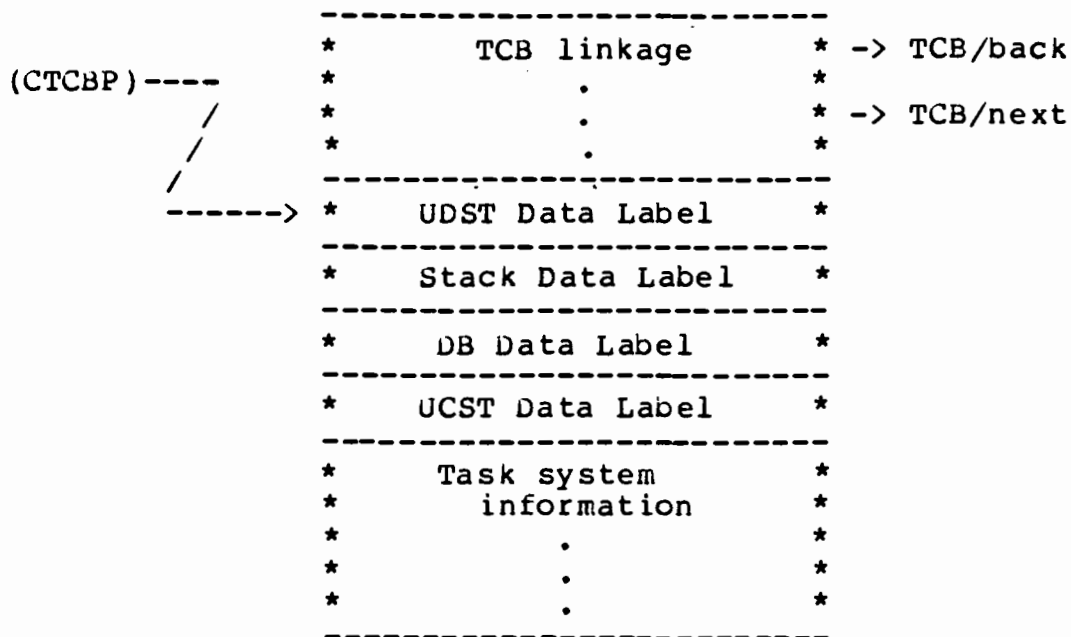
System Data Segment Table

The system data segment table resides in a fixed location starting at the memory address \$1000 and contains up to 4095 four word entries, one for each system data segment assigned space in virtual memory.

Current Task Control Block Pointer and Task Control Block

The current task control block pointer is a 16 bit word at memory location 4 that points to an entry in the task control block. This entry contains DST labels that point to the current executing task environment as described below in the diagram and label explanations. This table entry also is used by the dispatcher using the IXIT instruction to set up and launch a new task environment. A task control block exists for each task currently defined and contains operating system information related to the task.

The format of the Task Control Block is as follows:



UDST data label = label that points to the User data segment table. (Note: this must be in sys DST)

Stack data label = label that points to DST entry having the present location of the current task stack.

DB data label = label that points to DST entry having the present location of the current task DB data area.

UCST data label = label that points to the User code segment table.

Stack Segment Structure

The stack segment contains the user data stack delineated by the stack registers SB, Q, S, and Z, and it contains information used in the system. The hardware defined structure is shown in the following diagram.

Stack Segment

(Stack Data Label)-

```

      /----->
      /-----> * SB(seg. base relative) *
      /-----> *   Z (SB relative)         *
      /-----> *   ZMAX (SB relative)      *
      /-----> *           .               *
      /-----> *           .               *
      /-----> *           .               *
      /-----> *   S, Q (SB relative)      *
      /----->
Register SB --> *
      /-----> *
      /-----> *           .               *
      /-----> *           .               *
      /-----> *
      /----->
Registers S,Q --> *   Delta Q of Stck Mrk   *
      /-----> *
      /-----> *           .               *
      /-----> *           .               *
      /----->
Register Z --> *
      /-----> *
      /-----> *   Stck overflow space   *
      /----->
ZMAX -->

```

Device Reference Table (DRT)

This table contains one 4-word entry for each I/O device number, up to a maximum of 120 devices. The first word of each entry corresponds to a unique device and contains the address of the next I/O channel command for that device. The second word contains the address of the variable used by the channel command in word 1. The third word contains a program label that points to the interrupt handler for the device. The fourth word is used for Channel status by the Channel processor. Refer to the I/O ERS. Since only one word is reserved for their addresses, channel command programs and their variables must reside in memory bank zero. Note that device numbers 0 thru 7 do not exist since these addresses are used for dedicated system functions. The format of the entries is as shown in the following diagram.

```

-----
4 * DEV.#--->*   Channel Program Pointer   *
-----
*             Interrupt DB Pointer         *
-----
*             Interrupt Handler Label      *
-----
*                      reserved            *
-----

```



Interrupt Stack Base and Limit

These memory locations point to the start and end of the memory area allocated to a system stack called the interrupt control stack. Whenever an interrupt occurs, external or some internal, the current process is suspended and the stack registers are changed to this stack.

Interrupt Mask

This location contains a copy of the status of the I/O Channel mask bits. A bit set in this word indicates that interrupts are enabled on the channel assigned to that bit position.

I/O System

In this system I/O devices are controlled by means of hardware channels which interface to the system bus and which operate under control of a special channel instruction program that resides in main memory. Channel programs are initiated or halted by the Start I/O (SIOP) and Halt I/O (HIOP) software instructions. The SIOP instruction is issued to a device to initiate execution of a channel program which, when started, can proceed independent of CPU control. A channel program in progress may be suspended by issuing a HIOP instruction to the device which will halt the channel program of that device when it encounters the next "WAIT" channel instruction. This allows the channel program to be changed or modified with safety for such things as swing buffering. When halted, the first word of the DRT entry points at the channel WAIT instruction just interrupted.

There are two types of I/O processor hardware channels in the I/O system: The General I/O Channel which requires active CPU control of every function of its HP-IB bus, and the Program Interpretive Channel that is structured to autonomously and in parallel with normal CPU operation, execute channel programs.

General I/O Channel - GIC

This channel requires outside control to execute its function and two methods of control have been provided. In the first mode normal channel programs can be executed by means of a special CPU interrupt mode which, without disturbing the CPU software environment uses special microcode to issue the required instructions. In the second mode, special I/O instructions are provided to allow software to exercise direct control over this channel. These instructions are referred to as "direct I/O" instructions in that they involve transfers (data, status, command) between the device and the TOS. The CPU waits until the operation is complete before proceeding to the next instruction.

Program Interpretive Channel

The program interpretive channel has self contained hardware which, when initiated by the CPU with an SIO instruction, can execute channel programs without further help from the CPU. This allows CPU software programs and channel programs to function concurrently.

SIO Programming

An SIO transfer is initiated by the CPU executing a Start I/O instruction for the desired device. It is assumed that there is an I/O Channel Command program stored in main memory.

The I/O Command program consists of a set of instructions and a set of variables associated with these instructions to control the transfer of data between device and memory.

Interrupt and Trap Processing

External Interrupts

External Interrupt Structure

The external interrupt structure is a two level priority structure with the first level at the channel and the second level at the device within the channel.

Servicing of external interrupts is done in descending order of priority, i.e. the highest priority interrupt is serviced first. The interrupt priority of a device is determined first by the channel number to which it is attached and second by the device number it is assigned in the channel. The interrupts structure is nested such that a higher priority channel can preempt a lower channel. Within the channel however this preempting capability does not exist. A 16 bit Mask is provided for the purpose of masking off different channels. It is the responsibility of the interrupt handler, using the SMSK instruction, to clear the mask bits of the channels it wishes to deny interrupt capabilities before enabling the interrupt system with ION. This mask is saved in a dedicated main memory location.

Recognition of an interrupt request is enabled by STA (1) (the "External Interrupts" bit). When this bit is cleared the entire interrupt system is turned off.

Interrupt Response Time

The interrupt response time is defined to be the maximum time that may elapse between the interrupt request and the start of the execution of first instruction of the interrupt routine for the highest priority unmasked interrupt. This response time is typically 25 microseconds. Worst case time depends on the length of time of the longest instruction and the length of the command portion of a channel program that is executing.

Interrupt Sequence

An interrupt will be activated when the following conditions exist.

- a. The channel has recognized the device request, loaded the interrupt status of that device into its data register and set its interrupt request flip-flop.

- b. Mask bit is set for that channel.
- c. External interrupts bit in CPU STATUS word is set.
- d. No higher priority channel is requesting service.

Upon receipt of an external interrupt by the CPU, and upon completion of the currently executing instruction, a standard 4-word stack marker is created on the current stack. If the CPU is not already executing on the Interrupt Control Stack (ICS) (i.e. due to a previous lower priority interrupt), the relative value of the current S is saved at (SB-1). The hardware ICS flag is set and the stack environment is changed to the Interrupt Control Stack by setting:

```

Q := QI  <<from fixed memory location, address 5>>
Z := ZI  <<from fixed memory location, address 6>>
SB := QI - %100
S := Q+3
(S-2) := DB of the stack marker environment at Q.
(S-1) := Current Interrupt Mask
(S) := Device number of external Interrupt.

```

(Note that operating system guarantees (QI(1:15)) = 0)

The CPU then fetches the interrupt handler label from the third (4*DEV.No.+2) entry in the DRT for this device number. The program label may have either a local or external format. If local it is then interpreted as an absolute address with PB set to 0, PL set to %177777 and the Status Register set to segment zero. If external, the standard external program label evaluation then is executed. The status register is set to privileged mode, external interrupts disabled.

Execution of the code pointed to then is begun.

The steps in the External Interrupt processing described above are shown in the following table:

MEM REF	ACTION FOR CPU
0	An External Interrupt request is received by the CPU and detected upon completion of the current instruction.
1	A normal four word stack marker is created and pushed into memory, saving X, P + 1 - PB, STATUS and Delta Q.
2	
3	
4	
5	
6	The current value of the user's SB relative stack pointer is stored in (SB-1).
7	Set up ICS with Q=QI, Z=ZI, and ICS Flag=1 and
8	Set SB=Q-%100, S=Q+3 <<Q+1 contains Dispatcher DB>>
9	Read interrupt mask from address 7
10	Store interrupt mask in (Q+2)

- 11 Store device number in S. <<Q+3>>
- 12 Set interrupt DB from (DRT1) for device
- 13 Fetch interrupt handler label from DRT2 entry for device.
- 14,...,18 Set up code environment from label. Set Status to Priv. mode, External Interrupts disabled, and pointing to proper segment number.
- 19 Begin executing code.

Note: A fall-out from the firmware interrupt handler is that (CIR) is put in the X-reg. before servicing the interrupt. This may be of some use in such things as performance measurements. Exception: an interrupted IXIT instruction will have CIR = 0.

The table also shows the Items stacked when a higher priority device interrupts the current interrupt service routine. X, P+1-PB, STATUS, Delta Q are stacked and steps 7 through 12 are followed.

External Interrupt code segments must not reside in seg. 1. This segment is reserved for internal interrupts and traps, and is handled in firmware somewhat differently than the remaining segments.

As mentioned above, STA(1), the External Interrupts bit, inhibits the servicing of external interrupts. Pending interrupts are not lost; they remain pending and are serviced when ION sets STA(1:1) to 1 again.

Interrupt Control Stack, ICS Flag, and Dispatcher Flag

The interrupt control stack is defined by absolute locations 5 and 6. Note that this requires the ICS to reside in Bank 0. These locations contain the value of Q and Z, respectively, for interrupt processing. The current process is said to be running on the interrupt stack if the interrupt stack flag is 1. This flag is observable only on the microprogram level. It is set by the firmware interrupt handler when the pending interrupts are to be serviced on the interrupt stack, (or when executing the DISP instruction and not already on the ICS). These include external interrupts and the internal interrupts with executing stack designated ICS in the interrupt table below. The ICS flag is tested by the "DISP" instruction to control its execution, and cleared, when appropriate by the "IXIT" instruction.

when execution is transferred to the ICS the firmware interrupt handler sets the ICS flag =1, and then sets $Q=QI$, $SB=QI-100$ and $Z=ZI$ (Note: $(QI(1:15)=0)$, set by the operating system denotes the bottom of the stack). S is set to $QI+3$ to point to the parameter location (e.g. the device number for external interrupts.- see table of internal interrupts for other parameters). The proper code segment then is set up, and servicing of the interrupt begins.

when all interrupts have been serviced (noted by $(Q(1:15)=0)$) the `IXIT` instruction is processed to determine where to go next (note that all interrupt service routines must end with `IXIT`). `IXIT` has several options open to it depending on the state of the dispatcher flag (see below), (Q) , $QI(0)$, and $(QI-4)$. These include returning to the interrupted process, entering the dispatcher, either at the beginning or where it was interrupted, or returning to an interrupted interrupt process.

The dispatcher flag is a hardware flag, observable only by the firmware, which is set = 1 whenever dispatcher code is entered. The stack used is the ICS, set up as mentioned above, except S is set to $QI+3$.

Internal Interrupts and Traps

Internal interrupts and traps are serviced by procedures in Segment 1. Some use the current users stack and others transfer to the ICS stack and are serviced as described earlier. Except for Power Fail, Stack Overflow and `TIMER` interrupts which are not detected until the completion of the current instruction, interrupts abort the current instruction immediately upon detection and then are serviced as follows:

- a. An External Program Label is created which points to Segment 1, and whose STT No. is a function of the type of interrupt (see Internal Interrupt Table below).
- b. A standard 4-word stackmarker is pushed onto the stack in memory.
- c. A parameter is pushed onto the stack at $Q+3$. If none is required by the interrupt (see Table), the parameter will be its External program Label. The current instruction is placed in the X-register.
- d. Interrupt condition is cleared.
- e. Segment 1 PB and PL is set up from the CST.
- f. STATUS word is set to Priv. Mode, Seg. 1, all other bits are cleared (= %100001).

g. P is set from the local label, reached via the STT entry in the External Program Label.

n. Begin code execution.

User Traps, labeled * in the following table, are enabled by the user traps bit (STATUS(2)). If this bit is off when a trap is encountered, the overflow bit in the STATUS word is set and no trap is taken.

Traps differ from interrupts in that they are conditions detected by the firmware while executing instructions, as a result of something that happened during execution. Interrupts are external to instruction execution and are detected after the instruction has been completed. Power ON and Cold Load are special cases.

INTERNAL INTERRUPT AND TRAP TABLE

External Prog. Label (%)	STT No. (%)	Interrupt Type	Parameter (Q+3)	Executing Stack
100401	1	Bounds Violation	**	***
101001	2	Unused		
101401	3	CPU Non-Responding Module		
102001	4	Channel Non-Responding Module		
102401	5	Unused		
103001	6	CPU Mem. Parity Error		ICS
103401	7	Unused		
104001	10	Unused		
104401	11	Power Fail		ICS
105001	12	I/O Instruction Failure		
105401	13	Data Segment Bounds Violation		
106001	14	System Soft Halt		
106401	15	Ixit Launch Error		ICS
107001	16	Unused		
107401	17	Debug		
110001	20	Unimplemented Instruction		
110401	21	STT Violation		
111001	22	CST Violation		
111401	23	DST Violation		
112001	24	Stack Overflow		
112401	25	Stack Underflow		
113001	26	Privileged Mode Violation		
113401	27	Priv Mode DST Violation		

114001	30	Data label overflow	
114401	31	*User Traps (see below)	Integer
115001	32	Unused	
115401	33	Unused	
116001	34	Unused	
116401	35	Unused	
117001	36	CPU Timer Interrupt	ICS
117401	37	Absent Code Segment	
		-on PCAL,BR external	P-Label
		-on IXIT,EXIT	N << =0 for IXIT>>
120001	40	Trace-on PCAL,BR xtrn	P-Label
		----on IXIT,EXIT	N
120401	41	STT Entry Uncallable	P-Label
121001	42	Absent Data Segment	DST No.
121401	43	Power On	ICS
122001	44	Cold Load	Dump History ICS

* Enabled by "User Traps bit" in STATUS word.

** Parameters are as noted.

*** Unless noted, Interrupts are serviced on the current stack.

(CIR) is placed in the x-reg. prior to entering Seg. 1. This will be used principally by the "Unimplemented Instruction" routine.

All User Traps will be executed in the Seg. 1 routine pointed to by STT No. 31. Bits in the parameter word have been reserved to designate the trap condition as follows:

Bits 0-7	zeroes	- denotes a hardware trap
	1-255	- an integer supplied by the Math Library identifying the function which encountered the error.
Bits 8-11	Trap Code	0 = Overflow 1 = Underflow 2 = Invalid Argument (divide by zero) 3 = Invalid Character 4 = invalid Control Word 5-15 = unused
Bits 12-15	Result Type	0 = INTEGER 1 = DOUBLE 2 = REAL 3 = EXTENDED

4 = LOGICAL
5 = COMPLEX
6 = ASCII
7 = BOOLEAN(T/F)
8 = DECIMAL
9 = BINARY
10-15 = unused

Note: Result types were selected to comply with the data types of the FORMATTER.

Internal Interrupt and Trap Definitions

STT#	Definition
----	-----
1.	Bounds Violation. An address is outside of the legal bounds for that addressing mode (e.g. P>PL).
2.	Unused.
3.	CPU Non-Responding Module. Information requested by the CPU from some other module (e.g. memory) has not been returned within a specified time (the "specified time" is hardware controlled by a CPU timer circuit, and is normally 4.6 ms).
4.	Channel Non-responding module. Information requested by the CPU when executing channel programs has not been returned as in 3 above.
5.	Channel memory parity error. During execution of the command sequences in channel programs by the CPU a parity error has been detected on the 16-bit data word read from memory.
6.	CPU Memory Parity Error. During execution of software programs by the CPU a parity error has been detected on the 16-bit data word read from memory. On the "write" cycle, where the CPU sends data to memory, memory generates parity and stores it with the word. Normal memory use 1 parity bit and error correcting memory uses 6 bits. Error checking is done on data by the memory in the read cycle. when an error is detected on a memory read , the CPU immediately terminates the instruction and commences internal interrupt processing.
7.	Unused.

10. Unused.
11. Power Fail. A power failure has been detected.
12. I/O Instruction failure. An I/O instruction to a device # that is not in the range 10 to 177 has been attempted.
13. Data Segment Bounds Violation. An access has been attempted outside the bounds of a Data Segment or a set of linked Data Segments.
14. Soft System Halt. A system error condition has occurred where processing cannot continue but a message can be sent to the console identifying the system error. The following errors have been defined:
 1. Too many PSEB's
 2. Absent data segment when on ICS
 3. Absent code segment when on ICS
 4. ICS Stack Underflow
 5. Error when processing internal interrupt
 6. Exit underflow on ICS
 7. UCST PTR = 0
15. IXIT Launch error. Bad labels in CTCB or invalid user stack has been encountered. System stays on ICS.
16. Unused.
17. Debug. The debug instruction has been executed.
20. Unimplemented Instruction. The CPU has detected a bit pattern in its "Current Instruction Register" which is not a valid instruction.
21. STT Violation. A variety of conditions can force this trap as shown below:
 1. The STT No. in an external program label is greater than the STT length (pointed to by PL) in the referenced segment. This error can occur in PCAL, BR-external, and the Interrupt Handler, while attempting to set up a new segment.
 2. In LLBL, if the label fetched from PL-N is an internal label and N>128 (>177), the trap is invoked (this would require too large an STT No. when creating the external label).
 3. In PCAL, BR-external or the interrupt handler when setting up a new segment, if the STT No. in the ext. program label points to and ext. program label in the new segment, the trap is invoked.

22. CST Violation. An attempt is made to transfer to seg. 0 or \$300, or a segment number referenced through an external program label is greater than the CST length. The instructions involved here are PCAL, BR-external, EXIT, IXIT, DISP, and the firmware Interrupt Handler.
23. DST Violation. The DST segment no. referenced is greater than the number of entries contained in DSTL (1st word of DST). Note: entry 0 or 4000 are caught by software in absence handler.
24. Stack Overflow. S is greater than Z. Except for ADDS & SUBS, instructions are allowed to complete before the interrupt.
25. Stack Underflow. An attempt to move S less than Q. The trap occurs without moving S.
26. Privileged Mode Violation. The code segment being executed is non-privileged (STATUS(0) = 0), and an attempt is made to execute a privileged instruction. This violation also occurs in EXIT, if an attempt is made to exit from USER to PRIV. mode, or if exiting from USER mode an attempt has been made to alter the Ext. Interrupts in the STATUS register.
27. Privileged Mode Data Segment Violation. A data segment with the mode bit in the DST set to privileged has been referenced in user mode. (STATUS(0:1) = 0).
30. Data label overflow. Indexing of stack or global data label has caused overflow into label identifier bits.
31. User Traps. One of the following errors has occurred:
 - a. Integer Overflow. The result of an integer operation (ADD, SUB, etc.) exceeds the largest representable integer. The range of allowable integers is -32768 to +32767.
 - b. Floating Point Overflow. The result of a floating point operation is greater than the largest representable number.
 - c. Floating Point Underflow. The result of a floating point operation is less than the smallest representable number, but not equal to zero.

Note: Floating-point overflow and underflow can best be understood by referring to the chart below showing the range of valid numbers:

A	B	C	D	E
---	-----	---	---	----- ---

Valid Numbers N occur when

(A < N < B) or (D < N < E) or (N = C);
 If (N < A) or (N > E) then OVERFLOW;
 IF ((N > B) or (N < D)) and (N <> C) then UNDERFLOW;

where

A = -1.1579×10^{-77} /
 B = -8.63616×10^{-78} /
 C = 0.0
 D = $+8.63616 \times 10^{-78}$ /
 E = $+1.1579 \times 10^{-77}$ /

- d. Integer Divide-By-Zero. The divisor in DIV, DIVI, DDIV, LDIV is equal to zero.
 - e. Floating-Point Divide-By-Zero. The divisor in an FDIV or an EDIV instruction is equal to zero.
 - f. Invalid Character. An invalid ASCII or DECIMAL character has been encountered.
 - g. Invalid Control word/ASCII or DECIMAL
 - h. Decimal Divide-by-Zero.
- 32. Unused.
 - 33. Unused.
 - 34. Unused.
 - 35. Unused.
 - 36. CPU Timer Interrupt. A match between the Clock and the Clock Comparison Register has occurred causing a timer interrupt request when the timer is enabled.
 - 37. Absent Code Segment. The absence bit in the CST entry for the referenced segment is set = 1. This is checked in PCAL, BR external, EXIT, IXIT, DISP and the firmware Interrupt Handler. Note EXIT, IXIT, DISP, a new marker is not stacked.
 - 40. Trace. Non-local PCAL and Ext. Interrupts check the trace bit in the CST entry for the referenced segment. EXIT and IXIT check P(0) in the marker stacked by PCAL or the Ext. Interrupt (this bit is set by the trace routine software if it is desired to trace exits). In either case, if the bit tested = 1, the trace routine is entered. For PCAL's, BR-external's and Ext. Interrupts, another target marker is stacked first which is used by the EXIT from the trace routine. For EXIT and IXIT no marker is stacked; hence +

P(0) of the last marker stacked (prior to EXITing from trace) must be cleared by software in the trace routine. Otherwise an infinite trace loop could occur. Tracing Segment 1 traps result in a system halt. Tracing External Interrupts or the Dispatcher requires special software in the trace routine due to the differences in EXIT and IXIT.

41. STT Entry Uncallable. The uncallable bit in a local label (or in (PL) if STT# = 0) is set =1 and this label is referenced by a PCAL or BR-external from another segment running in user mode. This trap does not stack a new marker for PCAL.
42. Absent Data Segment. The absence bit in the DST entry for the referenced segment is set = 1.
43. Power On. Power has been applied to the machine, and the supplies have stabilized.
44. Cold load. The switch register number read on cold load is used to select the cold load device and then to enter the Segment 1 cold load program entry.

Special System Functions

Power-Fail/Power-On

The machine will detect a power-fail if the primary power is lost for a period longer than 6 ms. The CPU, in conjunction with the machine power supplies, will then guarantee a period of 5 ms for processing a power-fail routine in Seg. 1. Following this, a delay period is generated in the event the power-fail was momentary. This period is used to reset the CPU and to allow recovery of the power supplies. If power is restored by the end of this period, the POWER-ON firmware is entered.

When a power-failure is detected, the CPU inhibits all further interrupts. Then the power-fail routine in Segment 1 is set up, and this code begins execution.

When power is restored, microcode checks memory for parity errors and checks the contents of memory address 0 for all 1's or all 0's and finding none of these assumes memory is good. The ICS then is set up from low memory, the power-on routine in segment 1 is set up, and execution of the power-on routine begins. If any of the above error conditions were encountered when power was restored, memory is initialized and the cold load procedure then is instituted.

Cold-load

A special firmware program has been provided to read a start up environment from back up memory into main memory. The back up memory device # is read from the panel switch register. The contents of address %37 (possible dump address) is read and saved and then memory is loaded. The register environment is set up to the ICS stack and the saved dump parameter is passed to software at QI+3 to provide the dump disc address if a DUMP has preceded the COLD LOAD. S is set to QI+3 and the switch indicated Cold Load SIT entry in Segment 1 is entered. This cold load facility also is available through the software IPL instruction in which the load device # and cylinder address are specified in the TOS.

System Dump

A firmware program dumps the first 65K of memory to the dump device read from the panel switch register. During the dump of the second 32k the contents of all the CPU registers are stored in memory starting at address %16. The dump cylinder address and device # also are stored at address %37. Then the first 128 words in memory (this time containing the register values) are dumped to the next sector on the disc. The software SDMP instruction also performs the same action but takes its device # and cylinder address from the TOS. Refer to the SDMP instruction for the register locations in memory. The cold load microcode retrieves the contents of address %37 before the load so that if the SDMP instruction is followed immediately by an IPL instruction, this information will be passed to the cold load software at Q+3.

System Clock

The CPU contains a nonsettable 16 bit counter which may be read by a software instruction. It starts at a random count when power is turned on and is continuously incremented as long as power is applied to the system. The clocking interval is 2×10 internal CPU clock intervals and the range of the clock before rollover is approximately 6 seconds. A settable register and comparison circuit also is provided which generates an interrupt request when the contents of the clock and register are equal and Timer interrupts are enabled.

System Halt

In cold load or dump or in executing code certain error conditions are encountered that make further CPU processing impossible. In these cases the CPU halts with the run light +

steady off (not flashing as in software halt) and the lights of the NIR register on the CPU PC board set to indicate which condition caused the HALT. These conditions are as follows:

- (0) Stack overflow on the Interrupt Control Stack.
- (1) Code Segment Table Ptr at address 0 equals zero.
- (2) Unable to setup segment 1 for an internal interrupt.
- (3) Unassigned.
- (4) Cold load/dump error - not load or dump device.
- (5) Dump error - not a dump device.
- (6) Cold load error - DMA read error
- (10) Dump error - Disc data error
- (13) Dump error - failure to write 1st 32k words.
- (15) Dump error - failure to write 2nd 32k words

System halt numbers other than those above are cold load or Dump disc error messages.

Section 5 Central Processor Instructions

Definitions used in this section for memory reference instructions:

E = Effective word address
 PDQS = P+, P-, DB+, Q+, Q-, S-
 DQS = DB+, Q+, Q-, S-
 P = P+, P-
 * = Indirect addressing allowed
 D = Displacement field of instruction word used in address computation. It is always a word displacement even in byte and double word instructions.
 L = Displacement field of Conditional Branch instructions. The range is $P - 31$, $P + 31$.
 X = Indexing allowed (Note: X can also refer to the contents of the index register).
 Indexed = $CIR(4:1) = 1$
 Xc = If indexed $\langle\langle CIR(4:1)=1 \rangle\rangle$ then (X) else 0.
 fXc = If word instruction then Xc else

If byte instruction then $Xc/2$ else
 If double word instruction then $2*Xc$
 If quad word instruction then $4*Xc$.

Indicators

CCA = Condition Codes set according to definitions in Section 4.
 CARRY = Set to the value generated by the instruction. Unless otherwise defined, CARRY is set to the state of the hardware carry signal generated from the high order bit location by the arithmetic operation involved in that instruction.
 Overflow = Set but not cleared if generated by the instruction.

Memory Address Computation

If Memory Reference using standard Huffman coded displacement field then:

Begin

If CIR(5:1) = 0 then <<Direct>>

begin

If CIR(6:1) = 0 then <<P-Relative>>

Begin

E := PB Bank + (P +/- D + fxc)MOD 2**16;

If NOT(PB < E < PL) then BNDS Violation;

End;

If CIR(6:2) = 2 then <<DB Relative>>

begin

E := DB Bank + (DB + D + fxc)MOD 2**16;

If NOT(DB < E < DL) then BNDS Violation;

End;

If 12 <= CIR(6:4) <= 14 then <<Q-Relative>>

begin

E := SB Bank + (Q +/- D + fxc)MOD 2**16;

If NOT(SB < E < S) then BNDS Violation;

End;

If CIR(6:4) = 15 then <<S-Relative>>

begin

E := SE Bank + (S - D + fxc)MOD 2**16;

If NOT(SB < E < S) then BNDS Violation;

End;

End else

begin <<Indirect>>

If CIR(6:1) = 0 then

begin <<P-Relative>>

LBLADR := PB Bank + P +/- D; <<Label address>>

If NOT(PB < LBLADR < PL) then BNDS Violation;

E := LBLADR + ((LBLADR) + fxc)MOD 2**16;

If NOT(PB < E < PL) then BNDS Violation;

End else

begin <<Label(or pointer.) in stack or DB area>>

If CIR(6:2) = 2 then <<DB relative>>

begin <<DB relative>>

LBLADR := DB Bank + DB + D;

If NOT(DB < LBLADR < DL) then BNDS Violation;

End;

If 12 <= CIR(6:4) <= 14 then

begin <<Q relative>>

LBLADR := SB Bank + Q +/- D;

If NOT(SB < LBLADR < S) then BNDS Violation;

End;

If CIR(6:4) = 15 then

begin <<S Relative>>

LBLADR := SB Bank + S - D;

If NOT(SB < LBLADR < S) then BNDS Violation;

End;

LabelA := (LBLADR);

LabelE := (LBLADR + 1); <<not bnds tested>>

```

<<Label evaluation>>
  If LabelA(0:1) = 1 then
    Begin <<DB Relative address>>
      E := DB Bank + (DB + LabelA(1:15)/
        (If Byte then 2 else 1) + fXc)MOD 2**16;
      If NOT(DB < E < DL) then BNDS Violation;
    End;
  If LabelA(0:2) = 0 then
    Begin <<SB Relative address>>
      E := SB Bank + (SB + LabelA/
        (If Byte then 2 else 1) + fXc)MOD 2**16;
      If NOT(SB < E < S) then BNDS Violation;
    End;
  If LabelA(0:2) = 1 then
    Begin <<Label points to data segment>>
      DSTP := If LabelA(3:1) = 1 then DST register
        <<User data segment table>>
        else %1000; <<System data segment
          table>>
      If (DSTP) < 4*LabelA(4:12) then DST Violation;
      DSTEP := DSTP + LabelA(4:12)*4;
        <<address of DST entry>>
      RelAddr := ((If LabelA(2:1)=1 then LabelB
        else 0)/(If Byte then 2 else 1)
        + fXc)MOD 2**16;
      Length := DSTEP := (DSTEP) + 1;
      If RelAddr >= Length then
        Begin << Bnds problem, look for link DST>>
          If DSTEP(1)(0:1) = 1 then
            Begin <<consecutively linked DST's>>
              addseg := integer(RelAddr/Length);
              <<number of data segments needed>>
              If [LabelA(4:12) + addseg] >
                DSTEP(1).(4:12) then BNDS Violation;
              DSTEP := DSTEP + addseg * 4;
              RelAddr := RelAddr - Length * addseg;
              NOTE: All segments must be of equal
                length and of length 2**N bytes
                where N = 1 to 16.
            End
          else
            Begin <<random linked DST's>>
              while RelAddr > Length do
                begin
                  RelAddr := RelAddr - Length;
                  DSTEP := DSTP + DSTEP(1).(4:12)*4
                    <<link DST>>
                  If DSTEP(1).(4:12) = 0 then
                    BNDS Violation;<<link DST = 0>>
                  Length := (DSTEP) + 1;
                end
              End;
            End;
          End;
        End;
      If DSTEP(2)(1:1)=1 AND NPRV then DST Mode Violation;

```

```

        E Bank := DSTE(2).(8.8);
        E := DSTE(3) + RelAddr;
    End; <<End of data segment label evaluation>>
    End; <<End of addressing to data areas>>
    End; <<End of indirect address computation>>
    End; <<End of memory reference instruction addressing>>
    if conditional branch instruction then
    begin <<Conditional branch addressing>>
        If CIR(4:1) = 0 then <<direct>>
            P := PB Bank + (P +/-CIR(11:5))MOD2**16
        else <<indirect>>
            P := PB Bank + (P +/-CIR(11:5)
                + (P +/-CIR(11:5)))MOD2**16;
    End;

```

Note 1: Unless otherwise specified, when condition code is set, it is set on the new TOS.

Note 2: When English definitions are ambiguous, the ALGOL definitions define the instruction.

Note 3: When S is decremented by some amount (greater than zero), the contents of the memory cells that were deleted are undefined.

Load Instructions

1. LOAD E(*, D, X, PDQS) <Load>

Mem. OpCode=04

```

    S := S + 1;
    (S) := (E);

```

The content of E is pushed onto the stack.
Indicators = CCA

2. LDB E(*,D,X, PDQS)

Load byte

Mem. OpCode=15

CIR(6:1)=0

or Mem.OpCode=06

CIR(6:1)=0

```

If CIR(0:4)=6 then P-relative address computation;
If CIR(0:4)=15 then DQS relative address computation;
If Direct then
    Flag := if Xc(15:1) = 1 then TRUE else FALSE;
it INDIRECT then
    if P-relative then
        FLAG := if Xc(15:1) = 1 then TRUE else FALSE;
    if DQS-relative then
        FLAG := if (Label OFFSET + Xc) MOD 2 = 1
            then TRUE else FALSE;
S := S + 1;
(S(0:8)) := 0;
(S(8:8)) := if FLAG then (E(8:8)) else (E(0:8));

```

X contains a byte index. The byte referenced is loaded into the right half of the top of the stack. For Program relative indirect addressing the word referenced by the direct address (P + D) contains a self relative word offset and the byte index is added to obtain the effective byte address. For DQS relative indirect addressing the word referenced by the direct address (base + D) contains a byte label. The byte index is added to this relative byte address to obtain the effective byte address.
Indicators = CCb

```

3. LDD  E(*, D, X, PLQS)    <Load double>    Mem.OpCode=10
                                      CIR(6:1)=0
                                      or  Mem.OpCode=15
                                      CIR(6:1)=1

S := S + 2;
(S-1,S) := (E,E+1);

```

X contains a double word index. The double word in E, E+1 is pushed onto the stack, most significant word first. On indirect addressing the word referenced by the direct address contains a word label.
Indicators = CCA

```

4. LDX  E(*, D, X, PDQS)    <Load Index>      Mem. OpCode=13

X(16:16) := (E);
X(0:16) := 0;

```

The content of E is loaded into the low order 16 bits of index register. The hi order bits of X are zero.

Hardware will load (E) into X.

Note: To load a 32 bit quantity into X, it must first be loaded on the stack and then moved to X (32 bits) with an SBAX instruction.

Indicators = CCA

```

5. LRA  E (*, D, X, PDQS) <Load relative address>  Mem. OpCode=17
S := S + 1;
Xc := If indexed then X else 0;
If CIR(5:1) = 0 Then    << direct >>
begin
  If CIR(6:1) = 0 Then  <<prog rel >>
    (S) := [P+D + (P+D) - PB] + Xc
  Else                  << data relative >>
begin
  If CIR(7:1)=0 Then  << DB rel >>
begin
  (S) := D + Xc;
  (S) := (S) IOR %100000;
End else  << SB rel >>
  (S) := [BASE +/-D - SB] + Xc;
  S := S + 1;
  (S) := 0;
End;
End Else
begin                  << indirect >>
  If CIR(6:1) = 0 Then  << prog rel >>
begin
  (S) := [E + (E) - PB] + Xc;
End
Else                  << data rel >>
begin
  If (E(0:1)) = 1 Then    << 1xxx DB rel >>
begin
  (S) := (E) + Xc;
  S := S + 1;
  (S) := 0;
End Else
  If (E(1:1)) = 0 Then    << 00xx Stack rel >>
begin
  (S) := (E) + Xc;
  S := S + 1;
  (S) := 0;
End Else  << 01xx Segment Label >>
begin
  If (E(2:1)) = 0 Then  << 010x DST label >>
begin
  (S) := (E) IOR %020000;
  S := S + 1;
  (S) := Xc;
End Else  << 011x Data segment label >>
begin
  (S) := (E);
  S := S + 1;
  (S) := (E + 1) + Xc;
End;
End;  << Segment Label >>
End;  << Data relative >>
End.  << Indirect >>

```

Two words are always pushed on the stack by this instruction.

In the direct addressing mode a label is constructed that points at the direct address computed using indexing. For an address in code space this is a PB-relative offset. For a stack address this is an SB-relative offset and for a global address it is DB-relative with bit 0 set to 1 for identification. This relative address is pushed on the stack and then a zero is pushed to complete the pointer pair.

Examples of the action of LRA direct are:

LRA DE + N,X

-> (S-1) := [1, N + X]
(S) := [0]

LRA Q + N,X

-> (S-1) := [00, (Q + N - SB) + X]
(S) := [0]

LRA S - N,X

-> (S-1) := [00, (S - N - SB) + X]
(S) := [0]

LRA P + N,X

-> (S-1) := [00, (P + N - PB) + X]
(S) := [0]

For indirect addressing in program space the direct cell contains a self relative word offset and operates as follows:

LRA P + N,I,X

-> (S-1) := [00, (P + N + (P+N) - PB) + X]
(S) := [0]

For indirect addressing into data areas the pointer cell is assumed to contain a data label. A two word pair is constructed and indexed as follows:

STACK DATA LABEL

[00 OFFSET] -> (S-1) := [00 OFFSET + Xc]
(S) := [0]

GLOBAL DATA LABEL

$$\begin{bmatrix} 1 & \text{OFFSET} \end{bmatrix} \rightarrow (S-1) := \begin{bmatrix} 1 & \text{OFFSET} + Xc \\ & 0 \end{bmatrix}$$

DST LABEL

$$\begin{bmatrix} 010 & \text{DSEG} \end{bmatrix} \rightarrow (S-1) := \begin{bmatrix} 011 & \text{DSEG} \\ & Xc \end{bmatrix}$$

DATA SEGMENT LABEL

$$\begin{bmatrix} 011 & \text{DSEG} \\ & \text{OFFSET} \end{bmatrix} \rightarrow (S-1) := \begin{bmatrix} 011 & \text{DSEG} \\ & \text{OFFSET} + Xc \end{bmatrix}$$

Notice that a DST label is converted to a data segment label. These two words are pushed on the stack.
Indicators = Unaffected

6. QLD (*, D, X, PQS) <Quad load> MEM. OpCode=11

<<X is a Four word index>>

S := S + 4;

(S-3, S-2, S-1, S) := (E, E+1, E+2, E+3);

Using four word indexing, a memory address is computed. The four words starting at E are fetched from memory and pushed on the stack.
Indicators = CCA

Store Instructions

7. STOR E (*, D, X, DQS) <Store> Mem.OpCode=05
Cir(6:1)=1

(E) := (S);

S := S-1;

The content of the IOS is stored in memory location E. The stack is popped.
Indicators = Unaffected

8. STB E (*, D, X, DQS) <Store Byte> Mem.OpCode=16
CIR(6:1)=0

Only DQS address computation allowed;

If DIRECT then

Flag := If Xc (15:1) = 1 then TRUE else FALSE;

if INDIRECT then

FLAG := if (label offset + X) MOD 2 = 1

then TRUE else FALSE;

if FLAG then (E(8:8)) := (S(8:8)) else (E(0:8)) := (S(8:8));

S := S - 1;

X contains a byte index. On indirect addressing the word referenced by the direct address (base + D) contains a byte label. The byte index is added to the byte offset, and the right hand byte of the TOS is stored into the referenced byte. The TOS is deleted. Indicators = Unaffected

9. STD E (*, D, X, DQS) <Store double> Mem.OpCode=16
CIR(6:1)=1

(E, E+1) := (S-1, S);

S := S-2;

X contains a double word index. The top two words on the stack are stored at E, E+1. On indirect addressing the word referenced by the direct address contains a word label.
Indicators = Unaffected

10. QSTR E (*, D, X, DQS) <Quad store> Mem CpCode 1=06
CIR(6:1)=1

<<X is a four word index>>

(E, E+1, E+2, E+3) := (S-3, S-2, S-1, S);

S := S - 4;

using four word indexing, a memory address is computed. The four data words are popped from the stack and stored in memory at the computed address.

Indicators = unaffected

Memory Reference Integer Instructions

11. CMPM E (*, D, X, DQS) <Compare memory> Mem.OpCode=06
CIR(6:1)=1

CC := (S) : (E);

S := S-1;



The condition code is set to pattern C as a result of the comparison of(S):(E). The stack is popped.
Indicators = CCC

12. ADDM E (*, D, X, PDQS) <Add memory> Mem.OpCode=07

(S) := (S) + (E);

The content of E is added in integer form to the TOS. The result replaces the operand on the TOS.
Indicators = CCA, Carry, Overflow

13. SUBM E (*, D, X, DQS) <Subtract memory> Mem.OpCode=10
CIR(6:1)=1

(S) := (S) - (E);

The content of E is subtracted in integer form from the TOS. The result replaces the operand on the TOS.
Indicators = CCA, Carry, Overflow

14. INCM E (*, D, X, DQS) <Increment memory> Mem.OpCode=12
CIR(6:1)=0

(E) := (E) + 1;

The content of E is incremented by 1 in integer form.
Indicators = CCA, carry, Overflow

Branch Instructions

15. BR E (*, D, X, P or indirect DQS) <Branch> Mem.OpCode=14

```

if CIR(6:1) = 0 then <<P-relative branch>>
  if Cir(5:1)=0 then P := E <<direct>>
  else P := P+/-D +Xc + (P+/-D +Xc); <<indirect>>
    << NOTE: preindexing is used >>
If CIR(5:2)=3 then
  Begin
    Edir := DQS BASE +/- D + Xc;
    if (Edir(0:1))=0 then <<PB relative local label>>
      P := PB + (Edir.)
    else <<Branch to external segment>>
      P := Evaluation of External program label in E;
    if E = S then S := S - 1;
  End;
```

Control is transferred to new P. Note that for indirect branching, pre-indexing is used in computing the address of the indirect cell.
Indicators = Unaffected

Conditional Branches

CIR (10) is the sign of P+- displacement
Range (P-31, P+31)
CIR(4) is the indirect bit for cond. branches
L = signed range

16. BCC E(*,L,P) <Branch on Condition Code> Mem.OpCode=14
CIR(5:2)=01
If CC AND (CCF) <> 0 then P := E(*,L,P)
else P := P + 1;

The branch addresses are in the range P-31, P+31. Condition code field (CCF) in instruction is CIR (7:3). Control is transferred to location E(*,L,P) under the following conditions:

If CCF = 0, Never branch
1, Branch if CC = CCL <<BL>>
2, Branch if CC = CCE <<BE>>
3, Branch if CC = CCL or CCE <<BLE>>
4, Branch if CC = CCG <<BG>>
5, Branch if CC = CCG or CCL <<BNE>>
6, Branch if CC = CCG or CCE <<BGE>>
7, Always branch

Indicators = Unaffected

17. BCY E(*,L,P) <Branch on carry> Sub CpCode l=14
if STATUS(5) = 1 then P := E(*,L,P) else P := P + 1;

If the carry bit of the status word is on, control is transferred to E(*,L,P).
Indicators = Carry cleared

18. BNCY E(*,L,P) <Branch on no carry> Sub OpCode l=15
if STATUS(5) = 0 then P := E(*,L,P) else P := P + 1;

If the carry bit of the status word is off, control is transferred to E(*,L,P).
Indicators = Carry cleared

19. BOV E(*,L,F) <Branch on overflow> Sub OpCode 1=30

if STATUS(4) = 1 then P := E(*,L,F) else P := P + 1;

If the overflow bit in the status word is on, control is transferred to E(*,L,F).

Indicators = Overflow cleared

20. BNOV E(*,L,F) <Branch on no overflow> Sub OpCode 1=31

if STATUS(4) = 0 then P := E(*,L,F) else P := P + 1;

If the overflow bit in the status word is off, control is transferred to E(*,L,F).

Indicators = Overflow cleared

21. BRO E(*,L,F) <Branch on TOS odd> Sub OpCode 1=36

if (S(15)) = 1 then P := E(*,L,F) else P := P + 1;
S := S-1;

If the TOS is odd, control is transferred to E(*,L,F). The stack is popped.

Indicators = Unaffected

22. BRE E(*,L,F) <Branch on TOS even> Sub OpCode 1=37

if (S(15)) = 0 then P := E(*,L,F) else P := P + 1;
S := S-1;

If the TOS is even, control is transferred to E(*,L,F). The stack is popped.

Indicators = Unaffected

23. CPRB E(*,L,F) <Compare range and branch> SubOpCode 1=26

```

if (S-2) <= (S-1) <= (S) then
  begin
    CC := CCE;
    P := E(*,L,F);
  end else
  begin
    if (S-1) < (S-2) then CC := CCL;
    if (S-1) > (S) then CC := CCG;
  end;
  S := S - 3;

```

The integer in the stack at S - 1 is tested to determine if it is within the interval defined by the upper bound integer on the TCS and the lower bound integer found in the stack at S - 2. The condition code is set by the comparison. If the integer in S-1 is within the range then the branch to E(*,L,F) occurs. The three parameters on the stack are popped.

Indicators = Condition Code

$$\text{CIR}(6:1) = 0$$

```
VAR := ((S-3, S-2));    <Label>
STEP := (S-1);
FINAL := (S);
if STEP >= 0 then
    if VAR > FINAL then S := S - 4 else P := E;
if STEP < 0 then
    if VAR < FINAL then S := S - 4 else P := E;
```

The instruction requires that the top 4 elements of the stack (TOS, TOS-1, TOS-2, TOS-3) are the final value, the step size and the label address of the variable respectively. VAR is tested against the limit. If the limit is exceeded, then pop the top 4 words from the stack and execution continues at P + 1. If the limit is not exceeded, then control is transferred to E.

Indicators = Unaffected

```
25.  MTBA      E  (*,D,P)      <Modify,Test,Branch,A>      Mem . Cp Code=05
                                           CIR(4:1)=0
                                           CIR(6:1)=0
```

```

VAR      := ((S-3,S-2))      <Label>
STEP     := (S-1);
FINAL    := (S);
VAR      := VAR + STEP;
((S-3,S-2)) := VAR;
if VAR > +32767 or VAR < -32768 then S:= S-4;
                                                <<instruction ends>>
if STEP >= 0 then
    if VAR > FINAL then S := S - 4 else P := E;
if STEP < 0 then
    if VAR < FINAL then S := S - 4 else P := E;

```

This instruction requires that the top 4 elements of the stack (TOS, TOS-1, TOS-2, TOS-3) are the final value, the step size and the label address of the variable respectively. The step size is added in integer form to the variable, the sum replaces the old value of the variable, and the sum is tested against the limit. If the limit is exceeded or the sum is outside the range $-32768 \leq \text{VAR} \leq +32767$ then pop all 4 from the stack and execution continues at P + 1. If the limit is not exceeded, then control is transferred to E.
Indicators = Unaffected

Integer Instructions

```
26.  CVID      <Convert Integer to Double>      Stack OpCode=03
```

```
S := S + 1;
(S) := (S-1);
(S-1) := if (S(0:1))=1 then -1 else 0;
```

The integer on the top of the stack is replaced by the equal two word sign extended double integer.
Indicators = unaffected

```
27. CCI    <Convert Condition Code to Integer>    MiniOpCode=14
                                         CIR(12:4)=0
```

```
S := S + 1;  
If CC = CCL then (S) := -1;  
If CC = CCE then (S) := 0;  
If CC = CCG then (S) := 1;
```

The condition code states of CCL, CCE or CCG are used to generate a -1, 0, or +1 that is pushed on the stack.
Indicators = unaffected

28. CMP <Compare> Stack OpCode=17

```
CC := (S-1) : (S);
S := S - 2;
```

The condition code is set to pattern C as a result of the integer comparison of (S-1) : (S). The operands are deleted.
Indicators = CCC

29. ADD <Add> Stack OpCode=20

```
(S-1) := (S-1) + (S);
S := S - 1;
```

The top two words of the stack are added in integer form and they are then popped. The resulting sum is pushed onto the stack.
Indicators = CCA, Carry, Overflow

```
30. SUB      <Subtract>          Stack CpCode=21
```

```
(S-1) := (S-1) - (S);
S := S-1;
```

The top word of the stack is subtracted in integer form from the second word and they are popped. The resulting difference is pushed onto the stack.

Indicators = CCA, Carry, Overflow

31. MPY <Multiply>

Stack OpCode=22

```
(S-1) := (S-1) * (S);  
S := S-1;
```

The top two words of the stack are multiplied in integer form. The two words are deleted, and the least significant word of the double length product is pushed onto the stack. If the high order 17 bits (including the sign bit of the second word) are not all zeros, or all ones, overflow is set.

Indicators = CCA, Overflow

32. DIV <Divide>

Stack OpCode=23

```
TEMP := (S-1)/(S);  
(S) := (S-1) mod (S);  
(S-1) := TEMP;
```

The integer in the second word of the stack is divided by the integer on the TOS. The second word is replaced by the quotient, and the top word is replaced by the remainder.

Indicators = CCA on quotient, Overflow

33. NEG <Negate>

Stack OpCode=24

```
(S) := -(S);
```

The TOS is replaced by its 2's complement.

Indicators = CCA, Overflow, Carry

Logical Instructions

34. LCMF <Logical compare> Stack OpCode=57

CC := (S-1) : (S);
S := S - 2;

The condition code is set to pattern C on the comparison of (S-1):(S). The two operands are deleted.
Indicators = CCC

35. LADD <Logical add> Stack OpCode=60

(S-1) := (S-1) + (S) mod 2^{16} ;
S := S - 1;

The top two words of the stack are added as 16 bit positive integers and they are deleted from the stack. The resulting sum is pushed onto the stack. CCA is set as though the operation was performed on integers rather than logical quantities.
Indicators = CCA, Carry

36. LSUB <Logical subtract> Stack OpCode=61

(S-1) := (S-1) - (S) mod 2^{16} ;
S := S - 1;

The top word of the stack is subtracted in logical form from the second word and they are deleted. The resulting difference is pushed onto the stack. CCA is set as though the operation was performed on integers rather than logical quantities.
Indicators = CCA, Carry

37. LMPY <Logical multiply> Stack OpCode=62

(S-1, S) := (S-1) * (S);

The top two words on the stack are multiplied as 16 bit positive integers. The words are replaced by the double length product with the least significant half on the TOS.

Carry is cleared if the high order 16 bits are all zeros such that the low order 16 bits represent the true result. Otherwise the carry bit is set. CCA is set as though the operation was performed on integers rather than logical quantities.
Indicators = CCA, Carry

38. LDIV <Logical divide> Stack OpCode=63

```

TEMP := logical ((S-2, S-1)/(S));
(S-1) := (S-2, S-1) mod (S);
(S-2) := TEMP;
S := S - 1

```

The 32 bit positive integer in the 2nd and 3rd words of the stack is divided by the 16 bit positive integer on the TOS. The top three words are deleted. The quotient is pushed onto the stack and then the remainder is pushed on. If overflow occurs, the quotient will be modulo $2^{**}16$. CCA is set on the quotient as though the operation was performed on integers rather than logical quantities.
Indicators = CCA on quotient, Overflow

39. NOT <One's complement> Stack OpCode=64

```

(S) := (S);

```

Converts the top word of the stack to its one's complement.
Indicators = CCA

Double Integer Instructions

40. DCMF <Double compare> Stack OpCode=10

```

CC := (S-3, S-2):(S-1, S);
S := S - 4;

```

The condition code is set to pattern C as a result of the double word integer comparison of (S-3, S-2):(S-1, S). The two double words are deleted from the stack.
Indicators = CCC

41. DADD <Double add> Stack OpCode=11

```

(S-3, S-2) := (S-3, S-2) + (S-1, S);
S := S - 2;

```

The two double word integers contained in the top 4 elements of the stack are added in double length integer form and they are deleted. The double word integer sum is pushed onto the stack.
Indicators = CCA, Carry, Overflow

42. DSUB <Double subtract> Stack OpCode=12

$(S-3, S-2) := (S-3, S-2) - (S-1, S);$
 $S := S - 2;$

The double word integer contained in the 1st and 2nd words of the stack is subtracted from the double word integer contained in the 3rd and 4th words of the stack. The top 4 words of the stack are deleted and the double word integer result is pushed onto the stack.
Indicators = CCA, Carry, Overflow

43. DMPY <Double Multiply> Stack OpCode=13

$(S-3, S-2) := (S-3, S-2) * (S-1, S);$

The top two words of the stack and the next two words in the stack are multiplied in integer form. Four words are deleted from the stack and the least significant two words of the quad length product are pushed on the stack. If the high order 33 bits (including the sign bit of the third word) are not all zeros or all ones, overflow is set.
Indicators = CCA, Overflow

44. DDIV <Double Divide> Stack OpCode=14

$TEMP1, TEMP2 := (S-3, S-2) / (S-1, S);$
 $(S-1, S) := (S-3, S-2) \bmod (S-1, S);$
 $(S-3, S-2) := TEMP1, TEMP2;$

The double word integer in the fourth and third elements of the stack is divided by the double word integer in the second and first words in the stack. Four words are deleted from the stack, the double word integer quotient is pushed on the stack and then the double word integer remainder is pushed on the stack.
Indicators = CCA on quotient, Overflow

45. DNEG <Double negate> Stack OpCode=15

$(S-1, S) := - (S-1, S);$

The double word integer contained in the top 2 words of the stack is negated and replaces the original double word integer.
Indicators = CCA, Overflow, Carry

Floating Point Instructions

46. FCMP <Floating compare> Stack OpCode=50

CC := (S-3, S-2):(S-1, S);
S := S - 4;

The condition code is set to pattern C as a result of the floating point comparison of (S-3,S-2):(S-1,S). The two floating point double-words are deleted.
Indicators = CCC

47. FADD <Floating add> Stack OpCode=51

(S-3,S-2) := (S-3,S-2) + (S-1, S);
S := S - 2;

The two floating point numbers contained in the top 4 words of the stack are added in floating point form. The top 4 words of the stack are deleted and the 2 word sum is pushed onto the stack.

Note: Computed result = round (true result;22)
Indicators = CCA, Overflow

48. FSUB <Floating subtract> Stack OpCode=52

(S-3, S-2) := (S-3,S-2) - (S-1, S);
S := S - 2;

The floating point number contained in the 1st and 2nd words of the stack is subtracted from the floating point number contained in the 3rd and 4th words of the stack in floating point form. The top 4 words of the stack are deleted and the 2 word difference is pushed onto the stack.

Note: Computed result = round (true result, 22);
Indicators = CCA, Overflow

49. FMPY <Floating multiply> Stack OpCode=53

(S-3, S-2) := (S-3, S-2) * (S-1, S);
S := S - 2;

The two floating point numbers contained in the top 4 words of the stack are multiplied in floating point form. The top 4 words of the stack are deleted and the 2 word result is pushed onto the stack.

Note: Computed result = round (true result, 22);
Indicators = CCA, Overflow

50. FDIV <Floating divide> Stack OpCode=54

$(S-3, S-2) := (S-3, S-2) / (S-1, S);$
 $S := S - 2;$

The floating point number contained in the 3rd and 4th words of the stack is divided by the floating point number contained in the 1st and 2nd words of the stack. The top 4 words of the stack are deleted and the 2 word quotient is pushed onto the stack.

Note: Computed result = round (true result, 22);
Indicators = CCA, Overflow

51. FNEG <Floating negate> Stack OpCode=55

$(S-1, S) := -(S-1, S);$

The floating point number contained in the top 2 words of the stack is negated in floating point form.
Indicators = CCA

52. FLT <Float> Stack OpCode=47

$S := S + 1;$
 $(S-1, S) := \text{float}(S-1);$

Converts the integer on the TOS to a 32 bit floating point number. The TOS is deleted and the double word floating point result is pushed onto the stack.
Indicators = CCA

53. FIXT <Fix and truncate> Stack OpCode=71

$(S-1, S) := \text{trunc}((S-1, S));$

The floating point number contained in the top two words of the stack is converted to fixed point form and truncated.

Carry is cleared if the high order 17 bits (including the sign bit of the second word) are either all zeros or all ones, such that the low order 16 bits represent the true result. Otherwise, the carry bit is set.

Indicators = CCA, Carry, Overflow

54. FIXR <Fix and round> Stack OpCode=70

$(S-1, S) := \text{trunc}((S-1, S) + .5 * \text{sign}(S-1, S));$

The floating point number contained in the top 2 words of the stack is converted to fixed point form and rounded.

Carry is cleared if the high order 17 bits (including the sign bit of the second word) are either all zeros or all ones, such that the low order 16 bits represent the true result. Otherwise, the carry bit is set.

Indicators = CCA, Carry, Overflow

Extended Floating Point Instructions

55. ECMP <Extended floating compare> Stack OpCode = 77

$CC := (S-7, S-6, S-5, S-4) : (S-3, S-2, S-1, S);$
 $S := S - 8;$

The condition code is set to pattern C as a result of the floating point comparison of $(S-7, S-6, S-5, S-4) : (S-3, S-2, S-1, S)$. The two floating point quadruple-words are deleted.

Indicators = CCC

56. EADD <Extended floating add> Stack OpCode = 72

$(S-7, S-6, S-5, S-4) := (S-7, S-6, S-5, S-4) + (S-3, S-2, S-1, S);$
 $S := S - 4;$

The two floating point numbers contained in the top 8 words of the stack are added in floating point form. The top 8 words of the stack are deleted and the 4 word sum is pushed onto the stack.

Note: Computed result = round (true result; 54);
 Indicators = CCA, Overflow

57. ESUB <Extended floating subtract> Stack OpCode = 73

$(S-7, S-6, S-5, S-4) := (S-7, S-6, S-5, S-4) - (S-3, S-2, S-1, S);$
 $S := S - 4;$

The floating point number contained in the top four words of the stack is subtracted from the floating point number contained in the next 4 words of the stack in floating point form. The top 8 words of the stack are deleted and the 4 word difference is pushed onto the stack.

Note: Computed result = round (true result, 54);
Indicators = CCA, Overflow

58. EMPY <Extended floating multiply> Stack OpCode = 74

(S-7,S-6,S-5,S-4) := (S-7,S-6,S-5,S-4) * (S-3,S-2,S-1,S);
S := S - 4;

The two floating point numbers contained in the top 8 words of the stack are multiplied in floating point form. The top 8 words of the stack are deleted and the 4 word result is pushed onto the stack.

Note: Computed result = round (true result, 54);
Indicators = CCA, Overflow

59. EDIV <Extended floating divide> Stack OpCode = 75

(S-7,S-6,S-5,S-4) := (S-7,S-6,S-5,S-4) / (S-3,S-2,S-1,S);
S := S - 4;

The floating point number contained in the 5th thru the 8th words of the stack is divided by the floating point number contained in the 1st thru 4th words of the stack. The top 8 words of the stack are deleted and the 4 word quotient is pushed onto the stack.

NOTE: Computed result = round (true result, 54);
Indicators = CCA, Overflow

60. ENEG <Extended floating negate> Stack OpCode = 76

(S-3,S-2,S-1,S) := -(S-3,S-2,S-1,S);

The floating point number contained in the top 4 words of the stack is negated in floating point form.
Indicators = CCA

61. EFLT <Extended float> Stack OpCode=30

S := S + 2;
(S-3,S-2,S-1,S) := Efloat((S-3,S-2));

Converts the double word integer contained in the top 2 words of the stack to a four word floating point number.

Note: Computed result = round (true result, 22);
Indicators = CCA

62. EFXR <Extended fix and round> Stack OpCode = 34

```
(S-3,S-2) := trunc((S-3,S-2,S-1,S) + .5*sign(S-3));
S := S-2;
```

The extended floating point number contained in the top four words of the stack is converted to Double Integer form and rounded.

Carry is cleared if the high order 17 bits (including the sign bit of the second word) are either all zeros or all ones, such that the low order 16 bits represent the true result. Otherwise the carry bit is set.

Indicators = CCA, Carry, Overflow

63. EFXT <Extended Fix and Truncate> Stack OpCode = 41

```
(S-3,S-2) := trunc((S-3,S-2,S-1,S));
S := S - 2;
```

The extended floating point number contained in the top four words of the stack is converted to Double Integer form and truncated.

Carry is cleared if the high order 17 bits (including the sign bit of the second word) are either all zeros or all ones, such that the low order 16 bits represent the true result. Otherwise, the carry bit is set.

Indicators = CCA, Carry, Overflow



Boolean Instructions

64. OR <Or, logical> Stack OpCode=65

```
(S-1) := (S-1) or (S);
S := S-1;
```

The top two words of the stack are logically inclusive ORed together, the operands are deleted and the result is pushed onto the stack.
Indicators = CCA on the TOS

65. XOR <Exclusive or, logical> Stack OpCode=66

```
(S-1) := NOT(S-1) and (S) or (S-1) and NOT(S);
S := S-1;
```

The top two words of the stack are logically exclusive ORed together, the operands are deleted, and the result is pushed onto the stack.
Indicators = CCA on the new TOS

66. AND <And, logical> Stack OpCode=67

(S-1) := (S-1) and (S);
S := S - 1;

The top two words of the stack are logically ANDed together, the operands are deleted and the result is pushed onto the stack.
Indicators = CCA on the new TOS

Test Instructions

67. BTST <Test BYTE on TOS> Stack OpCode=31

Indicators = CCB on (S(8:8))

CCB is set on the right byte on the TOS.

68. TEST <Test TOS> Stack OpCode=25

Indicators = CCA on (S)

69. DTST <Test double word on TOS> Stack OpCode=27

Carry is cleared if the high order 17 bits (including the sign bit of the second word) are either all zeros or all ones, such that the low order 16 bits represent the true result. Otherwise, the carry bit is set.

Indicators = CCA on double word (S-1, S), Carry

70. QTST <Test Quad word on TOS> Stack OpCode=26

Indicators = CCA on Quad word (S-3,S-2,S-1,S)

Zero Instructions

71. ZERO <Push zero> Stack OpCode=06

 S := S + 1;
 (S) := 0;

A zero word is pushed onto the stack.
Indicators = Unaffected

72. DZERO <Double push zero> Stack OpCode=07

 S := S + 1;
 (S) := 0;
 S := S + 1;
 (S) := 0;

Two words containing all zeros are pushed onto the stack.
Indicators = Unaffected

Duplicate and Delete Instructions

73. DEL <Delete A> Stack OpCode=40

 S := S - 1;

The TOS is deleted.
Indicators = Unaffected

74. DDEL <Double delete> Stack OpCode=02

 S := S - 2;

The top 2 words of the stack are deleted.
Indicators = Unaffected

75. DELB <Delete B> Stack OpCode=01

(S-1) := (S);
S := S - 1;

The second word of the stack is deleted and the stack is compressed.
The TOS is unchanged.
Indicators = Unaffected

76. DUP <Duplicate A> Stack OpCode=45

S := S + 1;
(S) := (S-1);

The top word of the stack is duplicated by pushing a copy of the TOS onto the stack.
Indicators = CCA

77. DDUP <Double duplicate> Stack OpCode=46

S := S + 2;
(S-1, S) := (S-3, S-2);

The double word in the top two words of the stack is duplicated by pushing a copy of it onto the stack.
Indicators = CCA on new TOS Doubleword

78. QDUP <Quad Duplicate> Stack OpCode=42

S := S + 4;
(S-3, S-2, S-1, S) := (S-7, S-6, S-5, S-4);

The four top words on the stack are duplicated by pushing a copy of them onto the stack.
Indicators = CCA on the new Quad word

Exchange Instructions

79. XCH <Exchange A and B> Stack OpCode=32

TEMP := (S-1);
(S-1) := (S);
(S) := TEMP;

The top 2 words of the stack are interchanged.
Indicators = CCA on the new TOS

80. DXCH <Double exchange> Stack OpCode=16

```
TEMP := (S-1, S);  
(S-1, S) := (S-3, S-2);  
(S-3, S-2) := TEMP;
```

The top two double word pairs are interchanged on the stack.
Indicators = CCA on the new TOS double word.

1. QXCH <Quad exchange> Stack OpCode=33

```
TEMP := (S-3, S-2, S-1, S);  
(S-3, S-2, S-1, S) := (S-7, S-6, S-5, S-4);  
(S-7, S-6, S-5, S-4) := TEMP;
```

The top two Quad word sets on the stack are interchanged.
Indicators = CCA on the new TOS Quad word

82. XAX <Exchange A and X> Stack OpCode=35

```
TEMP := X;  
X := (S);  
(S) := TEMP;
```

The content of S is interchanged with the content of the index register.
Indicators = CCA on the new TOS

83. CAB <Rotate ABC> Stack OpCode=56

```
TEMP := (S-2);  
(S-2) := (S-1);  
(S-1) := (S);  
(S) := TEMP;
```

The third word of the stack is removed from the stack, the top two words are compressed onto the rest of the stack, and the original third word is pushed onto the stack.
Indicators = CCA on the new (S-1, S)

Index Transfer Instructions

84. LDXA <Load X onto stack> Stack OpCode=44

S := S + 1;
(S) := X;

The content of the index register is pushed onto the stack.
Indicators = CCA on the new (S)

85. STAX <Store A into X> Stack OpCode=43

X := (S);
S := S-1;

The content of the index register is replaced by the top word of the stack and the stack is popped.
Indicators = unaffected

Index Arithmetic Instructions

86. INCX <Increment index> Stack OpCode=04

X := X + 1;

The content of the index register is incremented by one in integer form.
Indicators = CCA, Carry, Overflow

87. DECX <Decrement X> Stack OpCode=05

X := X - 1;

The content of the index register is decremented by one in integer form.
Indicators = CCA, Carry, Overflow

88. ADAX <Add A to X> Stack OpCode=36

X := X + (S);
S := S - 1;

The top word of the stack is added to the content of the index register. The sum replaces the content of the index register, and the stack is popped.

Indicators = CCA on the new X, Carry, Overflow

89. ADXA <Add X to A>

Stack CpCode=37

$(S) := (S) + X;$

The content of the index register is added to the top word of the stack, and the sum replaces the top word of the stack.
Indicators = CCA on the new A, Carry, Overflow

90. XLSL CNT <X Logical Shift Left> Sub CpCode 1=17

The contents of the X register is shifted left CNT bits.
Indicators = Unaffected

Control Instruction

91. NOP <No operation>

Stack CpCode=00

The user program space and data space remain unchanged.
Indicators = Unaffected

Shift Instructions

Memory opcode=01
Sub CpCode = CIR(5:9)
Bit 4 is a true index bit
 $CNT = (CIR(10:6) + (if\ CIR(4:1) = 1$
 then X else 0)) mod 64;

Single Word Shift Instructions

92. ASL CNT <Arithmetic shift left> Sub OpCode 1 =00

The TOS is shifted left CNT bits. If the sign bit changes, overflow is set.
Indicators = CCA,Overflow

93. ASR CNT <Arithmetic shift right> Sub OpCode 1=01

The TOS is shifted right CNT bits, propagating the sign bit.
NOTE: low order bits are truncated which for negative integers is a magnitude round up, i.e. it does not divide by two.
Indicators = CCA

94. LSL CNT <Logical shift left> Sub OpCode 1=02

The TOS is shifted left CNT bits logically.
Indicators = CCA

95. LSR CNT <Logical shift right> Sub OpCode 1=03

The TOS is shifted right CNT bits logically.
Indicators = CCA

96. CSL CNT <Circular shift left> Sub OpCode 1=04

The TOS is shifted left CNT bits circularly.
Indicators = CCA

97. CSR CNT <Circular shift right> Sub OpCode 1=05

The TOS is shifted right CNT bits circularly.
Indicators = CCA

98. SCAN <Scan Bits> Sub OpCode 1=06
 CIR(10:6)=0

```

If CIR(4:1) = 0 then X := -1
If (S) <> 0 then
  begin
    while (S(0:1)) = 0 then
      begin
        logical shift left A by 1;
        X := X + 1;
      end;
    logical shift left A by 1;
    X := X + 1;
  end
else if CIR(4:1) = 0 then X := 16
                                   else X := X + 16;

```

The TOS is shifted left until S(0:1) = 1, then shifted left one more bit. The shift count is left in the index register.
 Indicators = CCA on final TOS

Double word Shift Instructions

99. DASL CNT <Double arithmetic shift left> Sub OpCode 1=20

The double word contained in (S-1, S) is shifted left CNT bits. If the sign bit changes, overflow is set.
 Indicators = CCA on Doubleword, Overflow

100. DASR CNT <Double arithmetic shift right> Sub OpCode 1=21

The double word contained in (S-1, S) is shifted right CNT bits, propagating the sign bit, S-1(0).
 Note: low order bits are truncated which for negative double integers is a magnitude round up, i.e. it does not divide by two.
 Indicators = CCA on Doubleword

101. DLSL CNT <Double logical shift left> Sub OpCode 1=22

The double word contained in (S-1, S) is shifted left CNT bits logically.
 Indicators = CCA on Doubleword

102. DLSR CNT <Double logical shift right> Sub OpCode 1=23

The doubleword contained in (S-1,S) is shifted right CNT bits logically.

Indicators = CCA on Doubleword

103. DCSL CNT <Double circular shift left> Sub OpCode 1=24

The doubleword contained in (S-1,S) is shifted left CNT bits circularly.

Indicators = CCA on Doubleword

104. DCSR CNT <Double circular shift right> Sub OpCode 1=25

The doubleword contained in (S-1,S) is shifted right CNT bits circularly.

Indicators = CCA on Doubleword

Four word Shift Instructions

105. QLSL CNT <Quad logical shift left> Sub OpCode 1=10

The four words contained in (S-3,S-2,S-1,S) are shifted left CNT bits logically.

Indicators = CCA on Quad word

106. QLSR CNT <Quad logical shift right> Sub OpCode 1=11

The four words contained in (S-3,S-2,S-1,S) are shifted right CNT bits logically.

Indicators = CCA on Quad word

Bit Instructions

107. LBIT (BD,X) <Load bit from memory>

Sub OpCode 1=32

BD is the bit displacement CIR(10:6)
 (S-1,S) contains a Data Label

```
DTEMP := Xc + BD;    <Double word>
BITL := DTEMP(28:4); <bit location in word>
fXc := DTEMP/16;
E := Label base + Label offset + fXc;
BNDSTEST;
S := S - 1;
If (E(BITL:1)) = 1 then S := 1 else S := 0;
```

Starting at the address in memory of the word pointed at by the label in the two top words of the stack, a bit location is computed by indexing from the word address, BD + Xc bits forward in memory. The data label is popped from the stack and the bit from the computed bit memory location is pushed on the stack in bit 15.

Indicators = CCA on the new TOS

108. SBIT (DB,X) <Store bit in memory>

Sub OpCode 1=33

BD is the bit displacement CIR(10:6)
 (S-1,S) contains a data label
 (S-2(15:1)) contains the bit to be stored

```
DTEMP := Xc + BD;    <Double word>
BITL := DTEMP(28:4); <Bit location in word>
fXc := DTEMP/16;
E := Label base + Label offset + fXc;
BNDSTEST;
(E(BITL:1)) := (S-2(15:1));
S := S-3;
```

Starting at the address in memory of the word pointed at by the label in the two top words of the stack, a bit location is computed by indexing from the word address, BD + Xc bits forward in memory. Bit 15 of the third word in the stack is stored in that bit location. Three words are deleted from the stack.

Indicators = Unaffected

Field Instructions

109. EXF J,K <Extract field> Sub OpCode 2=15

0 <= J <= 15 specifies the first (leftmost) bit in the source field.

0 <= K <= 15 specifies the number of bits in the field.

(S(16-k:K)) := (S(J:K));

(S(0:16-K)) := 0;

Bits J, J+1, J+2, ..., J+K-1 are extracted from the TOS, and the TOS is deleted. The K bit field is right justified with high order zeros, and this result is pushed onto the stack.
Indicators = CCA on the new TOS

110. DPF J,K <Deposit field> Sub CpCode 2=16

0 <= J <= 15 specifies the first bit in the destination field.

0 <= K <= 15 specifies the number of bits in the field.

(S-1(L:k)) := (S(16-K:K));

(S-1(0:J)) := (S-1(0:J));

(S-1(J+K:16-J-K)) := (S-1(J+K:16-J-K));

S := S - 1;

The K least significant bits of the TOS are placed in bits J, J+1, ..., J+K-1 of the second word of the stack; the remaining bits of the second word of the stack are unchanged. The source operand is deleted from the stack.
Indicators = CCA on the new TOS

Immediate Instructions

N = CIR(8:8)

0 <= N <= 255

111. LDI = N <Load immediate> Sub OpCode 2=02

S := S + 1;

(S) := N;

The immediate positive quantity N is pushed onto the stack as a positive integer.
Indicators = CCA on the new TOS

112. LDNI = N <Load negative immediate> Sub OpCode 2=12

S := S + 1;
(S) := -N;

The immediate positive quantity N is 2's complemented and pushed onto the stack as a negative integer.
Indicators = CCA on the new TOS

113. CMPI = N <Compare immediate> Sub OpCode 2=04

CC := (S) : N;
S := S - 1;

The condition code is set to pattern C as a result of the comparison of the TOS with the positive quantity N. The TOS is deleted.
Indicators = CCC

114. CMPN =N <Compare negative immediate> Sub OpCode 2=14

CC := (S) : -N;
S := S - 1;

The condition code is set to pattern C as a result of the comparison of the TOS with the 2's complement of the positive quantity N. The TOS is deleted.
Indicators = CCC

115. ADDI = N <Add immediate> Sub OpCode 2=05

(S) := (S) + N;

The immediate positive quantity N is added to the TOS in integer form, and sum replaces the TOS.
Indicators = CCA, Carry, Overflow

116. SUBI = N <Subtract immediate> Sub OpCode 2=06

(S) := (S) - N;

The immediate positive quantity N is subtracted from the TOS in integer form, and the result replaces the TOS.
Indicators = CCA, Carry, Overflow

117. MPYI = N <Multiply immediate> Sub OpCode 2=07

(S) := (S) * N;

The immediate positive quantity N is multiplied with the TOS in integer form. The 16 least significant bit integer result replaces the TOS. Overflow is set if the 16 most significant bits of the 32 bit result are not the same as the sign bit of the least significant word.

Indicators = CCA, Overflow

118. DIVI = N <Divide immediate> Sub OpCode 2=10

(S) := (S)/N;

The immediate positive quantity N is divided into the TOS in integer form; the 16 bit integer quotient replaces the TOS.

Indicators = CCA

119. ANDI = N <Logical AND immediate> Sub OpCode 3=17

(S) := (S) and N;

The immediate positive quantity N is expanded to 16 bits with high order zeros and ANDed with the TOS; the result replaces the TOS.

Indicators = CCA

Immediate Index Instructions

120. LDXI = N <Load X immediate> Sub OpCode 2=03

X := N;

The index register is loaded with the positive immediate operand N.
Indicators = Unaffected

121. LDXN = N <Load X negative immediate> Sub OpCode 2=13

X := -N;

The index register is loaded with the 16 bit 2's complement of the immediate operand.

Indicators = Unaffected

122. ADXI = N <Add immediate to X> Sub OpCode 3=05

$X := X + N;$

The immediate positive quantity N is added to the content of the index register in integer form.

Indicators = CCA on X, Carry, Overflow

123. SBXI = N <Subtract immediate from X> Sub OpCode 3=06

$X := X - N;$

The immediate positive quantity N is subtracted from the content of the index register in integer form.

Indicators = CCA on X, Carry, Overflow

Program Control Instructions

124. PCAL N Procedure call Sub OpCode 3=02

```

if N = 0 then
  begin
    TEMP := (S);
    S := S - 1;
  end else TEMP := (PL - N);
S := S + 1;
(S) := X;
S := S + 1;
(S) := P+1 -PB;
S := S + 1;
(S) := STATUS;
S := S + 1;
(S) := S - Q;
Q := S;
P := evaluation of label in TEMP;

```

Control is transferred to the location evaluated by the label contained on the TOS if N = 0 or by the label contained in PL - N otherwise. Then a four word stack marker is placed on the stack. If reference is made to a local label, transfer is made to the PL relative address and the stack linkage is created ignoring the U bit.

If reference is made to an external label whose Code Segment Table number is different than the present number and the machine is in user mode, the U bit of the target segment linkage must not be set, otherwise a trap occurs. Furthermore if an external label is +

referenced a local label must occur in the STT table of the called segment. If the Trace bit is on in the Target CST entry the PCAL will be made to the Trace routine. If a privileged user is calling a user segment it will run in privileged mode.
Indicators = Unaffected

125. EXIT N <Procedure exit> Sub OpCode 3=03

```

S := Q;
If (Q-2(0:1)) = 1 then INTERRUPT call to TRACE
else
begin
  TEMP := Q - 4;
  S := Q;
  Q := S - (S);
  S := S - 1;
STATUS := (S); Note that user mode may EXIT to privileged
               code but may not change the user mode bit or
               the external interrupt disable bit.
  PB, PL are set to code segment specified in the
               status word (Q-1(8:8)).
  S := S - 1;
  P := PB + (S);
  S := S - 1;
  X := (S);
  S := TEMP - N;
End;
```

This instruction is used to return from routines called by the PCAL instruction. The exit occurs by setting Q to Q - DeltaQ, STATUS to the previous status, X the previous X, and P to the return P.

If bit 0 of (Q-2) is set to one, S is set to Q and control is transferred to the interrupt TRACE routine instead of exiting.

126. SXIT N <Subroutine exit> Sub OpCode 3 =04

```

P := PB + (S);
S := S - N - 1;
```

The instruction is used to return from subroutines.
Indicators = Unaffected

127. HALT K <Halt> Spec. OpCode=17

This is a privileged instruction. The machine halts. This instruction is non-interruptable and manual intervention is required to restart the machine. If, on entry, the K field = %17 then the contents of the TOS is placed in the NIR register, otherwise CIR is transferred to the NIR. When all other communication has failed, +

this is used by the System to place an error number in the NIR lights on the CPU board.
Indicators = Unaffected

128. PAUS K <Pause> Spec. OpCode=01

This is a privileged instruction. The machine pauses. This instruction is interruptable. The K field is ignored.
Indicators = Unaffected

129. XEQ K <Execute> Spec. OpCode=06

CIR := (S-K);

The content of the word in the stack at S - K is placed in the current instruction register to be executed. Control is returned to the instruction following the XEQ unless a transfer of control was executed.

Note: Executing a stackop with the right stackop not a NOP is illegal.

Indicators = Set by executed instruction or StackOps.

130. IXIT <Interrupt Exit> Mini OpCode=17
 CIR (14:2)=0

This instruction is used only to exit from interrupt service routines and the dispatcher. Several alternative routes are available, depending on the setting or clearing of certain flags. These are:

- 1) Dispatcher exit to new process. This path is taken if the dispatcher flag is set upon entering the instruction.
- 2) Return to interrupted process. This path is taken after the last interrupt has been serviced and (Q=QI) = 0. Note that we do not automatically enter the dispatcher.
- 2a) Same as (2) except a DISP instruction had been executed while pseudo-disabled (i.e. an attempt was made to enter the dispatcher).
- 3) Return to an interrupted Interrupt Service Routine. This path is taken when an Int. Service routine is completed and Q(0) = 0, Q(1:15) is not equal to 0 (a lower priority interrupt had been interrupted).



- 4) Return to interrupted dispatcher. This path is taken upon completing the servicing of an interrupt that occurred while in the dispatcher. This is detected by $Q(0) = 1$ (set by the firmware interrupt handler.) and $QI(0) = 0$ (i.e. Q is not equal to QI).
- 4a) Same as (4) except a DISP instruction had been executed while pseudo-disabled.
- 5) Start dispatcher instead of returning to interrupted process. This is signified by $QI(0) = 1$, set by executing a DISP instruction.
- 6) Restart dispatcher instead of returning to interrupted dispatcher. This is signified by $QI(0) = 1$, set by executing a DISP instruction.
Note: (5) & (6) are treated as identical cases.

IXIT assumes it is executing on the ICS, with a standard 4-word marker at Q , $Q-1$, $Q-2$ & $Q-3$, the DB of the stack marker environment at $Q+1$, the previous interrupt mask at $Q+2$ and the device number of the interrupting device at $Q+3$. In addition, when dispatching a new process or returning to an interrupted process, the labels found in the CTCB (Listed below) are used to set up the new environment.

(CTCBP) contains DST label pointing at User DST
(CTCBP + 1) contains the Stack DST label
(CTCBP + 2) contains Global (DB-DL) DST label
(CTCBP + 3) contains the UCST DST label

Note: In launching a process off the ICS all labels are evaluated before the environment is changed and will trap on the ICS if an error is encountered. The stack register values also are checked with a special check of S against $ZMAX$ (not Z) and if in error a launch error trap will occur.

This is a priveleged instruction.
Indicators = Unaffected

```

131. DISP    <Dispatch>          Spec OpCode=03
                                   CIR (14:2)=2

    if (ICS FLAG = 1 OR (QI-4) <> 0)
      Then (QI) := %100000 else
        begin
          S := S+4;    <<for stack marker>>
          (S-3) := X; (S-2) := P+1 - PB;
          (S-1) := STATUS; (S) := S-Q;
          (SB-1) := S - SB; <save user S>
          Q := QI; Z := ZI; DISP FLAG := ICS FLAG := 1;
                                   <set up ICS>

          S := Q + 1;
          SB := QI - %100;
          DB := (QI+1);
          DL := DB + %1000;
          STATUS := (Q-1); X := (Q-3);
          Set PB & PL from CST;    (for STATUS)
          P := PB + (Q-2);
        End;

```

This instruction is used to enter the dispatcher directly from a user program (if privileged) or from the ICS after servicing all pending interrupts. Execution of this instruction is the only way to enter the dispatcher.

This is a privileged instruction.
 Indicators = If DISP taken then CCE else CCG.

The following two instructions must be executed in pairs within a given process (i.e. for every PSDB there must be a PSEB somewhere in the following code). The main memory location QI-4 (Dispatcher Disable Flag), while non-zero, prevents dispatching a new process. The results of these instructions are tested in IXIT and DISP.

```

132. PSDB    <Pseudo Interrupt Disable>      Spec OpCode=03
                                                CIR (14:2)=1

```

```

    (QI-4) := (QI-4) + 1;

```

The Dispatcher Disable Flag (= (QI-4)) is incremented. While this flag is non-zero, the dispatching of a new process is inhibited. This is a privileged instruction.
 Indicators = Unaffected

133. PSEB <Pseudo Interrupt Enable>

```
Spec OpCode=03
CIR (14:2)=3
```

```

(QI-4) := (QI-4) - 1;
if (QI-4) = -1 then SYSTEM SOFT HALT;
if (QI-4) = 0 then
  if (QI(0:1)) = 1 then
    if DISP Flag = 1 then <interrupted while dispatching>
      begin
        (QI) := 0;
        Abort dispatcher and restart;
      end
    else EXECUTE DISP INSTRUCTION;
  end
end

```

The dispatcher disable flag (QI-4) is decremented by this instruction. If the flag previously was zero, a system soft halt results indicating too many PSEB's. If it now equals zero, all "pseudo-disables" have been removed, and it is now allowable to dispatch a new process. If a dispatch request is present, i.e., QI(0) = 1 (signifying an attempt to dispatch while pseudo-disabled), AND we are not already in the dispatcher the sequence followed by the DISP instruction is executed. If we are already in the dispatcher when this instruction is executed, QI(0) is cleared, the dispatcher is aborted and then restarted at the beginning. If the disable flag is nonzero, or if no dispatcher request is present, execution continues at the next instruction. This is a privileged instruction.

Indicators = If (QI-4) = 0 then CCE else CCG

134. DBUG <Debug interrupt call>

Spec OpCode=14

```
S := S + 1;
(S) := X;
S := S + 1;
(S) := P + 1 - PB;
S := S + 1;
(S) := STATUS;
S := S + 1;
(S) := S - Q;
Q := S;
Debug label := %107401;
P := evaluation of Debug label;
```

Execution of this instruction transfers control to an Internal Interrupt procedure (CST = 1, STT = 17 octal) for special system processing, depending on the K-field value as listed in the following table.

K -field	Reserved for
-----	-----
0	System debug breakpoint processing
1	" " " "
2	" " " "
3	Symbolic debug breakpoint processing
4	" " " "
5	" " " "
6	(to be defined)
7	Graceful operating system death
10	Graceful subsystem death
11/17	(to be defined)

Indicators = unaffected

35. IPL <Initial program load> Spec OpCode=15
CIR(12:4)=0

This instruction expects the next instruction in code to be the complement of CIR and traps to Priv mode violation if not. It also expects a device number in the right byte of TOSA and a cylinder address in the left byte of TOSA. Starting First the contents of memory at address %37 (dump disc address) is read and saved. Then starting at sector 1 of the designated cylinder address, 65K words are loaded into the first bank in memory from the device specified. The stack then is set to the interrupt control stack location designated by the new contents of memory addresses 5 & 6 with SB set to QI - 100. The saved original contents of address %37 is stored in the stack at QI+3 and S is set to this location. If the IPL instruction has been immediately preceded by an SDMP instruction QI+3 now contains the device # and cylinder address used by the SDMP instruction, otherwise it will contain garbage. This information will be used by the load software to complete the system dump if so desired. Code segment 1 then is set up and P is set to the value contained in the cold load entry STT44. Software action now commences.

This is a privileged instruction

136. SDMP <System Dump> Spec OpCode=15
CIR(12:4)=1

This instruction expects the next instruction in code to be the complement of CIR and traps to Priv mode violation if not. It also expects a device # in the right byte of TOSA and a cylinder address in the left byte of TOSA. The first 65K words of memory are dumped to the device specified starting at sector 1 of the cylinder address designated. The contents of the hardware registers of the CPU plus the dump disc address then are stored in memory starting at address 16 and then an additional 128 words, starting at address 0 and containing these values, are dumped to the device specified at the next sequential disc address. The order of the registers stored are as follows:

16 - SB Bank
 17 - SB
 20 - Q
 21 - S
 22 - Z
 23 - DB Bank
 24 - DB
 25 - DL
 26 - PB Bank
 27 - PB
 30 - P
 31 - PL
 32 - DST Bank
 33 - DST
 34 - X
 35 - STATUS
 36 - CIR
 37 - TOSA <<dump cylinder address and device #>>

During the dump TOSA is stored in memory at address %37 and the stack is popped. P is incremented to P+2 to skip the compare word and software execution resumes.

This is a privileged instruction.

Indicators = CCE if dump is successful else CCL

Move Instructions

MEMORY OPCODE = 02
 SUB OPCODE 2 = 00
 MOVE OPCODE = CIR(8:3)
 SDEC = CIR(14:2)
 CCF = CIR(11:2)
 CIR(11:1) = Numeric
 CIR(12:1) = Alphabetic
 PB/DB (0/1) = CIR(11:1)
 Left to Rt/Rt to Left = CIR(13:1)

The following is a description of the address bounds checking done with byte strings.

```

Procedure BNDSTEST(Label,Type); <<begin bounds test>>
  Begin
    If NOT (Label Base <= Label Address <= Label Limit)
      then Bounds Violation trap;
    Return;
  end;
  
```

```

137. MOVE      <Move words>          Move OpCode=00
D := If CIR(13:1)=0 then 1 else -1;
DeltaS := if D = 1 then (S) -1 else -(S) + 1;
If CIR(11) = 0 then <<PB relative source>>
    If NOT((PB<=(S-4)+PB<=PL)AND(PB<=(S-4)+PB+DeltaS<=PL))
    Then Bounds Violations Trap else GOTO M
else Begin    <<Data label source pointer>>
    Source data label := (S-4,S-3);
    if not((Label Base <= Label Address <= Label limit) AND
        (Label Base <= Label Address+DeltaS<=Label Limit))
        then Bounds Violation trap;
    end;
M:          <<Data label target pointer>>
Target data label :=(S-2,S-1)
if NOT((Label Base <= Label Address <= Label limit) AND
    (Label Base <= Label address + DeltaS <= Label limit))
    then Bounds Violation Trap;
    <<Begin Move>>
while (S) <> 0 do
    begin
        (target address) := (source address);
        target label offset := target label offset + D;
        source label offset := source label offset + D;
        (S) := (S) - 1;
    end;
S := S - Logical SDEC

```

The instruction expects a word count in the TOS, a target label pointer in S-2,S-1, and a source label pointer in S-4,S-3. If the Direction bit, CIR(13:1), is zero, the words are moved in a left-to-right manner (increasing addresses) from the source area to the target area. A "one" direction bit will cause a right-to-left move from the source area to the target area. Prior to the move, bounds tests are applied to both the initial and final values of the source and target addresses. S is decremented to delete from the stack the number of logical parameters indicated in the SDEC field after the move is completed.

Indicators = Unaffected

```

138. MVB      <<Move Bytes>          Move OpCode=01

D := If CIR(13:1) = 0 then 1 else -1;
IF CIR(11)=0 Then Begin
    Type:= Prog;
    Source:= (S-4); <<Program label source>>
end
Else Begin
    Type:= Data;
    Source:= (S-4,S-3); <<Data label source>>
end;

```

```

Target:= (S-2,S-1); <<Data label target>>
while (S)<>0 Do
  Begin
    BNDSTEST(Source,Type);
    BNDSTEST(Target,Data);
    (Byte target address) := (Byte source address);
    Source label offset := Source label offset + D;
    Target label offset:= Target label offset + D;
    (S) := (S) - 1;
  end;
S := S - logical SDEC.

```

This instruction expects a byte count in the TOS, a source byte label pointer in S-4 if program relative, or in S-4,S-3 if data relative, and a target byte label pointer in S-2,S-1. If the direction bit, CIR(13:1), is zero, the bytes are moved in a left-to-right manner (increasing addresses) from the source area to the target area. A Direction bit =1 will cause a right-to-left move from the source area to the target area. The pointers, and byte count are updated with the move. S is decremented to delete from the stack the number of logical parameters indicated in the SDEC field after the move is completed.

Indicators = Unaffected

139. MVBW <Move Bytes While> Move OpCode=04

```

Source := (S-4,S-3); <<Source Byte Data Label>>
Target := (S-2,S-1); <<Target Byte Data Label>>
BSAD := source byte label address;
BTAD := target byte label address;
MOVING := TRUE;
while MOVING do
  begin
    BNDSTEST(Source,Data);
    If (CCB on (BSAD) < CCF) then
      begin
        BNDSTEST(Target,Data);
        (BTAD) := If CIR(13) = 1 <<upshift bit>> and (BSAD)
                  is a lower-case letter then upshift (BSAD)
                  else (BSAD);
        Source label offset := Source label offset + 1;
        Target label offset := Target label offset + 1;
        X := X + 1;
      end
    else MOVING := FALSE;
  end;
S := S - logical SDEC.

```

This instruction expects a source byte data label pointer in S-3,S-2, and a target byte data label pointer in S-1,S. As long as the byte in the source string is one of the types specified by CCF, and the addresses are valid, the byte is moved. If the byte is to be moved and the upshift bit is on and the byte is a lower-case letter, the target byte will be an upshifted copy of the source byte. X is incremented by the number of bytes moved. S is decremented to delete from the stack the number of logical parameters indicated in the SDEC field.

Indicators = CCB on the last character scanned

```
140. SCW      <Scan while>      Move OpCode=02

D := if CIR(13:1) = 0 then 1 else -1;
If CIR(11:1) = 0 then
  begin
    Type := Prog;
    Source := (S-4); <<program label source>>
  End
Else
  Begin
    Type := Data;
    Source := (S-4,S-3); <<Data label source>>
  End;
BSAD := Source Byte Label Address;
BNDSTEST(Source,Type);
while (BSAD) = (S-1(8:8)) do
  Begin
    Source label offset := Source label offset + D;
    X := X + 1;
  End;
CARRY := If (BSAD) = (S(0:8)) then 1 else 0;
CC := CCB on (BSAD);
S := S - Logical SDEC;
```

This instruction expects a a terminal character in the left byte and a test character in the right byte of S. The Source label in S-2,S-1 is a program label or data label depending on bit 11 of the CIR. If the address is valid, bytes are scanned until the source string presents a character different from the test character. If the terminating character is the same as the left byte of the TCS, the CARRY bit is set. If the direction bit, CIR(13:1), is zero, bytes are scanned in a left-to-right manner (increasing addresses). A direction bit =1 will cause a right to left scan of the source string. X is incremented by the number of bytes scanned. S is decremented to delete from the stack the number of logical parameters indicated in the SDEC field. (S-2,S-1) is considered a logical parameter.

Indicators = CCB on last character scanned, CARRY

141. SCU <Scan Until> Move OpCode=03

```

D := If CIR(13:1) = 0 then 1 else -1;
If CIR(11:1) = 0
  Begin
    Type := Prog;
    Source := (S-4); <<Program label source>>
  End
Else
  Begin
    Type := Data;
    Source := (S-4,S-3); <<Data label source>>
  End;
BSAD := Source byte Label address;
BNDSTEST(Source,Type);
while (BSAD) <> (S-1(8:8)) AND (BSAD) <> (S-1(0:8)) do
  begin
    Source label offset:=Source label offset + D;
    X := X + 1;
  End;
CARRY := If (BSAD) = (S(0:8)) then 1 else 0;
S := S - logical SDEC;

```

This instruction expects a terminal character in the left byte and a test character in the right byte of S. The source label in S-2,S-1 is a program label or data label depending on bit 11 of the CIR. If the address is valid, bytes are scanned until either the terminal or the test character is encountered. If the last character scanned is the same as the left byte of the TOS, the CARRY bit is set. X is incremented by the number of bytes scanned. S is decremented to delete from the stack the number of logical parameters indicated in the SDEC field.

Indicators = CARRY

142. CMPB <Compare Bytes> Move OpCode=05

```

D := If CIR(13:1) = 0 then 1 else -1;
If CIR(11) = 0 then Begin
  Type := Prog;
  Source := (S-4); <<Program label source>>
end else
  Begin
    Type := Data;
    Source := (S-4,S-3); <<Data label source>>
  end;
Target := (S-2,S-1); <<Data label source>>
BSAD := Source Byte Label Address;
BTAD := Target Byte Label Address;

```

```

While (S) <> 0 AND (BTAD) = (BSAD) Do
  Begin
    BNDSTEST(Source,Type);
    BNDSTEST(Target,Data);
    Source label offset:=Source label offset + D;
    Target label offset:=Target label offset + D;
    (S)      := (S) - 1;
    X := X + 1;
  End;
CC := If (S) = 0 then CCE .
     else if (BTAD) > (BSAD) then CCG else CCL;
S  := S - logical SDEC;

```

This instruction expects a byte count in the TOS, a source byte label pointer in S-4 if program relative source, or in S-4,S-3 if data relative source, and a target byte label pointer in S-2,S-1. If the Direction bit, CIR(13:1), is zero, bytes from the source area are compared to bytes from the target area in a left-to-right manner. A Direction bit = 1 will cause a right-to-left comparison. The instruction terminates when either a comparison fails or the byte count in the TOS reaches zero. X is incremented by the number of bytes that compared equal. S is decremented to delete from the stack the number of logical parameters indicated in the SDEC field.

```

Indicators = If (byte count = 0) then CCE else
              If (final target byte > final source byte.) then
                CCG else CCL.

```

I/O and Interrupt Instructions

For the I/O instruction assume the following definitions:

K is a four bit field $0 \leq K \leq 15$.

All I/O and Interrupt control instructions are PRIVILEGED except RMSK.

If Chan/dev# is illegal (i.e. not in the range #10-177) a trap to segment 1,STT #12 will occur.

```

143. SIOP          <Start I/O Program>          Spec OpCode=07
                                                Cir (12:4)=0

(S) = Channel program pointer
(S-1) = Device number
If DRT3(2:1) = 1 then CCL else
  Begin          <<abort bit above is clear>>
    CC := CCG;
    If DRT3(0:1) = 0 then
      Begin      <<00,01>>
        If DRT3(1:1) = 0    <<halted>>
          OR (DRT3(1:1)=1 AND DRT3(15:1)=1)
            <<halting in WAIT>>
        then begin
          DRT1 := (S); <<Chan ptr to DRT1>>
          DRT3(0:2) := 3; <<set starting state>>
          Send SIO command to Channel;
          CC := CCE;
        end;
      End;
    End;
  End;
S := S - 2;

```

This instruction expects a channel program pointer in S and channel/device number in S-1. The third word of the device DRT entry (DRT3) is read with a semaphore read. This delays execution of the instruction by a possible independent program channel until all the information is in place. If bit 2 of DRT3 (the abort bit) is 1, the instruction is aborted and CCL is set. If the channel program is halted (if bits 0,1 of DRT3 are both 0), or if an HIOP instruction has been issued but not yet serviced and the channel is in a wait instruction state (bits 0,1 of DRT3 are 0 and 1 and bit 15 of DRT3 is a 1), then the channel program pointer in S is placed in DRT0 of that device, bits 0,1 of DRT3 are set to 1,1 (SIO starting state), an SIO command is sent to the channel and CCE is set. Otherwise if the above conditions are not met then CCG is set. This is a privileged instruction.

Indicators = Condition Code

```

144. HIOP          <Halt I/O Program>          Spec OpCode=07
                                                Cir (12:4)=1

(S) = Device #
If DRT3(0:2) <> 0 then
  Begin
    if DRT3(0:1) = 1 then
      Begin
        If DRT3(1:1) = 1 then DRT3(15:1) := 1;
        DRT3(0:2) := 01;
        Send HIOP command to device;
      end;
    CC := CCE;
    If DRT3(15:1) = 0 then CC := CCG;
  End
Else CC := CCE;    <<Already halted>>
S := S - 1;

```

This instruction expects a device number on the top of the stack. If DRT3(0:2) of that device indicates that the device channel program is starting or running, a halt I/O program command is sent to the channel to stop execution of that device's channel command program at the occurrence of the next WAIT channel command. If starting, a halting in WAIT state is set to properly terminate the channel program. If now the channel program is halting but not yet halted and not in a WAIT instruction, CC is set to CCG. All other states are set to CCE. This includes the already halted state. When halted, the DRT entry for that device points at the WAIT instruction at which the channel program halted, bits 0 and 1 of DRT3 are set to 0. If the channel program was not at a wait instruction when the HIOP was issued, an interrupt request will be generated when the channel program is halted.
This is a privileged instruction.

Indicators = Condition Code



146. RIOC <Read I/O Channel> Spec OpCode=12

```
Channel Command := (S);
Send RIOC command to Channel;
If DRT3(2:1) = 1 then <<abort bit set?>>
Begin
  CC := CCL;
  Terminate instruction;
End else
Begin
  (S) := Data read from the channel;
  CC := CCE;
End;
```

This instruction expects an IMB read channel command to be in the top of the stack. If the abort bit is not set for the device, the RIOC command from the TOS is sent to the channel (or channels if global command) and the data read is pushed on the stack.
This is a privileged instruction.
Indicators = Condition Code

147. WIOC <write I/O Channel> Spec OpCode=13

```
Channel Command := (S-1);
Data word := (S);
If DRT3(2:1) = 1 then <<abort bit set?>>
Begin
  CC := CCL;
  Terminate instruction;
End Else
Begin
  Send WIOC Command and data word to channel;
  S := S - 2;
  CC := CCE;
End;
```


The TCS must contain the mask word. The 16 bit mask is transmitted to all channels. A copy of the MASK is stored in memory loc. 7. If no error is detected, the TCS is deleted. This is a privileged instruction.

Indicators = Condition Code

This function is executed by loading \$120000 on the stack and issuing an RIOC instruction. This sequence identifies the presence of I/O channels on the system. The presence of each channel is represented in a word pushed on the stack by a "1" state of the bit position in the word corresponding to its jumpered channel number. Thus the (S) yields bits = 0 at channel numbers not occupied by a channel.

Register Control Instructions

157. PSHR N <Push registers> Sub OpCode 2=11

```

if CIR(15) = 1 then PUSH(S-SB);
if CIR(14) = 1 then PUSH(Q-SB);
if CIR(13) = 1 then PUSH(STATUS);
if CIR(12) = 1 then PUSH(Z-SB);
if CIR(11) = 1 then PUSH(DL-DB);
if CIR(10) = 1 then
  Begin
    PUSH (UDST BANK);
    PUSH (UDST);
  End;
if CIR(9) = 1 then
  Begin
    PUSH (DB BANK);
    PUSH (DB);
  End;
if CIR(8) = 1 then
  Begin
    PUSH (SB BANK);
    PUSH (SB);
  End;
if CIR(8:8) = 0 then
  Begin
    S := S + 3;
    PUSH(PB BANK);
    PUSH(PB);
    PUSH(P);
    PUSH(PL);
    PUSH(ISR); <<Internal hardware status register>>
  End;

```

where...

```

PUSH(REGISTER) is defined to be
Procedure PUSH (Register);
Value register; Integer register;
Begin
  S := S + 1;
  (S) := REGISTER;
End;

```

Note for pushing S-SB the following occurs:

```

S := S + 1;
(S) := S - SB - 1;

```

The registers specified by CIR (8:8) are pushed onto the stack. Note: this instruction executes in the sequence shown.
Indicators = Unaffected

158. SETR N <Set registers> Sub OpCode 2=17

```

if (CIR(8:5) <> 0 and NOT Priv Mode)
  then MODE VIOL. TRAP;
if CIR(8) = 1 then Begin POP(SB); TEMP := (SB BANK);End;
if CIR(9) = 1 then Begin POP(DB); POP(DB BANK);End;
if CIR(10) = 1 then Begin POP(UDST); POP(UDST BANK);End;
if CIR(11) = 1 then POP(DL + DB);
if CIR(12) = 1 then POP(Z + SB);
if CIR(13) = 1 then if PRIV MODE then
  POP(STATUS(0:16)) else
  POP(STATUS(2:1,4:4));
if CIR(14) = 1 then POP(Q + SB);
if CIR(15) = 1 then POP(S + SB);
if CIR(8) = 1 then SB BANK := TEMP;

```

where...

```

POP(REGISTER) is defined to be
procedure POP(register);
begin
  Register := (S);
  if Register = "S"
    then S := S - 1;
end;

```

The registers specified by CIR (8:8) are filled by an absolute value from the TOS for UDST, STATUS, DB, DB bank, SB, SB bank and an absolute value computed by adding to the TOS for the others. After each register is set, the TOS is deleted.
Indicators = Unaffected except by loading STATUS reg.

159. ADDS N <Add to S> Sub OpCode 3=12

```

if N > 0 then S := S + N;
if N = 0 then
  Begin
    Temp := (S);
    S := S-1; <<This leaves original (S) undefined>>
    S := S + Temp;
  End;

```

If the eight bit operand is zero, the content of the TOS is added to S - 1 (the TOS is popped and (S) spaces are then pushed onto the STACK). Otherwise the operand is added to S (N spaces are pushed onto the stack).
Indicators = Unaffected


```
Clock comparison register := (S);
S := S - 1;
```

Indicators = unaffected

```

164. PLDA      <Privileged load from absolute      Mini OpCode=15
               address>                               Cir(12:4)=0
      S      := S + 1;
      (S)    := (X);

```

Indicators = CCA

```

165. LSEA      <Load single word from extended          Mini OpCode=16
               address>                                   Cir (14:2)=0
               S := S + 1;
               (S) := ((S-2),(S-1));

```

Indicators = CCA

```

166. LDEA      <Load double word from extended      Mini OpCode=16
              address>                               Cir(12:4)=2
      S      := S + 2;
      (S-1)   := ((S-3), (S-2));
      (S)     := ((S-3), (S-2) + 1);

```

Indicators = CCA on double-word

171. MCS <Memory command and status return>

Mini Opcode=17

$$\text{CIR}(12:4)=2$$
$$S := S + 1;$$

(S) := (S - 2, S - 1)Special Memop

The special control command in S-1,S is sent over the address bus to the memory controller specified by bits 12, 13, 14 of (S-1) and a word of status is returned to the TOS.

This is a privileged instruction.

Indicators = Condition code

Semaphore Instruction

This instruction is used to implement a mechanism whereby a memory location may be read and altered without modification by another module during the read-alter sequence.

72. SEML

<Semaphore Load>

Mini OpCode=17

$\text{Cir}(12:4)=1$

S-1,S contains a data label

$$S := S + 1;$$

(S) := (Data label address);

```
(Data label address) := -1;
```

```
<<NOTE: This occurs in memory as a special
uninterruptable read contents and
write all 1's sequence>>
```

CARRY := If (S) = -1 then 1 else 0;

The content of the memory location addressed by the data label in the two top elements of the stack is read by a special memory operation that reads the location and replaces its contents with 1's before handling the next memory request. The original contents of this memory location is pushed on the stack.

Indicators = CCA on the new TCS

CARRY If $(E) = -1$

List Search Instructions

```
173. LLSH      <Linked List Search>      Mini OpCode=14
                                         Cir (12:4)=1
```

```

WHILE X > 0 AND ((S-1,S) + (S-3)) < (S - 2)
AND (S-1,S)<>0 do
Begin
    (S)      := ((S-1), (S) + 1);
    (S-1)    := ((S-1), (S));
    X        := X - 1;
End;

```

S-1, S contain an absolute address (consisting of a bank addr. and a 16 bit absolute addr. within the bank) into a linked list. Each double word link in the list is an absolute memory address which points to the next link. The third word in the stack (S-2) is a test word. The fourth word in the stack (S-3) is an offset which indicates the position, relative to each link, of a target number. The count is in the index register.

At each step the test word is compared to the target number. If the test word is logically less than or equal to the target number, or the link address is equal to zero, the instruction terminates. Otherwise, (S-1), (S) is replaced by the next link, the count (index register) is decremented, and the instruction repeats again. This is a privileged instruction.

```

Indicators = CCL if terminated on  $x = 0$ 
             CCE if terminated on  $((S-1), (S) + (S-3)) \geq (S-2)$ 
             CCG if terminated on  $((S-1), S) = 0$ 

```

Label Instructions

174. LLBL N <Load Label> Sub OpCode 3=07

```

if N > (PL(8:8)) then STT violation trap else
Begin
  If ((PL-N)(0:1)) = 1 then
    Begin      <<external>>
      S := S + 1;
      (S) := (PL-N);
    End else   <<internal>>
  if N > %177 then STT violation trap else
    Begin
      S := S + 1;
      (S(0:1)) := 1;      <<convert to external>>
      (S(1:7)) := N;
      (S(8:8)) := STATUS (8:8);
    End;
End;

```

The label in the STT table at PL-N is loaded onto the TOS if it is in the STT, i.e., N is less than or equal to STTL. If the label is internal and N is less than or equal to %177 the label is converted to external type and stacked. If the label is internal and N is greater than %177 then a STT violation trap is invoked.
Indicators = Unaffected

175. CWLB <Convert word label to byte label> Mini OpCode=14
CIR(12:4)=2

```
(S-1,S) contains a data label;
If (S-1(0:1)) = 1 then
  Begin    <<DB label>>
    If (S-1(1:1)) = 0 then (S-1) := ASL(S-1)
    else data label overflow;
  End;
If (S-1(0:2)) = 0 then
  Begin    <<SB label>>
    If (S-1(2:1)) = 0 then (S-1) := ASL(S-1)
    else data label overflow;
  End;
If (S-1(0:3)) = 3 then
  Begin    <<Data segment label>>
    If (S(0:1)) = 0 then (S) := ASL(S)
    else data label overflow;
  End;
```

The offset field in the data label on the stack is multiplied by two and checked for overflow.

Indicators = Unaffected

176. CBLW <Convert byte label to word label> Mini OpCode=14
CIR(12:4)=3

```
(S-1,S) Contains a byte data label;
If (S-1(0:1)) = 1 OR (S-1(0:2)) = 0 then
  Begin    <<DB or SB label>>
    (S-1(2:14)) := (S-1(1:14));
    (S-1(1:1)) := 0;
    (S-1(0:1)) := (S-1(0:1));
  End
else (S) := ARS(S);
```

The offset field in the data label on the stack is divided by two.

Indicators = byte address to carry

177. CLAA <Convert Label to Absolute Address> Mini OpCode = 15
CIR(12:4)=2

(S-1,S) := Address evaluation of Data label in (S-1,S);
CARRY := If LabelOffset(15:1)=1 then 1 else 0;

The Data label contained in the top two elements of the stack is replaced by the absolute address pointed to by that data label. The byte address of the label is placed in CARRY. This is a privileged instruction.

Indicators = unaffected

178. GEL <Generate Extended Label> Mini OpCode = 15
CIR(12:4)=3

(S-1,S) Contains a Data segment label
(S-3,S-2) contains double integer index
DX := (S-3,S-2);
Offset := (S);
Dlabell := (S-1);
DSTP := if Dlabell.(3:1) = 1 then DST register
 else %1000; <<address of DST>>
If (DSTP) < Dlabell.(4:12)*4 then DST violation;
DSTE := DST + Dlabell.(4:12) * 4; <<address of DST entry>>
RelAddr := DX + If Dlabell.(0:3)=3 then Offset else 0;
Length := (DSTE) + 1;
If RelAddr > Length := (DSTE) then
 begin
 If DSTE(1).(0:1) = 1 then
 begin
 addseg := integer(RelAddr/length);
 <<number of data segments>>
 If (Dlabell.(4:12) + addseg > DSTE+1.(4:12)
 then Bounds Violation;
 DSTE := DSTE + addseg * 4;
 RelAddr := RelAddr - Length;
 end else
 begin
 While RelAddr > Length do
 begin
 RelAddr := RelAddr - Length;
 DSTE := DSTP + DSTE(1).(4:12) * 4;
 <<link DST>>
 If DSTE(1).(4:12) = 0 then
 Bounds Violation; <<Link DST was 0>>
 Length := (DSTE);
 end
 end
 end
 end
 end
Dlabell.(4:12) := DSTE/4;
S := S - 2;
(S - 1) := Dlabell;
(S) := RelAddr;
(S - 1).(3:1) := 1;



The top two words in the stack contain a data label pointing to a large array located in multiple data segments. The fourth and third words in the stack contain a double integer to be used instead of the index register for indexing into the large array. Using this double integer, normal array indexing is performed through linked DST entries to obtain the new indexed address. The final DST # and offset within this segment of the element address are then used to generate a new data label that points directly to this element. The original data label and double index values are deleted from the stack and the new label is pushed on the stack.
Indicators = unaffected

```

179. VLAB                <validate Label>                Mini OpCode=17
                                                                CIR(12:4)=3

S-3 contains an SB relative stack fence
S-2(15:1) contains the instruction mode bit
S-1,S contains data label;
If S-2(15:1)=1 and STA(0:1)=0 then set CCL else
Begin
  CC := CCE
  If (S-1(0:1)) = 1 then
    If (S-1(1:15)) > DL-DB set CCL;
  If (S-1(0:2)) = 0 then
    If (S-1) > (S-3) then set CCL;
  If (S-1(0:2)) = 1 then
    If (Valid address AND (If S-2(15:1)=PRIV
      Or If DST MODE bit=user.) then CCG else CCL;
End;
S := S - 2;

```

This instruction expects a data label in the top two words in the stack, an instruction mode bit in S-2(15:1) and a offset stack fence or limit in the fourth word in the stack. Labels that point into the stack are bounds tested using the limit value in S-3 instead of S. Labels that point into the global data area (DB rel) are tested in the normal way. Labels that point into data segments are tested to give a valid address and to pass the PRIV mode test with the mode bit in S-2.

Indicators = CCE if valid user address in the stack
 or global data area.
 = CCG if valid data segment label.
 = CCL if invalid address or priv segment
 with user mode bit.

Extended Instruction Set (Decimal Arithmetic)

This section contains 16 instructions that operate on decimal data. Some also use, in addition to the decimal data type, another data type. This group includes conversions to and from decimal, add/sub/mult/divide, test, negate, absolute, and edit.

The parameters on the stack for decimal instructions are grouped into items. Each item contains 3 words. Words 0 and 1 form a 2 word byte label addressing the low address of the string. Word 2 describes the length of the string (in decimal digits). Word 2 also indicates whether half-rounding is performed and, where the implied decimal point is. Since 2 word byte labels cannot reference PB relative operands, there is no PB relative addressing in decimal instructions.

There exists an S-dec, for POPing of the stack after the instruction. Bit 15 of the instruction op code indicates that the result ITEM is to be POPed if a 1 and, not POPed if a 0. In any case, all source ITEM's are POPed. In the case of PLIV, the remainder is always POPed. In the case of CVPE, the extended result is never POPed.

The following figure shows the position of each parameter in an item.

```

ITEM = word 0 : 2 word Byte :
          1 : Label :
          :-----:
          2 : H : DP : Len :
          :-----:
              0 :1 7:8 15
  
```

H = 1 implies Half-Rounding is performed.

DP = the number of digits following the implied decimal point

Len = the number of digits in the string $0 \leq \text{Len} \leq 28$
 (Note: If Len = 0 then the label is not bounds checked.)

Half-Rounding may cause the result to be incremented. If the left most digit that has been right truncated is a 5 or greater, then the stored result is incremented. The Half-Rounding bit is ignored except on the result ITEM of an PADD or PSUB instruction.

Notation for identifying the items, Item 0's word 2 is at location S-1. Item 1's word 2 is at location S-4. And Item 2's word 2 is at location S-7, and Item 3's word 2 is at location S-10. Different instructions will require a different number of items. The usage placed on each item depends upon the instruction being performed. Generally speaking, Item 2 is a destination pointer, and Item's 0 and 1 are source pointers. A four item stack for a PLIV (with one zero word) instruction would look like this:

S-12	: 2 word Byte	:	
S-11	: Label	:	Item 3 - Pointer to the Quotient
S-10	: : DP : Len	:	
S- 9	: 2 word Byte	:	
S- 8	: Label	:	Item 2 - Pointer to the Remainder
S- 7	: : DP : Len	:	
S- 6	: 2 word Byte	:	
S- 5	: Label	:	Item 1 - Pointer to the Dividend
S- 4	: : DP : Len	:	
S- 3	: 2 word Byte	:	
S- 2	: Label	:	Item 0 - Pointer to the Divisor
S- 1	: : DP : Len	:	
S	: ZERO	:	Interrupt restart word.

In the case of PADD or PSUB the stack would look as follows.

S - 9	: 2 word Byte	:	
S - 8	: Label	:	Item 2 - Half Roundable Result
S - 7	: H : DP : Len	:	
S - 6	: 2 word Byte	:	
S - 5	: Label	:	Item 1 - Source 1 to be added to source 0.
S - 4	: : DP : Len	:	
S - 3	: 2 word Byte	:	
S - 2	: Label	:	Item 0 - Source 0 to be added to source 1.
S - 1	: : DP : Len	:	
S	: Z E R O	:	Interrupt Restart word

Note: The ZERO pushed on the stack last is an interrupt word and is to be pushed for ALL decimal instruction, with the exception of EDIT.

Traps

The traps which can occur as a result of a Decimal instruction execution are as follows:

Trap Name	Parameter	Comment
Overflow/PACKED	!08	Result exceeds field size.
Overflow/BINARY	!09	Significance lost.
Invalid Argument/PACKED	!28	Decimal Divide by Zero.
Invalid Character/EXTENDED	!33	Not Decimal digit.
Invalid Character/ZONED	!37	Not Decimal digit.
Invalid Character/PACKED	!38	Not Decimal or ASCII digit.
Invalid Character/BINARY	!39	Not Decimal digit.
Invalid Control word/EXTENDED	!43	Len or DP too large or source.
Invalid Control word/ZONED	!47	Len too large.
Invalid Control word/PACKED	!48	Len or DP too large.
Invalid Control word/BINARY	!49	Len or DP too large.

If one of these conditions occurs and user traps are enabled, the S-dec is preformed and then a copy of the destination ITEM is PUSHed (3 words, and only the Quotient in the case of PDIV). Then a transfer is made to the Trap handler, with the appropriate parameter. If user traps are disabled, overflow is set, the S-dec is performed and the instruction ends.

Condition codes are set on the stored result.

Decimal Zero and Overflow

Decimal zero can be represented as a positive zero, negative zero, or an absolute zero. All instructions produce only a positive zero, except CVAP.

The left-truncated result is stored and decimal overflow occurs if all significant digits to the left of the implied decimal point do not fit in the result field. If this happens, then if a) User traps are ENABLED, Trap %010 occurs or b) User traps are DISABLED, Overflow is set, the S-dec is preformed, and the instruction ends.

Firmware Support

Some of the Decimal instructions will be done in firmware and some in software. Those in firmware are PADD, PSUB, PCMP, CVAP, CVPA, MVP, PABS, PTST, PNEG. Those in software are PMPY, PDIV, CVPB, CVBP, CVEP, CVPE, EDIT.

The instructions supported by the firmware increase S by 38 words (decimal) temporarily during execution to perform the interrupt restart process. Also, if any operand of these instructions crosses two Data Segment boundaries (Linked Data Segments), then a DST violation will occur. This means that the minimum D.S size is 7 words.

Arithmetic Decimal Instructions

180. PADD <Add Packed>

OP CODE is 034000

PADD expects 3 items, and a zero, where (ITEM 2) := (ITEM 1) + (ITEM 0). CCA is set on the stored result. If in either source specification the length is zero, then that corresponding operand is considered to be zero and the label is not checked for validity. This feature is used in the MVP instruction.

Indicators: CCA, OVFL

Traps:	Overflow/PACKED	!08
	Invalid Character/PACKED	!38
	Invalid Control word/PACKED	!48

181. PSUB <Subtract Packed>

OP CODE is 034004

PSUB is exactly like PADD except that (Item 2) := (Item 1) - (Item 0). Traps and Indicators are the same.

182. PCMP <Compare Packed>

OP CODE is 034010

PCMP expects 2 items, and a zero, where (Item 1) is compared with (Item 0) and CCA is set to CCG if Item 1 is greater than Item 0, CCE if equal, and CCL if Item 1 is less than Item 0.

Indicators: CCA, OVFL

Traps:	Invalid Character/PACKED	!38
	Invalid Control word/PACKED	!48

183. PMPY <Multiply Packed>

OP CODE is 034050

PMPY expects 3 items (and one zero word) on the stack and (Item 2) := (Item 1) * (Item 0). No indication of underflow is given.

Indicators: CCA, OVFL

181. PSUB <Subtract packed>

193. PTST <Test packed>

Traps:	Overflow/PACKED.	!08
	Invalid Character/PACKED	!38
	Invalid Control word/PACKED	!48

184. PDIV <Divide Packed>

OP CODE is 034054

PDIV expects 4 items (and one zero word) on the stack, where (Item 3) := (Item 1) / (Item 0). Item 2 <- the remainder. If Item 2's Len parameter is zero, then the remainder is not stored, yet the instruction produces a quotient.

Indicators: CCA, OVFL

Traps:	Overflow/PACKED	!08
	Invalid Argument/PACKED	!28
	Invalid Character/PACKED	!38
	Invalid Control word/PACKED	!48

Conversion Decimal Instructions

185. CVAP <Convert ZONED to Packed>

OP CODE is 034024

CVAP expects 2 items on the stack. Item 0 is an ASCII Numeric String and is converted to a Packed string and the result is stored at Item 1. The ASCII source must be leading blanks or decimal digits. The rightmost digit may be signed. Leading blanks will be converted to decimal zeros; embedded blanks are illegal. An all blank field converts to an unsigned (absolute) zero target field. An unsigned ASCII operand will produce an unsigned decimal result. If the number of target digits is less than the number of source digits, the source will be converted until the target field is filled, producing a left truncated result. If the source digit count is less than the target digit count then the remaining target field will be filled with zeros. Both digit counts must be in the range $0 \leq \text{Len} \leq 28$. If the source digit count is zero, then only the S-dec is performed.

Note: If an illegal character is found in the source, then the trap is delayed if it occurred on a zoned digit (appearing in a position other than the sign) until the result is copied to the destination item. The zoned digits would be converted to the corresponding digit. The trap would be taken at the completion of the instruction.

Note: All source digits are always checked.

Indicators: CCA, OVFL

Traps:	Invalid Character/PACKED	!38
	Invalid Control word/PACKED	!48

186. CVPA <Convert Packed to ZONED> OP CODE is 034020
 Numeric Data

CVPA expects 2 items on the stack. Item 0 is a decimal data string and is converted to an ASCII Numeric Data string and the result is stored at Item 1.

An unsigned result is considered positive when setting the condition code, resulting in CCG or CCE. Leading zeros are converted to zeros. If either length is zero, then nothing but the S-dec is performed. The least significant digit is always modified by the sign byte of the decimal source to appear as specified earlier. A set absolute (PABS) instruction is provided to get a digit in the sign position.

Note: The source and destination fields cannot overlap at all in memory since an intermediate copy is not maintained!

Indicators: CCA, OVFL

Traps:	Invalid Character/ZONED	!37
	Invalid Control word/ZONED	!47

187. CVBP <Convert Binary to Packed> OP CODE is 034034

CVBP expects two items on the stack; Item 0's Label must evaluate to a word address (i.e. offset is even). Item 0's Len parameter in word 2 is the length in words of the binary number to be converted. This length must be less than or equal to 6. If this length parameter specifies a length greater than 6, then User Traps disabled will cause OVFL to be set, User Traps enabled will cause Trap !48 to occur. The 2's complement source at Item 0 is converted to decimal and stored at Item 1. The target is filled with leading zeros if necessary. If the source overflows the target, the left truncated result is stored and a decimal overflow occurs. The length of Item 1 must be less than or equal to 28. If the length of Item 0 is zero, then nothing is performed.

Indicators: CCA, OVFL

Traps: Overflow/PACKED !08
Invalid Control word/PACKED !48

188. CVPB <Convert Packed to Binary> OP CODE is 034030

CVPB expects two items on the stack; the label in Item 1 must evaluate to a word address. In Item 1, the specification Len in word 2 is length in words of the binary number. If this length is greater than 6, or if the length of Item 0 is greater than 28, and user traps are enabled, a TRAP !49 is performed and OVFL is not set. If the length of Item 0 is zero, then nothing is performed.

Indicators: CCA, OVFL

Traps: Overflow/BINARY !09
Invalid Character/BINARY !39
Invalid Control word/BINARY !49

189. CVPE <Convert Packed to Extended Floating Point> OP CODE is 034040

CVPE expects 1 ITEM at S-3 and 4 words at S-7 thru S-4 for the Extended floating point result to appear. The contents of the ITEM at S-3 is converted to 4 word floating point and stored at S-7 thru S-4. The Item is always PCP'd and the Extended result is left. The stack should appear as follows.

```

-----
S-7 :                                     :
S-6 :   Space for the Extended          :
S-5 :   Floating Point Result           :
S-4 :                                     :
-----
S-3 :   Source Label for Zoned           :
S-2 :   Decimal Field                   :
-----
S-1 :   Parameter word                   :
-----
S   :   Z E R O                         :
-----

```

Indicators: CCA, OVFL

Traps: Invalid Character/EXTENDED !33
Invalid Control word/EXTENDED !43

190. CVEP <Convert Extended Floating
Point to Decimal>

OP CODE is 034044

CVEP expects Item 1 (at S-7 thru S-5) as a destination pointer and at S-4 to S-1 is a 4 word floating point number which is converted to decimal and stored at Item 1, then the S-dec is performed.

```

-----
S-7 : Destination Label points :
S-6 :       to Decimal Field   :
-----
S-5 :       Parameter word     :
-----
S-4 :       Extended Floating Point :
S-3 :       Source to be converted :
S-2 :       and stored in the      :
S-1 :       Decimal Field         :
-----
S   :       Z E R O              :
-----

```

Indicators: CCA, OVFL

Traps: Overflow/PACKED !08
Invalid Control word/PACKED !48

Special Decimal Instructions

191. MVP <Move Packed Field>

OP CODE is 034014

MVP expects two items on the stack, where Item 0 is a source and Item 1 is a destination. Item 0 is moved to Item 1 and CCA is set on the value. If the destination length is longer than the source, then leading zeros are filled in. If the destination length is less than the source, then truncation occurs at the left end. This instruction simulates the following sequence of Macrocode.

```

DZRO,ZERO;
PALD;

```

Since 3 zeros are PUSHed, the following must be considered:

S must be less than Z-3 else stack overflow will occur.

Shifts can be done with MVP as follows:

```
Shift RIGHT by X digits
      then destination DP = 0
           and source DP = X
Shift LEFT by X digits
      then destination DP = X
           and source DP = 0
```

The shifted results can have half rounding applied since an RADD instruction is preformed.

Indicators: CCA, Overflow

```
Traps:  Overflow/PACKED           !08
        Invalid Character/PACKED  !38
        Invalid Control Word/PACKED !48
```

192. PABS <Set Packed Field to Absolute> OP CODE is 034070

PABS expects one item on the stack and will set CCA on the value of the decimal string pointed to by this item. In addition, it will change the sign byte to absolute (a value of % (2)1111).

Indicators: CCA

```
Traps:  Invalid Character/PACKED  !38
        Invalid Control Word/PACKED !48
```

193. PTST <Test Packed> OP CODE is 034060

PTST expects a single item on the stack and will set the condition code to CCA on its value.

Indicators: CCA

```
Traps:  Invalid Character/PACKED  !38
        Invalid Control Word/PACKED !48
```

194. PNEG <Negate Packed> OP CODE is 034064

PNEG expects a single item on the stack and will negate the decimal string. If the result is zero, only positive zero will be generated.

Indicators: CCA

```
Traps:  Invalid Character/PACKED  !38
        Invalid Control Word/PACKED !48
```

Decimal Edit Instruction

195. EDIT <Edit Decimal>

OP CODE is 034074

The stack contains three 2 word byte labels and a single length parameter. The length of the target string is implied by the edit mask. EDIT does not generate any traps. There is no S-dec. The stack appears as follows:

S-6	:	2 word Byte	:	Target Label
S-5	:	Label	:	
:-----:				
S-4	:	2 word Byte	:	Edit Mask Label
S-3	:	Label	:	
:-----:				
S-2	:	2 word Byte	:	Packed or Zoned Source Label
S-1	:	Label	:	
:-----:				
S	:	Length	:	Packed or Zoned Source Length
:-----:				

Indicators: CCA if numeric editing

Traps: (to be inserted.)

Summary of Decimal Instructions

Below is a complete list of the decimal instruction package. Those instructions appended with a # sign are implemented in firmware.

Inst	OP CODE	Inst	OP CODE
PADD#	034000	CVPB	034030
PSUB#	034004	CVPE	034040
PCMP#	034010	CVEP	034044
EMPY	034050	MVP#	034014
PDIV	034054	PABS#	034070
CVAP#	034024	PTST#	034060
CVPA#	034020	PNEG#	034064
CVBP	034034	EDIT	034074

Appendix A Alphabetical Listing of Instructions

88.	ADAX	<Add A to X>
29.	ADD	<Add>
115.	ADDI	=N <Add immediate>
12.	ADDM	E(*,D,X,PDQS) <Add memory>
157.	ADDS	N <Add to S>
89.	ADXA	<Add x to A>
122.	ADXI	=N <Add immediate to X>
66.	AND	<And, logical>
119.	ANDI	=N <Logical AND immediate>
92.	ASL	CNT<Arithmetic shift left>
93.	ASR	CNT<Arithmetic shift right>
16.	BCC	L(*,L,F) <Branch on Condition Code>
17.	BCY	L(*,L,F) <Branch on carry>
18.	BNCY	L(*,L,F) <Branch on no carry>
20.	BNOV	L(*,L,F) <Branch on no overflow>
19.	BOV	L(*,L,F) <Branch on overflow>
15.	BR	E(*,D,X,P or indirect DQS) <Branch>
22.	BRE	L(*,L,F) <Branch on TOS even>
21.	BRO	L(*,L,F) <Branch on TOS odd>
67.	BTST	<Test byte in TOS>
83.	CAB	<Rotate ABC>
176.	CBLW	<Convert byte label to word>
27.	CCI	<Convert Condition Code to Integer>
177.	CLAA	<Convert Label to Absolute Address>
28.	CMP	<Compare>
142.	CMPB	<Compare bytes>
113.	CMPI	=N <Compare immediate>
11.	CMPM	E(*,D,X,DQS) <Compare memory>
114.	CMPN	=N <Compare negative immediate>
23.	CPRB	L(*,L,F) <Compare range and branch>
96.	CSL	CNT<Circular shift left>
97.	CSR	CNT<Circular shift right>
185.	CVAP	<Convert zoned to packed>
187.	CVBP	<Convert binary to packed>
190.	CVEP	<Convert extended Flt. pt to decimal>
26.	CVID	<Convert Integer to Double Integer>
186.	CVPA	<Convert packed to zoned>
188.	CVPB	<Convert packed to binary>
189.	CVPE	<Convert packed to extended Flt.Pt.>
175.	CWLB	<Convert Word Label to Byte Label>
41.	DADD	<Double add>
99.	DASL	CNT<Double arithmetic shift left>
100.	DASR	CNT<Double arithmetic shift right>
134.	DEBUG	<Debug Interrupt Call>
40.	DCMP	<Double compare>
103.	DCSL	CNT<Double circular shift left>
104.	DCSR	CNT<Double circular shift right>
74.	DDEL	<Double delete>
77.	DDLE	<Double duplicate>

44.	DDIV	<Double Divide>
87.	DECX	<Decrement X>
73.	DEL	<Delete A>
75.	DELB	<Delete B>
131.	DISP	<Dispatch>
32.	DIV	<Divide>
118.	DIVI	=N<Divide immediate>
101.	DLSL	CNT<Double logical shift left>
102.	DLSR	CNT<Double logical shift right>
43.	DMPY	<Double Multiply>
45.	DNEG	<Double negate>
110.	DPF	<Deposit field>
42.	DSUB	<Double subtract>
89.	DTST	<Test double word on TOS>
76.	DUP	<Duplicate A>
80.	DXCH	<Double exchange>
72.	DZRO	<Double push zero>
56.	EADD	<Extended Floating Add>
55.	ECMP	<Extended Floating Compare>
195.	EDIT	<Edit decimal>
59.	EDIV	<Extended Floating Divide>
61.	EFLT	<Extended Float>
62.	EFXR	<Extended Fix and Round>
63.	EFXT	<Extended Fix and Truncate>
58.	EMPY	<Extended Floating Multiply>
60.	ENEG	<Extended Floating Negate>
57.	ESUB	<Extended Floating Subtract>
109.	EXF	<Extract Field>
125.	EXIT	N<Procedure exit>
47.	FADD	<Floating add>
46.	FCMP	<Floating compare>
50.	FDIV	<Floating divide>
54.	FIXR	<Fix and round>
53.	FIXT	<Fix and truncate>
52.	FLT	<Float>
49.	FMPY	<Floating multiply>
51.	FNEG	<Floating negate>
48.	FSUB	<Floating subtract>
178.	GEL	<Generate extended label>
127.	HALT	K <Halt>
144.	HIOP	<Halt I/O Process>
14.	INCM	E(*,D,X,DQS) <Increment memory>
86.	INCX	<Increment index>
154.	INIT	<Initialize I/O Channels>
149.	IOFF	<Disable External Interrupts>
148.	ION	<Enable External Interrupts>
135.	IPL	<Initial Program Load>
152.	ITOF	<Interrupts & timer off>
153.	ITON	<Interrupts & timer on>
130.	IXIT	<Interrupt Exit>
35.	LADD	<Logical add>
34.	LCMP	<Logical compare>
107.	LBIT	<Load Bit>
2.	LDB	E(*,D,X,PLQS) <Load byte>

3.	LDD	E(*,D,X,PCQS) <Load double>
166.	LDEA	<Load double word from ext. address>
111.	LCI	=N <Load immediate>
38.	LDIV	<Logical divide>
112.	LDNI	=N <Load negative immediate>
6.	LDX	E(*,D,X,PCQS) <Load index>
84.	LDXA	<Load X onto stack>
120.	LDXI	=N <Load X immediate>
121.	LDXN	=N <Load X negative immediate>
174.	LLBL	N <Load Label>
173.	LLSH	<Linked list search>
37.	LMPY	<Logical multiply>
1.	LOAD	E(*,D,X,PCQS) <Load>
5.	LRA	E(*,D,X,PCQS) <Load relative address>
165.	LSEA	<Load single word from ext. address>
94.	LSL	CNT <Logical shift left>
95.	LSR	CNT <Logical shift right>
36.	LSUB	<Logical subtract>
171.	MCS	<Memory Command and Status Return>
137.	MOVE	<Move words>
31.	MPY	<Multiply>
117.	MPYI	=N <Multiply immediate>
25.	MTBA	E(*,D,P) <Modify, Test, Branch A>
138.	MVB	<Move bytes>
139.	MVBW	<Move bytes while>
33.	NEG	<Negate>
91.	NOP	<No operation>
39.	NOT	<One's complement>
64.	OR	<Or, logical>
192.	PABS	<Set packed field to absolute>
180.	PADD	<Add packed>
128.	PAUS	K <Pause>
124.	PCAL	N <Procedure call>
182.	PCMP	<Compare packed>
184.	PDIV	<Divide packed>
164.	PLDA	<Privileged load from absolute address>
183.	PMPY	<Multiply packed>
194.	PNEG	<Negate packed>
132.	PSDB	<Pseudo Interrupt Disable>
133.	PSEB	<Pseudo Interrupt Enable>
157.	PSHR	N <Push registers>
167.	PSTA	<Privileged store into absolute address>
78.	QDUP	<Quad Duplicate>
6.	QLD	E(*,D,X,PCQS) <Quadruple Load>
105.	QLSL	<Quad logical shift left>
106.	QLSR	<Quad logical shift right>
10.	QSTR	E(*,D,X,DQS) <Quadruple Store>
70.	QTST	<Quad Test>
81.	QXCH	<Quad Exchange>
162.	RCLK	<Read Clock>
146.	RIOC	<Read I/O Channel>
156.	RMSK	<Read Mask>
170.	RSW	<Read Switch register>
108.	SBIT	<Store Bit>

123.	SBXI	=N <Subtract immediate from X>
98.	SCAN	<Scan bits>
163.	SCCR	<Set Clock Comparison Register>
141.	SCU	<Scan until>
140.	SCW	<Scan while>
169.	SDEA	<Store double word into ext. address>
136.	SDMP	<System Dump>
172.	SEML	<Semaphore Load>
158.	SETR	N <Set registers>
143.	SIOP	<Start I/O Program>
155.	SMSK	<Set Mask>
168.	SSEA	<Store single word into ext. address>
85.	STAX	<Store A into X>
8.	STB	E(*,D,X,DQS) <Store double>
9.	STD	E(*,D,X,DQS) <Store double>
7.	STOR	E(*,D,X,DQS) <Store>
30.	SUB	<Subtract>
116.	SUBI	=N <Subtract immediate>
13.	SUBM	E(*,D,X,DQS) <Subtract memory>
158.	SUBS	N <Subtract from S>
126.	SXIT	N <Subroutine exit>
24.	TBA	E(*,D,P) <Test, branch, A>
68.	TEST	<Test TOS>
151.	TOFF	<Timer Off>
152.	TON	<Timer On>
179.	VLAB	<Validate Label>
147.	WIOC	<Write I/O Channel>
82.	XAX	<Exchange A and X>
79.	XCH	<Exchange A and B>
161.	XCHG	<Exchange Global Data Area>
129.	XEQ	K <Execute>
90.	XLSL	<X-register Logical Shift Left>
63.	XOR	<Exclusive or, logical>
71.	ZERO	<Push zero>

