UP YOUR ACP

Version 02.00
April 17, 1980

Ralph W. Stamerjohn
Monsanto

FOREWORD

Dedicated to Esther, who knows more about ACP's than any other 370 programmer in the world.

This document is written by an RSX-11M user for RSX-11M users. The initial seed for this document was planted when I researched a project to implement DECNET DAP protocol at the FCS level and discovered, contrary to popular belief, there is nothing to prevent an user from writing an ACP. Quite the contrary, I can now implement ACP's as easily as device drivers.

The one thing lacking is documentation. This is my attempt to fill the gap and save others the long nights spent poring over listings and manuals. This is not to say ACP's are trivial. It is important to remember that while the concept of an ACP is simple, the implementation will be as difficult as the problem you are attempting to solve.

The manual is directed towards three types of readers. First is the developer faced with the task of writing an ACP. The entire manual applies to such a reader. Before refering to this material, the RSX-11M Guide to Writing an I/O Driver and the chapters in the RSX-11M System Logic Manual on the I/O mechanism and data structures should be covered.

Next, the manual can be used by someone who merely wants a better understanding of the ACP mechanism. The first three chapters should be sufficient for an introduction. Finally, the manual can be used for reference. In particular, the appendices contains useful reference material.

A sample ACP and supporting enabling and disabling tasks have been developed to be used with the manual. The source code should also be studied, especially when reading chapter 3. The sample code can also be used as a basis for a user-written ACP.

This document reflects ACP's as of version RSX-11M V3.2. It does not apply to IAS or VAX/VMS. The thrust of the document is how to fit an ACP into RSX-11M, not how to write any specific ACP.

Naturally, neither the author, Monsanto, nor DECUS claim any responsibility of the accuracy for the material in this document. If mistakes are found or misinterpretations occur, please notify the author so that the material can be corrected. ACP's are moving targets and, at best, this document can only move slightly behind them.

The reader should be aware that some material has only recently been added and has not been proofread by anyone but the author. This includes all of chapters 4, 5, and 6. The manual should be considered with a grain of salt, especially, the new chapters.

The manual would have been impossible without the aid of many people. The management of the Monsanto Agricultural Research department, especially John Schaefer and Larry Jasper, must be thanked for providing an environment that allows programmers to be imaginative. From the RSX SIG, Rick Aurbach, Fred Veck, John Wood, and Jim McGlinchey have all contributed ideas and time. Jim and Rick has been especially valuable in editting the material and providing insight into how the material should be presented. Finally, the manual would never have been completed without the constant aid of by wife, Esther.

Ralph Stamerjohn
Monsanto, Zone T1A
800 N. Lindbergh
St. Louis, MO, 63166

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

CHAPTER 1

ACP PHILOSOPHY

Ancillary Control Processors (ACP's) are an integral component of the RSX-11M operating system. Yet they remain one of the least understood portions of the system. This chapter will develop a definition for "ACP" and explore the functionality ACP's provide in an RSX-11M system.

1.1 DEFINITION OF "ACP"

Ancillary control processors are a part of the RSX-11M I/O mechanism. They implement device protocols for a variety of I/O devices. Because the operation of ACP's is hidden from the programmer, they have been considered untouchable in the past. Conceptually, however, an ACP is nothing more than a task which is tied to a device driver via the I/O mechanism. As this manual will show, the interface between an ACP and a driver is very simple and allows ACP's to be a powerful tool for system developers.

Every operating system's I/O mechanism can be broken into various functional components. At the lowest level are the device service routines. These routines perform the actual device I/O. The next level are the device protocol routines. The protocols allow various devices to be classed together as a single logical type and extend the device functionality beyond the actual device capabilities. File systems and communications protocols fall into this class. At the next level is the mechanism for program I/O requests. The same mechanism can be used for each device or different devices may use different methods. The final level of functionality are the logical I/O services. Record I/O falls into this category. Logical I/O services allow data to be represented in logical, rather than the actual machine format. In many systems, this level is also used to implement device independence.

While different operating systems use different implementations for their I/O mechanism, similarities do exist.

Device service routines are usually included in the monitor because of their requirements for responsiveness and close coupling with the hardware devices. On the other hand, while a property of the operating system, device protocol code is usually excluded from the kernel monitor because of its complexity and size. Some mechanism is used to allow the device protocol code to communicate with the device service routine.

RSX-11M uses ACP's to implement device protocols. The concept of an ACP was first used for RSX-11D. Each device was serviced by a task which also contained the interrupt service code. These tasks were called device handlers and allowed the programmer to use the power of both task and executive state. For very complicated protocols such as the file system, other tasks were used to perform the various file I/O functions.

This approach suffered from several flaws. For simple devices such as line printers, using a task for device service was overkill. For complex devices, the structural limitations complicated the implementation. When RSX-11M was introduced, device service and device protocol were separated into distinct components. Simple device drivers were included in the kernel executive. Special tasks (ACP's) were coupled to the executive to perform the device protocols. The coupling mechanism has changed with various releases, but the separation of the two components has remained.

In summary, an ACP is a task which is tied to the I/O mechanism and is used to provide a protocol for a class of devices. ACP's enjoy both the features of a task and the power of the executive. As will be shown later, the coupling of ACP's to the executive and device drivers is very simple, making user-written ACP's a workable solution to a wide variety of application problems.

1.2 ACP CHARACTERISTICS

Several ACP's have been written by Digital. The common Digital ACP's are the the disk file system (F11ACP), ANSI magtapes (MTAACP), and the NSP portion of DECNET (NETACP). These ACP's have the following attributes and characteristics.

1. Each ACP is a privileged task and has all the attributes of a task (stack, priority, task name, checkpointing, etc.).

2. Each ACP can be considered a portion the executive. ACP's are mapped into the executive and run in kernel state whenever necessary.

3. Each ACP implements a protocol for a class of devices:

   1. F11ACP implements the Files-11 protocol for all disks (RK05, RP06, etc.) and DECtapes.

   2. MTAACP implements the ANSI magtape protocol for all supported tape drives (TU10, TE16, etc.).

   3. NETACP implements the NSP portion of the DECNET protocol for a wide variety of communication interfaces (DL11, DMC11, etc.).

4. The protocols extend the functionality of the devices in several ways:

   1. The burden of direct device manipulation is removed from the programmer and is handled by the ACP in conjunction with the device drivers.

   2. The burden of protocol level data manipulation is handled by the ACP's. The ACP's strip any data related to protocol and return only the information related to the application.

   3. The ACP's allow the devices to function as logical, rather than physical, entities. In the case of Files-11, a program deals with files instead of physical disk blocks. F11ACP handles all block mapping and allocation. Programs view files as a set of contiguous blocks, regardless of the actual allocation.

   4. The ACP's allow different devices to be treated the same from the program's viewpoint. DECNET's support of logical links over asynchronous, synchronous, and parallel interfaces is the most vivid example of this.

   5. The devices can be shared for simultaneous access. Each accessing process is protected from others by the ACP's. The ACP's also synchronize access to the physical devices when necessary.

   6. The devices are protected against unauthorized and destructive access. When an ACP is used with a device, the ACP "owns" the device.

5. Each ACP can be mounted/dismounted for a given device. When mounted, the device is available only for use in the context of the device protocol provided by the ACP.

One lack of similarity must also be mentioned. The internal implementation of the Digital ACP's have nothing in common. The only similarities are in the interface to the operating system. How an ACP accomplishes its purpose is left to the individual ACP.

1.3 I/O HIERARCHY

As was stated before, all I/O mechanisms can be broken into four functional components: logical I/O services, program I/O requests, device handling, and device protocol services. This section will discuss how each of these functional components are implemented by RSX-11M. In particular, the relationship of each component to an ACP will be discussed.

The I/O mechanism for RSX-11M can be mapped directly into the four areas. The FCS/RMS libraries provide the logical I/O services. The QIO directive and the associated executive processing form the I/O request mechanism. Device drivers are used for device service. And finally, ACP's are used to implement device protocols.

1.3.1 FCS/RMS

Logical I/O services for RSX-11M are provided by the file/record service libraries: Record Management Services (RMS) and File Control Services (FCS). The two most important services provided by these packages are device independence and logical records. The packages also provide a convenient programming interface for issuing I/O requests.

Both FCS and RMS are implemented as macro and subroutine libraries. RMS is the newer package. It extends the concept of logical records, particularly in the area of record organization (relative and keyed). FCS is the traditional package and is more widely used in RSX-11M systems. However, the libraries are equivalent in their relationship to the other components of the I/O mechanism.

When called, FCS/RMS translate the logical I/O requests into device specific I/O requests. This is done by examining the characteristics of the assigned device and issuing the appropriate QIO directives. The packages are also aware of the existence of the Files-11 and magtape ACP's and issue I/O requests specific to these ACP's. It is important to remember that FCS/RMS do not implement the file system, they are only aware of how to interface to it to accomplish the services

requested by the programmer. A new ACP can be interfaced to FCS/RMS by teaching the packages the proper usage of the ACP.

1.3.2 QIO Directive

All user I/O is requested by using the QIO directive. The directive can be issued directly by a program or indirectly from the various run-time systems.

The QIO directive together with the executive processing form the program I/O request component of RSX-11M. The RSX-11M executive is responsible for translating the QIO requests into their internal representation (I/O packets), checking for parameter validity, and dispatching the I/O packet to the correct processing routine.

The device data bases are used by the executive to determine how a packet is to be processed. The executive may pass the function directly to a device driver or queue the packet to the driver I/O queue. If an ACP is mounted for the device, the packet is specially processed and sent to the ACP instead of the device driver. It is extremely important to understand this mechanism when writing an ACP.

1.3.3 Device Drivers

Device drivers are the lowest level in the RSX-11M I/O hierarchy. Drivers provide the device service functionality and are small, responsive modules. RSX-11M device drivers do little more than physical data transfers and device manipulations. The driver I/O parameters are specified in a basic, often device dependent form.

An ACP can interface to a device driver in a variety of ways. The most common approach is to use the normal QIO directive. An ACP will typically take the I/O request queued to it and issue device specific I/O requests to satisfy the request. The driver code itself is not aware that an ACP is calling it.

Device drivers serve another purpose in implementing ACP's. The driver mechanism is convenient for extending executive processing. As will be seen later, pseudo drivers can be implemented to provide the processing necessary for ACP I/O requests that is normally done by the executive. The use of such drivers permits user-written ACP's to be implemented without modifying the kernel executive.

1.3.4 ACP's

Finally, ACP's are the component which implement device protocols. These tasks take I/O functions from the executive and perform the desired operation according to the rules of the protocol.

1.3.5 Summary

At this stage, a simplistic view of the RSX-11M I/O mechanism has been developed. The simplest case is a QIO request from a task to a non-ACP device. The flow is through the executive's QIO processor to the device driver to the device. By adding FCS/RMS, the task makes a subroutine call to the run-time library which then issues the actual QIO. The final case (Figure 1-1) adds an ACP to the device. Here, the I/O request is routed to the ACP by the executive. The ACP may issue its own I/O requests to the device driver or satisfy the request using its internal data bases. In some special cases, the executive may route the packet directly to the driver.

1.4 ACP ABILITIES

It is not often clear why ACP's are such a necessary part of the RSX-11M I/O mechanism. This section will discuss the abilities which are unique to ACP's and make them such a powerful application tool.

The key abilities come from the fact an ACP is a task. ACP's are allowed to do things which are impossible from device drivers. They can issue all the RSX-11M directives. For example, the ability of an ACP to issue QIO's allows it to service a complicated I/O request by using the simpler services provided by a device or another ACP.

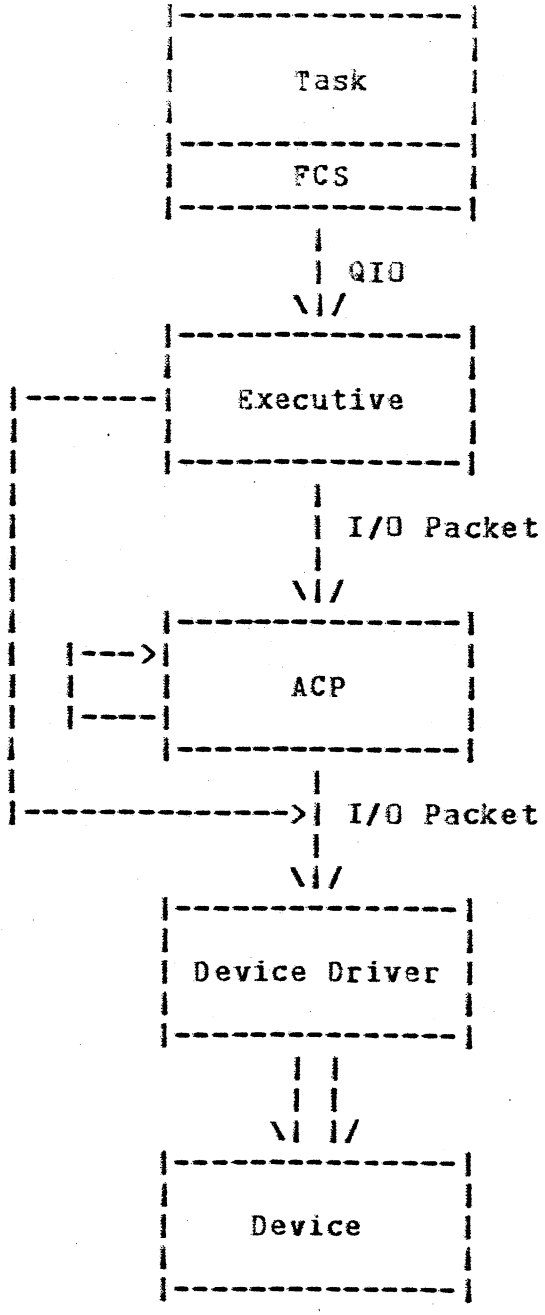


Figure 1-1
RSX-11M I/O MECHANISM

At the same time, ACP's are privileged tasks. The full abilities of the executive are available. For example, ACP's can allocate and return system pool merely by switching to system state and calling the executive routines. In addition, as privileged tasks, ACP's are mapped into the system pool and can examine and manipulate all RSX-11M executive data structures.

Also, because ACP's are tasks, they can be used with all the programming tools provided by RSX-11M. For example, they may use disk and memory-resident overlays, be checkpointed, and link to ODT. The last ability is a very key feature as it allows ACP's to be debugged much less destructively than device driver code.

The executive interface to ACP's also has abilities not normally available to device drivers. A feature of the interface allows ACP's to establish a mapping between several separate I/O requests. The process of opening a file, reading and writing disk blocks, and closing the file is an example of this mechanism. Similarly, DECNET supports establishing a logical link, transmitting and receiving data, and terminating the link.

This feature allows an "I/O process" to be established for a logical unit in a task. Each process uses an ACP unique data structure called a window to establish the linkage between the lun and the operations being performed by the ACP. The window address is stored in the second word of the logical unit table. This address is passed to the ACP in the I/O packet and allows the ACP to reference the window when serving a request. The mechanism also includes support for notifying the ACP when the task terminates and I/O processes are still outstanding, even if the task has no outstanding I/O.

1.5 ACP IMPLEMENTATION

This manual will cover three types of ACP implementations: standard, user, and foreign. While each form fulfills the definition of "ACP", standard ACP's follow the rules used by Digital in writing F11ACP and MTAACP. Such ACP's are "known" to the executive and special code is included for their support.

On the other hand, user ACP's are not "known" to the system, only to the device driver which uses their services. The major advantage of the approach is that no changes to existing executive code are necessary for the user ACP, yet, all the advantages of having an ACP remain.

Foreign ACP's are between standard and user ACP's. The ACP

is known to the executive, yet no special services are provided for support of the ACP.

Throughout this paper, references will be made to the requirements for writing standard, user, and foreign ACP's. In order to simplify the terminology, "ACP" will be used when discussing material relevant to all styles. "DCP" (Digital Control Processor) will be used when refering strictly to standard ACP's. "UCP" (User Control Processor) will be used for references concerned with the user form and "FCP" (Foreign Control Processor) will refer to ACP's which use the foreign feature of RSX-11M.

The choice of writing a DCP, UCP, or FCP depends on the purpose of the ACP. The major difference between the various forms of ACP's is how and where the I/O requests designated for the ACP are processed. No matter which form is implemented, a certain amount of pre-processing is required before an I/O request is received by an ACP. The next three sections examine how each form of ACP interfaces to the executive.

1.5.1 DCP Approach

Figure 1-2 diagrams the flow of an I/O request from an user task to a DCP. When a QIO is issued from a task, the directive dispatcher calls the entry $DRQIO. The common QIO processing is performed first. The current task state is checked to see if the QIO can be issued at this time. If it can, the common QIO fields are checked for legality. Finally, an I/O packet is allocated from the system pool and initialized with information taken from the QIO directive parameter block.

Once the common processing is completed, the QIO processor dispatches to the function unique code. The current device status stored in the Unit Control Block (UCB) and the function bit masks in the Device Control Block (DCB) determine the type of function processing selected. In the case of an I/O packet designated for a DCP, the device driver must be loaded, the device marked as mountable, mounted, and not foreign, and the function marked as legal and ACP in the DCB mask fields. When these conditions are met, the ACP special packet processing is invoked. Each I/O function supported by Digital ACP's is processed using special polish-driven routines.

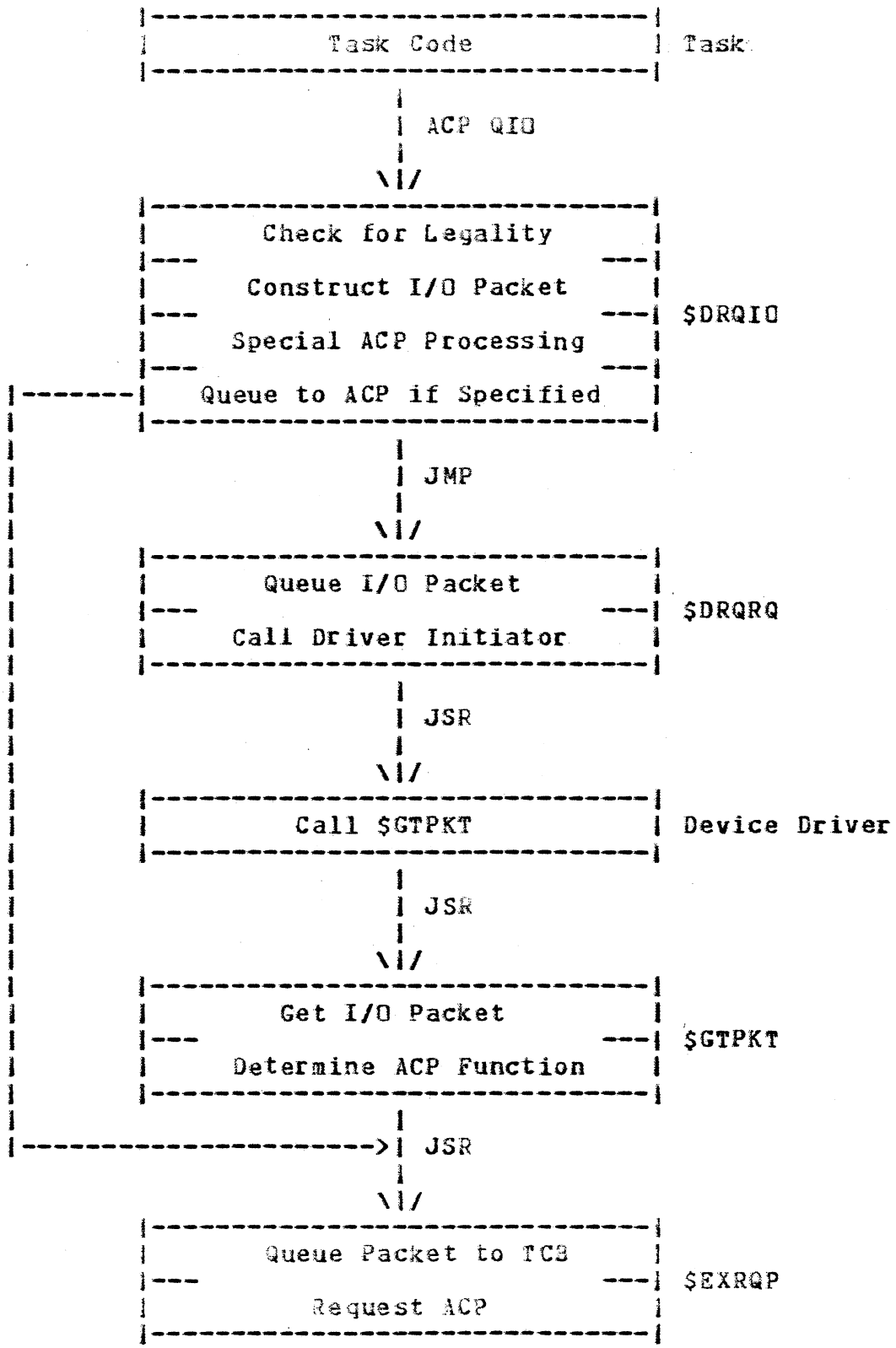


Figure 1-2
DCP FLOW

One function of the special processing is to decide whether the I/O packet needs sequencing with the device driver state or whether it can be queued directly to the ACP. If the packet does not need sequencing, DRQIO queues the I/O packet to the receive queue of the ACP and schedules the ACP for execution. Otherwise, the packet is placed in the I/O queue of the driver and the driver initiator is called. The packet will be queued to the ACP when it is dequeued by $GTPKT. This occurs transparently to the driver.

I/O packets are queued to an ACP via the ACP task's receive queue. This is the same queue used by normal tasks to receive messages from other programs. The entries $EXRQP/$EXRQF in the executive queue the I/O packet and also cause the ACP to be scheduled for execution.

In short, a DCP is known to the executive. Special code in the executive's DRQIO module handles functions designated for a DCP. The features of this approach are as follows:

1. I/O functions processed by the DCP are marked as legal and ACP functions in the DCB function masks. DCP function codes are required to be from 7-31(10).

2. The device is marked as mountable, mounted, and not-foreign in the UCB.

3. The I/O packet is specially processed by DRQIO. The type of processing performed depends on the I/O function code and is done by the polish-driven routines in DRQIO.

4. The I/O packet is queued to the DCP by the executive, however, for some functions the packet is queued directly by DRQIO and for others it is handled by GTPKT.

The DCP approach is appropriate for a user-written ACP when the new ACP processes current Digital ACP functions. In this case, no modification of the code in the executive is necessary to support the user-written ACP. It is also important to understand the DCP approach when using the methods discussed in the next two sections. The FCP and UCP approaches are not short-cuts, they merely move the special processing performed in DRQIO to outside the executive.

1.5.2 UCP Approach

The UCP approach provides all the functionality found in Digital's implementation of ACP's. The approach uses features of the RSX-11M I/O mechanism to move the special code for ACP I/O processing to outside the executive. This permits a user to write a totally new ACP without modifying the executive.

The key to the UCP approach is the UC.QUE bit in the UCB. When this bit is set, I/O packets are not queued to either the device driver or any associated ACP. Instead, the driver initiator is called directly from the DRQIO module. The driver can then perform any necessary special packet processing in-line with the executive's I/O processing. When finished the driver is responsible for placing the I/O packet in the correct queue.

The fact that no context switch can occur between the executive and the call to the device driver is critical to the UCP implementation. Certain types of processing done on the I/O packet, such as address checking and relocation, must occur in the context of the task which issued the I/O request. Once a packet is placed in driver or ACP queue, a context switch may occur before it is retrieved. In the case of a packet queued to an ACP, a context switch is implied by the fact that the ACP must be scheduled.

Figure 1-3 illustrates the flow of an I/O request from an user task to a UCP. First, the common QIO processing occurs. The first difference between a DCP and UCP is evident when the dispatch is made to the function unique code. The UCP approach marks all functions serviced by the UCP as legal and control functions. By marking the function as a control function, the executive merely copies the six QIO parameters to the I/O packet. This avoids any special processing built into the executive for DCP-style ACP's.

Once the I/O packet is constructed, the driver initiator is called without any queueing, due to UC.QUE being set. The driver can then process the I/O parameters in the I/O packet as needed. The driver then queues the packet and requests the UCP by calling the normal ACP scheduling routine. If the packet should not be routed to the UCP, the driver queues the packet to itself and calls $GTPKT to correctly synchronize driver packet processing.

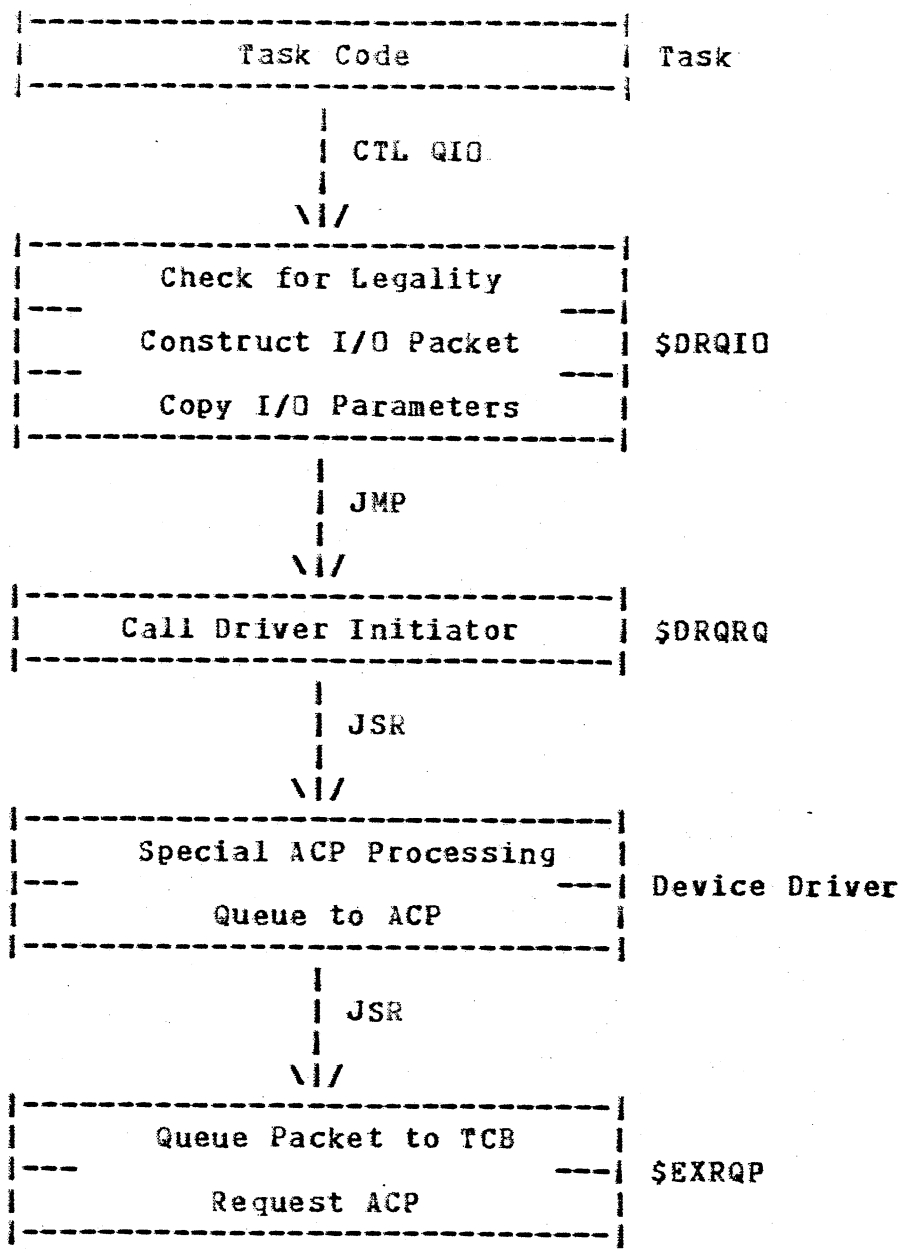


Figure 1-3
UCP FLOW

Therefore, an UCP is known to the device driver. The executive is unaware that a UCP is being interfaced to the driver. This has the disadvantage of generally limiting a UCP to supporting only one device driver. The features of the UCP approach are as follows:

1. The I/O functions processed by the UCP are marked as legal and control functions in the DCB function masks.

2. The device is typically marked as mountable, mounted, and not-foreign in the UCB. The driver is also set for no queueing of the I/O packet (UC.QUE=1).

3. The I/O packet is processed as a control function by DRQIO. Any special processing required by the UCP is performed in the device driver.

4. The I/O packet is queued to the UCP by the device driver when it finishes its special processing.

1.5.3 FCP Approach

The FCP approach is between the DCP and UCP techniques. The approach is based on the RSX-11M V3.2 executive's support of the foreign ACP bit in the UCB (US.FOR). When this bit is set, the special processing applied to DCP functions is bypassed. The I/O parameters are merely copied to the I/O packet.

Figure 1-4 shows the flow of an I/O request to a FCP. Functions to be routed to the FCP are marked as legal and ACP functions in the DCB function bit masks. The device state in the UCB must be mountable, mounted, and foreign ACP. When these conditions are met, the QIO parameters are copied without any checks to the I/O packet. This is the same as the processing applied to a DCP. At this point, the I/O packet is routed to the FCP by DRQIO if the queue disable bit is not set (UC.QUE). This is acceptable only if no special processing needs to be applied to the packets.

If this condition cannot be met, the same approach used for UCP's must be applied. The packet must be passed without queueing to the driver initiator which then applies any special processing needed. When completed, the driver can either queue the packet directly to the FCP or queue it to itself and let $GTPKT dequeue the packet and send it to the FCP.

While FCP's are known to the executive, the same processing is applied no matter what the I/O function. The following is a summary of the FCP approach:

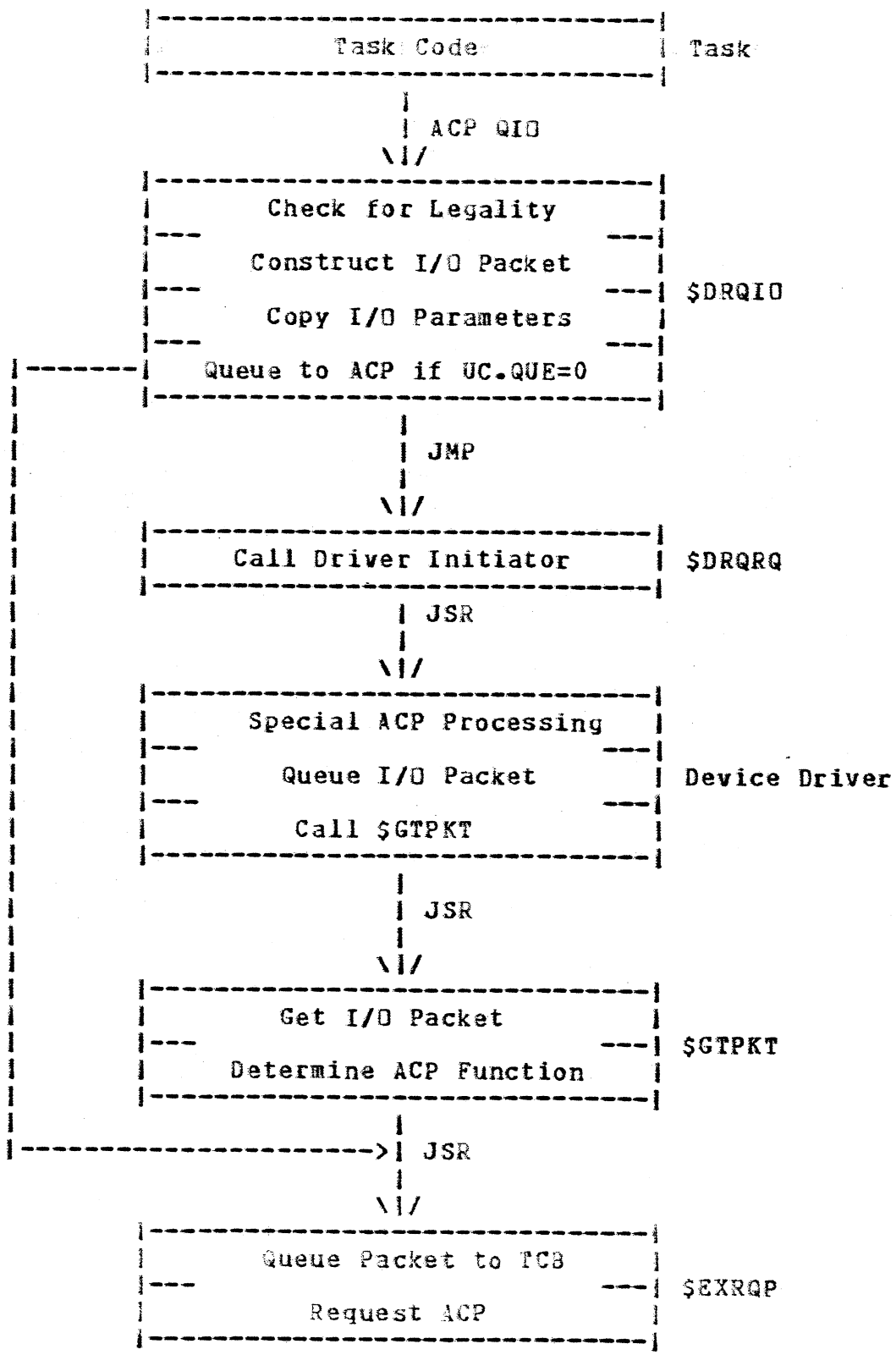


Figure 1-4
FCP FLOW

1. The I/O functions processed by the FCP are marked as legal and ACP functions in the UCB function masks.

2. The device is marked as mountable, mounted, and foreign in the UCB. Any necessary special packet processing is performed by the device driver and the queue disable bit (UC.QUE) must be set.

3. The executive only copies the QIO parameters to the I/O packet. As stated above, any special processing is must be done by the device driver

4. The I/O packet can be queued to the FCP by DRQIO, the device driver, or GTPKT.

The advantage of the FCP over the UCP approach is not in the I/O packet processing. The executive services are minimal and like all Digital software, subject to change. The advantage is the support provided by the MOU and DMO task for mounting and dismounting the foreign ACP.

1.6 ACP APPLICATIONS

ACP's can be written for a wide variety of applications. One type of application is a non-Files-11 file system, particularly, if the ACP is interfaced to FCS/RMS for device/file independence. Two examples are listed below:

1. If a site supports mixed systems (RT-11 and RSX-11M for example), an ACP could be implemented to support the foreign file structure. If interfaced to FCS, the RSX-11M utilities could be used to access the foreign disk directly.

2. If a site has a requirement to process foreign magtapes, an ACP could be written for this task. This is basically what occurred when MTAACP was introduced by Digital.

ACP's could also be used to implement communications protocols. Digital took this approach with DECNET. A user-written ACP could be used to communicate with other computers via some established protocol (BYSYNC, SDLC, etc.) or a site-specific protocol. Alternatively, an ACP could be written to replace NSP within the DECNET architecture. This ACP could implement a specialized link level protocol.

Another class of applications is be to extend the functionality of a current Digital device. For example, one

method of implementing transparent line printer spooling would be via a line printer ACP. This task would intercept write requests to the line printer and output them to a disk file. When the output is terminated, the file is queued to the printer.

ACP's can also be used to solve application problems. For example, an application system may use a complex file organization for storing and retrieving data. Usually with such implementations, special subroutine libraries have to be developed and linked to each task to access the information and coordinate updating the data. An ACP could be written that would essentially implement a special database management system for the application.

Many other applications for ACP's could be listed. Appendix F list various user-written ACP's that have been reported to the author. Other users who have written ACP's are invited to submit a summary of their implementation to be included in future versions of this manual.

CHAPTER 2

DATA STRUCTURES

ACP's, like most system tasks, deal with the RSX-11M system data structures. In the case of ACP's, some of the data structures are integral to the ACP concept. This chapter examines the RSX-11M data structures in the context of ACP usage. The emphasis is on the global relationship between the data structure, the RSX-11M I/O mechanism, and the ACP. Appendix B presents a detailed view of the data structures mentioned in this chapter.

The data structures of interest to ACP's can be divided into seven categories: QIO data structures, device driver data structures, common ACP data structures, ACP specific data structures, FCS data structures, RMS data structures, and other structures. This chapter deals with the first four areas. The FCS and RMS structures are discussed in Chapter 4. This chapter will also touch on the how some of the other data structures are used by ACP's.

2.1 QIO DATA STRUCTURES

The QIO data structures are used to package the I/O requests and route the request to the correct device. The data structures used to package I/O requests are the QIO directive parameter block and the I/O packet. The QIO directive parameter block is formed in the issuing task. The I/O packet is the executive representation of this structure. The data structure used to route I/O packets to the correct device is the logical unit table. This table is found in the task header and is used to map a logical unit number to a specific device.

2.1.1 QIO Directive Parameter Block

The QIO directive parameter block (DPB) is the form an I/O request takes in a user task. The block is normally constructed by the QIO$ and QIOW$ macros. All I/O requests, including requests to ACP's, are issued from a task by using the QIO$ and QIOW$ directives.

While the format of a QIO DPB is constant, some special consideration must be given the DPB when writing a new ACP. The ACP design must consider the I/O functions it will service and the format of the six QIO parameters.

The I/O function is a two byte field. The executive only considers the high byte in processing an I/O request. This byte is the function code and has a legal range of 0-31(10). The low byte is the subfunction field. All values are legal for this byte. The subfunction codes are used by drivers and ACP's to further delimit I/O functions. The only requirement for the subfunction code is that all subfunctions of a particular function code must be the same type of request (transfer, control, etc.).

Some rules also apply to the function code byte. Some codes have been traditionally reserved for special I/O functions; in particular, codes 0-7 all have traditional meanings and should not be considered for new ACP I/O codes. Also, if the new ACP follows the DCP implementation, the executive $GTPKT routine requires the ACP function codes to range from 7-31(10).

Because the I/O parameters for an ACP function are usually processed specially, there is almost complete freedom in what parameters are used. The only RSX-11M requirement applies to transfer functions. For such I/O requests, the executive expects the first parameter to be the address of a buffer and the second parameter to be the size of the buffer in bytes.

2.1.2 I/O Packet

The I/O packet is the internal representation of an I/O request. DRQIO allocates the I/O packet from system pool and initializes it with values taken from the QIO directive parameter block. The packet is then queued to the appropriate driver or ACP for servicing.

When writing a new ACP, the process of initializing the I/O packet parameter fields must be considered. For DCP's, this is the main purpose of the special ACP code in DRQIO. A typical

ACP I/O request may pass several addresses and special parameters. Each address must be checked and relocated before stored in the packet. In some cases, the parameters passed to the ACP have little physical resemblance to the original QIO parameters. The special ACP processing has repackaged the information for the ACP.

The rest of the I/O packet is typically of no interest to ACP's, with two exceptions: the function code and the address of the second lun word. The executive copies the function code from the QIO directive parameter block into the I/O packet. The function code can then used by the ACP to dispatch the packet to the appropriate function processor.

The second lun word address field is the one part of the I/O packet which is not typically used by device drivers. This location contains the address of the second word in the logical unit table entry for the lun of the I/O request. (see next section). The importance of this word will be explained in the section on window blocks.

2.1.3 Logical Unit Table

The purpose of the logical unit table (LUT) is to map logical unit numbers to devices. The LUT table is also used to map luns to I/O processes. The table is a portion of the task header and consist of a two word entry for each possible logical unit a task may use. The size of the table is fixed at taskbuild time by the UNITS option.

TKB fills the table with the ASCII device name and unit number. When a task is installed, the first word of each entry is replaced with the device UCB address and the second word is zeroed. When an I/O request is issued, the logical unit number maps the appropriate entry in the LUT. Any redirect pointers are followed and the I/O packet is dispatched to the resulting device. This may result in the packet being queued to an ACP serving the device.

The second word of the LUT entry is used by ACP's to map I/O processes to logical unit numbers. The address of the window block is kept in this word. A window block is the I/O process unique data structure and is discussed in a latter section. An example of window blocks is the F11ACP window blocks used to map virtual blocks to logical blocks. Note, an ACP window block is different from a PLAS or task mapping window.

The second LUT word has two other uses important to ACP's. Bit 0 of the word is the lun lock bit. When this bit is set,

the executive prevents the user task from issuing any QIO's for the logical unit. This can be used by ACP's to insure proper synchronization between a request it is currently servicing and the user task.

The other use of the second LUT word is to notify ACP's when a task exits with outstanding I/O processes. A task is not permitted to exit until all second LUT words in the logical unit table are zero. When a non-zero entry is found, the executive queues a special I/O request to the ACP. The ACP is responsible for terminating the I/O process and zeroing the second LUT word.

2.2 DEVICE DATA STRUCTURES

The device data structures are used to describe the generic device type, each separate device, and each hardware controller. The Device Control Block (DCB) names the device and contains the I/O dispatch tables. There is one DCB for each device type. The Unit Control Block (UCB) contains the information for each device unit. The Status Control Block (SCB) is used to coordinate activity among device controllers. There is one SCB for each controller. The RSX-11M executive typically permits only one I/O request to be outstanding for each controller. If the controller supports multiple units (UCB's) but requires serial access, one SCB will be used for all UCB's. If the controller permits multiple access, one SCB will be allocated for each UCB.

Most of the information in the device data structures is specific to the needs of device drivers. ACP's are usually only concerned with a few fields. The next three sections comment on how the DCB, UCB, and SCB are involved with ACP implementation.

2.2.1 Device Control Block

The Device Control Block (DCB) names the device, points to the device unit structures and controls how I/O functions for the device are processed. This last feature is important to ACP implementation. The DCB function masks decide whether a particular I/O function is legal or invalid. They further classify legal functions into four types: control, noop, ACP, and transfer. When implementing a DCP or FCP, functions which should be routed to the ACP are marked as legal and ACP in the DCB function masks. This causes the special packet processing in DRQIO for ACP functions to be invoked. When implementing a UCP, the functions are marked as legal and control.

2.2.2 Unit Control Block

The Unit Control Block (UCB) is the key device structure. There is one UCB for each separate device. The UCB's are the one device structure that are variable in length. The only requirement for UCB's is that all UCB's belonging to a DCB be allocated contiguously and that all are the same size.

When a device supports an ACP, the UCB's are extended at least two words beyond the normal length. These additional words are used to hold the Task Control Block address of the ACP (U.ACP) and the Volume Control Block address (U.VCB). The address stored in U.ACP is the task TCB address the executive will use when queueing ACP I/O packets.

The flag fields in the UCB are also of interest to ACP's. The control flags byte (U.CTL) are used to decide how various I/O requests are to be handled. One bit in this field is the UC.QUE bit used by UCP and FCP implementations to pass the I/O packet to the device driver for ACP processing. The unit status flags byte (U.STS) contains bits which define whether the unit is busy (US.BSY), mounted (US.MNT), foreign (US.FOR), or marked for dismount (US.MDM). The correct manipulation of all of these bits is necessary for the correct operation of an ACP. A device is marked busy whenever an I/O function is dequeued for the device driver. The busy bit is cleared whenever an I/O function is completed by calling $IODON. A function will not be dequeued for a device if it is marked busy. Because ACP functions are not processed by the device driver, the driver is not marked busy when an ACP function is dequeued and passed to the ACP. This allows the driver to continue to service functions and requires ACP's to not clear the busy bit when they finish an I/O request. The mounted bit signifies an ACP is mounted for the device and can process I/O requests. When an device is marked mounted, U.ACP and U.VCB are assumed to contain valid values. The US.MDM bit signifies a dismount request has been made for the unit. No new I/O processes should be allowed to start and when all outstanding processes are completed, the ACP should mark itself not mounted by setting US.MOU. The ACP exits when all devices are finally dismounted. Finally, the US.FOR bit is the indication to the executive that the foreign ACP logic should be invoked.

2.2.3 Status Control Block

The Status Control Block (SCB) is used to define and control access to the hardware controller. ACP's are usually not concerned with this data structure unless they are closely coupled to the device driver.

2.3 ACP COMMON DATA STRUCTURES

Two data structures are common to all ACP's: Volume Control Blocks and Window Blocks. The Volume Control Blocks are used to hold information about each separate device an ACP services. The Window Blocks hold the I/O process information. While all ACP's typically use these structures, their format is ACP dependent.

2.3.1 Volume Control Block

One feature of ACP's is their ability to service several different devices. The data structure used to keep the information for each device is called a volume control block (VCB). A VCB is allocated from the system pool when an ACP is mounted for a device. The address of the VCB is stored in offset U.VCB of the device UCB. The VCB is returned to the pool when the ACP completes dismounting the device.

The length and format of a VCB is ACP-dependent. Only the first word has a common meaning across ACP's. This word of the VCB is the volume transaction count and is used as a counter of the number of outstanding I/O requests queued to the ACP for the device. It also counts the number of I/O processes created by the ACP for the device. The ACP cannot dismount the device until the transaction count reaches zero, indicating no activity for the device.

For DCP and FCP implementations, the executive increments the transaction count whenever it queues an I/O packet to the ACP. It is the responsibility of the device driver to increment the transaction count when it queues a packet to a UCP. The ACP is then responsible for decrementing the counter when it completes an I/O function and incrementing and decrementing the counter when it creates and destroys I/O processes.

The remainder of the VCB can be used for any purpose desired by the implementation. The typical information kept in a VCB is any data which is unique to the volume. For example, F11ACP keeps the index file mapping information and volume defaults in disk VCB's.

2.3.2 Window Control Block

As mentioned before, one powerful feature of ACP's is their ability to establish a mapping between I/O requests. For

example, F11ACP maps read and write virtual block I/O to logical disk blocks. The information necessary to do this mapping is established when the file is opened.

This ability to create an "I/O process" uses a data structure called window blocks. These structures are completely ACP dependent in length and format and contain whatever information the ACP needs to keep for each separate process. Typically, window blocks keep access masks, pointers to other ACP unique data structures, and retrieval information for read and write requests.

Window blocks are allocated by the ACP when it creates an I/O process. While the window block is typically allocated from system pool, it can be allocated from the ACP task space if only the ACP accesses the window. The window block address is stored in the second word of the logical unit table entry. This address is passed in the I/O packet in offset I.LN2. Further I/O requests from the same logical unit will always reference the same logical unit table entry, allowing the ACP to retrieve the window block address.

NOTE

The address of the second LUT word cannot be kept internally by the ACP. The LUT table is located in the task header, which is destroyed and reallocated if the task is checkpointed. An ACP cannot depend on any location in the task header being the same across I/O requests.

2.4 ACP SPECIFIC DATA STRUCTURES

ACP's also can create their own data structures. For example, F11ACP creates File Control Blocks (FCB) for each open file. The type and extent of data structures created by a user-written ACP is one of the keys in the ACP design.

One guideline to ACP specific data structures is to minimize the usage of system pool. Any information used only by the ACP should be allocated first from an internal ACP pool and overflow to system pool only if allocation fails. One way to minimize system pool usage is to use the window block to point to the internal ACP data structures, allowing I/O process structures to be mapped on a process basis instead of keeping all information in the window block. Similarly, volume control blocks can point to information in the ACP pool which is used

only by the ACP.

2.5 OTHER DATA STRUCTURES

The structures mentioned so far are the key structures to ACP's. Other structures an ACP may need to use include the Task Control Blocks (TCB), task headers, and Partition Control Blocks (PCB). Depending on the application, any and all RSX-11M data structures may be used by an ACP.

One important structure to an ACP is its own TCB. An I/O request is queued to an ACP by placing it in the ACP receive queue. The ACP dequeues the I/O packet by getting its TCB address and removing an entry from the receive queue. The technique for this operation will be shown later.

CHAPTER 3

ACP CHECKLIST

ACP implementation is a paradox. ACP's are integrally coupled to the executive. It appears that a comprehensive knowledge of the RSX-11M design philosophy, data structures, and executive code is needed to implement an ACP. However, after implementing an ACP, it is obvious that the coupling to the executive is loose, logical, and straightforward. It is simple to interface an ACP to RSX-11M; any difficulties come from the internal ACP design.

This chapter walks through the design of an UCP-style ACP. In order to keep the emphasis on the executive/ACP interface, the example ACP serves no useful function. Rather, the ACP merely demonstrates the functionality available to an ACP. The chapter emphasizes the empirical rules that apply to ACP design. Later material will develop the rationale for the logic presented here.

NOTE

Included with the distribution of this manual are the sources for the sample ACP, its associated driver, and the enable and disable task. It would be useful to obtain a listing or the sources and refer to them when studying this material.

3.1 QIO DESIGN

The first step in ACP design is to decide what I/O services the ACP will provide. It is assume the nature of the services needed is known; these fall out of the reasons an ACP is the chosen solution. This section deals with selecting the I/O function codes to be used, the format of the I/O parameters for each function, and what relationships will the user I/O requests

have to each other.

The example ACP supports five user functions. The first function creates an I/O process. Another function terminates the process. Two other functions perform input and output to the process. The last function is a control function to a created process. The sample ACP services the requests by outputting a message to the console terminal that the request was processed and returns success. The driver serviced by the ACP supports two I/O requests, read and write logical block.

3.1.1 Function Code Selection

The choice of I/O function codes is fairly straightforward. If the ACP is emulating F11ACP for a new device, the Files-11 QIO functions will be used. If the ACP will provide unique I/O services, new QIO codes should be chosen.

The latter is the case with the sample ACP. It is advantageous not to redefine existing I/O codes because this could have undesirable effects if I/O is directed to the wrong device. For the sample ACP, the I/O function code 31(10) will be used for all user requests. The advantage of using this function code is that it is the least used by Digital device drivers and is not used by any Digital ACP. The function modifier field will be used to differentiate the separate requests. The following codes will be assigned:

```
IX.CRE 037,000 Create process
IX.CLO 037,001 Terminate process
IX.PUT 037,002 Output to process
IX.GET 037,003 Input from process
IX.CTL 037,004 Process control
```

ACP's also typically service ACP control functions such as mount and dismount ACP, and abort process. The sample ACP will use the traditional I/O function codes for these services:

```
IO.MOU 005,000 Mount ACP
IO.DMO 006,000 Dismount ACP
IO.CLN 007,000 Abort I/O process
```

The IO.MOU and IO.DMO symbolics are not defined by Digital. However, tradition has used these symbolic names and values as local definitions. The IO.CLN symbolic is defined and the value must be associated with this function.

Finally, the function codes for the driver must be chosen. For the sample driver, the traditional read and write function

codes are used, IO.RLB and IO.WLB.

3.1.2 I/O Parameter Selection

The next step in designing the QIO's is to decide on the format of the I/O parameters to be used for each I/O request. Because ACP functions are processed uniquely, you have almost complete freedom in this area. However, the ACP and driver implementations will depend heavily on the choice of I/O parameters and the format chosen for each I/O request. This is a critical area in the design process.

The parameters for the sample ACP I/O requests have been chosen mostly to reflect some of the more common features found in ACP parameter processing. The following is the format for each request. In order to keep the logic simple, the same parameter format is used for each I/O function. The only difference between functions is whether a particular parameter is required or not present (zero) for the function.

Parameter #1 - If present, the parameter is the output buffer address. Byte alignment is allowed for the buffer

Parameter #2 - If parameter #1 is present, this parameter is the size of the output buffer in bytes. The size may be byte aligned.

Parameter #3 - If present, the parameter is the input buffer address. Byte alignment is allowed for the buffer.

Parameter #4 - If parameter #3 is present, this parameter is the size of the input buffer. The size may be byte aligned

Parameter #5 - If present, the parameters is the access control value. The parameter has legal values from 0 to 3. A zero indicates no process I/O is allowed. A value of one indicates IX.PUT functions are allowed and a value of two indicates IX.GET functions are allowed. A value of three allows both type of process I/O

Parameter #6 - This parameter is never user and must always be zero.

The following table maps what I/O parameters are required (R) and illegal (blank) for the five ACP I/O requests:

I/O Code	P1	P2	P3	P4	P5	P6
IX.CRE					R	
IX.CLO						
IX.PUT	R	R				
IX.GET			R	R		
IX.CTL					R	

3.1.3 Relationship Between I/O Requests

The final step in QIO design is to decide the relationship between I/O requests. The typical requests serviced by an ACP have relationships defined by the I/O process. For example, it would be illegal to perform an input before the I/O process is established. The general rules that must be applied to each ACP I/O request are as follows:

1. Must the ACP be mounted to service the request? This rule applies to all functions serviced by the ACP. I/O request serviced by the driver may be allowed if no ACP is mounted. If an ACP is mounted, I/O requests serviced by the driver may only be allowed from privileged tasks.

2. Must the ACP not be marked for dismounting to service the request? If an ACP is marked for dismounting but has not yet completed the dismount, no new I/O processes should be allowed. However, all current I/O processes should be allowed to run to completion.

3. Must an I/O process be established to service the request? This rule applies to functions which the ACP will map to an I/O process. Typically, input, output, and terminate process functions cannot be serviced until the I/O process is established.

4. Must an I/O process not be in effect to service the request? This is the reverse of the above rule. A new process cannot be created until the previous process has terminated.

Any other rules that make sense for your design may be applied. In the case of the example ACP, the only special rule applies to the IX.PUT and IX.GET functions. When the process is

created using the IX.CRE function, parameter #5 specifies the type of process I/O allowed (see section 3.1.2). The access parameter can be changed by the IX.CTL function.

The following table summarizes how the four common rules listed above (R1-R4) and the one special case rule (SC) are applied to each ACP function. An entry of "A" means the rule is applied. Otherwise, the particular rule does not apply to the function.

I/O Code	R1	R2	R3	R4	SC
IX.CRE	A	A		A	
IX.CLO	A		A		
IX.PUT	A		A		A
IX.GET	A		A		A
IX.CTL	A		A		

3.2 DEVICE DATA BASE DESIGN

After it is determined how a user program will interface to the ACP, the next step is to add ACP support to the device data structures. This is a simple step. The basic structures documented in the Guide to Writing an I/O Driver manual still apply. Adding ACP support to the driver merely involves setting up some of the fields not normally used.

Only the device control block and unit control block need to be considered when adding ACP support to a driver. The status control block (SCB) is strictly driver related.

3.2.1 Device Control Block

The Device Control Block (DCB) is used to name the device and control the executive processing of I/O packets. For an ACP supported device, the documentation on pages 4-7 to 4-15 of the Guide to Writing an I/O Driver applies. The additional step necessary for the support of the ACP is to decide on how the ACP functions will be processed and therefore, how the DCB function masks should be set up.

If a DCP or FCP style ACP is being implemented, the

functions designated for the ACP will be marked as "legal and ACP" in the mask field. If a UCP is being implemented, the functions will be marked as "legal and control".

The following shows how the DCB function masks are set up for the sample ACP. Note how the IO.RLB and IO.WLB functions are marked as legal only, implying that the functions are transfer functions and will be handled by the driver. The IX.??? functions are marked as "legal and control". Also note that the special IO.CLN function is marked as "legal and control" but the mount and dismount requests are not included in the DCB masks. These functions are queued directly to the ACP by the enable and disable tasks and are not processed by the executive.

```
.WORD   000206  ;Legal   (IO.CLN, IO.WLB, IO.RLB)
.WORD   000200  ;Control (IO.CLN)
.WORD   000000  ;No-op
.WORD   000000  ;ACP
.WORD   100000  ;Legal   (IX.???)
.WORD   100000  ;Control (IX.???)
.WORD   000000  ;No-op
.WORD   000000  ;ACP
```

3.2.2 Unit Control Block

The Unit Control Block is the key structure as far as ACP's are concerned. The basic UCB design is documented in the Guide to Writing an I/O Driver Manual (pages 4-19 to 4-26). The fields in the UCB that must be properly set up for ACP's are as follows:

U.CTL - Unit Control Masks. The bit masks in this field will typically be set up for the driver's requirements. The one bit which may be turned on for ACP support is UC.QUE. This bit tells the executive to call the driver with the I/O packet without queuing first. This is the mechanism used for FCP and UCP style ACP's that allows the driver to specially process the ACP I/O packets.

U.STS - Unit Status Masks. Some of the bit masks in this field are used to mark whether an ACP is mounted, foreign, and/or marked for dismount. The field will be initialized as not mounted (US.MNT=1) for all ACP's and foreign (US.FOR=1) for FCP's. The ACP is then responsible for properly handling the states of these bits and the marked for dismount bit (US.MDM).

NOTE

The sense of the US.MNT mask is opposite to most masks. The bit is set if the ACP is not mounted and cleared when the ACP is mounted.

U.CW1 - Device Characteristics Word #1. The word contains bit masks which define the device characteristics. The masks are used by RSX-11M to properly handle different devices. For example, FCS uses this word to decide whether disk or terminal I/O is being performed.

For ACP's, the DV.MNT bit signifies that an ACP can be mounted for the device. The DV.COM and DV.F11 bits are used by RSX-11M MOU and DMO tasks to decide what ACP should be mounted. If a DCP is being implemented, the characteristics that apply to currently supported Digital devices should be used for the new device/ACP.

If an FCP or UCP is being implemented, only DV.MNT needs to be set. In addition, the low byte of the word should be set to accurately reflect the nature of the device itself. A description of the bit masks for the U.CW1 field can be found in Appendix B.

U.ACP - ACP TCB Address. This word is an extension to the normal UCB used by RSX-11M. The word must be allocated for devices which are connected to ACP's and follows the U.CNT field. When an ACP is enabled, the word will contain the address of the ACP's Task Control Block. This the the TCB the I/O packet will be queued to when it is sent to the ACP. U.ACP is initialized when the ACP is enabled.

U.VCB - Volume Control Block Address. This word is also an extension to the normal UCB and follows U.ACP. The word contains the address of the volume control block. It is initialized when the ACP is enabled.

The following code segment is the UCB for the example device. The values are set for a UCP-style ACP implementation.

```
        .IF DF  M$$MUP

        .WORD   0       ;(U.OWN ) OWNING TERMINAL UCB ADDRESS

        .ENDC

.AC0::                  ;REF. LABEL.
        .WORD   .ACDCB  ;(U.DCB ) POINTER TO DCB
        .WORD   .-2     ;(U.RED ) REDIRECT UCB POINTER
        .BYTE   UC.QUE  ;(U.CTL ) CONTROL FLAGS
        .BYTE   US.MNT  ;(U.STS ) STATUS FLAGS
        .BYTE   0       ;(U.UNIT) UNIT NUMBER
        .BYTE   US.RED  ;(U.STS2) STATUS FLAGS
        .WORD   DV.MNT  ;(U.CW1 ) DEVICE CHARACTERISTICS
        .WORD   0       ;(U.CW2 ) DEVICE CHARACTERISTICS
        .WORD   0       ;(U.CW3 ) DEVICE CHARACTERISTICS
        .WORD   1000    ;(U.CW4 ) BUFFER SIZE
        .WORD   $AC0    ;(U.SCB ) SCB POINTER
        .WORD   0       ;(U.ATT ) ATTACH WORD
        .WORD   0,0     ;(U.BUF ) BUFFER RELOCATION ADDRESS
        .WORD   0       ;(U.CNT ) BUFFER SIZE (BYTES)
        .WORD   0       ;(U.ACP ) ACP TCB ADDRESS
        .WORD   0       ;(U.VCB ) VCB ADDRESS
```

3.3 ACP DATA BASE DESIGN

The final step in the general ACP design is the ACP specific data bases. Here, the ACP implementation has complete freedom as RSX-11M imposes very few rules.

3.3.1 Volume Control Block

One data structure common to all ACP's is the volume control block. This structure is allocated from the system pool when an ACP is mounted for a device. The VCB is linked to the UCB by storing its address in U.VCB.

The VCB is used to hold all unit specific information for the ACP. The ACP philosophy allows several devices to use the same ACP. Typically, the ACP treats each device as an independent entity; therefore, a separate data base is need. The VCB serves this purpose.

The length and format of VCB's is ACP dependent. Only the first word has a common usage. This is the transaction count and is used to count the number of outstanding I/O requests and processes for the device. When the VCB is allocated, this word

should be zeroed.

3.3.2 Window Block

The other structure common to most ACP's is the window block. This structure is completely ACP dependent and need not be even allocated from pool. Window blocks are the structures used to keep I/O process-dependent information. The window is allocated when the process is created by the ACP. The address of the window is stored in the second LUN word of the LUT table for the task which issued the I/O request. The LUT table address can then be retrieved from future I/O packets from the same lun, and from it, the window address.

Window blocks contain whatever information is needed by the ACP to map the individual I/O request to the I/O process. For example, F11ACP windows contains the mapping information between the logical blocks of a file and the physical disk blocks. Window blocks also typically point to other data structures used by the ACP and form the key structure for the I/O process.

3.3.3 Other Data Structures

Finally, ACP's may create new data structures. Any ACP implementation should keep the system pool utilization to a minimum. One method of accomplishing this is to use an ACP task pool for the structures used only by the ACP and overflow into the system pool only when the internal pool is exhausted. The VCB and window blocks, which are usually in the system pool, are kept short and point to the new data structures. This technique is used by F11ACP for File Control Blocks.

3.4 DEVICE DRIVER IMPLEMENTATION

Once the design of the QIO requests and data bases is finished, the next step is to implement support for the ACP in the executive (DCP) or device driver (FCP, UCP). This section follows the example UCP, so the special support will be added to the sample device driver.

In most cases, the ACP is being used with a new device driver. If this is the case, the recommendation is to implement and test the device driver without ACP support. Once the driver is functional, it is a small step to add the ACP.

If the implementation uses a DCP, the driver is usually unaware of the ACP; not driver modifications are necessary. The ACP packet processing is done in DRQIO. In the case of FCP's and UCP's, the code to support the special processing for the ACP I/O packets is in the driver. In any case, the functionality provided by the code is the same. The next two subsections discuss the considerations on driver entry and ACP packet processing.

3.4.1 Driver Entry

For FCP's and UCP's, the driver is called directly from the executive and the I/O packet is not queued, because UC.QUE in the UCB is set. The driver logic then determines if the packet is for the ACP or for itself. If the latter, the packet is queued to the SCB and the driver calls $GTPKT to correctly sequence I/O service. The driver then processes the packet in the typical fashion. For ACP packets, the special packet processing discussed in the next section is used.

The following code segment illustrates how the driver works. The example is taken from the driver for the example presented so far.

```
;
; DRVINI IS CALLED WHENEVER A I/O REQUEST IS ISSUED TO
; THE SAMPLE DRIVER. THE PACKET IS NOT QUEUED AND THE
; FOLLOWING REGISTERS ARE SET:
;
;       R1 = I/O PACKET ADDRESS
;       R4 = SCB ADDRESS
;       R5 = UCB ADDRESS
;
DRVINI: MOVB    I.FCN+1(R1),R0  ;GET FUNCTION CODE
        CMPB    #IX.ACP/400,R0  ;IS THIS A ACP REQUEST?
        BEQ     DRVACP          ; IF EQ - YES, GO PROCESS
        CMPB    #IO.CLN/400,R0  ;IS THIS A IO.CLN REQUEST?
        BEQ     DRVACP          ; IF EQ - YES, GO PROCESS
;
; QUEUE PACKET TO DEVICE AND GET NEXT PACKET.
;
DRVPKT: MOV     U.SCB(R5),R0    ;GET DEVICE QUEUE LISTHEAD
        CALL    $QINSP          ;QUEUE PACKET TO DEVICE
        CALL    $GTPKT          ;GET NEXT PACKET
        BCC     DRVSRV          ; IF CC - SERVICE REQUEST
        RETURN                  ;NO PACKET, RETURN
```

3.4.2 Driver Packet Processing

Once the driver decides the packet is for the ACP, it performs the special packet processing and queues the packet to the ACP. There are any number of special checks that could be performed and any number of methods to perform the check. The Digital approach used in DRQIO is to use polish-driven routines to perform each check. This allows a list of routines to be specified for each type of I/O function. The same approach is used in the sample driver.

More important than the technique used to process the I/O packet is the type of processing performed. The typical processing performed before queueing the packet to the ACP is listed below (the order of the list is basically the order the items would be applied; however, this is not hard and fast). The list of items below are typical of the processing applied by the executive for DCP style ACP's. The example driver code also demonstrates all of the below operations in its processing of the ACP functions. The rule which decides if the executive/driver or the ACP should perform the processing is whether the processing needs to be performed in the context of the task which issued the I/O request. If not, the ACP can perform the operation. Otherwise, the operation must be performed by the executive for DCP's and by the driver for FCP's and UCP's. Once a packet is queued to an ACP, an indeterminate amount of time and events may happen before the ACP dequeues and services the request.

1. The function code is verified for legal ACP function. The executive only checks the high byte when processing function codes, the driver needs to check any subfunction fields.

2. The UCB is checked to see if the ACP is mounted for the device.

3. For functions which create I/O process, the UCB is checked to see if the ACP is marked for dismount.

4. For functions which create I/O processes, the logical unit table is checked to see if a process is already active for the lun.

5. For functions which must be mapped to an I/O process, the logical unit table is checked to see if a process is active for the lun.

6. For functions which require a specific relationship to another, the window is checked to see if the function is legal for the current state of the I/O process.

7. If specific parameters are not used or required, the I/O parameters are checked to see if they are zero or non-zero.

8. Any user task addresses are address checked.

9. Any user task addresses are relocated so the ACP can map the user task.

10. In some cases, user buffers are copied to system pool buffers. This allows the ACP to access information without mapping the user task. This is especially applicable to attribute lists.

11. The volume transaction count is incremented.

12. If the ACP requires no further I/O request be issued from the lun, the lun is locked. This is done by setting the low bit of the second lun word in the logical unit table.

13. The packet is queued to the ACP and the ACP scheduled for execution. This is a simple operation which is done by calling the executive routine $EXRQP with R0 set to the ACP TCB address and R1 set to the I/O packet address.

3.5 ACP IMPLEMENTATION

Once the driver interface is finished, the ACP implementation can begin. Some of the implementation will be common to all ACP's. However, most will be unique to the application being addressed.

3.5.1 ACP Packet Dequeuing

One common function of all ACP's is the dequeuing of the I/O packets queued to it from the executive. The executive passes I/O packets to ACP's by linking them to the ACP's receive queue and scheduling the ACP for execution. The ACP retrieves the packet by entering system state and dequeuing the first entry from its receive queue.

The following code segment is taken from the example ACP and illustrates the basic root of an ACP. On task startup, the ACP initializes itself. Once done, it enters a loop where

packets are dequeued and processed. If no work is found, the ACP stops itself. When the executive queues another packet, the ACP will be rescheduled and can repeat the loop.

```
NULACP::                            ;REF. LABEL
;
; GO PERFORM THE INITIALIZATION PROCESS.
;
        CALL    ACPINI              ;INITIALIZE ACP
;
; ENTER SYSTEM STATE AND CHECK RECEIVE QUEUE.
;
1000$:  CLR     R1                  ;MARK NOTHING DEQUEUED
        CALL    $SWSTK,2000$        ;;ENTER SYSTEM STATE
        MOV     $TKTCB,R0           ;;GET OUR TCB ADDRESS
        ADD     #T.RCVL,R0          ;;POINT TO RECEIVE QUEUE LISTHEAD
        CALL    $QRMVF              ;;REMOVE QUEUE ENTRY
        BCS     1100$               ;; IF CS - NO ENTRY, STOP
        MOV     R1,4(SP)            ;;RETURN PACKET ADDRESS
        RETURN                      ;;RETURN TO TASK STATE
;
; STOP TASK. WHEN WOKE, RETURN TO TASK STATE.
;
1100$:  CALLR   $STPCT              ;;STOP CURRENT TASK (US)
;
; IF NO ENTRY DEQUEUED,  LOOP.
;
2000$:  MOV     R1,R3               ;WAS ENTRY FOUND?
        BEQ     1000$               ; IF EQ - NO, CONTINUE LOOP
;
; DISPATCH ON I/O FUNCTION AND CALL PROCESSING ROUTINE.
;
        MOV     I.UCB(R3),R5        ;GET UCB ADDRESS
        <function dispatch code>
;
; WHEN FINISHED, CHECK FOR DISMOUNT PENDING.
;
4000$:  BITB    #US.MDM,U.STS(R5) ;IS A DISMOUNT PENDING?
        BEQ     1000$               ; IF EQ - NO, LOOP THROUGH QUEUES
        CALL    ACPMDM              ;TRY TO COMPLETE DISMOUNT
        BR      1000$               ; NO LUCK - LOOP THROUGH QUEUE

        .END    NULACP
```

3.5.2 ACP Packet Processing

The real work of an ACP takes place after the packet is dequeued. How ACP's process packets is strictly a function of the individual ACP. The sample ACP illustrates some of the common types of processing that may take place, such as reading

and writing buffers in the user task. However, most of code implemented to service the packets will be of your own creation.

3.5.3 ACP Packet Termination

The final step an ACP takes in processing a packet is packet termination. This is done by calling the $IOFIN routine in the executive. This routine, rather than the $IODON/$IOALT routines, should be used since the driver is not busy.

3.6 TASK TERMINATION

One important, but often overlooked, feature of ACP's is properly handling of user task termination. The RSX-11M executive will not let a task exit until all outstanding I/O requests are terminated and all outstanding I/O processes are destroyed. The former is typically a function handled by the I/O driver. In the latter case, the executive checks the second lun word for each entry in the logical unit table. If the value is non-zero, an I/O process is assumed. The value would normally be the window block address.

When a non-zero second lun word is encountered, the executive constructs a IO.CLN packet and issues it as if the task had issued the QIO from the lun with the outstanding I/O process. The ACP is expected to abort the I/O process and zero the second lun word entry.

The logic of the ACP must be able to handle an IO.CLN function no matter what the state of the I/O process. It is impossible to predict when a task will encounter an unexpected trap, be aborted by the operator, or exit without terminating all I/O processes.

3.7 ACP ENABLING

Two final steps remain after the ACP and driver are finished. A mechanism must be implemented to enable and disable the ACP for the device. The common name for these operations is mount and dismount. There are a variety of methods that can be used. These are discussed in detail in chapter 6.

ACP mounting is really quite simple. The complexity of Digital's MOU task comes from the multiple functions it provides

and the details of actually performing a Files-11 or ANSI magtape mount. The essential steps that must be accomplished when an ACP is mounted are listed below. The first list contains steps typically performed by a separate enabling task. The sample task ENA demonstrates each of the steps.

1. If wanted, get any "mount-time" options from the mount command line. It may be necessary to allow operator specified options to provide greater ACP flexibility.

2. Check if the device is mountable (DV.MNT=1).

3. Check if the device is not currently mounted (US.MNT=1).

4. Check if the device is not marked for dismount (US.MDM=0).

5. Allocate the VCB from system pool and store its address in U.VCB of the UCB.

6. Search the task directory for the ACP task and get its TCB address.

7. Check the found task is an ACP (T3.ACP=1).

8. Store the ACP's TCB address in U.ACP of the UCB.

9. If necessary, add an entry to the mounted volume list. This will typically not be required for user-written ACP's.

10. Schedule the ACP task for execution. The ACP will then typically start with its initialization code. This step is traditionally combined with a IO.MOU request being queued to the ACP. This request contains the "mount-time" parameters for the ACP. The IO.MOU request is not necessary if no information needs to be communicated to the ACP.

11. If the IO.MOU I/O request was issued, wait for success/failure. If failure, output an error message. Otherwise, just exit.

The remaining steps in the mounting process are typically performed by the ACP. These steps are as follows:

1. If an IO.MOU request was queued, dequeue it from the receive queue and process the parameters.

2. Count the device mounted. This is usually an internal ACP counter.

3. Perform any device I/O necessary for the ACP to become active.

4. Mark the device mounted (US.MNT=0).

5. Enter the ACP main loop.

The steps above are followed by the sample ACP and ENA task. This code can be used as a template for your ACP implementation, or one of the alternate approaches discussed in chapter 6 can be used. In any case, in order for an ACP to be successfully mounted, the following must have happened:

1. The VCB has been allocated and its address stored in U.VCB.

2. The ACP's TCB address must have been stored in U.ACP.

3. The ACP must have been marked as mounted (US.MNT=0).

4. The ACP must be active and ready to receive ACP functions.

3.8 ACP DISABLING

The last step in the checklist is to implement the ACP disabling (dismount) mechanism. If the system will not function without the ACP, no dismount mechanism other than rebooting may be needed. However, usually a dismounting method will be used. The following steps are usually accomplished by a dismount task. The sample DIS task demonstrates these steps.

1. Check if the device is mountable (DV.MNT=1).

2. Check if the device is mounted (US.MNT=0).

3. Check if the device is already marked for dismount (US.MDM=1).

4. Mark the device for dismount (US.MDM=1).

5. Construct and queue an IO.DMO request to the ACP.

6. Wait for completion of the IO.DMO and exit.

When an ACP receives a dismount request, it typically performs the following steps.

1. Marks a dismount is pending for the device. This can already be considered done as the US.MDM bit is set in the UCB.

2. Return the IO.DMO request.

After a dismount becomes pending, the ACP then waits for all activity to the unit to cease. This means no I/O requests are outstanding and all I/O processes have terminated. When this occurs, the following steps are taken.

1. Mark the device dismounted (US.MNT=1).

2. Mark the dismount complete (US.MDM=0).

3. Perform any device I/O necessary to deaccess the ACP from the device.

4. Return the VCB to the pool and zero U.VCB.

5. Zero the U.ACP word in the UCB.

6. When all mounted units are dismounted, exit the ACP.

3.9 CHECKLIST SUMMARY

This chapter has described all the steps necessary to implement an ACP. The checklist presented here was followed to implement the sample ACP, its driver, and the ENA and DIS tasks. These routines can be used as a basis for a user-written ACP, leaving only the job of solving the actual problem.

It is usually easiest to implement a UCP. This involves no modifications to the executive and is a straightforward approach once the checklist is followed. For the more advanced programmer, the DCP and FCP approaches may offer advantanges. However, you are at the mercy of Digital with regard to their future executive implementations.

The remaining chapters discuss the material presented here in more detail. In particular, they present the philosophy behind the actual operations.

CHAPTER 4

ACP/TASK INTERFACE

A user task interfaces to an ACP using the QIO directive. The I/O requests can be issued directly from the task or from one of the logical I/O service libraries: FCS or RMS. This chapter discusses the ACP/task interface from the first two perspectives. The RMS sources are unavailable to the author. However, the approach used for adding an new ACP to FCS is expected to apply to RMS.

4.1 QIO INTERFACE

The most common way to interface to a user-written ACP is the QIO directive. In an RSX-11M system, all I/O is requested using this directive, including I/O from ACP's. Adding support for I/O service from a user-written ACP is the same as adding new QIO directives for a user-written driver. The only decisions necessary are what function codes the ACP will service and the I/O parameters that will be used for each function.

The choice of I/O function codes is simple. If existing Digital function codes apply to the new ACP, they should be used. When using existing codes, the I/O parameters should also be the same as the current Digital format. Then, no errors will occur when I/O is directed to different devices.

However, the more typical case is that the new ACP will be implementing new functionality. In this case, the less frequently used I/O function codes should be considered. These are the codes from 25-31(10). The subfunction field can be used to actually distinguish between separate ACP functions.

Because ACP's typically use special processing for the I/O parameters, there is a great deal of freedom in their layout. The parameters can be as complex as needed to pass and receive information from the ACP.

One common technique used by ACP's to allow passing of many

parameters are attribute lists. An attribute list is a buffer which contains attribute descriptors. Each descriptor describes a separate parameter to the ACP. A typical attribute descriptor consist of a code to identify the attribute, the address of the actual attribute, and size of the attribute. When the I/O request is processed, the special processing code maps the attribute list and individually address checks and relocates the attribute addresses.

This technique is used by F11ACP to allow reading and writing of the various fields in the file header. When the attribute list is processed, a buffer is allocated from the system pool and the attribute code and size are copied from the user task. The attribute buffer is relocated and the relocation bias and displacement are stored in the system buffer. When the I/O packet is received by F11ACP, it can use the information in the system buffer to read and write the attribute buffers in the user task. Appendix D contains a detailed description of the Files-11 attribute list processing. Also see the DRQIO module for an example of how the executive processes attribute lists.

4.2 FCS INTERFACE

In some special cases, interfacing an user-written ACP to FCS can result in significant advantages. FCS is the component of RSX-11M that provides device independence and record I/O. FCS examines the device characteristics of the assigned device and issues I/O appropriate for the type of device. For example, if a terminal is the assigned device, a PUT$ request will result in a IO.WVB being issued to the terminal. However, if the device is a disk, FCS will block the record to the virtual disk block.

By adding support for new ACP's to FCS, users can perform I/O from FCS to the devices supported by the ACP. In addition, the languages and utility programs which use FCS will then support the new device. For example, Fortran I/O uses FCS. If FCS supports the new ACP, standard Fortran I/O statements could be a powerful method of performing I/O. The users of the ACP do not have to learn any new syntax to use the ACP and its devices.

Only a fraction of user-written ACP's are candidates for interfacing to FCS. The typical application will involve an implementation of an alternate file system. For example, if an ACP was written to support RT-11 disks, interfacing the ACP to FCS would allow the standard RSX-11M utilities to directly read and write RT-11 files. Another possibility is an ACP which supports remote I/O. For example, an ACP could implemented Digital's Data Access Protocol (DAP). If such an ACP was interfaced to FCS, RSX-11M programs could transparently access

remote file and devices.

There are two methods for interfacing a user-written ACP to FCS. If the ACP uses the Files-11 QIO's, the new ACP merely needs to emulate the services supplied by F11ACP. With this approach, little or no modifications would be needed to FCS. However, the ACP will probably have to go to great lengths to translate the device protocol into Files-11 compatible format.

The other approach involves modifying FCS to perform unique operations for the new ACP. Here, the goal is to provide complete compatibility at the user interface into FCS. In other words, the functionality documented in the RSX/IAS I/O Operations Reference Manual is provided by the combination of the modified FCS and the new ACP. As long as a program does not depend on Files-11 specifics, it should function correctly when used with the new ACP.

There are no hard and fast rules which apply to adding support for a new ACP to FCS. The overall approach is to examine how FCS operates for current ACP's and modifying the appropriate points to add logic for the new ACP. The next four subsections comment on some of the basic points. In addition, Appendix C provides detailed descriptions of each FCS module and the various data structures.

Some general rules are to apply conditional assembly statements to all new code and to maintain object-level compatibility. By conditionalizing the new code, different versions of the FCS library can be maintained. For user tasks which do not need to support the new ACP, the Digital standard version of FCS can be used. Only tasks requiring the new device support need to be linked to the special version. By maintaining object-level compatibility, a task needs only to be relinked to receive the benefits of the new support. Object-level compatibility requires that the code generated by the FCS macros is not changed by the modifications. In particularly, the space generated for the File Descriptor Block and Filename Blocks should stay the same. However, fields within these data structures can be redefined by the modified FCS for support of the new device.

4.2.1 File Specification

The FCS modules can be broken into four categories. The first are modules which are concerned with parsing the file specification into the device, directory, and filename components and assigning the user lun to the specified device. The major modules in this category are the four parsing modules: PARSE (parsing control), PARSDV (parse device name), PARSDI

(parse directory), and PARSFN (parse filename). In addition, the ASSLUN module assigns the user lun to the specified device and reads the device characteristics. The auxiliary directory modules (DIDFND, DIFND, GETDI, GETDID, GETDIR, PARDI, and PARDID) process files-11 directory files. Finally, the RSX-11M command scanning routines (CSI1, CSI2) are used to scan entire command lines

When adding support for a new ACP, the major concern for these modules is to add support for any new form of file specifications and to set up the FCS device characteristics. For example, if adding network file support, it would be necessary to add a nodename field to the device specification. If adding RT-11 support, the routines would have to be modified to ignore or declare an error if a directory was specified.

The key is the fact that the first step performed by FCS for all operations is to assign the lun to the specified device and establish the device characteristics. The ASSLUN module performs the assignment and gets the device characteristics. It stores the low byte of the devices U.CW1 field in F.RCTL and the device buffer size in F.VBSZ and F.BBSZ. The important field is the device characteristics byte. This byte contains the bit masks which are examined by FCS to determine the appropriate operation to take for a device. For example, if the record-oriented bit (FD.REC) is set, FCS assumes the device is a terminal-like device and performs no record blocking. If the bit is clear, a block-oriented device is assumed (disk) and records are block within virtual blocks. Therefore, the first step in adding support for a new ACP is to modify PARSDV and ASSLUN to correctly assign the new device and to establish device characteristics which will allow other FCS modules to uniquely identify the new ACP. Then, whenever special processing is needed to support the new ACP, the F.RCTL field can be tested and the a branch made to the appropriate code. For example, special directory and filename parsing logic can be invoked if the device is established as a RT-11 disk volume. The easiest method to identify a new type of device is to use one of the undefined bits in the F.RCTL field (bits 6 and 7) and set the remaining bits as appropriate for the new device.

4.2.2 File Access

The next category of FCS modules are the various OPEN routines. FCS supports three methods of opening files: normal, filename block, and file-ID. The first form calls the parsing routines to setup the filename block and then opens the specified device/file. The filename block version assumes the parsing routines have already been called. The final form uses information from a previous file access to directly open the

file. No parsing logic is invoked.

Adding a new ACP merely involves dispatching at the appropriate point in the OPEN logic to perform any I/O required by the ACP to establish an I/O process. The new logic is responsible for returning the appropriate information to the FDB, particularly the file-attribute section (F.RTYP, F.RATT, F.RSIZ, F.HIBK, F.EFBK, and F.FFBY). These fields are used by FCS to determine how records should be processed, when new blocks need to be allocated to the file, and where the logical EOF is positioned. For Files-11, these fields are stored in the file header. FCS retrieves them when an existing file is opened and stores the current FDB values when a file is created.

Unless the new type of device supports some unique file identifier, the open by file-ID mechanism will either have to be emulated by the ACP or made illegal. The most common use of file-ID's is by programs which open a file briefly, close it, and reopen it at a later point in time. The file-ID from the first open is saved and reused on the next access. For example, the MACRO assembler uses this technique. One method for an ACP to emulate file-ID's would be to save file names in a least-recently used buffer or file and assign a number to map the entry. If an open by file-ID is attempted, the file-ID is used to address the table and retrieve the original file specification. When a new filename is entered, the oldest previous entry is destroyed. In this fashion, the most common use of file-ID's by RSX-11M utilities can be supported.

4.2.3 Input/Output

FCS supports two forms of input/output. The first is record I/O which uses the GET$/PUT$ calls. The other form is virtual block I/O and is implemented by the READ$/WRITE$ routines. The general approach for adding support for a new ACP is the same as for the OPEN routines: add logic at the appropriate points to test for the new ACP and branch to special I/O logic.

It is important to remember that FCS is the element of RSX-11M that defines what is a record, not Files-11. When performing record output to a disk, FCS packs the records into disk blocks and issues write virtual blocks to F11ACP. Similarly, when inputting records from a disk, FCS inputs virtual blocks and unpacks the records using the record information it originally provided when the record was written. Most importantly, user programs do not care that block I/O is being performed for their record I/O. They are only interested in the record itself.

Therefore, if the concept of block I/O does not apply to the new ACP, there is no need for any new logic to be added to FCS to block the user records. Instead, each GET$/PUT$ call can result in an I/O request to the new ACP. FCS block I/O can be treated as merely fixed-length record I/O. An example would be an ACP which implements the DAP protocol. The protocol is only concerned with records, so each GET$/PUT$ call would generate an input/output record request to the ACP. This greatly simplifies the modifications necessary to FCS because all the code concerned with record blocking can be bypassed.

4.2.4 File Control

The final category of FCS routines are used for file control operations: delete, rename, truncate, extend, etc. The modules in this category are straightforward and can be modified for support of a new ACP with little difficulty.

If a particular operation is not appropriate for the new device, it can either be ignored or an error returned. For example, FCS has a user call to enter a filename into a directory. It would typically be appropriate to just return success if this operation does not make any sense to the new ACP.

CHAPTER 5

ACP/EXECUTIVE INTERFACE

The ACP/executive interface is almost totally contained in two executive modules: DRQIO and IOSUB. These modules are concerned with processing user I/O requests and providing the common I/O services used by drivers and ACP's. This chapter discusses the basic principles of ACP I/O processing and walks through the code used for each style of ACP.

5.1 EXECUTIVE PROCESSING REQUIREMENTS

The module DRQIO contains all the code to transform a user I/O request into its internal representation and route the resulting packet to the correct device and/or DCP. For FCP's and UCP's, DRQIO is used to perform the common operations and device drivers are used for any unique processing required by the ACP. For this section, where any particular piece of code is placed is ignored. The emphasis is on the operations that must be performed before an I/O packet can be queued.

Up to a certain point, user I/O requests must be processed in the context of the user task. The following list documents operations which fall into this class.

1. Any address in the user task space must be validated. Specifically, the address of the QIO directive parameter block, I/O status block, AST address must be checked. In addition, any addresses in the I/O parameters must also be address checked.

2. Any address in the user task space to be used by the device driver or ACP must be relocated to a address bias and displacement. In other words, the 16-bit user virtual address must be converted to a physical memory address in order for ACP's to access or store information.

The above list applies equally to I/O for drivers and ACP's. When the executive determines a I/O packet is to be routed to an ACP, there are some other tests which must be applied before the packet can be queued to the ACP. ACP's are running as tasks and are affected by the scheduling algorithms. Therefore, an indeterminate amount of time may elapse before an I/O packet queued to an ACP is actually processed by the ACP. This requires the state of the ACP and any I/O process be checked before the packet is queued and that no changes in the state can be allowed until the packet is dequeued. The following operations fall into this class:

1. The device must be checked to see if the ACP is mounted. This is done by checking the US.MOU bit in the U.STS field. The bit will be zero if the ACP is mounted.

   This rule implies the state of the US.MOU will not change once the I/O request is queued to the ACP until it is dequeued. This is the reason for the transaction count (see below). An ACP should not be marked as dismounted until the transaction count for the device is zero.

2. The device must be checked to see if the ACP is marked for dismount. This is done by checking the US.MDM bit in the U.STS field. The bit will be set if a dismount has been requested.

   When an ACP is in the marked-for-dismount state, no new I/O processes should be allowed. Therefore, this test is only performed for I/O requests which will create an new I/O process. The test will not be applied to any functions which operate on an existing I/O process.

3. For functions which create I/O processes, a check must be made to see if an I/O process is already created for the user lun. This is done by checking the second word in the LUT table entry for the presence of a window block address. If the word is zero, no I/O process is in effect and the request can be allowed.

   This rule implies that an I/O process will not be created for the user lun once the I/O request is queued to the ACP and until it is dequeued by the ACP. This is the reason the the lun lock bit in the LUT table. When the lock bit is set for a user lun, the user task cannot issue any further I/O request from the logical unit. Therefore, the I/O packet processing will set the lock bit when a I/O packet which creates a process is queued to the ACP and clear the bit once the process is created. Similarly, functions which destroy I/O processes will also set the lock bit to insure proper

synchronization between the ACP and the user task.

4. For functions which must be mapped to an I/O process, a check must be performed to see if the I/O process exists. This is done by checking the second word of the LUT table entry for the presence of a window block. If the word is non-zero, an I/O process is in effect and the request can be allowed.

5. Finally, any specific relationships required by the ACP between I/O requests are checked. For example, an ACP may not allow a particular operation until some other criteria have been established. This is done by checking information in the window block.

   Also, the window block may contain enough information for the I/O requests to be mapped at this point and queued directly to the device driver. This is what occurs for I/O to disk files. If the disk window block contains the mapping information for the read/write virtual request, the remapping is performed directly and the packet is sent directly to the disk driver. Otherwise, the read/write virtual request is sent to the ACP which updates the window information and routes the packet to the driver.

The purpose of the transaction count is to insure no change in the state of the ACP can occur between the time the state is checked and the I/O packet is actually received by the ACP. Therefore, an ACP can only be dismounted when the transaction count is zeroed. The transaction count is incremented whenever a packet is queued to the ACP and decremented when the ACP completes the packet. The transaction count is also used to indicate the presence of I/O processes.

Similarly, the lun lock bit is used to insure the I/O process state is not changed between the time the state is checked and the packet is dequeued by the ACP. When this bit is set, the executive guarantees no I/O request can be issued from the lun. Therefore, any I/O request which will change the state of the I/O process should lock the lun before the request is queued to the ACP. The bit does not need to be set if the I/O request will not affect the process state.

5.2 DCP INTERFACE

All of the operations described by the previous section can be observed by reading how I/O requests are processed for DCP's. The overall process is described by the RSX-11M Guide to Writing

an I/O Driver manual. In addition, the RSX-11M System Logic Manual contains a flow-chart of the executive processing in Chapter 6.

The above descriptions are oriented towards processing of functions which will eventually be routed to device drivers. The next two sections outline the processing that occurs for a I/O function which will be routed to a DCP. The following notes assume the I/O request is to be routed to F11ACP and that the following features have been selected/not selected/optional:

A$$CHK	Address checking	selected
A$$CPS	ACP support	selected
D$$IAG	On-line diagnostics	not selected
L$$DRV	Loadable drivers	optional
M$$MGE	Memory management	selected
M$$MUP	Multi-user protection	optional

5.2.1 $DRQIO Processing

All QIO directive processing starts at the label $DRQIO in the DRQIO module. The first operations apply to all I/O functions. The initial code checks that a QIO is legal at for the current task and device state. Then the parameters to the QIO are verified. The following checks are made. If all checks prove false, processing continues with at label 24$.

1. The lun is mapped to the device and any redirect pointers are followed. If the lun is illegal ([Q.IOLN] > [H.NLUN]), a directive error of -96. is returned.

2. If the lun is unassigned ([first lun word] = 0), a directive error of -5. is returned.

3. If the lun is interlocked for use (bit 0 [second lun word] = 1), the task PC is backed up to reissue the QIO and the task is placed in WAIT FOR SIGNIFICANT EVENT state.

   NOTE

   This is the feature use by ACP functions to synchronize access to the I/O processes.

4. If the driver is not resident ([D.DSP] = 0), a directive error of -6. is returned.

5. If an event flag was specified, clear it. A directive error of -97. is returned if the event flag is illegal.

6. If an I/O status block address was specified, check the address and clear both words.

7. Allocate an I/O PACKET. If this fails a directive error of -1. is returned.

NOTE

This is the last possible directive error. Both the event flag and I/O status are cleared before this error can be detected.

8. Increment the outstanding I/O count ([T.IOC] <- [T.IOC]+1).

9. If the function was QIOW and an event flag was specified, place the task in WAIT FOR EVENT FLAG state for the specified event flag.

If no directive errors occured, the common fields in the I/O packet are filled in. The code that performs this step starts at label 24$. The next step is dispatch based on the type of I/O function. The following steps are performed:

1. If the I/O function code was kill I/O ([Q.IOFN+1] = 0), control is passed to the kill I/O code. This code calls $IOKIL and returns success for the kill I/O function.

2. If multiuser support was selected (M$$MUP), the I/O request access to the device is checked. Access is denied and an I/O status code of IE.PRI is returned if the all of the following are true.

   1. The device is owned ([U.OWN] # 0).

   2. Public access is not allowed (bit US.PUB, [U.ST2] = 0).

   3. The current owner is not the task that issued the I/O request ([U.OWN] # [T.UCB of current task]).

   4. The current task is not privileged (bit T3.PRV, [T.ST3] = 0).

3. The I/O function code is checked to make sure it does not exceed the maximum (31.). If it is does, an I/O status code of IE.IFC is returned.

4. The I/O function code is changed into a bit mask.

5. The I/O function code is checked against the legal function table in the DCB. If it is not legal, an I/O status code of IE.IFC is returned.

6. The device is checked to see if it is offline (bit US.OFL, [U.ST2] = 1). If it is, an I/O status code of IE.OFL is returned.

7. The I/O function code is checked against the control function table in the DCB. If it is, control is passed to control function service. This service copies the six I/O parameters to the I/O packet and continues at $DRQRQ.

NOTE

This is the point a UCP departs from the normal DCP packet processing. See section 5.3 for a discussion of how the remainder of the packet is processed.

8. The I/O function code is checked against the no-op function table in the DCB. If it is, an I/O status code of IS.SUC is returned.

9. The device is checked to see if it is not mountable ([U.CW1] > 0). If it is not, control is passed to 80$ and the following checks are made.

   1. The I/O function code is checked against the ACP function table in the DCB. If it is not an ACP function, control is passed to the transfer function service code.

   2. If function is an ACP function, the function code is changed to IO.WLB unless the I/O function code was IO.RVB which is converted to IO.RLB. Control is then passed to the transfer function service code.

NOTE

If you get here, the I/O function code must be either IO.RVB or IO.WVB. All others will be handled incorrectly.

At this point, the function is known that the function is for a device which could have an ACP servicing it. The first check performed is to see if an ACP is mounted or is mounted as foreign. If the device is not mounted or mounted as foreign (bits US.MNT!US.FOR [U.STS] = 0), control is passed to 60$. If no ACP is mounted (bit US.MNT, [U.STS] = 1), the following two steps are performed.

1. If the function is an ACP function, an I/O status return of IE.PRI is made.

2. Otherwise the control is passed to the transfer function service code.

If instead, a foreign ACP is mounted, control is passed to label 65$ and the foreign ACP processing is used.

NOTE

This is the point a FCP-style ACP departs from the normal DCP packet processing. See section 5.4 for further details.

If the device is mounted by a known ACP, a check is performed to see if the function is an ACP function. If it is not, a branch is made to label 75$ and the following checks made:

1. If the I/O function is load overlay (IO.LOV), processing continues as if the function is a transfer request.

2. If the issuing task is privileged (bit T3.PRV, [T.ST3] = 1), the function is allowed and treated as a transfer request.

3. Otherwise, an I/O status error of IE.PRI is returned.

At this point, DRQIO has finally recognized the I/O function as an ACP function to a mounted device. The special processing code used for DCP's is then invoked. This is performed by using the polish tables listed in the front of DRQIO. These tables are used to correctly dispatch ACP functions to the appropriate setup routines. The tables have two parts. Table FCDSP contains the starting address of the appropriate polish table for each function. This is followed by the polish tables. The registers and stack are setup as shown below when the dispatch into the polish code begins.

```
R0      = Address of UCB.
R1      = Address of second lun word.
R2      = Index into polish function dispatch table.
R3      = Address of Q.IOPL in QIO DPB.
R4      = Address of I.PRM  in I/O PACKET.
R5      = Address of next polish dispatch vector.
(SP)    = Address of second lun word.
(SP)+2  = Address of UCB.
(SP)+4  = I/O function code.
```

The processing performed by the individual polish routines is listed below. To see which routines are used by a particular DCP function, see the tables in DRQIO. The routines are listed in alphabetical order.

BDPKT - This routine completes the I/O packet for F11ACP and MTAACP. It expects the following I/O parameters in the QIO request:

```
Q.IOPM+0  = Address of file-ID
Q.IOPM+2  = Address of attribute list
Q.IOPM+4  = Extend control word 1
Q.IOPM+6  = Extend control word 2
Q.IOPM+10 = Access control word
Q.IOPM+12 = Address of filename block
```

The file ID block and filename block are relocated if present. The attribute descriptor block is processed by allocating a buffer from the system pool and copying the attributes from the user task to the system buffer. The attribute addresses are relocated. If any address do not check out, an I/O status code of IE.SPC is returned. Otherwise it dispatches to the next entry.

When BDPKT is finished, the parameters in the I/O packet are formatted as follows:

```
I.PRM+00 = File-ID relocation bias
I.PRM+02 = File-ID relocation displacement
I.PRM+04 = Address of system attribute buffer
I.PRM+06 = Extend control word 1
```

I.PRM+10 = Extend control word 2
I.PRM+12 = Access control word
I.PRM+14 = Filename block bias
I.PRM+16 = Filename block displacement

CKALN - This routine checks if a file is already accessed on the lun ([second lun word] # 0) and returns an I/O status code of IE.ALN if so. Otherwise it dispatches to the next entry.

CKCON - This routine copies the QIO parameters into the I/O packet. The routine requires the first two parameters to be a buffer address and size and relocates the buffer before storing. If the buffer is not specified or is illegal, an I/O status code of IE.BAD is returned. Otherwise it dispatches to the next entry. This routine is used for DECNET processing.

CKDIS - This routine copies the QIO parameters into the I/O packet and dispatches to the next entry. It is used for processing the DECNET disconnect request.

CKDMO - This routine checks if the volume is marked for dismount (bit US.MDM, [U.STS] = 1) and returns an I/O status code of IE.PRI if so. Otherwise it dispatches to the next entry.

CKMOU - This routine checks the mounted volume list to see if the volume can be access by the user. If the volume is public or the user is entered in the mounted volume list, the routine dispatches to the next entry. Otherwise an I/O status code of IE.PRI is returned.

CKNLN - This routine checks if a files is accessed on the lun ([second lun word] # 0) and returns an I/O status code of IE.NLN if not. Otherwise it dispatches to the next entry.

CKRAC - This routine checks if the user has read privileges for the I/O process. If processing for F11ACP, the window is then addressed to see if the read request can be directly mapped and queued to the driver.

CKWAC - This routine performs the same processing for write requests that CKRAC performs for reads.

CKXIT - This routine cleans the stack, increments the volume transaction count ([V.TRCT] <- [V.TRCT]+1), and exits queue I/O PACKET code (see section 7).

When all polish processing is finished, control is passed

to label FCKIT. The polish code flags whether the packet can be queued directly to the ACP or needs to be queued to the device driver for proper synchronization. The listing of DRQIO contains a complete description of the queueing and interlocking performed by the executive.

5.2.2 $DRQRQ Processing

The second entry in the DRQIO module is $DRQRQ. This is the code that queues I/O packets to drivers and calls the driver initiator. The following steps are performed.

1. Get the SCB address ([U.SCB]).
2. If the packet is not to be queued (bit UC.QUE, [U.CTL] = 1), goto step 4.
3. Queue the I/O packet to the SCB I/O queue.
4. If the driver is loadable, map KISAR5 into the driver.
5. Call the driver at the initiator entry point.
6. If the driver is loadable, restore KISAR5 to its original contents.
7. Exit from the DRQIO code.

NOTE

The entry $DRQRQ can be used to queue an I/O packet that has been constructed by other means. R1 should be the packet address and R5 the UCB address.

5.2.3 $GTPKT Processing

The executive normally queues I/O packets to DCP's from DRQIO. However, if the packet needs synchronization with the device driver, it is queued to the device instead. The routine $GTPKT is used by device drivers to get their next packet. When $GTPKT recognizes the I/O function as an ACP function, it queues the packet to the ACP and loops to attempt to dequeue another packet for the driver. The tests for this step occur at label

80$. Basically, $GTPKT assumes if the device is mountable (DU.CW1) < 0) and an DCP-style ACP, all I/O functions from 7 to 31(10) should be queued to the ACP.

If the ACP is marked as foreign, $GTPKT will check the DCB function dispatch masks to see if the function is for the ACP. If the DCB mask bit is set, the packet will be queued to the ACP, otherwise, it is returned to the driver.

5.3 UCP INTERFACE

UCP-style ACP's are used to avoid modifying the executive, particularly DRQIO, for support for the new ACP. Section 5.2.1 notes the point in the I/O processing where the UCP approach deviates from the traditional ACP packet processing logic.

The deviation occurs because the I/O function is marked for control processing. Control functions are processed beginning at the label FCCTL. First, the device is checked to see if it is a mountable magtape device and the I/O request is rejected if the task is not privileged. To avoid this processing, the US.LAB bit in the UCB status byte (U.STS) should be zero.

The next step is to copy the six QIO parameters to the I/O packet. This done by a call to FCXP1 at label FCXOP. No checks are made on the parameters. Once the parameters are copied, the packet is passed to $DRQRQ for queueing to the device. Here, the packet is passed directly to the driver initiator because the UC.QUE bit in the UCB control byte (U.CTL) is set. The driver than performs any processing necessary and queues the request on to the ACP.

Note that the UCP interface does not require any modifications to the executive. In addition, the interface uses standard features of the I/O mechanism. Therefore, UCP-style ACP's can be interfaced easily to RSX-11M and future operating system releases will not affect the user-written code.

5.4 FCP INTERFACE

The interface for FCP's is very similar to the UCP processing. The FCP logic is invoked if the device is marked mounted and foreign and the function masks indicate an ACP request. Section 5.2.1 notes the point in the I/O processing where the FCP approach deviates from the DCP logic.

The FCP logic first checks if an ACP has been specified by

checking that U.ACP in the UCB is nonzero. If the location is zero, a privilege violation is declared (IE.PRI). Next, if a VCB address is stored in U.VCB, the VCB transaction count is incremented.

The next step is to copy the six QIO parameters to the I/O packet. This done by a call to FCXP1 at label FCXQP. Note, at this point, the processing is identical to the UCP interface. However, if the UC.QUE bit is not set, the packet is queued directly to the ACP. Otherwise, the packet is passed to $DRQRQ for queueing to the device and the packet is passed directly to the driver initiator because the UC.QUE bit in the UCB control byte (U.CTL) is set. The driver than performs any processing necessary and queues the request on to the ACP.

While the FCP approach does not require any modifications to the executive, it does depend on continued support for foreign ACP processing. For this reason, the UCP approach is favored if all other considerations are equal. The main advantage of FCP's is the support provided by MOU and DMO.

5.5 ACP INTERFACE

No matter how the I/O packet is processed, the actual queueing to the ACP is done by the executive routine $EXRQP or $EXRQF. These routines place the packet in the receive queue of the desired task and schedule the task for execution. $EXRQP queues the packet by priority. The alternate entry, $EXRQF, queues using first-in, first-out logic.

Note, that the routines are used by the executive to interface to other tasks besides ACP's. For example, this is the mechanism used to pass command lines to MCR and abort messages to TKTN.

An ACP receives the packet by entering system state and removing the first entry from its receive queue. The logic necessary has already been discussed in section 3.5.1.

5.6 TASK TERMINATION

The only other module in the executive which is aware of ACP's besides DRQIO and IOSUB is DREIF. This module contains the code used to clean-up the system data bases when a task exits. A part of this clean-up is to check that all outstanding I/O is completed and all I/O processes have been terminated.

The code for these steps starts at label 5$ in DREIF. The first step is to test for outstanding I/O by testing T.IOC. If the value is nonzero, $IOKIL is called for each device assigned by the exitting task.

Once the I/O count has been reduced to zero, the logical unit table is scanned again for attached devices and I/O processes. Any attached devices result in an IO.DET packet being issued by the executive on behalf of the task. Similarly, an I/O processes still active will result in an IO.CLN packet being queued to the device. This is done by calling $DRQRQ. The packet should then be forwarded to the ACP and it should terminate the I/O process in a timely fashion. Otherwise, the task exit will not be completed.

The test for an I/O process is to check the second word of each logical unit table entry. If the lun is non-zero, an I/O process is assumed. However, if the lun is locked, the IO.CLN function will not be issued.

CHAPTER 6

ACP ENABLING/DISABLING

An ACP must be enabled before it can be used for user I/O. Similarly, the reverse capability is usually provided. The common names for these operations are mount and dismount. This chapter will discuss the requirements for ACP enabling and disabling and present five approaches that could be used by an ACP implementation.

6.1 ENABLING REQUIREMENTS

ACP enabling merely means satisfying certain conditions that allow user tasks to issue I/O requests to the ACP. For the Digital ACP's, enabling a device is performed by the MOU task. However, the MOU task is not the only method for enabling an ACP. Essentially, an ACP can be considered as enabled when it is servicing I/O requests from user task. The conditions that must be met to reach this state are as follows:

1. The device is marked as mounted. The US.MOU bit in the device status byte of the UCB indicates if an ACP is mounted or dismounted. If the bit is zero, the ACP is considered as being mounted.

2. The TCB address of the ACP is stored in U.ACP of the UCB. This value will be used when an I/O packet is queued to the ACP. The field will be assumed to be valid if US.MOU indicates the device is mounted (US.MOU=0).

3. The VCB is allocated from the system pool. The address of the VCB is stored in U.VCB of the UCB. The first word of the VCB will be used as a transaction counter and will be incremented when an I/O packet is queued to the ACP. The field will be assumed to be valid if US.MOU indicates the device is mounted (US.MOU=0).

4. The ACP has performed any operations necessary to

initialize itself and the device. It has verified user I/O request will be legal for the device and the current state of the RSX-11M system.

5. Finally, the ACP is dequeuing I/O packets from its receive queue when they are queued to it by the executive.

The logic necessary to reach the state listed above is very simple. The complexity of the Digital MOU task is because of the device initialization operations and command processing it performs and not the actual mounting of the ACP. For example, when mounting a Files-11 disk, MOU must read the Files-11 data structures on the disk and setup all the information needed by F11ACP for using the volume.

Before an ACP can be mounted, the current state of the system must be checked to see if conditions will allow the operation. Among the more common checks performed are the following:

1. The device characteristics word (U.CW1) in the UCB is checked to see if the device is mountable. If the device is not mountable (DV.MNT=0), the mount request is disallowed.

2. The device status byte (U.STS) in the UCB is checked to see if the device is already mounted. If the device is already marked as mounted (US.MNT=0), the mount request is disallowed.

3. The device status byte (U.STS) in the UCB is checked to see if the device is marked for dismount. If a dismount request is pending (US.MDM=1), the mount request is disallowed.

4. The device DCB is checked to see if the device driver is loaded. If the driver is not loaded (D.DSP=0), the mount request is disallowed.

5. The system task directory is searched for the ACP. If the TCB is not found, the mount request is disallowed.

6. The third status word of the ACP's TCB is checked to see if the task was built as an ACP. If is was not (T3.ACP=0), the mount request is disallowed.

7. The privileges of the user are checked to see if he is allowed to mount the device. Typically, if multiuser protection was selected and the user is nonprivileged, he must own the device or it must be marked as public. Otherwise, the user must be privileged in order to

mount the device.

Besides the common checks list above, usually device and specific ACP checks will be made. For example, if labeling information is available on the device, it will be checked against user supplied values to insure the right volume is being mounted.

The other operation performed when an ACP is mounted is to process command options specified. A typical ACP will allow options which to be specified when it is mounted. The options override any defaults established at assembly and build time.

In summary, mounting an ACP involves determining if the mount request can be allowed, processing any options, and setting up the ACP to process user requests. A separate task can be used for this process, the ACP can be self-mounting, or some combination of the two approaches can be used.

6.2 DISABLING REQUIREMENTS

Disabling an ACP reverses the process of mounting it. The same approach is followed. First, the dismount request is validated, any options are processed, and the ACP is dismounted. An ACP is considered dismounted when it is no longer marked as mounted for a device (US.MOU=1). When all devices serviced by the ACP are dismounted, the ACP task exits.

One point to note is that the actual dismounting of an ACP occurs asynchronously from the dismount request. Before the device can be marked as dismounted, all outstanding I/O requests queued to the ACP must be completed and all I/O process must be terminated. For this reason, a mark-for-dismount bit (US.MDM) is available in the U.STS byte of the UCB. When this bit is set, it signifies a dismount request has been made. The ACP will complete the process when all outstanding activity on the device is complete. Also, no new I/O processes will be allowed to be initiated.

The usual approach to dismount an ACP is for a separate task to process the dismount request from the terminal and queue an IO.DMO packet to the ACP. Once the packet is received by the ACP, it completes the process when all activity for the unit is finished. Digital's DMO task is an example of this process. The checks performed by DMO are typical for any dismounting process:

1. The device characteristics word (U.CW1) in the UCB is checked to see if the device is mountable. If the

device is not mountable (DV.MNT=0), the dismount request is disallowed.

2. The device status byte (U.STS) in the UCB is checked to see if the device is mounted. If the device is not mounted (US.MNT=1), the dismount request is disallowed.

3. The device status byte (U.STS) in the UCB is checked to see if a dismount request is already pending. If a request is (US.MDM=1), the dismount request is disallowed.

4. The privileges of the user are checked to see if he is allowed to dismount the device. Usually, the same checks performed for mount privileges are applied.

When all checks are finished, the dismount process sets the US.MDM bit and queues an IO.DMO packet to the ACP. This packet is processed by the ACP in the following manner.

1. The IO.DMO packet is returned with the appropriate code. Usually, this will be success

2. If the transaction count in the VCB is used in the usually fashion, it will indicate the number of outstanding I/O packets queued to the ACP and I/O processes. When the count reaches zero, the ACP can complete the dismount process. Otherwise, it continues to service I/O packets, reexaming the transaction count after every packet is finished.

3. The ACP is marked as not mounted (US.MOU=1).

4. The mark-for-dismount bit is turned off (US.MDM=0).

5. The VCB is returned to the system pool.

6. The internal count of mounted devices is decremented. When the count reaches zero, the ACP can finally exit.

6.3 STANDARD APPROACH

Digital's solution to enabling and disabling ACP's is the MOU and DMO MCR tasks. The next two sections outline how these tasks functions and their applicability to user-written ACP's. In general, the use of the MOU and DMO task for user-written ACP's is not recommended because of the support problem for future releases of the operating system.

6.3.1 MOU Task

The MOU task supports mounting Files-11 disks and ANSI magtapes. It also contains support for DECNET Phase I. However, MOU is no longer used by DECNET. In addition, the RSX-11M V3.2 version of MOU provides support for the /FOREIGN switch. This is intended to allow MOU to be used for FCP-style ACP's.

The flow through MOU is diagrammed in Figure 6-1. MOU consist of two common modules used for all ACP's (MOUROT, MOUPAR) and ACP specific parsing and mounting routines. The overall approach to adding a new ACP is to code a new parsing and mounting routine and modify the dispatch points in MOU to correctly call these routines for the new device.

The task entry to MOU is in the module MOUROT. On task entry, an immediate dispatch is made to the entry $MPREP in the module MOUPAR. The initial command processing performed by MOUPAR is as follows:

1. The parameter area used by MOU is zeroed. This area is in MOUROT and is referenced throughout the task.

2. The command line is input from the users terminal.

3. The device to be mounted is parsed from the command line. The remaining switches are not examined at this point.

4. Lun 2 of the MOU task is assigned to the specified device.

5. The UCB address is retrieved from the LUT table and stored at label $MDEV.

6. The device DCB is checked to see if the device driver is loaded.

7. If the device is not a magtape (DV.REC=0), the unit is attached.

Once the common command parsing is finished, the device specific parsers are called from MOUPAR. The UCB device characteristics are examined to determine which device parser should be used: MPAR11 (Files-11), MPARMT (ANSI magtapes), or MPARCM (communications device). If MOU is to be used for a user-written ACP, the code at this point will have to be modified to dispatch to the new ACP parsing module.

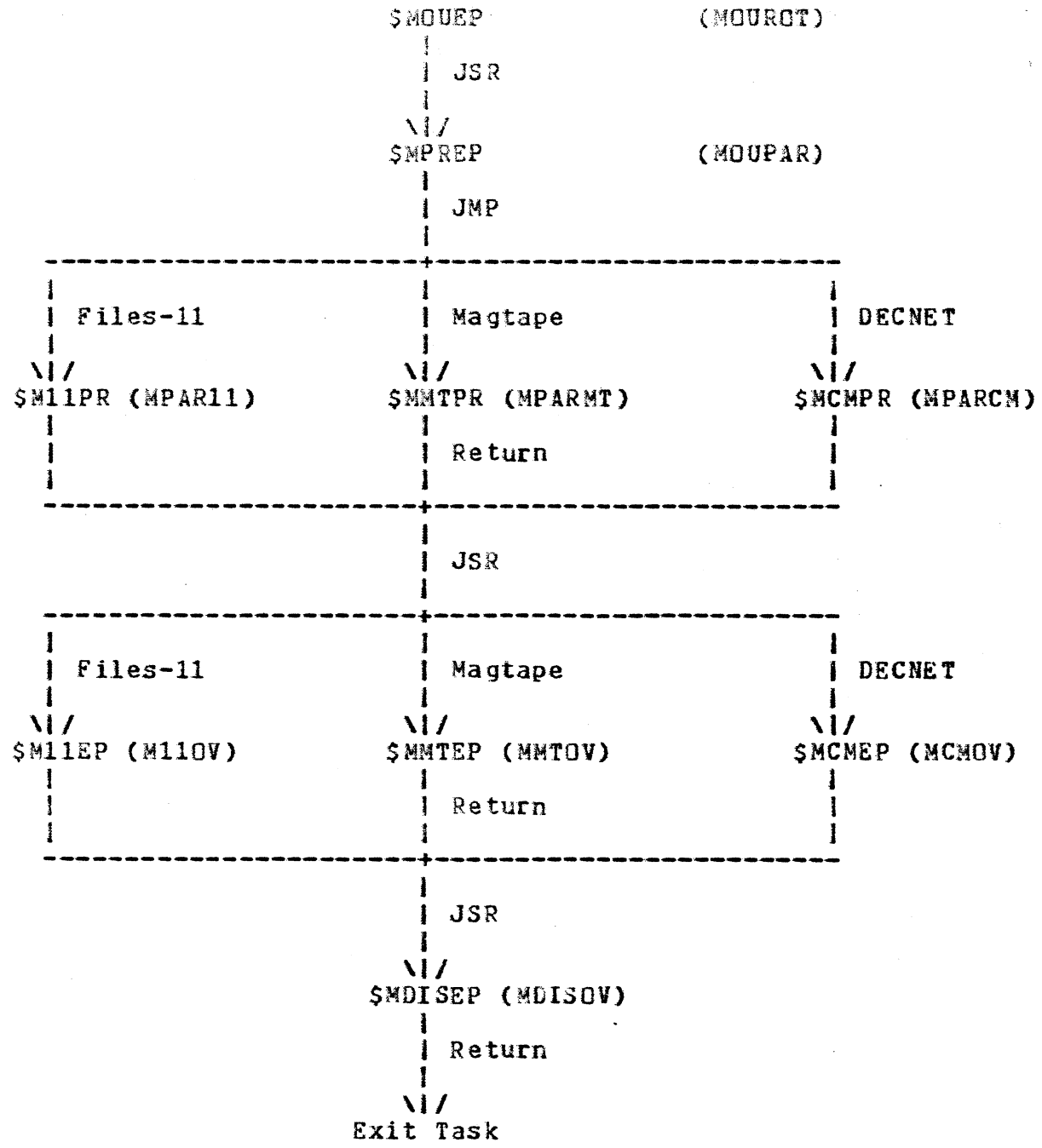


Figure 6-1
MOU TASK FLOW

The parsing modules perform basically the same operations. The following steps are performed:

1. The VCB is allocated from the system pool. The address of the buffer is stored in $MVCB and the size in $MVCBL. The initial VCB is zeroed.

2. The remainder of the command line is parsed for the ACP specific switches.

3. The user privileges are examined to see if the user has the necessary access to mount the device. The check is performed by the routine $PVTST in the module MOUPAR.

4. If needed, the mount list entry is allocated and filled in with the users terminal UCB. The mount list entries are used for disks and do not need to be used for user-written ACP's.

When all parsing is completed, control is returned to MOUROT. Here, some of the basic tests are performed and a dispatch is made to the device specific mount modules: M11OV (Files-11), MMTOV (ANSI magtapes), or MCMOV (communications devices). This is the second point modifications will need to be added to correctly dispatch for the new ACP.

The actual device mount takes place in the device specific mount modules. The disk and magtape routines perform device I/O to read the volume labeling information. The best routine to use as an example is MCMOV. This routine has no device specific I/O and therefore can be more easily read. The usual processing performed for a device is as follows:

1. The system task directory is searched for the ACP's TCB.

2. The TCB is examined to see if the task was built as an ACP.

3. The I/O packet for the IO.MOU request is allocated from system pool and initialized.

4. The device is detached.

5. The address of the VCB allocated previously is stored in the UCB.

6. The VCB transaction count is incremented.

7. The device is marked as mounted by clearing US.MNT.

8. The TCB address of the ACP is stored in the UCB.

9. The IO.MOU function is queued directly to the ACP. MOU is placed into I/O wait and it increments its I/O count.

When the ACP completes the IO.MOU function, it returns success/failure to MOU. The MOU task then cleans up, outputs any volume information, and exits.

6.3.2 MOU /FOREIGN

A new feature of MOU in RSX-11M V3.2 is the support of the /FOREIGN switch. This switch allows user-written ACP's to be enabled for devices. Unfortunately, the implementation is not easy to follow. However, the feature does provide a mechanism for mounting user-written ACP's without modifing MOU or writing any new code. If used, the MOU sources should be studied to see how the switch affects the normal processing performed by MOU.

6.3.3 DMO Task

The DMO task consists of one module, DISMNT. The procedure for dismounting a device is much simplier. In some cases, the DMO task may be used for user-written ACP's without modification.

The flow through DMO begins at the task entry, $DMOEP. The processing is as follows:

<To be written>

6.4 NON-STANDARD APPROACHES

As stated before, the use of the Digital MOU and DMO task is not recommended for user-written ACP's because of the support problem from release to release of the operating system. The following approaches can be used instead.

6.4.1 Self-Initialization

The easiest method of initializing an ACP is for it do the mounting itself when it is first scheduled. This approach is appropriate for ACP's which service only one device driver and involve no option processing. The ACP can allocate the VCB, setup the UCB status as needed, and perform any device initialization. Once mounted, the initialization code can be overwritten as data space. Another possibility is to place the initialization code in an overlay segment.

Usually, when this approach is used, the ACP is essential to the operation of the device and there is no reason to every dismount it. Therefore, the disabling step can be ignored.

6.4.2 Driver Initialization

A alternative to the ACP self-initialization approach would be for the device drivers serviced by the ACP to initiate the mount. If the UC.PWF bit is set in UCB, the power recovery entry in the driver is called when the system is booted or the driver is loaded. This entry can be used to start the mount process. Either the driver can set the necessary status directly or it could queue a IO.DMO packet to the ACP and let it finish the operation.

The same disadvantages apply to this approach as for the previous section. There is no convenient method for dismounting the ACP and inputting mount-time options. However, the approach does allow several different drivers to each mount the ACP separately for themselves.

6.4.3 Alternate Task

A final approach is to implement new system tasks to specifically mount and dismount the user-written ACP. The tasks can be modeled after the MOU and DMO tasks. The new tasks can perform any special processing required. The major advantage of this approach is the independence gained from future Digital modifications to MOU and DMO.

This approach was used by DECNET to mount NETACP and set up the other software. The sample ENA and DIS programs distributed with this manual can be used as a basis for the ACP specific enabling and disabling tasks.

CHAPTER 7

ACP IMPLEMENTATION

This chapter covers some of the basic considerations of ACP implementation. The material is not exhaustive and will not apply to every ACP. All ACP implementations will be different and no fixed set of guidelines can govern how an ACP will function internally.

7.1 TASK ATTRIBUTES

ACP's are constrained by the limits RSX-11M places on privileged tasks. RSX-11M associates a set of attributes to each task in the system. The proper definition for each attribute will have to be decided when the ACP is implemented. Some of the attributes involve trivial decisions: the partition, taskname, etc. Others are more important and are discussed in the following two sections.

7.1.1 Task Mapping

The major task attribute to be considered is the ACP task mapping. Figure 7-1 diagrams the mapping for a privileged task. The first four APR's (APR0-APR3) are always mapped to the executive. APR4 is mapped to the executive if 20K executive support was selected when the system was generated. If the ACP is to be portable between systems, APR4 should not be used.

Normally, the ACP is then left with APR5 and APR6 to map its task code and APR7 to map the I/O page. If 8 KW is not enough memory, the ACP can extend into the I/O page. However, when a switch is made to system state, the executive will unmap the ACP's APR7 and map the I/O page. If the top of the ACP is used for data and not code, this restriction will usually not be a problem.

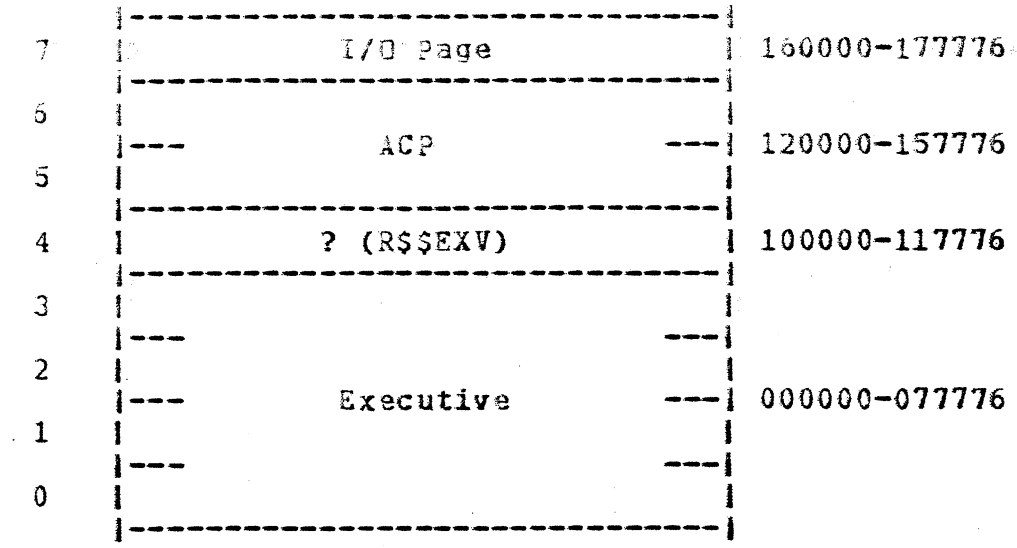


Figure 7-1
TASK MAPPING

7.1.2 Task Build Requirements

When linking the ACP, several TKB options apply. First, the task should be declared as an ACP with the /AC switch. This TKB switch causes T3.ACP to be set in the TCB when the ACP is installed. When T3.ACP is set, the executive prevents sends to the ACP task. This is desirable because the ACP is receiving the I/O packets on its receive queue and would have trouble if some other task were allowed to send it messages. The /AC switch is also useful for identifying the task as an ACP when the device is mounted.

The /AC switch also causes the task to be marked as privileged. The other TKB switches used can be set according the ACP implementation.

There are no special TKB options that apply strictly to ACP's. Therefore, the options specified will be a function of the ACP implementation and the environment the ACP will be executed in when it is mounted. One consideration is whether to use a separate partition for the ACP other than the one used for user tasks. A lock-out condition can occur if a user task issues an I/O request to an ACP which is swapped out because of the user task.

7.2 DESIGN CONSIDERATIONS

This section comments on some general design considerations for ACP's The list is not exhaustive and has been developed from past experience in writing ACP's.

7.2.1 ACP I/O

Besides receiving I/O requests from user tasks, ACP's usually issue I/O request. These request can be to device drivers or to other ACP's. Normally, the ACP would use the QIO directive for any I/O it needs to perform. However, when performance is an issue, the ACP can bypass the normal executive directives and interface directly to the drivers. In high-performance environments, the ACP and the driver may actually use cooperating code and bypass RSX-11M altogether. How an ACP performs its I/O using non-standard logic is strictly up to the implementation.

7.2.2 Transportability

Often, an ACP is required to run on a variety of machines. The ACP coding can be greatly simplified if the ACP is restricted to one type of machine. However, by using conditional assembly symbols and interfacing correctly to the executive, it is possible to write an ACP that will properly function on all PDP-11's.

The biggest difference between machines is the presence or absence of memory management. Typically, any code used to move information between the ACP and a user task must be assembled with conditionals for memory management. On a mapped system, the APR's must be manipulated. On an unmapped system, the entire system can be addressed directly. The problem can be avoided by using the executive routines to manage data transfers.

The other large difference between machines is arithmetic instructions. The older PDP-11's do not have the MUL/DIV instructions. If transportability is desired to such machines, the executives $DIV/$MUL routines should be used.

7.2.3 Pool Utilization

Most RSX-11M systems suffer from a lack of pool. ACP's are a potential large user of pool. Limits on the amount of pool an ACP can use and use of internal versus system pools should be considered in the design of an ACP.

The typical approach is to create an internal ACP pool that is structured the same as the RSX-11M system pool. The size of the pool is a mount or link-time option. Whenever the ACP needs to allocate an internal structure, an attempt is first made to allocate from the ACP pool. If this fails, the allocation is attempted from the system pool. When the structure is deallocated, the address indicates to which pool the memory is returned.

7.2.4 ACP Variables

The design of an ACP should consider the amount of flexibility that will be built into the code. There are four types of ACP variables: assembly-time statements, TKB switches, mount-time options, and run-time I/O functions. For example, an ACP may use a separate LUN for each I/O process it supports. A TKB option would specify the maximum number of luns the ACP can use. A mount-time option would then specify the number of actual luns to be used (up to the taskbuilder limit).

A good practice is to make the defaults for each option settable by the taskbuilder GBLPAT and GBLSYM commands. This allows each installation of the ACP to specifically configure its default options.

7.2.5 Performance Statistics

If the ACP supports options, it should then accumulate statistics on its performance. This would allow adjustment of the options to maximize performance in a given environment. Without statistics, the users of the ACP will be faced with a situation similar to F11ACP. There are many switches for mounting a disk. However, except for eyeballing the system performance, there is no way to tell how a particular option is performing.

If performance measurements are used by the ACP, it the performance code should be conditionally assembled. This would allow a smaller version of the ACP to be assembled after

sufficient experience with the various options is obtained. In addition, some method of examining the statistics is needed. This can be as simple as using the MCR OPEN command to examine known locations. In other cases, the ACP may support I/O functions to return is performance data. One final method is to keep the performance data in the system pool and implement a separate task to access the data and print reports.

7.2.6 Serial Versus Parallel Processing

The final consideration in the design of the ACP is whether it will process user I/O requests in serial or parallel. A serial ACP will only process one I/O request at a time. This greatly simplifies the implementation because only one set of internal data bases needs to be kept and the flow through the ACP is completely synchronous. The ACP only attempts to dequeue an I/O request from its receive queue when it has finished the previous request. F11ACP is implemented as a serial ACP.

The serial approach is appropriate for ACP's which service fast devices such as disks. There will be little delay for ACP I/O The disadvantage of the serial approach is that the ACP is a bottleneck in the I/O mechanism.

An ACP which allows parallel processing of user I/O requests will typically support one I/O request for each I/O process. Such an approach is particularly applicable for ACP's which service communication devices. NETACP is an example of a parallel implementation. There is a significant delay between the time the user DECNET QIO is issued and the necessary NSP protocol has been exchanged to fulfill the request.

However, this approach is much more complex to implement. Many of the data bases and variables used by the ACP will have to be duplicated for each I/O process. In addition, the logic of the ACP must be able to save the context of one operation when it reaches a checkpoint, process another I/O function for another process, and eventually resume the original process. This requires the ACP code to be reentrant and some sort of internal scheduling algorithm to be implemented.

7.3 DEBUGGING NOTES

Because ACP's usually involve complex logic to implement their protocols, much of the time spent in development will be devoted to debugging. In addition, because bugs in the ACP are likely to corrupt the operating system, the stability of the

machine for other users will be reduced.

One advantage of ACP's is that ODT can be used on the task portions of the implementation. However, ODT cannot be used to breakpoint within portions of the ACP code that run at system state. In such code, XDT would have to be used. Similarly, XDT would have to be used for debugging any driver or executive code involved with the ACP.

Another problem in developing ACP's occurs if the ACP terminates abnormally while still mounted. The data structures still mark the ACP as being mounted and the typical method to clean them up is to reboot the system. One alternative to this approach would be to write a special privilege task that destroys the ACP data structures to the point where you can remount the ACP. While some pool may be lost, the basic integrity of the system can be preserved.

CHAPTER 8

EXECUTIVE SERVICES

There are many executive services an ACP can use. This chapter briefly describes some of the services available from the executive. There are many other facilities which are not documented in the chapter. The actual calling sequences for the routines can be found in the executive listings. The routines are also documented in the RSX-11M System Logic Manual.

8.1 SYSTEM STATE

Before an ACP can use an executive service or manipulate system data structures, it must make sure its access to the executive is correctly sychronized with other system activity. This is done by switching to system state. Once a switch to system state is made, the ACP is running as a part of the executive and must follow all rules applicable to the kernel state. For example, directives cannot be issued from system state and any unexpected trap will cause the system to crash, rather than aborting the ACP.

The switch to system state is accomplished by issuing an EMT 376. This instruction is followed by the address to return to when a return is made to task state. The following summarizes the switching to system state process:

```
Call By:     CALL    $SWSTK,addr     (RSXMC.MAC)

Generates:   EMT     376
             .WORD   addr

Effect:      Returns to next instruction
             Switches to kernel mode
             Maps user APR's 4(?),5,6 to kernel space
             Kernel APR's 0,1,2,3,4(?) mapped to executive
             Kernel APR 7 mapped to I/O page
             Saves all registers
```

Exit By: RETURN instruction. Exit is to "addr"
Pre-kernel registers restored

Values can be returned in R0-R3 by storing value in 2(SP) to 10(SP) before RETURN

8.2 I/O TERMINATION

One service used by ACP's is I/O packet termination. The I/O termination code is located in the executive module IOSUB. Three entries are provided by the executive: $IODON, $IOALT, and $IOFIN. The first two entries are used by device drivers. In addition to the I/O packet termination, these routines free the device for the next I/O packet. The $IOFIN routine is the entry usually used by ACP's as it does not affect the driver data bases.

8.3 ADDRESS CHECKING

Another service provided by IOSUB is address checking. Whenever an address is passed from a user task to an ACP, it must be checked by the executive to make sure the address is within the boundaries of the user task. This check must be done in the context of the user task. Therefore, address checking must be performed before the I/O packet is queued to the ACP.

The executive provides five entries for address checking. The first three, $ACHKP, $ACHKW, and $ACHK2, are only used for directive processing and do not return errors to the caller. Instead, they return an directive status error if the address check failed.

The two entries used for I/O-related address checking are $ACHKB and $ACHCK. The first entry checks for proper byte alignment. The $ACHCK requires word alignment. The routines return with the carry bit clear/set as a success/failure indicator.

8.4 ADDRESS RELOCATION

In order for an ACP to address a user-task address on a mapped system, the user-task virtual address must be transformed into a relocation bias and offset. As in address checking, this process must be performed in the context of the user task.

Two entries in IOSUB provide support for address relocation. The first routine, $RELOC, transforms the user virtual address into a relocation bias to be loaded into APR6 and the APR6 displacement. These values can then be passed to the ACP, which can then use them to map the user task addresses.

The second routine is named $RELOM. The routine in addition to performing the address relocation, loads kernel APR6 with the address bias. This routine is used by the executive and/or drivers when processing an ACP I/O request and reading/writing user-task memory.

8.5 BUFFER ALLOCATION

The executive module CORAL contains support for buffer pool allocation. The routines are normally used to allocate and deallocate space in the system pool, however, they can be used to manage any buffer pool which follows the RSX-11M system pool mechanism. The System Logic Manuals provide details on this mechanism.

Two entries in CORAL can be used for buffer allocation. The $ALOCB entry is the general system pool allocation routine. The $ALOC1 is an alternate entry. Here, the buffer listhead address is passed to the routine instead of defaulting to the system pool as in $ALOCB.

NOTE

The $ALCLK (allocate clock queue entry) and $ALPKT (allocate I/O packet) should not be used by ACP's. These routines return a directive status error if the allocation fails instead of an error return to the caller.

8.6 BUFFER DEALLOCATION

CORAL also provides two buffer deallocation entries. $DEACB is the general system pool deallocation routine. $DEAC1 is the alternate entry for general deallocation of buffers for non-system pools.

8.7 QUEUE MANIPULATION

The module QUEUE provides a variety of entries to manipulate the various RSX-11M queues. The most used entries are $QINSF, $QINSP, and $QRMVF. The first two routines respectively insert by first-in, first-out or by queue entry priority. When priority insertion is used, byte 2 of the queue entry is assumed to contain the priority.

$QRMVF removes the first entry in the queue. This routine is used by all ACP's to remove I/O packets from the receive queue. It may also be used for internal ACP queues or other RSX-11M data structures.

8.8 DATA TRANSFER

The module BFCTL contains an extremely useful routine for passing data between ACP's and the user task. The routine $BLXIO moves a block of data between any two points in a mapped system. It is called with the number of bytes to move, the source APR5 bias and offset, and the destination APR6 bias and offset.

NOTE

The other routines in this module are for device driver usage. They expect the UCB to be set up and should not be used by ACP's.

8.9 TASK SCHEDULING

The last executive services commonly used by ACP's are found in the module REQSB. The routines in this module perform task scheduling. The most useful entries are $EXRQF, $EXRQP, and $EXRQN. These routines pass an I/O packet to the receive queue of the requested task and schedule the task for execution. If the task is not active, it is run for the first time. Otherwise, the task is unstopped and potentially becomes available for execution.

Another useful routine is $STPCT. This routine stops the current task and is used by ACP's to stop themselves when they have no work to do. Note, when called at system state to stop

the ACP, the routine merely marks the ACP TCB as being stopped. The actual rescheduling does not occur until the ACP exits system state.

APPENDIX A

READINGS/REFERENCES

This appendix lists documents that anyone writing an ACP should be familiar with. It is impossible to present an exhaustive list because "everything" one needs to know to write an ACP is an intimate knowledge of the system.

A.1 DIGITAL MANUALS

The following is a list of the most relevant RSX-11M manuals. No single manual deals with the subject of ACP's. In fact, the manuals contain little information directly applicable to ACP's. The information concerning ACP's must be extrapolated from descriptions of other subjects.

A.1.1 RSX-11M Executive Reference Manual

This manual describes the available executive services and system directives. It also presents some material on the structure and design of the operating system.

A.1.2 RSX-11M Crash Dump Analyzer Reference Manual

This manual describes the use of the Crash Dump Analyzer (CDA) utility. While the ACP implementor will certainly become very familiar with crashes, the most valuable section is Appendix B which list the system data structures and symbolic offsets.

A.1.3 IAS/RSX-11 I/O Operations Reference Manual

This manual describes the use of the File Control Services (FCS), including diagrams of the FCS data structures. The manual is only important to someone interfacing an ACP to FCS.

A.1.4 IAS/RSX-11 RMS-11 Programmer's Reference Manual

This manual describes the use of the Record Management Services (RMS). However, it contains no information on RMS internals or data structures. The manual is only important to someone interfacing an ACP to RMS.

A.1.5 RSX-11M I/O Drivers Reference Manual

This manual describes each RSX-11M I/O driver and the format of the I/O requests serviced by each driver. The appendices contain useful summaries of all Digital defined I/O requests and error codes.

A.1.6 RSX-11M Guide to Writing an I/O Driver

This manual contains the information necessary to write an I/O driver. It also presents the best description of the overall I/O process. It contains descriptions of the I/O data bases and basic I/O flow. This manual is a good starting point for anyone considering writing an ACP.

A.1.7 RSX-11M System Logic Manuals

This two-volume set is not included in the RSX-11M documentation kit. However, it should be considered a "must" purchase for implementing an ACP. The set contains information on the structure and design of RSX-11M. It also includes descriptions of all executive modules and data bases.

A.2 SOURCES

The final authority on RSX-11M is the source listings. The RSX-11M binary distribution kit includes the sources for the executive, device drivers, and MCR. The utility source kit supplies sources for all utilities, FCS, F11ACP, and the macro libraries.

The sources are very well written. This section mentions the most useful modules.

A.2.1 Executive Sources

A complete listing of the executive should be available before writing an ACP. In particular, the following modules should be read:

1. DRQIO - This module contains all the QIO processing code, including the executive ACP processors.

2. IOSUB - This module contains all routines related to I/O.

3. DREIF - This module contains the code related to task exit. Of interest to the ACP writer is the code relating to I/O cleanup.

A.2.2 MOU Sources

The sources for the MOU task are found on the RSX-11M MCR source disk in UIC [12,10]. The following modules comprise this task.

1. MOUROT - This module is the root for the task. It contains the dispatching code for the mount process and the error handling code.

2. MOUPAR - This module is the general parser for the task. It determines the device name specified in the command line and dispatches to the appropriate device-specific parser.

3. MPAR11 - This module is the command line parser for Files-11.

4. MPARMT - This module is the command line parser for ANSI magtapes.

5. MPARCM - This module is the command line parser for DECNET Phase I.

6. M11OV - This module is the mount processor for Files-11.

7. MMTOV - This module is the mount processor for ANSI magtapes.

8. MCMOV - This module is the command line parser for DECNET Phase I.

9. MDISOV - This module displays the volume attributes.

A.2.3 DMO Sources

The DMO task source is the module DISMNT.MAC. This file is found on the MCR source volume in UIC [12,10].

A.2.4 F11ACP Sources

F11ACP is the one Digital ACP for which sources are available at most sites. The ACP sources are found in UIC [13,10] on the MCR/FCP source disk of the RSX-11M V3.1 kits. The source code was omitted from the RSX-11M V3.2 distribution. Perhaps the most useful file is DISPAT.MAC. This module contains the I/O packet dequeueing and dispatching code.

A.2.5 DECNET Sources

The DECNET sources are another example of an ACP. However, the sources are only available by purchasing the DECNET source kit.

A.2.6 FCS-11 Sources

The FCS sources are found on the RSX-11M utility source kit in UIC [50,10]. If FCS must be modified to use your ACP, this kit should be purchased. FCS is very old and not written to the standards used for other code. The reader will find that the code jumps around considerably and is difficult to follow. One very useful way of listing FCS is to make a concantenated listing. The sources contain directions for doing this.

A.2.7 Other Sources

The RSX-11M source distribution kit contains many other useful modules. Among the most significant are the following:

1. [32,10] - PIP sources.
2. [55,10] - Command line routines (GCML, CSI).
3. [60,10] - PIP utility (PIPUTL) sources.
4. [66,10] - Macro definition files.

APPENDIX B

DATA STRUCTURE DETAILS

This section details the features of data structures of interest to the ACP implementor. Only the fields of interest to ACP's are discussed in this document. Further information can be obtained from the references listed. A diagram is presented for each structure, along with information on which macros and files define the symbolic offsets, values, and bit masks.

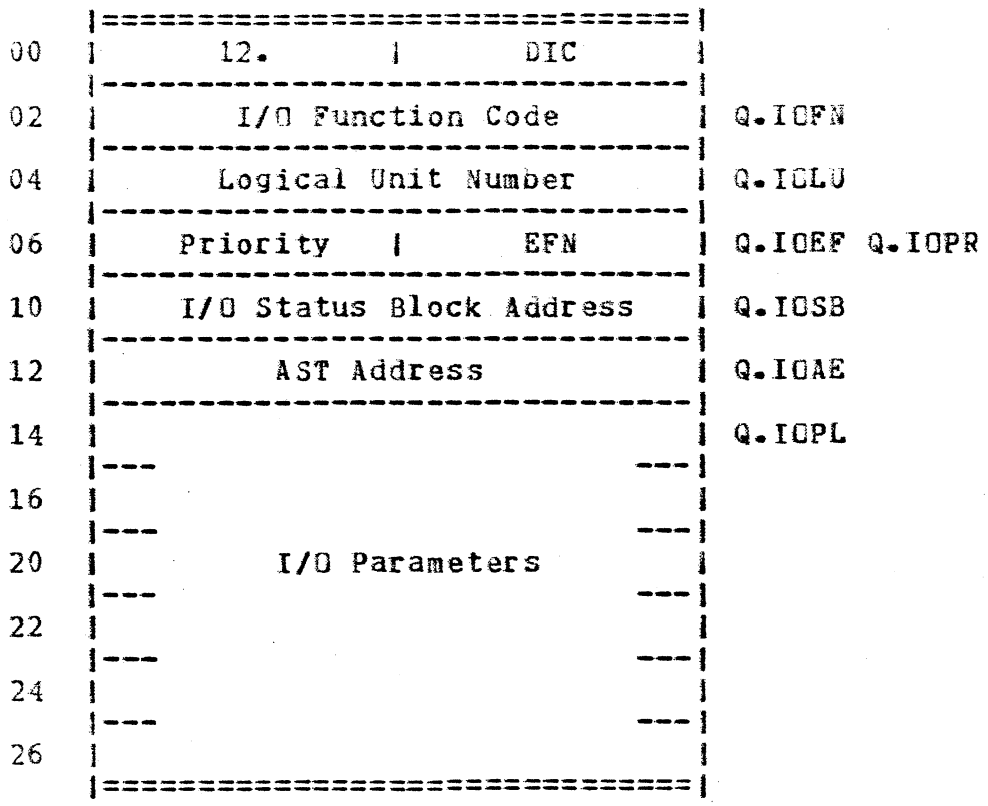
NOTE

The macro definition files refered to are located on the RSX-11M Utility Source kit in UIC [66,10]. A listing of these files is a very valuable reference tool.

The global definition file is the file used to create an object module containing the symbols defined as globals.

B.1 QIO DATA STRUCTURES

The QIO data structures structures are used to hold the I/O parameters and route the request to the correct device. The structures in this category are the QIO directive parameter block, the I/O packet, and the window block portion of the task header.



Symbol Definition Macro: QDPB$
Macro Source Filename: RSMDIR.MAC
Macro Library Filename: RSXMAC.SML
Global Definition File: DIRDEF.MAC
Object Library Filename: SYSLIB.OLB

Figure B-1
QIO DPB

B.1.1 QIO Directive Parameter Block

The user task constructs a QIO directive parameter block (DPB) and issues a QIO directive whenever I/O is to be requested from a device. This includes I/O requests to ACP's. The QIO DPB is normally constructed by the QIO$ and QIOW$ macros. The format of the QIO DPB is shown in Figure B-1.

The following fields in the QIO DPB must be considered when writing an ACP:

Q.IOFN The I/O function is split into two bytes. The high byte is the function code and must be from 0-31(10). This is the value which is used by the executive to determine the action to be taken for the function. The low byte is the function modifier. It is not processed by the executive but is passed to the driver/ACP.

The choice of I/O functions will depend on whether a DCP or UCP is being written. DCP typically use an existing set of function codes (such as Files-11). Note, DCP's require all ACP function codes to be greater than or equal to seven. In fact, $GTPKT assumes all I/O request with codes from 7-31(10) issued to a device with an ACP enabled are ACP functions and dispatches them to the ACP.

When writing a UCP, it is wise to choose a function code that is not used by other drivers and ACP's. If a I/O request is issued to the wrong device, this prevents unexpected operations from occuring. The modifier byte can be used to separate a class of I/O functions into separate requests. Currently, all function codes but 31(10) have been used by Digital. However, codes above 24(10) are used only for specialized functions for communications devices and real-time interfaces.

Q.IOPL How the I/O parameter words will be used by an ACP is ACP dependent. The only constraint on this set of parameters is that for transfer functions the first parameter must be a buffer address and the Q.IOPL+2 must be the buffer size in bytes.

References:

RSX-11M Executive Reference Manual, pages 6-90 to 6-94. This section documents the format of the DPB and how to setup the QIO macros.

RSX-11M Guide to Writing a Device Driver, pages 4-6, 4-7. This section documents the format of a DPB.

RSX-11M I/O Drivers Reference Manual. This manual documents all current, device driver QIO functions and their associated parameter lists. Of particular interest are Appendices A and B which summaries the QIO's, their parameters, and the various error codes.

IAS/RSX-11 I/O Operations Reference Manual, Appendix I. This appendix contains a listing of QIOSYM.MAC. This file defines all I/O function and error codes.

RSX-11M System Logic Manual, Volume 2, pages C-41 to C-45. This section documents the various macros used to create QIO DPB's and issue the QIO request.

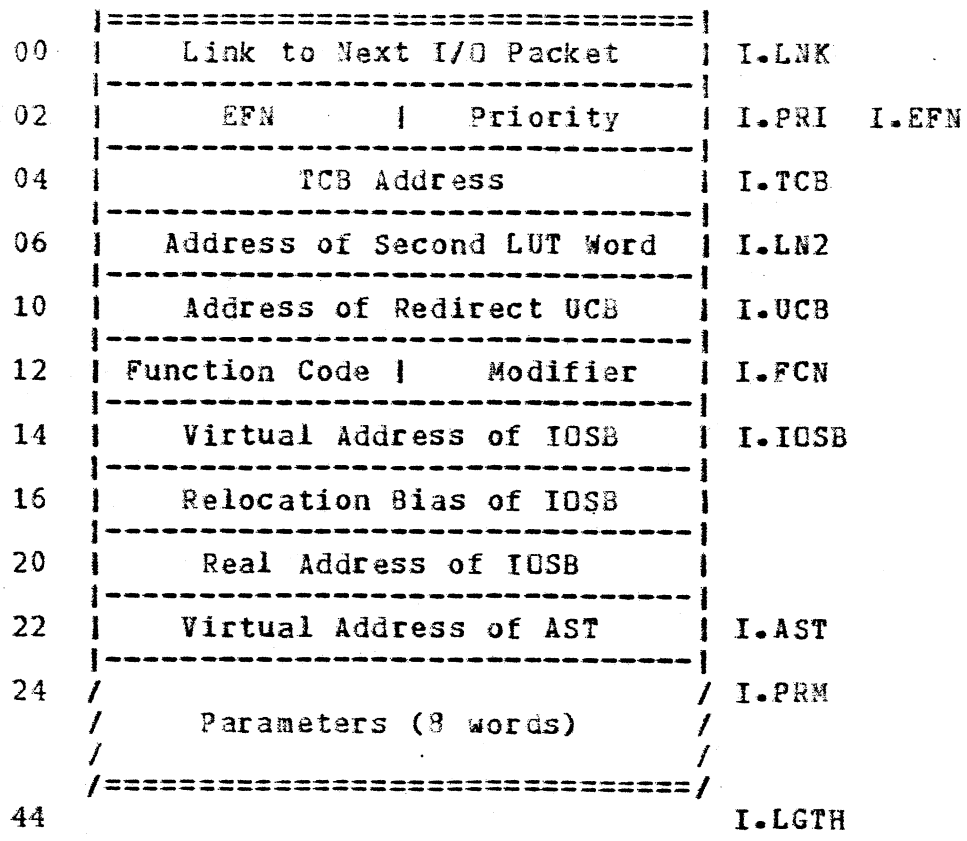
B.1.2 I/O Packet

The I/O packet is constructed by the executive from the information in the QIO directive parameter block. The I/O packet is then dispatched to the proper service routine. If an ACP is enabled for a device, the service routine is the ACP. The format of the I/O packet is shown in Figure B-2.

ACP's are concerned with the following fields in an I/O packet. In general, the I/O packet should be considered to be read-only. The only exceptions are the function code and parameter fields. These can be changed when the ACP is reformating the request and forwarding it to a device driver.

I.TCB This field contains the TCB address of the requesting task. The ACP may store this address away for further reference.

I.LN2 This is a very important field for ACP's because it contains the address of the address of the window block (see next section). Note, that this address cannot be saved but should be obtained from each separate I/O packet. Task headers may be destroyed and recreated during task execution. In general, only the TCB address can be saved by an ACP for a task with an active connection.



Symbol Definition Macro: PKTDF$
Macro Source Filename: PKTDF .MAC
Macro Library Filename: EXEMC .MLB
Global Definition File: EXEDF .MAC
Object Library Filename: EXELIB.OLB

Figure B-2
I/O PACKET

I.LN2 can also be used to obtained the original UCB address for the lun. The previous location in the logical unit table is the original lun. The redirect chain is followed for every QIO request and the final UCB address is stored in I.UCB.

I.FCN This is the I/O function issued by the user. It is copied from Q.IOFN of the QIO DPB.

I.PRM These are the parameters for the I/O request. Note that eight words are available as opposed to the original six in the QIO DPB. Except for these fields, common code is used to fill in the I/O packet. The parameter fields are established from the QIO parameters according to the type of request.

For control functions, the six QIO parameters are copied unmodified to the first six locations. For transfer functions, the buffer address doubleword is placed in I.PRM+0 and I.PRM+2. The remaining five parameters are copied into I.PRM+4, I.PRM+6, etc.

For ACP's, the QIO parameters are specially processed and the results stored in the I/O packet. All addresses passed as parameters must be address checked and relocated before the packet is queued to the ACP. In the case of a DCP, this processing takes place in DRQIO. For UCP's, the packet is processed as a control function by the executive and passed to the driver without queueing. The driver then performs the necessary checking, sets the I/O packet parameters and send the packet to the ACP.

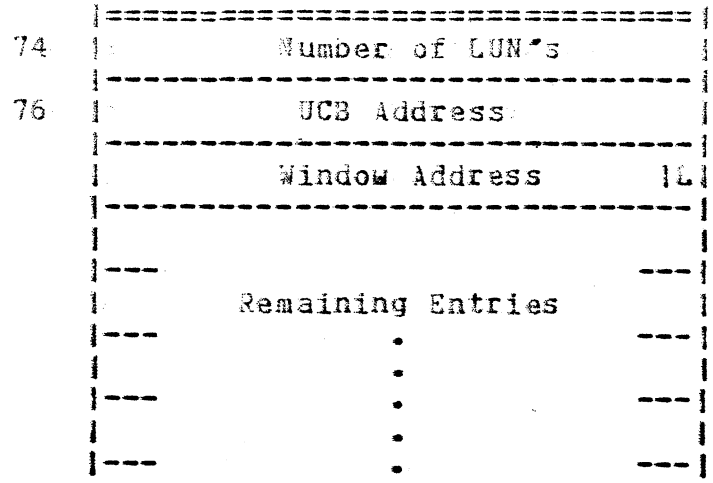
References:

RSX-11M Guide to Writing a Device Driver, pages 4-2 to 4-5. This section documents the various fields in the I/O packet.

RSX-11M Guide to Writing a Device Driver, page C-16. This section documents the macro PKTDF$. This macro is used to declare the I/O packet symbolic offsets.

RSX-11M Crash Dump Analyzer Manual, pages B-39 to B-41. This section documents the macro PKTDF$. This macro is used to declare the I/O packet symbolic offsets.

RSX-11M System Logic Manual, Volume 1, pages 8-30, 8-31. This section documents the macro PKTDF$. This macro is used to declare the I/O packet symbolic offsets.



Symbol Definition Macro: HDRDF$
Macro Source Filename: HDRDF .MAC
Macro Library Filename: EXEMC .MLB
Global Definition File: EXEDF .MAC
Object Library Filename: EXELIB.OLB

Figure B-3
LUT TABLE

B.1.3 Logical Unit Table

The logical unit table (LUT) is used to determine to which device the I/O request is directed. It is also used by ACP's to establish a logical connection between a specific LUN in a specific task and a I/O process (such as an open file or a NSP logical connection). A diagram of the LUT table is shown in Figure B-3.

The LUT table is located in the task header, beginning at offset H.LUN. The previous location, H.NLUN, contains the number of entries in the LUT table. The LUT table consist of a series of two word entries, one entry for every possible LUN number the task may use. The first word of the entry is the UCB of the assigned device. Note, this is not the redirected UCB.

The second word is used for the window block address. This is the structure which is used to link a LUN to an ACP process. This word also serves two other purposes. The low bit is refered to as the lock bit. If this bit is set, another I/O request cannot be issued on the LUN. The executive backs up the issueing task PC and places it in a wait-for-significant event state if such a request is issued. This is very useful when an ACP must enter a state where another I/O packet cannot be processed until the current request completes.

NOTE

Only QIO's are effected by the lock bit. Any packet in the ACP queue for the locked lun can be dequeued by the ACP. In order to effectively lock a lun, the bit must be set during the I/O packet processing prior to queueing to the ACP.

The final use of the second word is to notify ACP's about abnormal task termination when it has an active process for a lun. If the second word of the LUT entry is non-zero (exclusive of the lock bit), a IO.CLN (code 3400) will be issued to the ACP by the executive task termination routine.

B.2 DEVICE DATA STRUCTURES

The device data structures structures are used to describe a generic device type, each unit, and each controller. The Device Control Block (DCB) names the device and contains the I/O dispatch tables. One Unit Control Block (UCB) exists for each device unit and forms the data base necessary for handling one I/O request. The Status Control Block (SCB) is used to coordinate activity among device controllers.

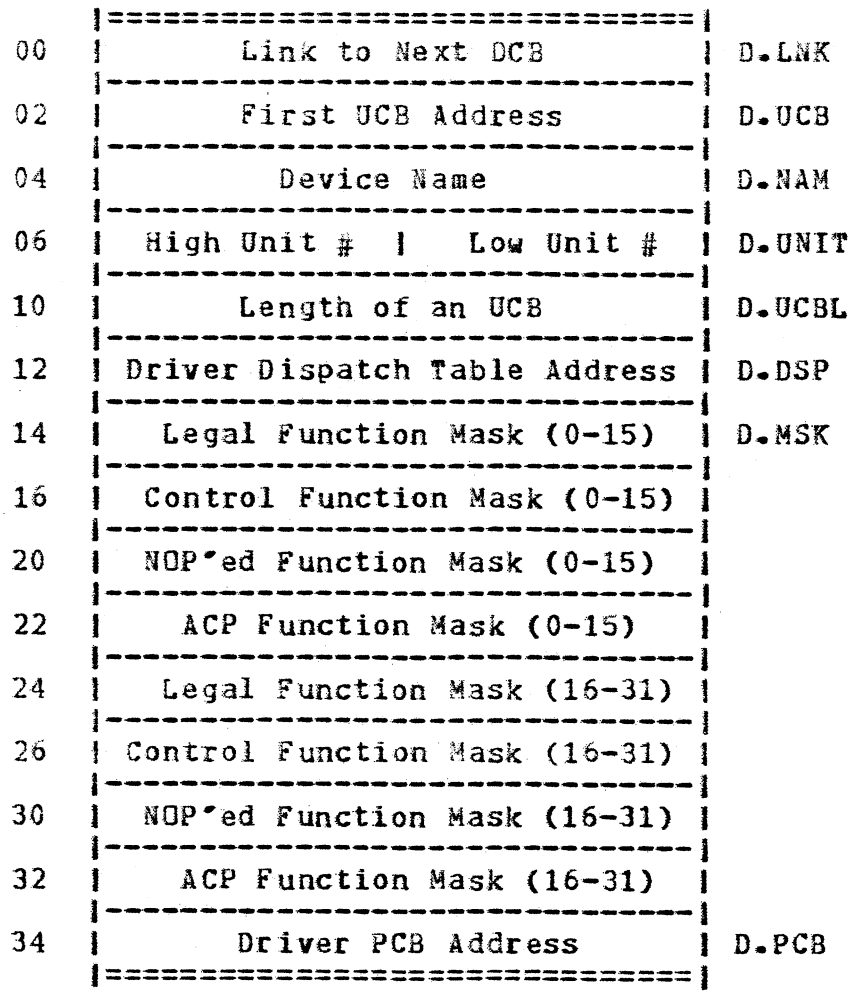
B.2.1 Device Control Block

One device control block (DCB) is created for each generic device type. The DCB supplies the device name, points to the device unit structures (UCB's), and controls how I/O functions will be processed for the device. It is the function mask fields (D.MSK) that are important to the ACP writer.

In general, DCP's will set the appropriate ACP function bit masks. When DRQIO matches the function against this field, the special processing code for ACP's is then used. UCP's, on the otherhand, will mark its functions as control functions. Section 4.2.1 discusses how the DCB fields should be setup in more detail.

References:

RSX-11M Guide to Writing a Device Driver, pages 4-8 to 4-15. This is a location by location description of the DCB. It is oriented to device drivers.



Symbol Definition Macro: DCBDF$
Macro Source Filename: DCBDF .MAC
Macro Library Filename: EXEMC .MLB
Global Definition File: EXEDF .MAC
Object Library Filename: EXELIB.OLB

Figure B-4
DCB

RSX-11M Guide to Writing a Device Driver, page C-5. This section documents the macro DCBDF$. This macro is used to declare the DCB symbolic offsets.

RSX-11M Crash Dump Analyzer Manual, pages B-8, B-9. This section documents the macro DCBDF$. This macro is used to declare the DCB symbolic offsets.

RSX-11M System Logic Manual, Volume 1, pages 8-34 to 8-36. This section documents the macro DCBDF$. This macro is used to declare the DCB symbolic offsets.

B.2.2 Unit Control Block

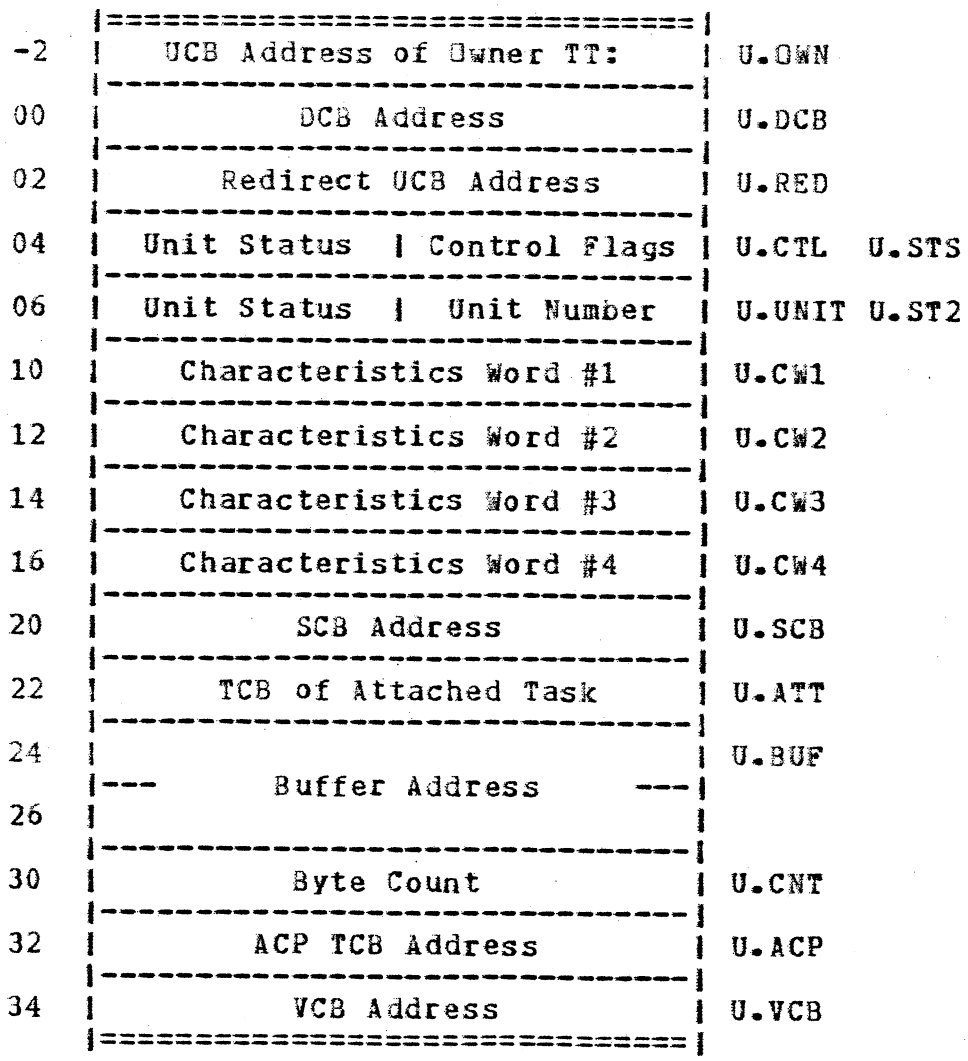
The Unit Control Block (UCB) is the key device structure and is very important to ACP's. There is one UCB for each separate device. The UCB provides the characteristics of the device, pointers to the other device structures, and working space for storing I/O related parameters.

The UCB's are variable in length. The following locations are of concern to ACP's:

U.CTL Control Flags. This byte contains flags which control how and when the RSX-11M executive will call the device driver. While ACP's are not directly concerned with this process, the setting of these bits is important to correct operation, particularly if writing a UCP. The following bits are defined for this byte:

UC.ALG Alignment bit. If this bit = 0, then byte alignment of data buffers is allowed. Otherwise, the buffers must be word-aligned. This setting is typically a function of the device and is not of importance to ACP's.

UC.ATT Attach/Detach notification. If this bit is set, the driver is called when $GTPKT process an Attach/Detach I/O function. However, if the device is marked mountable (DV.MNT=1) and is mounted (US.MNT and US.FOR=0), then attaches are never allowed. The device will not be notified regardless of the state of this bit.



Symbol Definition Macro: UCBDF$
Macro Source Filename: UCBDF .MAC
Macro Library Filename: EXEMC .MLB
Global Definition File: EXEDF .MAC
Object Library Filename: EXELIB.OLB

Figure B-5
UCB

UC.KIL Unconditional Cancel I/O notification. If this bit is set, the driver is called whenever a cancel I/O request is issued, even if the device is not busy. For ACP enabled devices (US.MNT=0), the I/O queue is never flushed. However, the driver will be called if the unit is busy and will always be called if this bit is set. This feature can be used during I/O rundown (see section 4.5).

UC.QUE Queue bypass bit. If this bit is set, the QIO processor calls the driver without queueing the request. When writing an UCP, this bit is set. This allows the driver to process the I/O request and send it to the ACP, bypassing the normal executive processing.

UC.PWF Unconditional call on power recovery. If this bit is set, the device driver will be always be called when power recovers and also whenever the driver is loaded or the system booted. If the driver is involved with enabling its ACP, this bit provides one mechanism for calling the driver so it can enable the ACP.

UC.NPR NPR device. This bit is set if the device is an NPR device. It determines how U.BUF will be setup. The setting of this bit is a function of the device and is not important to the ACP writer.

UC.LGH Buffer size mask. These bits determine the required buffer size. They are a function of the device and are not important to the ACP writer.

U.STS Unit status. This byte contains various flags related to the status of the device unit. The following bits are defined:

US.BSY Unit busy. This bit is set by $GTPKT whenever a packet is dequeued for a device driver and cleared by $IODON/$IOALT when the packet is finished. The bit is not affected by queueing a packet to an ACP and is not of concern to the ACP writer.

US.MNT Device mounted. The bit is cleared if an ACP is enabled for the unit. It is the responsibility of the ACP enabling process

to clear this bit. The ACP itself usually sets the bit when it finally exits (see US.MDM below).

US.FOR Foreign ACP. This bit is supposed to be set if non-Digital ACP's are enabled for a device. However, the interpretation of this bit by the executive is currently is a state of flux. For both DCP's and UCP's, this bit should be left off.

US.MDM Device marked for dismount. This bit is set by the disabling process when the ACP is initially requested to be disabled. ACP's cannot exit until all processes are terminated. Normally, the ACP examines this bit and its own internal state. When this bit is set and the ACP is idle, it may then exit properly.

U.CW1 Device characteristics word #1. This word contains bit flags defining the type of device. This word and other three device characteristics are returned by a GLUN$ directive. In particular, this word is used by FCS to determine the type of I/O request to issue to the assigned device. The following bits are defined for this word.

DV.REC Record-oriented device
DV.CCL Carriage-control device
DV.TTY Terminal device
DV.DIR Directory device
DV.SDI Single directory device
DV.SQD Sequential device
DV.MXD Mixed Massbus device
DV.UMD Device supports user-mode diagnostics
DV.SWL Device is software write-locked
DV.PSE Pseudo device
DV.COM Device mountable as DECNET Phase I device
DV.F11 Device mountable as Files-11 device
DV.MNT Device mountable

These bits should be set carefully. They are examined in many places and the interpretation is not always consistent. FCS considers any device with DV.REC off to be a block I/O device, supporting the Files-11 QIO's. It also tests DV.DIR to see if directory operations are allowed. The MOU/DMO tasks also examine the bits to determine the proper steps to be taken. In general DV.MNT should always be set for devices your ACP will service. The remainder of the bits should be setup according to the nature of the ACP and its

devices.

U.ACP ACP TCB address. This word is an UCB extension used for mountable devices. It is set when the ACP is enabled to contain the TCB address of the ACP.

U.VCB VCB address. This word is an UCB extension used for mountable devices. It is set when the ACP is enabled to contain the address of the VCB for the unit.

References:

RSX-11M Guide to Writing a Device Driver, pages 4-19 to 4-26. This is a location by location description of the UCB. It is oriented to device drivers.

RSX-11M Guide to Writing a Device Driver, pages C-21 to C-23. This section documents the macro UCBDF$.

RSX-11M Crash Dump Analyzer Manual, pages B-18 to B-25. This section documents the macro UCBDF$.

RSX-11M System Logic Manual, Volume 1, pages 8-51 to 8-56. This section documents the macro UCBDF$. This macro is used to declare the UCB symbolic offsets and bit masks.

B.2.3 Status Control Block

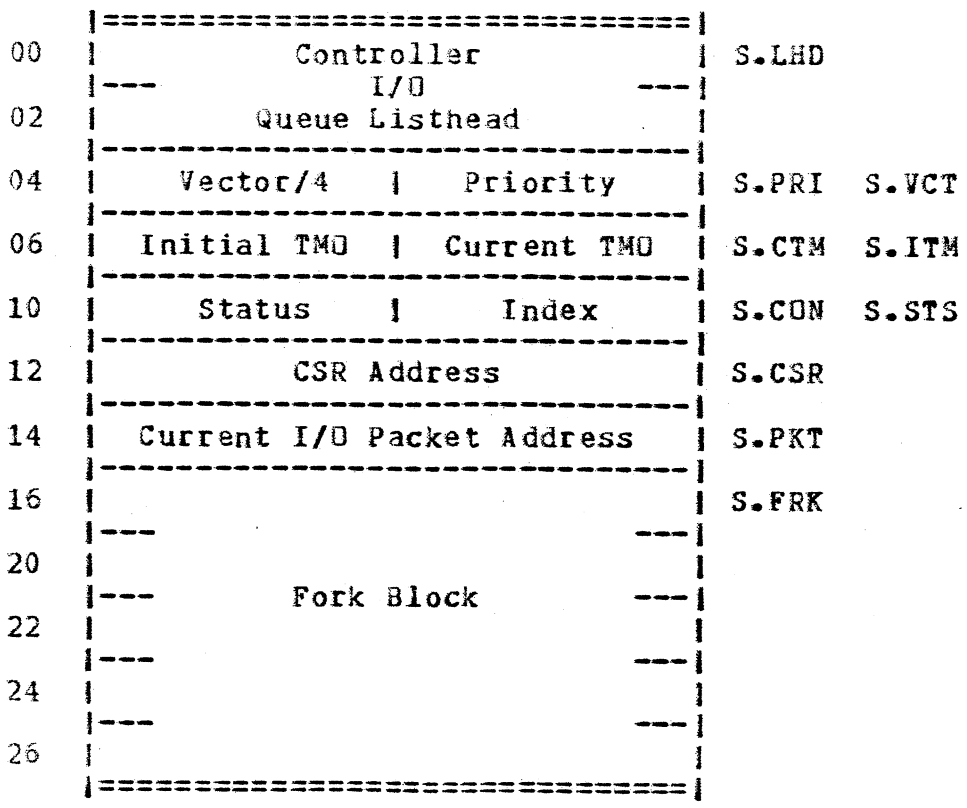
The status control block is used to control access to the hardware controller. ACP's are usually not concerned with this data structure unless they are closely coupled to the device driver.

References:

RSX-11M Guide to Writing a Device Driver, pages 4-15 to 4-19. This is a location by location description of the SCB. It is oriented to device drivers.

RSX-11M Guide to Writing a Device Driver, pages C-17, C-18. This section documents the macro SCBDF$. This macro is used to declare the SCB symbolic offsets.

RSX-11M Crash Dump Analyzer Manual, pages B-10, B-11. This section documents the macro SCBDF$. This macro is used to declare the SCB symbolic offsets.



Symbol Definition Macro: SCBDF$
Macro Source Filename: SCBDF .MAC
Macro Library Filename: EXEMC .MLB
Global Definition File: EXEDF .MAC
Object Library Filename: EXELIB.OLB

Figure B-6
SCB

RSX-11M System Logic Manual, Volume 1, pages 8-44 to 8-46. This section documents the macro SCBDF$. This macro is used to declare the SCB symbolic offsets.

B.3 ACP COMMON DATA STRUCTURES

Two type of data structures are common to all ACP's: window blocks and volume control blocks. Window blocks are used to map a logical process to a specific logical unit. Volume control blocks are used to hold information about each separate entity the ACP processes. While all ACP's use these structures, their content is ACP dependent.

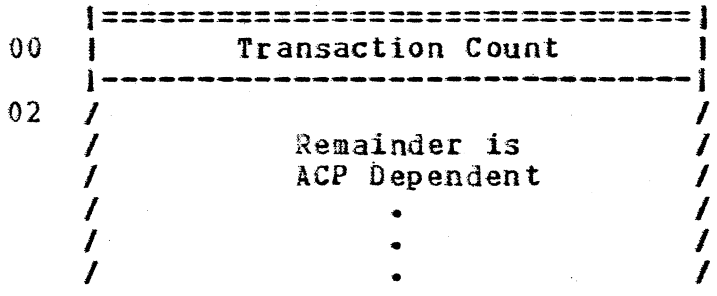


Figure B-7
VCB

B.3.1 Volume Control Block

Typically, each mounted unit serviced by an ACP has a volume control block. This is the data structure in which the ACP should hold information relating to each mounted unit. As such, VCB's are ACP dependent except for the first word.

The first word is called the transaction count. This counter is used by the ACP to determine whether it is idle or not. For DCP's, the counter is incremented whenever a I/O request is queued to the DCP and decrement by the DCP when the request is finished. It is also incremented by the DCP when a process is established and decremented when the process is terminated. The DCP cannot exit unit the counter reaches zero.

The executive does not touch the transaction count for UCP's, however, it is recommended that the UCP's use the transaction count for the same purpose. It is important that an ACP never exit with I/O requests still outstanding in its queue. These requests will never be terminated, leaving the issueing task stuck in I/O rundown.

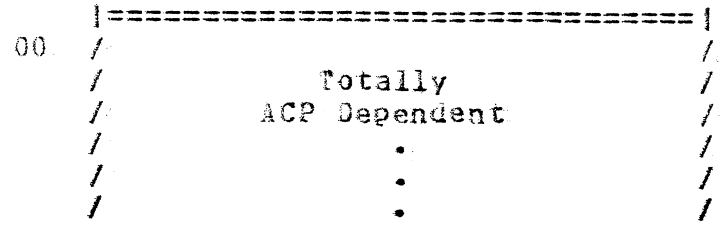


Figure B-8
WINDOW BLOCK

B.3.2 Window Block

One of the most useful features of an ACP is the ability to establish an I/O process and tie various I/O request to the process. In the case of Files-11, this takes the form of opening a file, reading and writing blocks and closing the file. For DECNET, logical links are created, data transmitted and received, and the link closed.

The window block is the data structure used for such processes. It ties the task's LUN to a particular process. However, the format of window blocks and how they are allocated and deallocated is ACP dependent.

The window block address is stored in the second word of the LUT entry. The address of this word is passed to the ACP in the I/O packet. While typically the window is allocated from the executive's pool but this need not be true. The executive never references window address, therefore, the window may come the the ACP's address space. However, if access to the window is needed in processing the I/O request, the window will have to come from the pool.

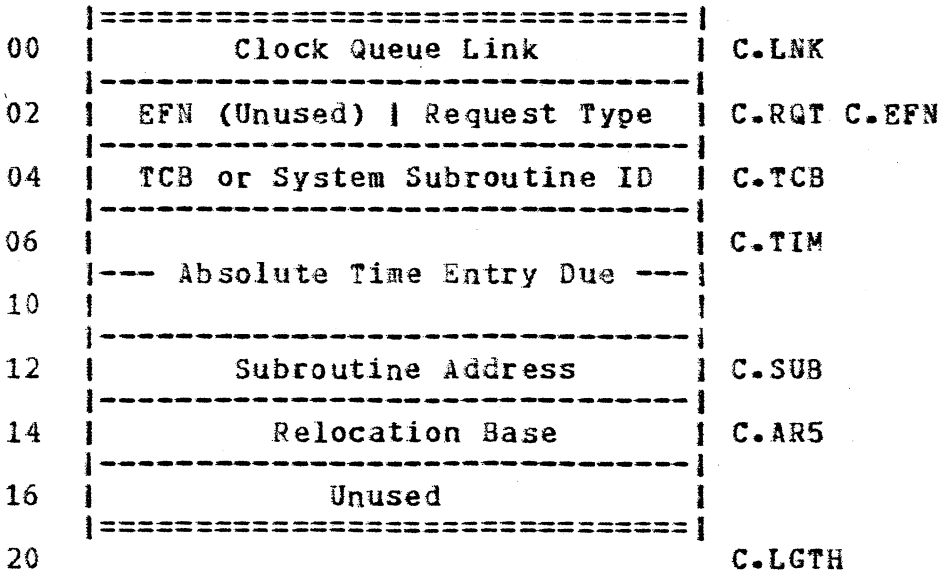
B.4 OTHER DATA STRUCTURES

This is a broad category of every thing else. ACP's are privileged task and have the ability to examine, modify, and create any structure necessary. Some of the most important are mentioned.

8.4.1 Clock Queue Entry

ACP's often have a requirement for keeping internal timers, particularly for event timeouts. One mechanism for doing this is to use the mark time and event flag directives like a normal task. An alternative mechanism is to issue an internal timer request. When this request expires, an executive or device driver routine is called. The format of such a request is shown below.

When the internal timer mechanism is used, a linkage must be made between the kernel code servicing the timer event and the ACP. One method for doing this is for the device driver to issue a special I/O request to the ACP that signifies timer service.



Symbol Definition Macro: CLKDF$
Macro Source Filename: CLKDF .MAC
Macro Library Filename: EXEMC .MLB
Global Definition File: EXEDF .MAC
Object Library Filename: EXELIB.OLB

figure B-9
CLOCK QUEUE ENTRY

The advantage of the internal method is related to the ACP stop mechanism. When idle, ACP's can be swapped out and not brought back until there is something to do. However, if the ACP maintains a constant timer, it will be swapped in continuely, even if idle. If the timer is moved to a driver, the driver can have the intelligance to notify the ACP only when an actual timer event processing is needed.

References:

RSX-11M Guide to Writing a Device Driver, pages C-3, C-4. This section documents the macro CLKDF$. This macro is used to declare the clock queue entry symbolic offsets and bit masks.

RSX-11M Crash Dump Analyzer Manual, pages B-26, B-27. This section documents the macro CLKDF$. This macro is used to declare the clock queue entry symbolic offsets and bit masks.

RSX-11M System Logic Manual, Volume 1, pages 8-31, 8-32. This section documents the macro CLKDF$. This macro is used to declare the clock queue entry symbolic offsets and bit masks.

B.4.2 Partition Control Block

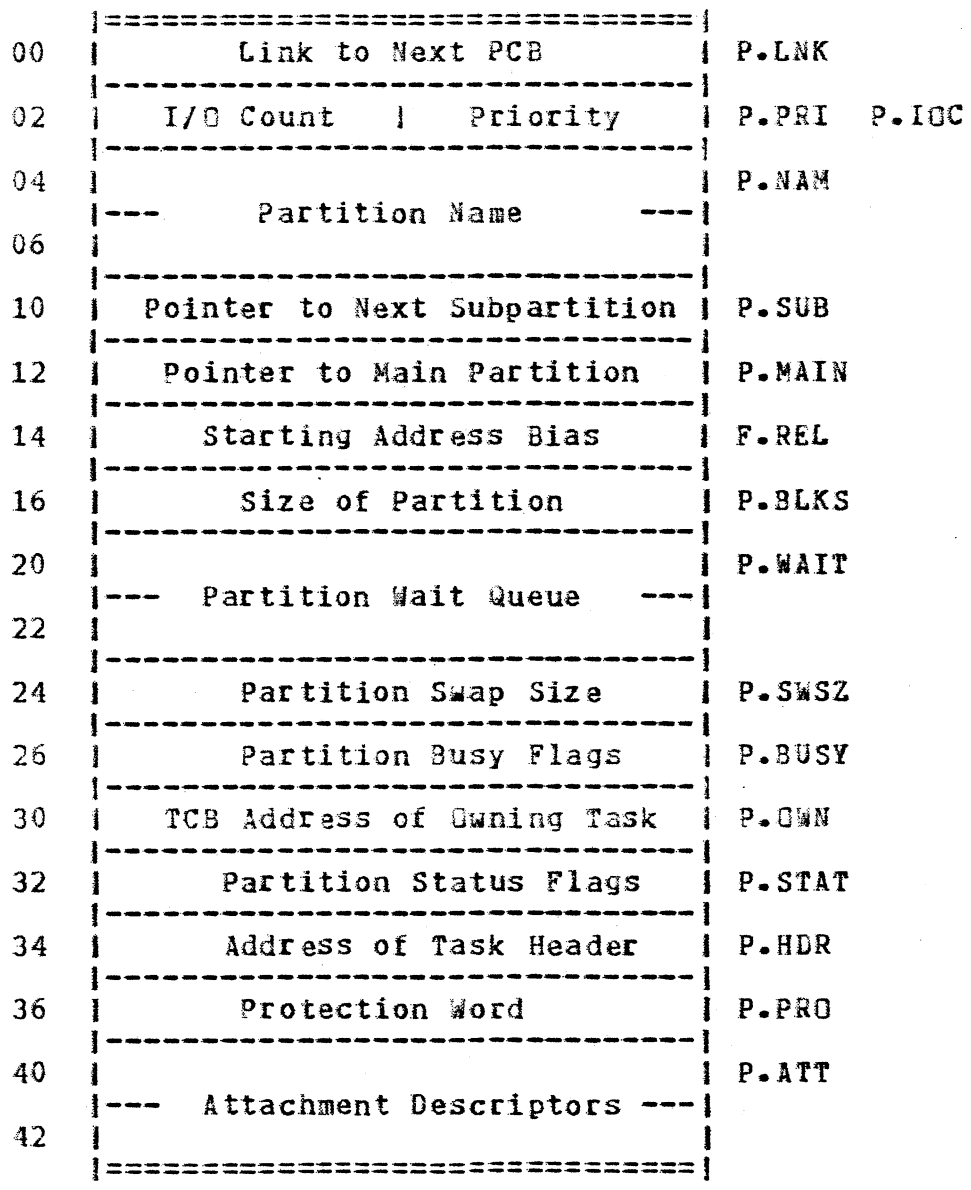
The Partition Control Block (PCB) defines how memory is allocated. It also provides the linkage between the TCB and the task header. Typically, ACP's are unconcerned with PCB's, however, special applications may involve their usage.

References:

RSX-11M Guide to Writing a Device Driver, pages C-14, C-15. This section documents the macro PCBDF$. This macro is used to declare the PCB symbolic offsets and bit masks.

RSX-11M Crash Dump Analyzer Manual, pages B-12 to B-15. This section documents the macro PCBDF$. This macro is used to declare the PCB symbolic offsets and bit masks.

RSX-11M System Logic Manual, Volume 1, pages 8-41 to 8-43. This section documents the macro PCBDF$. This macro is used to declare the PCB symbolic offsets and bit masks.




```
Symbol Definition Macro: PCBDF$
Macro  Source  Filename: PCBDF .MAC
Macro  Library Filename: EXEMC .MLB
Global Definition  File: EXEDF .MAC
Object Library Filename: EXELIB.OLB
```

Figure B-10
PCB

B.4.3 Task Control Block

The Task Control Block is the primary data structure for a task. ACP's are tasks and have a TCB. Besides the normal treatment, the ACP's TCB receives the following special treatment.

1. The address of the ACP's TCB is stored in the UCB for each enabled device. The address is used to determine to which task to queue the I/O request.

2. The I/O packets are queued to the ACP through the receive queue listhead (T.RCVL) and deallocated by the ACP by removing an entry from this queue.

3. ACP's are marked by a special bit in the third status word (T.ST3). This bit (T3.ACP) is used to prevent tasks from sending messages to the ACP. This is necessary to prevent confusion resulting from multiple usage of the receive queue. The bit is set by the task builder /AC switch.

4. For certain Digital ACP's (F11ACP, MTAACP), the second event flag word (T.EFLG+2) is used as a mounted volume counter and therefore event flags 16-31(10) are not used. However, this restriction applies only if desired.

References:

RSX-11M Guide to Writing a Device Driver, pages C-19, C-20. This section documents the macro TCBDF$. This macro is used to declare the TCB symbolic offsets and bit masks.

RSX-11M Crash Dump Analyzer Manual, pages B-15 to B-17. This section documents the macro TCBDF$. This macro is used to declare the TCB symbolic offsets and bit masks.

RSX-11M System Logic Manual, Volume 1, pages 8-47 to 8-49. This section documents the macro TCBDF$. This macro is used to declare the TCB symbolic offsets and bit masks.

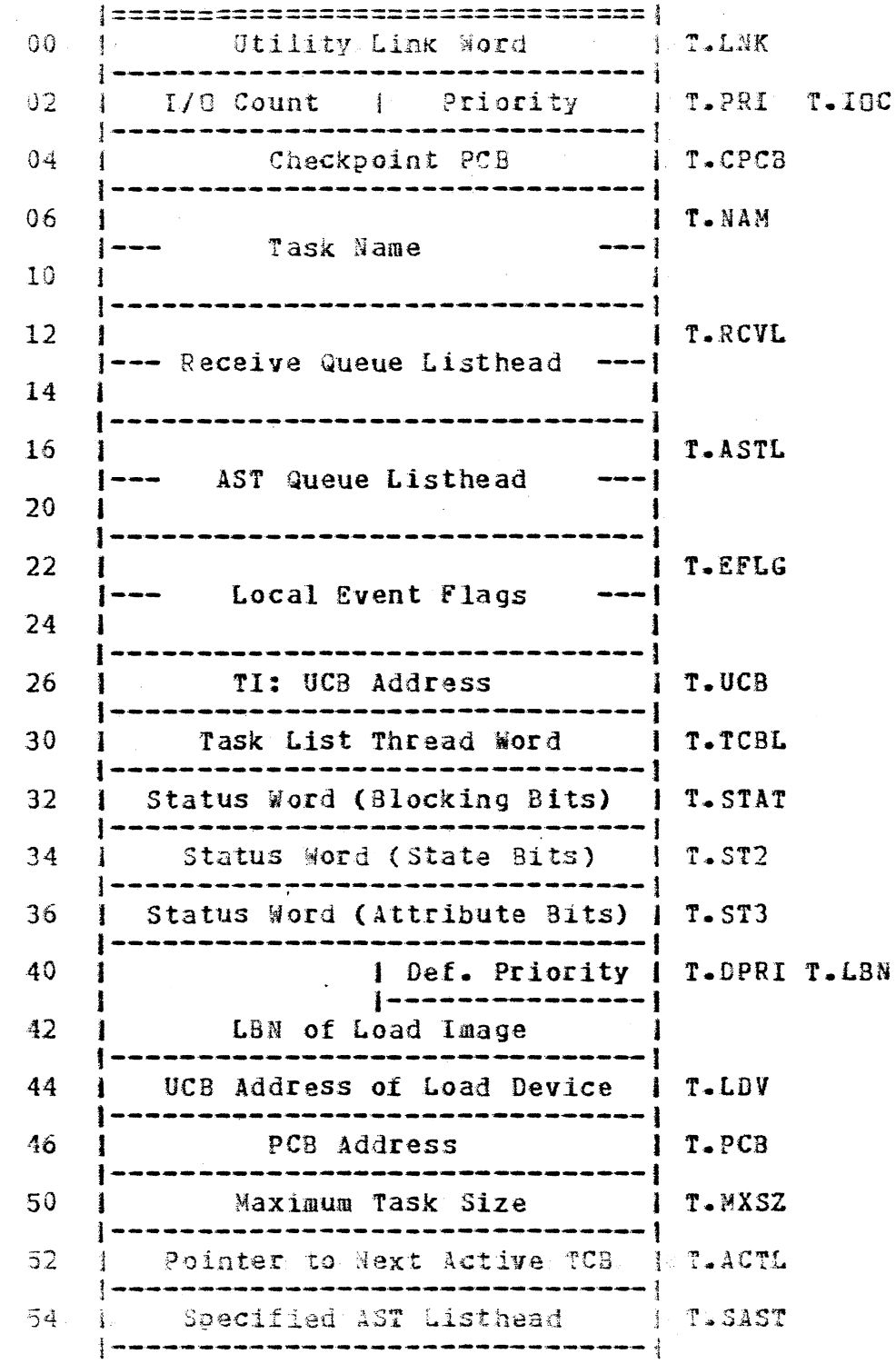
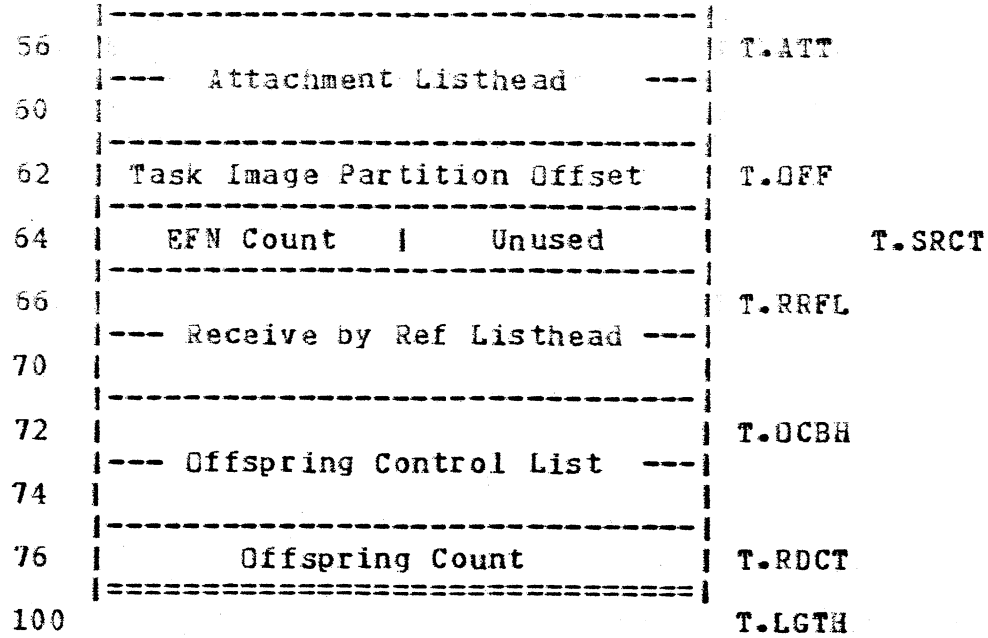


Figure B-11
TCB



Symbol Definition Macro: TCBDF$
Macro Source Filename: TCBDF .MAC
Macro Library Filename: EXEMC .MLB
Global Definition File: EXEDF .MAC
Object Library Filename: EXELIB.OLB

Figure B-11
TCB

B.4.4 Task Header

The task header contains the context of the task. The most important thing to remember about task headers is that they are destroyed when a task is checkpointed and recreated when it is swapped in. Therefore, ACP's cannot store the address of the header (or the LUT table) for later use.

References:

RSX-11M Guide to Writing a Device Driver, pages C-9, C-10. These sections document the macro HDRDF$. This macro is used to declare the task header symbolic offsets.

RSX-11M Crash Dump Analyzer Manual, pages B-4 to B-6. These sections document the macro HDRDF$. This macro is used to declare the task header symbolic offsets.

RSX-11M System Logic Manual, Volume 1, pages 8-49, 8-50. These sections document the macro HDRDF$. This macro is used to declare the task header symbolic offsets.

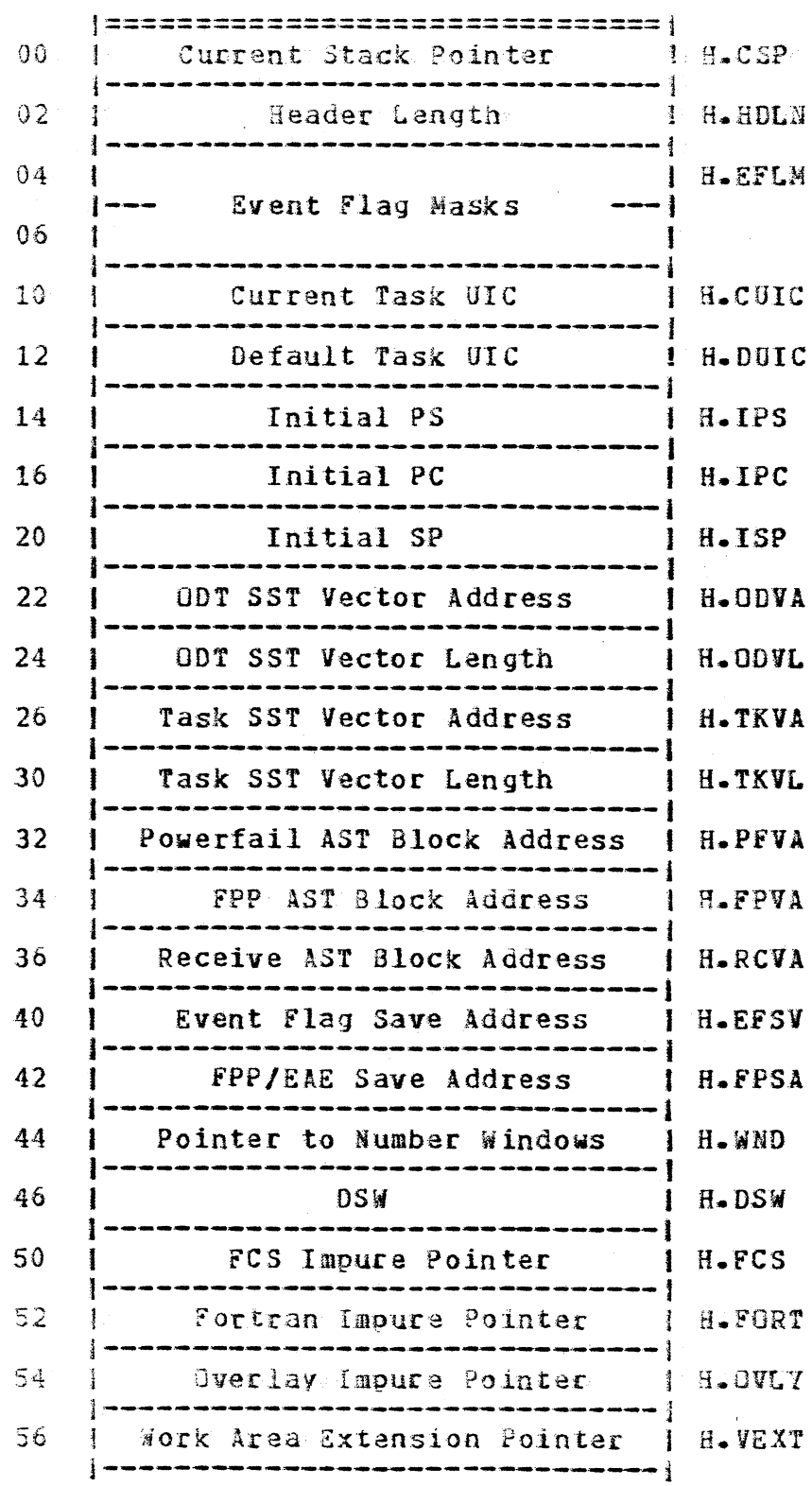
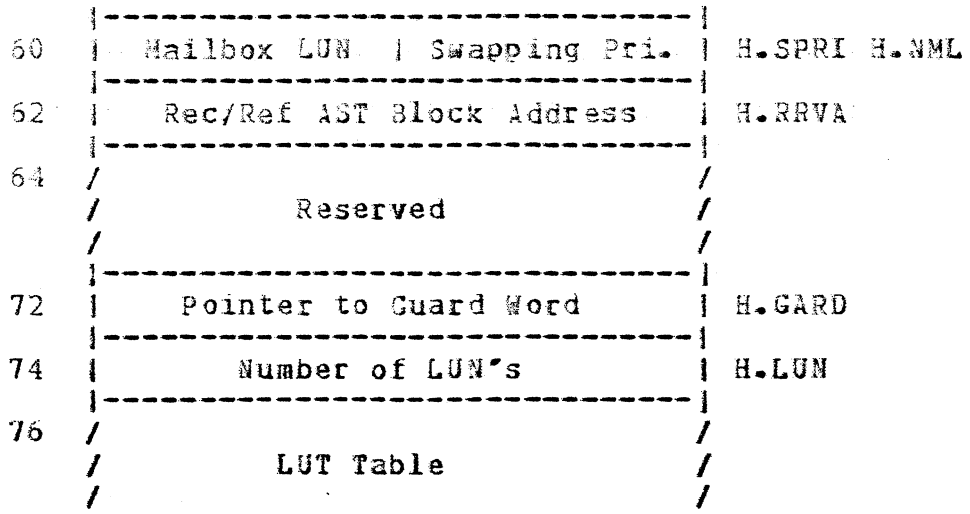


Figure B-12
TASK HEADER



Symbol Definition Macro: HDRDF$
Macro Source Filename: HDRDF .MAC
Macro Library Filename: EXEMC .MLB
Global Definition File: EXEDF .MAC
Object Library Filename: EXELIB.OLB

Figure B-12
TASK HEADER

APPENDIX C

FCS-11 MODULES

The File Control Services are a set of routines which are linked to a task to perform various I/O services. The routines are typically called via macros. The documentation for FCS is found in the IAS/RSX-11 I/O Operation Reference Manual.

There are two levels of FCS routines. The upper level routines are called directly by the user's task or via the FCS macros. These routines are named '.XXXXX'. The lower level routines are intended for FCS's own use and are not expected to be called directly by the user. These routines are named '..XXXX'. This appendix describes each FCS module, its entries, and their calling parameters and function. The appendix also discusses the FCS data structures and some basic internal concepts.

C.1 FCS CONDITIONALS

Each FCS module is assembled with the prefix file FCSPRE.MAC. The file establishes various symbolic definitions and common macros and defines the assembly control flags. FCS is very heavily conditionally assembled. The symbols used and their default RSX-11M values are as follows:

R$$ANI ANSI magtape support. This variable is set to zero for no support and to one if ANSI magtape support is desired. The default for RSX-11M is zero unless the ANSI prefix file, ANSPRE.MAC, is used for assembly.

R$$BBF Big buffer support. This feature is used for I/O devices that support block sizes greater then 512 bytes. The variable is set to zero for no support and to one if support is desired. The default for RSX-11M is zero unless the ANSI prefix file, ANSPRE.MAC, is used for assembly.

R$$DPB QIO DPB format. This variable is set to zero for

old style QIO's and to one for the new QIO DPB format. At this time, all systems (including RSX-11M) use the new format.

R$$EIS Extended Instruction Set (EIS) support. This variable is set to zero if no EIS instructions should be used by FCS and to one if EIS instructions are allowed. RSX-11M always sets this variable to zero.

R$$ELP Extened parsing support. This variable is set if extended parsing support for VAX/VMS systems is desired. RSX-11M always sets this variable to zero.

R$$LCL Force local definition of symbols. This variable is set to zero if global definitions for FCS variable should be used and to one if local definitions should be declared in FCSPRE.MAC. RSX-11M sets this variable to one.

R$$MPL RSX-11M PLUS support. This variable is set to one if FCS should be assembled for an RSX-11M PLUS system. RSX-11M always sets this variable to zero.

R$$MUL Multiple buffering support. This variable is set to zero if FCS supports only one I/O block buffer and non-zero if multiple I/O buffers are supported. The symbol R$$MBF is equated to this symbol. RSX-11M always sets this variable to zero.

R$$NAM Named directory support. This variable is set one or two for named directory support. This is a feature of the SCS-11 version of FCS. RSX-11M always sets this variable to zero.

R$$OPF Type of open. This variable is used to control the type of assembly desired for the module OPEN. If set to zero, the normal OPEN routine is assembled. Otherwise, a value of one (OPFNB.MAC) assembles the open by FNB and a value of two (OPFID.MAC) assembles the open by FID module. This variable is not set in FCSPRE.MAC.

R$$RSL Assemble for SYSLIB or resident library. This variable is set to zero if FCS should be assembled for placement in SYSLIB.OLB or one if a resident library should be built from FCS. RSX-11M always sets this variable to one. This causes FCS to assembled using the PSECT $$RESL.

R$$SCS SCS-11 support. This variable is set to one if FCS should be assembled for the SCS-11 operating system. RSX-11M always sets the variable to zero.

R$$SEQ Type of file I/O. This variable is used to control the type of assembly desired for the GET and PUT modules. If set to zero, the modules support both sequential and random I/O. If set to one (PUTSQ, GETSQ), the modules are restricted to sequential I/O. This variable is not set in FCSPRE.MAC.

R$$SPL Automatic spooling support. This variable is set to zero if no automatic spooling support is desired and to one if it is supported. RSX-11M always sets this variable to zero, support of automatic spooling requires the IAS spooling mechanism.

R$$VMS VAX/VMS support. This variable is set to one if FCS should be assembled for VAX/VMS operating system. RSX-11M always sets this variable to zero.

R$$11M RSX-11M support. This variable is set to zero if IAS support is desired and to one if RSX-11M support is to be generated. Naturally, RSX-11M sets this variable to one.

Figure C-1 summarizes the default conditional switch settings for the five operating systems supported by FCS. In general, RSX-11M sets the FCS conditional symbols to assemble the simplest version of FCS. It is unclear what will happen if some of the more useful features (R$$EIS, R$$SPL, R$$MUL) are turned on and used in an RSX-11M system.

	RSX-11M	RSX-11M+	SCS-11	VAX/VMS	IAS/11D
R$$11M	1	1	1	1	0
R$$MPL	0	1	0	0	0
R$$SCS	0	0	1	0	0
R$$VMS	0	0	0	1	0
R$$ANI	0	0	0	1	1
R$$BBF	0	0	0	0	0
R$$DPB	1	1	1	1	1
R$$EIS	0	0	1	0	1
R$$ELP	0	0	1	0	0
R$$LCL	1	1	1	1	1
R$$MPL	0	0	0	0	1
R$$NAM	0	0	2	0	0
R$$RSL	1	1	1	1	0
R$$SPL	0	0	0	0	1

FIGURE C-1
FCS DEFAULT CONDITIONALS

C.2 FCS DATA STRUCTURES

<To be written>

C.2.1 File Descriptor Block

<To be written>

C.2.2 Filename Block

<To be written>

C.2.3 Dataset Descriptor

<To be written>

C.2.4 $$FSR1 Region

<To be written>

C.2.5 $$FSR2 Region

<To be written>

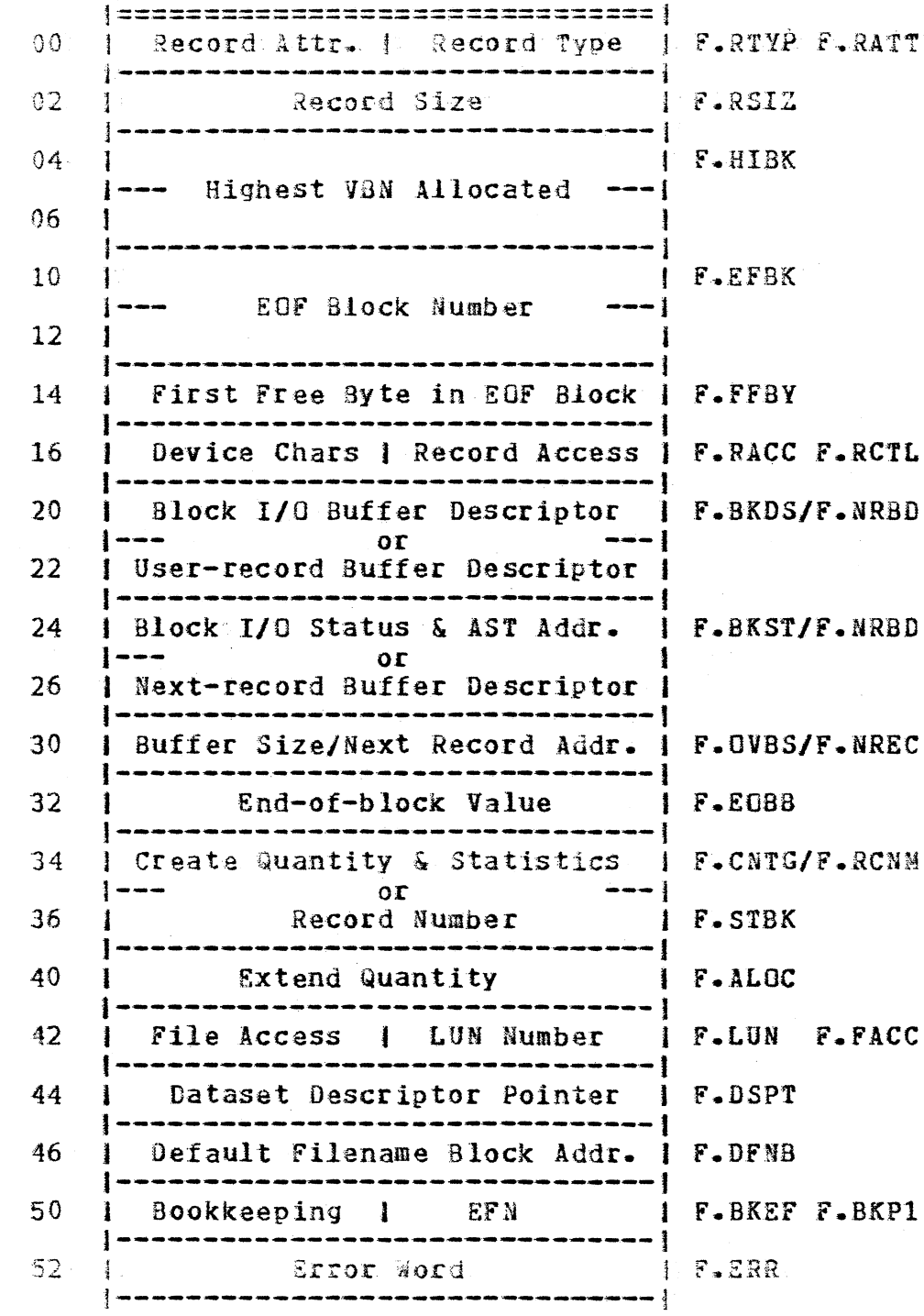
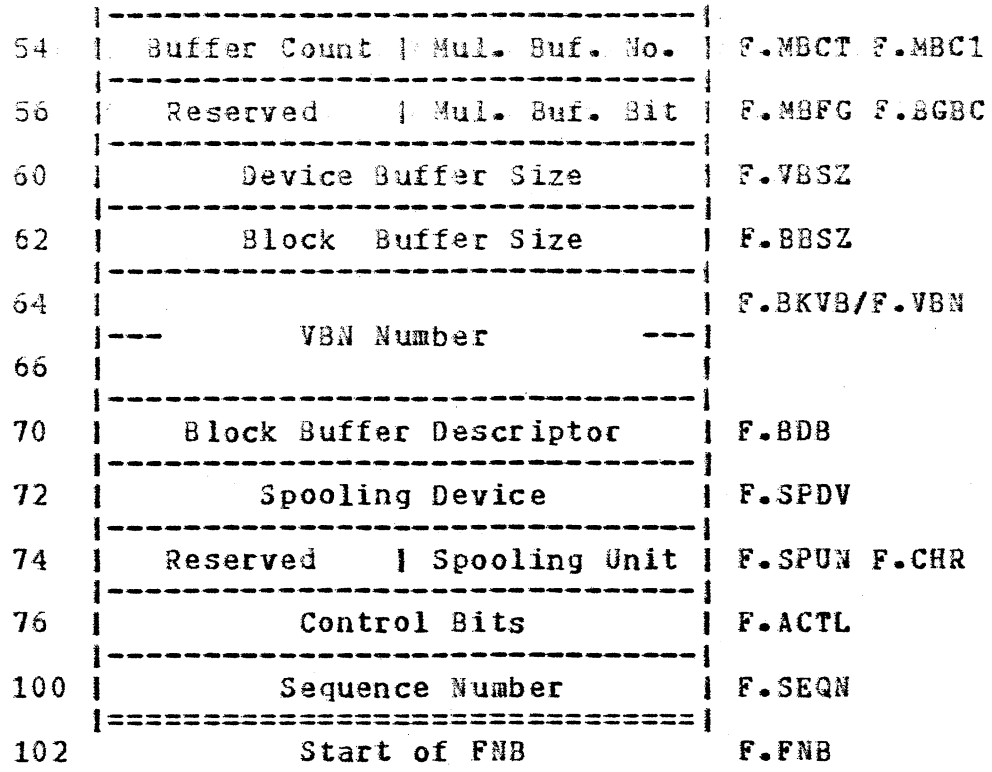
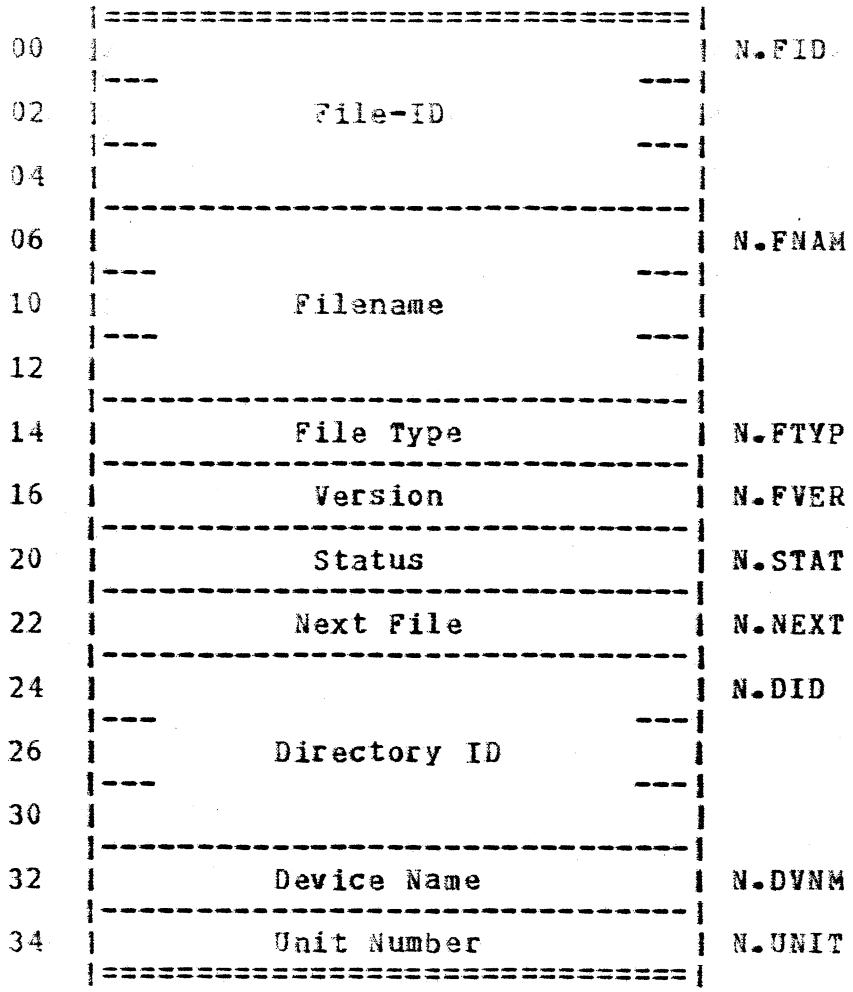


Figure C-2
FDB



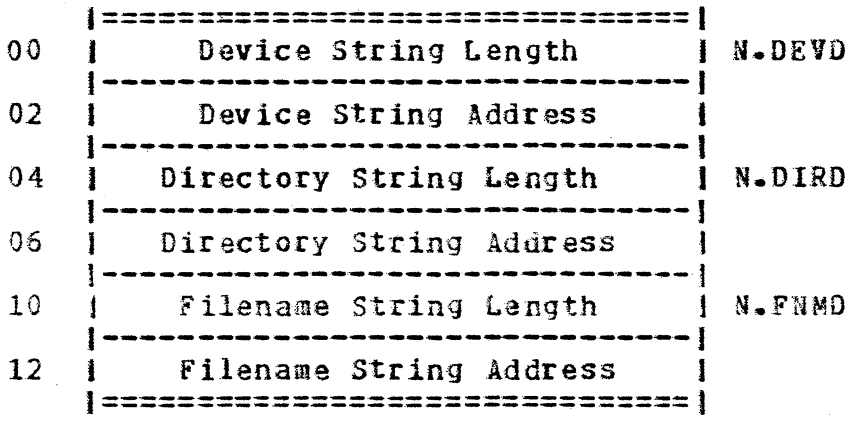
Symbol Definition Macro: FCSBT$, FDOFF$
Macro Source Filename: FCSMAC.MAC
Macro Library Filename: RSXMAC.SML
Global Definition File: FCSGBL.MAC
Object Library Filename: SYSLIB.OLB

Figure C-2
FDB



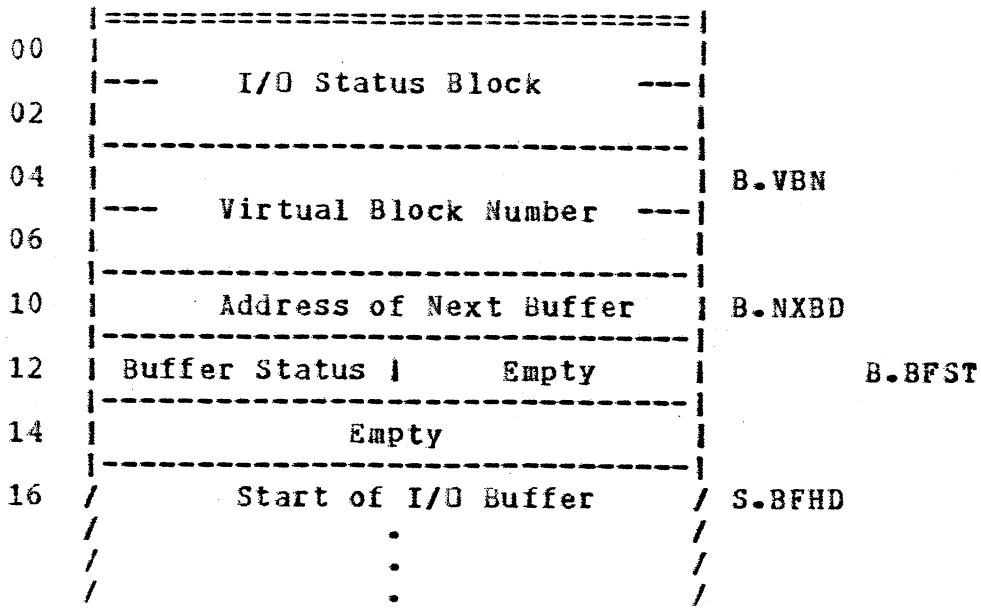
Symbol Definition Macro: FCSBT$, NBOFF$
Macro Source Filename: FCSMAC.MAC
Macro Library Filename: RSXMAC.SML
Global Definition File: FCSGBL.MAC
Object Library Filename: SYSLIB.OLB

Figure C-3
FNB



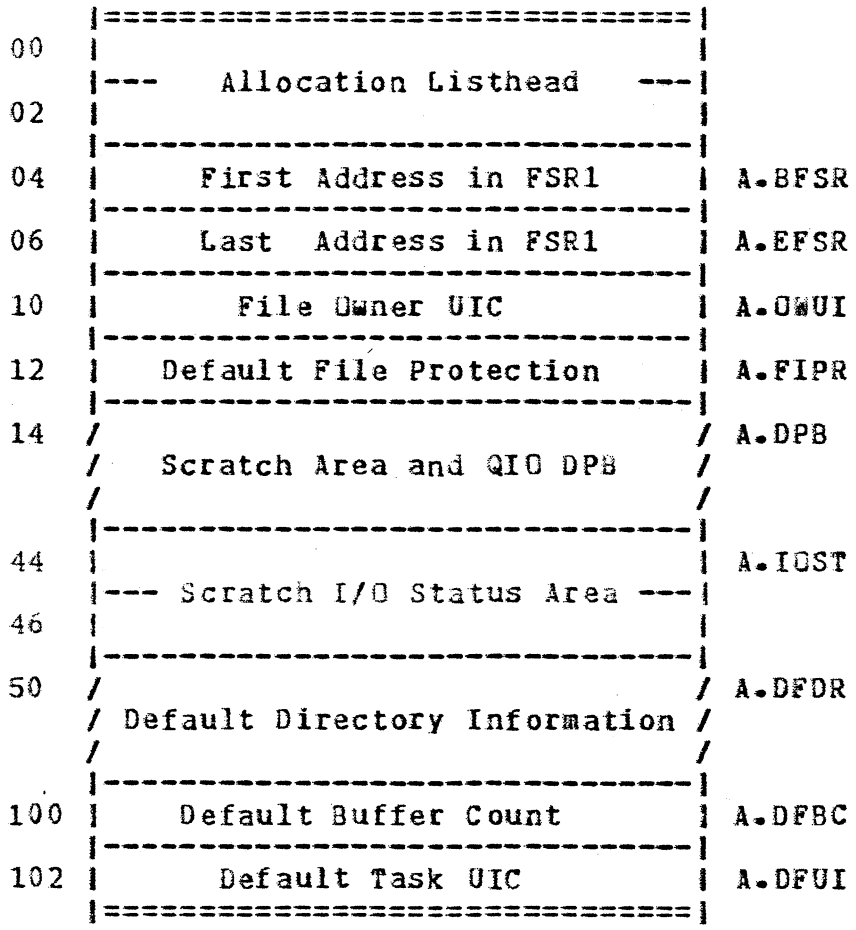
Symbol Definition Macro: FDSOF$
Macro Source Filename: FCSMAC.MAC
Macro Library Filename: RSXMAC.SML
Global Definition File: FCSGBL.MAC
Object Library Filename: SYSLIB.OLB

Figure C-4
DATASET DESCRIPTOR



Symbol Definition Macro: BDOFF$
Macro Source Filename: FCSMAC.MAC
Macro Library Filename: RSXMAC.SML
Global Definition File: FCSGBL.MAC
Object Library Filename: SYSLIB.OLB

FIGURE C-5
$$FSR1



Symbol Definition Macro: FSROF$
Macro Source Filename: FCSMAC.MAC
Macro Library Filename: RSXMAC.SML
Global Definition File: FCSGBL.MAC
Object Library Filename: SYSLIB.OLB

FIGURE C-6
$$FSR2

C.3 FCS INTERNALS

<To be written>

C.4 FCS MODULES

The remainder of this appendix covers each FCS source module. For each module, the calling label and inputs and outputs are detailed

C.4.1 ANSPAD

This routine pads the rest of the buffer for magtape ANSI "D" format. It is only generated if magtape support (R$$ANI) is selected.

Entry: ..ANSP

Input: R0 = FDB address

F.NREC(R0) = Starting address in buffer to pad
F.EOBB(R0) = Byte address beyond end of buffer

Output: C = 0, If buffer padded
= 1, If buffer not padded

R0-R4 preserved, R5 destroyed

Conditionals: R$$ANI

C.4.2 ASCPPN

This routine translates an UIC string in the form of "[200,210]" to the binary equivalent. Both the group and owner numbers are assumed to be octal unless followed by a period. In this case, the numbers are translated as decimal. The output is stored in ".BYTE owner,group" form.

Entry: .ASCPP

Input: R2 = Address of UIC string descriptor

R3 = Address to return the binary UIC

0(R2) = Size of UIC string
2(R2) = Address of UIC string

Output: C = 0, String converted and value stored
= 1, UIC string syntax error

All registers preserved

Conditionals: None

C.4.3 ASCR50

This routine converts an ASCII string to RAD50 and stores in the specified location(s).

Entry: ..SGR5

Input: R2 = Address of ASCII string
R3 = Size of ASCII string
R4 = Starting address to store RAD50 string

The storage area must be previously zeroed.

Output: C = 0, Conversion complete
= 1, Conversion failed, nonalphanumeric character

R4 = Address of last word written in RAD50

R0,R1 preserved, R2,R3,R5 destroyed

Conditionals: R$$EIS

C.4.4 ASSLUN

This routine assigns the lun in F.LUN of the FDB to the device specified by the filename block. If no device is specified, the previously assigned device is used. If no device was assigned, the default device (SY:) is assigned.

Once the device is assigned, the device characteristics are stored in the FDB. Specifically, the low byte of the the device characteristics (U.CW1) is stored in F.RCTL and the device buffer size (U.CW4) is stored in F.VBSZ and F.BBFS. Also, the true device name and unit are stored in the filename block.

```
Entry:  .ASLUN,..ALUN

Input:  R0 = FDB address
        R1 = FNB address

        N.UNIT(R1) = Device unit number
        N.DVNM(R1) = Device name or 0 if none

Output: C = 0, Assign successful
          = 1, Bad device name

        F.RCTL(R0) = Device characteristics
        F.VBSZ(R0) = Device block size
        F.BBSZ(R0) = Device block size

        If entry at .ASLUN, all registers preserved

        If entry at ..ALUN, R0,R1 preserved, R2-R5
        destroyed

Conditionals: R$$ANI,R$$SPL,R$$11M
```

C.4.5 BDBREC

This routine sets up the block buffer header and the FDB for the next virtual block I/O.

```
Entry:  ..BDRC

Input:  R0 = FDB address
        R1 = Buffer descriptor address

Output: B.VBN(R1)  = Set to F.VBN(R0)
        B.BBFS(R1) = Set to F.BBFS(R0)
        F.NREC(R0) = Set to start of I/O buffer
        F.EOBB(R0) = Set to end of I/O buffer

        R0,R2-R5 preserved, R1 destroyed

Conditionals: None
```

C.4.6 BIGBUF

This module contains routines used for large buffer support. This support is not enabled for RSX-11M (R$$BBF>0). INMBB checks for a specified VBN in the block buffer. RSTEOF

correctly processes EOF's for large buffer I/O.

C.4.7 BKRG

This routine sets up the registers for block I/O, using the block buffer definition in the FDB.

Entry: ..BKRG

Input: R0 = FDB address

F.BDB(R0) = Buffer descriptor address

Output: R1 = Address of I/O buffer
R2 = Buffer size (F.BBFS)
R3 = Carriage control, always set to zero

R0,R4,R5 preserved

Conditionals: None

C.4.8 CKALOC

This module contains routines to allocate blocks to a file. The routines are designed for internal FCS usage.

The ..ALOC and ..ALC1 routines are used to allocate blocks if necessary. The desired allocation is compared to the current allocation and more blocks are allocated if necessary.

Entry: ..ALOC

Input: R0 = FDB address

F.EFBK(R0) specifies block to allocate to.

Entry: ..ALC1

Input: R0 = FDB address
R1 = High VBN to allocate to
R2 = Low VBN to allocate to

Output: C = 0, Blocks allocated, F.HIBK(R0) updated
= 1, Allocation failed, F.ERR(R0) contains error code

All registers preserved

The ..EXTD and ..EXT1 routines are used to extend the file. ..EXTD computes the extend size, using F.ALOC if it is non-zero. Otherwise it subtracts the current block allocation from the contents of R1,R2 and uses the resulting value. ..EXT1 actually issues the extend QIO request.

Entry: ..EXTD

Input: R0 = FDB address
R1 = High VBN to extend to
R2 = Low VBN to extend to

Output: C = 0, Extend successful
= 1, Extend failed

All registers preserved

Entry: ..EXT1

Input: R0 = FDB address
R1 = 0, noncontiguous allocation
= 1, contiguous allocation
R2 = Number of blocks to extend

Output: C = 0, Successful extend, F.HIBK adjusted
= 1, Extend failed

R0 preserved, R1-R5 destroyed

Conditionals: R$$DPB

C.4.9 CLOSE

This module closes an open file, writing any remaining buffers and rewriting the file header with the final record attributes. The FDB is reset.

Entry: .CLOSE

Input: R0 = FDB address

Output: C = 0, successful completion
= 1, error writing header

All registers preserved

Conditionals: R$$ANI,R$$BBF,R$$MBF,R$$SPL

C.4.10 COMMON

This module contains two commonly used routines. The first, ..FCSX, is used to properly set the carry bit if an error code is set in F.ERR.

Entry: ..FCSX

Input: R0 = FDB address

Output: C = 0, F.ERR(R0) is positive
= 1, F.ERR(R0) is LE 0

All registers preserved

Conditionals: None

The second entry, .FATAL, is used by FCS to declare fatal errors. A BPT instruction is issued. The current version of FCS only uses this routine if an event flag directive fails.

C.4.11 CONTRL

This module is used to issue the IO.APC function. It is normally intended for use with magtapes.

Entry: .CTRL,..CTRL

Input: R0 = FDB address
R1 = Function code
R2 = Value
R3 = Parameter list address

Output: C = 0, operation successful
= 1, operation failed, F.ERR(R0) = error code

If .CTRL, all registers preserved

IF ..CTRL, R0 preserved, R1-R5 destroyed

Conditionals: None

C.4.12 CREATE

This module contains the internal FCS code for creating a new file on the disk. If called for a record I/O device (FD.REC=1), the routine is effectively a no-op. Otherwise, the IO.CRE function is issued.

Entry: ..CREA

Input: R0 = FDB address
R1 = FNB address

Output: C = 0, file created
= 1, operation failed, F.ERR(R0) = error code

R0,R1 preserved, R2-R5 destroyed

Conditionals: R$$ANI

C.4.13 DEL

This module contains the internal FCS delete code. The file is removed from the directory and marked for delete. The FNB is assumed to contain the filename and file-ID.

Entry: ..DEL1

Input: R0 = FDB address
R1 = FNB address

Output: C = 0, delete successful
= 1, operation failed, F.ERR(R0) set

R0,R1 preserved, R2-R5 destroyed

Conditionals: None

C.4.14 DELETE

This module contains the user interface for deleting a file. If the file is opened it is first closed. Then the FNB is set up using the dataset descriptor in F.DSPT(R0) and default filename block in F.DFNB(R0). Finally, the file is deleted.

Entry: .DELET

Input: R0 = FDB address

Output: C = 0, delete successful
= 1, operation failed, F.ERR(R0) set

All registers preserved

Conditionals: None

C.4.15 DIDFND

This routine begins to find the default task directory. The directory number stored in A.DFUI of $$FSR2 is converted to RAD50 and stored in the filename portion of the FNB. The routine continues at ..DID.

Entry: ..DIDF

Input: R0 = FDB address
R1 = FNB address
R2 = FSR2 address

Output: Binary UIC converted to RAD50 and stored in FNB.

See DIFND (next section) for remainder.

Conditionals: None

C.4.16 DIFND

This module continues the directory lookup process. The file type is set to DIR (RAD50) and the version to one. A find is then performed on the master file directory. If successful, the resulting file-ID is set as the directory-ID and the remainder of the FNB is cleared.

Entry: ..DID1, ..DID

Input: R0 = FDB address
R1 = FNB address

N.FNAM(R1) = Directory name

```
                0(SP) = Saved R2
                2(SP) = Saved R3
                4(SP) = Return address

                ..DID1 saves R2,R3 on stack and falls into ..DID

        Output: C = 0, directory found, N.DID setup
                  = 1, no directory, F.ERR(R0) set

                R0-R3 preserved, R4,R5 destroyed

        Conditionals: None
```

C.4.17 DIRECT

This module contains the code used to issue the directory primitives: enter filename in directory (IO.ENA), find filename in directory (IO.FNA), and remove filename from directory (IO.RNA).

```
        Entry:  ..ENTR, ..FIND, ..RMOV

        Input:  R0 = FDB address
                R1 = FNB address

                Filename block setup for desired operation

        Output: C = 0, operation successful, FNB filled in
                  = 1, operation failed, F.ERR(R0) set

                R0,R1 preserved, R2-R5 destroyed

        Conditionals: R$$ANI,R$$DPB
```

C.4.18 DIRFND

This module contains the first half of directory string processing. The directory string is turned into a RAD50 string and stored as the desired filename. Processing continues in DIFND at ..DID. The routine is mostly concerned with named directories, however, this code is disabled for RSX-11M.

```
        Entry:  ..DIRF

        Input:  R0 = FDB address
```

R1 = FNB address
R2 = Directory string descriptor

Output: C = 0, operation successful, directory-ID setup
= 1, operation failed, F.ERR(R0) set

R0-R3 preserved, R4,R5 destroyed

Conditionals: R$$NAM,R$$SCS

C.4.19 DIRNAM

This routine is only used for SCS-11 systems. It is a part of that systems directory processing.

C.4.20 DLFNB

This routine is the user interface for deleting a file by filename block. The FNB portion of the FDB is assumed to be setup with the filename, type, version (must be explicit), directory-ID, device, and unit. The file is first closed, removed from the directory, and then deleted.

Entry: .DLFNB

Input: R0 = FDB address

Output: C = 0, operation successful
= 1, operation failed, F.ERR(R0) set

All registers preserved

Conditionals: None

C.4.21 ELPARS

This routine contains code which bypasses the normal FCS filename parsing and uses the facilities provided by the operating system. Such support is only turned on if FCS is assembled for a VAX/VMS system

C.4.22 EOFCHK

This module contains the EOF checking code for record I/O. The routine ..SEFB checks to see if the current virtual block (F.VBN) is at or beyond the EOF block (F.EFBK).

Entry: ..SEFB

Input: R0 = FDB address

F.VBN(R0) = Current virtual block number
F.EFBK(R0) = EOF block number
F.FFBY(R0) = Last byte in EOF block

Output: C = 0, if before or at EOF block
= 1, if beyond EOF block

FD.EFB in F.BKP1(R0) set if at or beyond EOF block

All registers preserved

The routines ..EFCK and ..EFC1 check to see if the current record position is at or beyond the EOF position. The first entry calls ..SEFB to properly setup FD.EFB. ..EFC1 assumes this bit is already properly set. If at the EOF block, the current record position is compared against the placement of the EOF within the block.

Entry: ..EFCK, ..EFC1

Input: R0 = FDB address

Output: C = 0, if not at EOF
= 1, if at EOF, F.ERR(R0) = IE.EOF

R0,R2-R5 preserved, R1 destroyed

Conditionals: None

C.4.23 EXTEND

This module contains the user interface for extending a file. The file may be opened or closed.

Entry: .EXTND

Input: R0 = FDB address
R1 = Number of blocks to extend file by
R2 = Type of extend

0 = Non-contiguous
1 = Contiguous

Output: C = 0, operation successful, F.HIBK(R0) adjusted
= 1, operation failed, F.ERR(R0) set

All registers preserved

Conditionals: None

C.4.24 FCSFSR

This module contains the FSR region declarations. $$FSR1 is declared as a blank psect. The user is expected to expand it later. The $$FSR2 region is declared to be of size S.FSR2.

C.4.25 FINIT

This module contains the FCS initialization code. Specifically, the size of the $$FSR1 region is calculated and the default UIC is set from the task's UIC.

Entry: .FINIT, ..FINI

Input: R1 = Address of $$FSR2 region (..FINI only)

Output: FSR regions initialized

All registers preserved if entry at .FINIT

R0,R1 preserved, R3-R5 destroyed if entry at ..FINI

Conditionals: None

C.4.26 GET

This module contains the code for inputting a logical record from the device/file. The module can be assembled for either sequential-only support (R$$SEQ) or random/sequential

support. See the RSX-11/IAS I/O Operations Reference Manual, pages 3-18 to 3-23 for further details on calling sequence and return values.

Entry: .GET, .GETSQ

Input: R0 = FDB address

Output: C = 0, record input
= 1, operation failed, F.ERR(R0) set

All registers preserved

Conditionals: R$$ANI,R$$BBF,R$$EIS,R$$RSL,R$$SEQ,R$$11M

C.4.27 GETDI

This routine is the second part of the directory look-up process. It saves the current context of the FNB and calls either ..PDID or ..DIRF to set the directory-ID. The FNB is then restored.

Entry: ..GTDI

Input: R0 = FDB address
R1 = FNB address
R2 = Directory string descriptor address

0(SP) = Directory lookup routine (..PDID/..DIRF)
2(SP) = Return address

Output: C = 0, operation successful
= 1, operation failed, F.ERR(R0) set

All registers preserved

Conditionals: R$$SPL

C.4.28 GETDID

This routine gets the default task directory-ID and stores it in the specified filename block. It sets-up to call ..PDID and continues in ..GTDI.

Entry: .GTDID

Input: R0 = FDB address
R1 = FNB address

Output: See ..GTDI

All registers preserved

Conditionals: R$$ELP

C.4.29 GETDIR

This routine look ups the specified directory and stores it in directory-ID in the filename block. It sets-up to call ..DIRF and continues in ..GTDI.

Entry: .GTDID

Input: R0 = FDB address
R1 = FNB address
R2 = Directory descriptor address

Output: See ..GTDI

All registers preserved

Conditionals: R$$ELP

C.4.30 MKDL

This module contains the internal routine to mark a file for deletion. While the file is deleted by this routine, the directory entry is not removed.

Entry: ..MKDL

Input: R0 = FDB address
R1 = FNB address

Output: C = 0, delete successful
= 1, delete failed, F.ERR(R0) set

R0 preserved, R1-R5 destroyed

Conditionals: None

C.4.31 MOVREC

This routine transfers records to and from user buffers. It is designed to work correctly for odd address and odd byte counts.

Entry: ..MVR1

Input: R0 = FDB address
R1 = Address of source
R2 = Address of destination
R3 = Size of block to move

Output: R1 = Address of last byte moved+1
R2 = Address of last byte stored+1

R0,R4 preserved, R3,R4 destroyed

Conditionals: None

C.4.32 MRKDL

This module contains the user interface for deleting a file without removing the directory entry.

Entry: .MRKDL

Input: R0 = FDB address

Output: C = 0, delete successful
= 1, delete failed, F.ERR(R0) set

All registers preserved

Conditionals: None

C.4.33 OPEN

This module contains the code for accessing a file. It can be assembled into three different versions: normal open

(R$$OPF=0), open by filename block (R$$OPF=1), and open by file-ID (R$$OPF=2). See the RSX-11/IAS I/O Operations Reference Manual, pages 3-2 to 3-17 for further descriptions of the calling sequence and outputs.

Entry: .OPEN, .OPFNB, .OPFID

Input: R0 = FDB address

Output: C = 0, operation successful
= 1, operation failed, F.ERR(R0) set

All registers preserved

Conditionals: R$$ANI,R$$BBF,R$$DPB,R$$EIS
R$$MBF,R$$NAM,R$$OPF,R$$RSL,R$$11M

C.4.34 PARDI

This routine continues the directory parsing process for the default directory. If the device and unit agree with the current saved directory-ID's device, the saved default directory-ID is copied. Otherwise, the directory-ID is obtained and saved for later use.

Entry: ..PDI

Input: R0 = FDB address
R1 = FNB address

0(SP) = Directory lookup routine (..DIDF/..DIRF)
2(SP) = Return address

Output: C = 0, operation successful
= 1, operation failed, F.ERR(R0) set

All registers preserved

Conditionals: R$$3CS

C.4.35 PARDID

This routine is a short-hand version of the directory lookup process, specifically designed for default directories. It sets up to call ..DIDF for directory processing and continues

in ..PDI.

Entry: ..PDID

Input: R0 = FDB address
R1 = FNB address

Output: See ..PDI

All registers preserved

Conditionals: None

C.4.36 PARSDI

This module contains the logic for parsing the directory specification.

Entry: .PRSDI, ..PSDI

Input: R0 = FDB address
R1 = FNB address
R2 = DSD address or zero if none
R3 = Default FNB address or zero if none

Output: C = 0, operation successful
= 1, operation failed, F.ERR(R0) set

All registers preserved

Conditionals: None

Directory processing in FCS appears somewhat complex the first time it is examined. The following routines and entries are involved:

```
DIDFND  ..DIDF
DIFND   ..DID1 ..DID
DIRFND  ..DIRF
GETDI   ..GTDI
GETDID  .GTDID
GETDIR  .GTDIR
PARDI   ..PDI
PARDID  ..PDID
PARSDI  .PRSDI ..PSDI
```

For the three user-level entries (.GTDID,.GTDIR, and .PRSDI) the following flow occurs. Note, for .PRSDI, there are three flows:

dataset(1), default-FNB(2), and neither(3):

.GTDID	.GTDIR	(1)	(2)	(3)
..GTDI	..GTDI	..DIRF	none	..PDI
..PDID	..DIRF	..DID		..DIRF
..PDI	..DID			..DID
..DIDF				

C.4.37 PARSDV

This module contains the code to parse the device name and assign the FDB's LUN to the parsed device.

Entry: .PRSDV, ..PSDV

Input: R0 = FDB address
R1 = FNB address
R2 = DSD address or zero if none
R3 = Default FNB address or zero if none

Output: C = 0, operation successful
= 1, operation failed, F.ERR(R0) set

If .PRSDV, all registers preserved

If ..PSDV, R0-R3 preserved, R4,R5 destroyed

Conditionals: R$$ELP

C.4.38 PARSE

This module contains the routines to completely parse a file specification. The individual parsing modules (PARSDI, PARSDV, and PARSFN) are called for each component of a specification.

The routine ..STFN is an internal entry used to parse the filename if no file-ID has been set. If parsing is needed, it falls into ..PARS. Otherwise the routine merely exits.

Entry: ..STFN

Input: R0 = FDB address

Output: R1 = FNB address

R0 preserved, R2-R5 destroyed

The routines .PARSE and ..PARS parse a complete specification.

Entry: .PARSE, ..PARS

Input: R0 = FDB address
R1 = FNB address
R2 = DSD address or zero if none
R3 = Default FNB address or zero if none

Output: C = 0, operation successful
= 1, operation failed, F.ERR(R0) set

If .PARSE, all registers preserved

If ..PARS, R0-R3 preserved, R4,R5 destroyed

Conditionals: R$$ELP,R$$SPL

C.4.39 PARSFN

This module contains the code to parse the filename.

Entry: .PRSFN, ..PSFN

Input: R0 = FDB address
R1 = FNB address
R2 = DSD address or zero if none
R3 = Default FNB address or zero if none

Output: C = 0, operation successful
= 1, operation failed, F.ERR(R0) set

If .PRSFN, all registers preserved

If ..PSFN, R0-R3 preserved, R4,R5 destroyed

Conditionals: None

C.4.40 PGCR

This routine checks to see if the FDB is properly setup for record input and output. The file is checked to see if it has

been opened and the proper I/O mode is checked (sequential/random/block). The module is assembled into two versions depending on the definition of R$$SEQ.

The routine ..PGCR checks for proper random or sequential I/O. If random I/O, it checks that the device is not sequential in nature. The routine ..PGCS checks for sequential I/O only. The calling arguments to both are the same.

Entry: ..PGCR, ..PGCS

Input: R0 = FDB address

Output: C = 0, operation OK
= 1, operation illegal, F.ERR(R0) set

All registers preserved

Conditionals: R$$SEQ

C.4.41 PNTMRK

This module contains the user routines for returning to a previously noted position in a file and for getting the current position in the file. A file's position is designated by a double-word block number and a single word byte within the block.

The .POINT routine positions a file to a previous marked position.

Entry: .POINT

Input: R0 = FDB address
R1 = New VBN number (high part)
R2 = New VBN number (low part)
R3 = Byte number within block

Output: C = 0, operation successful
= 1, operation failed, F.ERR(R0) set

All registers preserved

The .MARK routine retrieves the current record position. The virtual block number is taken from the F.VBN doubleword. The byte number is calculated from the current position in the block I/O buffer (F.NREC+F.VBSZ-F.EOBB).

Entry: .MARK

Input: R0 = FDB address

Output: R1 = VBN number (high part)
R2 = VBN number (low part)
R3 = Byte number within block

R0,R4,R5 preserved

Conditionals: None

C.4.42 POINT

This module contains the internal code for repositioning a file. If the desired block is different from the current block, the old block is written if dirty and the new block read. The record pointers are then set up for the desired position.

Entry: ..PNT1

Input: R0 = FDB address
R1 = New VBN number (high part)
R2 = New VBN number (low part)
R3 = Byte number within block

Output: C = 0, operation successful
= 1, operation failed, F.ERR(R0) set

R0 preserved, R1-R5 destroyed

Conditionals: R$$ANI,R$$BBF

C.4.43 POSIT

This module contains the code for calculating the positioning information necessary for use with .POINT for a file with fixed-length records.

Entry: .POSIT, ..PSIT

Input: R0 = FDB address

F.RCNM doubleword set to desired record number

Output: C = 0, operation success
= 1, operation failed, F.ERR(R0) set

If successful, following registers returned:

R1 = VBN number (high part)
R2 = VBN number (low part)
R3 = Byte number within block

If .POSIT, R0,R4,R5 preserved

If ..PSIT, R0 preserved, R4,R5 destroyed

Conditionals: R$$EIS

C.4.44 POSREC

This module contains the routines used to position a fixed-length file to a specific record. Besides the entry documented below (.POSRC), the entries ..PSR1, ..PSRC, and ..PSRG are also in this module. However, they are merely call other routines are are therefore are not documented further.

Entry: .POSRC

Input: R0 = FDB address

F.RCNM doubleword set to desired record

Output: C = 0, operation successful
= 1, operation failed, F.ERR(R0) set

All registers preserved

Conditionals: None

C.4.45 PPNASC

This routine translates a binary UIC into its ASCII string representation.

Entry: .PPASC

Input: R2 = String address to store ASCII UIC
R3 = Binary UIC (.BYTE owner,group)

R4 = Conversion Flags

Bit 0 = 0, suppress leading zeros
= 1, output leading zeros
Bit 1 = 0, output separators ([,])
= 1, suppress separator output

Output: String output

R2 = Last byte output+1

R0,R1,R3-R5 preserved

Conditionals: R$$EIS

C.4.46 PPNR50

This routine translates a binary UIC into two RAD50 words.

Entry: .PPR50

Input: R1 = Doubleword address to store RAD50 results
R2 = Binary UIC (.BYTE owner,group)

Output: UIC converted to RAD50

R0,R4,R5 preserved, R1-R3 destroyed

Conditionals: None

C.4.47 PUT

This module contains the code for outputting a logical record to the device/file. The module can be assembled for either sequential only support (R$$SEQ) or random/sequential support. See the RSX-11/IAS I/O Operations Reference Manual, pages 3-23 to 3-28 for further details on calling sequence and return values.

Entry: .PUT, .PUTSQ

Input: R0 = FDB address

Output: C = 0, record input
= 1, operation failed, F.ERR(R0) set

All registers preserved

Conditionals: R$$ANI,R$$EIS,R$$RSL,R$$SEQ,R$$11M

C.4.48 RDRN

This routine inputs the next virtual block, writing the current one if it is dirty (FD.WRT=1).

Entry: ..RDRN

Input: R0 = FDB address

Output: C = 0, operation successful
= 1, operation failed, F.ERR(R0) set

R0 preserved, R1-R5 destroyed

Conditionals: None

C.4.49 RDWAIT

This module contains the routines for reading the next virtual block. ..RWAT increments the block number (F.VBN) and falls into ..RWAC. It also checks for record-oriented transfers and handles them correctly.

Entry: ..RWAT

Input: R0 = FDB address

See ..RWAC for remaining description

The routine ..RWAC reads the current virtual block. It only expects to be called for block I/O.

Entry: ..RWAC

Input: R0 = FDB address

F.VBN(R0) = Block number
F.BDB(R0) = Address of buffer header
F.BBFS(R0) = Number of bytes to read

Output: C = 0, operation successful
= 1, operation failed, F.ERR(R0) set

F.NREC(R0) = Address of start of block
F.EOBB(R0) = Address of last byte+1

R0 preserved, R1-R5 destroyed

Conditionals: R$$BBF,R$$MBF

C.4.50 RDWRIT

This module contains the common routines for READ/WRITE I/O. The routine ..RWCK checks that block I/O is allowed. It checks that the file is opened (F.DBD nonzero), READ/WRITE mode is set (FD.RWM=1), and the device is block oriented (FD.REC=0).

Entry: ..RWCK

Input: R0 = FDB address

Output: C = 0, Block I/O allowed
= 1, Block I/O not allowed, F.ERR(R0) = IE.ILL

All registers preserved

The routine ..WTRD actually issues the block I/O requests. The block number is bumped after the I/O completes.

Entry: ..WTRD

Input: R0 = FDB address
R4 = I/O function code (IO.WVB/IO.RVB)

F.VBN(R0) = Block number to read/write
F.BKDS(R0) = Buffer descriptor
F.BKST(R0) = I/O status block address
F.BKDN(R0) = I/O done AST address

Output: C = 0, operation successful
= 1, operation failed, F.ERR(R0) set

R0 preserved, R1-R5 destroyed

Conditionals: None

C.4.51 READ

This routine is the user interface for reading a block via block I/O mode. See the RSX-11/IAS I/O Operations Reference Manual, pages 3-28 to 3-31 for further details on calling sequence and return values.

Entry: .READ

Input: R0 = FDB address

Output: C = 0, operation successful
= 1, operation failed, F.ERR(R0) set

All registers preserved

Conditionals: None

C.4.52 RENAME

This routine is the user interface for renaming a file. Only the file's directory entry is manipulated, the filename internal to the file header is untouched.

Entry: .RENAM

Input: R0 = FDB address (old filename)
R1 = FDB address (new filename)

Output: C = 0, operation successful
= 1, operation failed, F.ERR(R0) set

All registers preserved

Conditionals: None

C.4.53 RETADR

This routine correctly sets up the FDB record pointers for locate mode I/O. If called for move mode, it is a no-op.

Entry: ..RTAD

Input: R0 = FDB address

Output: F.NRBD(R0) = Set to record size, address

R0,R1,R3-R5 preserved, R2 destroyed

Conditionals: R$$ANI,R$$BBF

C.4.54 RSTFDB

This routine resets a FDB so it can be used for another open. If record I/O was used, the block buffer is returned to the pool. Locations in the FDB that are assumed to be zero if no file is opened are cleared.

Entry: ..RFDB

Input: R0 = FDB address

Output: FDB reset to non-opened state

R0 preserved, R1-R5 destroyed

Conditionals: R$$MBF

C.4.55 RWBLK

This module contains the routines used to issue the read/write requests for record I/O. The routines are used for both record and block oriented devices. The only difference between the entries is the I/O codes used. ..RBLK uses IO.RVB. ..WBLK uses IO.WVB.

The carriage control and VBN are always stored in the constructed QIO. When issued to record devices, QIO parameters 4 and 5 are typically ignored. Similarly, block I/O devices ignore parameter 3. The I/O status block is assumed to be at the beginning of the buffer header, which is assumed to be immediately in front of the data buffer.

No implicit wait for I/O completion is made. The carry bit will be set only if the QIO directive fails.

Entry: ..RBLK, ..WBLK

Input: R0 = FDB address
R1 = Address of data buffer
R2 = Size of data buffer
R3 = Carriage control character

F.VBN(R0) = Block number

Output: R1 = Address of I/O status block

R0 preserved, R2-R5 destroyed

Conditionals: R$$MBF

C.4.56 RWFSR2

This module contains a collection of routines to allow the user to read and write fields in $$FSR2. The routines (.RDFDR, .WDFDR, .RDFFP, .WDFFP, .RFOWN, .WFOWN, .RDFUI, and .WDFUI) are described in the RSX-11/IAS I/O Operations Reference Manual, pages 4-2 to 4-6.

C.4.57 RWLONG

This routine is used to perform block I/O transfers when the byte count is greater than one block (512 bytes). This routine is selected by the .READ/.WRITE routines when appropriate. Otherwise, ..WTRD is used.

Entry: ..RWLG

Input: R0 = FDB address
R4 = I/O function code (IO.RVB/IO.WVB)

FDB setup for block I/O request

Output: C = 0, operation successful
= 1, operation failed, F.ERR(R0) set

All registers preserved

Conditionals: None

C.4.58 TRNCLS

This routine truncates a file to its EOF position and closes the file.

Entry: .TRNCL

Input: R0 = FDB address

F.EFBK, F.FFBY set to EOF position

Output: C = 0, operation successful
= 1, operation failed, F.ERR(R0) set

All registers preserved

Conditionals: None

C.4.59 UDIREC

This module contains the user interface routines for issueing the directory primitive functions: find filename (.FIND), enter filename (.ENTER), and remove filename (.REMOV). The calling sequences and return values are documented fully in the RSX-11/IAS I/O Operations Reference Manual, pages 4-12 to 4-14.

Entry: .FIND, .ENTER, .REMOV

Input: R0 = FDB address
R1 = FNB address

Output: C = 0, operation successful
= 1, operation failed, F.ERR(R0) set

All registers preserved

Conditionals: R$$NAM

C.4.60 UPWARD

This routine provides extended file lookup abilities for SCS-11 systems. It is not compiled for RSX-11M systems

C.4.61 WAITI

This module contains routines used to issue QIO's for FCS and wait for their completion. The first routine, ..QIOW, issues a QIO and falls into ..WAIT. It watches for errors due to insufficient pool and will loop, waiting for a significant event if this happens.

Entry: ..QIOW

Input: R0 = FDB address
R4 = I/O function code

Scratch DPB setup in $$FSR2

Output: C = 0, operation successful
= 1, operation failed, F.ERR(R0) set

R1 = I/O status block address

R0 preserved, R2-R5 destroyed

The routine ..WAIT stores the I/O status into F.ERR(R0). It waits for I/O completion by waiting for a non-zero I/O status value. If the I/O status is zero, it falls into ..WAEF and then repeats.

Entry: ..WAIT

Input: R0 = FDB address
R1 = I/O status block address

Output: C = 0, I/O completed successfully
= 1, I/O failed, F.ERR(R0) set from status block

All registers preserved

The routine ..WAEF waits for the FDB's event flag, F.EFN(R0). If no event flags was specified, event flag 32(10) is used. The flag is cleared after the wait completes.

Entry: ..WAEF

Input: R0 = FDB address

Output: Event flag cleared

All registers preserved

Conditionals: None

C.4.62 WAITU

This routine is the user interface for waiting for I/O completion when performing block I/O. If an I/O status block has been specified (F.BKST nonzero), the routine ..WAIT is used. Otherwise, the routine ..WAEF is called.

Entry: .WAIT

Input: R0 = FDB address

Output: I/O wait completed

All registers preserved

Conditionals: None

C.4.63 WATNOD

This routine checks the I/O status for errors due to insufficient pool (IE.UPN) and doers a wait-for-significant event if true. Otherwise it merely returns.

Entry: ..WAND

Input: R0 = FDB address
R1 = I/O status block address

Output: C = 0, IE.UPN occured, wait completed
= 1, Error was not IE.UPN

All registers preserved

Conditionals: None

C.4.64 WATSET

This routine waits for I/O completion and sets up the record pointers based on the contents the the second I/O status word.

Entry: ..WAST

Input: R0 = FDB address
R1 = I/O status block address

I/O status is assumed to be at start of buffer descriptor.

Output: C = 0, I/O completed successfully
= 1, I/O failed, F.ERR(R0) set

F.NREC(R0) set to beginning of data
F.EOBB(R0) set to end of data

R0,R1,R3-R5 preserved, R2 destroyed

Conditionals: R$$BBF

C.4.65 WRITE

This routine is the user interface for writing a block via block I/O mode. See the RSX-11/IAS I/O Operations Reference Manual, pages 3-31, 3-32 for further details on calling sequence and return values.

Entry: .WRITE

Input: R0 = FDB address

Output: C = 0, operation successful
= 1, operation failed, F.ERR(R0) set

All registers preserved

Conditionals: None

C.4.66 WTWAIT

This module contains the routines for writing the next virtual block. ..WTWA outputs the VBN and returns with the record buffers set up for the next PUT.

Entry: ..WTWA

Input: R0 = FDB address

```
                F.VBN(R0)  = Block number
                F.BDB(R0)  = Address of buffer header
                F.BBFS(R0) = Number of bytes to read

        Output: C = 0, operation successful
                  = 1, operation failed, F.ERR(R0) set

                F.NREC(R0) = Address of start of block
                F.EOBB(R0) = Address of last byte+1

                R0 preserved, R1-R5 destroyed

        Conditionals: R$$BBF,R$$MBF
```

C.4.67 XQIOI

This module contains the routines used to build and issue internal FCS QIO's. They use the QIO DPB in $$FSR2. The routine ..XQIO sets up the standard parameters and issues the QIO directive. It assumes the I/O parameters have already been set. This routine gets the I/O status block address and AST address from the FDB.

```
        Entry:  ..XQIO

        Input:  R0 = FDB address
                R3 = Directive code, size
                R4 = I/O function code
                R5 = DPB address

        Output: See ..XQI1 below
```

An alternate entry is ..XQI1. This routine is passed the I/O status block address and AST address. It gets the lun and event flag from the FDB, sets up the DPB, and issues the I/O request.

```
        Entry:  ..XQI1

        Input:  R0 = FDB address
                R1 = I/O status block address or zero
                R2 = I/O Done AST address or zero
                R3 = Directive code, size
                R4 = I/O function code
                R5 = DPB address

        Output: C = 0, operation successful
                  = 1, QIO directive failed, F.ERR(R0) set from
```

$DSW

R1 = I/O status block address

R0,R4 preserved, R2,R3,R5 destroyed

The routine ..IDPB initializes the DPB in $$FSR2 and returns the address of the parameter area.

Entry: ..IDPB

Input: None

Output: R5 = Address of parameter area

R0-R4 preserved

Conditionals: None

C.4.68 XQIOU

This routine is the user interface for issuing a QIO request using the lun and event flag from the FDB.

Entry: .XQIO

Input: R0 = FDB address
R1 = I/O function code
R2 = Size of parameter list or zero if none
R3 = Address of parameter list

Output: C = 0, operation successful
= 1, operation failed, F.ERR(R0) set

All registers preserved

Conditionals: None

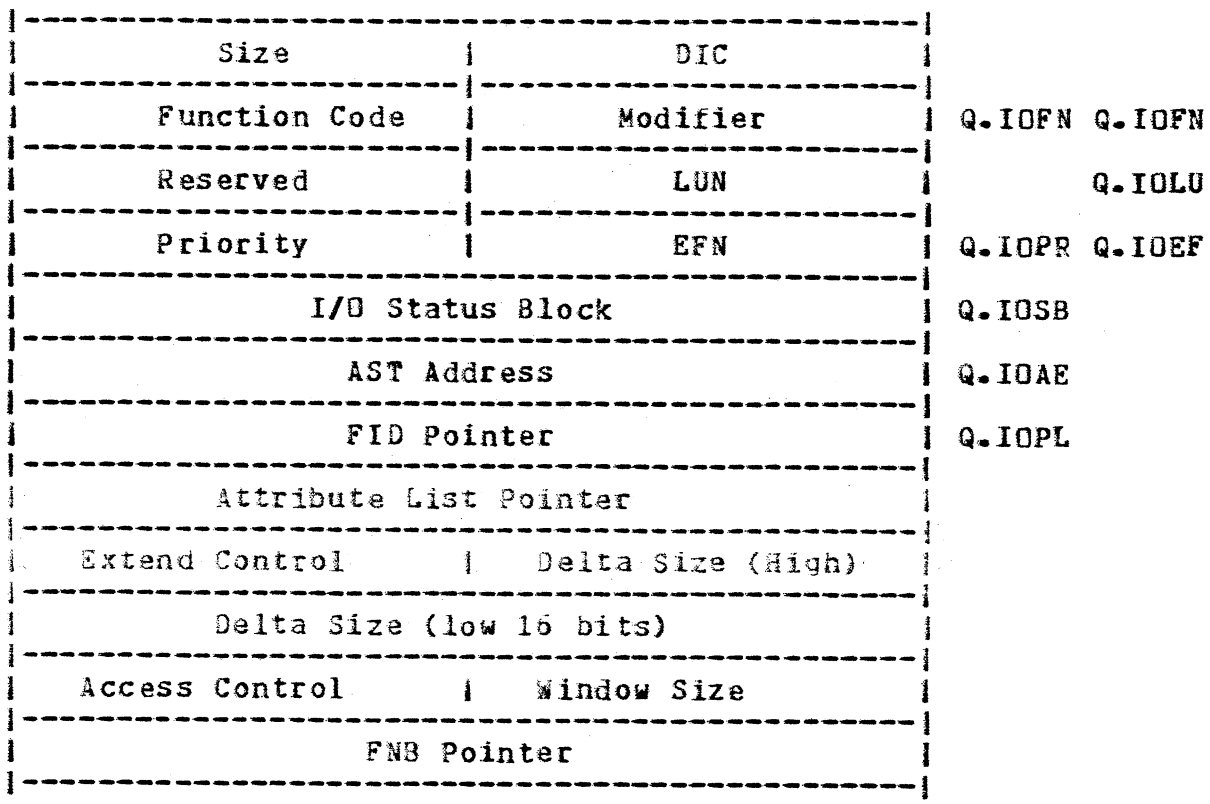
APPENDIX D

FILES-11 QIO'S

This section documents the various QIO's used by Files-11 (F11ACP and MTAACP)). The material is taken from an article written by Andrew Goldstein of Digital in November, 1976 and from examination of various source modules.

D.1 FILES-11 QIO DPB

All F11ACP QIO directive parameter blocks have the same format. The following diagram illustrates this format (note only the parameter fields are different from all other QIO's):



Each parameter field is used for a specific purpose. If the particular QIO does not require the field, it must not be specified (set to zero). F11ACP range checks all parameter fields.

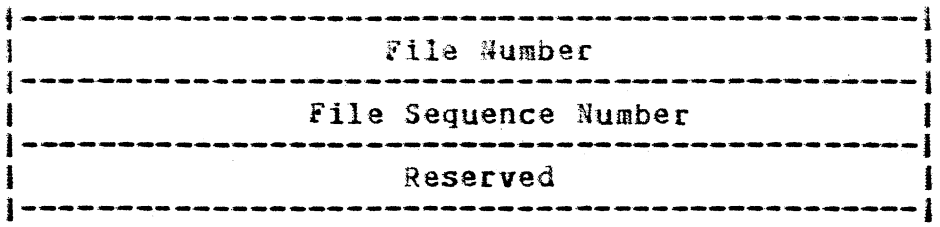
D.2 FILES-11 QIO PARAMETERS

This section will discuss the six parameter words in a Files-11 QIO. While none of the parameters require the FCS data structures, it is obvious tne fields are set up for use in the FCS environment.

D.2.1 Parameter Word #1 - FID Pointer

This word contains the address of a three word block in the issuing task's space. This block is or will become the file ID. If the word is zero, no file ID is specified.

In the FCS environment, this word will typically contain the address of the filename block. The file ID block has the following format:



The file number is used by F11ACP as an index to the file header block in the index file. The file sequence number is used to maintain header integrity. Each time a header block is used for a new file, the file sequence number is incremented. The final word has no current meaning.

D.2.2 Parameter Word #2 - Attribute List Pointer

This word contains the address of an attribute list in the issuing task's space. This list controls which file attributes are to be read or written by F11ACP. If no attribute list is specified, the word is zero.

File attributes are various fields in the file header.

These fields are documented in Appendix F of the IAS/RSX-11 I/O Operations Reference Manual (AA-2515C-TC).

An attribute list consists of zero to six attribute entries, followed by a byte of zero. Each attribute entry has the following format:

```
.BYTE   <Attribute type>,<N>
.WORD   <Pointer to 'N' byte buffer>
```

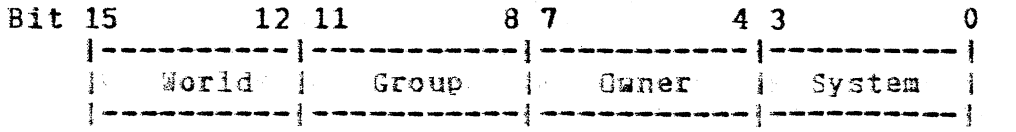
The sign of the attribute type determines the direction of the operation. If the attribute type is negative, the attribute is read from the file header to the buffer. If the attribute type is positive, the buffer is written to the file header as the new attribute. The magnitude of the attribute type and size of the buffer determine which fields in the file header will be accessed. The following table lists all valid read attribute types, valid buffer sizes, and the starting offset in the file header. To write the attribute, make the sign of the attribute type positive.

-01,02 Read file owner UIC (H.FOWN). The UIC is a binary word. The low byte (H.PROG) is the owner number. The high byte (H.PROJ) is the group number. Note that the file owner UIC is independent of the directory UIC.

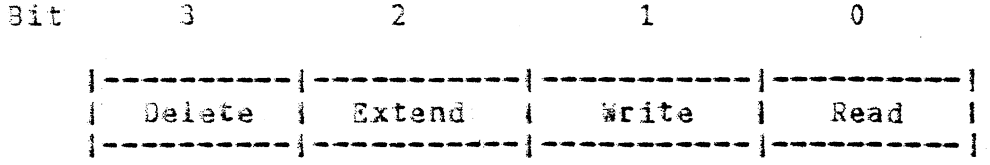
-01,04 Read file owner UIC, protection (H.FOWN). The UIC is returned as described above. The second word is set to the file protection code (see attribute -02,02).

-01,05 Read file owner UIC, protection, characteristics (H.FOWN). The UIC and protection are returned as described above. The fifth byte is set to the user-controlled characterics (see attribute -03,01).

-02,02 Read protection (H.FPRO). The file protection word is a bit mask with the following format:



Each of the four categories above has four bits. Each bit has the following meaning with respect to file access:



A bit value of zero (0) indicates the respective type of access is allowed to the file. A bit value of one (1) indicates access is denied.

-02,03 Read protection, characteristics (H.FPRO). The protection is returned as described above. The third byte is set to the user-controlled characteristics (see attribute -03,01).

-03,01 Read characteristics (H.UCHA). The user characteristics is a one byte field containing various bit definitions. The current bits defined are listed below:

UC.CON = 200 Logically continuous file. When the file is extended, this bit is cleared.

UC.DLK = 100 File improperly closed. When ever the file is opened for write, this bit is set. It is not cleared until the file is closed (deaccessed). This is the famous lock bit.

In addition to the user-controlled characteristics, the next byte in the header is the system-controlled characteristics. This byte cannot be accessed by an attribute field. The current bits defined in this byte are listed below:

SC.MDL = 200 File marked for delete.

SC.BAD = 100 Bad data block in file.

-04,40 Read record I/O area (U.UFAT). The first 7 words of this area are a direct copy of the first 7 words of the FDB when the file is opened (see Table A-1, I/O Operations Reference Manual, offsets F.RTYP to F.FFBY). The remaining 9 words of this area are not used. I do not know how this area is defined in the case of RMS-11.

-05,06 Read filename (I.FNAM). The filename is stored as nine (9) RAD50 characters.

-05,10 Read filename, type (I.FNAM). The filename is returned as described above. The type is returned

to the fourth word (see attribute -06,02).

-05,12 Read filename, type, version (I.FNAM). The filename and type are returned as described above. The version is returned to the fifth word (see attribute -07,02).

-06,02 Read type (I.FTYP). The type is stored as three (3) RAD50 characters.

-06,04 Read type, version (I.FTYP). The type is returned as decsribed above. The version is returned to the second word (see attribute -07,02).

-07,02 Read version (I.FVER). The version is stored as a binary number.

NOTE

The filename, type, and version are set when the file is created. If the file is renamed by PIP, these fields are not changed.

-10,07 Read expiration date (I.EXDT). The expiration date is intended to be the time the file becomes eligible for deletion. This feature is not implemented. The date is kept in ASCII form in the format day, month, and year (2 bytes, 3 bytes, and 2 bytes).

-11,12 Read statistics block. The statistics block is defined in Appendix H of the I/O reference manual. No specific fields exist in the file header for this attribute. Therefore, it cannot be written.

-12,00 Read entire file header. The buffer size is assumed to be 1000(8) bytes. This attribute has no corresponding write function.

-13,02 Read block size (ANSI labelled tape only). The block size is returned as a positive 16-bit number

-14,xx Read user label (ANSI labelled tape only). This attribute allows access to the user label on an ANSI standard tape. "xx" is the length of the label (maximum 80). If the function is a read, user header labels are read if a file is accessed. If no file is accessed, user trailer labels are read. If the function is a write, user header

labels are written during a create. User trailer labels are written during a deaccess.

-15,xx Read complete date information (disk files only). This attribute allows the revison, creation, and expiration dates to be read. Dates are stored and returned in the format day (2 bytes), month (3 bytes), and year since 1900 (2 bytes). Times are stored and returned in the format hours (2 bytes), minutes (2 bytes), and seconds (2 bytes). "xx" bytes of time/day information are returned in the following format:

00-01 Revision number. This number is incremented each time the file is closed after being opened for output.

02-10 Revision date.

11-16 Revison time.

17-25 Creation date.

26-33 Creation time.

33-42 Expiration date.

+16,16 Allocation control (disk files only). Used for file placement control, currently by RMS only. Processed only by create or write (i.e., write attribute only).

The magitude of the attribute type determines the maximun valid buffer size. Any smaller size is legal. The sizes listed above are sufficient to handle the named attributes. The largest size for each attribute is also the largest buffer allowed.

D.2.3 Parameter Words #3 and #4 - Size/Extend Control

These two parameters are used to specify how many blocks are to be allocated to a new file or added to an existing file. The words are also used to control the type of block allocation. The format of the two parameters is as follows:

```
.BYTE   <Delta size (high 8 bits)>,<Extend control>
.WORD   <Delta size (low 16 bits)>
```

The high bit of parameter word #3 (bit 15) controls whether the remaining fields are used by F11ACP. If the bit is zero

(0), no size change is desired. If the bit is one (1), the remaining fields are processed.

The high byte of parameter word #3 (extend control) determines the type of allocation desired. The low byte of this word and parameter word #4 form a 24-bit number of blocks to allocate (delta size). This number is the initial file size in the case of a create and the change in size in the case of an extend.

The extend control byte consists of a bit mask. The bits have the following meanings:

Bit 0 All blocks must be allocated contigously (EX.AC1).

Bit 1 Allocate largest contigious chunk up to delta size (EX.AC2). This bit is not examined unless bit 0 is on (1).

Bit 2 File must end up contigious at operations end (EX.FCO).

Bit 3 Use volume default as delta size (EX.ADF).

Bit 4 Placement control is desired (EX.ALL).

Bit 5 Unused.

Bit 6 Unused.

Bit7 Enable extend (EX.ENA) (see above).

D.2.4 Parameter Word #5 - Window Size/Access Control

This word is used to specify the window size (low byte) and the access control (high byte). The word is processed if the high bit (bit 15) is one (1). If this bit is zero (0), the word is disabled. The format of the word is shown below:

```
.BYTE   <Window size>,<Access control>
```

The window size is the number of mapping entries the window is to hold at once. This is not the window size in bytes. If the byte is zero, the volume default is used.

The access control byte consists of a bit mask. The bits have the following meanings:

Bit 0 Lock file from further accesses for write and/or extend (AC.LCK).

Bit 1 Enable deaccess lock (AC.DLK).

BIT 2 Enable block locking (AC.LKL).

Bit 3 Enable explicit block unlocking (AC.EXL).

Bit 4 Unused

Bit 5 Unused.

Bit 6 Unused.

Bit 7 Enable access (AC.ENB) (see above).

D.2.5 Parameter Word #6 - Filename Block Pointer

This word contains the address of a 13(10) word block in the issuing task's space. This block is the filename block. See Appendix B in the I/O Reference Manual for a description of a filename block. If the word is zero, no filename block is specified.

If the QIO function is a directory operation (IO.FNA, IO.RNA, IO.ENA), the directory ID field (N.DID) of the filename block is used. Files-11 directories are merely files with names of the form [0,0]gggooo.DIR;1 where [0,0] refers to master file directory and "ggg" is the group number and "ooo" is the owner number. The master file directory is the file with ID 4,4,0 and is also entered in the master file directory. The directory for UFD [123,456] is the file [0,0]123456.DIR;1

D.3 FILES-11 QIO FUNCTIONS

This section will discuss the various functions processed by F11ACP. This functions are normally issued by FCS or RMS, however, the clever user can use them directly and save the overhead of the run-time systems.

The following are the functions implemented by F11ACP. Each function is followed by a list of required and optional parameters. If a parameter is not listed, it must be set to zero. Also, parameter 4 is not shown as it is a part of parameter 3.

IO.CRE Create File

#1 - FID block pointer, FID value returned with ID

of file created.
#2 - Write attribute control list (optional).
#3 - Size/extend control (optional).
#5 - Access control (may be non-zero but must be disabled).

IO.DEL Delete or Truncate File

#1 - FID block pointer (optional if file is accessed).
#3 - Size/extend control. If not present or enabled, file is deleted. Otherwise, the remaining 31 bits specify the size the file is to be after truncation.

IO.ACR Access File for Read Only
IO.ACW Access File for Read/Write
IO.ACE Access File for Read/Write/Extend

#1 - File ID pointer.
#2 - Read attributes control list (optional).
#5 - Access control.

IO.DAC Deaccess File

#1 - File ID pointer (optional).
#2 - Write attribute control list (optional).
#5 - Access control (may be non-zero but must be disabled).

IO.EXT Extend File

#1 - File ID pointer (optional if file aready accessed).
#3 - Size/extend control.

IO.RAT Read Attributes

#1 - File ID pointer (optional if file aready accessed).
#2 - Read attribute control list.

IO.WAT Write Attributes

#1 - File ID pointer (optional if file aready accessed).
#2 - Write attribute control list.

IO.FNA Find Filename in Directory
IO.RNA Remove Name from Directory
IO.ENA Enter Name into Directory

#5 - Access control (may be non-zero, but must be

disabled).
#6 - Filename block pointer.

The following table summarizes the optional and required parameters for each I/O request. The key to the table is as follows:

* = Required parameter.
O = Optional parameter.
A = Optional if file already accessed.
D = May be non-zero, but must be disabled (bit 15 = 0).

If the entry is blank, the parameter must be zero or the I/O will be in error. Parmeter 4 is a part of parameter 3 and is not listed in the table.

Function	P 1	P 2	P 3	P 5	P 6
IO.ACE	*	O		*	
IO.ACR	*	O		*	
IO.ACW	*	O		*	
IO.CRE	*	O	O	D	
IO.DAC	O	O		D	
IO.DEL	A		O		
IO.ENA				D	*
IO.EXT	A		*		
IO.FNA				D	*
IO.RAT	A	*			
IO.RNA				D	*
IO.WAT	A	*			

D.4 PLACEMENT CONTROL

One undocumented feature of Files-11 is placement control. This feature allows the create or extend functions to specify an exact or approxomate position to allocate the desired blocks.

Placement control is implemented by an attribute list entry. The placement attribute is valid for either the IO.CRE or IO.EXT functions and if used, must be the first attribute in the attribute list. The format of the placement control attribute block is as follows:

```
.BYTE   <Placement control>,0
.WORD   <High order bits of VBN or LBN>
.WORD   <Low  order bits of VBN or LVB>
.BLKW   4       (Optional, see below)
```

The placement control byte consists of a bit mask. The bits have the following meanings:

Bit 0 Set if block specified is VBN, otherwise LBN is assumed (AL.VBN).

Bit 1 Set if approximate placement is desired, otherwise, exact placement is assume (AL.APX).

BIT 2 Set if starting and ending LBN information is desired (AL.LBN).

Bit 3 Unused.

Bit 4 Unused

Bit 5 Unused.

Bit 6 Unused.

Bit 7 Unused.

If AL.LBN is set, the control block must be 16(8) bytes long. F11ACP will return the starting LBN in the first two optional words and the last LBN in the last two optional words. Otherwise, the attribute size must be 6 bytes. If AL.VBN is specified, AL.APX must also be set and the attribute will allocate the new blocks as close to the specified block as possible. This is useful if file extensions are desired to be as close to the previous as possible.

D.5 BLOCK LOCKING

Another undocumented feature of Files-11 is block-locking support. This feature was implemented to support RMS, however, it can be used for FCS. Locking only occurs for access to shared files and AC.LKL set in the access control byte. For FCS users, this can be done by setting the FA.LKL bit in the F.ACTL field of the FDB before opening a file shared. From then on,

whenever a user reads or writes a block, the system will insure no other user can have accesses to it. If the block is locked, the second access will have a lock error (IE.LCK) returned to it. This feature, when properly used, allows multiple readers and writers to the same file.

There are two options for unlocking blocks. The simplest method is to unlock the block whenever a new block is read or written. This will occur when the AC.EXL bit is set in the access control byte. FCS users can enabled this feature by setting the bit in the F.ACTL byte of the FDB.

If the application requires multiple block locks, an unlock QIO is used to unlock blocks. To unlock a block, the IO.ULK function is issued with the following parameters.

Parameter #1 - Always zero.

Parameter #2 - Zero or number of blocks to unlock.

Parameter #3 - Always zero.

Parameter #4 - High part of VBN number or zero.

Parameter #5 - Low part of VBN number or zero.

Parameter #6 - Always zero.

If all the specified blocks are currently unlocked, an error will be returned (IE.ULK). If no VBN and count is specified, all currently locked blocks are unlocked if all I/O is completed. If a user is waiting for I/O, this could leave some blocks locked because the I/O was not finished. If a VBN and not count is specified, the first lock-list entry with the specified VBN is unlocked. Therefore, if the user issued two requests for the same VBN with different sizes, the first one encountered will be unlocked. The order of locked blocks bears no relationship to the order the orignal reads/writes were issued. Finally, if a VBN and count are specified, an exact match is located.

APPENDIX E

SAMPLE ACP

Included with the manual distribution is a sample ACP and its associated device driver. The kit also includes sources for sample ACP enabling and disabling tasks.

E.1 SOURCES

The software implements a UCP-stype ACP. No system modifications are required to run the sample ACP. The file ACPGEN.CMD is an indirect command file that will assemble and taskbuild all sample software. The other files in the kit are as follows:

COMACP.MAC - Prefix file used to define special symbols used by the ACP and its device driver.

NULACP.MAC - Source for sample ACP. The ACP performs no useful operations. However, it does demonstrate packet dequeuing, device mounts and dismounts, I/O process creation and termination, and data transfers from an ACP to a user task.

NULDRV.MAC - Source for sample driver. The driver contains a loadable database supports two devices (AC0:, AC1:). The driver demonstrates how ACP packets are processed and queued to an ACP.

ENABLE.MAC - Source for basic NULACP mount task. This task performs the basic steps required for enabling an ACP. It supports switches to specify the ACP task name (/ACP=name) and volume control block size (/VCB=size). The /PRM="string" switch can be used to pass an ASCII string to the ACP.

DISABL.MAC - Source for basic NULACP dismount task. This task performs the basic steps required for disabling an ACP. It supports the /PRM="string" switch.

NULTST.FOR - Source for test package for sample software. This is a Fortran program that queries the user for the I/O function to issue and can be used to test the other software.

The kit also includes command files to link all tasks. The ENABLD.CMD file allows the default ACP task name and volume control block size to be set. The current defaults are "NULACP" and 2 bytes respectively.

E.2 PROCEDURE

The procedure for using the sample code is to execute the ACPGEN command file and generate the tasks. The listings should then be printed and read. The ACDRV should be loaded using the MCR LOA command and NULACP mounted using the ...ENA task. The NULTST task can then be used to issue I/O requests to the ACP and report success/failure. When finished, the mounted devices can be dismounted with the ...DIS task and the driver unloaded.

The sample ACP processes I/O packets by outputting an appropriate message to the console terminal and returning success to the user task. The software is intended to be used in conjuction with the manual, particularly, Chapter 3 which outlines the designed processed used in the development of the sample ACP. The sources can also be used as a starting point for a user-written ACP.

APPENDIX F

USER-WRITTEN ACP'S

This appendix describes various user-written ACP's that have been reported to the author. The author welcomes new additions. Please send a description of your ACP to the following address:

Ralph W. Stamerjohn
Monsanto, Zone T1A
800 N. Lindbergh
St. Louis, MO, 63166

The submission should be typed and follow the format used for the current submissions. Please limit the description to one page if possible.

F.1 DAPACP

Author: Ralph W. Stamerjohn Date: 17-APR-80

Address: Monsanto, Zone T1A
800 N. Lindbergh
St. Louis, MO, 63166

Phone: (314) 694-4252

DAPACP implements Digital's Data Access Protocol (DAP). The ACP is interfaced to FCS and allows RSX-11M utilities and languages to transparently access remote files on other RSX-11M and DECsystem-10 systems. For example, normal Fortran IV Plus I/O statements can be used to read and write remote files.

ACP-Style:	UCP, using pseudo driver as gateway
I/O Functions Codes:	Uses 31(10), subfunctions
Serial/Parallel:	Parallel
Transportability:	Requires memory management, EIS
Pool Utilization:	Minimumal, uses internal ACP pool
Mounting:	Uses alternate task
Dismounting:	Uses alternate task
FCS Support:	Yes
RMS Support:	No
Contact Author:	Yes
Availablity:	No, proprietary Monsanto software
References:	Reports of the DECsystem-10/20 Fall '79 U.S. DECUS Meeting, pages 57-65.