HEWLETT PACKARD

RTE-A

System Design Manual

Software Services and Technology Division
11000 Wolfe Road
Cupertino, CA 95014-9804

Manual Part No. 92077-90013
E0495

Printed in U.S.A. April 1995
Seventh Edition

NOTICE

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

HEWLETT-PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THE 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.

RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARs 252.227.7013.

Printing History

The Printing History below identifies the edition of this manual and any updates that are included. Periodically, update packages are distributed which contain replacement pages to be merged into the manual, including an updated copy of this printing history page. Also, the update may contain write-in instructions.

Each reprinting of this manual will incorporate all past updates; however, no new information will be added. Thus, the reprinted copy will be identical in content to prior printings of the same edition with its user-inserted update information. New editions of this manual will contain new information, as well as all updates.

To determine what manual edition and update is compatible with your current software revision code, refer to the Manual Numbering File. (The Manual Numbering File is included with your software. It consists of an "M" followed by a five digit product number.)

Second Edition	Jun 1983	File System & VC+ Enhancement
Third Edition	Jan 1985	System Module Partitioning & User-Definable Directory Search Path Enhancements
Update 1	Jan 1986	
Reprint	Jan 1986	Update 1 Incorporated
Fourth Edition	Aug 1987	Revision 5000 (Software Update 5.1)
Update 1	Jan 1989	Software Revision 5.1 (5010)
Update 2	July 1990	Software Revision 5.2 (5020)
Fifth Edition	Dec 1992	Software Revision 6.0 (6000)
Sixth Edition	Nov 1993	Software Revision 6.1 (6100)
Seventh Edition	Apr 1995	Software Revision 6.2 (6200)

Preface

This manual describes the RTE-A Operating System. It is designed to provide information that will help you to configure a new operating system and to troubleshoot system difficulties.

Who should read this manual?

Anyone requiring information on system data structures, system memory utilization, functions performed by system modules, and other system generation and system installation considerations.

Table of Contents

Chapter 1
System Overview

Chapter 2
Memory Management

Chapter 3
Programs and Partitions

Chapter 4
System Boot-Up

Chapter 5
Operating System Modules

Chapter 6
System Symbols and List Structures

Chapter 7
I/O Drivers

Chapter 8
System Common/Shared Subroutines

Chapter 9
System Base Page and Link Words

Chapter 10
File System

Chapter 11
System Tables

Chapter 12
FMP Tables

Appendix A
Snapshot File Format

List of Illustrations

Tables

1

System Overview

Introduction

This chapter gives an overview of the RTE-A Operating System. It introduces the various structures used by the A-Series computer software. The computer software is Real-Time Executive (RTE) software that can be divided into two categories: operating system and user tasks.

The operating system is the software that manages the hardware, including the various input/output (I/O) devices such as the user terminals, printers, and disks. Software that falls into the second category is concerned with a particular application or job to be performed.

A real-time operating system, such as RTE-A, is designed for a wide variety of tasks. The system permits tasks to be prioritized and allows tasks whose purpose is computation or display to co-exist in the system with other tasks whose purpose is to control and react to measurements from instruments. In addition, the number of simultaneous tasks permitted is not limited to the number of user terminals. Thus, the real-time system enables the computer to respond to real-time stimuli such as those in the measurement and control environment.

The RTE-A Operating System is more than just a computational resource; it is a task manager as well. Thus the term executive is included in the name, Real-Time Executive operating system.

The fundamental unit of tasks (that is, the lowest level of user software managed by the system) is the program. Therefore, task management reduces to program management.

The basic functions of the operating system are:

Program Management
Input/Output (I/O) Management
Memory Management

These functions are accomplished with extensive use of system tables. Tables in the system maintain information about programs, I/O devices, system lists, and swap areas on a disk. The lists link together those programs or I/O devices that are in a similar state.

Two important tables at this level are the program table (known in the system as the ID segment) and the Device Table (DVT). An ID segment exists for each program that is ready to run and a DVT exists for every device. Much of the job of the RTE-A Operating System is to examine and change entries in the tables and link the tables into different lists as needed.

Program Management

Program scheduling, execution, and the various program states are shown in Figure 1-1. A program can be in one of the following states:

Dormant
Scheduled
Executing
Suspended

A dormant program is inactive in the system. It is moved out of the dormant state by being scheduled, placed in a list waiting to be run. The program to be run next is dispatched and executed. The cause of change of state from the dormant state may be:

A user request
A system request
A program request
An external event (for example, key pressed or relay closed)
Time to run (based on a system clock)

All programs are given a priority level, which is a number from 1 to 32767, with number 1 being the highest priority. The schedule list is ordered by priority such that the program with the highest priority (lowest number) is always at the top of the list.

The system has a clock that "ticks" at intervals of 10 milliseconds. At least every tick (it may be more often), the system checks the priority of the executing program (if any) against the priority of the program at the top of the schedule list. The highest priority program is put into the executing state and is dispatched.

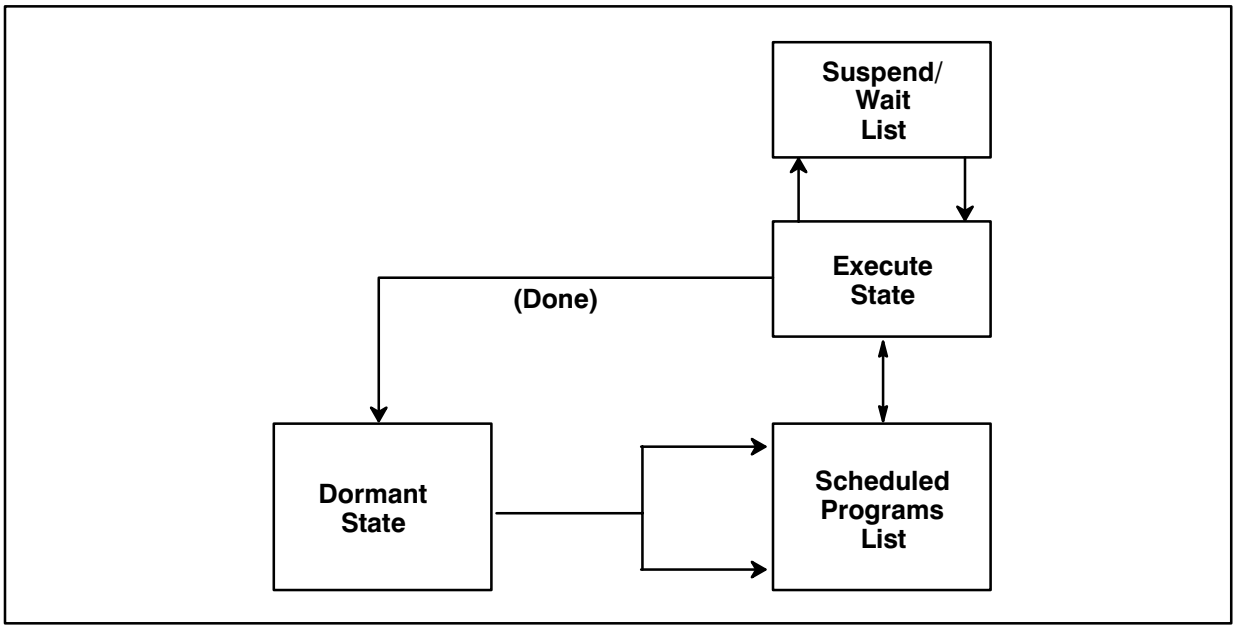


Figure 1-1. Program Scheduling and Execution

In the process of executing, the program may make a system request for a resource not immediately available. For example, it may make a write request to the line printer while the printer is busy with another request. Whenever a desired resource is not available, the program is suspended until the resource becomes available. So, the program is taken out of the executing state and placed into the suspend/wait list (which is broken down into several lists according to the reason for suspension/wait).

After processing any request, successful or not, the system always returns to the schedule list for the next program to be placed in the executing state. Thus, lower priority programs are permitted to execute during the time that higher priority programs are suspended. However, when the reason for suspension of the higher priority programs no longer exists, the lower priority program is suspended and the higher priority program is allowed to continue execution.

Since programs may schedule other programs, all it takes to start any process is a single program. The RTE-A Operating System allows the user to specify any program to be automatically scheduled when the system is booted up. The system may or may not require any attention from a user other than to turn on the power switch. The program scheduled at boot up may perform the designated function and there is no need for any further user interface.

At the other end of the scale is the general purpose system, which is used to develop programs and to implement systems such as the one mentioned above. A system can be used for both real-time applications (that is, time dependent) and non time-critical functions (such as compiling programs) by managing program priorities.

Real-Time and Background Programs

In RTE-A, real-time programs are generally those that must respond quickly to an instrument or any external event. Background programs are those that are not time-critical. The difference, however, is more than response to real-time stimuli.

To understand the difference between real-time and background programs, it is necessary to first define program swapping. Program swapping is a capability that allows programs to execute in the same portion of memory (the partition). Swapping is a system option. If available, then a program in a partition may be suspended and moved to the disk and another program allowed to use the partition. Later, the first program may be moved back to memory and allowed to complete. From the time a program begins until the time it finishes, it is not guaranteed exclusive use of the memory partition it runs in. Refer to the Program Swapping section in this chapter for more information.

In RTE-A, the distinction between a real-time program and a background program is accomplished through the priority that is assigned to the program. At generation time a priority swap boundary is defined. As a result, priority swapped programs (real-time programs) tend to stay in memory partitions. These programs can be removed only for higher priority real-time programs to provide the quick response needed for real-time activities.

Programs with priorities below the priority swap boundary are background programs that are swapped normally. That is, background programs contend on a priority basis for memory partitions.

Large Programs

Large programs can be handled with program overlays. The use of program overlays is one means of making efficient use of memory in systems without the Code and Data Separation (CDS) program feature provided in the optional HP 92078A Virtual Code+ (VC+) Package. Compilers and other large programs are divided into overlays. Programs with overlays in the system may run as either background or real-time programs. In systems with VC+, the CDS feature can be used to divide the large program into segments automatically.

The large non-CDS program is divided into a program main and two or more program overlays. The main and each overlay initially reside in separate areas on the disk. The main is loaded from the disk at the start of the program area. The main, in the course of execution, may make a system request to cause an overlay to be loaded into memory in the area behind the main. Program overlays are illustrated in Figure 1-2. When the program code in the current overlay has finished execution, it may return to the main or request that another overlay be loaded. Overlays may vary in size.

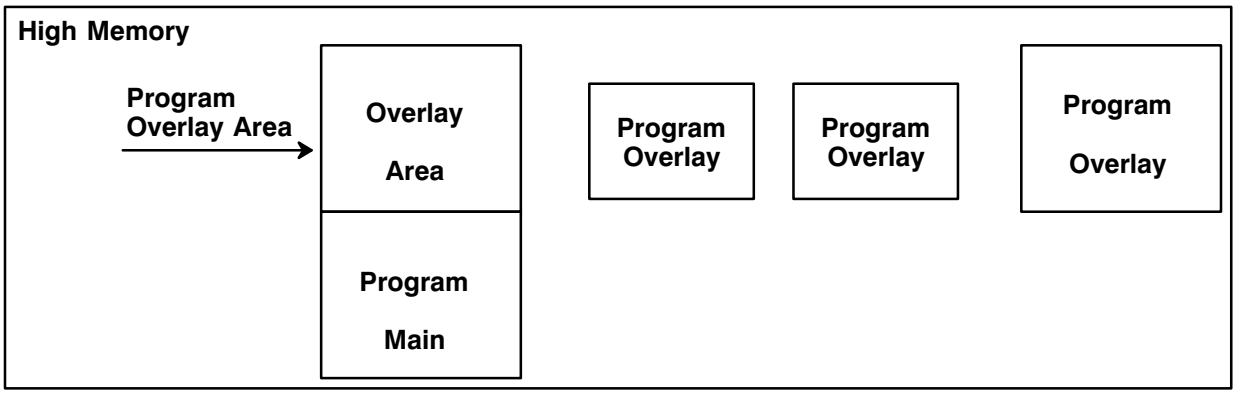


Figure 1-2. Program Overlays

In systems with VC+, the CDS feature can be used to divide the large program into segments automatically. Large CDS programs also can be segmented manually using LINK commands. The large CDS program is divided into code segments, which execute in a code partition, and a separate data partition. Each code segment is loaded into memory in a code block in the code partition as shown in Figure 1-3.

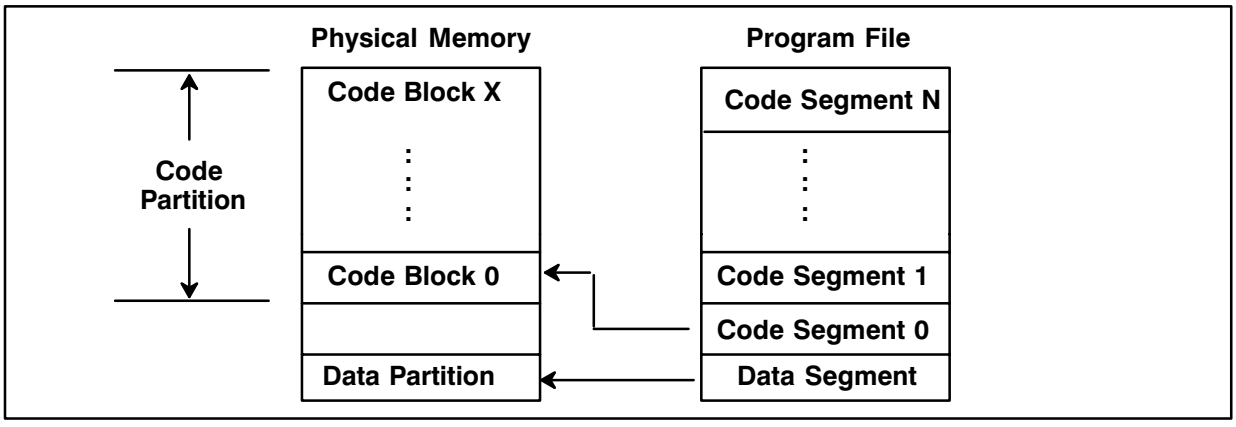


Figure 1-3. CDS Program Segments

Program Swapping

Program swapping is a technique used to allow executing programs to share memory. When one program suspends, or when its allotted time (timeslice) is complete, it can be swapped out, together with the data, to the swap area on disk. Then, another program can be copied in from the swap area if it has been swapped out, or from the disk if it has just been scheduled.

For a CDS program, when it is suspended or its timeslice is complete, the data may be swapped out so that another program can be loaded. The code portion of the program may be overwritten without swapping in order to load another program from the disk.

The algorithms that control the search for memory for a program being dispatched use the reserved memory-descriptor table and the dynamic memory-descriptor list. Refer to the reserved partition and dynamic memory allocation sections in Chapter 3 for details.

To swap a program means to move a program from memory to a temporary (swap file) area of the disk and move another program from the disk into the partition. The swap area is constructed by the BOOTEX program and the size is defaulted to 32 pages times the number of ID segments generated. Programs in the scheduled list appear to be running concurrently, although only one program is executing at any one time.

When the program is swapped out to the disk, it is a snapshot or an image of the program as it existed in memory. When it is swapped back from the disk, memory and CPU registers are restored to the same conditions as existed before the swap, and it is allowed to continue execution.

Program Development

In any operating system, program development requires a basic set of software tools. The software tools required and those available in RTE-A are given below.

1. A text editor, EDIT, to create and modify source files. The source files are written in the language chosen for the program development.

2. An assembler or compiler; for example, FORTRAN or Pascal compiler. (Macro/1000 assembler is also available.) This determines the language in which the program is written.

   The compiler or assembler accepts the source file as input and produces two output files:

   a. A listing of the program, along with any error messages.

   b. The program in relocatable binary code. This code is not executable in the system.

3. A process for producing a load module (that is, a file that is an executable program).

4. A debugger, Symbolic Debug/1000, to troubleshoot the program if any problem occurs.

In RTE-A, all executable programs are placed in program files by the LINK program. LINK accepts a command file that contains, among other instructions, the input relocatable files and what files to search for external references (to link modules together). The program file produced by LINK contains a memory image of the program as it will appear when it is actually loaded into memory. The file is loaded into memory when the program is run by a command available in the Command Interpreter (CI) program or scheduled by an EXEC call.

The first five characters of the file name normally become the name of the program when it is loaded. However, the operator command RP can be used to set up multiple ID segments (program tables) that identify the same program file with different names. That is, the name in the ID segment is the program name, which may be different from the name of the file containing that program.

Memory Management

Computer memory is one of the most valuable resources to be managed by the operating system. Memory partitions, which are overlayable, make efficient use of memory space. Partitions are made up of several contiguous pages of memory. Available memory also limits the maximum program size of non-segmented (or non-overlayable) programs.

The RTE-A Operating System has a combination of fixed (called reserved) and dynamic memory partitions. Only a program that is assigned to a reserved partition runs in that partition. Dynamic partitions are created as needed from an area in memory called the dynamic memory pool. Refer to Chapter 3 for more information on managing user partitions.

Program Partitions

User programs run in memory partitions, each partition occupied by one program at a time. With the swapping option configured in the system, multiple programs may be swapped back and forth between memory partitions and disk files.

System Common

System common refers to a shared data area. It may be shared by two or more programs. There are two types of system common:

- Blank (unnamed)
- Labeled

Blank common is a storage area that is initialized to zeroes by the generator. Any names given to the individual memory locations in blank common are unknown to the system itself.

Labeled common, however, is accessed by reference to the particular label or name assigned to the storage area. There may be several names, representing one or more words of computer memory. The purpose of system common is to contain subroutines and/or data to be shared by two or more programs. Blank common, together with labeled common, makes up the block of memory called system common. Values in labeled common may be initialized only at generation time.

A program common area is also possible and should not be confused with system common. Program common is also a shared data area but it may be shared only by the various parts of a single program; that is, it is not shared by more than one program. Program common may also be divided into a blank and labeled area.

Whenever the size or content of system common is changed by the generator, all programs that access that area should be checked to see if they require modification. All programs using system common should be linked again when system common changes.

System Available Memory (SAM)

System Available Memory (SAM) is a block of memory that the system uses to implement system requests that require memory space. SAM is used in the system for:

1. I/O Buffering

   This is discussed in detail in the I/O Management section.

2. String Passage

   String refers to a number of ASCII characters addressed to a program when it is scheduled. The string generally contains parameters to be decoded by the program.

3. Class I/O

   Class I/O is used to provide optional buffering of input and output requests and to provide program-to-program communication.

The maximum size of SAM is determined at generation or bootup time. When a request is made that requires memory, a contiguous block of memory locations is taken from SAM and allocated for the request. When the request completes, the block is linked back to any adjacent free blocks of SAM and is linked into the free SAM list. Thus the size of SAM is dynamic and, at times, the largest contiguous block may be smaller than the total number of words available.

Programs making a request that requires SAM when none is available become memory suspended. All memory suspended programs are in one list, which is ordered by program priority. When memory is available, the program at the top of the list is permitted to execute if there is enough memory to meet its requirements.

Extended System Available Memory (XSAM)

Extended System Available Memory (XSAM) is very similar to SAM. In fact it is possible to have SAM and XSAM use the same block of memory (see the System Generation and Installation manual for more detail). XSAM is used for memory requests that typically stay around longer than I/O requests. One ID segment extension is allocated from the beginning of XSAM for each ID segment in the system. XSAM is also used by:

1. Signals

   XSAM is used in different manners for signals. Every program that receives signals must have a Signal Control Block (SCB) and every signal that is generated uses a Signal Queue Block (SQB). Also timer signals are queued in XSAM until they expire.

2. UDSPs and LU access tables

   User Defined Search Paths and LU access tables are stored together in one block of memory per session. UDSPs tell the system where to look for commands that have been given. LU access tables tell the system to which LUs a session can have access.

3. Prototype ID segments

   The list of proto ID segments is kept in XSAM. Program ID segments are created quickly by duplicating the appropriate proto ID.

4. SHEMA table entries and SHEMA Association Blocks

   These are data structures related to shared memory areas.

The management of SAM and XSAM is identical. If SAM and XSAM occupy the same block of memory, there is only one free list that describes the free memory of both.

System Tables

The purpose of the tables found in the memory map of the RTE-A system is briefly described below. Details are given in Chapter 11 of this manual.

1. Program Table (ID Segment)

   The program table is commonly referred to as the ID segment. Each program must have an ID segment before it can be scheduled. The ID segment identifies the program by name and shows where it is in memory. If it is a disk resident program, it also identifies its location on the disk. Several words are used for temporary storage by the system. For example, some words are used to identify the state of the program (such as scheduled or suspended) and to link the program in a list with other programs in the same state. There is also an ID segment extension residing in XSAM for each ID segment; these hold less-frequently accessed data.

2. Several tables are related to I/O management. These are discussed below.

   a. The Logical Unit Table (LUT) associates the LU to which a program makes a request to the actual device. In fact, each entry in the LU table points to a particular DVT (device table).

   b. A Device Table (DVT) exists for each I/O device. The DVT identifies the device and its state. The user request is inserted in the DVT prior to execution of the device driver. Status information is passed back through the DVT when the request is completed.

   c. An Interface Table (IFT) exists for each interface card in the computer. The IFT is used to keep track of the current I/O request in progress. Several DVTs may be linked to a single IFT, just as several devices may be cabled to one interface card.

   d. The interrupt table is used by the system to identify which driver should handle the interrupt. It specifies the IFT to be called to service the interrupt.

3. Class Numbers

   Class numbers are used to implement buffering of input requests and to permit program-to-program communication.

   A table of class numbers is kept by the system and assigned to programs as they are requested. Class numbers permit a variation of the normal I/O operation, one that is associated with the program that makes the actual request. The concept of class I/O is to

associate the request with a class number instead of the program. The class number may be shared by several programs.

The actual class request is made in two steps: initiation of the request and completion of the request. The same program or different programs can perform the two separate operations.

The I/O operations done on behalf of a specific class number are available only to those programs that recognize the number. Thus different classes are isolated from one another.

A version of class I/O is called mailbox I/O. Mailbox I/O permits direct program-to-program communication through the system buffer area.

4. Resource Numbers

   A resource number table is kept by the system and resource numbers are assigned to programs as they are requested. Resource numbers are useful only if they are shared by cooperating programs. A resource number is typically claimed by a program prior to an operation that requires coordination between programs. For example, if two programs are updating the same file, each program attempts to claim the resource number prior to the update. If the number is not already used, the program proceeds with the update. If another program already claimed the number, then the program is forced to wait until the number is released by the program using it.

5. Swap Descriptor

   The swap descriptor table is a linked list of free swap areas of the swap file. It is used by the dispatcher to locate an area of sufficient size to swap a program from memory to disk.

User Interaction

Since a system may be configured without any user terminal, it is up to the person who configures the system to permit user interaction.

There are two uses for a terminal on the system:

1. One terminal may function as the virtual control panel (VCP). A switch on the interface card for that terminal identifies it as the VCP.

2. The terminal may function as the communications device between a person and the operating system or between a person and a user program.

The distinction is whether communication is with the computer itself or with the software running in the computer. In VCP mode, the user effectively halts the computer. A user can display and change the contents of registers and memory locations, and control execution from the VCP mode. This is similar to a diagnostic mode although its use does not imply anything wrong with the computer.

If a terminal has its interface card selected to operate as the VCP, then the VCP mode is entered by pressing the BREAK key. Depending upon how the internal switches are set on the computer, the power up sequence may also put the terminal in the VCP mode. The user then can start the system via commands entered at the terminal. The terminal may then be used by the operating system and user programs.

Although the terminal (in the VCP mode) always responds to user input on the keyboard, other terminals in the system do not respond unless they have been enabled. A terminal can be enabled either by an user command or a request from a user program. The enable command (or request) is passed to the I/O driver for the terminal and is accompanied by an user interface program name (according to the request).

If a terminal has been enabled, the user presses any key to gain the attention of the driver. The driver first attempts to schedule the user interface program (if any). If that fails, a second interface program is scheduled. If that fails, then the user interacts directly with the system. Otherwise, the user interacts with the program that was successfully scheduled for the terminal.

The choice of program to interact with the user can cover a wide range. It may be a program to monitor a test sequence. It may be simply a program providing additional commands to the system. A single program may be duplicated and given a unique name for each terminal, or a different program enabled for each terminal. Thus, each terminal may have the same or different capabilities.

The normal user interface program is the Command Interpreter program (CI). It can be scheduled automatically to handle all user requests. Capabilities provided by this program are described in the *RTE-A User's Manual*.

Direct interaction with the system provides a fairly limited set of capabilities. This is the base level of interaction necessitated by some system problem. A duplicate copy of the Command Interpreter is run to allow entry of a single request, usually running of a program to inquire the system status. This program terminates after every command. Hence, in a multiuser environment provided by VC+, it may be used from several terminals with no competition among different users.

Another user interface program called FMGR is provided. This program is mainly used to handle FMGR files. Unless indicated otherwise, all operator commands and files referenced in this manual are associated with the Command Interpreter.

System Boot Up

The term boot up means the process of getting the system to run when power is turned on. Boot up may be automatic, which is convenient if the system does not have a system console. In this case, the internal switches in the computer determine the device from which the system is loaded into memory. The same set of switches also selects whether the system automatically begins operation or waits for the user.

The computer has two modes in which it communicates with the system console:

1. The virtual control panel mode, in which the console is acting as the control panel on the computer itself. In early computers, direct control of the computer was implemented by a set of switches and indicator lamps on the front panel. The only control on the A-Series computer is the front panel power on/off switch.

2. The run mode, in which the user software has control. In this case, user software means the RTE-A Operating System and all user programs managed by the system.

In the virtual control panel mode, the user has the choice of loading the operating system into memory from one of several devices that may be available:

Flexible disk
Hard disk
Another computer through a network link
Magnetic tape
PROM I/O interface card

Several different systems (that is, different user-defined configurations of RTE-A) may reside on the same disk. The user can select which system is to be used. This permits switching the computer quickly from one application to another.

Space can be reserved for BOOTEX on the first track of every disk logical unit when it is initialized. BOOTEX is the boot extension that the Virtual Control Panel (VCP) program brings into memory when it loads or boots from disk.

When brought into memory, BOOTEX searches the disk LU for the file name passed to it from VCP. The default name is /SYSTEM/BOOT.CMD (or SYSTEM for a FMGR bootable LU). However, this file can be either a system file or a command file. If it is a system file, it is loaded into memory and control is transferred to it. If the file is not a system file, BOOTEX assumes it is a command file containing commands that specify the names of the system and snapshot files on the disk, how to divide up the memory partitions, which disk volumes to mount, and other BOOTEX commands. If the specified file is not found (or never specified), BOOTEX runs interactively and prompts the user at the VCP terminal with the following prompt:

```
BOOTEX:
```

When the END command is encountered either in the command file or input from the user, BOOTEX loads in and transfers control to the system, which comes up running.

I/O Management

All input/output (I/O) takes place through the software I/O drivers that are included in the system. The program must make an I/O request to the system in order to access the devices controlled by the drivers. The I/O drivers actually operate the interface cards physically connected by cables to the devices.

Three types of I/O requests are possible:

Read
Write
Control

A control request usually specifies a function to be performed by the device. For example, the function code 4 (octal) addressed to the cartridge tape causes the tape to be rewound.

An I/O request is addressed to a logical unit (LU), not an actual device. The LU is a decimal number from 1 to 255. For example, a FORTRAN statement such as:

```
WRITE (6,*) 'HELLO'
```

is addressed to LU 6.

When the system is generated, each device is assigned an LU. This provides device independence. (In systems with VC+, the LU redirection feature can be used to change the association between an output device and LU.) For example, the program may use LU 6 to print a report. But the user can redirect LU 6 to be a cassette tape unit and store the output on cassette tape.

LU 1 is special. By convention, it refers to a user terminal; even if there are several different users, each can refer to the respective terminal as LU 1. This applies only to LU 1; other LUs are common to all users.

I/O Drivers

I/O drivers are software modules that provide a uniform programmatic interface to the hardware. The use of drivers allows writing of programs that read and write to many different types of peripheral devices without any concern for device protocols, record blocking, and other interfacing requirements. The I/O drivers are the parts of the system that directly control the devices connected to the system.

The LU table is used by the system to determine the actual I/O driver to be called. Entries in the table refer to another set of tables, the device tables (DVTs). There is one DVT per device on the system. Information about the user request is transferred to the DVT by the system prior to calling the driver. When the driver finishes the request, the status of the device is passed back through the DVT.

Although it appears as one to the request, most drivers are in two parts; the device driver and the interface driver. The interface driver handles the actual I/O transfer. The device driver is associated with a particular device and formats the request for that specific device. The interface driver can handle any device that can be cabled to the actual interface card. Thus the device driver is similar to an interpreter.

For every interface card, there is an interface table (IFT), just as there is a DVT for every device. Multiple devices may be linked to the same interface by linking their DVTs to the IFT. This is similar to linking the actual devices through a single cable, such as the case with the HP-IB.

The major requirements of the driver are:

1. To start the device

2. To respond to the completion of a data transfer

The data is actually transferred in a block of words (or bytes) by the hardware and this transfer is independent of computer processing. That is, a program may be running at the same time that data is going out of computer memory or into computer memory.

Since the computer proceeds independently of the data transfer, an interrupt is used to notify the system when the transfer is complete. An interrupt causes the computer to stop whatever it is doing and go to a predefined location in memory associated with the interface card that caused the interrupt. The predefined location is called a trap cell and is a memory address on the base page.

Trap cells for standard I/O drivers contain an instruction to go to a central interrupt control routine in the operating system. The system uses an interrupt table to determine how to process the interrupt. Generally for I/O interrupts, the interrupt table identifies the driver associated with that trap cell. The operating system then calls the driver to finish processing. When all processing is completed, the system returns to dispatch the program that is at the top of the schedule list.

The program dispatched may or may not be the same program that was interrupted. During the waiting period, another program may have been running (the interrupted program, perhaps). When the request completes and a suspended program is rescheduled, it may be of a higher or a lower priority than the interrupted program; the higher priority program is always dispatched (put into the executing state).

I/O Buffering

When an I/O request is made to the system, a buffer in the user program is specified. A buffer is a data array. It may contain the data that the system sends out to a device or it can be the storage area in which data is placed on a read request. This user buffer is the only buffer about which the driver needs to know.

The request itself is defined to be buffered if the data goes through an intermediate area, the system buffer. For a buffered output request, the contents of the user buffer are moved to the system buffer and the driver takes the data from the system buffer. For a buffered input, the driver moves the data into the system buffer; when input is complete, the contents of the system buffer are moved to the user buffer.

One reason for buffering is to make it possible for the program to continue operation before fully completing the I/O request; that is, I/O without wait. This is because the user buffer is released immediately on buffered output and is not required on input until the read request is completed. Therefore, there may not be any need to suspend the program (that is, take it out of the schedule list).

The use of a system buffer, which contains the actual input/output data being transferred, also makes it possible to swap the program making the I/O request. When a program is swapped out to the disk and another program brought in to use the same partition of memory, no area of that user program partition can be used by the swapped out program. Buffering in system memory is the mechanism that permits a program to be swapped while the I/O request is being serviced.

The system buffer is an area of memory that is allocated to the system at the time it is configured. This area is called system available memory (SAM). When an individual request is made that requires system memory, a portion of SAM is temporarily allocated for that purpose. When the request is completed, the memory is returned to SAM.

I/O Without Wait

It is not always desirable to do I/O without wait. For example, if reading data from a device, the next action of the program may depend on the data. In fact, this is usually the case with input. Even if the data itself is not examined as it is read, the program must acknowledge when the end of data is reached.

The opposite situation is true on output. If a program is printing a report, there is no need to wait for a line to be actually printed before the program is allowed to continue. In fact, the program could format the next line of output during the time the previous line is being printed. Then the next output request can be made as soon as the previous request is finished. This makes the program run faster and also keeps the printer as busy as possible, both desirable goals.

For these reasons, the system makes the default mode to be I/O without wait for output and I/O with wait on input. However, the programmer may choose to reverse the default mode by changing the program to make a variation of the standard I/O request or by making a class I/O request.

The capability to do I/O without wait would be severely limited without a method of permitting multiple requests to be pending on the I/O device. That is, what happens when a program makes a request to a device that is already busy.

When a request is made to a device, the information about the request is inserted into a control block, which is a table in memory. If the request is buffered, then the control block is allocated from SAM. If the request is unbuffered (and not class), the program must wait for I/O completion and so the necessary information is stored in the ID segment (program table) and the ID segment contains the control block.

The ID segment (with the control block) or the SAM control block is then added to a list and attached to the DVT for the appropriate device. Many I/O requests may be in the list already; a new one is simply added on. As one request is completed, the system simply passes on the next request to the driver associated with that DVT.

Direct Memory Access

Direct Memory Access (DMA) allows the CPU to perform other tasks concurrently with one or more I/O transactions. Each I/O transaction needs to have a map set associated with it. The map set controls which 64K byte block of memory the data is placed in.

The DMA transfer uses map sets as follows:

- Map set 0 is used if the I/O is from the system partition.
- Map set 4 if the I/O is from a buffered request because SAM is used for the buffer.
- A map set allocated from the port map pool (map sets 8 through 31) is used if the request is nonbuffered and the data is in a user partition. The current user's map set is copied into the allocated port map. The allocated port map is returned to the port map pool when the request is completed. If all port maps are in use, the program that issued the request is suspended until a map becomes available.

Buffer Limits

In order to permit programs to run as fast as possible, all output should be buffered, linking up new requests to the device (through the DVT of the driver). However, every buffered request uses up some of the available memory in SAM (and requires some system overhead). Also, most programs can supply data to a device faster than the device can handle it. Therefore, a limit is placed upon the sum total of all buffered requests that can be linked to a DVT at any one time.

There is an upper buffer limit and a lower buffer limit. They work in unison as follows:

1. When a program makes an I/O request that is buffered, the system checks the sum of all requests currently linked to the device. If the sum exceeds the upper limit, the program is suspended and not permitted to make the request.

2. When the sum of the requests falls below the lower limit, all programs suspended for exceeding the upper limit are rescheduled and permitted to repeat their requests.

Buffer limits are set when configuring the system.

The system does not check the amount of SAM available if the priority of the requesting program is between 1 and 40. SAM can be used up quickly in this case.

Disk Mapping

As with all I/O devices, a read/write request to a disk is actually addressed to a logical unit (LU) instead of the actual device. However, depending upon the amount of disk space available, several LUs may be assigned to a single disk drive, with each LU pointing to a separate area. This division into several LUs is known as mapping the disk.

Some disks (for example, flexible diskettes) are mapped on a physical basis, although this is not required by the system. That is, each diskette is assigned an LU number. This is reasonable because of the amount of file space available on a diskette. A hard disk has much greater space on a single physical disk. Since there is no physical basis for subdividing the large disk, a mapping table is used.

A disk that is mapped is treated as several devices instead of a single device. Each LU that refers to a different area of the disk becomes a separate device and has a device table (DVT), as do other devices in the system. The portion of the disk associated with that LU is recorded in the DVT. Thus, there is no single, unified table describing the usage of the disk. Instead the information is scattered among all the LUs that refer to the disk. Each LU is considered a disk volume, which is also known as a disk cartridge, when dealing with the FMGR files. (Refer to the FMGR File Cartridges section in this chapter for more information.) This mapping arrangement is set or determined at the time the system is configured.

File Management

In the operating system, disk space is managed by a file management package or file system that allows user programs to define the disk area needed and to protect it or share it with other programs and the system itself. It also eliminates the need for programs to consider the characteristics of the disk since the method of access is the same regardless of disk type.

A file is a collection of records that may be of variable length or fixed length. Files may also be of fixed or variable length. In addition to the fixed and variable length files, a third category is the type 0 files, which are used to read or write to non-disk devices as if they were actually files. The file categories, types, and their descriptions are given in Table 1-1.

The file size is initially defined at the time the file is created. Creation of a file means that an entry is made in a file directory with the file name, file type extension, type, and size. The directory entry also reserves a portion of the disk for that file.

Table 1-1. File Types

Category	Type	Description
Symbolic link	−1	Pointer to another file descriptor
Control of Device	0	Non-disk device
Control length, random access	1	Fixed length 128 word records
	2	Fixed user-defined record length
Variable length, sequential access	3	Variable length records; any data type
	4	Source program file; ASCII or ASCII data file
	5	Object program file; relocatable binary
	6	Executable program; memory image code
	7	Object program file; absolute binary
	8	Type 8 and higher files are user-defined data formats with the following exceptions
	12	Byte stream file
	6004	CALLS catalog file

The files are organized in directories created for each user. They can be organized in a hierarchical manner with directories nested within directories. The nested directories are called subdirectories. Protection is associated with each directory (or subdirectory) that can be defined to control user access to the files.

In order to read from or write to a file, it must first be opened. A file is automatically opened when it is first created but may then be closed and reopened later. While the file is opened, the program calls subroutines from the file management package to read or write records in the file.

If a program writing to a file attempts to write past the last record, the file management package automatically extends the size of the file. Each file extension creates or modifies an entry in the file directory and allocates more disk space to the file. This additional space is known as a file extent. The extent may be located anywhere on the disk volume where there is sufficient space, not necessarily contiguous to the original file or the previous file extent. Up to 32767 file extents are permitted on the original file.

When a file is purged, the file space and the directory may be re-used when a new file is created. If for some reason the disk needs to be packed, the MPACK program is used to pack the disk volume. Refer to the *RTE-A User's Manual* for details.

Serial and Random File Access

Random access means that a record may be read/written independently of the surrounding records. Even though a disk may be accessed in a random manner, the individual records in a file may be accessed randomly only if the records are fixed length. Files that have variable length records must be accessed in a serial manner.

Records in the file (or in each extent) follow each other in sequence on the disk; that is, the second record in the file follows the first record, and so forth. If any given record is to be replaced or updated, then the new record does not interfere with the surrounding records if it takes the same amount of space on the disk.

If a file has variable length records, updating a record in that file requires reading through all records before the one to be updated. For example, if the update is to be made to record 10,000, it takes a considerable length of time for the program to reach that record. Then, should the update to this record increase the record size, more processing must take place. First, the original file is copied to a second file, up to the record that is to be updated. Then the updated record is written to the second file and the remainder of the original file is copied into the second file behind it. At the end, the second file may be copied back to the original file and then purged. However, if the information does not have to be accessed constantly, then variable length records can be used to conserve disk space. For example, the source code for a program is stored in a file type with variable length records.

A file with fixed length records may take more space since some records may contain less data than others (some may be empty); but the advantage is fast access. In order to reach record 10000, all that is necessary is to determine the record length, multiply by 10000 and offset from the beginning of the file.

Shared Files

Any file may be simultaneously opened to any number of programs, except for FMGR files, which are limited to seven programs. Alternately, it may be opened exclusively, thereby locking out other programs until the program closes the file, which releases it.

Programs accessing a shared file may be performing read, write, or a combination of both. If a shared file is to be read only, the file may contain fixed or variable length records. If it is to be updated, it must be of a file type with fixed length records; otherwise it is impossible to prevent the file from being corrupted, since each program may maintain a different position in the file. In addition, some method of preventing two programs from accessing the same record at the same time must be used. This problem is solved by the use of resource numbers, a feature provided by the operating system.

Disk Volumes

File space can be modularized in units of disk volumes in the file system. A disk volume may exist for every logical unit (LU) that refers to a disk. A simple example would be a dual flexible disk drive, which is capable of physically containing two diskettes.

The presence of each diskette is not automatically known to the file system. It must first be mounted, which means that it is added to a master list of available volumes and a check made to ensure that it has valid file directories. This is a logical mount, since it does not involve moving any equipment. However, the diskette must be physically in the correct position before it can be mounted.

When a disk volume is no longer needed, it is dismounted, which takes it out of the master list.

FMGR File Security

For a system containing files managed with the FMGR program, the file descriptor includes a file security code. The file security code can be used to protect the file from being overwritten, accidentally purged or even examined unless the security code is specified. The security code is necessary because FMGR files do not have ownership read/write protection as do other files. It is stored in a single 16-bit word in the directory entry and can be any of the items shown in Table 1-2.

Table 1-2. FMGR File Protection

Security Code	Protection
Zero (Null)	No protection
Positive integer or two ASCII characters	Protect against accidental purge and file overwrite. Only read access without code.
Negative integer	Protect against accidental purge. No read or write access without code.

FMGR File Cartridges

In the FMGR file system, the disk volumes are called cartridges. Each cartridge has its own directory of files that are present on that cartridge. A file cannot cross a cartridge boundary. This makes each cartridge independent of other cartridges.

A file cartridge has two identifiers, which are contained in the directory of that cartridge:

1. A cartridge number. This number is unique in the master list (you cannot have two cartridges with the same number). The number is an integer from 1 to 32,767 or two ASCII characters.

2. A cartridge label. This is a six character ASCII name and need not be unique in the master list.

Both the cartridge number and label may be assigned or changed by an operator command. As an example of usage, a cartridge number might be chosen to correspond to a project number and a cartridge label chosen to be the name of the individual to whom the cartridge belongs.

Since file cartridges are independent, a file name is unique only to the cartridge. That is, the same file can appear on different cartridges.

The identifier for a file is a namr. A file namr can be a negative logical unit number or a file descriptor. A file descriptor has the following format:

<file name>:<security code>:<cartridge number or −lu>:type:size

For example,

```
MYFILE:JB:1313:4:516
```

where MYFILE = file name
JB = file security code
1313 = cartridge number

The disk logical unit associated with the file cartridge may be given in place of the cartridge number, in which case the number appears as negative. However, the cartridge number is a more reasonable choice for programs since the cartridge number can remain the same from system to system.

When a FMGR file is purged, the file space and the directory may be re-used when a new file of the exact size as the purged file is created. Otherwise, to recover used space, the disk must be packed with the FMGR PK command. Refer to the FMGR program description in the *RTE-A User's Manual* for details.

2

Memory Management

Introduction

This chapter discusses how the RTE-A Operating System manages the computer memory. A physical memory map is shown in Figure 2-1.

The physical computer memory is categorized as follows:

User Partitions (Dynamic and Reserved)
Power Fail Storage Area
System Available Memory
Extended System Available Memory
System Message Block
System Common Partition
System Tables
OS/Driver Partition
System Partition

The User Partitions make up user memory. User programs reside in this area. The rest of physical memory, called system memory, contains the operating system and necessary components. No part of system memory is ever swapped to secondary memory (disk). RTE-A uses the Dynamic Mapping System to allow users access to a maximum of 32M bytes of physical memory

Dynamic Mapping System

The Dynamic Mapping System (DMS) is a scheme used in the 16-bit HP 1000 A-Series computers to allow users access of up to 32M bytes of physical memory. Only 64k bytes of physical memory can be accessed at one time. The DMS scheme maps the portion of the necessary physical memory into 64k bytes of logical memory. This is implemented via 32 sets of 32 map registers. A separate special-purpose register called the working map register holds the map set number that is currently active, indicating which physical memory pages are used for fetch, read, and write instructions.

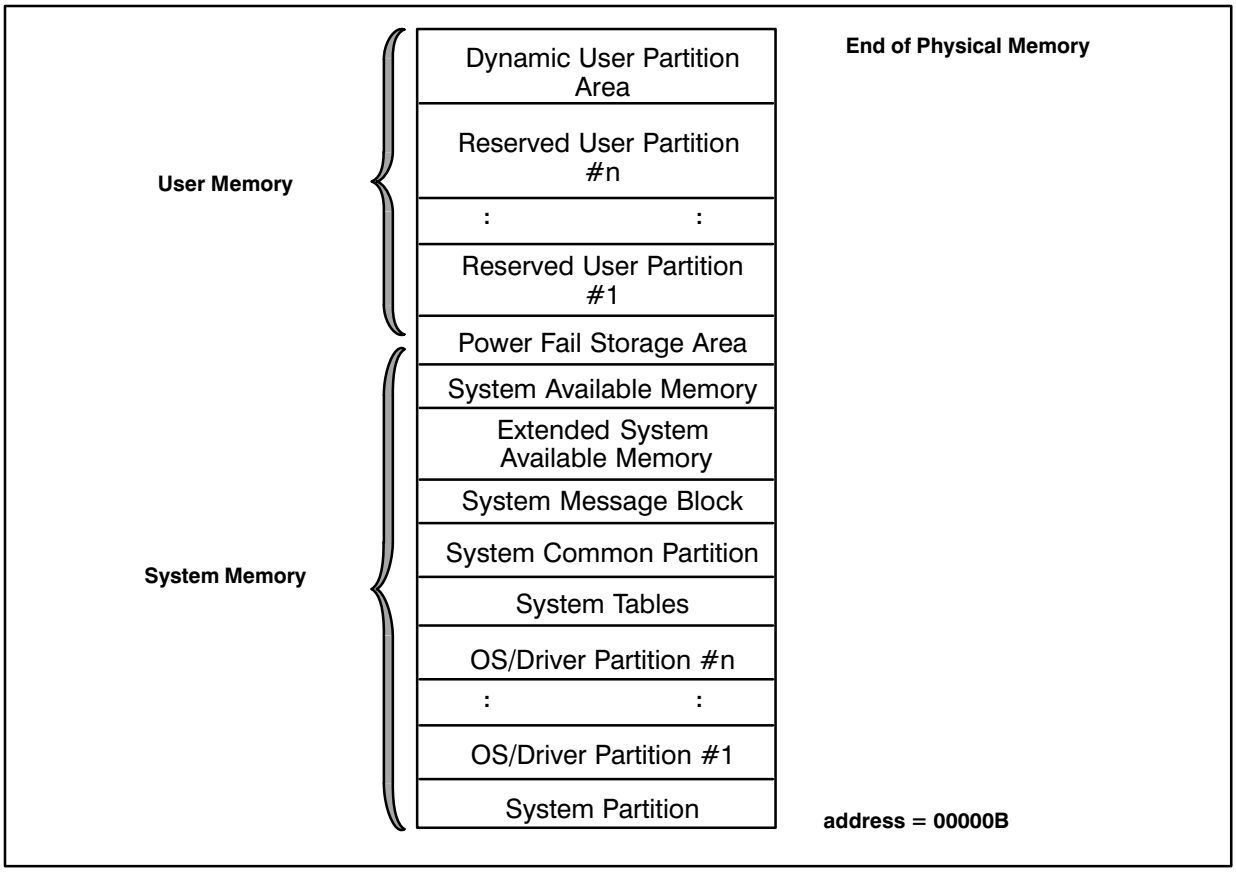


Figure 2-1. Physical Memory

In RTE-A, the following convention is used in the DMS:

Map 0 : Operating System (not including system common)

Map 1 : System Messages Block

Map 2 : User Program (or CDS Program Data)

Map 3 : CDS programs: User Program (Code). Non-CDS: Reserved

Map 4 : SAM

Map 5 : Reserved

Map 6 : Reserved for NS-ARPA/1000 and ARPA/1000

Map 7 : Auxiliary Map

Maps 8-31: Port Maps

The operating system is always mapped into map set 0, which is modified only when mapping a driver into the driver partition. When the operating system is executing, the working map register is map set 0 and when a program is executing, it is map set 2. For CDS programs, the user program is mapped into map set 2 (data) and map set 3 (code).

If SAM and XSAM occupy the same block of memory, map 4 is used to refer to XSAM and SAM; map 5 is unused in this case. Because of this, there is a variable $XSMAP in VCTR that should always be used to access XSAM. It contains the map number (4 or 5).

The operating system accesses its messages and SAM through special cross-map instructions that allow access to a different map set. Map set 7 allows the user program and port maps to remain unaltered during non-DMA I/O. This auxiliary map is primarily used by device drivers when the interface drivers are concurrently using the port map.

The 24 port map sets (8-31) are dynamically allocated and deallocated as needed for the 48 (maximum) I/O channels. This provides up to 24 channels of concurrent DMA.

User Partitions

User partitions are blocks of physical memory reserved for programs. The operating system allows for both reserved and dynamically allocated user partitions. The system allows multiple programs to exist simultaneously in memory, and to compete for CPU time on the basis of priority. Each program executes during the time that all higher priority programs (if any) are waiting for I/O completion or for some other event to occur.

In systems with a disk, there may be more active programs than the number that can be simultaneously in main memory. Programs with lower priority may be copied to disk (swapped) together with any data they have accumulated, to allow higher priority programs to be loaded into main memory for execution.

Program swapping may occur many times in one execution of the program; this can severely affect the performance of the program. Care should be taken to ensure that active real-time programs are assigned to their own reserved partitions, or that there is sufficient dynamic memory available for the real-time programs to remain in the dynamic memory area without swapping among themselves.

In a system with VC+, a CDS program has at least two partitions, one for program code and one for data. In this case, the term user partition applies to data partition unless noted otherwise. The code partition contains only program code plus code page 0 that is reserved for system use.

A typical user partition (reserved or dynamic) is shown in Figure 2-2. Items shown in this figure are described in the following paragraphs.

For RTE-A systems with VC+ using CDS programs, the user partition for CDS programs is shown in Figures 2-3 and 2-4.

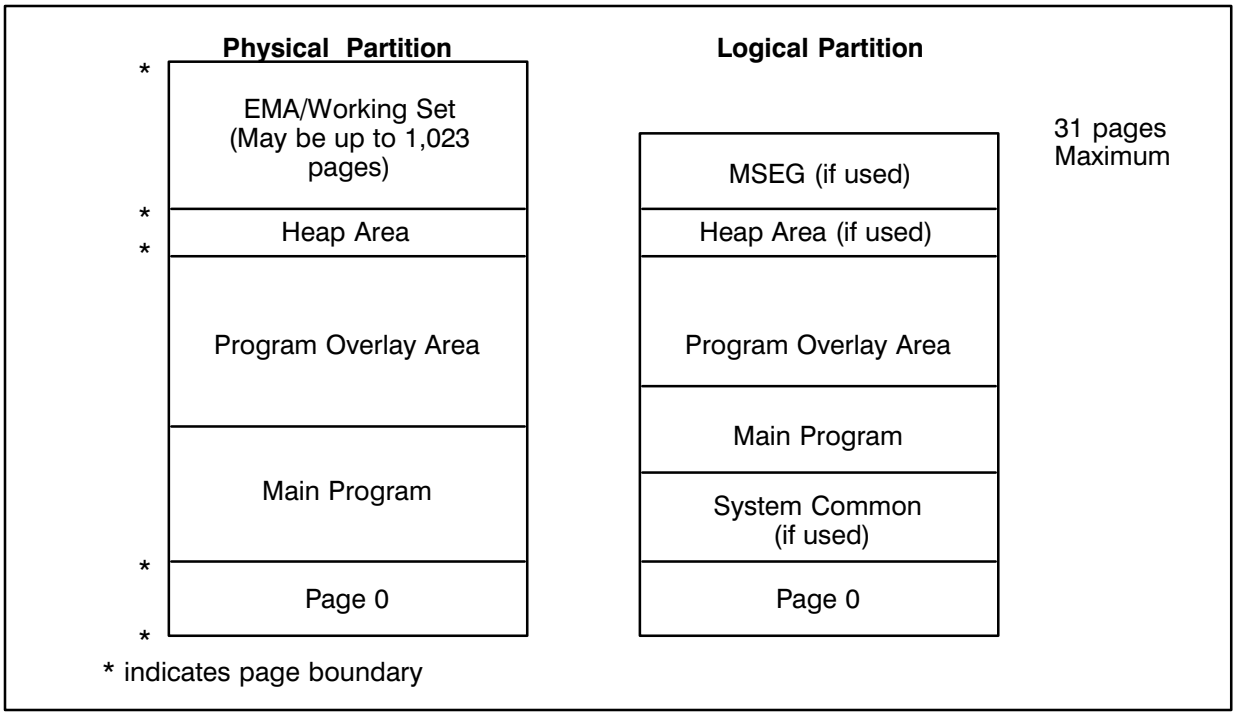


Figure 2-2. User Partition Memory Map

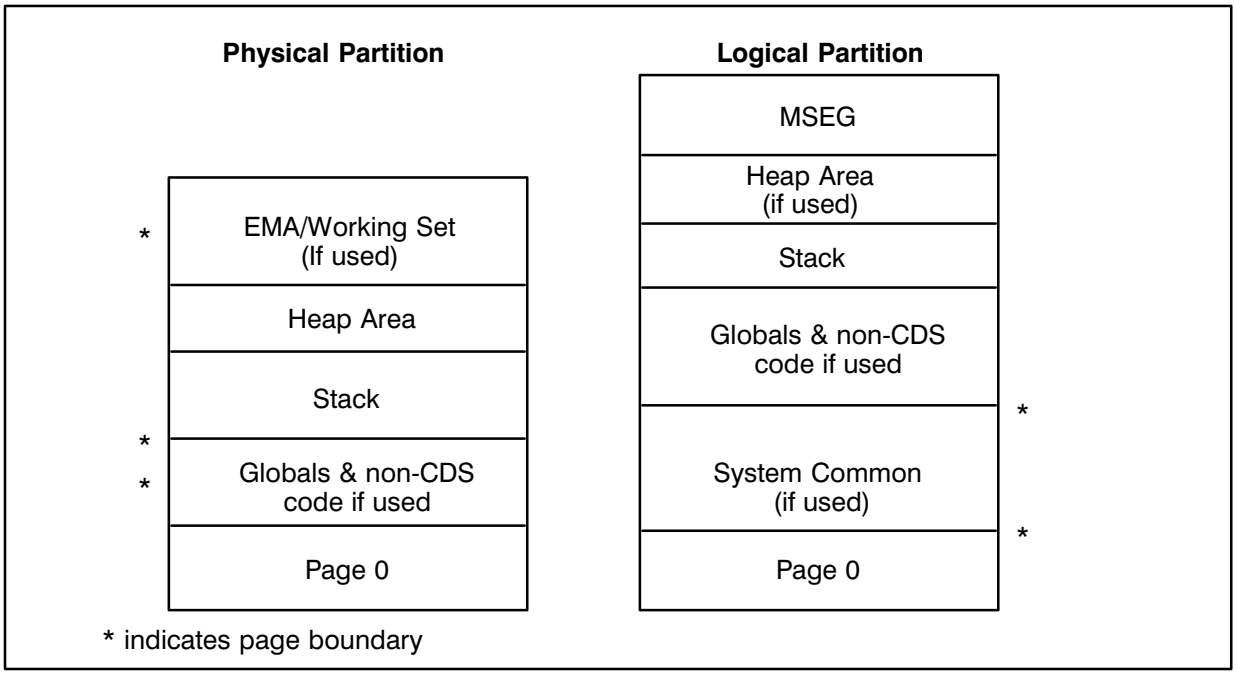


Figure 2-3. CDS Data Partition

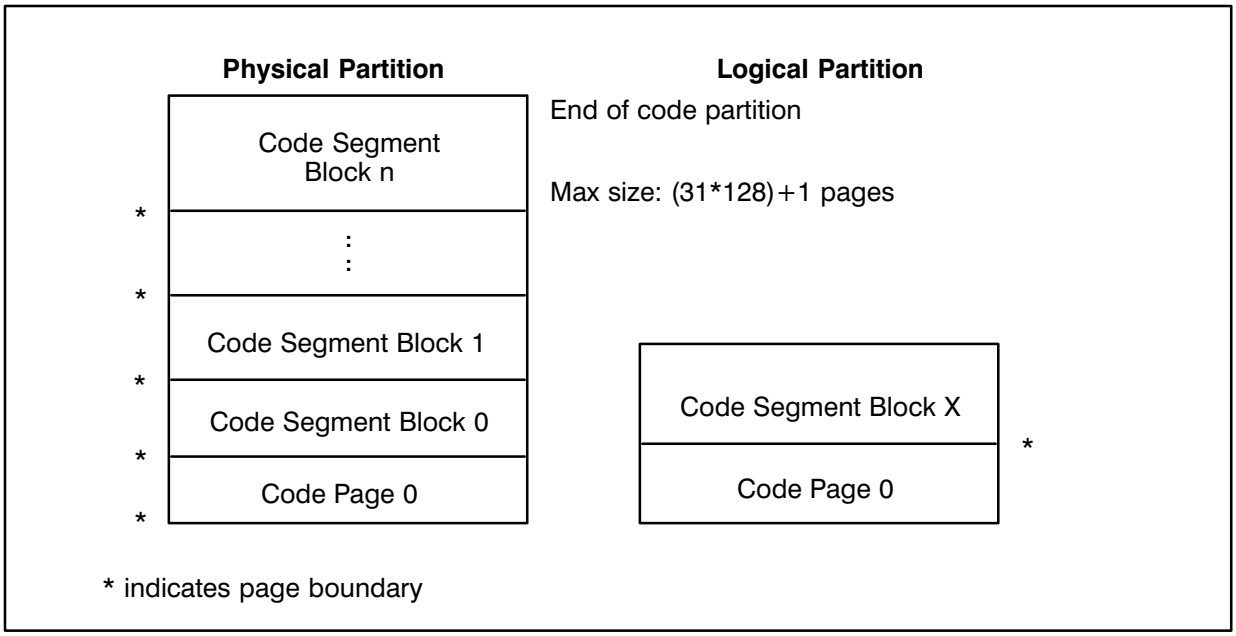


Figure 2-4. CDS Code Partition

VMA/EMA Mapping Area

The VMA/EMA mapping area belongs to the program that specifies it. VMA/EMA subroutines manage the mapping area, using the upper two pages of logical memory to map in VMA/EMA data from physical memory.

Program Available Memory

The size of a program may be declared to be larger than the actual size of the program. It can be done interactively with an operator command or when the program is relocated. The SZ or DT command can be used for non-CDS or CDS programs respectively. The additional declared memory can be used as buffer space. To do so, the program calls LIMEM or EXEC 26, to determine the starting address and size of the area, and to inform the system of the need to swap the extended area.

Overlay Area

A large non-CDS program may be separated into a main program and related overlays, thereby allowing it to execute in a partition smaller than its total size. While such a program is executing, its main is kept in the partition continuously, and its overlays are loaded from the disk as required via a call to SEGLD. A newly loaded overlay is written over the previously loaded overlay without saving any data that may have been accumulated in the overlay area. Therefore, overlays should keep all global data in the main program area.

Main Program

The total size of the main program (non-CDS), the largest segment, and any appended subroutines may be as large as 32k or as small as 2k (including the base page). Any logical pages in the 32k-word logical address space that are not required for proper execution of the program are write protected. This prevents a user program from accidentally modifying its own code or data.

System Common

System common does not physically reside in any user partition. It occupies a special partition that is mapped to those programs that request access at relocation time. Refer to Chapter 8 of this manual for more information.

Page 0

The following locations in the user base page (page 0 in the data partition) are reserved for the indicated values:

```
     loc 0   =  Starting physical page of partition
         1   =  Not used
         2   =  X-Register save word
         3   =  Y-Register save word
         4   =  $VMA$ or $EMA$ or 0 (virtual or extended memory
                fault-handler address)
         5   =  Page table (PTE) physical page high-order word
         6   =  PTE index for unbuffered class VMAIO
         7   =  Original Z-buffer request address
        10   =  user data map save area  - register 0
        11   =           .                - register 1
        12   .           .              .           2
         .   .           .              .           .
        47   .           .                 register 31
       50-107    Not used
       110   =   CS-mode (bit 15), Q-Register (bit 14-0) save area
       111   =   Z-Register save area (upper stack bounds)
       112   =   CS/Q-Register initial value
       113   =   Z-Register initial value
       114   =   PTE physical page low-order word
       115   .   reserved for future usage
        .    .           .
        .    .           .
       140   .           .
       141- 1777 are used for the user program base page links.

For CDS programs, the locations reserved are as follows:

     loc 0   =  Starting physical page of data partition
         1   =  Starting physical page of code partition
         2   =  X-Register save word
         3   =  Y-Register save word
         4   =  $VMA$ or $EMA$ or 0 (virtual or extended memory
                fault-handler address)
         5   =  Page table (PTE) physical page high-order word
         6   =  PTE index for unbuffered class VMAIO
         7   =  Original Z-buffer request address
        10   =  User data map save area  - register 0
        11   =           .                - register 1
        12   .           .              .           2
         .   .           .              .           .
        47   .           .                 register 31
        50   =  User code map save area  - register 0
        51   .           .              .           .
        52   .           .              .           .
         .   .           .              .           .
       107   .           .              .  register 31
       110   =  CS-mode (bit 15), Q-Register (bit 14-0) save area
```

```
 111   =  Z-Register save area (upper stack bounds)
 112   =  CS/Q-Register initial value
 113   =  Z-Register initial value
 114   =  PTE physical page low-order word
 115   .  reserved for future usage
  .    .          .
  .    .          .
 140   .          .
 141   =  Copy of Code Segment Table skeleton indicating
  .       location of all code segments on disk
  .       (0 - 512 words long)
  .                .
1141      (maximum)
1142   =  Used by loader for user base page links
          generated for non-CDS code
  .                .
1777               .
```

Power Fail Storage Area

A page of physical memory is reserved for use by the power fail driver (ID*43). The page number is stored in system entry point $PFMP.

If the power fail driver is not included when the system is generated, a page is not allocated. The unallocated page is included in the user partition area.

System Available Memory (SAM)

System Available Memory (SAM) is a block of memory managed by the system for the following purposes:

I/O Buffering
Class I/O
String Passage
Spool Nodes

This block of memory is not in the system logical memory; it is accessed through map set 4. Using the dynamic mapping system for SAM releases the system logical address space for other uses (for example, larger system tables and more I/O drivers). It also allows a maximum of 32763 words of SAM. The size of SAM can affect program performance in the system: the larger the size of SAM, the better the program performance.

Programs making requests that require memory are placed in the memory suspend list (ordered by program priority) if there is not enough SAM to meet the need. When some amount of SAM is returned, programs whose request requires an amount of SAM less than or equal to the amount returned are re-dispatched.

The following sections describe how SAM is managed and provide some guidelines for determining how much SAM is required during system configuration.

SAM Management

At system bootup, SAM is one contiguous block of memory. When system memory is required, the SAM module in the system is called to allocate a portion of SAM. If there is not currently a contiguous section of SAM large enough to meet the need, the program is placed in the memory suspend list.

When the system is done with the allocated portion of SAM, it calls the SAM module again to return the memory to the free list. Portions of the original contiguous block of SAM may be in use at any one time and they may be returned in an arbitrary order. This results in fragmenting the contiguous block into several blocks of SAM. Each free block is linked to the next block in the manner indicated in Figure 2-5. Note that two words of overhead are needed to keep track of each block of SAM.

When called upon to allocate a free block of SAM, the SAM module allocates the required block from the first block that can meet the need. The newly allocated block is taken from the existing block (at the high end of memory) and any remaining memory is linked into the free list.

When memory is returned to SAM, the SAM module checks if it is contiguous with the bottom of any free block. If so, the two blocks are joined and linked as one block in the free SAM list. No attempt is made to order the blocks by size in this linked list.

The size of the largest free block is maintained in word 0 of SAM. This location contains the one's-complement of the size of the largest free block.

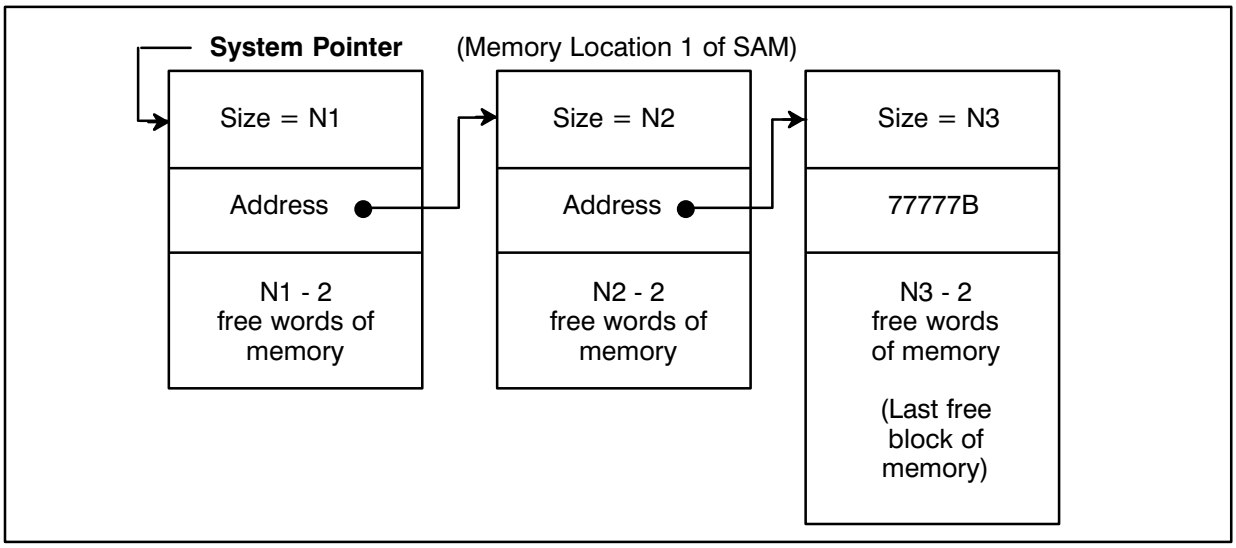


Figure 2-5. System Linking of Free Memory in SAM

Each block of SAM requested by the system is larger than the size of the user data (for example, an I/O buffer, a runstring or a class buffer) by an amount that depends upon the actual use. The additional memory is used for header information to keep track of the data. For example, a block of SAM used to construct a buffered request to a driver contains information such as the type of request and the buffer size as well as the user data. The overhead for each of the four uses of SAM is as follows:

I/O Buffering:	16 words
Class I/O:	16 words

String Passage: 3 words
Spool Nodes: 1 word

To determine the total number of words of memory for SAM at generation time, we may consider each of the four needs separately.

I/O Buffering

Each DVT in the system may have a list of requests linked to it. The number of words linked to each may exceed the upper buffer limit for the DVT, since only the length of the current list is checked when adding new requests. However, for the purpose of estimating SAM usage, assume that the maximum words linked on each DVT is equal to the upper buffer limit. Since the buffer limit can be set independently for each DVT and the choice of limits depends upon the application, this can be quite complex.

One way to start is to adopt a goal of two buffers linked to each DVT simultaneously. The buffer limits would then be set accordingly for each device.

The largest typical buffer length would probably be for the disk, which is about 129 words for a read or write operation. However, disks are not buffered because of possible disk read/write conflicts and because of the size requirement for each disk. Only a few disk LUs would take up most of the SAM available. Therefore, the disk need not be a consideration when calculating the size of SAM.

For terminals, assume a typical buffer length of 150 characters (75 words), approximately 150 words for two buffers. The buffer size must be multiples of 16. If the upper buffer limit for each terminal were set at 200, then up to 275 words (200 + 75 = 275) could be accumulated on each terminal at any one time.

The buffer length for a printer could use the same figure as calculated for the terminal. Thus, if we had one disk, three terminals, and a printer on the system, allocate 1000 words just for automatic buffering. Fewer words might suffice provided that fewer than the two buffers are actually accumulated for each device at any one time.

Class I/O

Class I/O is used by optional HP software products such as the networking products. Before system configuration, the information on SAM recommendations for each of these products must be determined and the actual figures added to the amount of SAM needed for the system.

The amount of SAM required for class I/O calls is totally dependent upon the way the programs making the calls are written. The system permits any size user buffer for class calls, up to the maximum size of SAM. Therefore, programs are normally written to ensure that class buffers do not remain in SAM for an extended period of time. This requires some sort of sychronization. Class I/O calls are described in the RTE-A Programmer's Reference Manual.

Another user of class I/O is the HP-supplied subroutines REIO and XREIO. These subroutines are used to translate an EXEC input request into a class read, followed by a class get. This suspends the program until the request completes but makes it swappable during the delay. However, if the LOAD module (which controls program swapping) and the CLASS module are not generated into the system, these calls are treated as an ordinary EXEC read request. For a

system that has three terminals with one read request pending at each, an allocation of 250 words of SAM is recommended.

String Passage

If the STRNG module is included in the system, then the runstring is saved in SAM when the program is scheduled. The first EXEC 14 request causes the memory saved for the string to be deallocated. This normally happens early in the program so that the string is removed quickly.

In the worst case, all of the programs in the scheduled list at one time would have their runstrings in memory. Typically, a runstring would be about 80 characters or 40 words. Multiply this by the number of programs that would accept runstrings to get the total SAM requirement.

Spool Nodes

Spool Nodes are linked off a DVT when there is spooling occurring on the device associated with the DVT. The spool node contains the number of the session that enabled spooling, the spool code or redirection LU, and other required information.

Typical SAM Requirement

In a small system (without subsystems that require large amounts of SAM), the total number of words required for SAM might be:

I/O Buffering	= 600 words
Class I/O	= 400 words
String Passage	= 500 words
Spool Nodes	= 80 words
Total	= 1580 words required for this system

Because SAM is allocated in increments of pages, 2048 words are allocated. If you are in doubt as to the size of SAM required, it is better to overestimate than to underestimate. More SAM allows more activities to occur simultaneously; not enough SAM decreases system performance.

Extended System Available Memory (XSAM)

Extended System Available Memory (XSAM) is very similar to SAM. It is used for the following purposes:

- ID segment extensions
- Signals
- UDSPs and LU access tables
- SHEMA table entries
- SHEMA Association Blocks

The management of SAM and XSAM is identical because of the fact that they can reside in the same block of memory. There are differences that arise because of the fact that they can share the same memory or can use separate blocks of memory. These are as follows:

Amount of memory used:

Shared: Up to 32763 words for both SAM and XSAM.
Separate: Up to 32763 words for each SAM and XSAM.

Map used to reference XSAM:

Shared: Use map 4.
Separate: Use map 5 ($XSMAP in VCTR contains the map number for XSAM in the current system).

Free list:

Shared: There is only one free list for both SAM and XSAM. It starts at memory location 1 in map 4.
Separate: SAM has a free list in map 4 and XSAM has a free list in map 5. Both start at memory location 1.

If a program makes a request that requires XSAM and not enough is available, it is placed in the XSAM memory suspend list always; this is true even if SAM and XSAM share memory.

XSAM is used for signals, UDSP/LU access tables, and the prototype ID list. Descriptions of how it is used for these applications are given in the following sections. XSAM is also used for shareable EMA table entries and SHEMA Association Blocks as described in Chapter 11.

Signals

Every program that receives signals has a Signal Control Block (SCB). It contains information necessary for signal operation and resides in XSAM. The signal buffer limits, active signal, blocked signals mask and pointers for queued signals are examples of the information contained in the SCB. The pointers that reside in the SCB point to Signal Queue Blocks (SQBs) that also reside in XSAM. There is one pointer in the SCB per signal number. If a pointer is nonzero it points to a SQB that contains the Signal Dependent Data for that signal. There is also one pointer that is used to point to SQBs that have been pre-allocated. An example of this is for class completion signals. The SQB is allocated when the class write (read, control, or read/write) request is made, even though it is not needed until the request completes. Additionally, when a timer signal is started, a block is built in XSAM that describes the signal. The system maintains the list and delivers a signal when the timer expires.

UDSP/LU Bit Maps

The table containing the User-definable Directory Search Path (UDSP) definitions for a session is kept in a block of SAM. LOGON allocates the SAM block when a user logs on, and places the address of the block in the user ID table.

The amount of SAM allocated for a UDSP is determined by the UDSP definition in the user account file. The values are set when a user's account is created or modified.

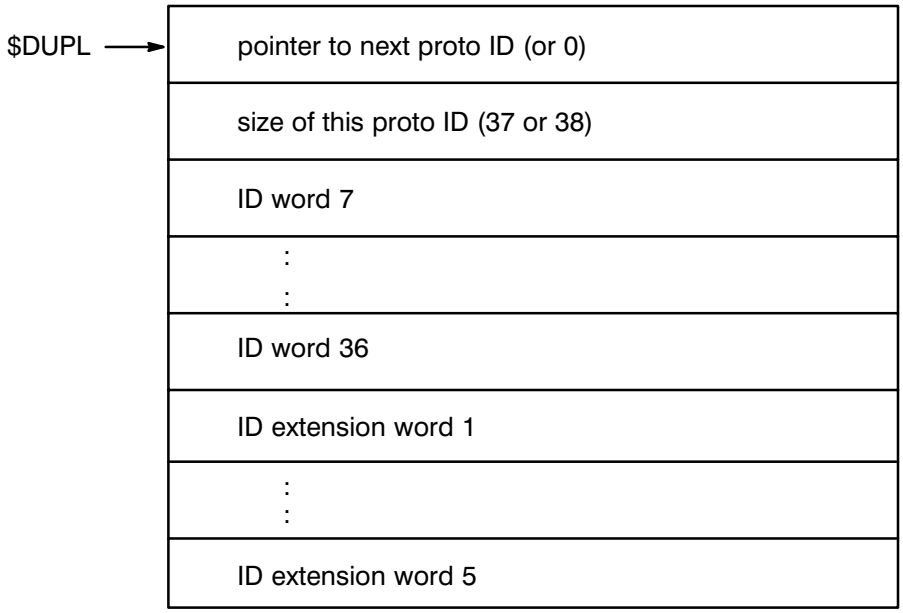
At the front of the UDSP are 16 words used for the LU access table. Each bit represents one LU. If the bit is set, the user can access the LU. If the bit is not set, the I/O system prevents the user from accessing that LU.

Prototype ID Segments

The list containing the prototype ID segments (proto IDs) defined in the system is kept in XSAM. When a user RP's a program with the "D" option to create a proto ID segment, the system allocates a block of XSAM, sets up the proto ID and puts it in the proto ID list.

The amount of XSAM allocated for a proto ID is 37 or 38 words (due to the XSAM management/allocation scheme).

The proto ID segment has the following format:



$DUPL is a pointer to the first proto ID segment.

Note that words 1 through 6, and words 37 through 45, are NOT saved.

The system maintains the list and uses the proto IDs to create program ID segments when appropriate. Proto IDs are removed from the list when a user issues an OF command with the "D" option. The list can be displayed with WH using the "D" option.

System Message Block

The system message block is a data structure that contains system messages and information that specifies how many numbers/ASCII strings are embedded and where. Each time the system wants to print a message it calls a routine ($PRMG) that integrates any numeric/ASCII tokens into the string and then prints it. The message block is accessed through map set 1. Refer to the Language Message Table description in Chapter 11.

System Common Partition

The system common partition may range in size from zero to thirty pages. When a program accesses system common, this partition is mapped into the user logical map. It does not reside in the program partition in physical memory. However, the combined size of system common and the largest program that accesses system common must not be more than 32 pages. The system common partition is shown in Figure 2-6. For more information, refer to Chapter 8 of this manual.

In Figure 2-6, $BCOM and $LCOM are system entry points that contain the logical starting address of blank and labeled common, respectively.

Figure 2-6. System Common Partition Memory Map

System Tables

The operating system is designed for maximum flexibility in the number and type of devices that it can control, and in the number of programs and memory partitions that may be required to accomplish the intended applications. In order to provide flexibility, and at the same time minimize system memory requirements, variable length tables are used to configure the operating system. The various system tables and their entry points are shown in Figure 2-7, and are described in more detail in Chapter 11.

	Device Tables	$DVTA
	Interface Tables	$IFTA
*	Logical Unit Table	$LUTA
	Interrupt Table	$INTA
*	Class Table	$CLTA
*	Resource Number Table	$RNTA
*	ID Segments	$IDA
*	Memory Descriptor Table	$MEM
*	Swap Descriptor Table	$FSWP
*	Shared Programs Table	$SPTB
*	System Memory Block	$SMB
	Multiuser Table	$UIDA

Figure 2-7. System Tables and Entry Points

When allocating space in system memory for the tables, the system first checks the end of the system partition. Because the OS/driver partitions must start on a page boundary, there may be unused space between the end of the system partition and the start of the OS/driver partitions. Tables marked with an asterisk are put in this space if they fit. Otherwise tables are put in the next available space in system memory allocated for system tables, beginning with Device Tables.

OS/Driver Partition

The OS/Driver partition is a contiguous set of pages in the system logical memory map. The size of this partition is the same as that of the largest OS/Driver partition defined at generation time. As a partitioned system routine or driver is about to be called, the system maps the appropriate OS/Driver partition into this logical address space for subsequent access. Note that privileged drivers and non-mappable drivers must be loaded with the non-partitioned system modules.

Chapter 5 contains a detailed discussion of partitionable modules.

System Partition

The system partition is the area of memory reserved for the operating system modules that cannot be located in a partition, and the device and interface drivers that do not support driver partitioning. Figure 2-8 shows how physical memory maps into the system partition.

In RTE-A, SAM and system messages are in physical address space and are mapped and accessed through separate maps.

Privileged and Non-Partitioned Drivers

The I/O drivers are the parts of the system that directly control the devices connected to the system. Drivers are software modules that provide a uniform programmatic interface to the hardware. The use of drivers allows writing of programs that read and write to many different types of peripheral devices without any concern for device protocols, record blocking, and other interfacing requirements.

Privileged drivers are drivers whose interrupts are not processed by the RTE-A Operating System. Such drivers offer improved response time but must perform their own internal housekeeping. Refer to Chapter 7 for more information on drivers.

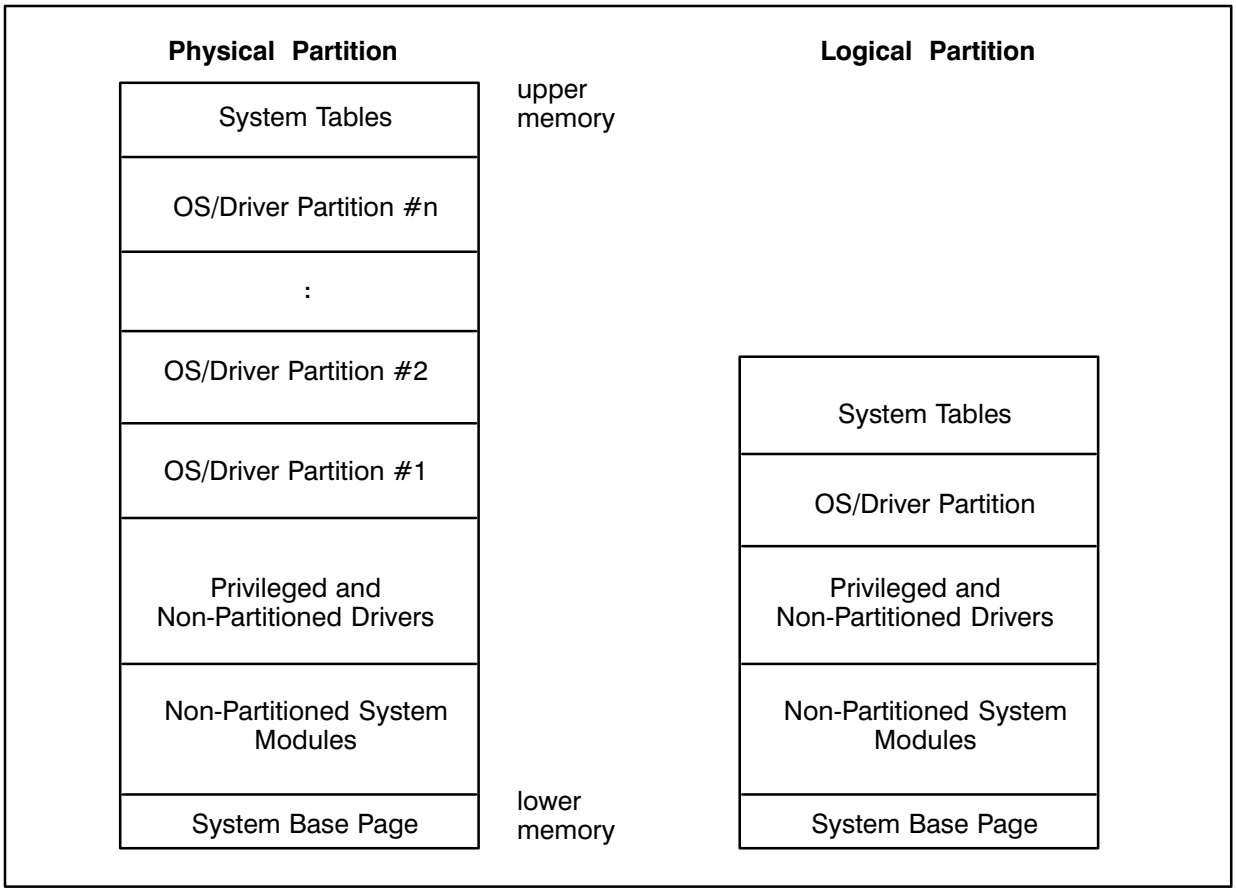


Figure 2-8. System Logical and Physical Partitions

Non-Partitioned System Modules

The system modules coordinate and govern the functions of the various programs and drivers. The operating system is designed to maximize the use of system resources such as main and secondary (disk) memory, CPU, and peripheral devices.

At the same time, it allocates these resources according to the priority of the requesting process. The RTE-A Operating System includes modules that handle CPU interrupts and I/O control, memory management, program loading and swapping, EXEC calls, and a small set of commands.

Some system modules can be placed in OS/Driver partitions to reduce the number of logical pages required for system modules (and thus increase the number of logical pages available for system tables.)

Chapter 5 contains detailed descriptions of the system modules. The descriptions indicate the function of the module and whether or not the module is partitionable.

3

Programs and Partitions

This chapter describes how the RTE-A Operating System manages programs and partitions. Throughout this chapter, references are made to Code and Data Separation (CDS) programs that are provided in VC+. If you do not have CDS programs, ignore all CDS references.

Program Priority Boundary

The generator BG command sets the priority boundary (N) between real-time and background programs. Real-time programs are defined as those programs whose priority is higher than the boundary (priorities 1 to N-1). Background programs are defined as those programs whose priority is equal to or lower than the boundary (priorities N to 32767).

The labels real-time and background apply only to programs, not to partitions. A given partition can at one moment contain a real-time program, and at another moment a background program. The states of user partitions are listed and described below.

State 0 - Unoccupied reserved (fixed) partition, or a block of free memory in the dynamic area.

State 1 - Occupied by a swappable background program.

State 2 - Occupied by a swappable real-time program.

State 3 - Shareable EMA area, or an area occupied by a non-swappable program (a program that executed an EXEC 22 request, or is I/O suspended with a buffer within the user partition).

Partition Assignment for Real-Time Programs

When a real-time program not in memory is scheduled for execution, a partition is assigned based on the following criteria; the partition must be in state 0, 1, or 2; the partition must be large enough to hold the program; and the partition must not be occupied by a higher priority real-time program.

A partition in state 1 is not used if it is possible to use a partition in state 0. Likewise a partition in state 2 is not used if it is possible to use a partition in state 0 or 1. A partition in state 3 is not used.

If the program is assigned to a particular partition, it runs only in the reserved partition to which it is assigned; otherwise, a partition is allocated from the free dynamic memory. If there is insufficient free memory, one or more other programs are swapped out of dynamic memory to make room for the program.

The system does not swap a real-time program if it can make room for the program by swapping background programs. Once a real-time program is in memory, it is never swapped to disk unless a higher priority real-time program requires the partition in which it resides, or it is suspended (non-scheduled) and a lower-priority program cannot execute unless the suspended real-time program is swapped to disk.

Partition Assignment for Background Programs

When a background program not in memory is scheduled for execution, the partition assigned is selected from a partition of sufficient size that is in state 0, 1, or 2.

A partition in state 1 is not used if it is possible to use a partition in state 0. If it is not possible to use a partition in state 0 or 1, a partition in state 2 is used. A partition in state 3 is not used.

If the program is assigned, it runs only in the reserved partition to which it is assigned; otherwise, a partition is formed for the program out of free dynamic memory. If there is insufficient memory, one or more other programs are swapped out of dynamic memory to make room for the program. No real-time programs are swapped out if swapping only background programs provides enough memory.

Timeslicing

At generation time, a timeslicing boundary and a timeslicing quantum number are established for the system. Programs with a priority lower than the timeslice boundary are subject to timeslicing. Only programs of equal priority are actually affected by timeslicing, and the timeslice quantum is equal for all programs below the timeslice boundary regardless of their priority.

As an example, suppose the timeslicing boundary (which is different from the background priority boundary) is set at 30. Two background programs at priority 35 are timesliced, and share CPU time equally. But if the priorities of these two programs are 35 and 40, the program at priority 35 runs until it is suspended before the program at 40 executes. This is because only programs of equal priority are actually affected by timeslicing. If the program at priority 35 is compute bound and never gets I/O suspended, then the program at priority 40 does not execute until the program at priority 35 terminates.

As another example, suppose four programs are competing for CPU time. Their priorities are 40, 40, 50, and 50. The two at priority 40 timeslice and, as an example, use up 76 percent of the CPU time. They are suspended for the other 24 percent of the time. The two programs at priority 50 then timeslice the remaining 24 percent of the CPU time.

If the above programs are actually four copies of the same program, then the two at priority 50 require more than 24 percent of the CPU (more than 12 percent each). Therefore, because of timeslicing, two of the four programs get all the CPU time needed (38 percent each), and the other two are forced to share what remains (12 percent each).

Program Overlays

A large non-CDS program may be separated into a main program and related overlays, thereby allowing it to execute in a partition smaller than its total size. While a large program with overlays is executing, its main is kept in the partition continuously and its overlays are loaded from the disk via a SEGLD or EXEC 8 call, as required. A newly-loaded overlay replaces the previous overlay without saving any data that may have been accumulated in the overlay area. Therefore, overlays should keep any global data in the main program area.

Program overlays are identified with short ID segments. One short ID segment is required for each program overlay. The short ID segment is created automatically by LINK and located in the program file, and is loaded and swapped with program main. Short ID segments are positioned between the program main and the overlay area in the program file.

CDS Program Structure

In RTE-A systems with VC+, CDS programs have separate code and data partitions. When a CDS program is linked, LINK puts all the code together and all the data together. Both code and data are stored in the program file on disk. When the program is executed, the operating system allocates two separate partitions for the program: one for code and one for data.

The first page, page 0, is used by the system for a special purpose. A maximum number of 31 logical pages is available for use for code or data. If the code requires more memory than 31 pages, then it is divided into code segments. Each code segment is 31 pages or less. A program may use up to 128 code segments, up to 3968 pages of code are allowed. If more than 31 pages of memory is needed for data, then Virtual Memory Areas (VMA) or Extended Memory Areas (EMA) may be used. (Refer to the RTE-A Programmer's Reference Manual for information on EMA and VMA programming.) The memory required for program data, including EMA if used, is called the data partition. The size of the data partition is controlled by LINK or by the operator DT command.

Code segments are created when a program is linked, either automatically by LINK or by specific LINK commands (refer to the RTE-A LINK User's Manual for details). If automatic segmentation is used, the LINK accumulates the program code until 31 pages are filled. A subroutine cannot be broken apart and the pieces placed in two different segments. Code segments are sized as close to 31 pages long as possible. User created code segments must be from 2 to 31 pages long.

When the program is executed, the operating system allocates a partition to contain the code. The code partition in physical memory may contain some or all of the code segments. The partition is divided into subdivisions called segment blocks. Each segment block is large enough to hold the largest code segment of the program. The code partition also includes code page 0, which is reserved for system use. The CDS program code partition map is shown in Figure 2-4.

The code partition may contain one or more segment blocks. When the program executes, the code segments are loaded from the program file on disk into a segment block as they are needed. When all the segment blocks contain code segments and another segment is referenced, the operating system overlays one of the segment blocks with the new segment. (The segment replacement algorithm is described later.) The more segment blocks that are in the code partition, the more code segments that can be kept in memory. Since loading a segment from disk is relatively slow, making the code partition larger decreases the execution time. The size of the code partition is controlled by the CD command, which is either entered when the program is linked or used as an operator command.

Shared Programs

The following description on shared programs is applicable only if your system has VC+. Shared programs are CDS programs that share one code partition while using separate data partitions, see Figure 3-1 for a typical illustration.

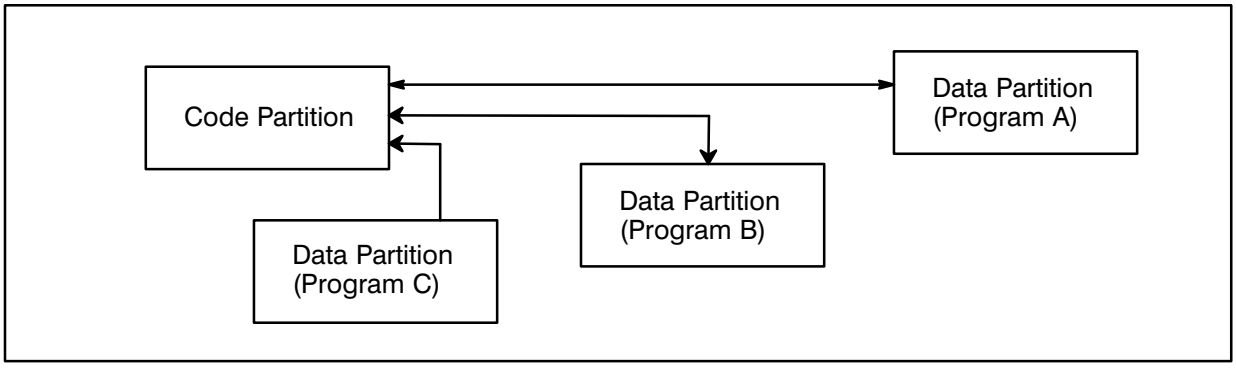


Figure 3-1. Shared Programs

Shared programs are implemented by using a shared program table. There is an entry for each shared program in this table containing information such as the disk address, in-use count, in-system count, Memory Descriptor (MD) pointer and ID segment list pointer of the shared programs.

A program is designated as a shared program only at the time it is linked with LINK. When the program is RP'd, the shared program table is scanned to determine if an entry exists for this program. If so, the in-system count is incremented; otherwise, an entry is established. During dispatch, the shared program table entry in-use count is incremented. If the program is the first of the shared programs to be dispatched, the code partition is loaded into memory. All code segments of a shared program must be in memory at all times during program execution. The data is loaded into another partition and the program is ready to execute.

During the abort/terminate processing, the following events occur: the data partition is deallocated; the in-use count is decremented; and, if it is zero, the code partition is deallocated. If there is no RP'd shared program, indicated by a zero in-system count, the shared program table entry is cleared.

Managing User Partitions

The RTE-A Operating System manages memory using fixed (called reserved) and dynamic memory partitions. The reserved partitions are used by programs specifically assigned to them. The dynamic memory is divided into partitions that are allocated to programs as needed.

When the operating system needs to load or swap-in a program that is not assigned to a reserved partition, it uses the first suitable free block of dynamic memory it finds. If a free block of sufficient size is not available, the operating system searches for a suitable set of non-free blocks of dynamic memory.

The operating system allows programs not assigned to reserved partitions to share dynamic memory on a priority basis, loading and swapping programs into areas of memory that are unoccupied or are occupied by dormant or terminated programs. Swapping is handled using a series of algorithms that combine speed and efficient use of memory.

Allocating Reserved Partitions

In allocating a reserved partition, the system (dispatcher) first checks if the partition is free when a program assigned to a reserved partition is not in memory but is to be executed. If so, the system loads the new program from disk into the reserved partition, and modifies the reserved partition Memory Descriptor (MD) to point to the program ID segment. If the reserved partition is in use as shareable EMA, then the new program cannot run until all programs using the shareable EMA have terminated, since shareable EMA is never swapped out of memory.

If the reserved partition is occupied by a program, then the occupant must be of either lower or equal priority, or higher priority but not running, in order to swap in another program. The system checks the reserved partition MD for the status of the occupant, and makes the swap if the occupant is swappable. A program is not swappable if it is memory locked (EXEC 22 call) or is in the process of doing nonbuffered I/O. If the program in memory is not swappable, the new program must wait until the status of the program in memory changes.

Note that even if the program in memory is real-time and of higher priority than a waiting program, it can still be swapped out by the system. This provision prevents deadlocks; for example, consider what could occur if a real-time program (PROGR) in a reserved partition scheduled with wait a background program (PROGB) that was assigned to the same reserved partition. If the system were not allowed to swap out the occupant, neither program would be able to run, and a deadlock would result.

On the other hand, this implies that for programs assigned to a reserved partition, the real-time fence has no meaning; any program in a reserved partition that is non-scheduled (blocked) may be swapped out for a lower priority program. This consideration should be kept in mind when assigning programs for which real-time response is desired to reserved partitions.

Allocating Dynamic Memory

Swapping a program into dynamic memory is not quite as straightforward as swapping into reserved partitions. In order to complete the swap, the system must find an area of memory of sufficient size that is either unoccupied or occupied by one or more programs that are in a swappable state. Since more than one program may have to be swapped out in order to make

room for the new program, and since one program can change state while the system is swapping another, the search algorithms are more complicated than those for reserved partition swapping. The search sequence is described in the following paragraphs.

First Choice

First, the system searches the Memory Descriptor (MD) list for a free block of memory of sufficient size to contain the new program. This is a search of the free dynamic memory list, which is the doubly-linked circular list of dynamic memory descriptors. This list is unordered. The system makes a first-fit search, starting at free MD list head $FREM.

If a free block of dynamic memory is found, a block of the proper size needed is allocated for the program, then the free dynamic memory list and the adjacent memory list are adjusted, and the system copies the program from its memory-image file or from the swap area on disk into the newly created dynamic partition.

If a suitable free block of dynamic memory is not found, the system notes the size of the largest free block, which was found during the search of free memory descriptors on the list.

Second Choice

If a suitable portion of free memory is not found, the system makes a second search on the adjacent MD list, starting at a cyclical start-search point, pointed to by system variable $STMD. Any set of contiguous blocks that does not include a block in use as a shareable EMA area, or a block holding a program doing nonbuffered I/O or having executed an EXEC 22, is a usable set of blocks, if the total size of the blocks is as large or larger than the new program.

If at any point in this dynamic memory search, a usable set of blocks is found that includes only one block occupied by a program, the system proceeds to make the swap if the program in memory is:

a. A background program,

b. Not scheduled,

c. No more than double the size of the new program, and

d. Not larger than 32k.

If such a set of blocks is found, it is immediately chosen for use.

As long as no set of blocks meeting these criteria is found, the search continues. In addition to the immediate-usage test, each set of blocks is examined in order to locate the best set of blocks (in case the immediate-usage test is not met).

For the purpose of selecting the best set, a set of blocks that does not include real-time programs is always better than a set occupied by real-time programs, regardless of the number of occupants or their size. This is the only way in which the memory management system treats real-time programs differently from background programs. A real-time program in the dynamic memory area may be swapped out for a lower priority program (real-time or background), but only if there is no way that room can be made in memory by swapping only background programs.

If there is more than one set of blocks containing no real-time programs, the third search algorithm compares these sets, giving them weight according to which set could be swapped in the least time. This depends on the number of occupants and their size. It takes more time to swap out several small programs than a single large program. To make its decisions, the system establishes a weight for each set of blocks. This weight is the product of the number of pages to be swapped by the number of programs to be swapped. The set of blocks with the lower weight is considered better.

Knowing the best set of blocks, the system now swaps out the first occupant and clears the memory-resident bit in its ID segment so the dispatcher does not try to run the swapped-out program.

The system swaps out the occupants of the set of blocks one at a time. Each of the remaining blocks in the set is examined after each swap to verify that the occupant to be swapped is still swappable. This swappable check covers the possibility that one of the occupants could have executed, entering a non-swappable state, while a previous occupant was being swapped.

Note that as programs are swapped out they have their memory-resident bits cleared, but they are not actually moved out of memory.

When all occupants have been swapped out, the memory area is released and a new partition for the new program is formed out of the resulting free block. The system then swaps-in the program from the swap file or copies it from its memory-image file on disk.

Restarts

When system status changes because of an asynchronous event and the set of blocks is not usable anymore, a cleanup must be performed. The system deletes all records of the previously performed search and starts a new search on the next entry.

Cleanup occurs when:

a. A program terminates, freeing a block of memory that is suitable for the newcomer.

b. When a program that has been swapped out and marked not to run changes state via some asynchronous event and becomes the highest priority program in the schedule list.

c. When a program to be loaded or swapped in is taken off the schedule list due to an operator command, an EXEC 6 call, or a similar terminating event.

d. When the swap file runs out of space in a multiple swap-out sequence.

Program Loading and Swapping

A program in memory is overlaid by another program when it terminates. It must be reloaded for execution again. To avoid reloading a program into memory before every run, it can be terminated saving resources or serially reusable.

When rescheduling a program that terminated saving resources, execution continues from the point at which it terminated. It can be swapped out if the need arises, then swapped back in when it is rescheduled. For CDS programs, the data partition can be swapped out and/or the code partition overlaid if the need arises. The data partition is swapped in and the code partition reloaded from the program file when the program is rescheduled.

If a program is dormant and has performed a serially-reusable termination, its partition can be overlaid by another program. This type of termination should be used only with disk-resident programs that can initialize their own buffers or storage locations. After a serially reusable termination, a program is reloaded from disk if its partition has been overlaid.

A program that terminates serially reusable starts execution at its primary entry point when rescheduled. If the need arises, it is overlaid without swapping out (either the code or the data partition for CDS programs) and a fresh copy of the program file is brought into memory when the program is rescheduled.

If the program terminates neither saving resources nor serially reusable, the system clears the memory-resident bit, resets the memory descriptor pointer in the ID segment, and releases the program partition. For CDS programs, the system clears the memory-resident bits for both the code and data partitions, resets the memory descriptor pointers in the ID segment and releases the partitions the program is using. For shared CDS programs, the system clears the code partition only when the in-use count reaches zero. The program file then must be copied into memory before the program can execute again.

Programs in a memory-based system are booted into memory along with the operating system (merged via the program BUILD). These programs are not associated with the original load medium at run time (the disk LU in the ID segment is zero). These programs are not reloaded each time they execute nor can they be swapped to disk.

Program Partition Deadlock

Program partition deadlock is a situation where a program cannot be allocated all the dynamic memory partitions it needs because of the manner in which partitions are allocated. Programs that require more than one memory partition are those that use shareable EMA (one partition for the program and one or more for the shareable EMA), CDS programs (two partitions), and CDS programs using shareable EMA (three or more partitions). A deadlock occurs if the first partition is allocated in the middle of the free dynamic memory pool, leaving the remaining free areas too small for the remaining partition(s).

For example, suppose a program needs a 30-page and a 20-page partition and the 30-page partition is allocated near the middle of a 50-page free dynamic memory, leaving a 15-page and a 5-page free area. Both areas are then too small for the 20-page partition needed by the program, and thus causing a program partition deadlock.

An algorithm to prevent such a deadlock is embedded in the dynamic memory searches. Checks are made whenever a possible position for a partition is considered, whether it is a block of free memory or a set of usable block containing swappable and overlayable programs. The possible positions for the remaining partitions needed by the program are determined, and the memory searches verify that the partitions do indeed fit into those positions. It is important to understand that this check deals only with "good" dynamic memory, not the bad pages downed due to parity error. The current use of the memory (for example, shareable EMA, program, and free) is not relevant. If the memory is in use for shareable EMA, the deadlock check passes (if the needed partition fits), but the memory searches are not able to allocate the partition since shareable EMA is always locked into memory until all programs that are accessing it terminate. The deadlock simply assures that the needed partitions fit into memory, regardless of the current status of the memory.

When a deadlock is detected, the program is aborted and an error (SC09) displayed. If this occurs, reduce your program size.

4

System Boot-Up

The normal boot-up sequence for disk-based systems is as follows:

1. The Virtual Control Panel (VCP), a program in the VCP/Loader ROMs (on the processor card), reads the first file on the disk into memory. This file should contain a BOOTEX program that has been correctly initialized by the INSTL program.

   The VCP uses your boot command string to decide where to find BOOTEX. For example, the string %BDC2027 specifies HP-IB address 2, disk surface 0, select code 27 (2027 is the default string.) BOOTEX must be the first file, because the VCP does not know anything about the file system.

   INSTL is used to set up some information in the BOOTEX file describing the disk from which BOOTEX was loaded. INSTL can also copy BOOTEX to the right place.

2. BOOTEX reads a file on the disk. This file can either be a type 4 ASCII command file or a type 1 system file. The name of the file may be specified via a command issued to the VCP. For example, %BDC2027 selects file /SYSTEM/BOOT.CMD, which is the default command file name. If the bootable LU is a FMGR LU, the default file is SYSTEM. If the file is a type 4 command file, BOOTEX executes the sequence of commands included in the command file, then brings the desired operating system into memory and starts it up. If the specified file is a type 1 system file, BOOTEX brings the operating system into memory and starts it up.

   Note that it is possible to have different command files on the disk to bring up different versions of the operating system, or to initialize the system in different ways. Since BOOTEX can access different command files, it is easy to keep several different system files around.

3. The operating system dispatches a user-specified start-up program to perform system initialization. This program may start the execution of any other desired programs.

BOOTEX Functions

The BOOTEX program may perform the following functions, as directed by the command file:

1. Mount any disk LUs that are needed for system initialization.
2. Initialize the swap file area. This involves creating the swap file if it does not exist, or finding it on the disk if it does.
3. Define certain system tables, such as the memory descriptor table.
4. Restore the ID segment used by certain programs, such as terminal handling programs (typically CI) and the directory manager program D.RTR.
5. Assign programs to reserved partitions.
6. Define reserved partitions.
7. Specify a system start-up program to execute immediately on boot completion. For disk-based systems, the normal start-up program is CI. CI can be scheduled on boot-up and directed to execute the commands in a file.

Start-Up Program Functions

When initializing your system, you may want your start-up program to perform some of the following functions:

1. Initialize any subsystems; for example, networking.
2. Initialize any peripheral devices that must be set to a known state before application programs start executing, such as terminals connected via the multiplexer card.
3. Start-up any of your application programs by using CI XQ or RU commands, or EXEC program scheduling requests from a user-written start-up program.
4. Write a message to the attached terminals, to inform users that system initialization is complete.

5

Operating System Modules

Introduction

This chapter contains a description of all the RTE-A Operating System modules. The only modules that are required for an operational system are VCTR, EXEC, RTIOA, IOMOD, ABORT, IORQ, MAPS, PROGS, SAM, UTIL, and the correct RPL module plus one I/O driver for a network or terminal interface to the system. The remaining system modules are optional; each adds additional capabilities to the system.

To aid in understanding the consequences of including or omitting a given module, Table 5-1 shows a summary of EXEC requests and the modules in which they are processed. If the necessary module is omitted, the EXEC request is not available for this operating system, and a a call to it causes an error.

To implement the modularity described above, a library of dummy modules is used. This library, $SYSA, contains routines that satisfy the references made to the omitted modules. These routines also handle the request being made to the module.

In some cases, the dummy module issues an error message indicating that a certain command does not exist. In other cases, it may replace the call to the routine with a NOP (no operation) instruction or it may even abort the program that made the illegal request and continue normal system operation.

$IDRPL

$IDRPL is a partitionable system module. $IDRPL contains the code to:

- get ID segment addresses of programs
- create program ID segments
- set up and search the proto ID list in XSAM
- create and remove proto IDs
- duplicate program ID segments from program ID segments and proto IDs.

If $IDRPL is not specified, a dummy module ($ID..) from the dummy library ($SYSA) is automatically included. It only gets the ID segment address of a program. If a program requests any other functions, it causes an error message.

Table 5-1. EXEC Requests and Processor Modules

EXEC Code	Meaning	Processor Module
1	I/O Read	IORQ
2	I/O Write	IORQ
3	I/O Control	IORQ
4	– not available –	
5	– not available –	
6	Program termination	ABORT
7	Program suspend	ABORT
8	load program segment	LOAD
9	Schedule with wait	SCHED
10	Schedule program	SCHED
11	Request system time	TIME
12	Time schedule	TIME
13	I/O status	IORQ
14	String passage	STRN
15	– not available –	
16	– not available –	
17	Class read	CLASS
18	Class write	CLASS
19	Class control	CLASS
20	Class write/read	CLASS
21	Class get	CLASS
22	Define Swappability	MEMRY
23	Queue schedule w/wait	SCHED
24	Queue schedule w/o wait	SCHED
25	– not available –	
26	Memory Status	STAT
27	– not available –	
28 *	– not available –	
29 *	Locate program	UTIL
30 *	Lock LU	LOCK
31 *	Allocate class number	CLASS
32	Lock resource number	LOCK
33 **	Determine effective LU	STAT
34	#NQUE and #RQUE	DSQ
35	Interface driver entry	RTIOA
37	Signals	SIGNL
38	Timers	ALARM
39	Environment Block Access	ENVRN

* indicates EXEC codes used only by library subroutines:

29 - IDGET
30 - LURQ
31 - CLRQ
32 - RNRQ

** indicates EXEC code used only by DS software.

$SYSA

This is the dummy library to be searched to resolve any references to modules that have been omitted from the system relocation. The dummy modules use the first three characters of the name of the real module followed by two periods. For example, the dummy module TIM.. takes the place of the module TIME.

To distinguish between the real modules and the dummy modules, system entry points called module flags are used. In each module, the module flag $.xxx (where xxx represents the first three characters of the real module name) is set to zero in the real module and to −1 in the dummy module. For example, to test whether the real CLASS module or the dummy CLASS module is included in the system, the following sequence of instructions could be used:

```
XLA1 $.CLA      Load A-Register with $.CLA
SZA
JMP DUM         Here if the dummy CLASS module is available
JMP REAL        Here if the real CLASS module is available
.
.
```

These modules may also be differentiated at link time. In each of the real modules, the module flag $$xxx exists. (Again, xxx are the first three characters in the name of the real module.) This flag does not exist in each of the dummy modules. In order to use this feature, simply reference this module flag if your program must use the real module. LINK finds an undefined external reference if the dummy module is included in the system and does not link the program.

Below is a description of the system modules with the capabilities provided by each one. If a module is omitted, and the dummy module is used, the corresponding capabilities are not included in the system.

ABORT

ABORT is a required system module. ABORT contains the code for handling EXEC 6 (stop program execution) and EXEC 7 (program suspend) requests, as well as program abort processing. ABORT is a partitionable module.

ALARM

ALARM is a partitionable module that is required when using interval timers. ALARM contains all the routines necessary to set, query, and modify interval timers.

If ALARM is not specified, a dummy module (ALA..) from the dummy system library ($SYSA) is automatically included .

CDSFH

CDSFH is a partitionable module that is used only in systems with VC+. This module is required for use with CDS programs and must be included in system generation.

If CDSFH is not specified, a dummy module (CDS..) from the dummy system library ($SYSA) is automatically included. This is the case for all HP 92077A systems.

CHECK

CHECK is a partitionable module that is required when using SECURITY/1000.

If CHECK is not specified, a dummy module (CHE..) from the dummy system library ($SYSA) is automatically included.

CLASS

CLASS is a partitionable module that permits processing the class I/O calls associated with the following EXEC request calls:

17 = Class read
18 = Class write
19 = Class control
20 = Class write/read
21 = Class get
31 = Allocate class number (CLRQ)

The dummy CLASS module (CLA..) aborts any program that makes one of the above requests and causes an error message.

DSQ

DSQ is a partitionable module that must be relocated during system generation if the system includes NS-ARPA/1000 or ARPA/1000. DSQ is used to re-queue requests between the various DS monitor programs.

If the dummy DSQ module (DSQ..) is used in a system with the networking software listed above, an error is issued.

ENVRN

ENVRN is a partitionable module that is required for access to the Environment Variable Block via the EXEC(39, ...) interface.

A program that attempts to call EXEC(39, ...) when the dummy ENVRN module (ENV..) is in the system receives an OP39 EXEC error code.

If you are using an A900 processor, please refer to the "Modules for A900 with %ENVRN or Networking Products" section in this chapter for information on how to get ENVRN into your system.

ERLOG

ERLOG outputs error messages to the user terminal (the terminal LU number is stored in the program ID segment). ERLOG gets the System Language Number (SLN) from DVT24 and uses the number as an index to the Language Message Address table ($LMAT). The Language Message Address table contains the logical address of the system message block from which the error message is to be extracted. If the no-abort bit is not set when the program aborts, the program is aborted after the messages are issued.

The dummy ERLOG module (ERL..) does not output any message. Otherwise, it takes the same actions as the complete system module.

EXEC

EXEC is a required system module. All EXEC requests are routed through this module. The EXEC module also contains the code for the initial processing of most interrupts (TBG, MP, UIT, I/O). EXEC may, in turn, pass the requests to other modules.

EXEC also processes privileged mode operations (GOPRV, $LIBR, $LIBX, Dispatchlock, Dispatchunlock, $SJSx).

ID*43

ID*43 is called the power fail driver module. Unlike other drivers it does not handle an interface card. It provides the capability to save the state of the computer upon power failure and restore it upon recovery of power. Additional processing is provided through the optional AUTOR program if it is available. (AUTOR is not required for power fail recovery.)

AUTOR aborts any pending requests on terminal MUX LUs (MUX interface card with device type 0, 5, or 12) and sends a power failed message to all terminal LUs. AUTOR must be RP'd in the welcome file. The AUTOR program is supplied in source form, and can be modified by the user to provide customized power fail recovery.

The dummy module (POW..) causes the system to halt with the T-Register equal to a HLT 4.

IOMOD

IOMOD is a required system module. It contains several subroutines needed for I/O operations. These subroutines are used by the operating system as well as the I/O drivers.

IORQ

IORQ is a required system module. IORQ is also a partitionable module. All normal I/O requests are handled within this module. However, both IOMOD and RTIOA must be present to actually perform I/O. EXEC request calls processed by IORQ are:

1 = Read
2 = Write
3 = Control
13 = Status

The following EXEC request calls cause IORQ to transfer control to the optional CLASS module. The dummy CLASS module causes the program making these requests to abort.

17 = Class read
18 = Class write
19 = Class control
20 = Class write-read

The class get request (EXEC 21) is also processed in the CLASS module but is routed to the module directly from EXEC.

LOAD

This module enables a program to be loaded from a disk, and handles program swapping. LOAD is a partitionable module.

LOAD processes EXEC 8 (overlay load) requests.

The dummy LOAD module (LOA..) causes an error message and aborts any program that makes the above requests. Program swapping does not occur.

LOCK

LOCK is a partitionable module that handles LU and resource number locks. It is called by subroutines LURQ and RNRQ. Also, when a program aborts, LOCK is called to release any locally allocated LU locks and/or resource numbers.

The dummy LOCK module (LOC..) treats any LU lock request or resource number lock as an error, and treats any request to clean up LU or resource number locks as an NOP (no operation).

MAPOS

This module contains routines that control access to partitioned system modules. MAPOS is required only in systems that have system modules residing in OS/Driver partitions. It is not required or needed in any other systems.

MAPOS is not a partitionable module; it must be relocated during the system relocation phase of the generator.

There is no dummy MAPOS module because systems without system modules in the OS/Driver partition never reference any MAPOS routines.

MAPS

MAPS is a required system module. MAPS contains the code dealing with the dynamic mapping system for programs, OS/Driver partitions, and I/O.

MEMRY

This module searches for memory to run a program, allocates partitions, manages the internal Tables that describe memory, and processes the EXEC 22 call that changes the status of a partition. MEMRY is a partitionable module.

The dummy MEMRY module (MEM..) contains minimal functionality to allow a memory-based system (a system created by BUILD where all programs are memory-resident) to execute properly. An EXEC 22 request does not have any effect.

MSGTB

The MSGTB module contains pointers to the messages printed by RTE-A in response to operator commands such as SZ, or in response to some error. This module is used with the system message blocks that contain the actual message text.

If the dummy MSGTB module (MSG..) is used, no messages are printed. Be aware that this can cause confusion because errors are not reported.

OPMSG

This module contains operator messages required by the modules SYCOM, XCMND, and STAT.

The dummy OPMSG module (OPM..) returns a "??" if any of the operator messages is called for.

PERR

PERR is automatically invoked when a memory parity error is detected. It determines whether or not the parity error is reproducible. It marks the bad memory pages as unusable (down) when necessary and prints a message on the system console. PERR is a partitionable module.

If a dummy module (PER..) is used and a parity error occurs, the system halts with the T-Register equal to a HLT 5 error.

PROGS

PROGS is a required system module. PROGS contains the code that deals with scheduling programs and program state changes.

RTIOA

RTIOA is a required system module. RTIOA contains the code for the logical and physical drivers that provide the interface to device and interface drivers. RTIOA also contains XSIO, the operating system I/O request processor.

SAM

SAM is a required system module. This module allocates and deallocates portions of system available memory (SAM) and extended system available memory (XSAM).

SCHED

This module enables a program to schedule another program. The following request codes are processed:

9 = Schedule with wait
10 = Schedule without wait
23 = Queue schedule with wait
24 = Queue schedule without wait

The dummy SCHED module (SCH..) causes an error message and aborts the program making the above request unless the no-abort bit is set.

SECOS

SECOS is a partitionable module that is required when using SECURITY/1000.

If SECOS is not specified, a dummy module (SEC..) from the dummy system library ($SYSA) is automatically included.

SIGNL

SIGNL is a partitionable module that is required when using signals. SIGNL contains all the routines necessary for delivering, receiving, and controlling signals.

If SIGNL is not specified, a dummy module (SIG..) from the dummy system library ($SYSA) is automatically included.

SPOOL

SPOOL is used only in systems with VC+; it contains subroutines that are used by the spool system programs. There are no user-callable subroutines or EXEC request processors in SPOOL.

During system generation for a system with VC+, SPOOL is included. For a system without VC+, a dummy module, “SPO..” is included in $SYSA.

STAT

This module provides an extension of the operator commands found in SYCOM. STAT is a partitionable module.

The commands provided by this module are:

DS = Device status
PS = Program status

This module also processes the memory status request and effective LU request:

EXEC 26 = Memory status (LIMEM)
EXEC 33 = Effective LU

The dummy STAT module (STA..) prints ILLEGAL COMMAND if this command is attempted.

STRNG

STRNG permits the runstring to be saved in SAM (program writing to the scheduling program) and a runstring to be picked up from a scheduling program or from the operator. STRNG processes the EXEC 14 request code.

If the dummy STRNG module (STR..) is used, then an EXEC call to pick up the runstring produces the following values in the registers upon return to the program:

A = 1 (No runstring)
B = 0 (Size of string)

An EXEC call to write a string to a scheduling program with the dummy STRNG produces a SC10 error (insufficient memory).

SYCOM

SYCOM parses all system level operator commands and processes them by passing the command parameters to the appropriate routine. In addition to these command processing functions, this module executes the following operator commands:

EX = No operation
OF = Program termination
UP = Set device up
RU = Run program
XQ = Execute program

The dummy SYCOM module (SYC..) does not read operator input and ignores MESSS calls.

TIME

TIME is a partitionable module that allows a program to request the system time and to time schedule another program or itself. TIME processes the following EXEC calls:

11 = Read time
12 = Time schedule a program

The dummy TIME module (TIM..) treats request EXEC 11 as a NOP (no operation), allowing the program to continue with its parameters unchanged. However, it causes an error message for EXEC 12 and aborts the program making the request. The system time may not be changed or displayed; all requests are ignored.

UTIL

UTIL is a required system module. UTIL contains the startup code, many system variables, unpartitionable parts of partitioned modules, and utility routines used in other parts of the operating system.

VCTR

VCTR is a required system module. VCTR contains many system entry points (for example, constants, variables, and arrays). This module must be relocated first in system generation.

When a program is linked and all system references are resolved in the VCTR module, the program's ID segment is modified at link time to indicate that this program is transportable.

VEMA

This module provides the capability to perform VMAIO system calls. It must be included in any system that may execute a program that makes VMAIO calls. VEMA is a partitionable module. Some programs that utilize VMAIO are D.RTR, TF, ASAVE, and FST.

If the dummy VEMA module (VEM..) is used, a program issuing a VMAIO call is aborted with a VM92 (VMA not available) error.

XCMND

This module provides an extension to the operator commands found in the SYCOM module. XCMND is a partitionable module.

The commands provided by this module are:

AS = Assign program to a reserved partition
BR = Set break flag in program's ID segment
CD = Display or change code partition size of CDS program
DN = Set device down
DT = Display or change data partition size of CDS program
GO = Resume program
PR = Change program priority
SS = Suspend program
SZ = Display or change program size
UL = Unlock a shareable EMA partition
VS = Display or change program VMA size
WS = Display or change program working-set size

If the dummy XCMND module (XCM..) is used, the message ILLEGAL COMMAND is issued if any of the above commands are attempted.

RPL Modules

The RPL module contains the special microcoded instruction formats. One RPL module is required in the RTE-A Operating System. You should choose one from the list shown in Table 5-2, based on the type of processor and features used in your system.

Table 5-2. RTE-A RPL Files

Processor	CDS	Double Precision Floating Point	RPL File
A400	no	no	%RPL40
	no	yes	%RPL41
	yes	no	%RPL42
	yes	yes	%RPL43
A600	no	no	%RPL60
A600+	no	yes	%RPL61
	yes	yes	%RPL63
A700	no	no	%RPL70
	no	yes (HWFP)*	%RPL71
	yes	no	%RPL72
	yes	yes (HWFP)*	%RPI73
A900	no	yes	%RPL90
	yes	yes	%RPL91
			Refer to the following paragraph for more information.
A990	no	yes	RPL_A990.REL
	yes	yes	RPL_A990_CDS.REL

* HWFP – Hardware Floating Point Card must be installed in the system.

Modules for A900 with %ENVRN or Networking Products

If you are using the %ENVRN module or any of networking products NS-ARPA/1000, ARPA/1000, or X.25 on an A900 processor, then you will need to relocate additional modules depending on the revision code of the firmware installed on the processor. If your firmware is Rev. 4 or later, then the firmware equivalents may be used by relocating the following module:

/RTE_A/RPL_A900_REV4.REL

For earlier firmware revisions, the software equivalent modules must be used. These are found in the file:

/RTE_A/XMB.REL

XMB.REL contains the software for six cross-map move byte instructions that are not in the A900 RPL files %RPL90 or %RPL91. These instructions are listed below next to the modules which call them:

ENVRN −− .mb01, .mb10, .mb12, .mb21
IDZ00 −− .mb02, .mb12
DDX00 −− .mb01, .mb02, .mb10, .mb12

(.MB20 is also included in XMB.REL but it is not required by the operating system.)

All six routines may be relocated by relocating XMB.REL. However, memory space may be conserved by relocating only those that are necessary for your system. For example, the following command will relocate only the .MB01 software from the MB01 module:

RE,/RTE_A/XMB.REL,MB01

If the proper modules are not relocated, an undefined external error is displayed. Descriptions of the mbxx cross-map move byte instructions are given in your HP 1000 A-Series Computer Reference Manual.

Optional Modules

The RTE-A Operating System has optional modules that can be specified when the system is generated. Some of these optional modules are dependent on other optional modules. All are dependent on the required modules. The chart below lists both the required and optional modules that can be used to create different configurations of RTE-A.

If any of the modules are omitted, the dummy library, $SYSA, must be searched. Also, if any system modules are relocated in the Operating System Driver Partition Area, $SYSA must be searched after the non-partitioned modules have been relocated and before relocating the partitioned modules. If no system modules are being relocated in the Operating System Driver Partition Area, $SYSA should be searched before beginning driver partition relocation.

```
REQUIRED SYSTEM MODULES: ABORT, EXEC, IOMOD, IORQ, MAPS, PROGS, RPLxx,
                         RTIOA, SAM,  UTIL, VCTR, $SYSA, driver.
OPTIONAL SYSTEM MODULES: $IDRPL, ALARM, CDSFH, CHECK, CLASS, DSQ, ENVRN,
                         ERLOG, ID.43, LOAD, LOCK, MAPOS, MEMRY, MSGTB,
                         OPMSG, PERR, SCHED, SECOS, SIGNL, SPOOL, STAT,
                         STRNG, SYCOM, TIME, VEMA, XCMND.
```

The following is a small sample of possible RTE-A configurations that can be created by adding different optional modules. The list shows four systems with increasing functionality.

```
Minisys   : A non-CDS memory based system that executes
            a start up program.
Smallsys  : A non-CDS system with the ability to load,
            swap and schedule programs.
Medsys    : A CDS based system with ability to load, swap
            and schedule programs.  Also includes
            modules necessary for use of the optional
            DS link software.
Largesys  : A full featured system with all modules
            included.
```

To generate these systems, different optional modules are used. The following chart shows which modules can be used to create the configurations listed. Note that all systems must include the required modules.

	* RTE-A OPTIONAL MODULES *		
MINSYS	SMALLSYS	MEDSYS	LARGESYS
$IDRPL	X	X	X
CDSFH		X	X
CHECK			X
CLASS	X	X	X
DSQ		X	X
ENVRN		X	X
ERLOG	X	X	X
ID*43			X
LOAD	X	X	X
LOCK		X	X
MAPOS	X	X	X
MEMRY	X	X	X
MSGTB	X	X	X
OPMSG		X	X
PERR			X
SIGNL	X	X	X
SCHED	X	X	X
SECOS			X
SPOOL		X	X
STAT	X	X	X
STRNG	X	X	X
SYCOM	X	X	X
TIME		X	X
VEMA		X	X
XCMND	X	X	X
ALARM	X	X	X

Partitionable Modules

Partitionable system modules allow RTE-A to logically address more than 32K words (32 pages) of RTE-A code and/or data. The system modules to be placed in an OS/Driver partition are defined when the system is generated.

Not all system modules can be placed in an OS/Driver partition; some system modules contain routines that must reside in logical memory at all times. These modules are non-partitionable and the generator reports an error if one is relocated into the OS/Driver partition area. The system module descriptions in this chapter indicate which modules are partitionable; see the System Generation and Installation Manual for further information.

Partitioned system modules and I/O drivers are mapped into the logical address space reserved for the OS/Driver partition as the modules and drivers are needed. Reducing the amount of logical address space required for system modules and I/O drivers allows more system tables to be generated. The additional table space increases the overhead involved in manipulating the dynamic memory maps; therefore, partitioning more code and/or data increases the amount of logical address space available, but degrades system performance.

A system in which many system modules are partitioned can contain more ID segments, IFTs, and DVTs than an unpartitioned system. For example, if a large number of programs (ID segments) and/or devices (IFTs and DVTs) are required, all partitionable drivers and system modules can be placed in OS/Driver partitions. This system supports the devices or programs, but its performance is degraded. If few devices and/or programs are required, a system with minimal driver and system module partitioning can be generated and system performance is not affected.

The OS/Driver Partition

The OS/Driver partition is a contiguous set of pages in the system logical memory map. The size of this partition is the same as the size of the largest OS/Driver partition that is defined at generation time. When a partitioned system module or driver is called, the system maps the appropriate OS/Driver partition into this logical address space.

A system module located in an OS/Driver partition is called a partitioned module. Routines contained in the module are called partitioned routines. Partitionable system modules contain special information about the routines in them that is used by the generator when it puts the module in an OS/Driver partition.

Some system modules contain routines that must reside in memory at all times. These modules are non-partitionable and the generator reports an error if one is relocated into the OS/Driver partition area. Specific information about partitionable and non-partitionable system modules can be found in the *RTE-A System Generation and Installation Manual.*

Tags

An interface routine, called a tag, is used to enter a partitioned system routine. The tag saves information about the currently mapped OS/Driver partition, copies any parameters into the tag, alters the physical mapping registers to point to the OS/Driver partition that contains the called system routine, and transfers to the routine. The routine then retrieves any parameters from the tag, not from the original routine call.

The tags are created by the generator and are located in the tag area, a group of contiguous words in the non-partitioned portion of system memory. The generator automatically changes calls to partitioned system routines to enter the tag associated with the routine.

6

System Symbols and List Structures

The RTE-A software system depends upon various tables and lists (and sometimes lists of tables) to perform memory and program management tasks. The location of most of these tables and lists can be found from the symbols reported in the generation map or from the snapshot file produced by the generator. (Appendix A contains a description of the snapshot file format.)

A table is a group of contiguous memory locations that contain related information. For example, the swap descriptor table describes the location of free areas of the swap file; this table is described in Chapter 11 of this manual. Most tables consist of multiple entries, where each entry is a table itself. For example, the Interface Table (IFT) consists of entries of about nine words each. Associated with each table are several entry points that contain information about the table: the starting address, the number of entries, and the number of words (memory location) in each entry.

An entry point is a symbol associated with a certain memory location. Entry points are used for holding information (for example, the number of entries in a table, system checksum value, and current time), holding the address of a subroutine, and so forth. When the system is generated, the location of each entry point and its symbol are placed in the snapshot file.

System Symbols

Most symbols belonging to the system start with the dollar sign character ($). These entry points may refer to pointers, counters, buffers, subroutine entry points (called via JSB), or code entry points (called via JMP). Only special instructions are capable of examining or modifying system data structures or executing system code from the user area. As additional protection against accidental modification of the operating system, those instructions that store into the system or jump into system code may only be executed with memory protect disabled. The mechanism for temporarily disabling memory protect is described in Chapter 8.

System pointers are entry points that contain the address of a table or other data structure. For example, the system entry point $IDA contains the address of the first ID segment, and is therefore a pointer. This is illustrated in Figure 6-1.

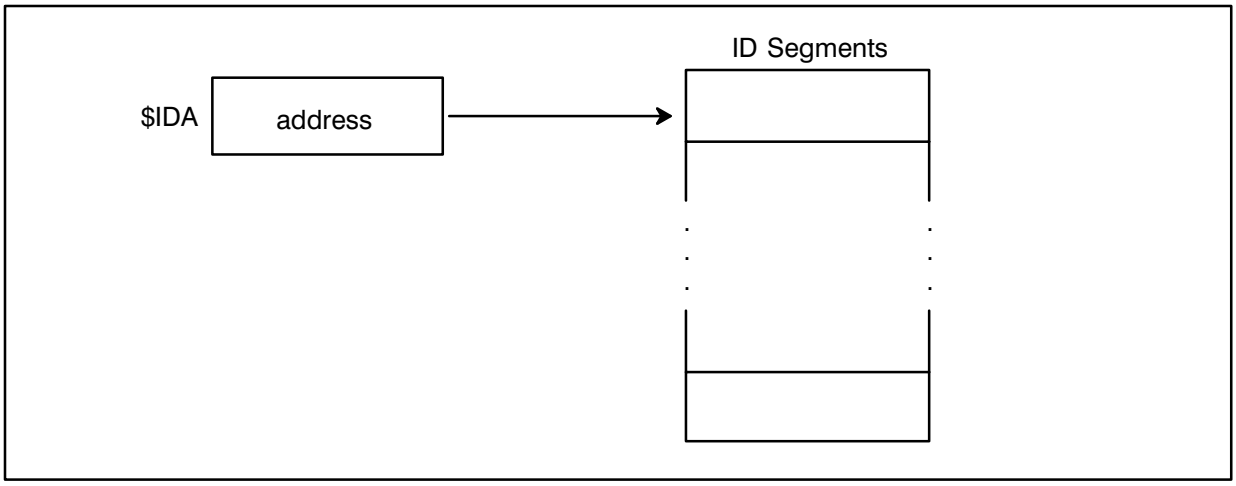


Figure 6-1. System Pointer $IDA

For example, if the generation listing shows the entry

$IDA 2066

and the contents of location 2066B is 34402B, then the first ID segment starts at location 34402B.

Counters are used to keep track of such things as the number of elements in a table or the length of the table. These are used in conjunction with pointers and addresses in order to access specific elements of specific tables.

System subroutine and code entry points are used by the operating system to jump between modules.

The operating system also contains a few fixed-length buffers and Tables that are not defined by the generator, such as the system command parsing buffer $PB.

The system also contains five user-definable entry points, $CSTM1 through $CSTM5. These variables are initially zero and are not used by the system. They are defined in VCTR so that any program that references them may be transportable. These entries may be used by customers' drivers or applications.

Lists

In the operating system, the following forms of linked lists are used: linear linked lists, circular linked lists, linked lists using offset pointers, linear doubly linked lists, and circular doubly linked lists. The five forms of linked lists are described in the following paragraphs.

Linear Linked Lists

A linear list generally has a list head. This is a word in memory that is a pointer to the first element in the list. Then each of the elements, which may be tables or lists themselves, contains pointers that reference the next element in the list. The last element in the list contains a flag to show that it is the last element. This may be a zero or a negative one or some other value that is not a legal address.

One such list in the operating system is the scheduled list. This list is shown below in Figure 6-2. It has a list head, $SC, that points to the first word of the ID segment for the first program in the scheduled list. This word in the ID segment is the link word that contains the address of the ID segment of the next program to be scheduled and so on. The last ID segment in the list has a 0 in its link word to indicate that the end of the list has been reached. For example, the scheduled list in Figure 6-2 includes PROGA and PROGB, but not PROGC.

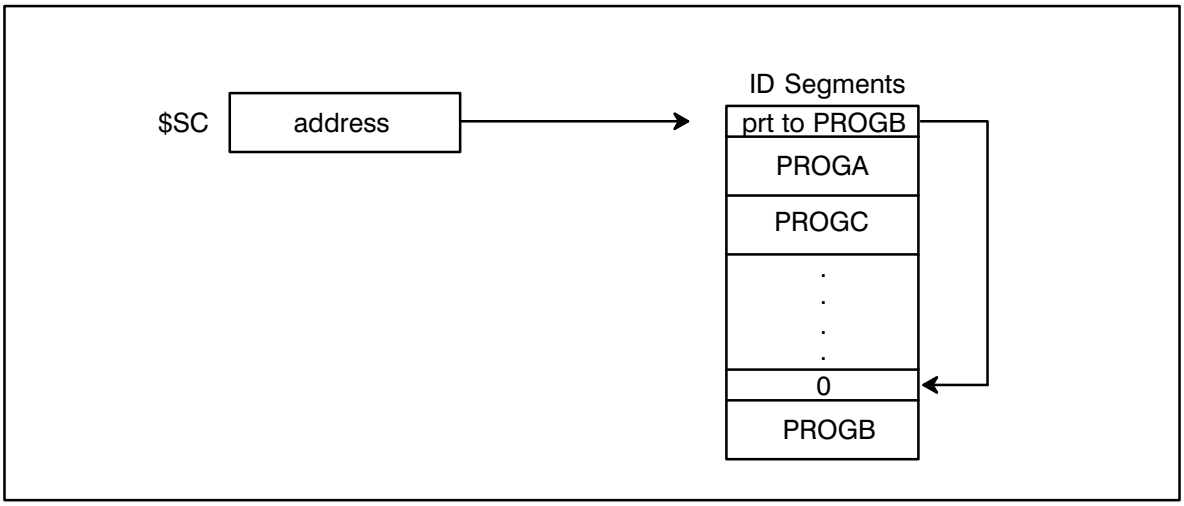


Figure 6-2. Example of Linear Linked List

Circular Linked Lists

In the circular linked lists are elements each containing a pointer to the next element in the list just as with the linear list. The last element points back to the first element of the list. This allows all of the elements to be accessed in turn. A sample circular linked list is shown in Figure 6-3.

The node list is one example of a circular linked list. This is a list that links together the Device Tables (DVT) for devices that must be accessed serially, such as the LUs of an HP 7908 disk drive. Each disk LU on the HP 7908 drive must be accessed serially.

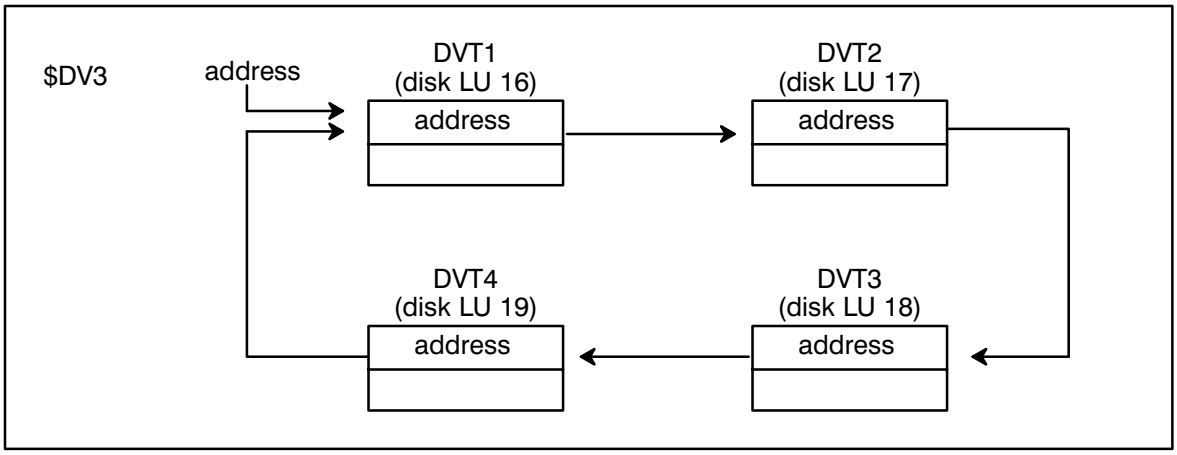


Figure 6-3. Example of Circular Linked List

Lists with Offset Pointers

Generally, linear and circular lists are structured so that the pointer (or link word) for a given element contains the address of the pointer of the next element in the list. This allows the list to be traversed quite easily because the link word of any element points to the link word for the next element. In some cases, the pointers for the list do not point at the link word of the next element. In these cases, the link word is offset from the referenced word by a specific number of words. This form of linking allows easy access to the particular word in the element referenced. An example of this type of linked list is the circular DVT list. This list contains, as elements, all of the DVTs connected to a particular IFT. The links are shown in Figure 6-4.

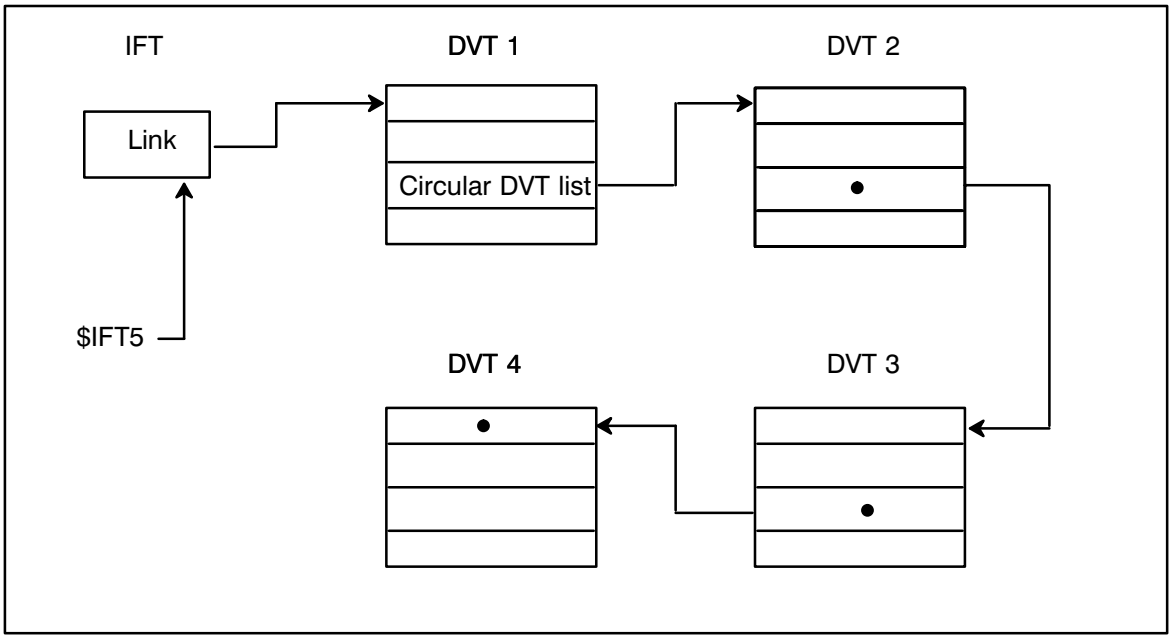


Figure 6-4. Example of List with Offset Pointer

Linear Doubly Linked Lists

A linear doubly linked list is useful because elements from the list can be inserted or deleted easily. The memory descriptor adjacency list is an example of a linear doubly linked list, shown in Figure 6-5 with list head $MEM.

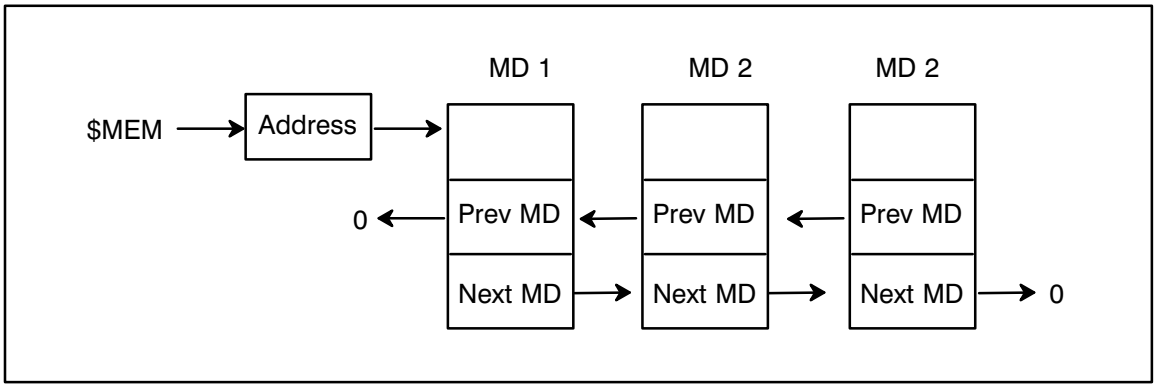


Figure 6-5. Example of Linear Doubly Linked List

Circular Doubly Linked Lists

The circular doubly linked lists allow list searches to begin anywhere in the list. The Memory Descriptor (MD) free list is an example of a circular doubly linked list, shown in Figure 6-6. The list head is $FREM.

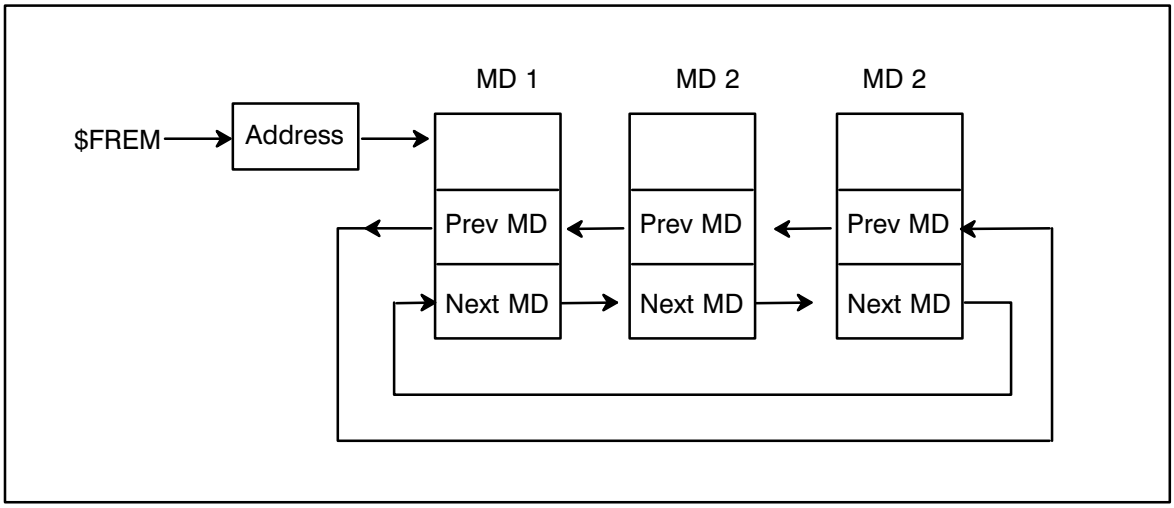


Figure 6-6. Example of Circular Doubly Linked List

7

I/O Drivers

A user program performs I/O requests (read, write or control) by issuing EXEC requests to the system with the proper request code. In the case of FORTRAN programs, this is transparent to the program; READ and WRITE statements are implemented by EXEC requests made by the FORTRAN formatter subroutine. (This routine is appended to the user program when it is relocated.) These requests are handled by special routines, called drivers, that convert the programmatic request into a form usable by the device.

There is a driver for each device and interface card. Device drivers are routines that process requests for specific devices, converting the request to a form suitable to that device. This request is passed along to an interface driver, which communicates the request to the appropriate interface card and thereby to the device. Simple devices (such as some HP-IB devices) may not require a device driver, in which case the interface driver handles the request without a device driver.

The path from request to action is illustrated in Figure 7-1.

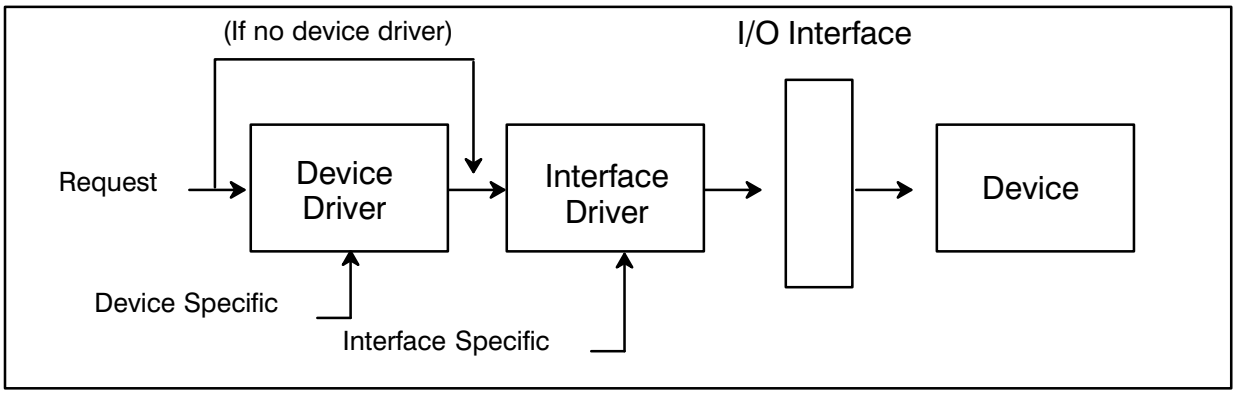


Figure 7-1. I/O Request Path

Associated with these drivers are various tables, which are explained more fully in Chapter 11. These tables are used to handle the requests in the manner described below.

Each request references a logical unit (LU). The system uses the LU Table (LUT) to map each logical unit to the Device Table (DVT) for the requested device. A control block, containing the request code, and other information needed by the driver, is then built and linked to the DVT. Several control blocks may be linked to a single DVT.

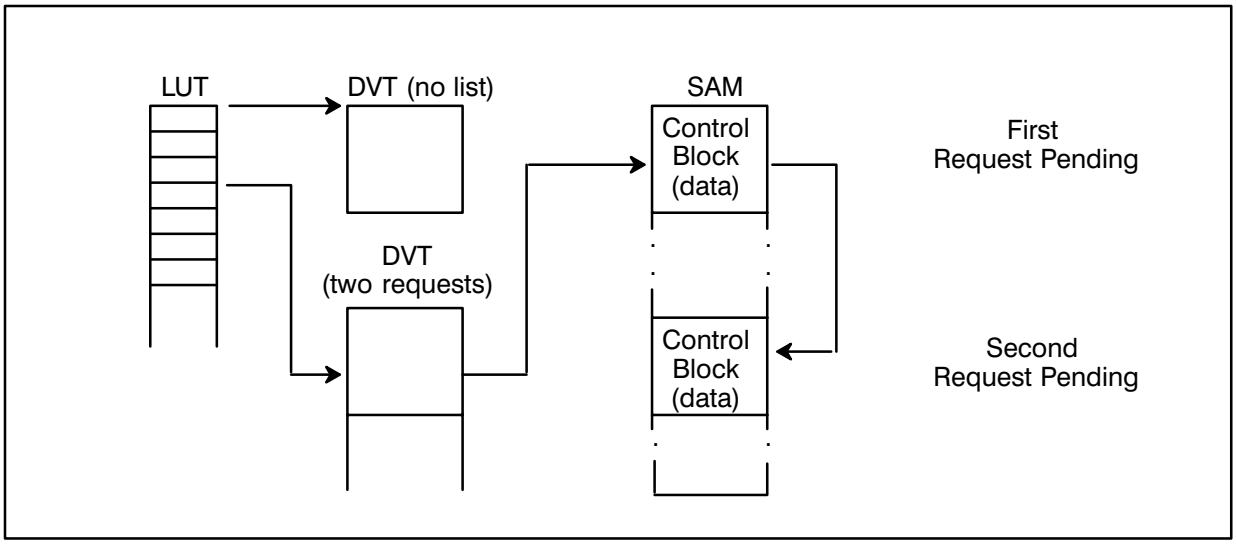


Figure 7-2. Buffered Request Example

If the request is buffered, the control block includes the user buffer and is built within SAM. To control the amount of SAM used by any device, the upper buffer limit is used as an upper bound on the total size of all the buffers linked on any one DVT at any one time. If the request is not buffered, the program ID segment contains the control block. Figure 7-2 illustrates a request to a buffered device, where two control blocks are shown linked to one DVT:

As requests are completed, the I/O request queue is relinked to skip the control blocks for the completed requests. When a request reaches the list head, the necessary information is copied from the control block to the DVT. The control block itself remains linked to the DVT until the request completes.

Each interface card also has an Interface Table (IFT) and an interface driver. The IFT is linked to all the DVTs for devices interfacing via this card. All the DVTs on each IFT are in a circular DVT list.

The relationship of the control block to the DVT and IFT, and the DVT to the IFT for pending requests is shown in Figure 7-3. In this figure, since all DVTs shown are linked to the same IFT, they also appear in a circular DVT list that is not shown.

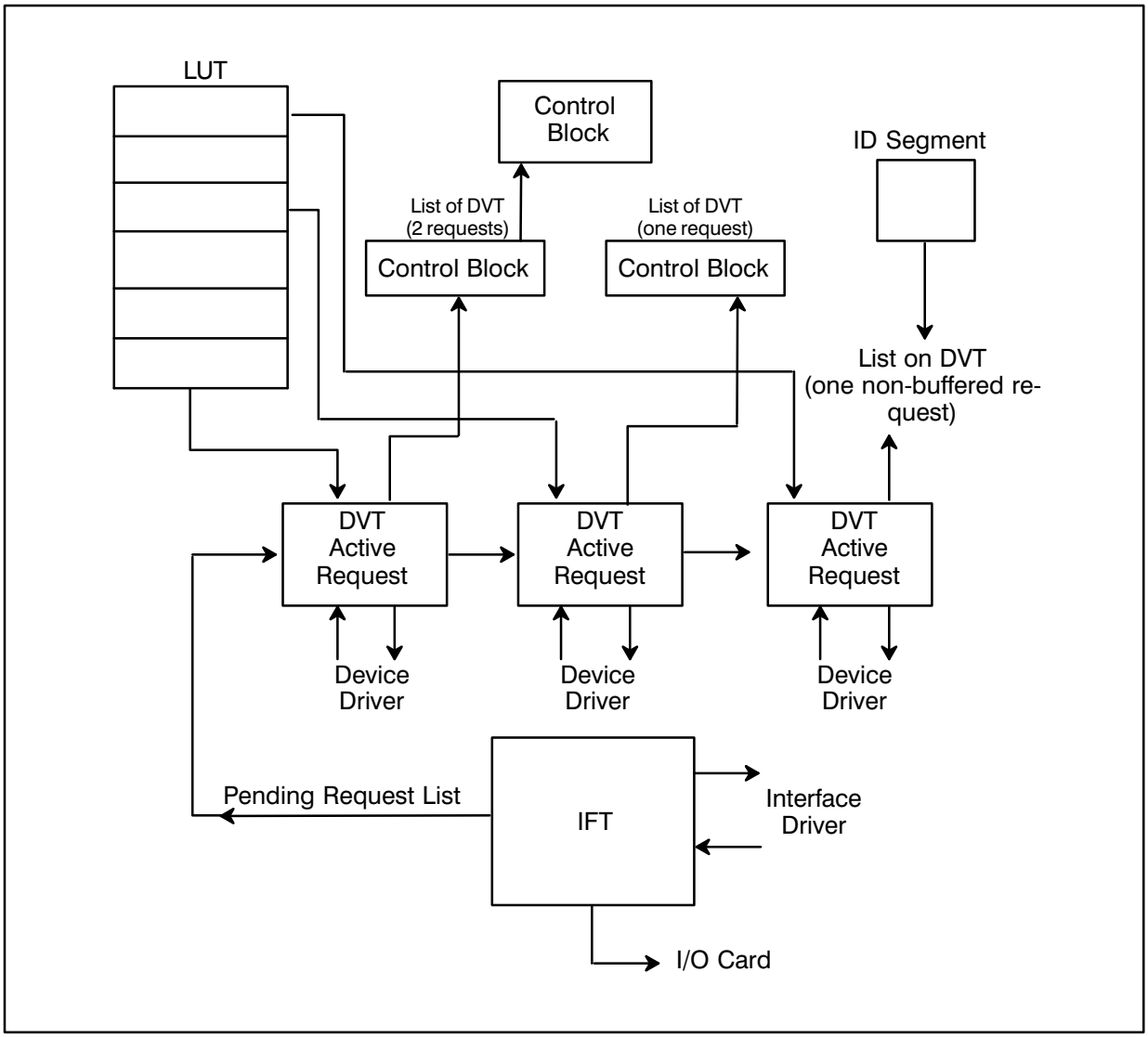


Figure 7-3. Request Lists on DVT and IFT

8

System Common/Shared Subroutines

System Common

Common refers to a means of sharing data or, in some cases, subroutines. Various types of common are used in FORTRAN and Macro programs. FORTRAN and Macro users are familiar with labeled common, which comes from using named common in FORTRAN, or EXT, ENT and ALLOC statements in Macro. Blank common is unlabeled common; this is also available to Macro users, though it is used less frequently. Common can be grouped into two major categories, local and system; each category may have labeled and blank commons. When common is used to share information between subroutines in a program it is called local common; when it is used to share information between programs it is called system common.

Local common is not dealt with here; it is discussed in the FORTRAN and Macro manuals.

Blank system common is an area set up during system generation. When you link a program, you can make any blank common used in the program refer to the blank system common. This is a simple way to share data when there is only a small number of programs involved. However, there are problems if two conflicting sets of programs want to share data in this way.

Labeled system common is more flexible. At system generation time, a set of code modules is placed in labeled common. This is usually just a set of FORTRAN named common blocks, or a module containing Macro storage areas with entry points. The entry points and common blocks found in labeled system common areas are put into the snapshot file. LINK uses these entry points and common blocks when you link a program and specify the LC option. The LC option allows access to labeled system common. Several sets of programs can use system common in this way, as long as the names used do not conflict.

Whenever a program uses any blank or labeled system common, all of system common is made available to it. This can be inconvenient, as it reduces logical address space and program size. Several HP products make use of labeled system common; for example, DS/1000-IV. Any applications that use blank or labeled system common bring in DS table areas if they were included in the system generation.

System common is most useful for sharing small amounts of data, as they decrease program size. These areas are best reserved for frequently used data that must be accessed very quickly. Access to areas in common is as fast as access to any normal variable.

When there is a large amount of data (more than several pages) to share, shareable EMA access is the preferred choice. Shareable EMA can be many, many pages (including one page reserved for the page table), and are easy to use from a high-level language; refer to the *RTE-A Programmer's Reference Manual* for details. Access to shareable EMA variables is typically 5 to 10 times longer

than access to local data (or common). Note however that nonbuffered and buffered I/O to and from EMA is allowed, and incurs no performance degradation. This makes shareable EMA suitable for use as a common data buffer area.

Synchronizing Programs

Whenever data is shared, the programs sharing the data must be written carefully to avoid race conditions when data is being read and written simultaneously by different programs. There are several techniques for avoiding race conditions; one of the best techniques uses resource numbers. Refer to the description of RNRQ in the *RTE-A Programmer's Reference Manual* for using resource numbers to control access to shared data.

The RTE-A Operating System provides several other mechanisms for controlling process execution. These are the system subroutines $LIBR, $LIBX, .ZPRV, and .ZRNT, and so are most suitable for use from Macro subroutines.

Subroutine $LIBR disables the interrupt system, allowing the calling program full access to the operating system except to make EXEC calls, including processes that can cause violations. The interrupt system remains off until the program calls $LIBX.

Subroutine .ZPRV must be in labeled system common; it prevents any other program from executing until the calling program calls $LIBX. This is a way to prevent race conditions, because no other program can run until the calling program is done. Subroutine .ZPRV leaves interrupts on; protection violations abort the calling program.

Subroutine .ZRNT is useful only when executable subroutines are placed in labeled system common. It ensures mutual exclusion of access to the subroutine, so only one program uses the subroutine at any time. Placing subroutines in system common is NOT recommended, as it makes the application harder to understand. The only benefit of placing subroutines in labeled system common is a small amount of physical memory savings, which is probably not worth the trouble this technique can cause. Also, a .ZRNT subroutine placed in system common cannot be called by a PCAL instruction (by CDS code), only via a JSB instruction (by non-CDS code). An interface subroutine in non-CDS code can be used.

Generating System Common

Labeled system common is produced by relocating a module (or several modules) at the appropriate point in the generation process. Access to labeled system common is by entry points declared in the module. Any module type may be relocated in labeled system common. The determination of whether program or system common is used by the program is made at when the program is linked.

For example:

```
MACRO
    NAM DATA,30 THIS MODULE GOES INTO LABELED COMMON
    ENT A1,A2,A3
A1  NOP     LABEL ACCESSIBLE BY PROGRAM
A2  NOP     LABEL ACCESSIBLE BY PROGRAM
A3  NOP     LABEL ACCESSIBLE BY PROGRAM
    END
```

Blank system common is a reserved block of consecutive memory locations. The block is initialized to zero by the generator. There is no label associated with blank common. The system merely keeps track of it as the common area.

The following Generator command declares 40 words of system blank common to the generator:

```
COM,40
```

An example of a FORTRAN program that accesses both types of common:

```
$ALIAS /A1/, NOALLOCATE
    PROGRAM SAVER(4,89),SAMPLE PROGRAM TO USE COMMON
    COMMON/A1/SAVE1,SAVE2,SAVE3
    COMMON IA1,IA2,IA3,IA4
    SAVE1 = IA1 * IA2 + IA3 * IA4
    END
```

When this program is linked using the LC and SC LINK commands, the program takes the first four words of blank system common and puts the result of the calculation into location SAVE1 in labeled system common. Note that the FORTRAN variable referencing A1 may be any legal name but that the name of the common block (in this case A1) must correspond to the name of the entry point given in the module relocated in labeled common. Also, because of the way the module DATA was written, SAVE2 corresponds to A2 and SAVE3 corresponds to A3.

Relocation of Programs Using System Common

All programs calling for blank common are assumed to mean local common unless changed by the SC command in the LINK command file.

Similarly, labeled common is assumed to be local to the program unless the LC command is specified. If the LC command is not specified, the required modules are appended to the program if they exist in the user modules or system libraries searched, or LINK reports the symbols as undefined external references.

Whenever the size or content of system common is changed via the generator, all programs that access this area should be examined to see whether they need corresponding modifications to their source code. Additionally, these programs are required to be linked again.

Shared Subroutines

Using shared subroutines is one of two ways a program can operate in privileged mode. (The other method is explained in the Privileged Operation chapter of the *RTE-A Programmer's Reference Manual*.)

Shared subroutines must be used with caution. You have to be very careful of what you call at what time. Program errors are timing-dependent and almost impossible to reproduce.

A shareable subroutine in RTE-A must be generated into system common. Notice that the generator and LINK modify the code according to whether the subroutine is actually placed in system common where it is shared by calling programs or merely appended to each program that uses it; that is, the entry points .ZRNT and .ZPRV are dummy instructions to the generator or LINK. Whenever LINK encounters a JSB .ZRNT or JSB .ZPRV instruction, the instruction is replaced by the RSS instruction. The generator replaces the calls to .ZRNT and .ZPRV with calls to $LIBR. A parameter passed to $LIBR determines whether the call is:

Level 1 subroutine - Privileged subroutine coded using $LIBR
Level 2 subroutine - Privileged subroutine coded using .ZPRV
Level 3 subroutine - Reentrant subroutine coded using .ZRNT

Level 1 should be used sparingly, since normal I/O interrupts and memory protect are completely disabled while the subroutine is executing. However, while the level 1 subroutine is executing, privileged interrupts may be serviced. The three levels of shared subroutines are described in the following paragraphs. The formats of the three levels of shared subroutines are shown in Tables 8-1 through 8-4.

Level 3 Shared Subroutines

Level 3 subroutines have the least impact upon other programs. The call may be interrupted and other programs may be dispatched. If, however, another program attempts to call this subroutine before it has been exited by the first program, the second program is shared-resource (SR) suspended until this resource becomes available.

Level 3 subroutines may call any other subroutines in any level. During the operation of a level 3 subroutine, memory protect is enabled. I/O instructions and instructions that store or jump into the system partition are not allowed. The interrupt system is on, allowing normal I/O interrupts to occur.

Level 3 subroutines may not be called via a PCAL instruction. Thus, level 3 subroutines are not accessible from CDS code. These subroutines must be called via a JSB instruction (non-CDS code).

Level 2 Shared Subroutines

While a level 2 subroutine is in operation, no other program can be dispatched. All other programs must wait for the subroutine to complete. The SS and PR user commands have no effect until the subroutine exit is made.

A level 2 subroutine can call other level 2 subroutines or level 1 subroutines. It also can make EXEC, MESSS, and LURQ type calls if the no-suspend option is specified. This is because a program must never be suspended while it is executing a level 2 subroutine. (Suspended status includes LU lock suspend, program wait suspend, I/O suspend, and time suspend.) If the subroutine makes an EXEC call, which would normally cause the program to be suspended, the program is aborted with an SR error. It may not call level 3 subroutines. During the operation of a level 2 subroutine, memory protect is enabled. I/O instructions and instructions that store or jump into the system partition are not allowed. During its operation, the normal interrupt system is on and normal interrupts may occur.

Level 1 Shared Subroutines

While a level 1 subroutine is executing, normal I/O interrupts are off; only privileged interrupts are serviced. The level 1 subroutine has total control of the computer and the subroutine has the same privileges as the operating system itself. An improperly coded level 1 subroutine can crash the system.

A level 1 subroutine may call other level 1 or 2 subroutines but may not call any level 3 subroutines, EXEC, MESSS, LURQ, or any type 7 (utility) subroutines. If a level 1 subroutine calls a level 2 subroutine, the system treats the level 2 subroutine as if it were a level 1 subroutine; that is, interrupts are off and MP is disabled.

During the operation of a level 1 subroutine, the normal I/O interrupts are turned off and memory protect is disabled.

Guidelines for Using Shared Subroutines

The hierarchy of the levels is shown below:

Level 3 may call level 3, 2 or 1
Level 2 may call level 2 or 1
Level 1 may call level 2 or 1

If a subroutine calls another subroutine at the wrong level, the program is aborted with the SR error message.

Note Because of the impact upon other programs, it is important to restrict the amount of time spent in any shared subroutine. A good rule of thumb is to limit level 1 and level 2 subroutines to 1 millisecond. Level 3 subroutines may be longer, but remember that a low priority program calling a level 3 subroutine can hold off a higher priority program that calls the same subroutine.

The generator modifies the subroutine if it is placed in system common. LINK replaces the call with an RSS instruction.

When using the exit sequence through $LIBX, the actual return address is in TDB+2, placed there by call to $LIBR. Therefore, if adjusting the return address, increment both the contents of SUB and TDB+2 by the same amount. Only one of SUB or TDB+2 is actually used for finding the return address, according to whether or not the subroutine is in system common.

Table 8-1. Shared Subroutine Format - Levels 3 and 2

Level 3 Subroutines - No Parameters

Assembled form		In System Common		Appended to program	
TDB	NOP	TDB	NOP	TDB	NOP
	DEC n+3		DEC n+3		DEC n+3
	NOP		NOP		NOP
TEMPS	BSS n	TEMPS	BSS n	TEMPS	BSS n
SUB	NOP	SUB	NOP	SUB	NOP
	JSB .ZRNT		JSB $LIBR		RSS
	DEF EXIT		DEF TDB		DEF EXIT
	.		.		.
	.		.		.
EXIT	JMP SUB,I	EXIT	JSB $LIBX	EXIT	JMP SUB,I
	DEF TDB		DEF TDB		DEF TDB
	DEC 0		DEC 0		DEC 0

Level 2 Subroutines - No Parameters

Assembled form		In System Common		Appended to program	
TEMPS	BSS n	TEMPS	BSS n	TEMPS	BSS n
SUB	NOP	SUB	NOP	SUB	NOP
	JSB .ZPRV		JSB $LIBR		RSS
	DEF EXIT		DEC −1		DEF EXIT
	:		:		:
EXIT	JMP SUB,I	EXIT	JSB $LIBX	EXIT	JMP SUB,I
	DEF SUB		DEF SUB,I		DEF SUB,I

Table 8-2. Level 2 Subroutine with Parameters

Level 2 Subroutines with Parameters

```
TEMPS   BSS m       TEMP VARIABLES USED BY SUB
PARMS   BSS n       PARAMETER ADDR PICKED UP BY .ENTP
SUB     NOP
        JSB .ZPRV
        DEF EXIT
        JSB .ENTP   BRING IN PARAMETER ADDRESSES AND
        DEF PARMS   ADJUST RETURN ADDRESS IN ENTRY POINT
        :
EXIT    JMP SUB,I
        DEF SUB
```

In System Common

```
TEMPS   BSS m
PARMS   BSS n
SUB     NOP
        JSB $LIBR
        DEC -1
        JSB .ENTP
        DEF PARMS
        :
EXIT    JSB $LIBX
        DEF SUB,I POINTS TO RTN
```

Appended to Program:

```
TEMPS   BSS m
PARMS   BSS n
SUB     NOP
        RSS
        DEF EXIT
        JSB .ENTP
        DEF PARMS
        :
EXIT    JMP SUB,I
        DEF SUB
```

Table 8-3. Format of Level 1 Shared Subroutine

No Parameters

```
SUB     NOP
        JSB $LIBR
        NOP
        :
        :
        :
EXIT    JSB $LIBX
        DEF SUB
```

Called with Parameters

```
TEMPS   BSS n       PARAMETER ADRES-
SESSUB  NOP
        JSB $LIBR
        NOP
        JSB .ENTP
        DEF PARMS
        :
EXIT    JSB $LIBX
        DEF SUB
```

Note: The level 1 subroutine is not modified by either the generator or LINK.

Table 8-4. Level 3 Subroutine with Parameters

Level 3 Subroutine with Parameters

```
TDB     NOP
        DEC m+n+3
        NOP          IF IN COMMON, SYSTEM STORES RETURN ADDR HERE
TEMPS   BSS m        TEMP VARIABLES USED BY SUB
PARMS   BSS n        PARAMETER ADDR PICKED UP BY .ENTP
SUB     NOP
        JSB .ZRNT
        DEF EXIT
        JSB .ENTP
        DEF PARMS
        STA TDB+2    UPDATE RETURN ADDR IN CASE THIS SUBROU-
        .            TINE
        .            GETS PLACED IN COMMON
        .
EXIT    JMP SUB,I
        DEF TDB
        DEC 0
```

In System Common:

```
TDB     NOP
        DEC m+n+3
        NOP
TEMPS   BSS m
PARMS   BSS n
SUB     NOP
        JSB $LIBR
        DEF TDB
        JSB .ENTP
        DEF PARMS
        STA TDB+2  UPDATE RETURN
        .
        .
        .
EXIT    JSB $LIBX  EXIT THRU TDB+2
        DEF TDB
        DEC 0
```

Appended to Program:

```
TDB     NOP
        DEC m+n+3
        NOP
TEMPS   BSS m
PARMS   BSS n
SUB     NOP
        RSS
        DEF EXIT
        JSB .ENTP
        DEF PARMS
        STA TDB+2  NO EFFECT
        .
        .
        .
EXIT    JSB SUB,I  EXIT THRU SUB
        DEF TDB
        DEC 0
```

9

System Base Page and Link Words

This chapter describes the system base page and memory links. The system base page is an area of the system partition that contains link addresses for the operating system. Memory linking is accomplished with a link word generated by the RTAGN program to access a location on another page via a link on the same page (current page linking) or base page (base page linking).

System Base Page Format

The system base page contains interrupt vectors as well as link addresses for the operating system. The format of the system base page is shown in Figure 9-1 and described in the following paragraphs.

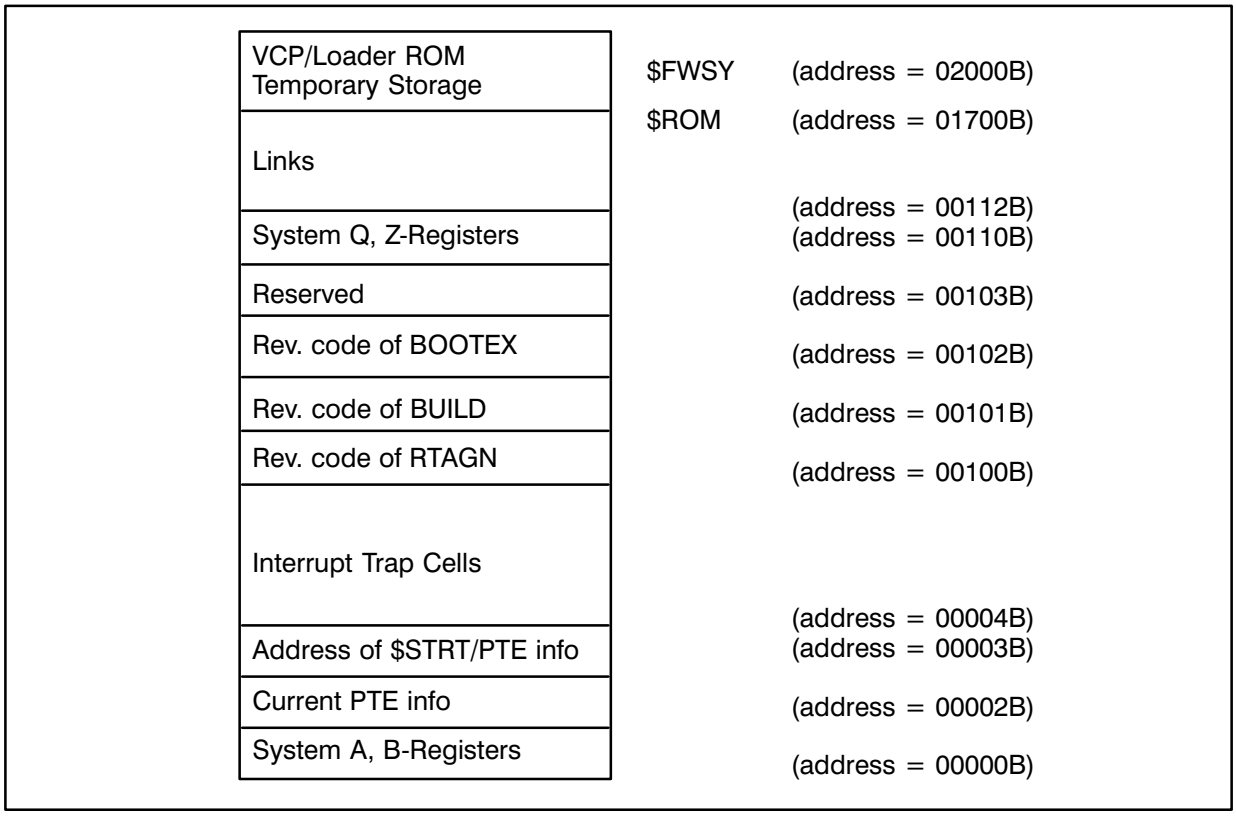


Figure 9-1. Memory Map of System Base Page

VCP/Loader ROM Temporary Storage

The HP 1000 A-Series computer has no switches or lights on its front panel for examining or modifying memory and hardware registers. Instead, a program that resides in ROM on the processor board allows a normal user terminal to function as a virtual control panel (VCP). The same ROMs also contain the programs needed to load the operating system into main memory from a disk or other secondary storage, and to perform the CPU/memory/backplane selftest. The last 64 words of system base page are reserved for temporary storage for the VCP/loader programs.

Interrupt Trap Cells

Trap cells are memory locations that contain instructions to be executed whenever an interrupt occurs on a select code that is numerically equal to the address of the trap cell.

Some trap cells are used by I/O device interfaces to request service by the corresponding software drivers. Others are reserved for interrupts that are generated by the CPU itself:

Select Code	
4B	Power fail auto restart, and system initialization flag
5B	CPU parity error (PE)
6B	Time Base Generator/system clock (TBG)
7B	Memory Protect (MP)
10B	Unimplemented Instruction (UIT)
11B	Reserved
12B	EMA/VMA Page Fault
13B	Code Segment Fault (CDS programs only)
14B-17B	Reserved
20B-77B	I/O Interrupts

In the case of an I/O device, an interrupt is usually generated when a read or write request completes. The service routine $CIC, in the system module RTIOA, handles all non-privileged drivers. In $CIC, an LIA 4 instruction determines which select code created the interrupt. The select code is used to index into the interrupt table, which contains the address of the interface table (IFT) associated with the driver for that interface card or an ID segment address of a program to schedule. In most cases, the interrupt table contains an IFT address. The IFT is used to identify the interface driver, which is to be entered by the system after setting up some pointers that may be used by the driver.

Power Fail Trap Cell/System Initialization Flag

System base page location 4 is used as a flag by the boot loader in case of power failure during the boot process. After the operating system has been initialized, it is changed from a zero to a subroutine call to the power fail restart routine, if the power fail driver is generated into the system. If the value of location 4 is zero when CPU power is restored (and the content of memory has been preserved by the battery backup) then the operating system had

not been completely loaded and initialized. In this case, the operating system code in memory is incomplete, and cannot be restarted. Instead, either the boot load is restarted, or the VCP is entered, depending on the position of the processor card switches.

Address of $STRT

Location 3 contains the address of the RTE-A startup routine, $STRT. The system generator places a JMP 3,i instruction in location 2. When the system is booted, the VCP transfers control to location 2 after loading the system into memory. After system initialization, this word is reused to hold information about the current program's PTE.

Current PTE Information

Locations 2 and 3 on the system base page are used as a communication area for the VMA/EMA microcode. When a program that uses VMA/EMA is dispatched, the operating system puts information about the location of the page table (PTE) associated with the program into these locations. The VMA/EMA microcode uses this information to access the page table of the program.

Link Words

A link word is created by the RTAGN program whenever a one-word memory reference instruction accesses a location not on the same page as the instruction. This is required because these memory reference instructions contain only a 10-bit field for the address to be referenced, the remaining bits being used for the OP code of the instruction itself. These 10 bits allow 1024 words of memory, or one page, to be directly accessed by this instruction.

In the above case, the generator changes the instruction to an indirect reference (the sign bit is set) to a link word on the same page (current page link) or on the base page (base page link). The linkage editor (LINK) also creates link words for programs that it links.

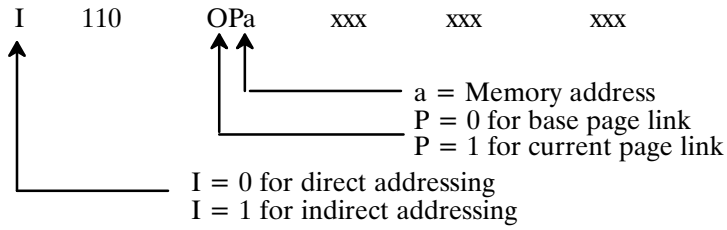
The link word contains the 15-bit address of the location to be accessed. A 15-bit address allows 5 bits to be used as a logical page number (accessing 32 pages), and 10 bits to be used for the word offset on the page (1024 words per page).

For example, if the instruction is an LDA at location 2020 and the location to access (load into the A-Register) is 4000B, then the generator can use either of the following.

	Base Page Linking			**Current Page Linking**		
	Location	**Contents**		**Location**	**Contents**	
Base Page	0300	4000 ←	BP link	0300	????	
Page 2	2010	????		2010	4000 ←	CP Link
	2020	160300 ←	LDA DATA	2020	162010 ←	LDA DATA
Page 3	4000	xxxx ←	DATA	4000	xxxx ←	DATA

where ???? indicates that the contents are irrelevant.

Note that the LDA instruction format is:



Generator Current Page Linking

The generator can be told to use either base page links or current page links. If current page linking is specified, some base page links may have to be allocated by the program due to the following factors:

1. If a module crosses more than one page boundary, any links generated by the portion of the module that completely fills a page must be put on base page.

2. The links required for references to other pages are estimated before the module is relocated. In some cases, the estimate may not be large enough and any links over the estimate must go to the base page.

Figure 9-2 shows how memory is used for current page links.

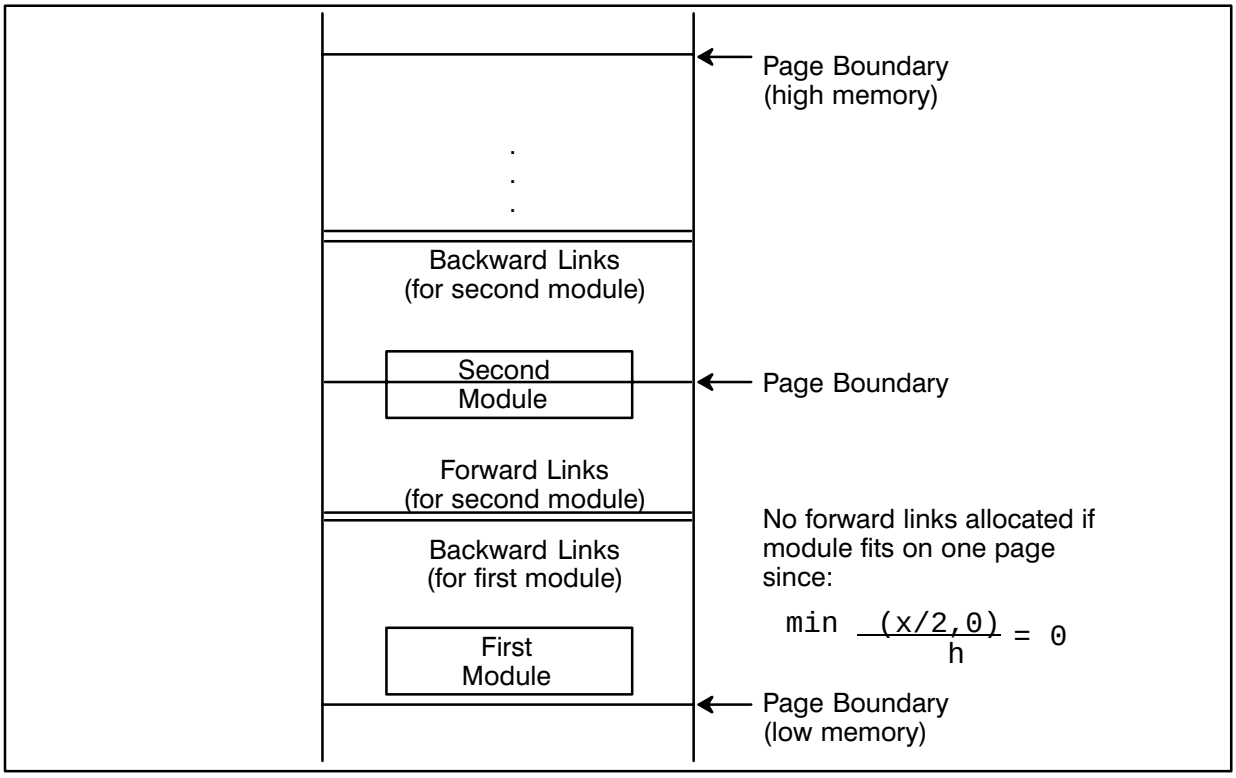


Figure 9-2. Memory Usage for Current Page Links

The algorithm for estimating the number of forward links that may be required is:

$$\frac{\min(x/2,\ y)}{h}$$

```
min = function to take the minimum of the two values
  x = number of words left on the page at start of process
  y = number of words of module on the next page
  h = a constant (currently 16)
```

The algorithm assumes an even distribution of the references requiring the formation of links. In some cases, this may cause an overestimate of the links required and some of the forward link area may be unused. In other cases, the algorithm may underestimate the requirement and then base page links have to be generated for the overflow.

The start of the backward link area is computed by adding the size of the module (known from the NAM record) to the starting address of the relocated module. The size of this area is equal to the number of links required by the portion of the module relocated on this page. The relocation of the next module starts immediately after the last link generated.

Base Page Linking

During the relocation process, LINK creates base page links (only in the data partition of CDS programs), and the generator creates some base page links whether or not current page linking was specified. If current page linking was specified, significantly fewer base page links are generated than if base page linking was specified.

During system common relocation, any base page links required are allocated starting from the top of the user base page (location 1777B) down toward low memory. The links are stored in the snapshot file, not in the system file. They are copied into the base page link area of each program that accesses system common as it is relocated.

Links required by operating system modules and drivers are allocated starting from the VCP/Loader temporary storage area in system base page (location 1700B) toward the lower boundary (location 112B).

Current Page Links in CDS Programs

Base page links are not possible in the code segments of CDS programs. This is because what were formerly base page references would now be interpreted by the hardware as locations on the stack in the data partition. Therefore the loader must generate current page links for all Memory Reference Group (MRG) instructions that access locations in the data partition that are not locations on the stack in the data partition. All the links on each page are contiguous. The link area is placed between modules whenever possible, however, it can be placed in a module at a location designated by the language processor. Refer to the Macro/1000 Reference manual for a discussion of the BREAK pseudo-op. The size of the link area can be influenced by commands given to LINK.

10

File System

This chapter describes how the file system is implemented in the RTE-A Operating System. Topics contained in this chapter are:

- File System Organization
- FMP Routines
- Directory Organization
- Disk Management
- FMGR Files
- Remote File Access
- Spool System Interaction (VC+ only)

File System Organization

The file system consists of File System Package (FMP) routines, program D.RTR, additional programs supporting remote file access, and disk data. Programs such as CI or TF are users of the file system and are not part of the file system.

The file system provides disk storage for data and facilitates access and maintenance of the files in the file system. It handles allocating disk space to files, maintains directories containing file names and other information, and provides the means for moving data into and out of files.

The FMP routines are used in the file system to provide fast and simple file access. These routines are front-ends to the directory manager D.RTR, which is the main program of the file system operations.

Several of the FMP routines use a Data Control Block (DCB) to record information about an open file. The DCB is contained in the program to improve the performance of read, write and position operations. The DCB consists mostly of information needed to read and write the disk, plus a packing buffer to facilitate extracting data that is not on a block boundary.

Most of the file system work is done in D.RTR. This program is scheduled by the FMP routines. D.RTR is the only program allowed to change information in the directory. If several programs all need to schedule D.RTR at once, all but one of them has to wait. This ensures mutual exclusion of access to directories. D.RTR performs as much as possible of the uncommonly used calls (such as

rename) in order to reduce logical and physical memory requirements in systems with many programs.

D.RTR maintains all of the long-term information about the file system on disk, maintaining file system integrity through system shutdowns and crashes. The most important tables are the file directories and the space management tables. These tables contain all of the information that is known about a file, directory, or disk volume.

Besides the main function of managing disk files, the file system provides access to devices through the same calls that are used to access disks. This provides a large degree of device independence when dealing with common devices such as terminals and printers. Device management is handled mostly by the FMP routines, although D.RTR does create the DCB for a device-open call.

The file system also uses a technique for using the hierarchical file system calls to access files located on remote systems connected through the Distributed System DS/1000-IV or NS-ARPA/1000. The file system implements this through two programs that communicate over the DS link. These programs handle reading and writing data across DS, as well as handling remote D.RTR calls and DS concerns. Remote access is described later in this chapter.

FMP Routines

There are approximately 80 File Management Package (FMP) routines. With a few exceptions, the names of these routines begin with the letters FMP. These routines can access all features of the file system.

Most FMP routines are small and simple because they are interfaces to D.RTR functions (for example, DCB headers are set up by D.RTR). FMP routines that must be fast, such as FmpRead and FmpWrite, do not interface with D.RTR.

Directory Organization

Directories are the central file system data structure. Directories maintain the file system state across system halts and crashes. All information pertaining to a file is kept in a directory. Directories may be included in other directories; these are considered subdirectories. Nesting of subdirectories is allowed to provide a hierarchical file system structure. At the top is a root directory that contains only unique global directories. There is one root directory per disk volume.

Mounting a disk volume makes directories on that volume accessible. All global directories on that volume are made known to the system when a volume is mounted. Global directory names must be unique in the file system. Files in a directory are not identified by disk volume.

D.RTR uses its free space for a global directory/open file table. Each global directory is stored in an abbreviated form that takes up five words for each entry. An entry for an open file takes up six words in the table. The non-CDS version of D.RTR has enough free space to support approximately 180 global directories and 180 open files. If you want to reduce the number of global directories on your system, you can use the MO operator command to convert global directories to subdirectories. This would allow more files to be open at the same time. Note that there is one entry in the table used for each open file/program pair; 200 programs opening one file

is the same as one program opening 200 files. The CDS version of D.RTR should be used by systems that have a large number of global directories and/or require a large number of open files to be handled by the system.

The CDS version of D.RTR offers a number of advantages over the non-CDS version. The CDS version can be configured at link time to tailor the free memory usage according to the system's needs. The file "ddmax.mac" can be modified and compiled to reflect the system's global directory requirements. The relocatable output can then be used when linking the CDS version of D.RTR. Symbolic links are only supported on systems that have a CDS D.RTR. This version of D.RTR also supports the use of the 40b/41b control requests to control the removeable media status of the SCSI magneto-optical drives. Depending on the generation parameters, you may elect to have D.RTR spin down or eject the removeable media when the media is no longer in use as an RTE-A file system. By default, the file "ddmax.mac" sets up a global directory/open file table that can support up to 200 global directories and 200 open files. The SCSI 40b/41b functionality is disabled by default.

Each directory consists of a doubly linked set of disk blocks containing entries for files, extents, and subdirectories. Each of these entries is 32 words. In addition, 32 words at the beginning and end of each directory are used for bookkeeping information. Directories are extendable, although performance is improved when they are kept to a small number of extents; 32 words are also needed at the beginning and end of each extent of the directory.

Each file is kept as a single file entry in a single directory. The file entry contains the file name, information about the disk space used by the file, and other information such as time stamps and protection.

Purged files are flagged by a bit in the directory entry. When a file is purged, the directory entry is marked as purged, and its disk space is marked as free. The file can be unpurged until its directory entry or disk file space is allocated to a file or extent. Note that it is possible that a purged file has its space allocated to a new file, then the new file is purged; now the first file appears to be recoverable but actually is not. This situation is rare, but it can be confusing when it does occur.

Files are extended when necessary to hold additional data. Sequential files always have extents at least as large as the main file, but the extent size is doubled for extent number 4, 6, 8 and so on. For example, the first 10 extents of a 1-block file have size 1,1,1,1,2,2,4,4,8,8 blocks. This prevents having files with large numbers of extents, which slows access. Random access files always have extents the same size as the file did when it was created, but there can be missing extents. Files with missing extents are known as sparse files, and serve to conserve disk space. VMA backing store files are the only common type of sparse file because no disk space is required for records that are never accessed.

Information about open files is maintained by D.RTR. Each time a program opens a file, D.RTR records the file directory address and the calling program's ID segment number in an internal table. This table is checked at each open to ensure that for an exclusive open, only the calling program is opening this file, and that only the calling program is purging the opened file or dismounting the volume. The limit to the number of programs that can open a file or to the number of files that a program can open is dependent on both the size of the global directory/open file table and the number of global directories that exist on the system.

Each ID segment contains a bit that indicates whether the open file table has been scanned since the last time that ID segment was recycled for use by a new program. The RTE-A system clears this bit and D.RTR sets the bit after scanning the open flag table and flushing any flags that belonged to the previous occupant of that ID segment. Obsolete flags are also flushed when found by the checks made when opening a file or dismounting a disk volume.

Disk Management

Disks are divided into volumes and each volume is assigned a logical unit number. There can be one or more LUs per disk drive. When a disk volume is mounted, everything physically located on that LU is available. This may include a large number of directories. Volumes are completely self-contained, and never reference other volumes. This allows each volume to be dismounted independently of other volumes.

The last track of a volume contains a volume header, which points to the other tables located on the volume. The volume header is the location where disk cartridges used by the FMGR files can be differentiated from the disk volumes used by the hierarchical file system. It is the only data on the disk that has a fixed location that cannot be changed.

The volume header identifies the storage allocation unit for the volume. This storage allocation unit is typically one block (256 bytes), but in some situations it is more than one block. This occurs because the file system maintains a bit map for each volume identifying whether each allocation unit is used or free. The corresponding bit is set when the block is used, cleared when the block is free.

The bit map table is kept relatively small to decrease the time to find free space, and it allows the whole table to fit in memory at the same time. D.RTR has an 8k word buffer available, which is 128k bits. The allocation unit for disk space is chosen to keep the bit map size at 8k words or less. Thus this unit is one block for volumes up to 128k blocks, two blocks for up to 256k, four blocks for up to 512k, and so forth. Here is a table of disk volume size in MBytes, blocks, and allocation unit size:

MBytes	Blocks	Allocation Unit
16	64k	1
50	200k	2
120	480k	4
400	1600k	16

Using larger allocation units wastes disk space when creating files, extents, or directories smaller than one allocation unit. In most cases this is not a problem; however, the drive can be subdivided into several smaller volumes if problems occur.

A small number of large volumes is recommended. This allows allocating files from a common pool of free space, thus making efficient use of space. Otherwise, errors may occur when one volume is out of space while there is a large amount of empty space on other volumes on the same disk drive. Having a small number of volumes increases the options available with the operator MO command and reduces the system table size. When a large volume is out of space, it may take longer for the MPACK utility to compact the files on it.

Files are always allocated as contiguous groups of one or more blocks. Allocation is done via first fit, starting at block zero on the disk. This tends to increase locality while keeping fragmentation to a tolerable level. Files can be placed at a desired location by allocating the lower numbered blocks to a file temporarily, then creating the file. If the desired number of contiguous blocks cannot be found, the file is not created. Space is reclaimed immediately when files are purged. Using the bit map table, holes can be automatically merged into bigger holes as space becomes available.

Record Lengths

Excepting type 6 and type 12 files, the file types 3 and above have record length information stored with the data. There is a record length word before and after each record. The length word consists of the number of bytes in the record rotated right one place. This means that the sign bit of the word is set for odd byte length records and cleared for even byte length records.

This format allows reading disk files from previous RTE systems that stored record lengths in words. However, it causes problems when moving files from RTE-A to these systems (for example, RTE-6/VM, RTE-IVB, or RTE-XL). If you must move a file to any of these systems, make sure that all record lengths are even. Otherwise, programs such as FC that copy disk images will copy data with record lengths that cannot be read correctly by the previous RTE systems.

One way to move a file to any of the previous RTE systems is to copy it to a FMGR disk cartridge. Then use EDIT to open the file for editing and replace it with the ER command. EDIT makes all byte lengths an even number when it writes to a FMGR disk. Now you can move the file with FC to the destination system.

Symbolic Link Files (VC+ only)

A symbolic link is a file whose contents is a file descriptor that points to a device, another file, or a directory. This file descriptor can be specified by either a relative ('../dir/file') or absolute ('/dir/file') path name. If a symbolic link to a relative path name is encountered during path name resolution, the contents of the symbolic link replace the symbolic link component and is expanded into the path name being resolved. If a symbolic link to an absolute path name is encountered, the contents of the symbolic link replaces all components up to and including the symbolic link and is expanded into the remainder of the path name. All symbolic links are resolved in this manner except when the symbolic link is the last component of a file descriptor which is passed to one of the following FMP calls:

FmpPurge,
FmpRename,
FmpSetOwner,
FmpSetProtection,
FmpReadLink, and
FmpMakeSLink.

With these calls, the symbolic link itself is accessed or affected.

A symbolic link may refer to any arbitrary path name and may span different LUs. Symbolic link files may be FMGR files. Links to FMGR directories, FMGR files, and FMGR type 0 files are not supported. The path name may point to another symbolic link. Thus it is possible that a symbolic link points to itself or another symbolic link in such a way that it forms a closed loop. D.RTR limits the number of symbolic links it traverses while translating a path name so that it can detect this situation. When D.RTR traverses more than eight symbolic links, an FMP −260 error is reported. The protection and ownership of a symbolic link is ignored by the system. The protection and the ownership of the actual file being accessed is still checked by the system.

Symbolic links may be used to access remote files using DS transparency. There are some restrictions that apply when using symbolic links to remote files :

- A user's working directory may not be set to a remote directory.
- Remote type 6 files cannot be executed.
- Remote access of a symbolic link that points to another remote file is not supported. Attempts to access the remote symbolic link result in FMP error −262.
- FMP masking does not recursively search subdirectories that are linked to a remote directory. Masking recursively searches a remote directory when the search starts at the remote directory.

For example :

The directory /home.dir contains two files. A local subdirectory 'local.dir' and a symbolic link to a remote directory 'remote.dir'. The command 'dl /home/@.ftn.s' recursively searches only the 'local' subdirectory for the files with the '.ftn' type extension. However, the command 'dl /home/remote/@.ftn.s' recursively searches the remote directory for the desired files.

Symbolic links that point to a directory must have a '.DIR' type extension.

Given the path name /e/i/x/o, where x is a symbolic link to ../e/i.dir, the original path name would be interpreted as /e/i/../e/i/o. If, instead, x were a symbolic link to an absolute path name such as /w/h.dir, the same path name would be interpreted as /w/h/o.

Symbolic links can be used to shorten the path name required for frequently used files or directories. They can also be used when certain applications require files to be in specific directories. Symbolic links can make it appear as though files from several volumes are all under the same directory tree.

The LNS utility is used to create symbolic link files. Refer to the *RTE-A User's Manual* for a detailed description of the LNS program.

FMGR Files

The FMP routines and D.RTR support, to a certain extent, the existing FMGR files that are different from those created in the hierarchical file system. In the FMGR file system, each volume has only one directory; collectively the pair is known as a disk cartridge. The directory contains less information than the CI file system directory. For example, the file names are restricted to six characters and there are no time stamps. Because of these and other restrictions, the FMP routines are not always able to perform their designated functions on FMGR files. The following paragraphs describe how the FMGR disk cartridges are supported. Refer to the *RTE-A User's Manual* for a detailed description of the FMGR program.

FMGR Cartridges

There is a cartridge list that indicates the cartridges mounted at any one time. This is used to find the LU associated with a particular cartridge name, or Cartridge Reference Number (CRN). The FMP routines go to the cartridge list when there is no directory on a normal volume that has the required name. If there is a CRN with the required name, the subroutines try to find the file on that CRN. CRN zero means search all of the FMGR cartridges in the order they appear in the cartridge list. Note that there may be a directory with the same name as a CRN; the file system does not prohibit this, but you should watch out for it, as it can cause confusion.

The following functions can be used with files on FMGR cartridges:

```
Open  Close  Create  Purge   Rename  Size  Truncate
```

All CI commands that make these types of calls also work, such as LI, DL, CR, and PU. Other types of calls, such as UNPU or PROT, do not work, and return an error.

Differences between FMGR and RTE-A Files

Files on FMGR cartridges observe a different set of rules. The following are the major differences that apply to FMGR files:

Names limited to six characters.

No file type extension.

Periods, slashes, and brackets are not treated as special characters.

One directory per volume, not automatically extendible.

No subdirectories.

No time stamps or backup bit.

No record count, record length, or EOF pointer.

Unpurge is not available.

Open flags maintained on disk.

Limit of seven programs opening a file concurrently.

A program can exclusively open a file already open to it.

No protection information, except a 16-bit security code that must be matched to write to the file if it is positive, or to open the file if it is negative.

Disk space is not managed with a bit map; instead, a next track and sector pointer is maintained, and space is reused when a purged entry exactly matches the required size.

File size −1 means all of the space left on the cartridge.

Type zero files that describe a device are available.

Remote Access

Systems with the DS/1000-IV can take advantage of access to hierarchical files on remote systems, owned by other accounts, or both. The following paragraphs describe how remote access is implemented.

Remote access is implemented by a small amount of code in FMP and D.RTR, and by two monitor programs that communicate through low-level services of DS/1000-IV. Remote access is indicated through special characters appearing in the file name; the name

```
File.ext::Dir[PAL/light]>System
```

indicates a file on node System, owned by user PAL with password light. When D.RTR gets a request to access a file with either an account name or node name or number in it, it tells FMP to reroute the schedule request to the monitor DSRTR. DSRTR figures out which node to send the request to, and maintains internal information needed to keep track of this request. In particular, for open requests, DSRTR replaces the first word of the DCB with a connection number. This number is flagged by D.RTR, which tells FMP to reroute all access to the DCB through DSRTR.

DSRTR communicates with a remote monitor called TRFAS; TRFAS receives a DS message and schedules D.RTR on the remote system, returning any parameters to DSRTR. TRFAS can also handle requests to read or write to the disk, or to log on or off. DS routes requests to the local TRFAS if an account was specified with no remote node.

A connection password is used to verify that TRFAS has not been restarted due to system restart. If DSRTR ever presents the wrong connection password, TRFAS assumes its system has been restarted and rejects the request, because the file system may have changed. While TRFAS is running, all files opened by it are recorded by D.RTR at the remote system. TRFAS does not know anything about the open files, as it maintains no state, but tries to close a file on request. This is useful if the local system goes down, or if a program gets OF'd without closing a file; otherwise, the files stay open to TRFAS.

11

System Tables

This chapter describes the system tables and their formats. The system tables are:

ID Segments

Resource Number Table

Logical Unit Table

Device Table

Interface Table

Map Set Table

Interrupt Table

Class Table

Swap Descriptor Table

Shareable EMA Table

SHEMA Association Blocks

Cartridge Directory

Memory Descriptor Table

Shared Program Table

Multiuser Tables

CDS Tables

Language Message Table

For information on the system security tables, refer to the *RTE-A System Manager's Manua*l, part number 92077-90056.

ID Segment

An ID segment is a 45-word array that contains program identification information. It contains the name, status, size, and location of the program, as well as temporary storage for use by the operating system when the program suspends.

Space for ID segments is allocated during system generation. The ID segment is initialized when the program is connected to the system. This is done by the BUILD program for memory-based systems, or by the BOOTEX program and user RU/XQ/RP commands for disk-based systems.

Some of the words can have two meanings: some entries in the ID segment of a Code-and-Data-Separation (CDS) program contain different values than the ID segment of a non-CDS program. The code-block-size field in word 26 (bits 4-0) distinguishes the two types; the value of this field is zero for non-CDS programs and non-zero for CDS programs. The two formats are shown in Figures 11-1 and 11-2 and are described below.

NO TAG shows the words appended to the ID segment image in the type 6 file header created by LINK. They are used by IDRPL but are not included in the ID segment in memory. Parameter T (bit 0) in word 65 indicates transportability; if T is zero, the program is not transportable. If the T-bit is set, the type 6 file can be executed on a system with a different generation. If the T-bit is not set, the checksums in the type 6 file must match the checksums of the destination system in order to execute the program.

Short ID segments are 8-word arrays used to identify non-CDS program overlays. One short ID segment is used for each overlay. The short ID segment format is shown in NO TAG.

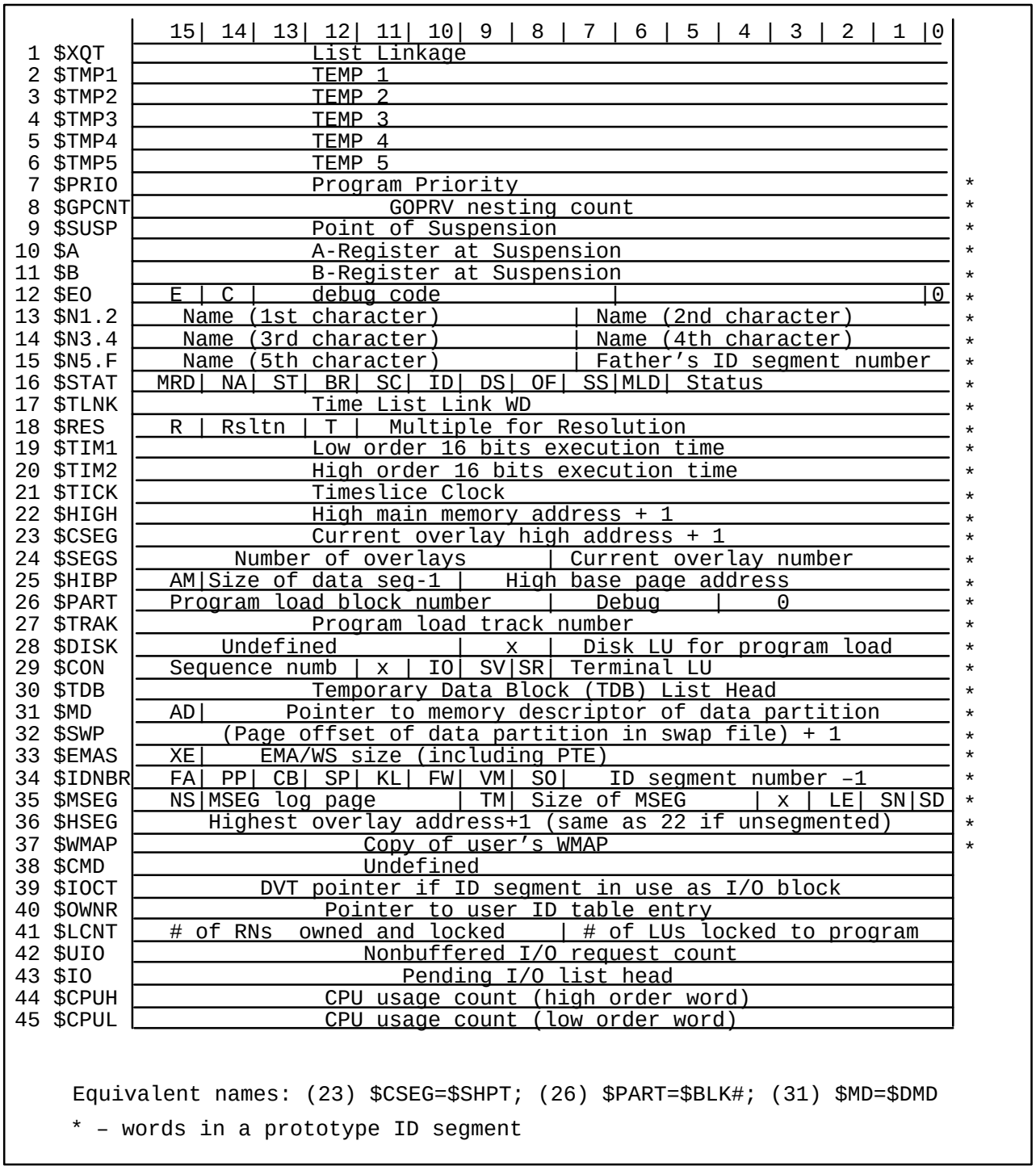


Figure 11-1. ID Segment Format for Non-CDS Programs

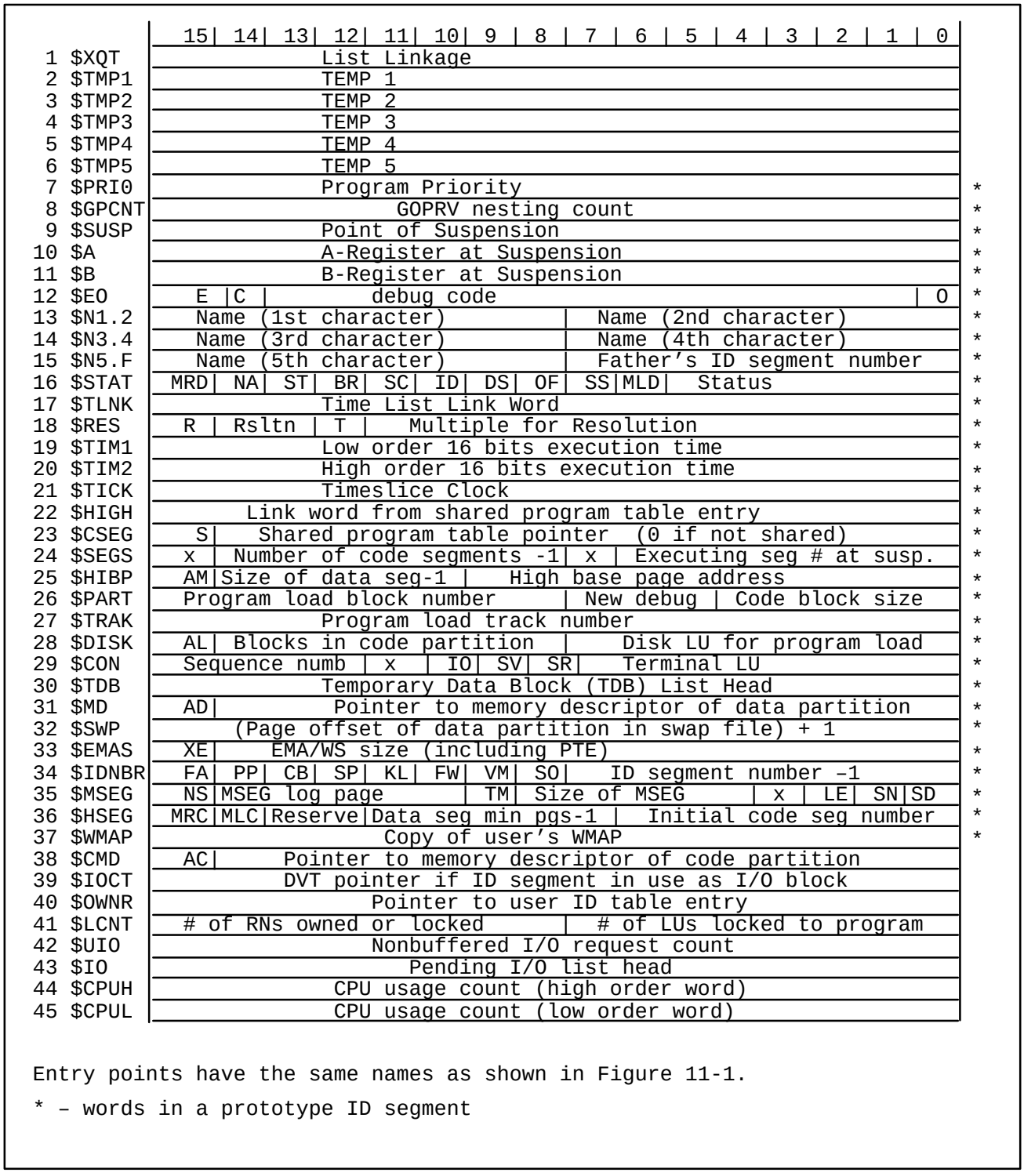


Figure 11-2. ID Segment Format for CDS Programs

In addition to the ID segment itself:

$IDA is a pointer to the first ID segment.
$ID# contains the number of ID segments.
$IDSZ contains the size of each ID segment in memory (45 words at Rev. 6000).

In Figures 11-1 and 11-2, the first column numbers indicate the word number. Those that are not self explanatory are listed and described below (x indicates a reserved bit).

Words 2 - 6 : Temporary information storage recoverable by RMPAR. This information is modified by each EXEC request. The I/O control block for nonbuffered I/O requests is built here.

Word 8: Nesting count of GOPRV/UNPRV calls made by program

Word 12: E = E-Register at suspension

O = O-Register at suspension

C = C-Register at suspension

Debug Code = Debug Reason Code

Word 16: MRD= Set if the program is in memory. For CDS programs, this is set if the data is in memory.

NA = No-abort.This is set to the sign bit of the control word in an EXEC request.

ST = Set when a string in SAM is created for program. System clears on next EXEC request (releasing string) or subroutine SAVST clears.

BR = Break bit. Set by user break and may be tested (and cleared) by function IFBRK.

SC = If set, program accesses system common.

ID = If set, the ID segment is deallocated when the program terminates.

DS = If set, the program has DS/1000 resources locked.

OF = If set, this is an instruction to the system to set the program dormant at the earliest opportunity.

SS = If set, this is an instruction to the system to suspend the program at the earliest opportunity.

MLD= If set, the program has executed an EXEC 22 request and is locked in memory (not swappable). For CDS programs, this bit represents the data segment's memory locked status.

The following is a list the program states related to the possible values (in octal) of the status bits (5 through 0) of $STAT:

Program State	$STAT
Dormant	0
Dormant saving resources	0
Dormant and in time list	0
Program abort in process	1
I/O suspend	2
Program wait suspend	3
Operator suspend	6
Pause	7
Waiting for signal	10
Signal buffer limit suspend	46
Time suspend	47
Locked device suspend	50
Resource number suspend	51
Class I/O suspend	52
Queue suspend	53
Down device suspend	54
I/O buffer limit suspend	55
Load suspend	56
Shared subroutine suspend	57
Scheduled	60
System available memory suspend	61
Spool suspend	62
Extended system available memory suspend	63

Word 18: R = Program has been time-scheduled relatively (not absolutely). Valid only when T is set.

Rsltn = Resolution in:

00 = Milliseconds 10=Minutes
01 = Seconds 11=Hours

T = Program is in the time list.

Word 22: For non-CDS programs: High main memory address + 1 (bits 0 - 15)

For CDS programs: Link word from shared program table entry (Valid only if ID segment Word 23, bit 15 is set.)

Word 23: For CDS programs:

S = Set if program is shared (bit 15) and bits 0-14 contains the shared program table pointer.

Word 24: For CDS programs:

Bit 15 and bit 8 are reserved.
Bits 8-14 represent number of code segments in program
Bits 0-6 represent executing code segment number at suspension. Numbers are 1-128; above values are 0-127.

Word 25: AM = data segment (bit 15).

Set if any free memory within the partition must be swapped with the program. The LINK or operator SZ command results in the setting of this bit. For CDS programs, this applies only to the data partition.

Bits 10-14 = data segment size −1 pages (CDS)
= program size −1 (non-CDS)

Word 26: Bits 0-4 used in CDS programs to indicate size of code segment block in pages; for non-CDS programs, value is zero.

Bits 5-6 are used by Symbolic Debug.

Word 28: For CDS programs:

AL = all code segments are memory-resident (bit 15).
(mandatory for shared programs)

Bits 8-14 = number of code segment blocks in code partition.

Word 29: SR = Set if program terminated serially reuseable (bit 8)

SV = Set if program terminated saving resources (bit 9)

IO = Set during processing of VMA I/O or nonbuffered EXEC call (bit 10)

DB = Symbolic Debug bit (bit 11 set)

Word 31: AD = Assigned bit indicating data is assigned to a reserved partition (bit 15 set)

Word 32: Page offset in swap file + 1 (bits 0 - 15)

Word 33: XE = 1 if the program is a Large or Extended model EMA/VMA program.

Word 34: FA = Program has made file access (bit 15 set)

PP = Primary program in session (bit 14 set)

CB = Program in Gocrit State (bit 13 set)

SP = System process if set (bit 12)

KL = Program may be OF'd by any user (bit 11 set)

FW = Program is scheduled with wait by another program that is waiting for its termination (bit 10 set)
VM = Program is a VMA program.
SO = Program is SHEMA-only.

Word 35: Logical start page of MSEG (bits 10 - 14)
TMR timer bit; timer signal processing (bit 9)
Size of MSEG (bits 4 - 8)
LE = Program uses Large model EMA or VMA.
SN = Program is allocating memory for a secondary SHEMA.
SD = Signal to be delivered.

Word 36: For non-CDS programs:
Highest overlay address + 1 (bits 0 - 15);
if no overlays, then Word 36 is equal to Word 22

For CDS programs:

MRC = Code in memory (bit 15 set)

MLC = If bit 14 set, the program has executed an EXEC 22 request and the code segment is locked in memory (not swappable).

Bits 12-13 = reserved
Bits 7-11 = Minimum size of data segment in pages
Bits 0-6 = Initial code segment number

Word 37: Copy of user's working map register at suspension

Word 38: Not used in non-CDS programs

For CDS programs:

AC = Code partition assigned to a reserved partition (bit 15 set)

Bits 0-14 = Pointer to code partition memory descriptor

Word 39: Back pointer to DVT associated with normal nonbuffered I/O request by the program (a word to serve this function also was added to the other I/O request blocks)

Word 40: Address of entry in user ID table

Word 41: Bits 8-15 = Number of resource numbers locally owned or locked by this program
Bits 0-7 = Number of LUs locked by this program

Word 42: Count of nonbuffered I/O requests

Word 43: List head of pending I/O list; used for abort processing

Words 44-45: CPU usage count; used for accounting and METER

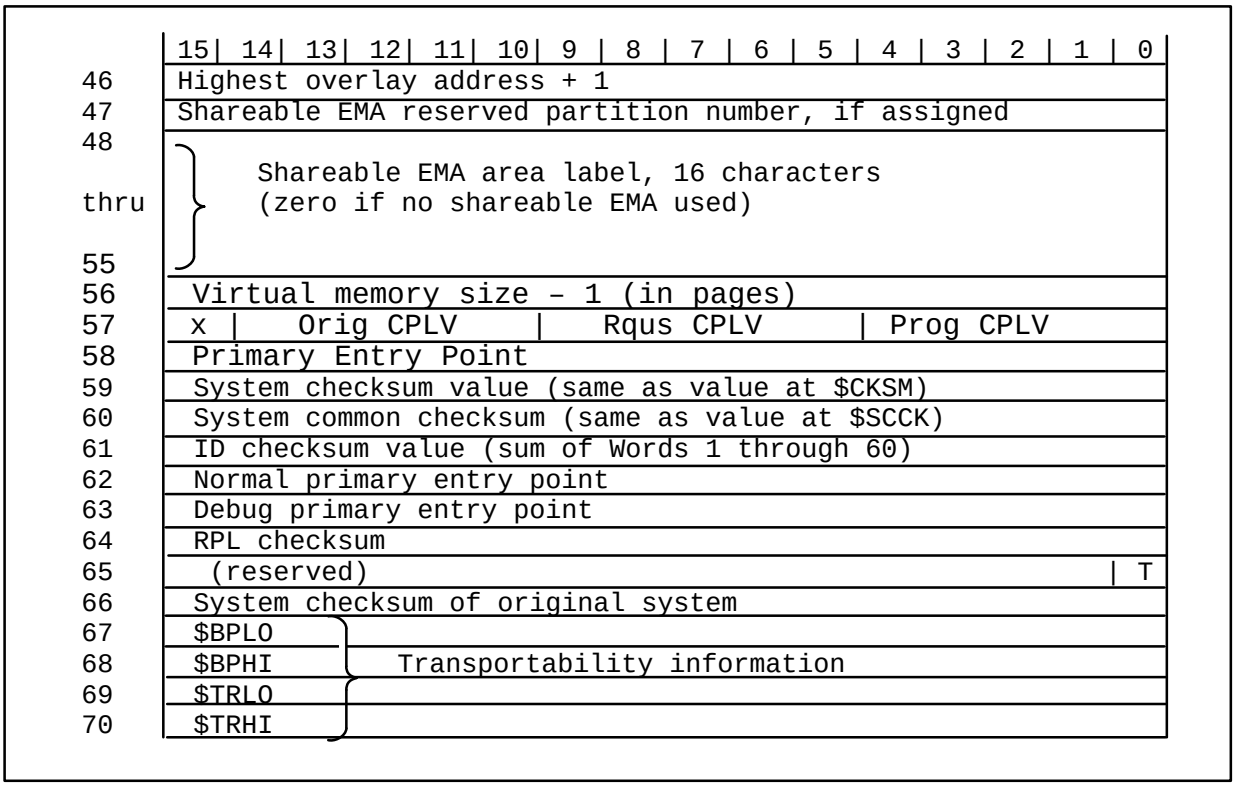


Figure 11-3. Words Appended to ID Segment Image In Type 6 File

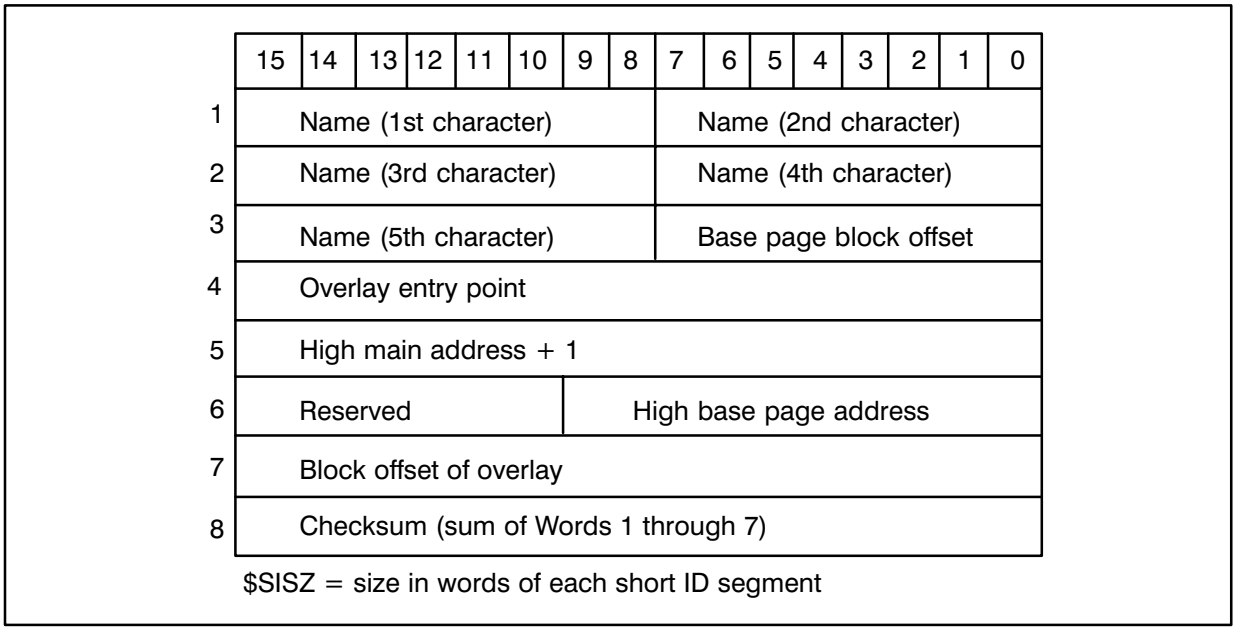


Figure 11-4. Short ID Segment Format

ID Segment Extensions

One ID segment extension exists for each ID segment defined in the system. The memory for all of the ID segment extensions is allocated as a block of XSAM at system startup time. The block starts at XSAM address 3. Each extension consists of the number of words given by entry point $IDXSZ in $VCTR. Entry point $IDEXT holds the XSAM address of the ID segment extension for the currently executing program. The ID segment extension format appears in Figure 11-5.

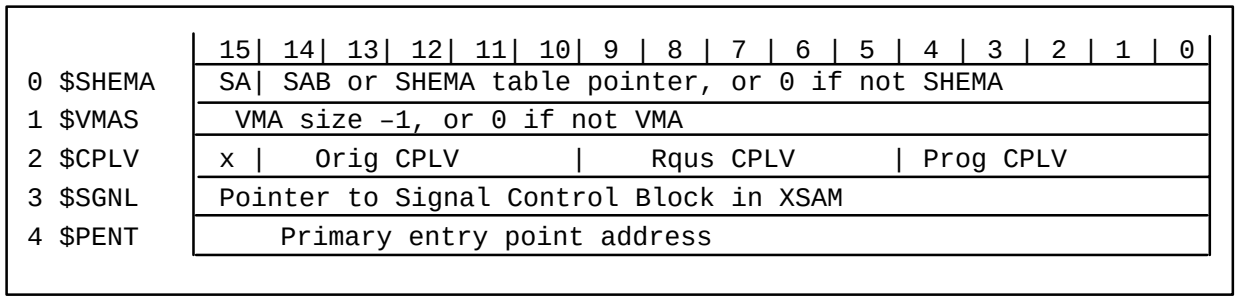


Figure 11-5. Format of ID Segment Extension

Word 0: SA = 0 if bits 0−14 point to a SHEMA table entry in XSAM or are zero, indicating that this program does not use SHEMA.

SA = 1 if bits 0−14 point to a SHEMA Association Block (SAB) in XSAM, describing multiple SHEMA associations.

Bits 0−14 point to either a SHEMA table entry or a SHEMA Association Block in XSAM, or are zero if the program does not use SHEMA.

Word 2: Org CPLV = Capability assigned at link time (bits 10−14)
Rqus CPLV = Required user capability level (bits 5−9)
Prog CPLV = Current program capability level (bits 0−4)

Word 3: Pointer to the program's Signal Control Block in XSAM, or zero if the program does not use signals.

Resource Number Table

The Resource Number Table is an array in memory that contains information pertaining to the allocation, deallocation, ownership, and locked status of resource numbers. Space for the table is allocated by the generator, which also sets the first word of the array to the length of the table.

The resource table format is shown in Figure 11-6. This table is used whenever any program makes a call to RNRQ.

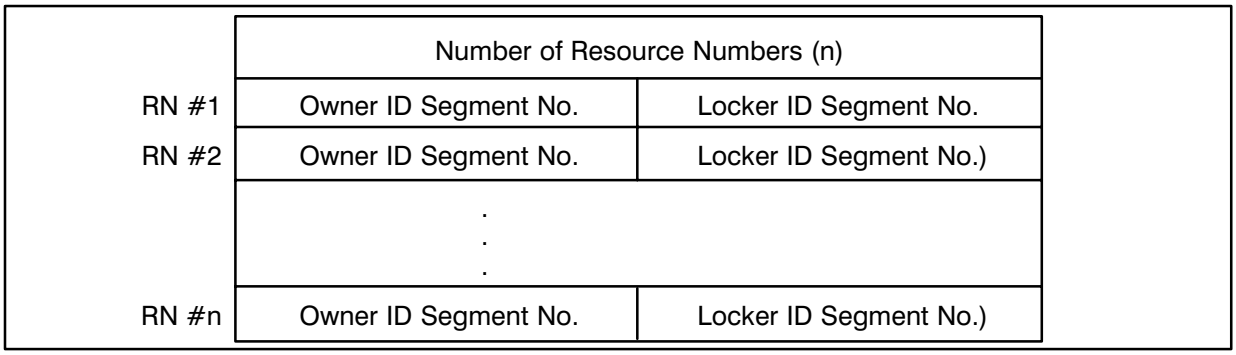


Figure 11-6. Resource Number Table Format

The pointer to the length word of the resource number table is $RNTA. In the table itself, an entry of 377 (octal) for either the owner or locker field indicates that the number is globally owned or locked. The entry contains an ID segment number if the number is locally owned or locked. An entry of zero for either the owner or locker field indicates that the number is not owned or not locked.

Logical Unit Table

The Logical Unit Table (LUT) is a table that maps logical unit numbers to the appropriate device table. In the table, each word is indexed by an LU number and contains the address of the DVT associated with that LU number. This table is set up by the generator. There may be up to 255 logical units in the system. The format of the logical unit table is shown in Figure 11-7. The pointer to the first word of the LU table is $LUTA and $LUT# contains the number of entries in the LUT.

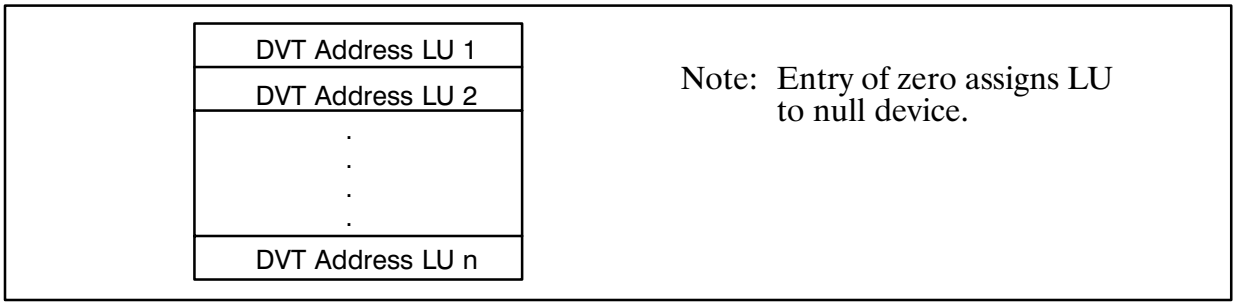


Figure 11-7. LU Table Format

Device Table (DVT)

The Device Table (DVT) is a table in memory that contains device-specific information such as the driver to be used to communicate with the device, the device status, the device time out and buffer limit values, and other device parameters. This table is used to identify each device to the system. The LU number of each device is linked to the DVT for that device. The DVT is then used in managing the requests to the particular device.

Each device has a DVT, which is created at generation and which is linked to the other DVTs on the same interface as well as to the interface table itself. A few of the parameters (for example, the buffer limits and timeout) may be altered on-line with operator commands. Figure 11-8 shows the device table format. The entry points pointing to the currently active DVT are set up by the operating system before the device driver is called.

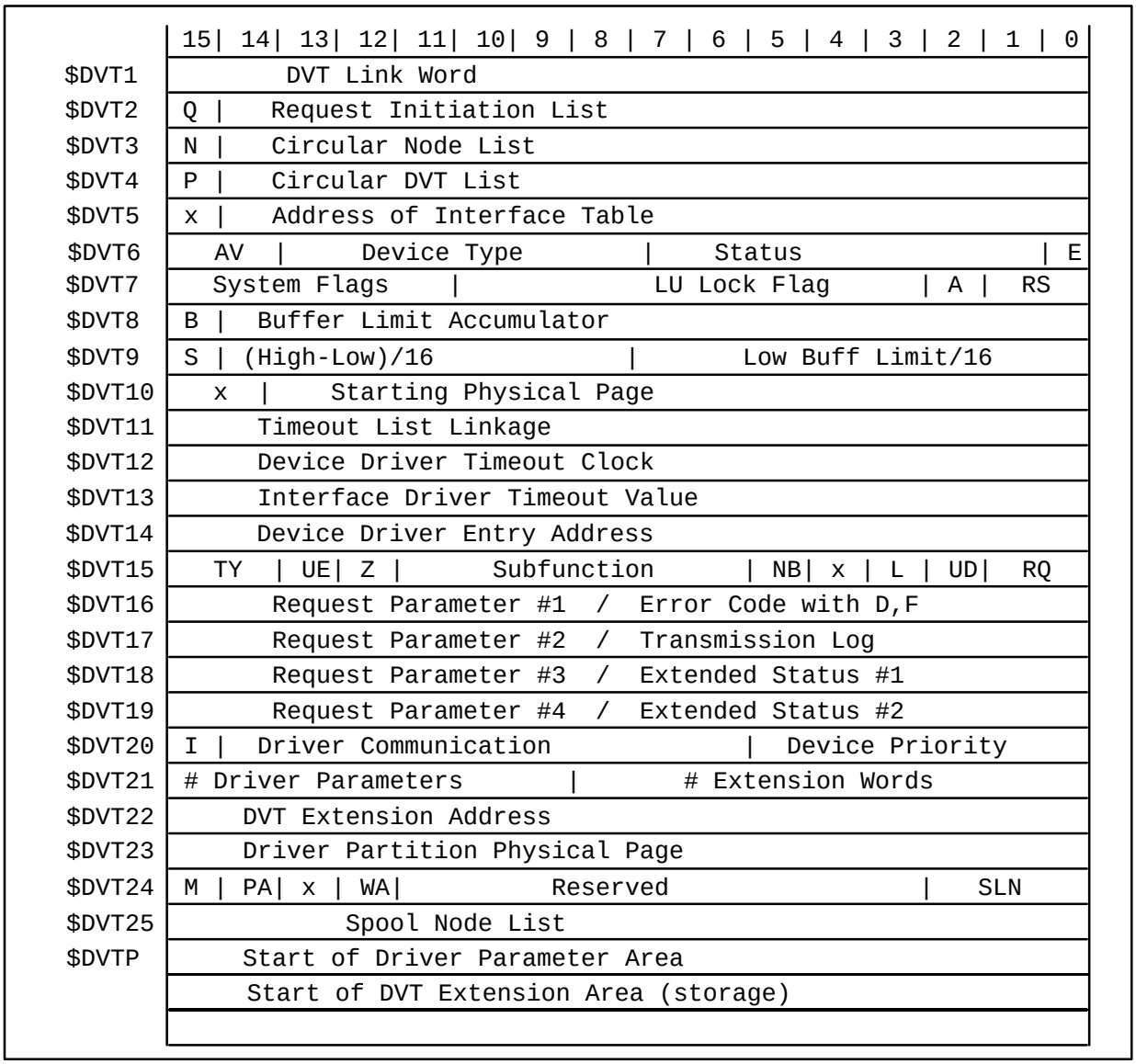


Figure 11-8. Device Table Format

The pointer to the first DVT is $DVTA; the number of DVT entries is contained in $DVT#, and the size of each DVT in $DVSZ. The parameters shown in Figure 11-8 are explained below (x indicates a reserved bit).

Word 2: Q = Queuing option of I/O requests to the DVT. If 0, then queuing by program priority is specified. If 1, then queuing is FIFO (first in, first out).

Word 3: N = Node status bit. If 0, then this DVT is not busy. If 1, then this DVT is busy. This bit is used in determining whether a request from another DVT on the same node list may be acted on.

Word 4: P = Power fail service flag. If 0, do not call the associated driver on power fail. If 1, call this driver on power fail to allow it to perform any power fail operations necessary for its devices.

Word 6: AV = Availability field:

0 Device is available.

1 Device is down.

2 Device is busy.

3 Device is down and busy.

Word 7: A = Abort bit. If 1, the system is in the process of aborting the pending request.

RS = Request status:

0 Request queued on IFT (request initiation list).

1 Request queued on IFT (current head of list).

2 Request queued on IFT request complete list.

3 Request queued on DVT request complete list.

Word 8: B = Buffering option. If 0, the device is nonbuffered; if 1, the device is buffered.

Word 9: S = Buffering status bit. If 1, the request has exceeded the upper buffer limit.

Word 10: Starting physical page of the I/O transfer. The first map register may be set to this value. In the case of VMAIO, this value is not relevant (bits 0-13).

Word 14: Entry address of the device driver. If 0, the device does not have a device driver.

Word 15: TY = Request type.

0 User program request

1 Buffered user program request

2 System I/O request

3 Class I/O request

UE = User error bit, specified in EXEC request.

Z = Double buffer bit, specified in EXEC request.

NB = Nonbuffered bit, specified in EXEC request.

L = If set, the data is in the user partition. If clear, the data is in the system partition, or in SAM for buffered or CLASS requests.

UD = If set, bypass the device driver (call only the interface driver). Specified in EXEC request.

RQ = Request type, specified in EXEC request code:

0 Multibuffered request

1 Read request, Write/Read request

2 Write request

3 Control request

Word 20: I = Initial request bit. If 1, this is the first time the driver has serviced this device.

Word 23: Starting physical page of the driver partition where the device driver resides. If 0, then the driver is not in a partition.

Word 24: M = Location of the I/O control block pointed to by $DVT2. If 0, it is in the system map; otherwise, it is in SAM.

PA = Pseudo abort bit. If 1, pseudo abort is active.

WA = Waiting to abort bit. If 1, an abort request is pending.

SLN = System language number (0-7). This 3-bit field defaults to zero during system generation.

Word 25: Spool node list pointer into SAM. Non-zero indicates spooling or LU redirection is in effect for this device.

Interface Table

The Interface Table (IFT) is a memory resident table that contains interface-specific information. It identifies the interface to the system. Each IFT handles all the interface I/O requests made as a result of an I/O request made by a program to a device. This table contains such information as the interface driver, the select code, and the interface type for a particular interface. This table is created and initialized by the generator during the table generation phase. The format of the IFT is shown in Figure 11-9. The entry points to the IFT are set up by the operating system before control is transferred to the interface driver.

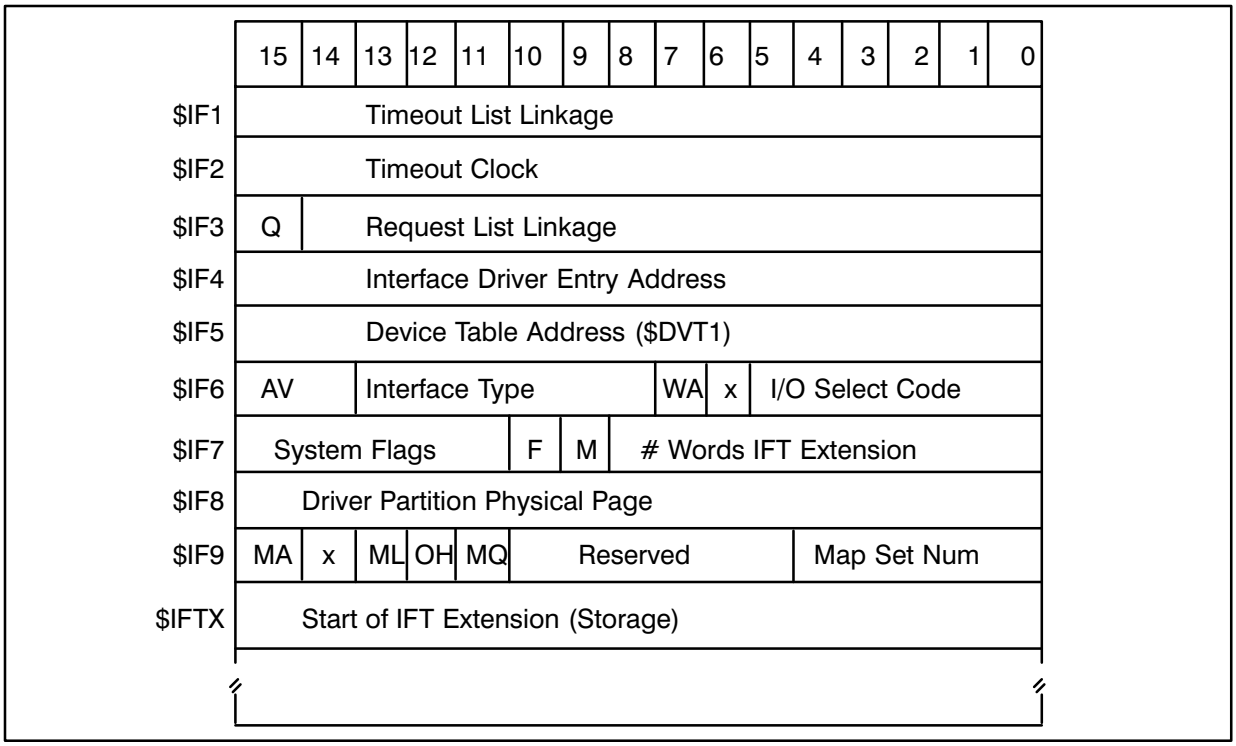


Figure 11-9. Interface Table Format

The pointer to the first IFT is $IFTA; the number of IFT entries is contained in $IFT#, and the size of each IFT in $IFSZ. The parameters in Figure 11-9 are described below (x indicates a reserved bit).

Word 3: Q = Queuing option of the DVTs with active requests. If 0, the requests are queued by device priority. If 1, the requests are queued in a FIFO manner.

Word 6: AV = Availability field:

0 Interface is available.

1 Interface is locked to DVT.

2 Interface is busy.

3 Interface is locked and busy.

WA = Waiting to abort bit. If 1, an abort request is pending.

Word 7: F = First entry bit. If 1, this is the first time the driver has serviced a request on this IFT.

M = If 1, the interface driver is responsible for de-queuing the list.

Word 8: The starting physical page of the partition where the interface driver resides. If zero, then the interface driver is not in a partition.

Word 9: Contains flags dealing with the mapping of I/O channels into map sets.

MA = Map allocated bit. If 1, a map set is allocated for this I/O channel.

ML = Map locked bit. If 1, the system deallocation routine $MSRTN does not deallocate the map set. If 0, the map is dynamically allocated and deallocated.

OH = On hold bit. Contains a copy of the H bit from the system flags area of IFT7. This bit is copied when I/O is suspended to wait on a map set.

MQ = Map set queued bit. If 1, this request is in the map set suspend queue and waits until a map set is available.

Map Set Num = The number of the map set that is allocated for this I/O channel. If the MA bit is clear, this field is meaningless.

I/O Control Blocks

Each I/O request has a block of memory associated with it that defines the request. This block is known as a control block. These blocks have different formats and are located in different places depending on whether the request was a normal, buffered, system, or class request. The format for each kind is shown in Figure 11-10. Note that the control block format is similar for each of the different requests.

The XSIO control block contains words that are not necessary for any of the other requests. A short description of the remaining words follows:

***** Word 1 *****

List Linkage = Links the control block onto a DVT. DVT2 points to the first control block and the last block will have this word equal to zero. If this is a class request, this word can also be used to link the request into a class number's completed class queue. There can be a queue for each of the class numbers that starts at word 1 of a class number's entry in the class table. In this case the list is terminated with the class status word for this class number. This word has the same format as word 1 of the class table would have if the class was only allocated. See the description of the class table elsewhere in this chapter.

***** Word 2 *****

Control Word = Same as DVT15 except that the L bit is not defined. See the "Device Table (DVT)" section in this chapter for more detail.

***** Words 3 through 6 *****

Parameters = The active request parameter area.

***** Word 7 *****

Priority = The priority of the program that made the request.

Normal Request Stored in ID Segment startling at $XQT	TY = 0	Word	System Request Stored in XSIO Block	TY = 2
		−1	LU	
		0	Completion Address	
I/O List Linkage		1	I/O List Linkage	
I/O Control Address		2	I/O Control Word	
PRAM1 / Buffer Address		3	PRAM1 / Buffer Address	
PRAM2 / Buffer Length		4	PRAM2 / Buffer Length	
PRAM3 / Z-Buffer Address		5	PRAM3 / Z-Buffer Address	
PRAM4 / Z-Buffer Length		6	PRAM4 / Z-Buffer Length	
Priority		7	Priority	
		8	Starting Physical Page of Data	
		9	Status Return	
		10	Transmission Log	
		11	* 1st Extended Status	
		12	* 2nd Extended Status	
		13	* 3rd Extended Status	
Buffered Request Stored in SAM	**TY = 1**	**Word**	**Class Request Stored in SAM**	**TY = 3**
I/O List Linkage		1	I/O List Linkage	
I/O Control Word		2	I/O Control Word	
PRAM1 / Buffer Address		3	PRAM1 / Buffer Address	
PRAM2 / Buffer Length		4	PRAM2 / Buffer Length	
PRAM3 / Z-Buffer Address		5	PRAM3 / Z-Buffer Address	
PRAM4 / Z-Buffer Length		6	PRAM4 / Z-Buffer Length	
Priority		7	Priority	
I/O Block Length		8	I/O Block Length	
ID Segment and Run Numbers		9	Class Information	
Undefined		10	User Defined Value	
Undefined		11	ID Segment Address	
Undefined		12	VMAIO Control Wrod	
Undefined		13	VMAIO Z-Buffer Address	
ID Segment Forward Pointer		14	ID Segment Forward Pointer	
ID Segment Backward Pointer		15	ID Segment Backward Pointer	
DVT Address		16	DVT Address/Previous Length	
Data		17	Data	
•		•	•	
•		•	•	
•		•	•	
Last Word of Data		Block Length	Last Word of Data	

* If the “X” bit of the LU word is set

I/O Control Word

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TY		UE	Z	Subfunction						NB	0		UD	RQ	

Figure 11-10. I/O Control Block

***** Word 8 *****

Block Length = The length of the entire control block, (including the buffer area at the end of the block) if it exists. Block Length − 16 = Buffer Length.

***** Word 9 *****

If class, Class Information = Same as the class parameter that is passed to EXEC except that the NW bit in bit 15 is replaced with the FL bit. If the FL bit is set, the class completion code will flush this class. The 13-bit class number that is passed to EXEC contains the actual class number in the low eight bits and the low five bits of the ID segment number of the allocating program in the top five bits.

If buffered, ID Segment and Run Numbers = Contains the ID segment number of the program that made the request with the sequence number (word 29 of the ID segment) merged into the high four bits.

***** Word 10 *****

User Defined Value = The user defined value that is passed in the EXEC class request.

***** Word 11 *****

ID Segment Address = Address of the ID segment of the program that made the class request.

***** Word 12 *****

VMAIO Control Word = If NV (bit 15 of this word) is set, the request is for nonbuffered class VMAIO and the low ten bits contain the low ten bits of the virtual page in which the buffer starts.

***** Word 13 *****

VMAIO Z-Buffer Address = If VMAIO, contains the Z-Buffer address.

***** Words 14 and 15 *****

ID Segment Pointers = These words make up a doubly linked list that links an ID segment to all active buffered and class requests that the program has made. This list is used to abort all of a programs buffered and class I/O requests, if the program aborts.

Forward Pointer = Word 43 of the ID segment will point to the forward pointer of the request most recently made by the program. This word will contain either the address of the next forward pointer or a zero signifying the end of the list.

Backward Pointer = This word will contain either a pointer to the next backward pointer with bit 15 set, or a pointer to word 43 of the ID segment with bit 15 clear.

***** Word 16 *****

DVT Address = Points to the first word of the DVT that this request is queued on.

Previous Length = When the request completes, the original request length is gotten from word 4 of the control block and saved here for re-thread purposes.

***** Word 17 through Block Length *****

Data = If this is a buffered or a buffered class request, then this area will be used as the buffer for the request. If a Z-buffer is specified for the request, it will also be in this area (immediately following the regular buffer).

Map Set Table

The Map Set Table is 24 words long. Each entry represents one map set. The entries are for map sets 8 through 31. The format of the map set table is shown in Figure 11-11.

The information in the entry for each map set depends on bit 15 (not available bit) and can be one of the following:

State 1: If a map set is available, then:

Bit 15 = 0

Bits 0-14 = Pointer to next free map set. The value is zero if end of list.

State 2: If a map set is in use, then:

Bit 15 = 1

Bits 0-14 = Pointer to IFT that is using the map set.

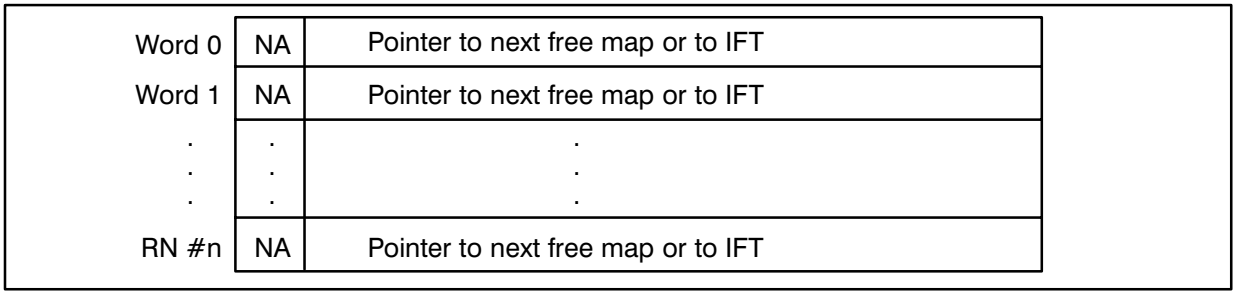


Figure 11-11. Format of Map Set Table

Interrupt Table

The Interrupt Table specifies the IFT that is to be called to service an interrupt from a select code. The interrupt table is set up by the generator. The format of the interrupt table is shown in Figure 11-12.

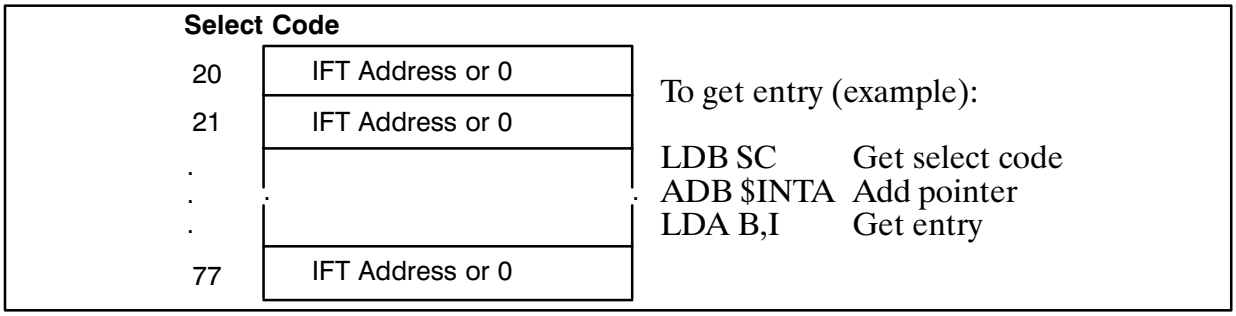


Figure 11-12. Interrupt Table Format

The pointer to the first interrupt table address (select code 20B) is $INTA and the number of interrupt table entries is in $INT#.

The table contains 48 words. Entries corresponding to select codes not defined during system generation contain a zero. If an interrupt occurs from an undefined select code, the system issues the error message “Illegal interrupt on select code xx”.

If an entry point of a privileged driver is specified to handle the interrupts from a given select code, the interrupt table is bypassed. A JSB to this entry point is placed in the trap cell for that select code and a zero is placed in the interrupt table word for this select code, as shown in Figure 11-12. The trap cell and interrupt table entry mapping is shown in Figure 11-13.

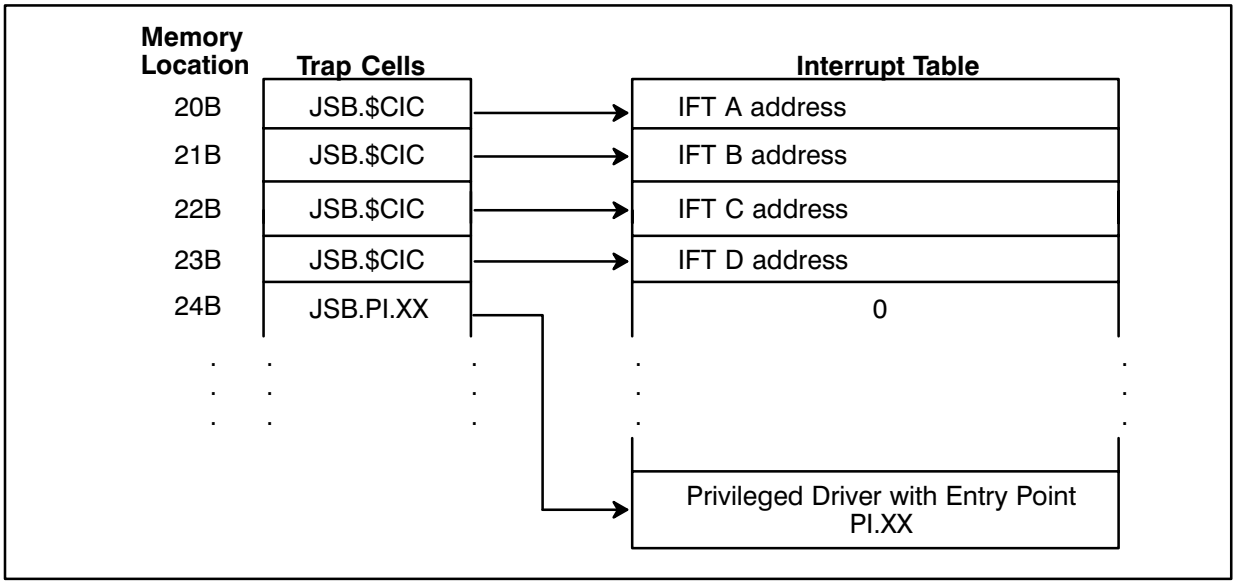


Figure 11-13. Trap Cells and the Interrupt Table

Class Table

The Class Table contains information on the allocation, deallocation, and ownership of class numbers. It is a two word per entry table. Space for this table is allocated by the generator in response to the CLASS,n command. The class table format is shown in Figure 11-14.

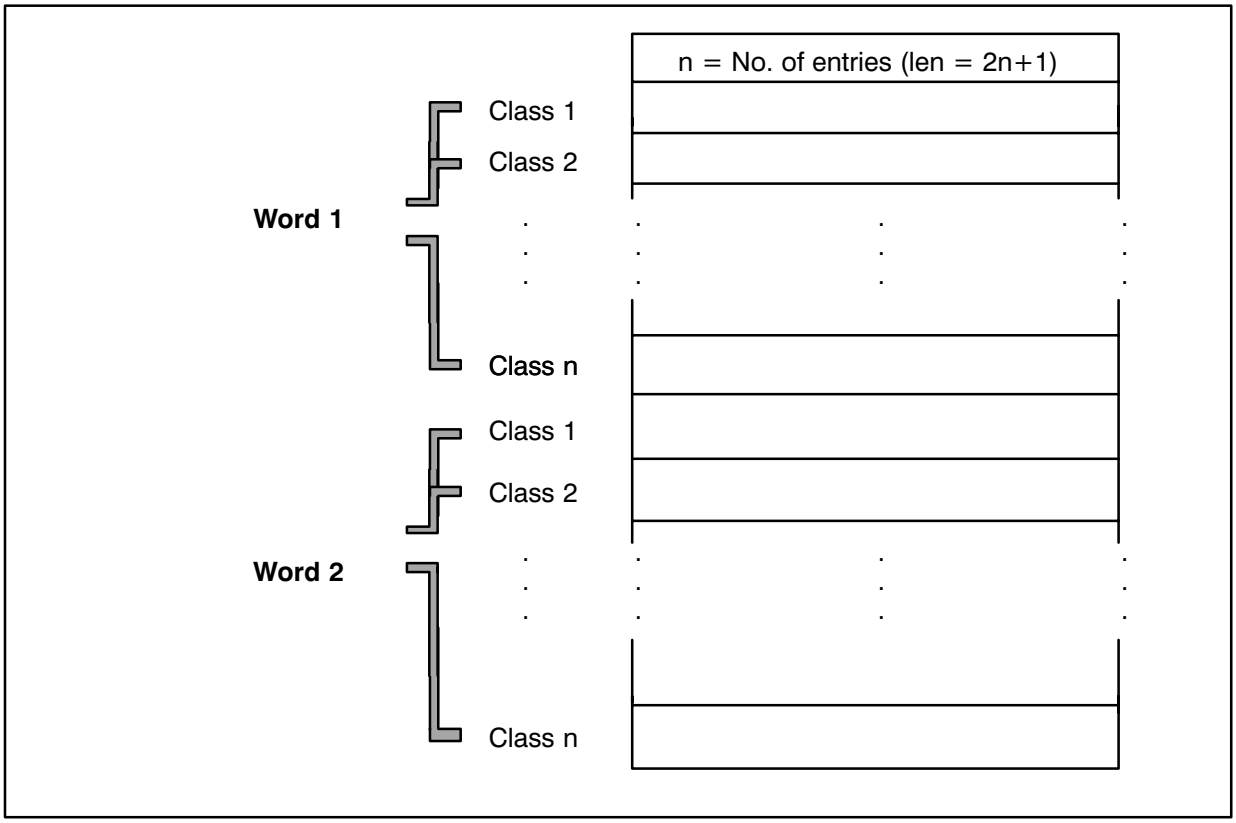


Figure 11-14. Class Table Format

$CLTA is a pointer to the first word of the class table.

Word 1

Contents	**Meaning**
0	Class number is free
positive	Address of queue of completed class I/O requests
negative	Class ID word

Word 2

Contents	**Meaning**
0	Class number free or not assigned ownership
positive	ID segment number of class number owner
negative	Class number in process of being flushed and deallocated by the system.

The class ID word is either an entry in the table (if there are no completed requests) or it is at the end of the completed class queue. Each SAM block in the queue is linked via the first word of the block. Figure 11-15 shows the format of the class ID word (x indicates a reserved bit).

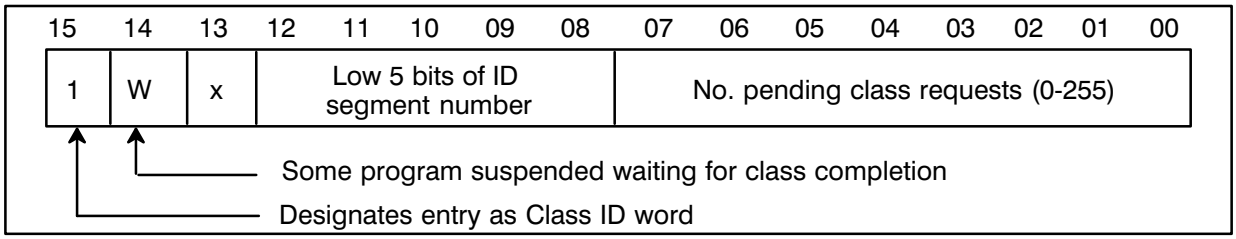


Figure 11-15. Class ID Word Format

Swap Descriptor Table

The system contains the following information about the swap file:

$SWLU : LU of swapping disk
BLK/T : Number of 128-word blocks per track

These values and the swap area descriptors are set up by BOOTEX. The free areas of the swap file are described by a linked list of free swap area descriptors, linked in order of the position of the swap areas on disk. This Swap Descriptor Table format is shown in Figure 11-16.

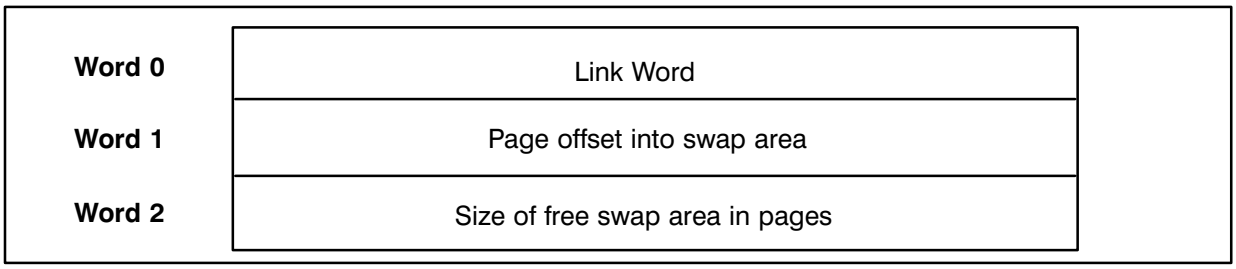


Figure 11-16. Swap Descriptor Table Format

The following values are used in conjunction with this table:

$SWPS: Pointer to free swap area descriptor prior to the one at which to start a search for a free area.

$FSWP: Head of the free swap area descriptor list.

$UFSD: Head of the unused free swap area descriptor list.

When the dispatcher determines that a program is to be swapped out, it starts searching for a large enough free space at the free swap area descriptor pointed to by $SWPS.

When the first large area of sufficient size is found, word 32 in the ID segment (page offset in swap file + 1) is set to the value of word 1 of the free swap area descriptor plus one. Word 1 of the free swap area descriptor is incremented by the size of the program to be swapped. Word 2 of the free swap area descriptor is decremented by the size of the program to be swapped.

If word 2 is reduced to zero, then this free swap area descriptor is removed from the list and put into the list of unused free swap area descriptors (the link word of the descriptor pointed to by $SWPS is set to the link word value of the unused descriptor).

Note that during the search phase, if the end of the list is encountered, the search is restarted at the head of the list. Also, if the first free swap area descriptor is the one found, and if it becomes unused, then the list head $FSWP is updated.

If a swap area of large enough size cannot be found, then the message, “No room in swap file!!!” is printed to the system console. A maximum of 50 such messages are printed during a system bootup.

Shareable EMA Table

A shareable EMA table entry describes a shareable EMA, or SHEMA, in the system. Each entry is allocated in XSAM. Entry point $ShemaTbl in $VCTR contains the XSAM address of the first entry in the list, or zero if no SHEMA table entries currently exist. The SHEMA table entry format is given in Figure 11-17.

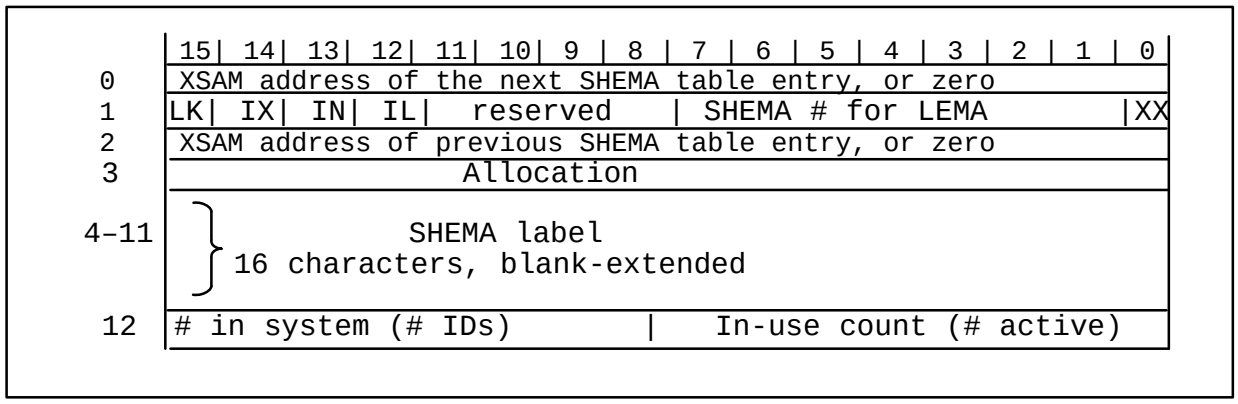


Figure 11-17. Format of Shareable EMA Table

Word 1: LK = 1 if SHEMA partition is locked

IX = 1 if SHEMA partition has been initialized as an Extended SHEMA, or as a Large SHEMA if IL = 1

IN = 1 if SHEMA partition has been initialized as a Normal SHEMA

IL = 1 if SHEMA partition has been initialized as a Large SHEMA, in which case IX will also be = 1

XX = 1 if an extra word of XSAM was allocated for this SHEMA table entry (that is, 14 words were allocated)

For Large EMA programs only, “SHEMA # for LEMA” is set to the EMA segment number minus one for which this SHEMA has been initialized.

Word 3: Specifies the allocation of the area:

0 = Shareable EMA partition is not allocated; it is to be allocated from dynamic memory

1−1023 = Shareable EMA partition is not allocated; it is to be allocated in this reserved partition number

1024 and above =
Shareable EMA partition is allocated; this word holds the address of the Memory Descriptor (MD) for the partition

Word 12: The "# in system" count tells how many programs are RP'ed that use the SHEMA.

The "in-use count" tells how many of the RP'ed program are active, that is, not in the dormant state.

The IX, IL, and IN flags tell whether a SHEMA partition has been initialized, and if so under which model it has been initialized. Once the SHEMA partition is initialized under one of these three models then only programs of the same model may use the partition until the partition is deallocated. Any program not conforming to the same model that attempts to access the partition will be aborted with an EM90 error.

The "SHEMA # for LEMA" field appears in the SHEMA table to allow the system to check that a Large-model program attaching the SHEMA is attaching it at the same EMA segment for which it has been initialized. An attempt by a Large-model program to attach the SHEMA at an EMA segment other than the segment for which it has been initialized will incur an RteAllocShema error. This restriction is necessary because of the format of the EMA page table (PTE) used by the Normal and Large models.

Whenever a program using a particular shareable EMA area is restored in memory with the RP command, the system creates a table entry for the area used by the program, unless an entry for the area already exists. In the former case, the system sets the number-in-system field to 1 and sets the use count to zero. In the latter case, the system increments the number-in-system count by one.

Whenever a program is removed from the system (for example, by an OF,prog,ID command), the number-in-system count is decremented by one. When the count reaches zero, the system deallocates the table entry, unless the area is locked, as described below.

Whenever a program is scheduled using a shareable EMA area, the system increments the use count in the appropriate EMA table entry, provided the program did not previously terminate saving resources. Conversely, whenever a program is terminated other than saving resources, the system decrements the use count in the appropriate EMA table entry.

Whenever the use count goes from zero to one, the system allocates the shareable EMA partition in memory. Whenever the use count goes from one to zero, the system deallocates (releases) the shareable EMA partition, unless the area is locked.

A shareable EMA area may be locked via a LKEMA subroutine call by a program using the area. The LKEMA subroutine sets the LK bit in the appropriate shareable EMA table entry. When it is locked, a shareable EMA partition is not deallocated when the use count (number of active programs using the area) goes to zero.

A shareable EMA area may be unlocked via a ULEMA subroutine call or via the UL operator command. Such a subroutine call or unlock command clears the LK bit in the appropriate shareable EMA table entry. If the use count is zero, the shareable EMA partition is deallocated at this point. If the number-in-system count is also zero, the table entry is also deallocated.

SHEMA Association Blocks

A SHEMA Association Block, or SAB, is created when a program attaches a secondary SHEMA to itself via the RteAllocShema routine. This data structure informs RTE of all the SHEMA attachments that exist for the program.

SABs are allocated dynamically out of XSAM as needed. Each SAB describes up to 8 SHEMA attachments. When another attachment is to be described and all 8 SAB entries are full, another SAB is allocated and linked to the previous one. The format for each SAB is given in Figure 11-18.

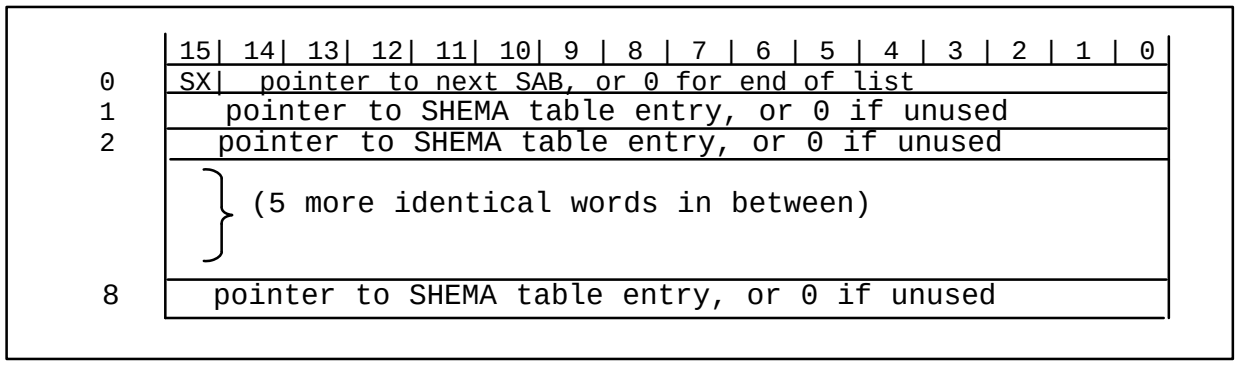


Figure 11-18. Format of SHEMA Association Block

Word 0: SX = 1 if an extra word was allocated in XSAM for the SAB

Bits 0−14 contain the XSAM address of the next SAB in the list for the associated program, or zero if this is the last SAB for the program.

Words 1−8: Each word contains the XSAM address of a SHEMA Table Entry that the associated program has attached to itself, or zero if the SAB entry is unused.

At startup of a Large or Extended E/VMA program, the $SHEMA ID segment extension word points either to the SHEMA table entry for the primary SHEMA of a SHEMA-only program, or is zero if local E/VMA used. In either case, the SA bit of the $SHEMA word is clear.

The first time the RteAllocShema routine is called to allocate a secondary SHEMA to the program, RteAllocShema:

1. Allocates an SAB. All 8 SHEMA table pointer words are zeroed.
2. If the $SHEMA word is non-zero then that value is copied into the first SHEMA table pointer word of the SAB (word 1).
3. The pointer to the new SHEMA table entry being allocated is placed in the next available SHEMA table pointer word in the SAB.
4. The $SHEMA word becomes a pointer to the SAB with the SA bit set. Subsequent allocations fill successive words in the SAB until all 8 entries are used, at which time another SAB is allocated and linked to the old SAB.

Cartridge Directory

The Cartridge Directory contains disk cartridge identification information for all of the currently mounted disks. It is mainly useful for determining the search order for FMGR disks and for determining what disks were mounted when the system was last running. This table can be read by using the FMGR subroutine FSTAT.

This table is maintained in the first two blocks of the swap file. It has room for only 63 disk LUs; the last entry is never used, and marks the end of the table. Disk LUs with FMGR directories appear as indicated in this table; disk LUs in the hierarchical file system appear with cartridge number of zero. The cartridge directory format is shown in Figure 11-19.

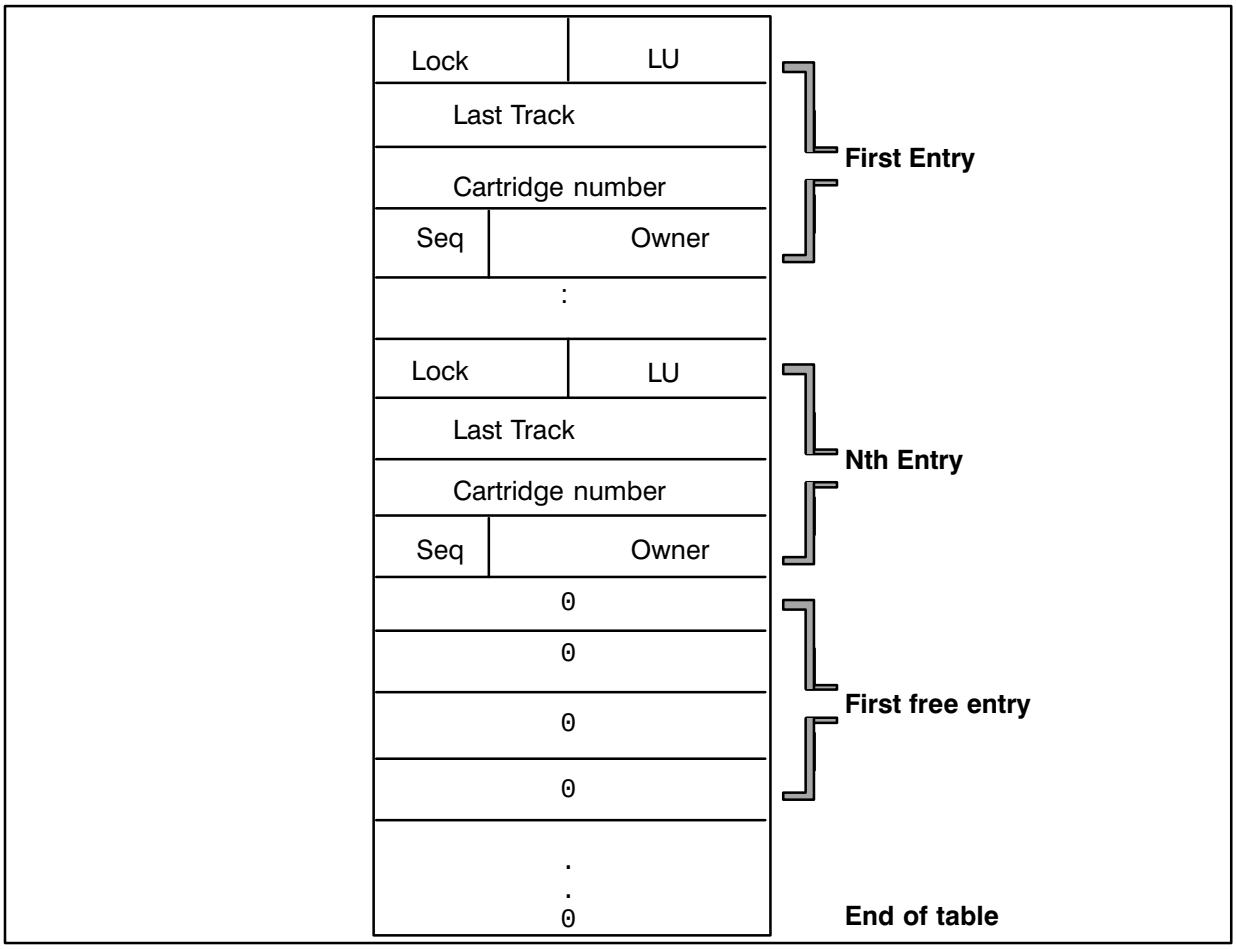


Figure 11-19. Cartridge Directory Format

A disk cartridge is temporarily locked during a mount, dismount, pack, or initialize operation. If a cartridge does not have a valid directory on the last track when it is mounted, it remains locked until it is initialized or dismounted.

The lock flag is an eight-bit field containing the ID segment number of the program that has the LU locked, or zero if the LU is not locked. The sequence number field is a four-bit field containing the sequence number for the locking program; it is ignored if the LU is not locked.

Memory Descriptors

In the area of memory set aside for running user programs, a block of memory is any series of contiguous pages. These blocks of memory are fixed in size, location, and number in the reserved partition area, but variable in size, location, and number in the dynamic memory area.

To keep track of the user program area, the operating system sets up, for each block of memory, a Memory Descriptor (MD) in the reserved-partition MD table or one of the dynamic MD lists. The system allocates dynamic MDs as they are needed to describe the dynamic memory area. Space for memory descriptors is allocated by RTAGN.

The reserved partition MD table consists of four-word descriptors containing the size, location, and status of the reserved partitions. The dynamic MD lists are linked lists of seven-word descriptors containing the size, location, and status of blocks of dynamic memory. A reserved partition MD is made up of words 0, 1, 2, and 3 of the dynamic MD.

Memory Descriptor Variables

System variables contain information the system needs to manage reserved and dynamic memory. System variable $RPTN heads the reserved-partition MD table, while $MEM heads the dynamic adjacency-MD list, which describes the status of all non-reserved user program memory.

Dynamic memory is further described by the free-MD list headed by $FREM. All memory descriptors not currently in use to describe a memory block are linked together in the unused-MD list, headed by $UMD. The remaining MD system variables provide summary information:

$RPT# = Number of reserved partitions.

$LGST = Size of current largest free block in the free list. Zero implies that the size of the current largest block is unknown.

$LGS1, $LGS2, $LGS3 = Sizes of the three largest available blocks of dynamic memory. The system uses the information in these variables and in variables $STL1, $STL2, and $STL3 to perform deadlock avoidance checks.

$STL1, $STL2, $STL3 = Starting pages of the blocks described by $LGS1, $LGS2, and $LGS3.

$RPTN = Reserved partition MD table head. Reserved partition MDs are contiguous in memory, starting at $RPTN.

$FREM = Free-MD list head; points to some MD in the free list; linked through next-free-block pointer entry of each MD in the list, and back linked through the previous-free-block pointer entry of each MD in the free-MD list. If $FREM is equal to zero, there are no free blocks of dynamic memory.

$MEM = Adjacency-MD list head; linked through the next-adjacent-block pointer entry of each MD; back-linked through the previous-adjacent-block pointer entry in each MD; list is in order of position of blocks in memory. Zero implies that there is no dynamic memory.

$MDS# = Total number of dynamic MDs.

$UMD = Unused-MD list head; points to some dynamic memory descriptor that is not in use, and therefore is available to describe a new block of dynamic memory set up by the system. Word 0 of each unused MD points to another unused MD, with word 0 of the last unused MD set to zero. Thus, $UMD points to a singly-linked list of unused memory descriptors.

$RMDS = Size of reserved MD in words.

$SZDY = Number of pages of memory in the dynamic area.

Dynamic Memory Descriptors

The dynamic MD lists are the adjacency list, the free list, and the unused list. Each dynamic MD, describing a block of dynamic memory, is in a linear doubly linked adjacency list headed by the pointer $MEM. This adjacency MD list is backward linked through the previous-adjacent-block pointer in each MD and forward linked through the next-adjacent-block pointer of each MD. Any two MDs that are next to each other in the adjacency-MD list describe blocks of memory that are also next to each other.

To enable the system to find blocks of free memory without having to scan the entire adjacency-MD list, each dynamic MD that describes a block of free memory is also in a doubly and circularly linked free list headed by the pointer $FREM. This free-MD list is backward linked through the previous-free-block pointer of each MD in the free list, and forward linked through the next-free-block pointer of each MD in the free list.

Whenever the system needs a new MD for a newly created dynamic partition, it selects one from the unused-MD list, enters the data describing the partition, and links the MD into the adjacency list.

Each dynamic MD consists of seven words, where words 0, 1 and 2 describe the size, location, and status of the block of memory. If the block is free, word 2 is negative and points to the previous free block. Word 3 is the memory block information word. Word 4 is the next-free-block pointer, while words 5 and 6 are the adjacency-MD list pointers, in the format shown in Figure 11-20.

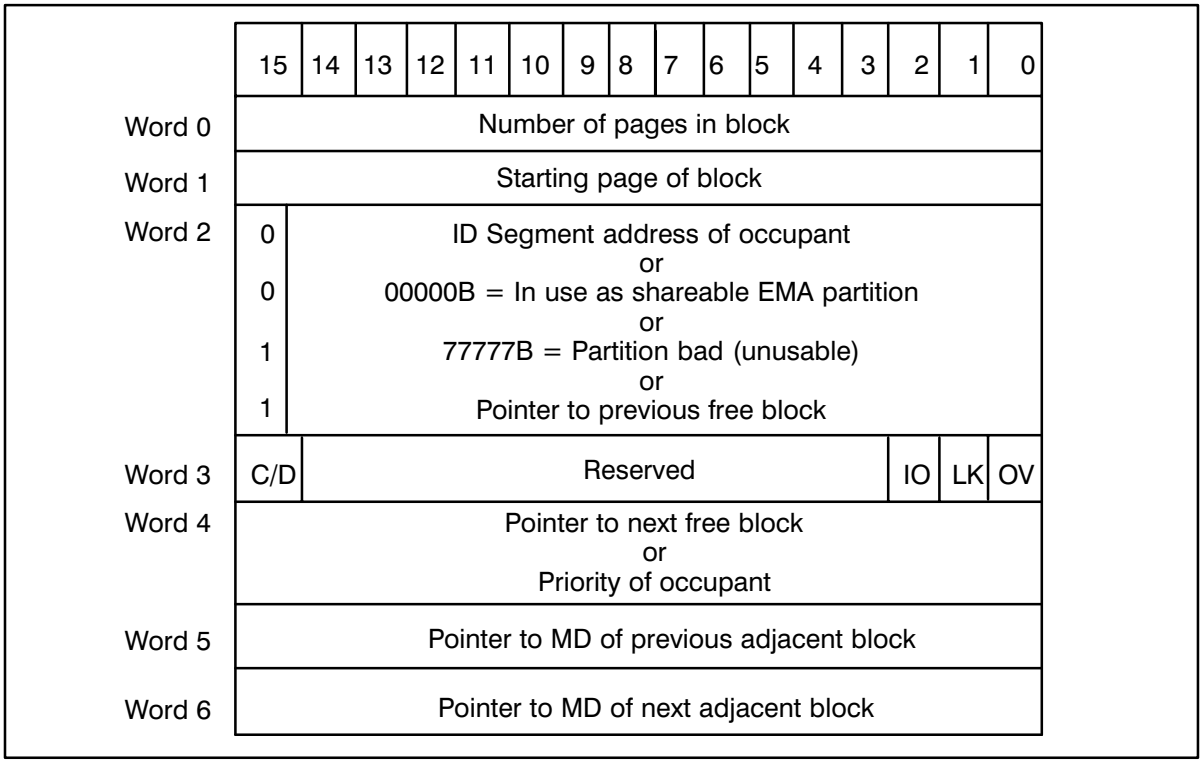


Figure 11-20. Dynamic Memory Descriptor Format

The entries of Figure 11-20 are described below:

Word 0: Number of pages in the block of memory.

Word 1: The starting page of the block.

Word 2: If the block described is occupied, then bit 15 is zero, and the remainder of the word points to the ID segment of the program occupying the block, unless the block is in use as a shareable EMA area, in which case the remainder of the word is set to all zeros.

If the value of the whole word is −1 (all bits set), then the block described is bad (parity error detected). If the sign bit is set but the remainder of the bits are not all set, then the block is free (no occupant, not a shareable EMA partition, and not bad), and bits 0-14 point to the MD describing the previous free block in the free memory list.

Note that if there is only one free block, then the MD describing this block has this word and word 4 pointing to itself, since the free list is circularly linked.

Word 3: Word 3 contains information about the state of the block described by the MD.

Bit 14 - 15 (CDS only) 00 if data partition
10 if code partition
11 if shared code partition

Bit 3 - 13 reserved

Bit 2 This bit is set if the partition is holding data, nonbuffered I/O is in progress and the partition must not be swapped.

For CDS programs, this bit is set if the partition is holding code, and has been overlaid and reloaded from the disk program file since last execution of program.

Bit 1 This bit is set if the partition is locked due to an EXEC 22 lock call, in use as shareable EMA, or the block is bad.

Bit 0 This bit is set if the partition is overlayable.

If no status bits (0-2) are set, the partition is swappable.

Word 4: If the block is free (word 2 negative, but not equal to −1) then word 4 points to the MD of the next free block of memory. If the block is occupied (word 2, bit 15 = 0), then the block is not linked into the free list, and word 4 is the priority of the program running in the block. If the block is bad (word 2 = −1), this word is meaningless.

Word 5: A pointer to the MD describing the adjacent block of memory in the backward direction. Zero in this word means there is no previous block of dynamic memory.

Word 6: A pointer describing the MD of the adjacent block of memory in the forward direction. Zero in this word means there is no next adjacent block of dynamic memory. Thus, the dynamic memory descriptors form a linear doubly linked list of blocks (free, allocated or bad) ordered by position of the blocks in memory.

Reserved Partition Memory Descriptors

The reserved partition memory descriptors (MDs) are maintained separately from the dynamic memory MDs. Forward and back link words (words 4, 5, and 6) of the reserved partition MDs do not exist. The format of the reserved partition descriptor is shown in Figure 11-21.

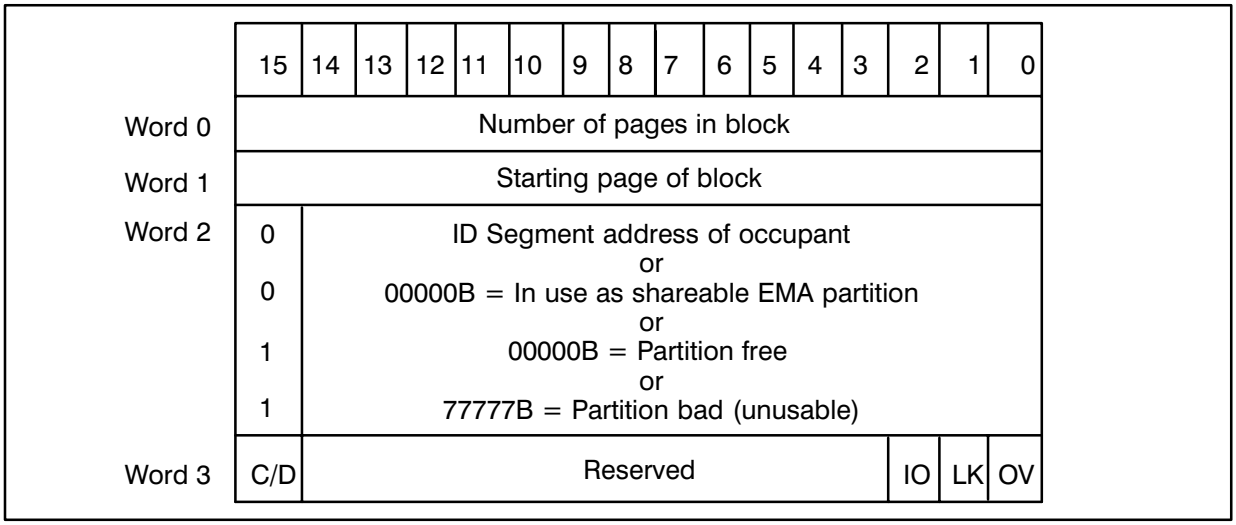


Figure 11-21. Format of Reserved Partition Memory Descriptor

There is no explicit linking of reserved partition MDs. They lie in contiguous memory starting at $RPTN. They are referenced by number, where the number describes both the position of the reserved partition MD in the reserved partition MD table and the position of the block of memory described, with relation to the other reserved partitions.

Thus, the sequential number of a given reserved partition is the same as the sequential number of its MD. For example, if the operating system ends at physical page 31, and there are two reserved

partitions, each five pages in size, then reserved partition 1 will be pages 32 through 36, and will be described by the first MD in the reserved partition MD table. Similarly, reserved partition 2 will be pages 37 through 41, and will be described by the second MD in the reserved partition MD table.

On bootup, immediately following the last reserved partition MD will be a MD for the first block of dynamic memory. Continuing the example above, initially there will be only one dynamic memory MD, describing a free block starting at page 42 and extending to the end of physical memory. $FREM and $MEM will both point to this MD; the adjacent pointer in this MD will be zero; the free list pointers will both point to the MD itself.

Immediately following the free memory MD will be a linked list of unused MDs, headed by $UMD, linked through the word 0 of each of the MDs in the list, and terminated by a link word of zero.

By default, the total number of MDs is at least one more than four times the number of ID segments for system using CDS programs (and one more than twice the number of ID segments for systems using non-CDS programs). This is done to keep track of code, data partition, and holes between them. Note, however, that this is guaranteed to always be enough MDs only if no reserved partitions are defined, there are no bad blocks, and there is no shareable EMA allocated.

Shared Program Table

The shared program table are used to implement shared programs. Space for the shared program table is reserved at system generation when the maximum number of shared programs is specified. Each set of shared programs has its own shared program table entry. The format of the table is shown in Figure 11-22.

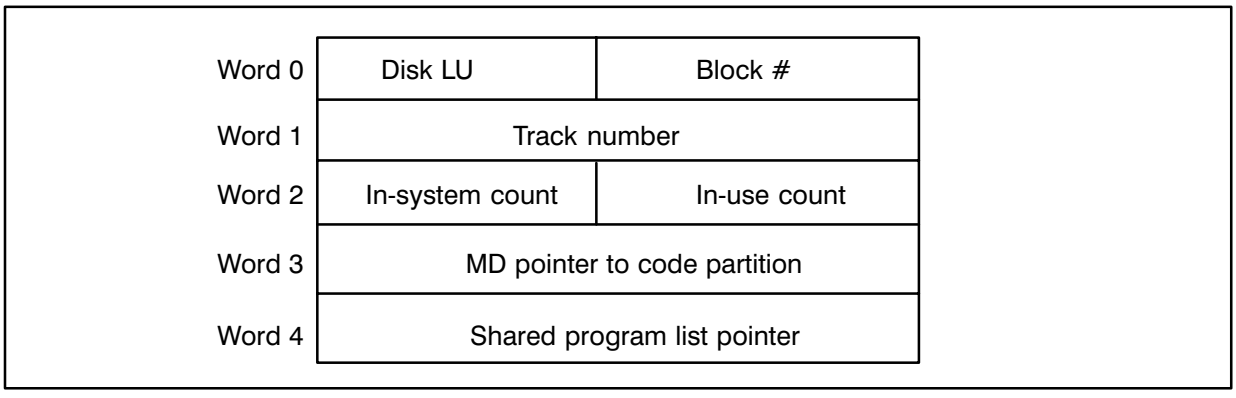


Figure 11-22. Shared Program Table Format

The address of the first shared program table entry is in $SPTB and the number of shared programs is in $SPR#. The table entries are as follows:

Words 0-1: These two words contain the program disk address that uniquely identifies the program. The disk LU is bits 8-15 and the block number is bits 0-7 of word 0.

Word 2: In-system count (bits 8-15) is the number of existing ID segments that are related to this shared program.

In-use count (bits 0-7) is the number of active program related to this shared program (active programs are either non-dormant or dormant after terminating saving resources).

Word 3: Pointer to the memory descriptor (MD) for the code partition for this shared program. If 0, a code partition has not been allocated.

Word 4: Pointer to the first program in the linked list for this shared program entry. The linked list uses word 22 of the ID segment for CDS programs (see Figure 11-2).

Multiuser Table

The following paragraphs describe the table and files used in the multiuser environment in RTE-A systems with VC+. The information provided includes the user ID table, the user configuration file, and the master account file. If your system does not have the VC+ option, skip these paragraphs.

User ID Table

The user ID table is a memory-resident table containing all pertinent user identification data. It keeps a record of all users logged on to the system. Each time a user logs on, an entry is made to this table. When that user logs off the system, the entry is released for use with another user that is logging onto the system.

The user ID table is associated with the following system variables:

$UIDA - Address of the first word of the user ID table. Note the first entry is reserved for the system.

$UID# - Number of user ID entries

$USZ - Size of user ID entry

$#CUS - Number of currently logged on users

$OWNR - Word 40 of the program ID segment points to the first word of the user entry

The format of the user ID table entry is given in Figure 11-23. The content of the user ID entry is described on the following diagram.

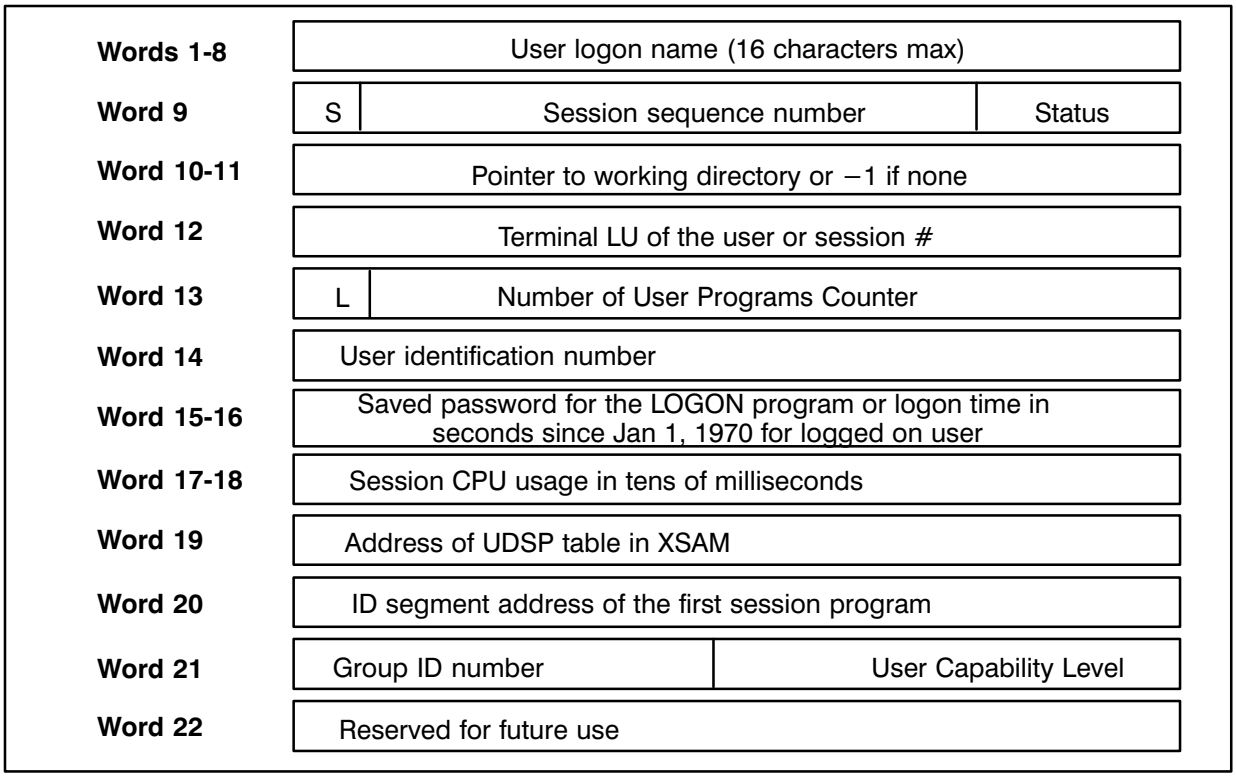


Figure 11-23. User ID Table Entry

Words 1-8: User log-on name, 16 characters maximum; matched with the user configuration file and filled into the table by LOGON.

Word 9: A flag word obtained from the user configuration file and filled into the table by LOGON.

S = superuser bit (bit 15); set if this entry is for a superuser.

Session sequence number (bits 4-14) is used to identify the session

Status bits (0-3) are filled in by LOGON

0 = this is a free entry
1 = non-interactive session
2 = active session
3 = this session set up programmatically.
Program counter (word 13) may be zero.
4 = transition state, waiting for password

Words 10-11: Working directory pointer; a double integer calculated by the LOGON program.

Word 12: The session number is calculated by the LOGON program and filled into the table.

Word 13: L = Logoff program/command file bit (bit 15); set if there is a logoff program or command file defined for this entry. The user programs counter is incremented

when a program is scheduled and decremented when a program becomes dormant (bits 0-14).

Word 14: The user identification number assigned by the system is obtained from the user configuration file.

Words 15-16: The user password obtained from the configuration file is temporarily saved here if the password is not entered with the user's name. The LOGON program prompts for the password and compare the two values. If the values are the same, words 15 and 16 are replaced with the current time in seconds since January 1, 1970.

Words 17-18: These words contain the value of this session's CPU usage in tens of milliseconds. When a program from this session terminates, its CPU usage is added to this value.

Word 19: The address of the User-Definable Directory Search Path (UDSP) table in XSAM. LOGON allocates space for the table in SAM when a user logs on. If there is not enough XSAM for the table, this word is set to zero. This address is also used to locate the LU access table, which sits directly above the UDSP in XSAM.

Word 20: The address of the ID segment of the program that was executed when a user logs on; filled in by the LOGON program.

Word 21: The user capability level (bits 0-4) and the group ID (bits 5-15) of the group with which the user is associated for this session.

Word 22: spare

Initial Entry

Data in the user ID table is obtained initially from the user logon entry and then from the group and user configuration files set up in the system. Entries are made to the table by the LOGON program.

The logon name supplied by the user is matched with the user configuration file and filled into the table by the LOGON program. LOGON also performs the following functions:

- The group configuration file is opened using the supplied group name or the default group name (if no group name is supplied) in the user configuration file.
- The group resource limits are checked to make sure they have not been exceeded.
- The session LU access table is constructed by computing the inclusive OR of the user and group access tables. The terminal LU must be in the session LU access table for the user to log on.
- The resource limits for the user in the group are checked to make sure they have not been exceeded.
- After the user's password is checked, the session's logon time is placed in words 15 and 16.
- Information obtained from the user configuration file is filled into the flag word (word 9).
- The appropriate status bits are set.

- The pointer to the working directory is calculated and written into words 10 and 11. This pointer is also modified by the file system when the working directory is changed.
- The terminal LU is placed in word 12.
- The user program counter is incremented when a program is scheduled or time scheduled; and decremented when a program becomes dormant or leaves the time scheduled list.
- The user identification number assigned by the system is placed in word 14.
- A chunk of XSAM is allocated for the UDSP table and the LU access table.
- The starting address of the UDSP table is placed in word 19. This address is used to access both the table and the access table.
- Word 21 is filled with the user's capability level, bits 0-4, and the group identification number assigned by the system, bits 5-15.

LU Access Table and UDSP

Space for the LU access table and the User-definable Directory Search Path (UDSP) table is allocated in XSAM by LOGON when a user logs on. The LU access table sits above the UDSP table. The starting address of the UDSP table is placed into word 19 or the user ID table entry. The size of the table is determined by word 20 in the user configuration file. The format of the UDSP table is shown in Figure 11-24.

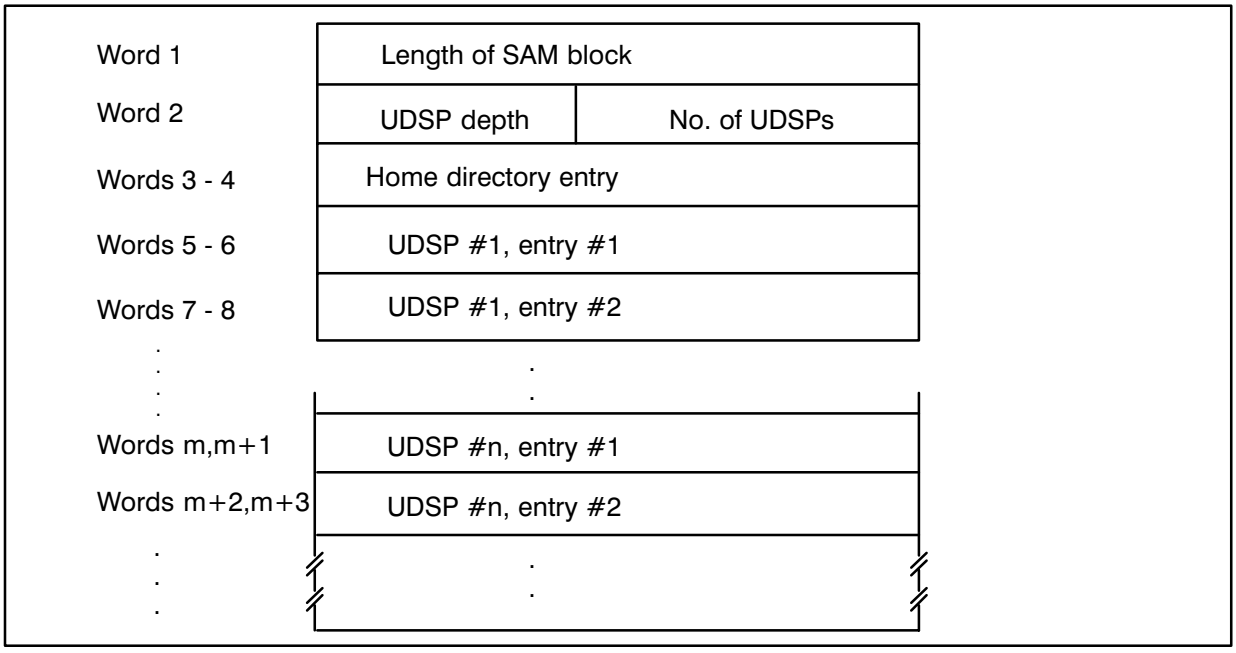


Figure 11-24. UDSP Table Format

Word 1: Amount of SAM allocated for the UDSP table

Word 2: UDSP depth (bits 8-15) is the maximum number of entries the user can define in a User-Definable Directory Search Path

Number of UDSPs (bits 0-7) is the maximum number of User-Definable Directory Search Paths that the user can define

Words 3-4: Two word pointer to the home directory (UDSP #0)

The remaining entries in the table (word 5 through the end of the table) are two-word directory pointers to a specific UDSP entry. For example, words 5 and 6 are a two word pointer to entry one of UDSP #1.

Space for the Session LU access table is allocated in XSAM above the UDSP table when a user logs on. The pointer to the UDSP table in word 19 of the user ID table entry is used to located the LU access table (address of the UDSP table minus sixteen). LOGON creates the session LU access table by performing an inclusive OR of the user and group access tables with which the user associated at logon. It is comprised of 16 words where bit 0 of word 1 corresponds to LU 0, bit 0 of word 2 corresponds to LU 16, bit 0 or word 3 corresponds to LU 32, and so on.

The total length of the XSAM block is 16 words for the LU access table plus the value in word 1. The length stored in word 1 is determined by the following calculation.

length = (number of UDSPs * depth * 2) + 4 words

Note that the length that is stored may be one word larger than is indicated by the above calculation because of the way XSAM is allocated.

If the account does not have a UDSP defined, the UDSP table consists of the first four words. If LOGON cannot find enough XSAM to build the complete table and LU access table, it tries to build the minimum sized table (4 words) and access table (16 words) and outputs a warning message. If this occurs, only the home directory can be used.

If there is not enough XSAM for a 4-word table and 16-word access table, the user is informed and the session is aborted (that is, logon is denied). The singled exception is user MANAGER>SYSTEM. If there is not enough XSAM for the minimum table and access table when MANAGER.SYSTEM logs on, LOGON does not try to build the table and access table. The address word (word 19) in the user ID table entry is set to zero. This is interpreted as the user having no working directory and access to all LUs.

Use of ID Table

The user ID table is used by the operating system and the file system. Other system programs and user written utilities may also access the appropriate data required in this table; for example, the WHZAT program. The operating system uses the following information:

Superuser (bit 15, word 9)
The number of user programs (word 13)
The status bits (bits 0-3, word 9)
CPU usage (words 17-18)
Address of UDSP table (word 19) is used to locate the LU access table
User capability level (bits 0-4, word 21)

The file system uses the following information:

User identification number (word 14)
Default directory (words 10, 11)

Superuser (bit 15, word 9)
UDSP table address (word 19), UDSP table, and LU access table
Group identification number (bits 5-15, word 21)
User capability level (bits 0-4, word 21)

User ID Table Modification

The user ID table entries are modified only when certain data is being changed. As the number of user programs changes, the operating system changes the user program counter (word 13). The file system, the program D.RTR, and the operator working directory (WD) command can change the default directory field (words 10 and 11). When a program terminates, the system adds the amount of CPU time used by the program to the user's accumulating CPU usage (words 17-18).

Use of LU Access Table

The LU access table is used by the file system and the operating system. The SESLU utility is used to modify entries in the access table. D.RTR uses the access table to determine whether or not a user has access to an LU. For all I/O requests, the modules IORQ and LOCK in the operating system check the access table to determine whether the user has access to the requested LU.

Use of UDSP Table

The User-definable Directory Search Path (UDSP) table is used by the file system, primarily by PATH and D.RTR. The PATH command modifies entries in the table and D.RTR uses the table to search for a file using a defined UDSP.

User Configuration File

The user configuration file is a type 1, fixed record length (128 word), extendible file that contains information about a user that is defined in the system and can log on. The name of the file corresponds to the user's logon name. The file resides on the USERS directory, which should be write protected for security reasons.

The information in the configuration file is filled in during the creation of a new user account by the System Manager, a superuser (Security/1000 is turned off), or a user with the required capability (Security/1000 is turned on). The user identification number is assigned by the system when the account is created.

Information in the configuration file may be modified by the owner of the account, the System Manager, a superuser (Security/1000 is turned off), or a user with the required capability (Security/1000 is turned on). If Security/1000 is not being used, the owner is allowed to modify all user configuration file information except (if not a superuser) the superuser bit and the accounting limits. If Security/1000 is turned on, the owners' capability levels determine what information can be modified.

	Word	15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 Content
Block 1	1-15	User's Real Name (30 characters max)
	16	S | Reserved for future use
	17-18	Encoded Password (14 chars max) or a constant
	19	User Identification Number
	20	UDSP depth | Number of UDSPs
	21-60	Program to run at logon (80 chars max)
	61-92	Working Directory name (64 chars max) or −1
	93-94	Last logoff time, in seconds since Jan 1, 1970
	95-96	Cumulative connect time in seconds
	97-98	Cumulative CPU usage in tens of milliseconds
	99	Reserved for future use | User capability level
	100-115	LU access table
	116	Block number of the first block in the user's group list
	117	Number of entries (records) in the group list
	118	Index from the base of the group list to the record of the default logon group
	119-120	Last Logon Time, in seconds since Jan 1, 1970
	121	Group ID number last logged on with
	122	LU last logged onto
	123	| EVB size in pages
	124 to 126	Reserved for future use
	127	Conversion word; (−1 if file in new format)
	128	Check sum of all previous words
Blocks 2-N	1-8	Group Name (16 characters max) -or- Word one equals 0; signals empty block
	9-10	Cumulative CPU usage in tens of milliseconds for the user in this group
	11-12	Cumulative connect time in seconds for the user in this group
	13-14	CPU usage limit for the user in this group
	15-16	Connect time limit for the user in this group
	17-56	Program to run at logon (80 characters max)
	57-96	Program/command file to run at logoff (80 characters max)
	97-128	Working directory name (64 chars max) or −1

Figure 11-25. User Configuration File

Block 1 (Unique User Information)

Words 1-15:	User's real name (30 characters maximum)
Word 16:	S = superuser bit (bit 15); set if the entry is for a superuser. Reserved for future use (bits 0-14)
Words 17-18:	User's password (16 characters maximum) encoded in 32 bits
Word 19:	The user identification number assigned by the system
Word 20:	UDSP depth (bits 8−15) is the maximum number of entries the user can define in a User-Definable Directory Search Path. Number of UDSPs (bits 0−7) is the maximum number of User-Definable Directory Search Paths the user can define.
Words 21-60:	Runstring of the program to be executed when the user logs on associated with the group NOGROUP (80 characters maximum)
Words 61-92:	Directory to be the user's working directory at logon (64 characters maximum) or −1 if no working directory at logon when the user associates himself with the group NOGROUP
Words 93-94:	User's last logoff time in seconds since January 1, 1970 (part of multiuser accounting information)
Words 95-96:	Cumulative logon time in seconds for the user in all groups (part of multiuser accounting information)
Words 97-98:	Cumulative CPU usage in tens of milliseconds for the user in all groups (part of multiuser accounting information)
Words 99:	User capability level for the user in all groups he/she is a member of (bits 0-4) Reserved for future use (bits 5-15)
Words 100-115:	LU access table; used to limit user's access to LUs
Word 116:	Block number of the first block in the user's group list (this allows for more user description blocks if they are ever needed)
Word 117:	Number of entries (records) in the user's group list. This includes used as well as empty records
Word 118:	Index from the base of the group list to the record of the default logon group
Words 119-120:	User's last logon time in seconds since January 1, 1970
Word 121:	Group Identification Number that the user was associated with at the last logon
Word 122:	LU that the user was last logged onto
Word 123:	The size of the user's Environment Variable Block (EVB) in pages (bits 0 − 5)
Words 124-126:	Reserved for future use
Word 127:	Code word; −1 signals that the User Configuration file has been converted to the new form
Word 128:	Checksum: sum of words 1 thru 127; used to ensure that this file is a user configuration file and has been tampered with

Block 2–N (User. Group Information)

Words 1-8:	Group name (16 characters maximum) to which user belongs - or - Word 1 = 0; signals that this is an empty entry
Words 9-10:	Cumulative CPU usage in tens of milliseconds for the user in this group (part of multiuser/group accounting information)
Words 11-12:	Cumulative connect time in seconds for the user in this group (part of multiuser/group accounting information)
Words 13-14:	CPU usage limit in tens of milliseconds for the user in this group (part of multiuser/group accounting information)
Words 15-16:	Connect time limit in seconds for the user in this group (part of multiuser/group accounting information)
Words 17-56:	Runstring of program to be executed when the user logs on associated with this group (80 characters maximum)
Words 57-96:	Program or command file to be run when the user logs off from a session associated with this group (80 characters maximum)
Words 97-128:	Directory to be the user's working directory at logon when he associates himself with this group (64 characters maximum)

Group Configuration File

The group configuration file is a type 1, fixed record length (128 words), extendible file that contains information about a group that is defined in the system. Block one contains the definition of the group's resources. Blocks 2 through N contain records that form a list of the group's members. Each user record contains the user logon name of the member and an index into the group records in the user's configuration file, which enables the system to locate the appropriate group record quickly. If word one of the eight-word user record is 0, the record is empty. The group configuration file resides on the USERS directory and its name corresponds to the group's logon name with the type extension .GRP to distinguish it from the user configuration files on USERS.

The information in the group configuration file is filled in by the System Manager, a superuser (Security/1000 is turned off) or a user with high enough capability (Security/1000 is turned on) during the creation of anew group account with the GRoup and User Management Program (GRUMP). When the new group is created the system assigns the group identification number using the GROUPACCOUNT file. Information in the file may be modified through GRUMP by the System manager, a superuser (Security/1000 is turned off), or a user with high enough capability (Security/1000 is turned on).

Block 1

Word	Content
1	Group Identification Number
2-3	Cumulative CPU Usage in Tens of Milliseconds
4-5	Cumulative Connect Time in Seconds
6-7	CPU Usage Limit in Tens of Milliseconds
8-9	Connect Time Limit in Seconds
10-25	LU access table
26	Block number of the first block in the group members list
27	Number of entries in the list of members
28 : 127	Reserved for future use
128	Checksum of all previous words

Block 2-N

Word	Content
1-8	User Logon Name of Member 1 (16 Chars Max) - or - Word one equals 0; signals an empty record
9	Index into Member 1's User Configuration File group list
10-17	User Logon Name of Member 2 (16 Chars Max) - or - Word one equals 0; signals an empty record
18	Index into Member 2's User Configuration File group list
:	
127-128	Reserved for future use

Figure 11-26. Group Configuration File Format

Block 1

Words 1: The group identification number assigned by the system. This can be a maximum of 2047, with Group ID 0 always being that of NOGROUP and ID 2 being that of the group SYSTEM.

Words 2-3: A cumulative total of CPU usage in tens of milliseconds for the whole group. (part of group accounting information)

Words 4-5: A cumulative total of connect time in seconds for the whole group. (part of group accounting information)

Words 6-7: CPU usage limit in tens of milliseconds for the whole group. (part of group accounting information)

Words 8-9: Connect time limit in seconds for the whole group. (part of group accounting information)

Words 10-25: LU access table; enables group members to access LU's/devices.

Word 26: Block number of the first block in the group members list.

Word 27: Number of entries in the list of members. (includes used and empty records)

Words 28-127: Reserved for future use

Word 128: Checksum: sum of words 1 through 127; used to ensure that this is a group configuration file and has not been tampered with

Blocks 2-N

Each block contains a maximum of 14 user records. Words 127 and 128 are not used; thus no user records span block boundaries.

User records are 9 words consisting of:

Words 1-8: User logon name of member (16 characters maximum)

Word 9: Index into members user configuration file group list (from the base of the list). Used to find the record in the user configuration file corresponding to this group quickly

- or -

Word 1: equals 0; signals that it is an empty record

MASTERGROUP File

The MASTERGROUP file is a protected file on the USERS directory containing system information and the logon name of all the groups on the system. This is a type 2, fixed record length (8 words), extendible file. The first record contains the last group identification number assigned by the system. The second through the last record contain the logon name of each group defined in the system. A group's identification number is its corresponding record number in the MASTERGROUP file, with the maximum group ID number being 2047. The contents of the record is the group's logon name (maximum of 16 characters). This file makes determination of the group logon name from the group ID number easy. The format of the MASTERGROUP file is shown below.

Record	Content
1	Word 1 contains the last assigned group ID
2	Logon name corresponding to group ID 2
:	
n	Logon name corresponding to group ID n

Figure 11-27. MASTERGROUP File Format

Note:

1. Group ID number zero is always associated with the group NOGROUP.

2. Group ID number two is always associated with the group SYSTEM.

3. The maximum group ID number is 2047. They can be reused but this should be done with caution, because there is no utility at the present time to go out and remove the group ID, of the purged group, from the file system. Thus, a new group could inherit the file access rights of the group that previously had the group identification number, which were not intended for the new group.

   At the present time, bit 15 of word 1 the record in the MASTERGROUP file is set when the group with that identification number is purged and that flags the group ID as reusable. Note that this may change in the future.

MASTERACCOUNT File

The MASTERACCOUNT file is a protected system file on the USERS directory. This file is created by GRUMP during the initialization phase. It contains the logon names and corresponding user IDs for all users on the system. It is a type 2, fixed record length, extendible file. The format of the MASTERACCOUNT file is shown in Figure 11-28.

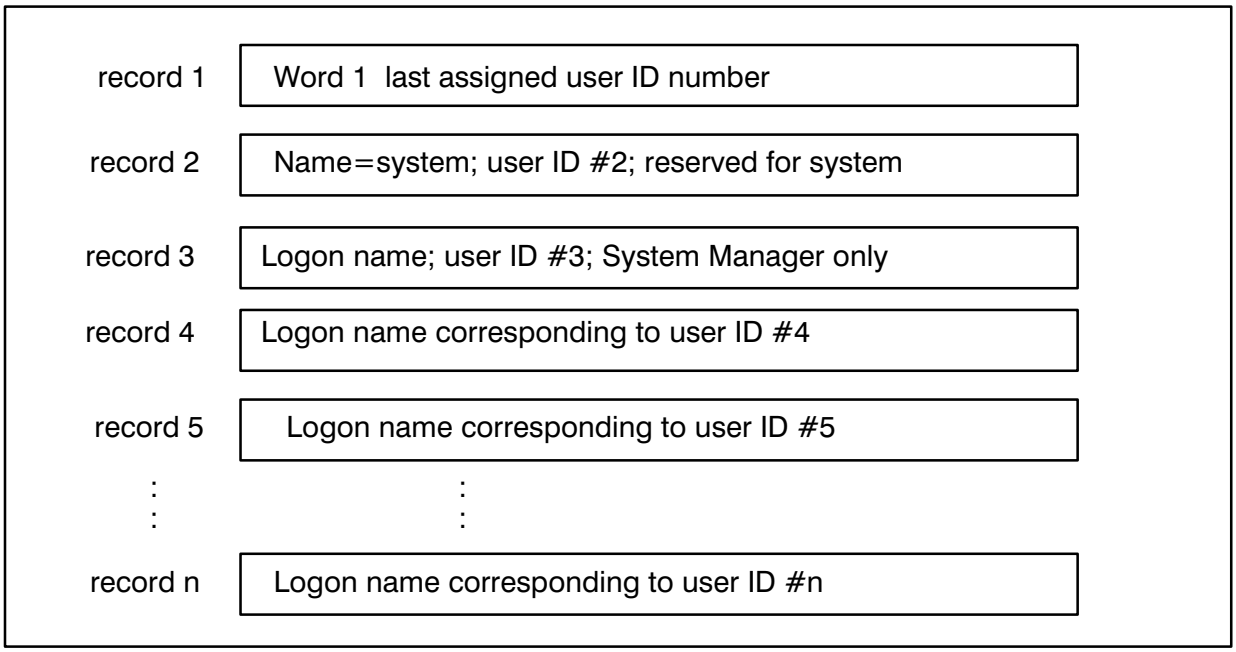


Figure 11-28. MASTERACCOUNT File Format

The first record contains system information. Each of the remaining records contains the logon name of a system user. The user identification number is the record number in the MASTERACCOUNT file. Each record contains the user logon name (maximum of 16 characters). For example, the logon name of user ID 26 is in record 26 of the MASTERACCOUNT file. Each user is assigned a unique ID number when the account is

created. It is therefore easy to determine owner names through the MASTERACCOUNT file. This file is never searched because users may have been deleted and new users aborted at creation. If there is a need for determining the name of any user, use the user ID number. Subroutine IDtoOwner can be used for this purpose.

The second record is reserved for system use and has the name SYSTEM.

The third record is reserved for the System Manager and has the name MANAGER. MANAGER is the first user created by the GRoup and User Management Program during user account initialization. The remaining records are used for other system users.

CDS Tables

In RTE-A systems with VC+, a Code Segment Table (CST) and a Segment Replacement Table (SRT) are contained in page 0 of the user partition. The CST and SRT formats are shown in Figure 11-29.

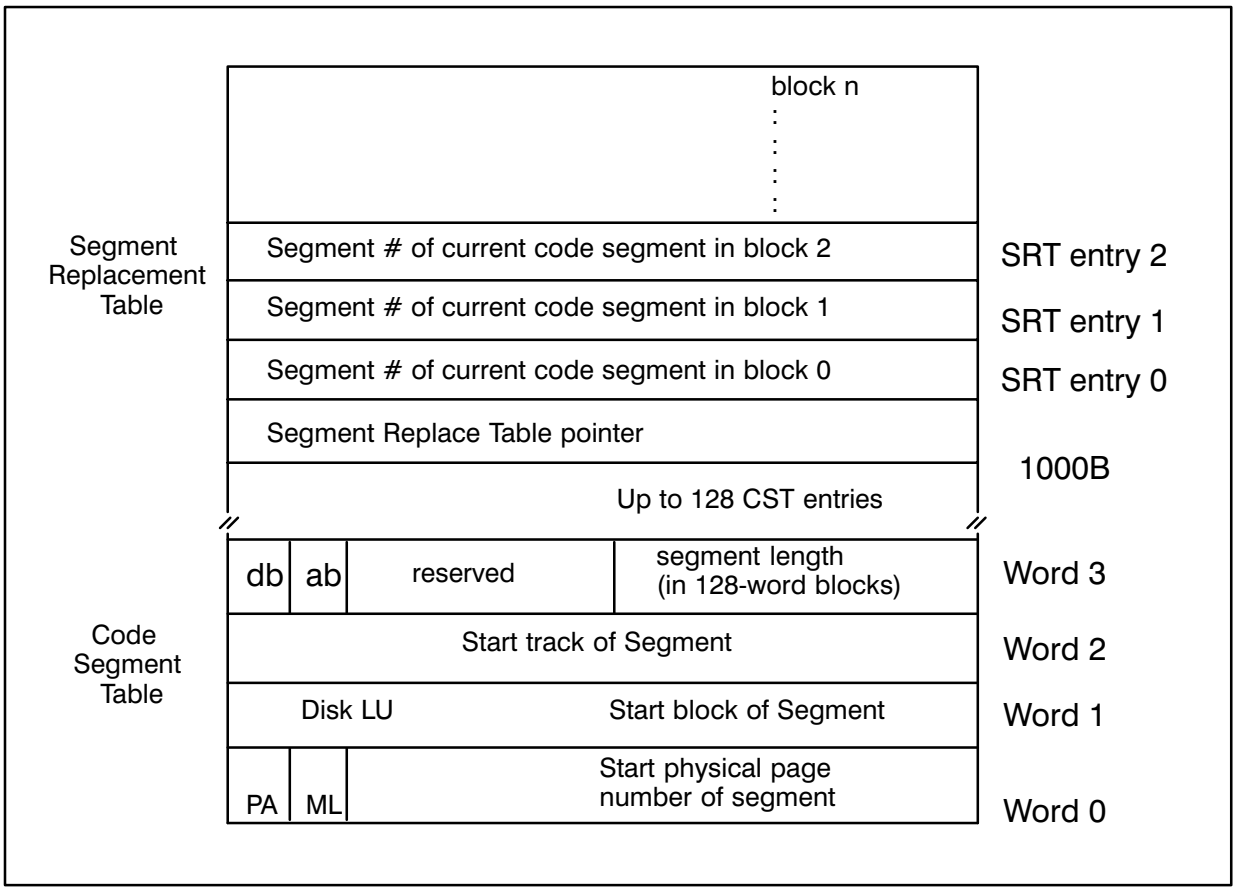


Figure 11-29. Code Segment and Segment Replacement Tables

In the Code Segment Table,

db = Bits 15 and 14 are used by the
ab Symbolic Debug/1000.

PA = Bit 15 is 0 if segment is in memory, 1 if segment is absent.

ML = Bit 14 is 1 if segment is locked (by the LINK ML command)

Language Message Address Table

$LMAT is a table of eight words containing the logical address pointers into the system message map (map 1) of the eight system message blocks relocated by the generator. $LMAT resides in module ERLOG. The pointer to the first word of the language message table is $LMA, which resides in module VCTR.

If less than eight system message blocks are generated into the system, the remainder of the table defaults to the logical address of system message block 0.

Figure 11-30 shows the format of the Language Message Address table.

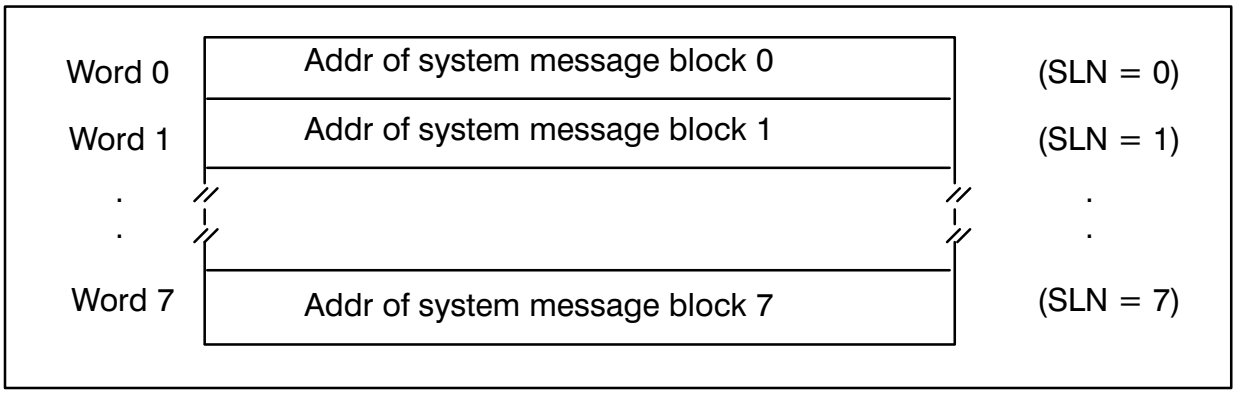


Figure 11-30. Language Message Table Format

12

FMP Tables

This chapter describes the various tables used by the File Management Package (FMP). Included are the formats for disk volume header, directories, subdirectories, Data Control Block (DCB), and other table entries. Also included in this chapter are those tables used by the FMGR files.

Disk Volume Header

The disk volume header is found in sector zero of the last track on the LU. It is used to indicate that this is a disk volume in the RTE-A file system and to store information pertinent to that physical portion of the disk. The format of the disk volume header is shown in Figure 12-1.

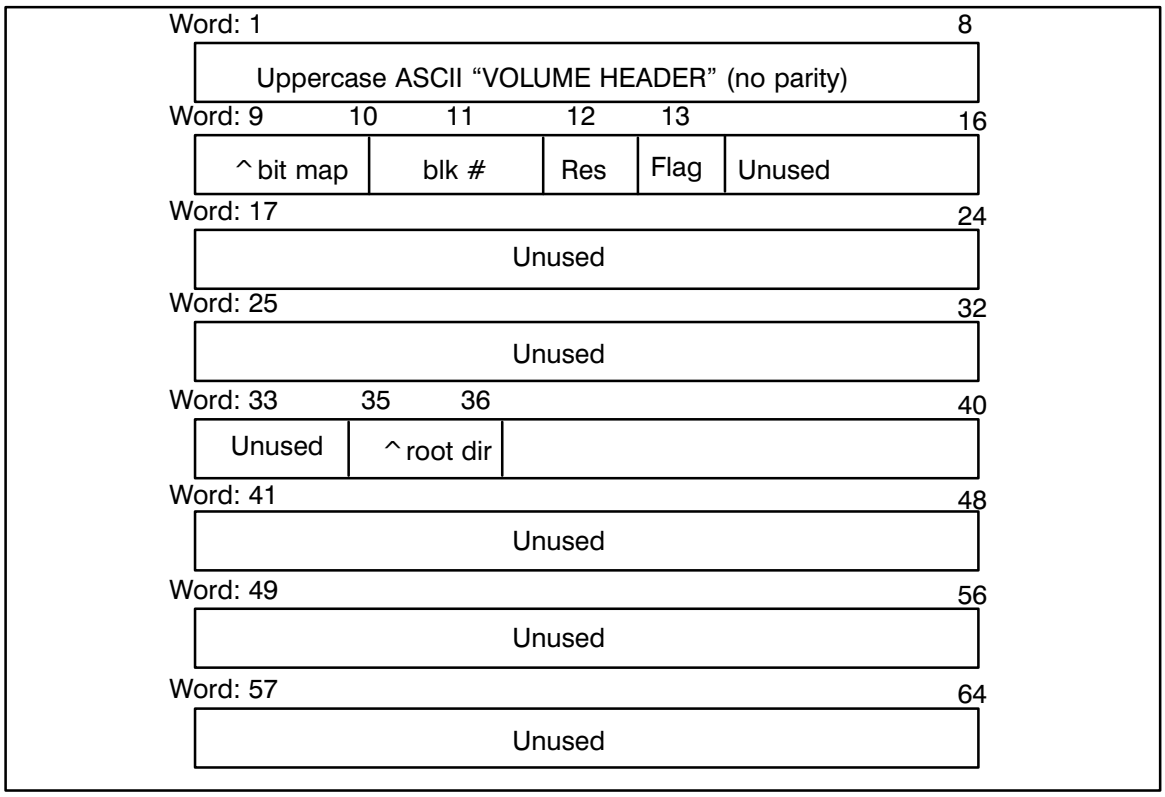


Figure 12-1. Disk Volume Header Format

The disk volume header has 64 words. The bit map field of the header (words 9 and 10) contains the block address of the bit map, bits 0 through 23 with remaining bits set to zero. Word 11 stores the number of blocks represented by a bit in the bit map. Word 12 is the reserve field that shows the number of blocks marked as reserved in the bit map. Only one reserved area can be defined, starting at block zero. These blocks are not referenced by any file.

Bit 0 of word 13 is a flag that defines the meaning of words 7 and 8 of the root directory header. When bit 0 is set, the root directory contains a pointer back to the volume header. Words 35 and 36 of the volume header contain the block address of the root directory, a directory that contains only unique global directories. This entry is used to find the root directory for that LU.

Directory Structure

Directories are linked sets of disk blocks that contain directory entries. Each directory entry consists of 32 words, and contains a flag to tell what type of entry it is. Note that any directory entry can be unused at any time. Directories are extendable, and extents are linked together as a doubly linked list. The directory structure shown in Figure 12-2 assumes the first part of the directory is n blocks big, and the first extent is x blocks big.

Note that directory entries in the root directory (and subdirectory entries anywhere) appear as files of type extension .DIR.

Directory searches are just linear searches that start at the beginning of the directory and continue until the file is found, or until an available entry or the last directory trailer is found. The search can pass over purged file and extent entries, and can cross extents.

There are some cases where the user will want the path name that corresponds to a given directory address. This can be done by doing a backward search of the directory to get the directory name, and following the subdirectory chain if necessary.

The directory structure is shown in Figure 12-2. In the illustration, the first word of each entry is the flag word, and bits 0-3 define a type code for this entry:

0: empty; no files follow
1: directory header
2: directory trailer
3: file entry
4: extent entry
5: purged file entry
6: purged extent entry
7: empty, but files may follow
8-15: not currently used

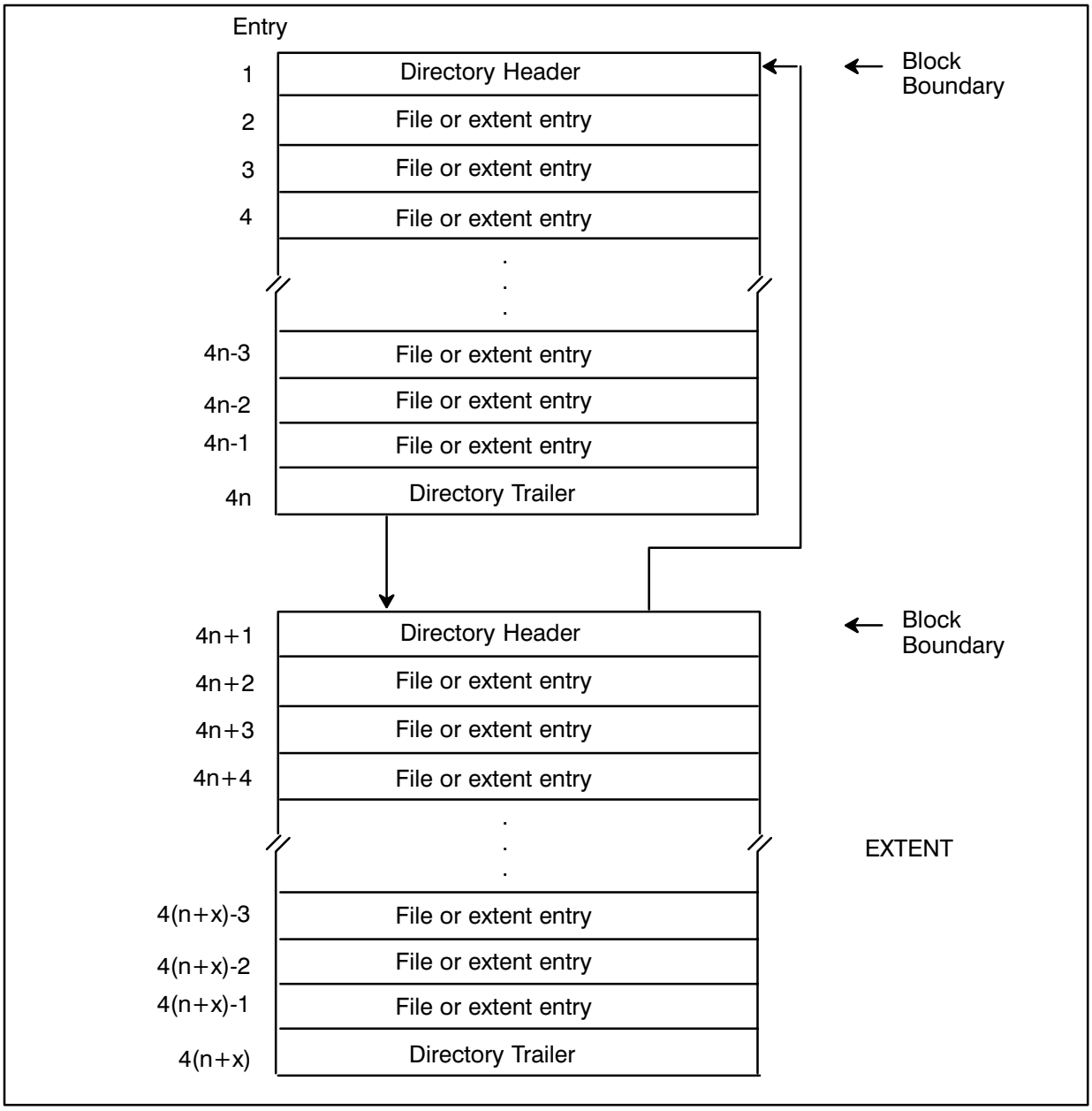


Figure 12-2. Directory Structure

Root Directory Header/Trailer

The root directory contains the global directories found on an LU. The root directory header is the same as any other directory header except that fields, such as owner, that have no meaning are left zero.

There is no limit on the number of global directories other than the limits of D.RTR (see Chapter 10, “Directory Organization”); the global directory table on disk is expanded as necessary to accommodate the additional entries in the same way that any other directory is expanded.

The format of the root directory header/trailer is shown in Figure 12-3. The flag word indicates if the entry is a header (flag = 1) or a trailer (flag = 2). In addition, the flag word contains protection bits (see explanation under Directory Header/Trailer). The size word contains the number of blocks in this root directory table. The tag field (words 3 and 4) contains a unique bit pattern to help recreate the disk.

Words 5 and 6 is a field that contains a null (−1) for the first header and the last trailer, otherwise it is the block address of the previous header or next trailer. The block address is contained in bits 0 through 23 of this field.

When bit 0 of word 13 of the volume header is set, words 7 and 8 contain the negative block address of the volume header. If bit 0 is not set, words 7 and 8 are not used.

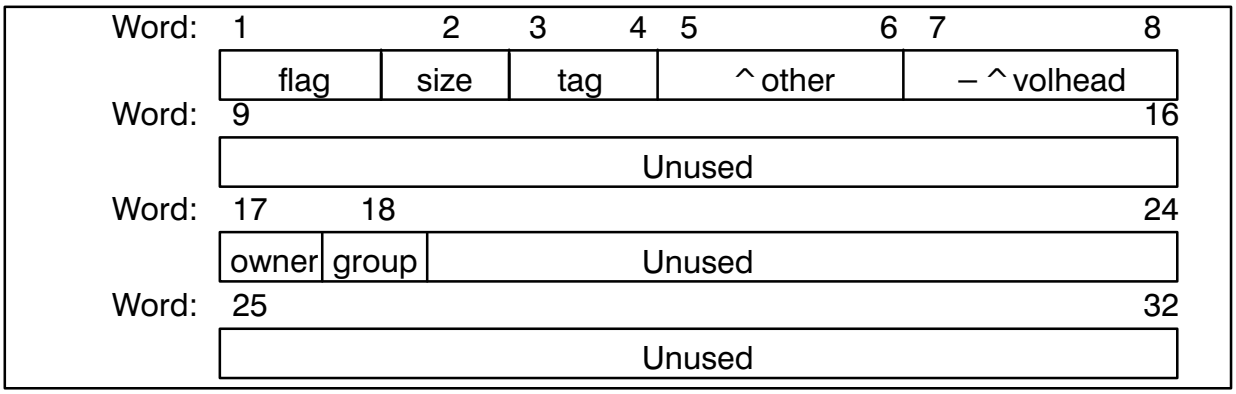


Figure 12-3. Root Directory Header/Trailer

Root Directory Entry

Root directory entries associate directory names with the disk address of the directory. All entries in the root directory are global directories for that LU. These entries are similar to subdirectory and file entries. Figure 12-4 shows the root directory entry format

The flag word is set to 3 for used entries. The directory field has a block address in bits 0 through 23, with remaining bits zero.

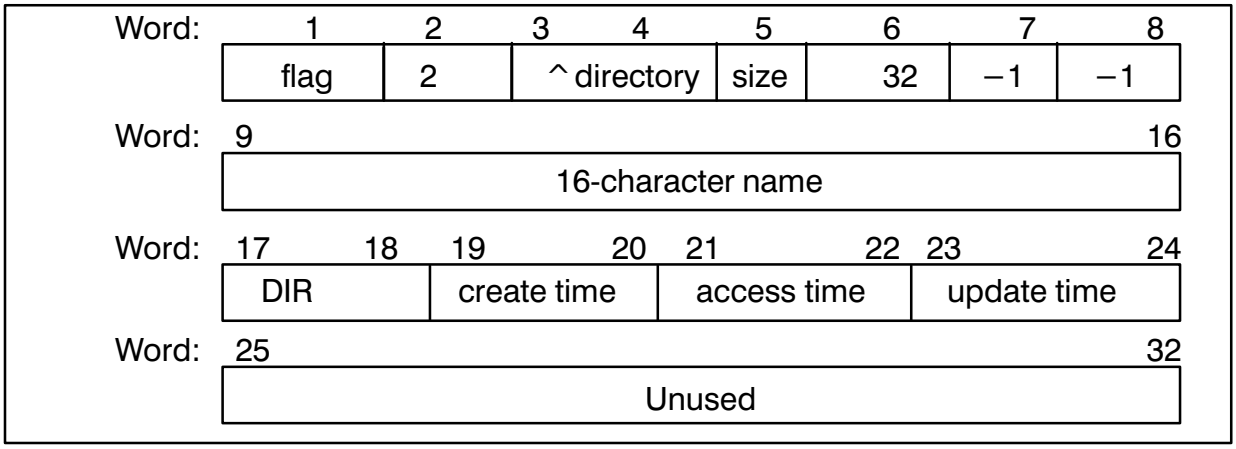


Figure 12-4. Root Directory Entry Format

Directory Header/Trailer

The format of the directory header/trailer is shown in Figure 12-5. The ^parent and owner fields are used only in the first header. Other fields are contained in all headers and trailers.

The flag field (word 1) contains the in-use code previously defined (in bits 0 through 3, 1 for header or 2 for trailer). In addition, this word has protection bits in the first header: owner read, bit 7; owner write, bit 6; group read, bit 11; group write, bit 10; others read, bit 5; and others write, bit 4. Bit 8 is set if the directory needs to be backed up.

Word 2 is the size word that shows the number of blocks in this directory.

The tag field contains a double integer with a distinctive bit pattern to allow some reconstruction/verification of trashed disks.

The ^other field contains the double integer block pointer to the previous or next part of this directory, or null (−1) if none. The block number is in bits 0 through 23 with remaining bits set to zero.

The ^parent field contains the double integer block pointer to the directory of this subdirectory; null (−1) if no this is not a subdirectory. The block number is in bits 0 through 23 with remaining bits set to zero.

The owner field contains a 16-bit identification number. This number identifies the owner of this directory. The group field contains a 16-bit identification number that identifies the owner's group.

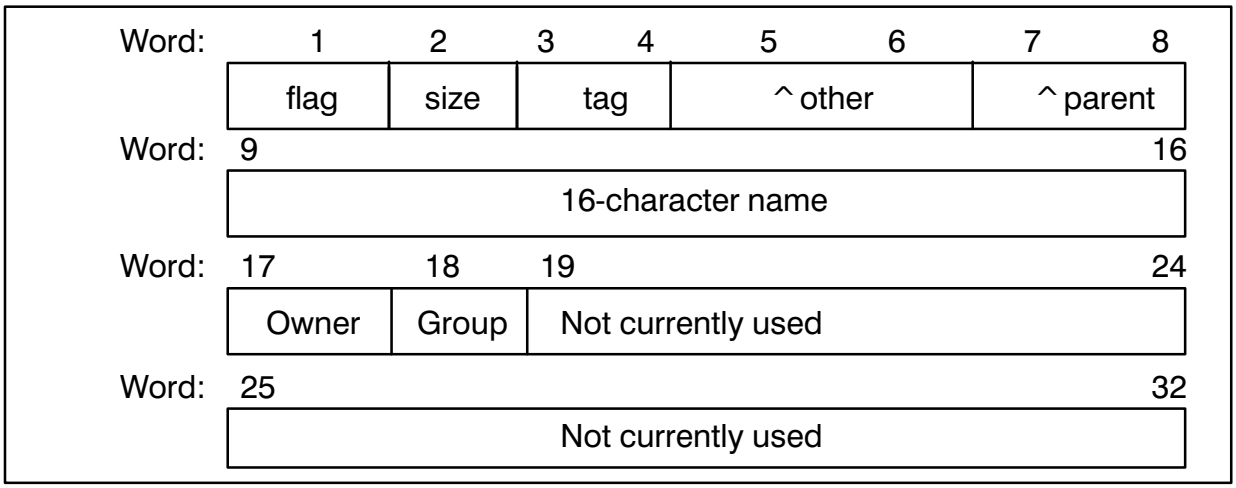


Figure 12-5. Directory Header/Trailer Format

File Entry

The file entry in the directory is shown in Figure 12-6. The various fields are explained below.

The flag field contains the following information:

Bits 0-3	=	3 if file exists 5 if it is purged
Bit 4	=	Set if non-owners have write access
Bit 5	=	Set if non-owners have read access
Bit 6	=	Set if owner has write access
Bit 7	=	Set if owner has read access
Bit 8	=	Set if the file needs to be backed up
Bit 9	=	Set if the file is temporary
Bit 10	=	Set if group members have write access
Bit 11	=	Set if group members have read access

The type field shows the FMP file type, value is a positive number.

The ^file field shows the block address of file data in bits 0 through 23, with the remaining bits set to zero.

The size field shows the size of the data pointed to by ^file. If positive, it is the number of blocks; if negative, it is the number of 128-block chunks. Note that this is not exactly the same as in the FMGR directory.

The reclen field shows the record length in words. For type 2 files this is the fixed record length; for type 3 and above files (except for type 12), this is the longest record length. The reclen field is undefined for type 12 files.

The ^extent field contains the double integer directory entry pointer to the first extent block for this file. This field contains the block address in bits 2 through 25; the entry in the block is in bits 0 and 1; the remaining bits are set to zero.

The type ext field contains the 4-character file type extension.

The file time stamp fields, create time, access time, and update time, show time in seconds since January 1, 1970. Time in each field is indicated by a double integer.

The nblocks field contains a double integer number that shows the blocks of disk space used by the file, including all extents.

The ^eof field contains a double integer word pointer to the eof mark in the file. For types 1, 2, and 6 files, the eof mark is the word after the highest numbered extent. For type 12 files, the ^eof field contains a double integer byte pointer to the eof that is rotated right one place.

The nrecords field shows the number of records in the file. The nrecords field is undefined for type 12 files.

The openflag field is the multi-computer open flag. This word is currently not used

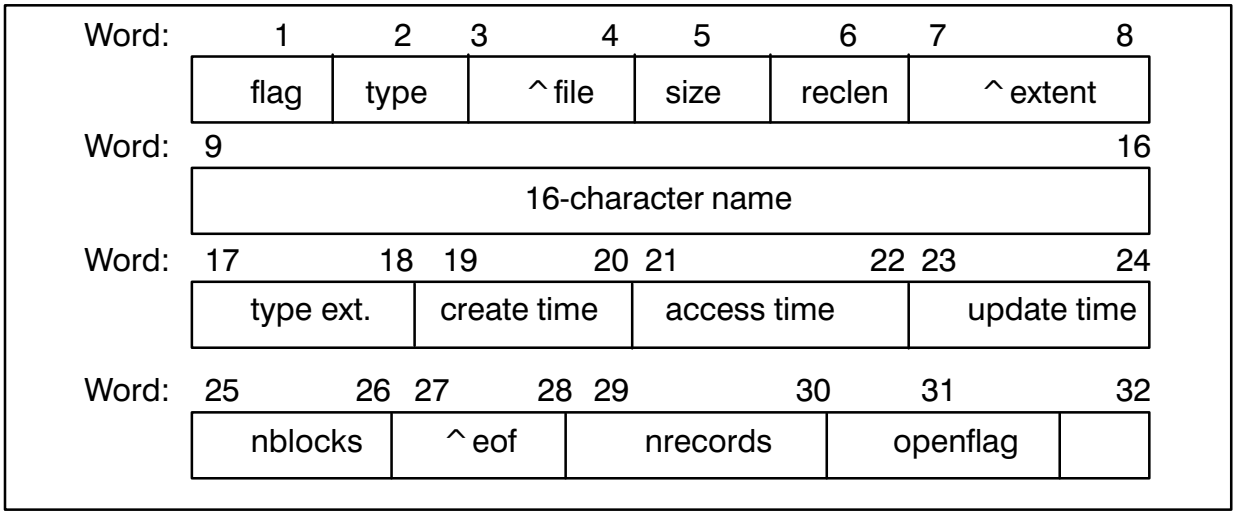


Figure 12-6. File Entry

Subdirectory Entry

The subdirectory entry is shown in Figure 12-7. The flag field contains a use code in bits 0 through 3, set to 3 if used. The protection bits are not used and therefore are set to zero.

The ^ directory field contains the block address of the subdirectory. Bits 0 through 23 are used for the address, other bits are set to zero.

The size field indicates the size of the subdirectory in blocks.

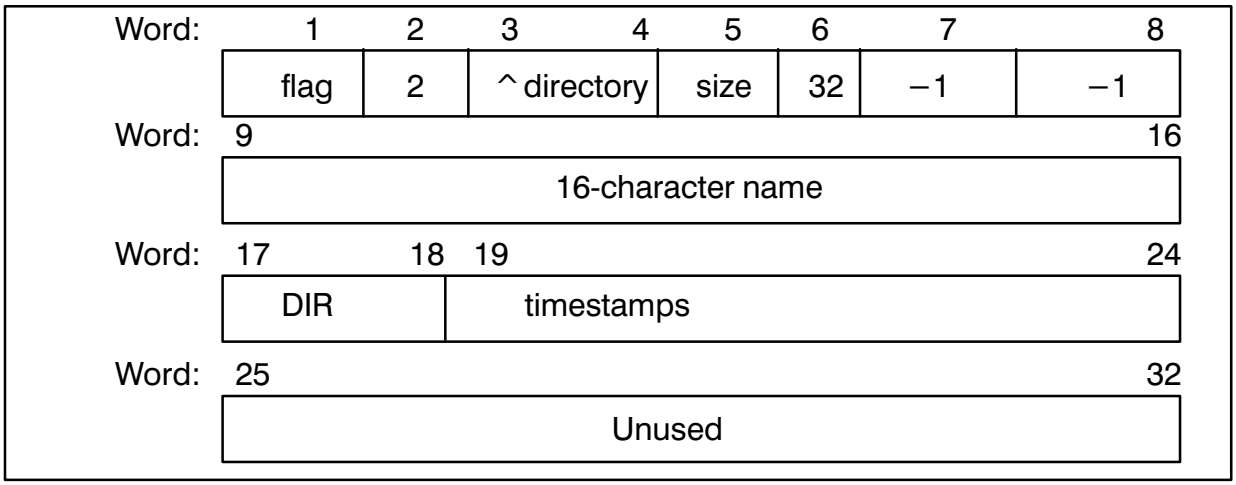


Figure 12-7. Subdirectory Entry

Extent Entry

The extent entry format is shown in Figure 12-8. The flag field contains a use code in bits 0 through 3. A value of 4 indicates an extent block, value of 6 for purged extent block. There are no protection bits.

The ^ext1 through ^ext9 fields contain double integer block pointers to nine extents for this file; extents 1-9 in first extent entry, 10-18 in second, etc. The block address is in bits 0 through 23 with remaining bits set to zero.

The size1-size9 fields show the size of each extent. If the file is a type 3, then the format is the same as the main file entry. If this is a type 1 or 2 file, then this is the extent number, which can be up to 32767.

The ^previous field contains a double integer directory entry pointer to the previous extent entry, or to the main file directory entry if this is the first extent entry. The block address is in bits 2 through 25; the block offset is in bits 0 and 1.

The ^next field contains a double integer directory entry pointer to the next extent entry. If this is the last extent entry, it is a null (−1). The block address is in bits 2 through 25, block offset in bits 0 and 1.

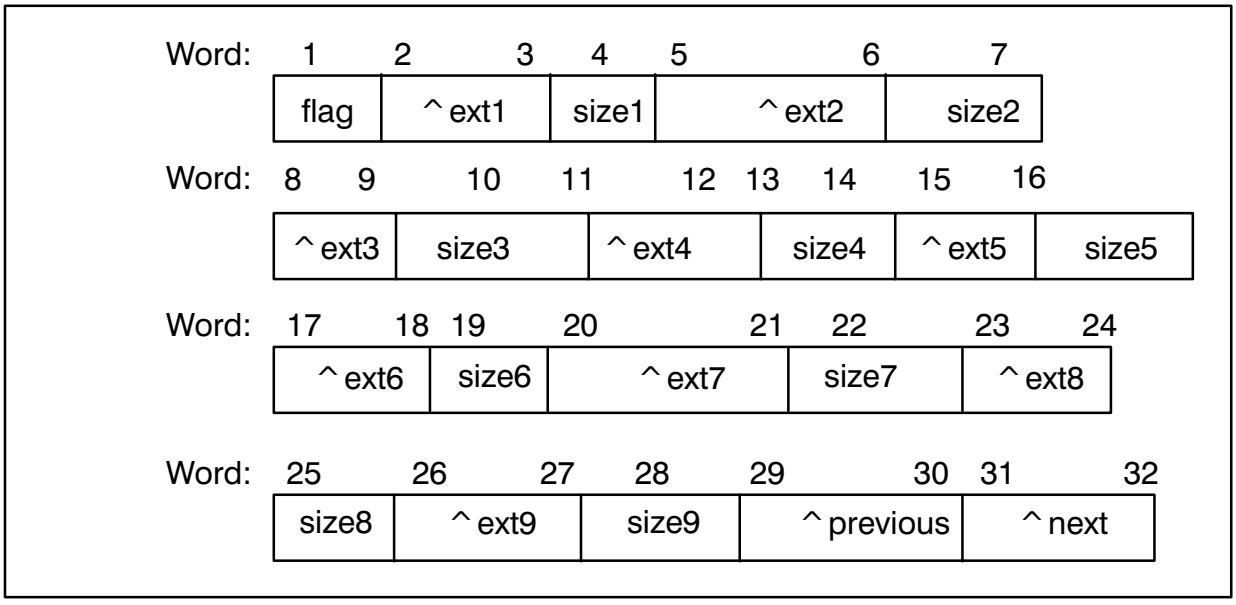


Figure 12-8. Extent Entry

Disk File DCB

The disk file Data Control Block (DCB) format is shown in Figure 12-9. Explanation of the DCB words is given below with the bit numbers shown in angle brackets. (The disk file Data Control Block format for a type 12 file is shown in Figure 12-10.)

Word 0: <13-15>: 16-word entry offset
<7-12> : Directory sector
<6> : File opened by DS transparency software
<0-5> : LU

Word 5: Size in sectors (+) or −(128-block) chunks

Word 7: <15> : Write allowed <14>: Buffer partially full
<7-13> : DCB buffer size in blocks
<6> : File modified <5>: Extents not allowed
<4> : Read allowed <3>: Update mode
<2> : Data in buffer <1>: EOF read
<0> : Data to be written

Word 12: Index of current word in zero-based DCB array.

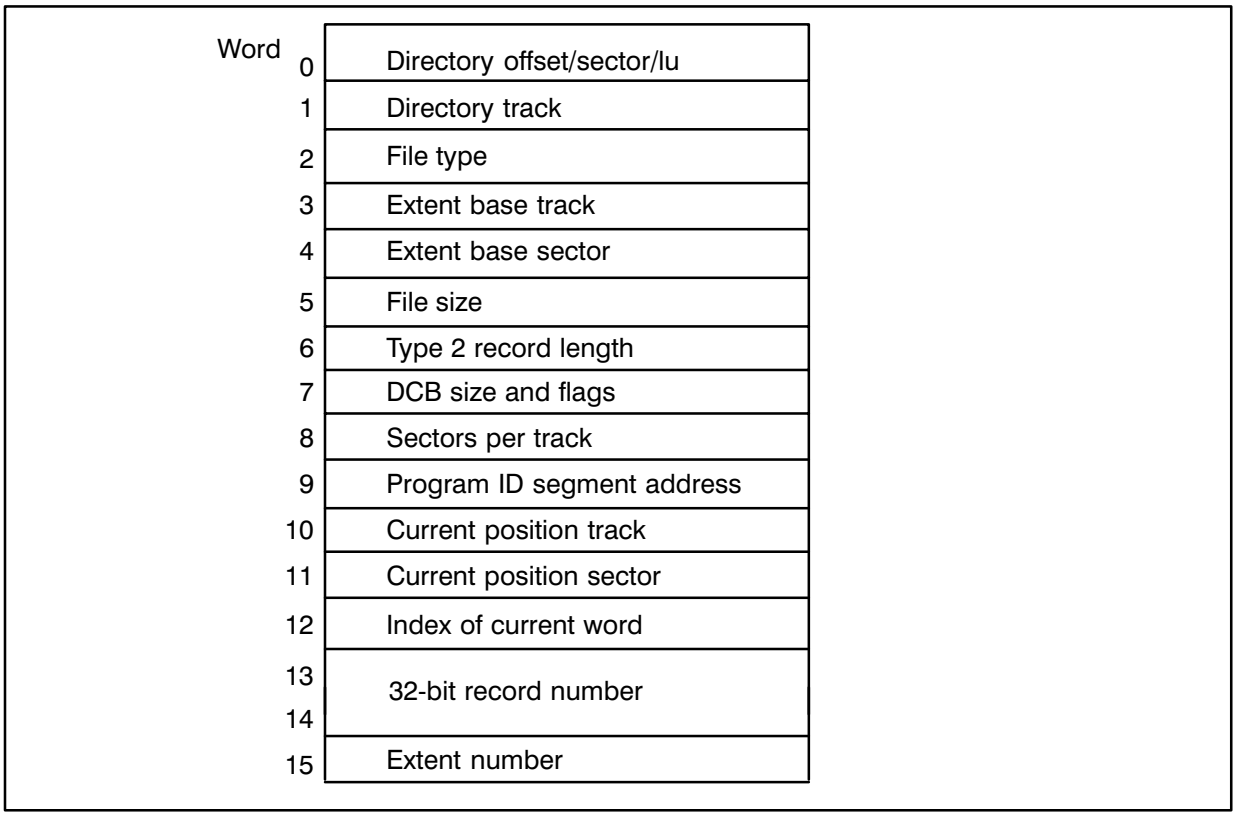


Figure 12-9. Disk File DCB

DCB Definitions for Type 12 Disk Files

The Data Control Block (DCB) definitions for a type 12 disk files are given below with the bit numbers shown in angle brackets. The disk file Data Control Block format for a type 12 file is shown in Figure 12-10.

Word 4: <8-15>: Extension of word 6; that is, the block number in the file of where the EOF is located
<0-7> : Extent base sector

Word 6: When the EOF flag bit is clear in word 15, word 6 is a pointer to the block containing the EOF. This is a 24-bit quantity; the lower 16 bits are located here in word 6, the remaining 8 bits are located in the high byte of word 4.

When the EOF flag bit is set in word 15, word 6 is a pointer to the location of the EOF within the DCB buffer. This is a byte offset (rotated right one bit).

Word 12: Index of the current byte rotated right one bit

Word 15: <15> : EOF flag bit
<0-14>: Extent number

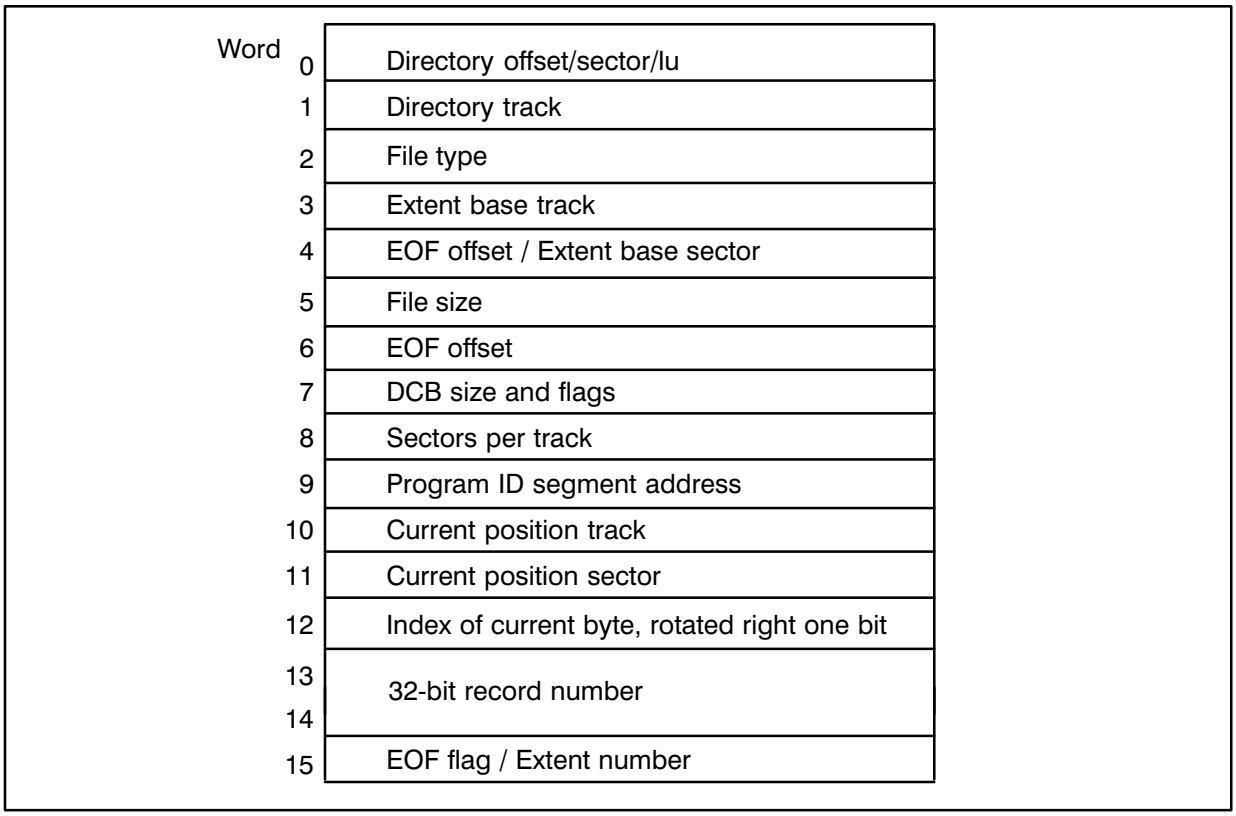


Figure 12-10. Disk File DCB for Type 12 Files

Device File DCB

The device file DCB is shown in Figure 12-11. The word format is:

Word 5: <15>: Backspace legal
: <0>: Forward space legal

Word 7: <15>: Write allowed
: <4>: Read allowed
: <1>: EOF read

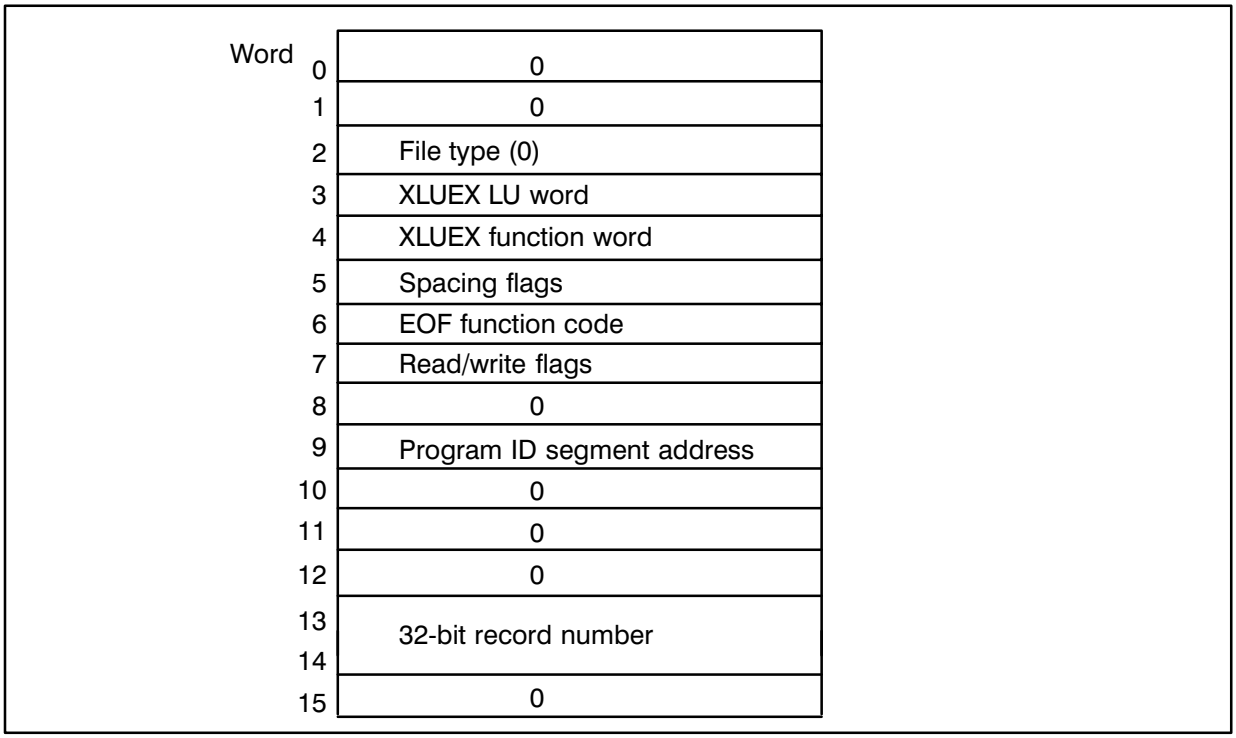


Figure 12-11. Device File DCB

FMGR Directories

This section describes the format of directories on FMGR disk cartridges.

FMGR Cartridge File Directory

The file directory for a disk cartridge starts on the last track of the cartridge (at sector 0) and extends towards lower numbered tracks. Each block of the directory (128 words) contains eight entries, and each entry contains 16 words of information. The directory blocks are staggered to accommodate FMGR directories. The sector address for a given directory block is given by the formula:

sector = (block * 14) MOD (sectors/track)

The first entry of the file directory is the cartridge header. It contains information about the structure of the cartridge, such as size label and next available space. Successive entries contain information about contiguous segments of data tracks on the cartridge. These entries can describe disk files, file extents, purged files, non-disk files and the end-of-directory.

FMGR Cartridge Header

This entry contains information concerning the cartridge structure. It is characterized by the sign bit of word 0 set to 1. The format of this entry is shown in Figure 12-12.

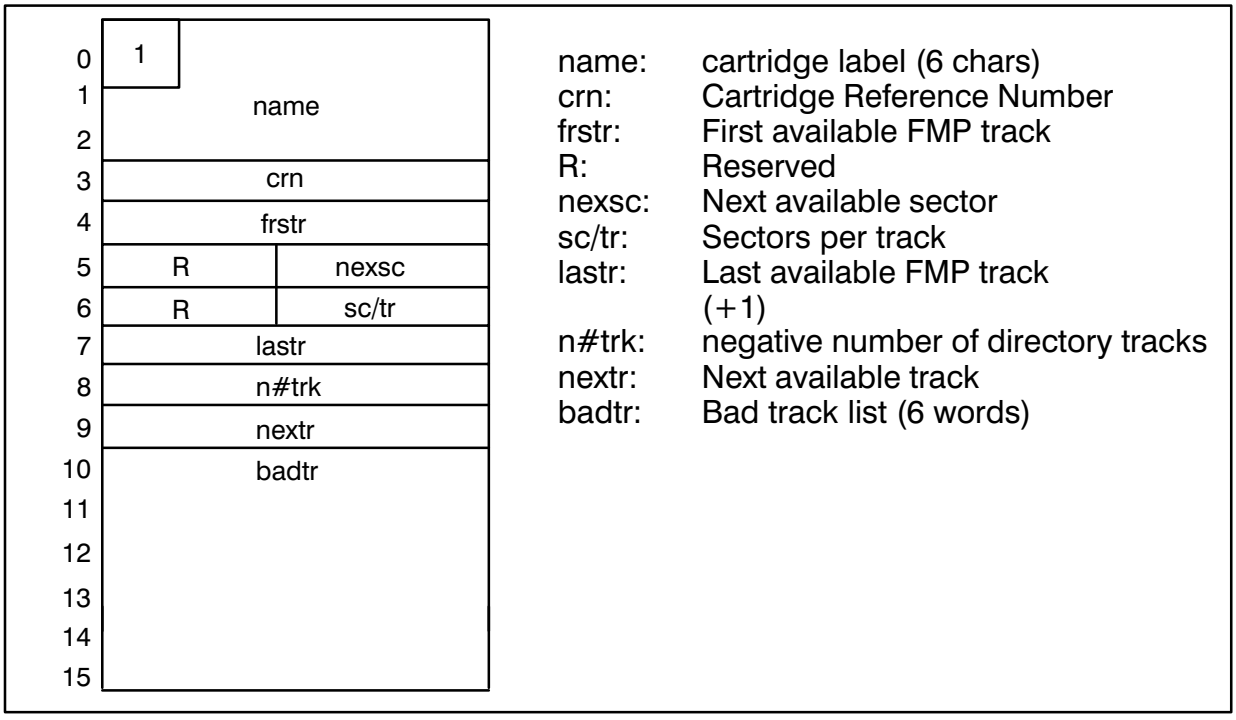


Figure 12-12. FMGR Cartridge Header

FMGR Disk File Entry

This entry type contains information concerning a disk file main extent. It is characterized by the sign bit of word 0 set to 0, word 3 set to non-zero, and the high byte of word 5 set to 0. The entry format is shown in Figure 12-13.

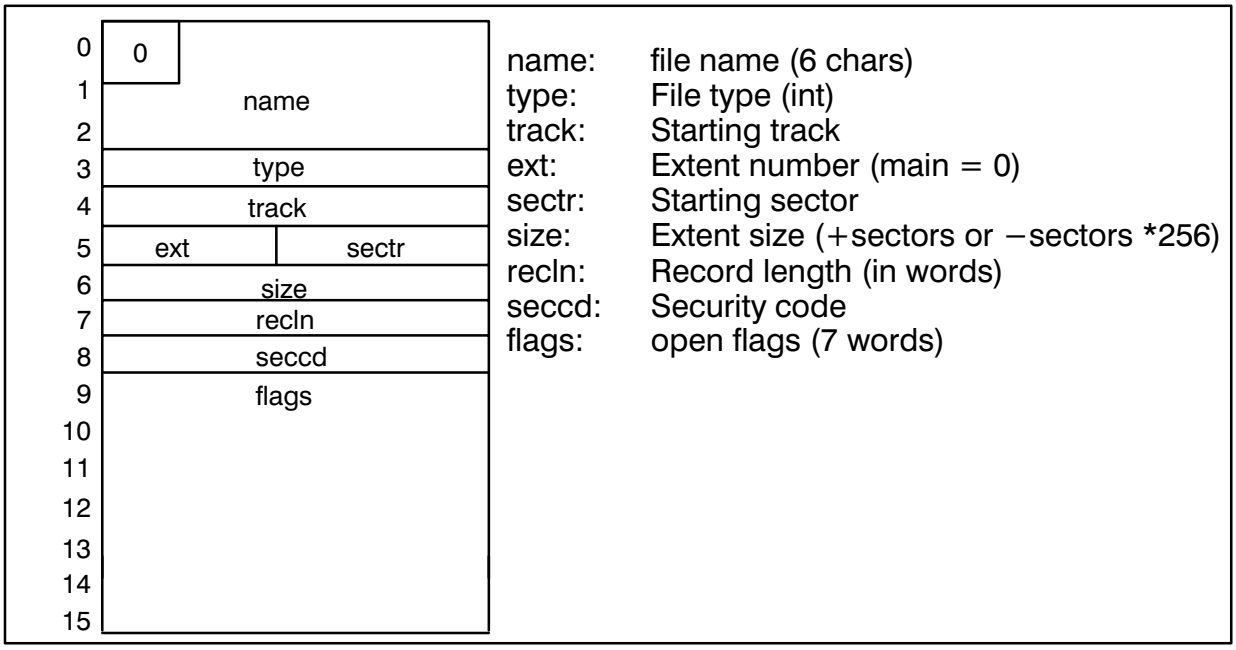


Figure 12-13. Disk File Entry

FMGR File Extent Entry

This entry type contains information concerning an extent of a disk file. It is characterized by the sign bit of word 0 set to 0, word 3 set to non-zero, and the high byte of word 5 set to non-zero. The entry format is shown in Figure 12-14.

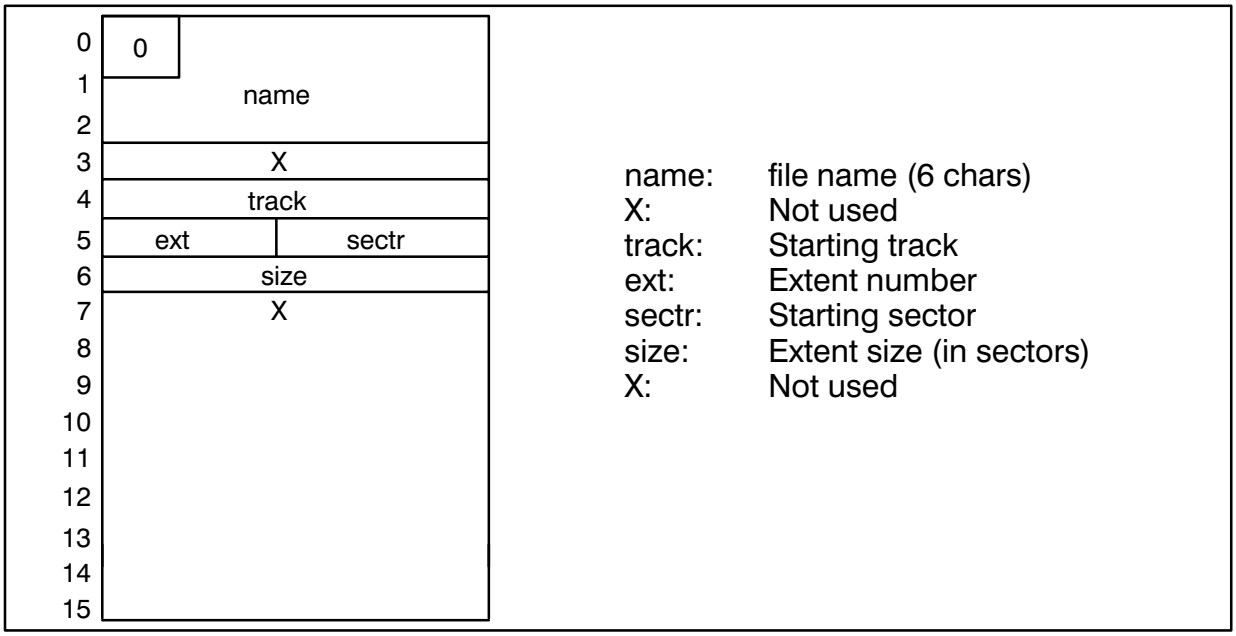


Figure 12-14. FMGR File Extent Entry

Non-Disk File Entry

This entry type contains information concerning a non-disk file (LU). It is characterized by the sign bit of word 0 set to 0, and word 3 set to 0. The format is shown in Figure 12-15.

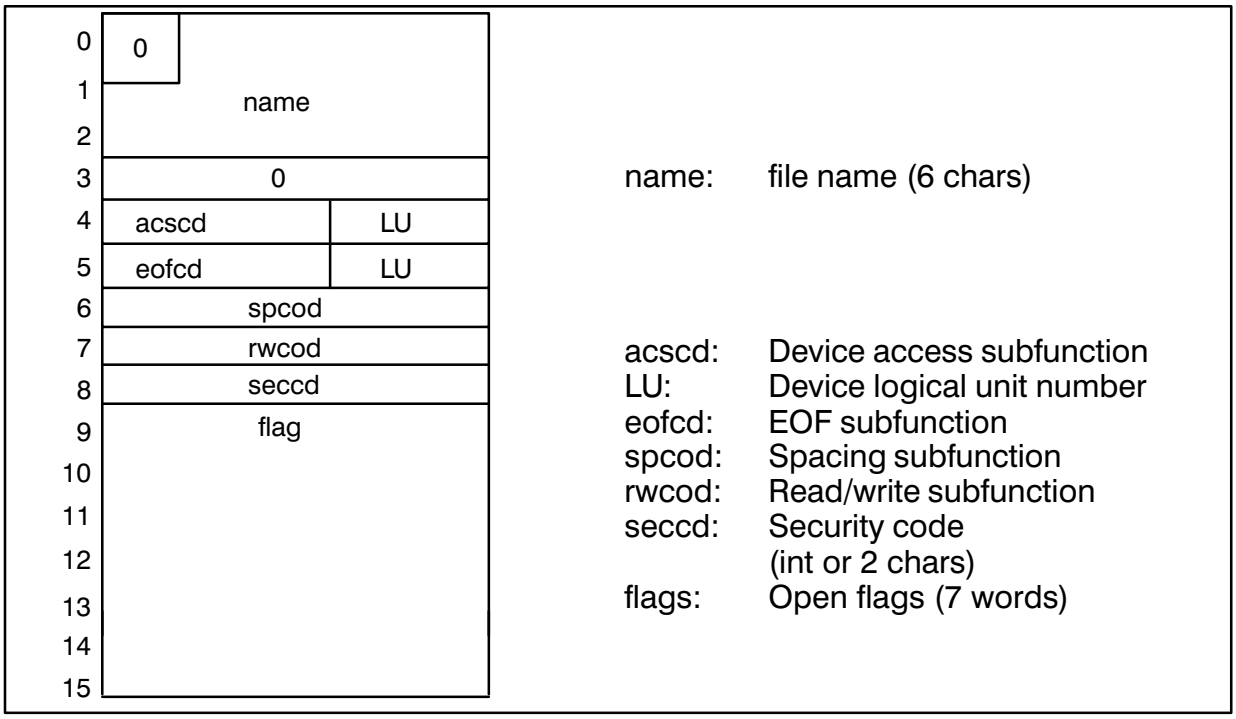


Figure 12-15. Non-Disk File Entry

FMGR Purged File Entry

This entry type indicates that a file has been purged, but its space has not been reclaimed yet. It is characterized by word 0 set to −1.

FMGR End-of-Directory Entry

This entry indicates the logical end of the file directory. It is characterized by word 0 set to 0.

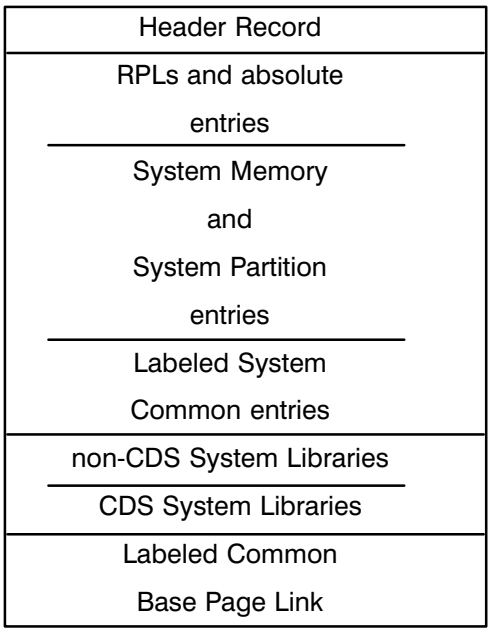
A

Snapshot File Format

The snapshot file is a type 3 variable length file consisting of four major sections:

1. A header record with information about system configuration.
2. A list of the names, values, and types of the system entries.
3. A list of system library files named at generation.
4. A memory image of the labeled common base page link area.

The format of the snapshot file is as follows:



Header Record

The Header Record section of the snapshot file contains information about the configuration of the system. The format of the Header Record is as follows:

Word	Contents
1	# Total Entries
2	# System Entries
3	# non-CDS System Libraries
4	$LCOM
5	$BCOM
6	First Word Available after Common
7	System RPL Checksum
8	# Labeled Common links on base page
9	System ID checksum
10	Labeled System Common Checksum
11	# CDS System Libraries
12	Spare
13	Spare
14	Record # of first non-CDS Library Entry
15	Record # of first CDS Library Entry
16	Spare
17	Record # of first Base Page Entry
18	Spare
19	Spare
20	Checksum of Words 1 through 19

Total Entries (Word 1)

The total number of symbol entries in the snapshot file. This number includes labeled common entries and system entries.

System Entries (Word 2)

The number of symbols in the snapshot file that are defined by the generator while relocating the system. The number may include system RPLs, ABS, and system memory symbols. These symbols are the first listed in the system entries section of the snapshot file. This number includes all entry points not defined in the system common relocation phase.

non-CDS System Libraries (Word 3)

The number of system libraries that LINK searches when relocating non-CDS programs.

$LCOM (Word 4)

The logical address of the start of labeled common. The address is always 2000B.

$BCOM (Word 5)

The logical address of the start of blank system common.

First Word Available After Common (Word 6)

The logical address of the first word available for relocation when a user program accesses system common.

System RPL Checksum (Word 7)

A rotating checksum of the RPL and ABS entries defined when relocating the operating system. The checksum does not include RPLs or absolutes defined when relocating the labeled common area.

Labeled Common Links On Base Page (Word 8)

The number of words of labeled common links on the base page. These links are provided to the linkage editor (LINK) for inclusion on the base page of any program referencing labeled system common.

System ID Checksum (Word 9)

A checksum of system memory and system partition symbols.

Labeled System Common Checksum (Word 10)

A rotating checksum of the symbols in the snapshot file defined when relocating labeled system common. The checksum includes the logical starting address of blank, labeled, and program spaces when system common is accessed. The checksum is used to detect any change to the system common area of the operating system.

of CDS System Libraries (Word 11)

The number of system libraries that LINK searches when relocating CDS programs.

Record # of first non-CDS Library Entry (Word 14)

The record number of the first non-CDS library entry record in the snapshot file.

Record # of first CDS Library Entry (Word 15)

The record number of the first CDS library entry record in the snapshot file.

Record # of first Base Page Entry (Word 17)

The record number of the first record in the snapshot file containing labeled common base page links.

System Entries

The System Entries section of the snapshot file contains a list of names, values, and types of the system entries. The format of the System Entries area is as follows:

Word Contents		
1	Length of symbol name (in words)	
2	1st character	2nd character
3	3rd character	4th character
4	5th character	6th character
. . .	. . .	
N	Entry point type	
N+1	Entry value	

Entry point type can have the following values:

Type	Location of Symbol
0	System memory and system partition
2	System common
3	Absolute
4	Replacement (RPL)

System Libraries

The System Libraries section of the snapshot file contains the names of the non-CDS and CDS system libraries defined when the system was generated. The format of the System Libraries section is as follows:

Word Contents		
1	1st character	2nd character
2	3rd character	4th character
3	5th character	6th character
.	.	.
.	.	.
.	.	.
32	63rd character	64th character

Labeled Common Base Page Links

The Labeled Common Base Page Links section of the snapshot file contains a memory image of the labeled common base page area. The format of the Labeled Common Base Links section is as follows:

Word Contents		Link Address
1	value	1777B
2	value	1776B
3	value	1775B
.	.	.
.	.	.
.	.	.
N	value	140B

Index

M

N

O

P

R

S