HYDRA USER'S MANUAL

(Preliminary Version)

Ellis Cohen (Editor)
Dave Jefferson
Tom Lane
Roy Levin
Fred Pollack
Bill Wulf

Dept Computer Science
Carnegie Mellon University

Nov 1974

This is a preliminary version of the Hydra Manual for friends. Please report any corrections or comments to Ellis Cohen [N810EC03]@CMUA

This work was supported by the Advanced Research Projects Agency of the Department of Defense under contract no. F44620-73-C-0074 and monitored by the Air Force Office of Scientific Research.

↑L

.SEC |INTRODUCTION|

This document is a user's manual for the HYDRA Kernel. A certain amount of tutorial material can be found in the manual. Readers with a sketchy background in protection are advised to first read the HYDRA article in the CACM.

We want to stress strongly that HYDRA is not by itself an Operating System in the usual sense, rather it augments the PDP-11 to provide a well-protected basis on which an Operating System can be built. Hence, HYDRA is known as the KERNEL of an Operating System. In fact, many different Operating Systems can be running on HYDRA simultaneously. A standard System is available and is the one that a user initially interacts with when she logs in. This standard system is described in a separate document.

HYDRA provides a software virtual machine implemented on C.mmp (Carnegie Multi-Mini Processor, though "C." actually stands for "Computer"), a network of PDP-11 processors. The virtual machine instructions are known as KALLs (Kernel cALLs). They are described in terms of a standard set of BLISS-11 Macros (available on HYKALL.R11[N810HY00]). Hence, no knowledge of the PDP-11 is necessary to understand much of the contents of this manual. The Appendix contains a listing of HYKALL.R11 as well as examples of the machine code calling sequence for various KALLs.

↑L

.SEC |THE BASIC KERNEL|

.SUBSEC |A CAPABILITY SYSTEM|

The HYDRA Kernel provides an execution environment in which protection plays a key part. In some systems, FILEs are the units of protection, in others, SEGMENTS. In HYDRA, the basis of protection is an entity called an OBJECT.

Many traditional operating systems are 'Access Control Systems'; that is, protection information is associated with the Object being protected. For example, in the PDP-10 TOPS Operating System, when an executing procedure tries to open a file (using an ASCII encoding of the file name), the access key associated with the file is checked.

HYDRA, on the other hand, is a 'Capability System'. As we noted, the basis of protection in HYDRA is an entity called an OBJECT, and the protection system is invoked to determine whether particular accesses to Objects will be allowed. In a Capability System, associated with each executing Procedure is a C-List, a list of Capabilities; each Capability contains the name of an Object and a set of Rights which determine how that Object may be accessed by the executing procedure.

Each different Object is assigned a unique name by the Kernel. Rather than showing 'real' unique names in diagrams, (represented internally by unique 64 bit combinations), we will instead substitute unique alphanumeric names for pictorial clarity.

In HYDRA, Objects are Typed. Examples of Types built into HYDRA (called Kernel Types) are PAGEs, DEVICEs and PROCESSes. There is also a facility to allow the creation of new user types. Certain types represent physical resources (e.g. Objects of Type DEVICE represent actual devices; one may represent a disk, another a line printer, etc.), but in general, Types represent abstractions of resources, both physical and virtual, and Objects of such a Type have meaning only in terms of their 'Representation' and how that representation is accessed and manipulated.

HYDRA is a paged system. When a procedure executes, its code (and directly accessible data) is contained in pages represented by PAGE Objects. Capabilities for these PAGE Objects must be in the C-List of the executing procedure. The Paging Section describes how to indicate to the Kernel which of these should be made directly addressable.

In HYDRA, an executing Procedure is a distinct type of Object, called an LNS (Local Name Space) and differs from the Type representing its static counterpart, a PROCEDURE. PROCESS Objects are the scheduling entities of the Kernel. Each running Process has an LNS associated with it which determines the 'Environment' in which the process runs. HYDRA provides a CALL Mechanism to change environments - by associating a different LNS with a process.

.SUBSEC |OBJECTS, CAPABILITIES AND PATHS|

Every type of Object has two parts,a C-List containing a list of Capabilities, and a Data-Part containing data. The C-List and Data-Part of an Object together comprise its 'Representation'.

Both the C-List and Data-Part are linearly ordered, based at 1. The maximum number of Capabilities in a C-List and the maximum length of a Data-Part varies from type to type. The Appendix contains those numbers for Kernel types. Since C-Lists are linearly ordered, we will often refer to a Capability as being in the k'th 'Slot' of a C-List.

As examples, consider the representation of some Kernel Objects: A PAGE Object contains an empty C-List and its Data-Part contains the location of the page (Disk, Drum or Core address) and its status. The Data-Part of a Device Object contains a code identifying the device. The Data-Part of an LNS contains (among other things) trap addresses, a mask of processors on which the LNS may execute, and paging information, while the C-List of the LNS contains the Capabilities which define the 'Environment' of the LNS.

There are facilities for creating new Types of Objects as well as for creating Objects of existing types and erasing them. For example, a user might create a new Type of Object, a FILE, whose C-List might contain Capabilities for PAGEs and whose Data-Part might contain information about the file (it could even be used to hold access keys as part of a system that could provide file access checking in a way similar to that of the PDP-10 TOPS monitor). Or a user might create a DIRECTORY Type. Objects of type DIRECTORY might have a C-List containing Capabilities for FILEs and other DIRECTORYs. This could be used to build up an hierarchical FILE system similar to the one in MULTICS.

C-Lists and Data-Parts can only be accessed and manipulated through the Kernel via KALLs. The Kernel provides some very basic Kalls that do the following kinds of things: Delete Capabilities from the C-List of some Object, Move a Capability from the C-List of one Object to the C-List of another Object (perhaps the same) (with or without deleting the first Capability) and move data to and fro between the Data-Part of some Object and directly addressable memory. Of course, we again stress that these operations cannot be performed on arbitrary objects, rather, the executing LNS must have a Capability for the Object to be accessed.

Most KALLs require some arguments which specify Capabilities. In the simplest case, these are denoted by SIMPLE INDEXes into the C-List of the LNS. For example, there is a KALL, 'DELETE', and DELETE (3) Kalls the Kernel to eliminate the 3rd Capability in the LNS executing that KALL. Often, the Kernel will allow a Capability to be denoted by a PATH INDEX (See Diagram 2). For example, DELETE (Path(3,4,2,1)) will delete the 1st Capability in the Object referenced by the 2nd Capability in the the Object referenced by the 4th Capability in the Object referenced by the 3rd Capability in the executing LNS. The Capability deleted is called the TARGET of Path(3,4,2,1). The Capability denoted by Path(3,4,2) is called the PRETARGET and the

Capabilities denoted by Path(3,4) and 3 are called STEPS. (Note: the denotation Path(3) is the same as just 3; such paths are called Simple)

.SUBSEC |KERNEL RIGHTS AND RIGHTS RESTRICTION|

As we noted, HYDRA implements basic protection through a set of rights. The right to perform some class of accesses (via KALLs of course) with respect to a Capability is determined by the presence of a particular bit in the Rights field of a Capability. (For a listing of all rights and respective bits, see the Appendix) The following is a description of the rights relevant to basic The following is a description of the rights relevant for basic Kernel Kalls. In describing these rights, we consider the effect of Capability CAP having the right in question. If CAP is an Object Reference, we write OBJ as a shorthand for the Object Referenced by CAP.

Capability Rights

DLTRTS - Allows CAP to be Deleted

ENVRTS - Allows CAP to be Stored in some Object

C-List Rights

LOADRTS - Allows a Capability to be Loaded from OBJ's C-List

STORTS - Allows a Capability to be Stored into OBJ's C-List

APPRTS - Allows a Capability to be Appended onto OBJ's C-List

KILLRTS - Allows a Capability to be Deleted from OBJ's C-List

Data-Part Rights

GETRTS - Allows data to be gotten from OBJ's Data-Part

PUTRTS - Allows data to put into OBJ's Data-Part

ADDRTS - Allows data to be appended onto OBJ's Data-Part

Restriction Rights

MDFYRTS - Allows modification of either OBJ's C-List or Data-Part

UCNFRTS - Allows OBJ to be 'UnCoNFined', that is, an Object accessed through OBJ may be modified.

Some examples:

DELETE (3)   (The Capability denoted by) 3 requires DLTRTS

DELETE (Path(3,4))   3 requires KILLRTS & MDFYRTS,

```
                                  Path(3,4) requires DLTRTS

  DELETE ( Path(3,4,2,1) )        3 and Path(3,4) require LOADRTS & UCNFRTS
                                  **,
                                  Path(3,4,2) requires KILLRTS & MDFYRTS,
                                  Path(3,4,2,1) requires DLTRTS
```

LOAD(x,y) is a KALL which moves the Capability at y to x, retaining the Capability at y. x must be a Simple Index.

```
  LOAD ( 4, Path(3,4,2) )         3 requires LOADRTS
                                  Path(3,4) requires LOADRTS
                                  4 must be an empty slot
```

Note that when a Capability is moved, it picks up DLTRTS, while the other rights remain the same as in the original.

TAKE(x,y) is just like LOAD but also deletes the Capability at y.

```
  TAKE ( 5, Path(3,4,3) )         3 requires LOADRTS & UCNFRTS
                                  Path(3,4) requires LOADRTS,
                                     MDFYRTS & KILLRTS
                                  Path(3,4,3) requires DLTRTS
                                  5 must be an empty slot
```

There is often a desire to restrict the Rights of a Capability when it is copied from one's own LNS to the C-List of another Object. Hence, the Kall, STORE(x,y,a) moves the Capability at y to x (y must be a Simple Index), and then restricts the rights of the Capability at x according to the contents of a mask at address a (See the Appendix for the format), by eliminating those rights not represented by a 1 in the mask.

```
  STORE ( Path(3,4,3), 2, addr )          3 requires LOADRTS & UCNFRTS
                                  Path(3,4) requires STORTS & MDFYRTS
                                  Path(3,4,3) must be an empty slot
                                  2 requires ENVRTS
```

If the address designating the rights restriction mask is zero, no rights are restricted. If the address is non-zero, then ALLYRTS (described in a later section) are always restricted regardless of whether the mask indicates that they should be.

.SUBSEC |AUXILIARY RIGHTS AND KERNEL TYPES|

The Rights we have seen so far are called Kernel Rights because they have meaning for any Capability regardless of the Type of the Object it references. In addition, each Capability also contains a field of Auxiliary rights that may be defined differently for each new Type of Object. Their use will become apparent in future examples.

The Kernel recognizes a basic set of Types and treats them seperately. Their auxiliary rights have predefined meanings and the Kernel also limits the Kernel rights that any Capability for an Object of one of these Types may have.

.SUBSEC |TYPES NULL, DATA & UNIVERSAL|

Objects of Type NULL represent absolutely nothing. They are constrained by the Kernel to have neither a C-List nor a Data-Part. What we have thus far referred to as an 'Empty slot' in a C-List contains a NULL Capability. The 'Length' of a C-List is the index of the last non-Null in the C-List. A Capability slot is said to be 'Defined' if its index is not greater than the Length of the C-List it refers to. In actuality, the preceding is a bit of a simplification. More details can be found in the Subsection on Nulls Revisited.

It is often convenient to be able to create a new Object which simply encapsulates some data. The Kernel provides a Kall, 'DATA' which does the encapsulation, creating a new Object of Type DATA whose Data-Part contains the data. DATA Objects have no C-List and have no defined Auxiliary rights.

It is also convenient to provide a UNIVERSAL Object, one with both a C-List and a Data-Part. The Kall UNIV creates just such an Object.

.SUBSEC |KALL VALUES AND SIGNALS|

Any KALL that executes successfully returns a non-negative value in register R$0. KALLs that fail (e.g. inadequate rights) return a negative value, called a "Signal" (In addition, certain additional signal related information is sometimes placed in SIGDATA, a fixed location in the stack page). There is also a mechanism that can force signals to cause user traps (See the section on Procedure & LNS Context Blocks for more details). A complete listing of signals and their values can be found in the Appendix. The meaning of the various signals that can occur during basic Kernel KALLs can be found in the Appendix.

.SUBSEC |LOCKING OF OBJECTS|

Since it is possible for two separate LNS's to contain Capabilities for the same Object, it is possible that both will be running simultaneously (on different processors) and will try to STORE different Capabilities in the same C-List slot of the shared Object. Such operations are performed indivisibly; when a Capability or Data is being moved either to or from an Object, that Object will (in general) be LOCKED. Hence, in the motivating example above, one LNS (nondeterministically will gain access to the Object and STORE a Capability in it, while the other waits on the Lock. When the STORE Kall completes, the other LNS will gain access to the Object, but its STORE Kall will fail (signal), since the slot in the shared Object will no longer be Empty.

For certain Kalls, if some referenced Object cannot immediately be locked, the Kall will fail. To do otherwise in those cases would allow the possibility of deadlock. For the same reason, any Kall that accesses a PROCEDURE Object (except when an LNS is being incarnated from it) must be able to lock the Procedure immediately or else the Kall will fail.

.SUBSEC |MEMORY ADDRESSES & THE STACK|

PDP-11's as modified for C.mmp have a 16 bit address space and a paged architecture. Pages are 8192 bytes long. The lower 13 bits of a 16 bit address designates a byte within a page. The high order 3 bits select one of 8 pages that may be directly addressable at any given time. Page 0 is designated the Stack Page to be used in conjunction with the PDP-11 SP register and is treated somewhat specially by the Kernel. HYDRA contains various KALLs that allow the user to change other pages (virtual overlaying). More details can be found in the section on PAGING. More details on the C.mmp hardware may be found in a separate document.

Many KALLs require one or more arguments to be memory addresses. Such memory address is expected to be the origin (low order address) of a block of memory from which the Kernel will either store or retrieve information. The KERNEL demands that these 'Legitimate Stack Memory addresses' have the following properties:

1) Such addresses be in the stack page (high order 3 bits of the address must be 0)

2) The block of memory to be accessed must lie within the active region of the stack or within the Process Communication Area, locations 0 - #176. (When an LNS begins execution, SP, the stack register, is set to point to an initial stack location. The modified PDP-11 hardware insures that SP can never be set higher than this initial value, that is the stack grows down. The region between the initial SP contents and the current contents of SP is called the Active Region of the stack).

3) The address must be on a word boundary (low order bit 0)

The stack may also be directly accessed using PDP-11 instructions since the stack is page 0. The modified C.mmp hardware prevents accesses to page 0 above the LNS's initial stack location, however, any access below that is allowed.

Locations 0 - #377 have special uses. Locations 0 - #177 comprise the Process Communication Area. It can be accessed by all LNS's that execute within a particular Process. Locations #200 - #377 comprise the Kernel Data Area. When signals, traps and errors occur, certain additional information is placed in locations within this area (The Appendix lists these fields) The Kernel also uses part of this area as working storage during Kalls.

.SUBSEC |INDIRECT KALLS|

Often it is useful to be able to build up the argument stack for a KALL independently of the actual KALL itself (especially for interpretive and debugging programs). The Appendix contains all details necessary for constructing the argument stack.

The special KALL, INDKALL (Mem), where Mem is the beginning address of the argument stack and must be a Legitimate Stack Memory Address provides this function.

.SUBSEC |CONVENTIONS FOR KALL SPECIFICATIONS|

A) KALLs are described in terms of Bliss Macros. See the Appendix.

B) The 'Parameters' section. Parameters to KALLs fall into three classes.

1) An integer value

2) A Legitimate Stack Memory Address - in the sense of the Subsection on Stack Memory Addresses. Where a memory address is optional, its absence is denoted by 0. The block of memory will in general be used either in conjunction with movement of data to or from a Data-Part or rights restriction. See the Subsection on Kernel Types and Rights Restriction and the Appendix)

3) A Denotation for a Capability - either a Simple index, (sometimes negated or 0 for a special effect) or a Path index, or a Call Parameter (to be defined in the Intermediate Kernel section). We will also indicate necessary rights, type or kind (Object Reference or Template) for the target Capability and its pretarget.

Unless we note otherwise in the specifications, we require that each STEP in a Path (Capabilities in the Path other than the Target or Pretarget) be an Object Reference Capability with LOADRTS.

We will not list restrictions on arguments that seem obvious or redundant and produce obvious signals if the restrictions are not met - most notably, indexes into C-Lists or Data-Parts less than 1 or greater than the maximum length.

C) 'Effect' is the effect of the Kall if no signal occurred. Except for two small subcases (of LNS incarnation and Page Set initialization), Kalls that fail have no side effects.

D) 'Signals' indicate unusual signals that may occur. Signals that indicate bad arguments or arguments that denote capabilities of the wrong kind or type or having inadequate rights are not mentioned. These are a possibility in almost every KALL and are described in the section on Signals above.

E) 'Result' is the value of the Kall (returned in R$0) assuming no signal occurred. (If a signal occurred, the value of the Kall is the signal value instead)

.SUBSEC |SPECIFICATIONS FOR BASIC KERNEL KALLS|

INFORMATIONAL KALLs

GETCLOCK (Mem)

Parameters:
- Mem - Legitimate Stack Memory address
- The current LNS must not be Blind (See next section)

Effect: Puts a reading of the system clock into the 4 word block of memory beginning at Mem. See the Appendix for the format.

Signals:
- SBLND - Current LNS is Blind

Result: 0

LENTH

Parameters: None

Effect: None

Result: Length of the C-List of the Executing LNS

CLENTH (Path)

Parameters:
- Path - Path index; Pretarget: LOADRTS; Target: Object Reference, LOADRTS

Effect: None

Result: Length of the C-List of the Object Referenced by Path's Target.

DLENTH (Path)

Parameters:
- Path - Path index; Pretarget: LOADRTS; Target: Object Reference, GETRTS

Effect: None

Result: Size of the Data-Part of the Object Referenced by Path's Target.

WHAT (Memd, Path)

Parameters:
- Memd - Legitimate Stack Memory address
- Path - Path index; Pretarget: LOADRTS; Target: Defined

- The current LNS must not be Blind (See next section)

Effect: Information about the Capability targeted by Path is stored in the 16 word block of memory beginning at Memd. See the Appendix for the format.

Signals:

SBLND - Current LNS is Blind

Result: 0

COMPAR (Path, Ncur)

Parameters:

Path - Path index; Pretarget: LOADRTS; Target: Defined

Ncur - Simple index, Defined or 0

Effect: None

Result: A word of bits which indicate how the Capabilities targeted by Path and Ncur compare. If Ncur is 0, then just those bits pertaining to the Capability targeted by Path are set. See the Appendix for the meanings of each bit.

SIMPLE DATA & UNIVERSAL MANIPULATION

GETDATA (Memd, Path, Disp, Knt)

Parameters:

Memd - Legitimate Stack Memory address

Path - Path index; Pretarget: LOADRTS; Target: GETRTS

Disp - Positive integer less than or equal to Dlenth(Path)

Knt - Positive integer

Effect: Moves up to Knt words of data from the Data-Part of the Object referenced by the Target to the block of memory beginning at Memd. The data is copied beginning at the Disp'th word of the Data-Part and continuing for a total of Knt words or until the end of the Data-Part is reached.

Result: Total number of words copied

PUTDATA (Path, Memd, Disp, Knt)

Parameters:

Memd - Legitimate Stack Memory address

Path - Path index; Steps & Pretarget: LOADRTS,UCNFRTS;
Target: PUTRTS,MDFYRTS

Disp - Positive integer

Knt - Positive integer

Effect: Copies Knt words of data beginning at Memd into the Data-Part of the Object targeted by Path. The data is stored beginning at the Disp'th word of the Data-Part.

Result: 0

DATA (Path, Memd, Knt, Memr)

Parameters:

Path - Path index; Steps: LOADRTS,UCNFRTS; Pretarget: STORTS,MDFYRTS; Target: Empty
Memd - Legitimate Stack Memory address
Knt - Non-negative integer
Memr - Legitimate Stack Memory address

Effect: Creates a Data Object and places a Capability for it in Path's Target. The Data-Part of the created Object will contain the Knt words of data copied from the block of memory beginning at Memd. The Capability will have all relevant rights except ALLYRTS & FRZRTS and will be further restricted by the contents of Memr if Memr is non-zero.

Result: 0

ADDATA (Path, Memd, Knt)

Parameters:
Path - Path index; Steps & Pretarget: LOADRTS,UCNFRTS; Target: ADDRTS,MDFYRTS
Memd - Legitimate Stack Memory address
Knt - Positive integer

Effect: Copies the Knt words of data from the block of memory beginning at Memd onto the end of the Data-Part of the Object referenced by Path's Target.

Result: 0

UNIV (Path)

Parameters:
Path - Path index; Steps: UCNFRTS,LOADRTS; Pretarget: STORTS,MDFYRTS; Target: Empty

Effect: Creates a Universal Object and places a Capability for it with all but ALLYRTS & FRZRTS in Path's Target.

Result: 0

SIMPLE MANIPULATION OF CAPABILITIES

PASS (Path, Ncur, Memr)

Parameters:
Path - Path index; Steps: LOADRTS,UCNFRTS; Pretarget: STORTS,MDFYRTS; Target: Empty
Ncur - Simple index, DLTRTS; if Path is not Simple, requires ENVRTS as well
Memr - Legitimate Stack Memory address or 0

Effect: Copies the Capability in the Ncur'th slot of the current LNS to Path's target, restricting rights (if Memr is nonzero) according to the contents of Memr. Then, the Capability at Ncur is deleted.

Result: 0

TAKE (Nnew, Path)

```
Parameters:
        Nnew - Simple index, Empty
        Path - Path index; Steps: LOADRTS,UCNFRTS;
               Pretarget: KILLRTS,LOADRTS,MDFYRTS; Target: DLTRTS
Effect:   Copies the Capability targeted by Path to the Nnew'th
        slot of the current LNS.  If Pretarget lacks UCNFRTS, then
        Nnew will lack UCNFRTS, MDFYRTS & ALLYRTS.  Then deletes the
        Capability targeted by Path.
Result:   0


  STORE ( Path, Ncur, Memr )
Parameters:
        Path - Path index; Steps: UCNFRTS,LOADRTS;
               Pretarget: MDFYRTS,STORTS; Target: Empty
        Ncur - Simple index, Defined; If Path is not Simple,
               requires ENVRTS as well.
        If Path and Ncur are the same, then none of the above Rights
                requirements holds, rather the Capability needs DLTRTS.
        Memr - Legitimate Stack Memory address  or  0.
Effect:   Copies the Capability in the Ncur'th slot of
        the current LNS to Path's target, setting DLTRTS, and (if Memr
        is nonzero) restricting rights according to the contents on Memr.
        If Path and Ncur are the same, however, the rights
        in the target are simply restricted according to the
        contents of Memr (if Memr is nonzero).
Result:   0


  LOAD ( Nnew, Path )
Parameters:
        Nnew - Simple index, Empty
        Path - Path index; Pretarget: LOADRTS; Target: Defined
Effect:   Copies the Capability targeted by Path
        to the Nnew'th slot of the current LNS,
        and sets DLTRTS.  If any Capability in Target's Path lacks
        UCNFRTS, Nnew will have UCNFRTS, MDFYRTS & ALLYRTS removed.
Result:   0


  PASSAPPEND ( Path, Ncur, Memr )
Parameters:
        Path - Path index; Steps & Pretarget: LOADRTS,UCNFRTS;
               Target: MDFYRTS,APPRTS
        Ncur - Simple index, DLTRTS,ENVRTS
        Memr - Legitimate Stack Memory address  or  0
Effect:   Appends the Capability in the Ncur'th slot of the current
        LNS onto the end of the C-List of the Object referenced
        by Path's target, restricting rights (if Memr is nonzero)
        according to the contents of Memr.  Then, the
        Capability at Ncur is deleted.
Result:   0

  APPEND ( Path, Ncur, Memr )
```

```
Parameters:
        Path - Path index; Steps & Pretarget: UCNFRTS,LOADRTS;
               Target: MDFYRTS,APPRTS
        Ncur - Simple index, ENVRTS
        Memr - Legitimate Stack Memory address  or  0
Effect:   Appends the Capability in the Ncur'th slot of the current
        LNS onto the end of the C-List of the Object referenced
        by Path's target, setting DLTRTS, and restricting rights
        (if Memr is nonzero) according to the contents of Memr.
Result:   0



  DELETE ( Path )
Parameters:
        Path - Path index; Steps: UCNFRTS,LOADRTS;
               Pretarget: MDFYRTS,KILLRTS; Target: DLTRTS
Effect:   Deletes the Capability targeted by Path.  See
        the Section on Types, Creating & Erasing in
        the next section for other potential effects.
Result:   0


  INTERCHANGE ( Path, Ncur, Memr )
Parameters:
        Path - Path index; Steps: UCNFRTS,LOADRTS
               Pretarget: MDFYRTS,KILLRTS,LOADRTS,STORTS;
               Target: DLTRTS
        Ncur - Simple Index, DLTRTS
        Memr - Legitimate Stack Memory address  or  0
Effect:   Interchanges the Capabilities targeted by Path and by Ncur.
        Restricts rights (if Memr is nonzero) of the Capability
        placed into Path's target according to the contents of Memr.
        If Pretarget lacks UCNFRTS, Ncur will have UCNFRTS, MDFYRTS &
        ALLYRTS removed.
Result:   0



↑L
```

.SEC |THE INTERMEDIATE KERNEL|

.SUBSEC |DOMAIN SWITCHING|

When an executing program wishes to invoke another program (e.g. call a subroutine), the caller may not trust the called program and may wish to isolate it in a separate environment (LNS), specifying as arguments only Capabilities for those Objects in its own LNS that it wishes the called program to be able to access. Alternatively, a program that manipulates a data base needs Capabilities to access the data base but it should never be necessary for callers of the program to have direct access to the data base.

To solve both problems, HYDRA provides PROCEDURE Objects. The Kall CALL(Rtrn,Proc,A1,...,Ak) creates a new LNS in which the Procedure's code will execute and transfers control to it. (Proc denotes a Capability for a Procedure Object, A1 through Ak denote Capabilities to be passed as arguments to the called procedure and Rtrn denotes a slot where the called Procedure may return a Capability) The Kall KRETURN passes control back to the calling LNS, optionally returning a Capability.

The C-List of a PROCEDURE contains Capabilities that will be duplicated in each LNS incarnated from the PROCEDURE (these are called inherited Capabilities and can be used to solve the Data Base problem mentioned just above). In addition, some of the Capabilities in the Procedure's C-List are Parameter Templates. Capabilities passed as arguments to the Procedure will appear in those slots in the LNS's C-List where Parameter Templates appeared in the Procedure's C-List. In addition to specifying where Call arguments appear in the incarnated LNS, Parameter Templates also specify a type and check-rights. A Call will fail (signal) if some argument is not of the same type and does not contain the minimum rights specified by the corresponding Parameter Template.

It is often useful to build 'Protected Subsystems'. Consider a Directory system where users have Capabilities for directories they can access, but because the 'Directory Subsystem' maintains the directories in a special private format, users should not be able to directly access or manipulate their directories except through PROCEDURES which comprise the 'Directory Subsystem'. HYDRA accomplishes this through 'Rights Amplification'. Capabilities passed as arguments in a CALL need not have the same rights in the incarnated LNS as in the LNS of the CALLer. The Parameter Template may specify new rights which may be greater than the rights of the Capability passed as an argument; in the incarnated LNS, the Capability will have these new amplified rights.

The diagram notes how this solves the Directory problem through the use of auxiliary rights and parameter templates which specify new-rights. The user's Capability for a Directory does not contain rights which allow manipulation or access to the directories directly. Rather various procedures of the 'Directory Subsystem' have parameter templates which specify these rights as new-rights, so that manipulation or access of a directory can only take place in the protected environment of the 'Directory Subsystem'. Note how auxiliary rights are used to control how a Directory may be used. Since different procedures specify different check-rights for Directories passed as arguments, auxiliary rights provide a way of specifying procedural protection. HYDRA does not permit parameter Templates which specify new-rights to be created anywhere, otherwise the protection afforded by the directory system could be easily circumvented. Templates which specify new-rights can only be created using special Capabilities (See the Subsection on Types, Creating & Erasing), and since Templates are Capabilities, their dispersion can be controlled. In the above case, the presumption is that only PROCEDURES of the 'Directory Subsystem' would have Parameter Templates of Directory Type with New-Rights.

Creation of an LNS and transfer of control to its code can be separated. The Kall MAKLNS incarnates an LNS from a Procedure and arguments, while the Kall LNSCALL transfers control to the LNS. The advantage of having such 'Canned' LNS's is efficiency as well as the ability to build coroutine structures. Once an LNS KRETURNs, it may be LNSCALLed again. Execution continues after the KRETURN. The LNS's pages, its C-List and registers R$0 and the PC will be retained, however, the rest of the registers will be clobbered and the stack will be reinitialized.

.SUBSEC |TEMPLATES & MERGING|

The process of comparing a Capability to a Template and producing a new Capability is called 'Merging'. It is useful not only as part of the Call Mechanism, but at other times as well. Hence, there are Capability Templates (for general merging) as well as Parameter Templates (for Call-time merging). Templates contain 2 flags.

TMPLFLAG - 1 - Capability Template
0 - Parameter Template

NEWFLAG - 1 - Amplify rights in Merging (new-rights)
0 - No amplification

These flags, if set, may be cleared in exactly the same way that rights may be restricted. Once cleared, they may not be set again. Since unlike Object References, Templates do not refer to specific Objects, there is little need for Templates to have rights. Therefore, without much conflict, rights and new-rights have been combined. Even when new-rights are specified, there are certain rights that cannot be amplified. This is true of the Kernel rights ENVRTS, UCNFRTS, FRZRTS and ALLYRTS. They will be the same in the merged Capability as in the original regardless of amplification.

.SUBSEC |NULLS REVISITED|

'Empty slots' have already been defined as slots containing NULL

Capabilities. In fact, it is impossible to create a NULL Object, and empty slots contain NULL Templates.

NULLs have one auxilliary right predefined, NULLRTS. We use the term 'Truenull' to mean a Null Template with both NULLRTS and TMPLFLAG set. When an Object is initially created, its C-List is set to contain all 'Truenulls' with all Kernel rights. A deleted Capability is also replaced by a Truenull.

The 'Length' of a C-List is really the index of the last non-Truenull in the C-List. Hence NULL Parameter Templates or NULL Templates lacking NULLRTS are included in the Length.

.SUBSEC |CONFINEMENT, FREEZING, BLINDNESS & REVOCATION|

A number of Kernel rights are provided to solve some interesting protection problems. ENVRTS, MDFYRTS & UCNFRTS are all used to solve variants of the 'Confinement Problem'. That is, they may be used to guarantee that Capabilities and data do not escape from particular LNS's; those LNS's are then said to be confined or partially confined with respect to the information whose leakage we wish to protect against.

ENVRTS can be used to guarantee that Capabilities are not stored by a Callee who is passed the Capability. Without ENVRTS, the Capability cannot be placed in the C-List of any Object. It may be used as an argument to an LNS which the Callee Calls, but ENVRTS cannot be gained through rights amplification.

As an example, Capabilities for LNS's never have ENVRTS and thus can never be accessed or manipulated outside of the Process in which the LNS has been incarnated.

MDFYRTS and UCNFRTS can be used to protect Objects from modification through Capabilities lacking those rights. If an LNS calls another LNS passing a Capability lacking MDFYRTS, that guarantees that the Callee cannot modify the accessed Object through that Capability regardless of amplification. This is because MDFYRTS cannot be gained through rights amplification and any Kall that modifies an Object requires a Capability for that Object with MDFYRTS as well as other relevant rights.

UCNFRTS also cannot be gained through amplification and prevents modification of any Object reached through the C-List of an Object referenced through a Capability lacking UCNFRTS.

Users may wish to guarantee that information passed to an untrusted procedure will not be leaked to another user. The Kernel right UCNFRTS also provides this guarantee. Any LNS incarnated from a Procedure Capability lacking UCNFRTS will be 'Confined'. Each Capability in the LNS inherited from the Called Procedure will lose UCNFRTS & MDFYRTS. Confinement is then provided in the following way. The reader may note that any Kall which modifies an Object requires that the

Capability for the Object have MDFYRTS and that other Capabilities in the Path to the Object have UCNFRTS. Additionally, whenever a Capability is loaded into an LNS through a Path where some Capability lacks UCNFRTS, the loaded Capability will have UCNFRTS, MDFYRTS and ALLYRTS removed. Hence, information and Capabilities cannot be stored by a Confined LNS through any Capabilities except those passed as parameters in incarnating the LNS.

Note that if a Procedure Capability with UCNFRTS is used as an argument in incarnating a Confined LNS, the Confined LNS will be able to Call an Unconfined LNS through it. Otherwise, since all inherited Capabilities of the Confined LNS lack UCNFRTS, any LNS called will be Confined as well.

There are still a small number of ways to covertly leak a few bits of information out of a confined LNS. It would be counterproductive to list these. However, no large leakage of data is possible.

Users may also wish to guarantee that an Object they have access to is 'Frozen', that is, the Object and all Objects reached by taking a Path through it will NEVER be modified, even by concurrently executing LNS's that may have a Capability for the same Object. The right FRZRTS is used like a flag to guarantee that an Object is frozen. The Kall FREEZE effectively freezes an Object by setting FRZRTS and eliminating UCNFRTS & MDFYRTS in what must be the only extant Capability for the Object. Since UCNFRTS & MDFYRTS cannot be gained through amplification, all Capabilities for the Object will lack them, guaranteeing that the Object will never be modified once frozen. FREEZE only succeeds if all Capabilities in the Object's C-List are already Frozen. So that FRZRTS can represent a guarantee of Frozen-ness, it also cannot be gained through amplification.

Users might further like LNS's to run 'Blind'. That is, no external information can be made available to it (the clock, process related information and other things that might change in different executions). FRZRTS also provides that function. Any LNS incarnated from a Procedure Capability with FRZRTS will be made Blind. In addition, an LNS incarnated by a Blind executing LNS will be Blind unless it is incarnated from a Procedure Capability with UCNFRTS.

Note that if a Procedure Capability with UCNFRTS is used as an argument in incarnating a Blind LNS, the Blind LNS will be able to Call an Unblind LNS through it. Otherwise, since all inherited Capabilities of the Blind LNS must have FRZRTS and thus must lack UCNFRTS, any LNS called will be Blind as well. Thus, with suitable arguments, execution of two Blind LNS's incarnated from the same Frozen Procedure Capability will be indistiguishable.

HYDRA allows Objects to act as Aliases for other Objects. Accessing such an Alias-ing Object actually causes access of the aliased Object. Aliases themselves may have aliases, allowing up to 23 levels of indirection. The Object finally accessed at the end of the alias indirection chain is called the 'Terminal Object' of an Alias.

An Alias may be created for any Object, and a Capability will be provided for the Alias-ing Object with ALLYRTS. With ALLYRTS, the ALiasing Object may be RE-ALLYed to act as Alias for a different Object or even for no Object at all. Thus, if a user wishes to share a Capability for an Object with another user, but might want to revoke the Capability at some later time, he need simply create an Alias for the Object and share the Capability for the Alias.

To guarantee that RE-ALLYing cannot be used to illicitly gain rights, whenever rights are restricted in a Capability, ALLYRTS are removed as well.

.SUBSEC |TYPES, CREATING & ERASING|

Objects of Type TYPE represent all Objects in the equivalence class of a given type. For example, the Object whose name is PROCEDURE and whose Type is TYPE represents all Objects whose type is PROCEDURE. Objects of Type TYPE are used to generate Templates of the Type named by the TYPE Object. A Template of a given Type is then used in CREATing an Object of that Type. There is a single Object in the system whose Name and Type are both TYPE which represents all the Objects in the system (including itself) whose Type is TYPE. (See diagram)

The way of creating a new Object of some type, say FILE, is to use the Kall CREAT, supplying as an argument a FILE Template with CREARTS. A FILE Template can first be gotten by using the Kall TEMPLATE, supplying a Capability for the FILE TYPE Object with TMPLRTS.

Initially, HYDRA provides Templates for each Kernel Type (though users may not directly be able to access these). These Templates do not have all Kernel rights, but rather a restricted set, depending on the Type. For these rights limitations, see the Appendix.

CREAT may expect some additional arguments when creating an Object of a Kernel type. For instance, in CREATing a new TYPE Object, CREAT expects a Memory address as an additional argument. The Kernel will use the information in that block of memory to store the following data in the Data-Part of the TYPE Object:

* PNAME - the Type's Print Name. While all Objects have a 64 bit bit unique name, TYPE Objects also have a Print Name. The Kall WHAT, given a Capability, produces (among other information), the PNAME of its Type.
* CAPINIT & CAPMAX - the initial length of the C-List (filled with Truenulls) and the maximum length of the C-List of any Object of the Type CREATed.
* DATAINIT & DATAMAX - the initial length of the Data-Part (zeroed) and the maximum length of the Data-Part of any Object of the Type CREATed.
* RTRVFLAG - An indication of whether Objects of this type are to be retrieved when all references to the Object are deleted (See following paragraph)

When all Capabilities for an Object have been deleted, the Object is normally garbage collected. However, it is possible to retrieve such Objects and prevent garbage collection on a Type by Type basis (see RTRVFLAG above). The Kall TYPRETRIEVE returns a Capability for an Object, all of whose references have been deleted (including aliases). To really garbage collect a retrievable Object, the Kall ERASE rather than DELETE must be used to delete the last Capability for the Object. Aliasing Objects are never retrieved.

.SUBSEC |PROTECTED SUBSYSTEMS|

Since Protected Subsystems are generally built around a particular type of Object (e.g. - the Directory Subsystem mentioned earlier), HYDRA provides a way to use a Subsystem without unnecessarily proliferating Capabilities for the Procedures which define it.

The C-List of a Type Object is used to implement protected subsystems easily by listing the Procedures which define it, and supplying access to those Procedures through the Kall TCALL.

If the Ndx'th Capability in the current LNS is of type T, and we use T[j] to denote the j'th Capability in the C-List of the T-Type Object, then TCALL(Rtrn,Ndx,j,a2,...,ak) is the same as CALL(Rtrn,T[j],Ndx,a2,...,ak). See the diagram.

.SUBSEC |SPECIFICATIONS FOR INTERMEDIATE KERNEL KALLS|

TEMPLATE MANIPULATION

TEMPLATE (Path, Nnew, Memr)

Parameters:

    Path - Path index; Steps: LOADRTS,UCNFRTS;
           Pretarget: STORTS,MDFYRTS; Target: Empty
    Nnew - Simple index, Type TYPE, TMPLRTS
         - or a negative integer between -1 and -13
    Memr - Legitimate Stack Memory address or 0

Effect: If Nnew is a Simple index, then TEMPLATE places a Template in Path's Target whose Type is the Name of the Nnew'th Capability in the Current LNS. The Template will have all flags and rights but FRZRTS & ALLYRTS.

If Nnew is negative, then a Template for the (-Nnew)'th Kernel Type is placed in Path's Target with TMPLFLAG set as well as various rights depending on the Type. The first 13 types are the predefined Kernel Types.

In either case, the rights of the new Template are further restricted according to the contents of Memr (if Memr is nonzero).

Result: 0

SETCHKRTS (Path, Mem)

Parameters:

Path - Path index; Steps: LOADRTS,UCNFRTS;
Pretarget: LOADRTS,STORTS,KILLRTS,MDFYRTS;
Target: Template, DLTRTS

Mem - Legitimate Stack Memory address

Effect: Sets the Check-Rights of the Template at Index according to the contents of Mem.

Result: 0

OBJECT MANIPULATION

CREAT (Nnew, Ncur, <arguments>)

Parameters:

Nnew - Simple index, Empty

Ncur - Simple index, Template, CREARTS; must not be NULL;
Also requires UCNFRTS if the Type is Retrievable

For description of additional arguments (only applicable when CREATing a Kernel Object) see the Appendix

Effect: Creates a new Object of the same Type as Ncur and places a Capability for it in Nnew. The rights in Nnew are the same as those in Ncur plus DLTRTS.

Result: 0

COPY (Nnew, Ncur, <arguments>)

Parameters:

Nnew - Simple index, Empty

Ncur - Simple index, Object Reference, COPYRTS

For description of additional arguments (only applicable when COPYing a Kernel Object) see the Appendix

Effect: Creates a new Object of the same type as Ncur and places a Capability for it in Nnew. In addition, the C-List and Data-Part of the new Object will be made the same as those of the original.

The rights of the new Capability in Nnew will be exactly the same as those for Ncur plus DLTRTS, unless the Object is of a Kernel Type in which case additional rights may be added. See the Appendix for details.

Result: 0

SWITCH (Path, Ncur)

Parameters:

Path - Path index; Steps & Pretarget: LOADRTS,UCNFRTS;
Target: Object Reference, MDFYRTS,OBJRTS

Ncur - Simple index, same Type as Path's Target, OBJRTS,MDFYRTS or 0

Effect: If Ncur is not zero, switches the C-List and Data-Part of the Objects referenced by Path's Target and Ncur. If Ncur is zero, destroys the Object referenced by the Target (same effect as ERASE). Future accesses of the Object will fail with either SCBND or SDBND signals.

Signals:

SLOCK - If the Object referenced by Ncur cannot be locked immediately

Result: 0

FREEZE(Ncur)

Parameters:

Ncur - Simple index, must be only extant reference to an Object, OBJRTS,UCNFRTS; Object must not be an Alias; Each Capability in C-List of Object must have FRZRTS

Effect: Effectively freezes the Object by doing the following to the only Capability for the Object: Sets FRZRTS and turns off UCNFRTS & MDFYRTS.

Signals:

SFRZ - Some Capability in the Object's C-List is not frozen. SIGDATA indicates the index of the last such Capability.

SUNQ - Ncur is not the only reference to the Object.

SALIAS - Ncur references an Alias

Result: 0

ALIAS (Nnew, Ncur)

Parameters:

Nnew - Simple index, Empty

Ncur - Simple index, Object Reference

Effect: Creates an Object of the same type as Ncur to act as an Alias for the Object referenced by Ncur. Any future references to to the new Object (unless changed by REALLY) will in fact access Ncur's Terminal Object. Nnew will have the same rights as Ncur except DLTRTS and ALLYRTS will be added and it will not have FRZRTS.

Result: 0

REALLY (Nnew, Ncur)

Parameters:

Nnew - Simple index, ALLYRTS (insures Aliasing Object)

Ncur - Simple index, Object Reference of same type as Nnew, except for DLTRTS & ALLYRTS, must have at least all the rights as Nnew has.
- or 0

Effect: If Ncur is not zero, re-allies the Object referenced by Nnew to be an alias for the Object referenced by Ncur. If Ncur is zero, the Object referenced by Nnew will become an alias for nothing and future references to it will fail with signal SALLY.

Result: 0

TYPRETRIEVE (Nnew, Ncur)

Parameters:

Nnew - Simple index, Empty or 0

Ncur - Simple index, TYPE Object Reference, UCNFRTS,RTRVRTS

Effect: If Nnew is not zero, retrieves a Capability for an Object of Type Named by Ncur, all of whose references have been deleted. The Kernel maintains the retrieval queue for each Object in FIFO order. The retrieved Capability has all rights set except FRZRTS and ALLYRTS (Aliasing Objects are not retrieved). If Nnew is non-zero, the Kall is executed for its Result only.

Result: Number of Objects in Ncur's Type's Retrieval queue (including Object retrieved - if any. Note a result of 0 indicates no Object was retrieved).

ERASE (Ncur)

Parameters:

Ncur - Simple index, must be only reference to Object, OBJRTS

Effect: Deletes last reference to an Object without placing it in its Type's retrieval queue. Also deletes each Capability in the Object's C-List. (If the Capability is for an aliasing Object, or no retrieval is indicated for the type, simply deleting the last reference to the Object has the same effect as ERASEing it.)

Signals:

SUNQ - Ncur is not the only reference to the Object

Result: 0

THE CALL MECHANISM

MERGE (Nnew, Ntmpl, Path)

Parameters:

Nnew - Simple index, Empty

Ntmpl - Simple index, Template, TMPLFLAG

Path - Path index; Pretarget: LOADRTS; Target: Defined, Rights must contain all those specified by Check-Rights field of Ntmpl. If Ntmpl is not Null, must be an Object Reference and must be of the same Type as Ntmpl. If Ntmpl is Null, may be of any Type and may be either an Object Reference or a Template.

Effect: Copies the Capability targeted by Path to the Nnew'th slot of the current LNS and sets DLTRTS. If Path's Target is a Capability for an Aliasing Object and Ntmpl has NEWFLAG set, a Capability for the Alias's Terminal Object is copied instead.

If Ntmpl has NEWFLAG set, Ntmpl's rights are copied to Nnew, except for ENVRTS, UCNFRTS, MDFYRTS & FRZRTS which are the same as in Path's Target.

If any Capability in the Path lacked UCNFRTS, then MDFYRTS, UCNFRTS & ALLYRTS will be removed from Nnew.

Signals:

SRTSM - Check-Rights failure

```
          SKNDT - Ntmpl is not a Template or does not have TMPLFLAG set.
          STYPC - Types of Path's Target and Ntmpl are not the same.
Result:   0


  MAKLNS ( Nnew, Nproc, <arguments> )
Parameters:
          Nnew - Simple index, Empty
          Nproc - Simple index, Procedure Object Reference
          - The 0 or more arguments must each be of the following form:
                  1] Path - Path index; Pretarget: LOADRTS;
                          Target: Requires ENVRTS if Nproc has PRCSRTS
                  2] Restrict ( Path, Memr ) - Path is as for [1] and
                          Memr is a Legitimate Stack Memory address  or  0
                  3] Transfer ( Path, Memr ) - Path is a Path index;
                          Steps: UCNFRTS,LOADRTS;
                          Pretarget: MDFYRTS,LOADRTS,KILLRTS;
                          Target: DLTRTS, also requires ENVRTS if
                                  Nproc has PRCSRTS.
                          Memr is a Legitimate Stack Memory address  or  0
                  4] Memdata ( Memd, Knt ) - Memd is a Legitimate
                          Stack Memory address  and  Knt is a positive
                          integer
                  5] Stkdata ( <data> ) - <data> is 0 or more words
                          of data
                  The Capability denoted by each argument must also
                  satisfy the requirements of its corresponding Parameter
                  Template (see MERGE)
Effect:   An LNS is incarnated from the Procedure and arguments and
          a Capability for it is placed in Nnew with DLTRTS.  In
          addition it will have UCNFRTS & FRZRTS, and the
          auxiliary rights LNSRTS & PRCSRTS if Nproc does.
            The LNS will be made Confined if Nproc lacks UCNFRTS.  The
          LNS will be made Blind if Nproc has FRZRTS or if the Current
          LNS is Blind and Nproc lacks UCNFRTS.
            All Capabilities in the C-List of the PROCEDURE which are
          either Object References or Capability Templates (TMPLFLAG set)
          are copied to the same slot in the C-List of the incarnated
          LNS.  If Nproc lacks UCNFRTS, each of these will have UCNFRTS,
          MDFYRTS & ALLYRTS removed.
            Parameter Templates in the C-List of the PROCEDURE are
          Capabilities specified by the Arguments.  Arguments are matched
          with Parameter Templates from last to first.  If fewer arguments
          are specified than Parameter Templates, the additional Parameter
          slots at the beginning of the LNS may be filled by Nulls (See
          the Section of PROCEDURE & LNS CONTEXT BLOCKS for details).
            The Capabilities that will be placed in the parameter slots
          of the LNS are the result of MERGEing the Parameter
          Template with a Capability specified by the corresponding
          argument.  For details of each individual merge, see the Effects
          part of the MERGE Kall.  As noted, arguments come in 5 flavors.
          The Capabilities they specify and additional side effects are
          as follows:
            1] Capability is Path's Target
```

2] Capability is Path's Target, restricted by Memr's contents if Memr is non-zero

3] Capability is Path's Target, restricted by Memr's contents if Memr is non-zero. In addition, the Capability at Path's Target is deleted. (N.B. use wisely, since, even if the Kall fails, the Capability may be lost)

4] Capability is for a newly created Data Object with all rights but FRZRTS & ALLYRTS. The Data-Part of the new Object will contain the Knt words of Data copied from the block of Memory beginning at Memd.

5] Capability is for a newly created Data Object with all rights but FRZRTS & ALLYRTS. The Data-Part of the new Object will consist of '<data>'.

Signals:

- If an argument is bad or any merge failed, the usual signal will be generated with SLNS orred in as well. In addition, the fixed location SIGDATA in the stack page contains the index of the affected slot in the incarnated LNS in its low order byte and the number of the affected argument in its high order byte.

SFARG - Too few arguments. SIGDATA indicates the minimum number of arguments acceptable.

SMARG - Too many arguments. SIGDATA indicates the maximum number of arguments acceptable.

SXCNF - LNS is not allowed to be made Confined. (See Section on PROCEDURE & LNS CONTEXT BLOCKS)

SXBLND - LNS is not allowed to be made Blind

Result: 0

LNSCALL (Rtrn, Nlns)

Parameters:

Rtrn - Simple index, Empty

Nlns - Simple index, LNS Object Reference, LNSRTS; The LNS must be "useable" (see Subsections on User Traps and Process Objects)

Effect: The LNS is Called and execution begins in its environment. When the Called LNS KRETURNs, it may specify a Capability to be returned. If Rtrn is not zero, it designates the slot where that Capability will be put. If Rtrn is zero, a returned Capability is simply discarded.

Signals:

- For Paging related signals, see the Paging Section

SSTK - Inadequate stack space available to run the LNS (See Section on PROCEDURE & LNS CONTEXT BLOCKS). SIGDATA contains amount of additional stack space needed.

SCNTRL - Callee returned by 'Punting a Control' rather than a KRETURN (See PROCEDURE & LNS CONTEXT BLOCKS).

SLOCK - LNS is currently in use (See PROCESS CREATION)

SREUSE - LNS may not be Reused (See next Section)

- When the Callee KRETURNs, it specifies a Return Value. If that value is negative, it is treated as a signal.

Result: Value returned by the Callee

CALL (Rtrn, Nproc, <arguments>)

Parameters:

Rtrn - Simple index, Empty or 0

Nproc - Simple index, Procedure Object Reference, CALLRTS

- Specifications for arguments are exactly as for MAKLNS. In addition to the 5 specified in MAKLNS, there are two more possible specifications:

6] Lns

7] Lnsrestrict (Memr) - Memr is a Legitimate Stack Memory address or 0

Effect: The effect is almost equivalent to the sequence MAKLNS (*, Nproc, <arguments>); LNSCALL (Rtrn, *). That is, the Kernel incarnates the LNS and Calls it, without the Caller ever having a Capability itself for the incarnated LNS. The only difference is that, unless required by Check-Rights in a Paramter Template, an argument's target does not require ENVRTS, regardless of whether or not Nproc has PRCSRTS.

The Capabilities denoted by the additional argument specifications noted above are:

6] Capability is for the Caller's LNS with DLTRTS, MDFYRTS, UCNFRTS, LOADRTS, STORTS, APPRTS, KILLRTS, GETCBRTS, SETCBRTS, GSTKRTS and PSTKRTS.

7] Capability is as in [6] with rights additionally restricted by the Memr's contents if Memr is non-zero.

Signals:

See MAKLNS & LNSCALL

Result: Value returned by Callee

KRETURN (Value, Ncur, Memr)

Parameters:

Value - Integer

Ncur - Simple index, ENVRTS or 0

Memr - Legitimate Stack Memory Address or 0

Effect: Causes return of control to current LNS's Caller with result Value. If Value is negative, Value is signalled as well in the Caller's environment. If the Caller specified a Rtrn slot and Ncur is non-zero (and the return slot has not otherwise had a Capability STOREd into it), the Capability denoted by Ncur is returned to that slot in the Caller's LNS with rights restricted by the contents of Memr (if Memr is not zero) and with DLTRTS added.

If the current LNS has no Caller, the current PROCESS will be stopped. Attempts to restart it will be unsuccessful.

Result: Current value of R$0. Control returns to Caller (unless a signal occurs). Control only continues normally after a KRETURN if the current LNS is subsequently LNSCALLed again.

PROTECTED SUBSYTEMS

```
  TLOAD ( Nnew, Ncur, Ntyp )
Parameters:
        Nnew - Simple index, Empty
        Ncur - Simple index, Defined
        Ntyp - Simple index into the C-List of the TYPE Object
               whose Name is the Type of Ncur, Defined
        - Current LNS must not be Blind
Effect:   If Ncur is a Capability of Type T, then the Capability in
        the Ntyp'th slot of the T TYPE Object is copied to the
        Nnew'th slot of the current LNS with DLTRTS added.  If
        Ncur lacks UCNFRTS, then MDFYRTS, UCNFRTS & ALLYRTS will
        be removed from Nnew.
Signals:
        SBLND - Current LNS is Blind
Result:   0


  TCALL ( Rtrn, Ncur, Ntyp, <arguments> )
Parameters:
        Rtrn - Simple index, Empty  or  0
        Ncur - Simple index, Defined
        Ntyp - Simple index into the C-List of the TYPE Object
               whose Name is the Type of Ncur, PROCEDURE Object
               Reference, CALLRTS
        - Current LNS must be Blind
Effect:   The effect is exactly equivalent to the sequence
        TLOAD ( *, Ncur, Ntyp );  CALL ( Rtrn, *, <Ncur,<arguments>> ).
        That is, the Kernel CALLs the Procedure in the Type Object
        without the Caller getting a Capability itself for the
        Procedure.  Ncur becomes the first argument of the CALL.
Signals:
        See TLOAD & CALL
Result:   Value returned by Callee

↑L
```

.SEC |MORE ON PROCEDURES & LNS'S|

.SUBSEC |PROCEDURE & LNS CONTEXT BLOCKS|

The Data-Parts of PROCEDUREs and LNS's are respectively known as Initial Context Blocks (ICB's) and Local Context Blocks (LCB's) and contain information relevant for execution and debugging. Information may not be gotten from or stored directly in Context Blocks using the standard Data-Part Kalls (GETDATA & PUTDATA), but rather specific Kalls (GETICB, SETICB, GETLCB & SETLCB) are used in conjunction with the auxiliary rights GETCBRTS and SETCBRTS. The list of fields in the Context Blocks, whether they can be read or written (in ICB or LCB), and their initial values (set at Procedure Creation time) can be found in the Appendix.

When an LNS is incarnated from a PROCEDURE, its LCB is copied from the ICB of the PROCEDURE, except for the field LVREG, which is set to the value of register R$0 at incarnation time.

When one LNS Calls another, the general registers of the Caller are saved in its LCB, as well as the bounds of its active stack region and the contents of three fixed locations in the stack, SAVREG, SAVVAL and STKOWN, known collectively as SAVAREA. These values are all restored when the Called LNS returns. The SP, PS and PC are saved in fields LSP, LPS and LPC of the LCB. Registers 1-5 are saved in fields LR1 - LR5, Register 0 is saved in LVREG, the upper bound of the active stack is saved in SPUFLO and the three fields of the SAVAREA are saved in SVREG, SVVAL and SVOWN.

When the Callee begins execution, its PC, PS and R$0 are initialized from the LCB (Paging information which determines the LNS's Page Set is also taken from the LCB - See the PAGING SECTION for more Details). When the Callee KRETURNs, R$0 and the PC are saved in the LCB (as well as Paging information), thus if the LNS is LNSCALLed again, execution will continue immediately following the KRETURN, though except for R$0 and the PC, the other registers will be clobbered and the stack and Page Set will be reinitialized.

.SUBSEC |USER TRAPS|

The LCB contains a number of user trap addresses which indicate the PC at which execution should continue after a Trap. Some of the traps roughly parallel the PDP-11 hardware (such as EMT & IOT) while others are provided by the HYDRA 'Virtual Machine'. Whenever a Trap is taken, the current PS and PC are pushed on the stack and execution proceeds at the Trap PC address with the PS same as the current PS except that Trace Trap Enable (bit 4) is turned off if it was on. The PS has the following format:

Bit	Meaning
0-3	Condition Codes
4	Trace Trap Enable

5-7	Hardware Priority
8-9	Hardware Space
12	Reuse Flag
13	Confined Flag (0 if Confined)
14	Blind Flag (0 if Blind)
15	Error Flag

The PDP-11 RTI instruction may be used to restore the old PC and PS. Bits 0-3, 4, 12 and 15 may have been changed in the stacked PS in any way. However, the Kernel checks RTI's and guarantees that fields 5-7, 8-9, 13 and 14 do not have values greater than when the LNS was incarnated.

The following Trap PC fields are used for Hardware traps:

EMTPC - EMT instruction
BKTPC - BKT instruction
TRCPC - Trace Trap
IOTPC - IOT instruction

In addition,

SIGPC - Signal PC, used when a Kall produced a signal

For all of the above Kalls, if the Trap PCs are 0 (especially important for signals), no Trap is performed.

Any hardware error that occurs while the user is executing causes a Trap to the PC found in ERRPC. In addition, after the trap is taken, the Error Flag is turned on in the current PS. It can be cleared by RTI'ing with a PS in which Error Flag is not set (such as the one pushed on the stack when the error trap was taken). An error that occurs while the Error Flag is set (instead of causing a new trap) causes the process to be stopped. If ERRPC is zero, the trap is not dismissed; again, the process is stopped. In any case, the reason for the error is or'ed into the fixed location ERRCODE in the stack (See the Appendix for the meanings of the various error codes).

The PRMASK is a mask of Processors on which the LNS can run. The mask is necessary since all C.mmp processors are not identical. Some have hardware floating point arithmetic, some run faster than others, and some may have a writable control store. If none of the needed processors are up, an Error will be caused. The PRMASK will be set to all 1's, and the old PRMASK will be put in SIGDATA.

The CONTROL Kall (See next section) provides an inter-process interrupt mechanism. It is meant to be used only for debugging and 'emergency' situations. The Kernel Objects PORTs and POLSEMs are meant to be used by users for interprocess communication and signalling. The CTLMASK field in the LCB is a mask of those control interrupts the current LNS will accept (there are 16 bits, hence, 16 different control interrupts). Regardless of the contents of CTLMASK, a Blind LNS will accept no interrupts. Any interrupt not accepted simply pends till it is accepted. CTLPC contains the Control Trap address. The control interrupts accepted will be or'ed into the fixed location CTLCODE in the stack . If CTLPC contains a 0, the current LNS will be forced to return, giving an SCNTRL signal to the Caller. In addition, all Controls indicated in CTLCODE will be re-controlled and thus may affect the Caller (as well as any control interrupts pending). This is known as 'Punting a Control'.

Control interrupts may also be used as part of a more desperate debugger. Before CTLPC is checked, the contents of CTLCODE are compared against the field DBGMASK. If any bits match, a debugging PROCEDURE is Called that will have complete access to the environment of the current LNS.

If DBGMASK matches any bits of CTLCODE, the contents of the field DBGNDX in the LCB is used to index the current LNS's C-List. It should denote a Capability for a Procedure Object with CALLRTS. If so, the Procedure is CALLed with one argument, a Capability for the current LNS (see the LNS specification in CALL). If the CALL results in any kind of Signal, the CTLPC trap is taken, otherwise, CTLPC is completely ignored.

Since the Debugging Procedure is incarnated with an argument for the LNS to be debugged, it can manipulate and access its C-List, its LCB (via SETLCB & GETLCB) and its stack (via the Kalls GETSTACK & PUTSTACK) - in short anything the executing LNS could do itself.

After execution of the Debugging Procedure, the value of R$0 will be restored from LVREG of the current LCB just as are the other registers. Thus, unless LVREG is changed by a SETLCB executed by the Debugging Procedure, R$0 will be the same as it was before the Control Interrupt was accepted. The value returned by the Debugging Procedure is only inspected to determine if it is negative, in which case, as a signal return, it forces execution to continue at CTLPC as noted above.

It should be noted that Capabilities for LNS's with access rights are only generated in CALLs, and thus it is impossible to access any LNS (except the current executing one) while that LNS is executing.

.SUBSEC |THE PS AND THE STACK|

The subsection on User Traps noted how RTI's were restricted in some ways so that the current PS would not become more priveleged than when the current LNS was called. The PS of another LNS (given a Capability for that LNS with SETCBRTS) can be modified as well, through modifying the field LPS with the Kall SETLCB. The restriction on fields 5-7, 8-9, 13 and 14 are the same.

Bit 12 of the LPS field is the Reuse Flag. It controls whether a KRETURNed LNS can be reused, either through a subsequent LNSCALL or by using the LNS to initialize a Process. Only if bit 12 is set may it be reused.

The LPS field of an ICB can be set as well. The restriction is that the priority and space fields (5-7 & 8-9) can be set no greater than those of the current PS. Bits 13 & 14 of the LPS in the ICB act as incarnation and Call requirements. If bit 13 is set, then Confined incarnations of the ICB's PROCEDURE are not allowed. If bit 14 is set, then Blind incarnations of the ICB's PROCEDURE are not allowed.

All LNS's in a Process use the same Stack Page. However, the stack is protected so that one LNS cannot access another's stack except through the Kalls GETSTACK and PUTSTACK. When an LNS Calls another LNS, the current bounds of its stack are stored in the LCB. SPUFLO (which cannot be altered) contains its upper bound, and LSP contains its lower bound. LSP can be changed as long as it is set below SPUFLO and above the address KALBND (See the appendix for the actual address of KALBND).

The active stack of an LNS which is not executing extends from SPUFLO to the value of SP when the LNS Called its Callee - #20. PUTSTACK can (given a Capability for an LNS) modify any portion of its active stack. The additional #20 bytes at the bottom of the stack provide a small area in which a debugger can extend the stack. Note that the actual value of LSP can be set even below that, but data cannot be put there. This is because it would run into the top of the stack of the LNS's Callee.

The field STKGROW is an estimate of the stack needed by an executing LNS. If not enough space is available on the stack to permit that much growth of the stack, the signal SSTK will be given when an attempt is made to Call the LNS.

.SUBSEC |MORE ON CONTEXT BLOCKS|

There is often a need to allow PROCEDUREs to accept a variable number of arguments when Called. If fewer arguments are passed to a Procedure than there are Parameter Templates, then, if the number of arguments is greater than or equal to the value of field ARGMIN in the ICB, the Call will succeed and the unfilled Parameter Templates will be filled with Nulls in the LNS; otherwise, the Call fails with signal SFARG.

ARGCALL in the LCB contains the actual number of arguments used in incarnating the LNS.

RTRNDX contains the index in the LNS that Called this one where a returned Capability will be placed.

PROCDATA is an 8 word field that can be used to identify the PROCEDURE. It is modifiable in the ICB, but when copied into the corresponding field of an incarnated LNS, it is not modifiable. The 8 word field LNSDATA is writable in both.

The remainder of the fields in the ICB/LCB have to do with Paging and are described in the Paging Section.

```
.SUBSEC |SPECIFICATION FOR CONTEXT BLOCK KALLS|


  GETICB ( Memd, Path, Code )
Parameters:
        Memd - Legitimate Stack Memory address
        Path - Path index; Pretarget: LOADRTS;
               Target: PROCEDURE Object Reference, GETCBRTS
        Code - Positive integer, legitimate code
Effect:   Copies information from the Initial Context Block of
        the Procedure into a block of Memory beginning at Memd.
        The content and amount of information copied depends
        on the Code.  For legitimate codes and what gets copied,
        see the Appendix.
Signals:
        SCODE - Bad Code
Result:   0


  SETICB ( Path, Memd, Code )
Parameters:
        Path - Path index; Steps & Pretarget: LOADRTS,UCNFRTS;
               Target: PROCEDURE Object Reference, SETCBRTS,MDFYRTS
                must reference a PROCEDURE Object.
        Memd - Legitimate Stack Memory address
        Code - Positive integer, legitimate code
Effect:   Uses information in the block of Memory beginning at
        Memd to set various values in the Initial Context
        Block.  For legitimate codes and their effects, see the
        Appendix.
Signals:
        SCODE - Bad Code
        SLPS - Bad PS (See Subsection on PS & the Stack)
Result:   0


  GETLCB ( Memd, Path, Code )
Parameters:
        Memd - Legitimate Stack Memory address
        Path - Path index; Pretarget: LOADRTS;
               Target: LNS Object Reference, GETCBRTS
             -  or  0
        Code - Positive integer, legitimate code
Effect:   Copies information from the Local Context Block of
        the LNS into a block of Memory beginning at Memd  (If Path
        is 0, then the current executing LNS is used).
        The content and amount of information copied depends
        on the Code.  For legitimate codes and what gets copied,
        see the Appendix.
Signals:
        SCODE - Bad Code
Result:   0
```

```
  SETLCB ( Path, Memd, Code )
Parameters:
        Path - Path index; Steps & Pretarget: LOADRTS,UCNFRTS
               Target: LNS Object Reference, SETCBRTS,MDFYRTS
             -  or  0
        Memd - Legitimate Stack Memory address
        Code - Positive integer, legitimate code
Effect:   Uses information in the block of Memory beginning at
        Memd to set various values in the Local Context
        Block of the LNS (if the Path is 0, then the current
        executing LNS is used).  For legitimate codes and their effects,
        see the Appendix.
Signals:
        SCODE - Bad Code
        SLPS - Bad PS
        SLSP - Bad SP
Result:   0


  GETSTACK ( Memd, Ilns, Meml, Knt )
Parameters:
        Memd - Legitimate Stack Memory address
        Ilns - Simple index, LNS Object Reference, GSTKRTS
        Meml - Legitimate Stack Memory address in the active
               stack of the LNS denoted by Ilns.
        Knt - Positive integer
Effect:   Moves up to Knt words of data from Meml to Memd.  Fewer
        than Knt words will be copied if there are fewer than
        Knt words above and including Meml in Ilns's active stack.
Signals:
        SLMEM - Meml is a bad stack address
Result:   Number of words copied


  PUTSTACK ( Ilns, Meml, Memd, Knt )
Parameters:
        Ilns - Simple index, LNS Object Reference, PSTKRTS,MDFYRTS
        Meml - Legitimate Stack Memory address in the active stack
               of the LNS denoted by Ilns
        Memd - Legitimate Stack Memory address
        Knt - Positive integer
Effect:   Moves Knt words of data from Memd to Meml.
Signals:
        SLMEM - Meml is a bad stack address
Result:   0

↑L
```

.SEC |PROCESSES, POLICIES & SEMAPHORES|

.SUBSEC |PROCESS OBJECTS|

Process Objects are the scheduling entities of the HYDRA Kernel. Unlike many systems, there is no explicit process hierarchy in HYDRA. To stop or start a process, one merely needs a Capability for the Process with the appropriate rights. Starting or stopping of one process has no effect on any other process.

Process creation is accomplished using the Kall CREAT already described.

CREAT (Nnew, Nprcs, Nlns) - Creation of Process Object

Parameters:

Nnew - Simple index, Empty
Nprcs - Simple index, PROCESS Template, CREARTS
Nlns - Simple index, LNS Object Reference, PRCSRTS;
The LNS must be "useable" (not currently active in an LNSCALL or Process CREAT which has not yet returned, and must have its REUSE Flag set if it has already been LNSCALLed and subsequently returned).

Effect: Creates a PROCESS Object and places a Capability for it in Nnew. The rights in Nnew are the same as those in Nprcs plus DLTRTS.
The LNS referenced by Nlns provides the initial environment (LNS) of the Process when it is first STARTed.

Signals:

- For Paging related signals, see the Paging Section
SLOCK - LNS currently active
SREUSE - LNS may not be reused

Result: 0

.SUBSEC |THE PROCESS BASE|

Optionally associated with a Process is a Process Base, a UNIVERSAL Object that remains associated with the Process over calls and returns. The Kall BLOAD loads a Capability from the current Process's Base into the current LNS and BCALL CALLs a Procedure in the Process Base. A Process Base can be used to provide generally available facilities to a Process or more likely, a group of processes.

If an LNS is confined, the Capabilities in the Process Base act as though they lacked UCNFRTS. If an LNS is Blind, the Process Base may not be used.

.SUBSEC |POLICY SUBSYSTEMS & LONG-TERM SCHEDULING|

Before a Process is able to run, it must be associated with a POLICY Object via the POLICY Kall (which also can associate a Process with its Base). Processes have specific resource needs, space (both for pages, in core and out, and for Objects) and cpu time. POLICY Objects provide the mechanism for allocation of these resources. By a 'Policy Subsystem', we mean the set of Procedures that manage the scheduling and allocation of the Processes associated with a particular Policy Object.

To allow multiple Policy Subsystems to coexist, each Policy Object is provided (via the Kall MAKEPOLICY) with resource guarantees (a percentage of CPU-time and memory allocation guarantees). In turn, a Policy Subsystem may fix memory guarantees for each process associated with it, which acts as an upper limit to the memory resources the process may use when running.

The Kalls START and STOP start and stop Processes and are the means by which a Policy Subsystem implements long-term scheduling.

The Kall START (given a Capability for a PROCESS with STARTS) swaps a process's pages into memory and makes the process available for execution. STARTing a Process associated with a POLICY Object P will fail, if the Process's memory guarantee added to the sum of the Process memory guarantees of all the running Processes assocaited with P exceeds P's memory guarantee.

When a Process is stopped, either by the Kall STOP or for some other reason, its pages may be swapped out and the memory allocated to it is made available for reallocation by the Policy Subsystem.

.SUBSEC |KMPS & THE PCB|

After a Process is started and until it is stopped, short-term scheduling is provided by KMPS, the Kernel MultiProgramming System. A Policy Subsystem can affect KMPS's scheduling by setting some fields (FPRIORITY, FNSLICES & FSLICE) in the Data-Part of the Process, its PCB (Process Context Block).

The fields in the PCB which affect KMPS scheduling are:

PRMASK - Processor mask, a mask of the processors upon which the Process may run. It is the same as the PRMASK of the LNS currently executing under the Process.

PRIORITY - Relative importance of a Process. When a processor becomes available, KMPS first chooses a Policy Object and then runs the highest priority Process associated with that Policy that can run on the processor. If the high order bit of PRIORITY is 1, the Process will not be stopped when it runs out of time (i.e. NSLICES & SLICE are ignored).

NSLICES, SLICE - Number of time slices & time slice size (in microseconds). KMPS will run a Process for NSLICES time slices of SLICE size each. When the process has used up its total time quantum, it is stopped, and must be reSTARTed before KMPS will schedule it again.

In addition, KMPS contains the following fields:

POLID - A word used by a Policy Subsystem to identify the Process (see THE POLICY QUEUE).

CPSMAX - Core Page guarantee. Maximum number of pages in the working set of any LNS executing under the Process.

CPSCUR - Number of pages in current working set.

TIMER - Remaining time is current slice.

NUSLICES - Number of time slices used (cleared when the Process is STARTed).

RSTATE - Running state. There are four possibilities:

0 - RUNNING. Process is actually running on a Processor.
1 - FEASIBLE. Process is in KMPS waiting to run.
2 - BLOCKED. Process is in KMPS but blocked.
3 - STOPPED. Process is not in KMPS.

RCVCODE - Policy Receive Code (See THE POLICY QUEUE). Contains bits indicating additional status of the process, including reasons why the process has been stopped. More than one bit may be set (See Appendix for meanings of each bit). The field is cleared when the process is restarted.

CTLMASK - Controls accepted by the LNS executing under the Process.

CTLCODE - Controls pending. A Control interrupt may be sent to a stopped process. If it matches any bits in CTLMASK, it will strike as soon as the Process begins running. Any control interrupts not accepted by CTLMASK will continue to pend until accepted by a change of CTLMASK.

.SUBSEC |EXECUTION PROTECTION|

Though HYDRA/C.mmp has been designed to be an extremely reliable system, a hardware failure can halt the execution of an LNS at an arbitrary time. Hence, users should adopt (in general) the MULTICS philosophy: When operating on sensitive information, leave enough audit information around so that a recovery procedure can complete the operation regardless of where in the operation a crash might have occurred.

More generally, while a user may build his own Policy Subsystem, it is likely that he will elect to use one made generally available to the user community. A Process may be STOPped at any time, and it is certainly within the range of possibility (especially using a buggy Policy Subsystem) that the Process may never be restarted.

A Policy Subsystem also has available the CONTROL Kall to send interrupts to a Process. A buggy subsystem may send so many interrupts that the executing LNS will spend all of its time fielding the control interrupts.

To solve all of these problems (except for the problem of unexpected crashes), the RUNTIME Kall is provided. RUNTIME specifies an amount of time during which the current Process will neither be stopped nor will receive any Control interrupts. RUNTIME also solves a more useful problem, to wit: Consider a Data Base that is accessed and changed frequently by cooperating concurrent processes. If access and modification are fast operations, then if the operations are execution-protected by RUNTIME, a busy-wait lock which is part of the Data Base may suffice to provide mutual exclusion rather than more complex (though better structured) use of synchronization objects (SEMAPHOREs, POLSEMs & PORTS).

Some uncertainties about execution can be resolved if a user has some information about the Policy Subsystem and its status with which her program executes. The Kall INFPOLICY returns a word that reflects such information. The value of that word is set when the POLICY Object was created.

.SUBSEC |SEMAPHORES|

SEMAPHORE Objects are supplied to provide short term synchronization for trusted Subsystems. In general, users will not have Capabilities for Semaphores but will use POLSEMs (POLicy SEMaphores) and PORTs instead.

Semaphore Objects are created with an initial count (parameter for Semaphore CREAT) that specifies the number of PSEM's more than VSEM's that may be executed without causing the Process to wait. A Process waiting on a SEMAPHORE is not stopped, and in fact, cannot be STOPped (and thus swapped out) until it passes the SEMAPHORE.

When a SEMAPHORE is erased, it is first V'd as many times as are necessary to wake up all Processes waiting on the Semaphore.

For reliability, a limit is set for the amount of time a Process may be blocked on a SEMAPHORE. If the Process is blocked for a longer time, the Process continues execution and its PSEM (the Kall which P's a Semaphore) fails.

.SUBSEC |THE POLICY QUEUE|

The Kernel keeps a queue for each POLICY Object. When a Process stops, information about the stopped process is placed in the POLICY queue. The Kall RCVPOLICY is used to extract an entry from the Policy queue in FIFO order. (The Policy queue is also used for other Process related messages. See the section on PORTS & POLSEMS for further details). The information extracted includes POLID so that the Policy Subsystem can identify the Process affected.

```
.SUBSEC |SPECIFICATIONS FOR PROCESS, SEMAPHORE & POLICY KALLS|


PROCESS CONTEXT BLOCKS


  GETID
Parameters:
        - Current LNS must not be Blind
Effect:   None
Signals:
        SBLND - Current LNS is Blind
Result:   Process ID of the current Process


  GETPCB ( Memd, Path, Code )
Parameters:
        Memd - Legitimate Stack Memory address
        Path - Path index; Pretarget: LOADRTS;
               Target: PROCESS Object Reference, GETCBRTS
        Code - Positive integer, legitimate code
Effect:   Copies information from the Process Context Block of
        the Process into a block of Memory beginning at Memd.
        The content and amount of information copied depends
        on the Code.  For legitimate codes and what gets copied,
        see the Appendix.
Signals:
        SCODE - Bad Code
Result:   0


  SETPCB ( Path, Memd, Code )
Parameters:
        Path - Path index; Steps & Pretarget: LOADRTS,UCNFRTS
               Target: PROCESS Object Reference, SETCBRTS,MDFYRTS;
               Unless the PROCESS is the current one, the PROCESS
               must be stopped.
        Memd - Legitimate Stack Memory address
        Code - Positive integer, legitimate code
Effect:   Uses information in the block of Memory beginning at
        Memd to set various values in the Process Context
        Block.  For legitimate codes and their effects, see the
        Appendix.
          If current PCB is being changed, then any current RUNTIME
        is cancelled.
Signals:
        SPRCS - Process not stopped
        SCODE - Bad Code
Result:   0



PROCESS BASE
```

```
  BLOAD ( Nnew, Ncur )
Parameters:
        Nnew - Simple index, Empty
        Ncur - Simple index into the current Process's Base, Defined
        - Current LNS must not be Blind
Effect:   Copies the Ncur'th Capability from the current Process Base
        to the Nnew'th slot of the current LNS adding DLTRTS.  If the
        current LNS is Confined, Nnew will lack UCNFRTS.
Signals:
        SKNDC - No Process Base
        SBLND - Current LNS is Blind
Result:   0


  BCALL ( Rtrn, Ncur, <arguments> )
Parameters:
        Rtrn - Simple index, Empty  or  0
        Ncur - Simple index into the current Process's Base,
               PROCEDURE Object Reference, CALLRTS
        - Current LNS must not be Blind
Effect:   The effect is exactly equivalent to the sequence
        BLOAD ( *, Ncur );  CALL ( Rtrn, *, <arguments> ).
        That is, the Kernel CALLs the Procedure in the Process
        Base without the Caller getting a Capability itself for
        the Procedure.
Signals:
        See BLOAD & CALL
Result:   Value returned by Callee



SCHEDULING & CONTROL


  START ( Nprcs )
Parameters:
        Nprcs - Simple index, PROCESS Object Reference, STARTS,UCNFRTS;
                Process must be stopped but runnable
Effect:   Pages in the Process and enters it in KMPS
Signals:
        SPRCS - Process is not Stopped
        SPOL - Process not associated with Policy Object
        SPOP - Initial LNS of Process has returned
        SGUAR - Policy Object guarantee has been exceeded.  SIGDATA
                contains more information (See Appendix).
Result:   0


  STOP ( Nprcs, Code )
Parameters:
        Nprcs - Simple index, PROCESS Object Reference, STOPRTS,UCNFRTS;
                Process must be in KMPS
        Code - Integer
```

Effect: Removes Process from KMPS and enters an entry (including Code - called the Rcvcode) in the associated Policy's RCVPOLICY queue.

Result: 0

CONTROL (Nprcs,, Code)

Parameters:

Nprcs - Simple index, PROCESS Object Reference, CTLRTS,UCNFRTS - or 0

Code - Integer

Effect: Causes Control interrupts specified by Code to be sent to the Process (Current process if Nprcs is 0). See Subsection on User Traps.

Result: 0

RUNTIME (Tim)

Parameters:

Tim - Integer

- Current LNS must not be Blind

Effect: If Tim is zero, forces KMPS to reconsider its scheduling, which will cause a runnable process at the same or higher priority to run instead. In addition, though CTLMASK & PRMASK may be changed in the current LCB, the change only becomes effective if a RUNTIME (or call or return) is executed.

RUNTIME also provides for uninterrupted execution. During that time the process may not be stopped (except due to errors, WORKSET and PPOLSEMs) and no Control interrupts are accepted.

If Tim is positive, then if Tim is available in the total time remaining in the current and all remaining time slices, then execution proceeds uninterruptably (except for short term rescheduling by KMPS). Tim is in 1/2 seconds up to 1 minutes.

If Tim is negative, then if -(Tim) is available in the current time slice, execution proceeds uninterruptably (except for hardware device interrupt handling). If -(Tim) is not available in the current time slice, but is less than or equal to the time slice size and at least one time slice remains, then before uninterrupted execution begins, the current time slice is ended and rescheduling is considered (but the process may not be STOPped or Control Interrupted). -(Tim) is in 16 microseconds up to 1/2 second.

In either case, if the requested time is not available, the process is stopped. When reSTARTed, if the PCB has not been changed to make the requested time available, the Kall fails.

If RUNTIME succeeds and a subsequent RUNTIME is executed in the uninterruptable period, pending STOP's and Control interrupts are re-enabled before the new RUNTIME takes effect.

Signals:

STIM - Requested time not made available

SBLND - Current LNS is Blind

Result: 0

SEMAPHORES

PSEM (Path , Tim)

```
Parameters:
        Path - Path index; Steps & Pretarget: LOADRTS,UCNFRTS;
               Target: SEMAPHORE Object Reference, MDFYRTS
        Tim - Positive integer
Effect:   P's the Semaphore
Signals:
        SSEM - Process has been blocked on the Semaphore for more than
               Tim seconds.
Result:   0
```

CPSEM (Path)

```
Parameters:
        Path - Path index; Steps & Pretarget: LOADRTS,UCNFRTS;
               Target: SEMPAHORE Object Reference, MDFYRTS
Effect:   Conditionally P's the Semaphore.  The P is only executed if
        the process will not have to wait on it.
Result:   1 if the P was executed, 0 if not.
```

VSEM (Path)

```
Parameters:
        Path - Path index; Steps & Pretarget: LOADRTS,UCNFRTS;
               Target: SEMAPHORE Object Reference, MDFYRTS
Effect:   V's the Semaphore
Result:   0
```

VASEM (Path)

```
Parameters:
        Path - Path index; Steps & Pretarget: LOADRTS,UCNFRTS;
               Target: SEMAPHORE Object Reference, MDFYRTS
Effect:   V's the Semaphore exactly as many times as are needed to
        wake up all Processes waiting on it.
Result:   Number of V's done
```

POLICY KALLS

POLICY (Nprcs, Npol, Nuniv)

```
Parameters:
        Nprcs - Simple index, PROCESS Object Reference, MDFYRTS;
                If Npol is non-zero, requires POLRTS;
                If Nuniv is non-zero, requires BASERTS
        Npol - Simple index, POLICY Object Reference, POLRTS,MDFYRTS
```

```
                 - or 0
         Nuniv - Simple index, UNIVERSAL Object Reference, ENVRTS
                 - or 0
Effect:    If Npol is non-zero, associates POLICY with the PROCESS.
           If Nuniv is non-zero, makes the UNIVERSAL Object the
         Process's Base.
Result:    0


  RCVPOLICY ( Memd, Npol )
Parameters:
         Memd - Legitimate Stack Memory address
         Npol - Simple index, POLICY Object Reference, RCVRTS,MDFYRTS
Effect:    Extracts an entry from the Policy's queue and puts the
         information from the entry into the 16 word area in memory
         beginning at Memd.
           If the queue is empty, the Process waits until an entry
         arrives.
Result:    0


  MAKEPOLICY ( Nnew, Ncur, Memd )
Parameters:
         Nnew - Simple index, POLICY Object Reference, MAKERTS,MDFYRTS
         Ncur - Simple index, POLICY Object Reference, MAKERTS,MDFYRTS;
         Memd - Legitimate Stack Memory address
Effect:    Transfers allocations and guarantees between the two
         POLICY Objects.  The 16 word block beginning
         at Memd contains information about how allocations and
         guarantees are to be transferred.
Signals:
         SGUAR - Bad guarantee specification.  SIGDATA indicates
                 what was wrong.  See Appendix for Details.
Result:    0


  WHATPOLICY ( Memd, Npol )
Parameters:
         Memd - Legitimate Stack Memory address
         Npol - Simple index, POLICY Object Reference
Effect:    Information about the guarantees and allocations of the
         POLICY Object is put into the 16 word area beginning at Memd.
Result:    0


  INFPOLICY ( )
Parameters:
         - Current LNS must not be Blind
Effect:    None
Signals:
         SBLND - Current LNS is Blind
Result:    One word of Policy information (set by Policy CREAT)

↑L
```

.SEC |PAGING|

.SUBSEC |INTRODUCTION|

The single largest impact of the PDP-11 on the design of the paging system is that the PDP-11 processor is only able to generate a 16-bit address. Thus user programs, at any instant, may address at most 64K bytes, or 32K words. The second largest impact arises from the fact that the relocation hardware divides the user's address space into eight 8K-byte units called "Page frames". Since this is a rather small address space, much of the design of the paging system is oriented toward making these restrictions somewhat easier to live with.

In the following material we shall use the term "Page" to refer to an Object, in the HYDRA-technical sense of that word, of type PAGE. In many contexts the term "Page" may also be read to mean the information contained in the PAGE Object. The term "Page frame", or simply "frame", on the other hand, will be used to refer to the area of physical primary memory (core) in which the information content of a Page Object resides. The term "frame" is also used to indicate a portion (1/8th) of the user's address space; context should disambiguate these uses.

Since Pages are Objects, a user program may, and generally will have one or more Capabilities which reference specific Pages. These Capabilities may be in the LNS of an executing LNS or contained in some Object, e.g., a Directory, which can be named by a Path rooted in the current LNS. Possession of a Capability for a Page, however, does not make it addressable. In particular, it is possible that many more Pages may be named in some particular LNS than can be simultaneously addressed by the PDP-11 hardware. Thus the paging system defines means by which the user may specify and alter the set of Page Objects which are physically present in primary memory and which of these may be directly accessed at any instant.

Each active LNS has associated with it a CPS (Current Page Set) and an RPS (Relocation Page Set). The set of pages referenced by the CPS is guaranteed to be core-resident while the LNS is executing. The set of pages in the RPS (a subset of those in the CPS) is precisely the set whose Page frames are named by the relocation hardware of C.mmp (excluding the stack page which is fixed by the Kernel for the life of a Process). Thus the Pages in the RPS (plus the stack page) are those whose information may be accessed directly by instructions executed by the PDP-11 processor which is executing the user's program. Of necessity the RPS must refer to seven or fewer pages; no such restriction exists for the CPS.

Memory allocation (as well as long term scheduling) are controlled by the particular Policy Subsystem with which the user's Process is associated. While in principle, the CPS may be of arbitrary size, in practice it is advantageous for a user to limit the size of her CPS to make scheduling more likely, though such guarantees depend on the particular Policy Subsystem.

.SUBSEC |MANIPULATING PAGE SETS|

The Kall CPSLOAD enters pages into the CPS. Loading the current LNS's CPS implies that the designated pages must be brought into core, and the user may assume that they are. In reality however any i/o necessary to make the Pages core- resident is merely requested at this point and a wait-for-i/o-complete, if necessary, is done only when the user requests that a Page be included in his RPS. It should be noted that if a designated CPS slot previously contained a reference to some other Page, that reference is lost and the corresponding Page may become eligible to be swapped out of core, assuming, of course, that the pages are not referenced by the CPS of some other executing LNS.

The Kall RRLOAD provides the user with the ability to move pages from the CPS to the RPS, and hence to be able to reference these Pages directly. As noted above, this operation may imply waiting for the specified Page to become physically resident in primary memory. Once the Page is resident, however, it will remain resident so long as it remains in the CPS and the procedure is active. When the user's Process is stopped, the pages may be swapped out. They are swapped back in when the Process is reSTARTed.

The CPS, RPS, and the functions listed above effectively define a three level memory system - the Pages namable by, or through, the LNS, those named in the CPS, and those named in the RPS. Normally each of these is a subset of the preceding (the exception being that once a Page Capability is loaded into the CPS it may be deleted from the LNS). For the small program, these sets may be identical and the user need not concern herself with the paging system. For larger programs, the user must manage these sets, and the way in which she does so may significantly impact the performance of her program.

.SUBSEC |INITIALIZATION|

An LNS's LCB contains an IPS (Initial Page Set) which specifies how the CPS/RPS is to be initialized when it is Called (by automatically performing CPSLOADs and RRLOADs).

INCPS - Initial size of the CPS

ICPS - 47 words long, the first 'Incps' of which are used to initialize (CPSLOAD) the CPS. Each word contains:

- 0 - CPS slot will be empty
- +m - CPS slot will be CPSLOADed with the Page whose Capability is in the m'th slot of the LNS's C-List
- -m - Just like +m, except the Capability is deleted from the LNS's C-List as well (This is useful for pages which the program never manipulates, but must be used carefully, since the Capability may be deleted even if the Call fails)

IRPS - Seven words used to initialize the seven RPS slots. Each word either contains an index into the CPS (that page will then be RRLOADed) or 0 (addresing such a page will cause a NXM error - Non eXistant Memory. The same thing occurs if the CPS slot was empty).

MAXSIZE - Maximum CPS size. Fixed for the life of the LNS.

When an LNS is incarnated from a PROCEDURE, the IPS in the LNS's LCB is copied from the IPS fields of the PROCEDURE's ICB. Slots in the ICPS may denote Page Templates in the PROCEDURE's C-List. In the LNS, these will denote Capabilities for Page Objects passed as arguments in incarnating the LNS.

When an LNS Calls another LNS, the pages in the Caller's LNS become eligible to be swapped out. When the Callee returns, the Caller's pages are automatically first swapped back into core if necessary before execution proceeds.

An LNS's IPS remains unchanged during the life of the LNS. Hence, if an LNS KRETURNs and is subsequently LNSCALLed again (or made the initial LNS of a Process), its CPS and RPS will be re-initialized using the same IPS, even though the C-List of the LNS may have changed as a result of previous execution, and even though execution will continue at the PC following the KRETURN.

Multiple usage of an LNS may of course be prevented by use of the REUSE Flag in the LCB's PS word (See Subsection on the LCB & ICB)

.SUBSEC |CPS SIZE & THE WORKING SET|

There are 2 limits placed on the size of a CPS. First, the Kernel has a fixed limit on the total number of CPS slots allocated to active LNS's (those Called which have not yet Returned) in a Process. Secondly, a Process's PCB contains a field (CPSMAX) which limits the maximum CPS size for any LNS executing under the Process. A Call may fail if the Called LNS's MAXSIZE exceeds the first limit, or if the LNS's INCPS field exceeds the Process's CPSMAX.

The Kall WORKSET provides a way for (all but Blind) LNS's to dynamically change the size of the CPS (the LNS's Working Set). It is always possible (and usually advantageous) to lower the CPS size. It may not be raised at all above the LNS's MAXSIZE, but it may be raised over the Process's CPSMAX. If it is, the Process is stopped, and much like the Kall RUNTIME, the Policy Subsystem is given a chance to raise the Process's CPSMAX so that the WORKSET Kall will succeed when the Process is restarted.

A Call or Return always causes a WORKSET to be implicitly executed since the CPS size may differ in the Caller and Callee. If, on a Return, CPSMAX is lower than the Caller's CPS size, not only will the Process be stopped, but it will not be successfully restarted until CPSMAX is adequately raised (it will just be stopped again). One small additional point; a Blind LNS may not Call an LNS whose initial

CPS size is greater than the current CPS size.

.SUBSEC |AUXILIARY RIGHTS FOR PAGES|

Two pre-defined auxiliary rights for pages have a somewhat special property. They are used by the C.mmp hardware when loaded into the RPS to determine how the page may be addressed by PDP-11 instructions.

A Page loaded from a Capability lacking PGWRTS (or the Kernel right MDFYRTS) may not be written into.

A Page loaded from a Capability with CACHRTS (and the Kernel right FRZRTS) is cacheable. The right will be used in conjunction with the PDP-11 code cache when it is implemented.

In addition, the auxiliary right CPSRTS allows the Page to be CPSLOADed. If a Page Capability lacks CPSRTS but does contain COPYRTS, it is called an Initialization Page. The Page may be COPYed, and the Capability for the COPYed Page will have CPSRTS (as well as PGWRTS and CACHRTS).

.SUBSEC |COPYING PAGES|

When a PAGE is COPYed, a CPS slot must additionally be specified indicating where the page may be CPSLOADed. So the COPY Kall for Pages is specified as follows:

COPY (Nnew, Npage, Ncps) - Copying of Page Object

Parameters:

Nnew - Simple index, Empty

Npage - Simple index, PAGE Object Reference, COPYRTS

Ncps - Positive integer, no greater than the current LNS's CPS size

Effect: Creates a new Page Object and places a Capability for it in Nnew. In addition, the contents of the page referenced by Npage is copied into the new page. The new page is then CPSLOADed in the Ncps'th CPS slot.

The Kernel rights of the new Capability in Nnew will be the same as those in Npage plus DLTRTS, however, all Auxiliary rights will be set in Nnew.

Signals:

SCPSBND - Ncps is out of bounds

Result: 0

.SUBSEC |SPECIFICATIONS FOR PAGING KALLS|

PAGE (Path)

Parameters:

Path - Path index; Steps: UCNFRTS,LOADRTS; Pretarget: STORTS,MDFYRTS; Target: Empty

```
Effect:   Creates a Page Object and places a Capability for it with all
        relevant rights but ALLYRTS & FRZRTS in Path's Target.
Result:   0


  CPSLOAD ( Nlns, <cps-page-pairs> )
Parameters:
        Nlns - Simple index, LNS Object Reference, MDFYRTS,SETCBRTS
             -  or  0
        <cps-page-pairs> - One or more pairs of < Ncps, Path >, where:
                Ncps - Positive integer, no greater than the LNS's
                       current CPS size
                Path - Path index; Pretarget: LOADRTS;
                       Target: PAGE Object Reference, CPSRTS
                      -  or  0
Effect:   For each pair, loads the Page targeted by Path into
        the Ncps'th CPS slot of the LNS denoted by Nlns (the current
        LNS if Nlns is 0).  If Path is zero, the CPS slot is just
        emptied.
          See RRLOAD for additional effects.
Signals:
        SCPSBND - Some Ncps is out of bounds (above the CPS size or
                  below 1).  SIGDATA contains the index of the bad pair
        - The usual signals can occur because of a bad Path
          specification.  In addition, SPAGE will be or'ed in and
          and SIGDATA will contain the index of the bad pair.
Result:   0


  RRLOAD ( Nlns, Nrps, Ncps )
Parameters:
        Nlns - Simple index, LNS Object Reference, MDFYRTS,SETCBRTS
             -  or  0
        Nrps - 1 through 7
        Ncps - Positive integer, no greater than LNS's CPS size  or  0
Effect:   Loads a page into the Nrps'th RPS slot of the LNS denoted
        by Nlns (the current LNS if Nlns is 0) from the Ncps'th CPS
        slot.  If Ncps is zero, the RPS slot will be set to NXM.
          If the CPS slot was CPSLOADed from a Capability
        with both CACHRTS & FRZRTS, the page may be cached.  If the
        CPS slot was CPSLOADed from a Capability with
        both PGWRTS & MDFYRTS, the page may be written into.
Signals:
        SCPSBND - Ncps is out of bounds.
        SRPSBND - NRPS is not 1 through 7
Result:   CPS slot index of the page previously loaded in the
        Nrps'th RPS slot (0 if RPS slot was NXM).


  WORKSET ( Nlns, Size )
Parameters:
        Nlns - Simple index, LNS Objects Reference, MDFYRTS,SETCBRTS
             -  or  0, in which case, the current LNS must not be Blind
        Size - Positive integer, no greater than the LNS's CPS MAXSIZE
```

```
Effect:   Changes the CPS size of the LNS denoted by Nlns (the current
        LNS if Nlns is 0).
          If Nlns is zero and Size is greater than the current Process's
        CPSMAX, then the Process is stopped.  If CPSMAX has not been
        raised to cover Size when the Process is restarted, the Kall
        fails.
Signals:
        SIPSMAX - Size greater than MAXSIZE.
        SCPSMAX - CPSMAX has not been raised to cover Size.  SIGDATA
                  contains CPSMAX.
        SBLND - Current LNS is Blind
Result:   0


Paging Signals for LNSCALL & Process CREAT:
        - The usual signals occur if an ICPS entry denotes something
          other than a Page Object Reference with CPSRTS, however,
          SPAGE will be or'red with the Signal.  SIGDATA will
          contain the bad ICPS index in its low order byte and
          the bad LNS slot is denotes in its upper byte.
        SCPSBND - An IRPS slot contains a bad index into the CPS.
                  The low order 3 bits of the signal indicate the
                  bad RPS slot (1 - 7).
        SIPSMAX - INCPS is greater than MAXSIZE
        SCPSMAX - One of three things may be wrong:
          1) MAXSIZE > available remaining Process CPS allocation
          2) Current LNS is Blind and INCPS > current CPS size
          3) Current LNS is not Blind and INCPS > CPSMAX even after
             the Process has been stopped and restarted.
          If the current LNS is not Blind, SIGDATA contains CPSMAX
          in its low order byte and the available remaining Process
          CPS allocation in its high order byte.

↑L
```

.SEC |THE PASSIVE GST|

.SUBSEC |INTRODUCTION|

The collection of Objects is called the GST (Global Symbol Table). The entire GST is too large to completely reside in main memory. So, only actively referenced Objects (the Active GST) are kept in core. The remainder of the GST (the Passive GST) is kept in secondary memory.

If an Object is in the Passive GST, it will be brought into the Active GST when it is referenced. Normally, it will migrate back to the Passive GST when no Capabilities for the Object are in Active Objects. Though not currently implemented, there will be a limit to the amount of Active GST space that a Process may use (similar to the CPS limit, CPSMAX, in the PCB). Thus, it is necessary to allow a user to PASSIVATE an Object. The Active GST space occupied by the Object will then no longer be charged against the Process until an LNS executing under the Process subsequently references the Object. The Kall PASSIVATE will not actually cause the Object to migrate back to the Passive GST unless no other processes are actively referencing it.

The Kernel takes great care to insure the reliability of the GST. For example, if an error occurs in an Active Object due to faulty memory, the Kernel will attempt to fix it by using available redundant information in the Object structure as well as the most recent copy of the Object in the Passive GST. Thus, it is useful to provide a Kall, UPDATE, that for reliability reasons, updates the most recent copy of the Object in the Passive GST, regardless of whether or not other Processes are actively referencing it.

.SUBSEC |SPECIFICATIONS FOR PASSIVE GST KALLS|

PASSIVATE (Path)

Parameters:

Path - Path index; Pretarget: LOADRTS; Target: Defined.

Effect: If Path's Target is last Active reference for the Object it references, the Object will migrate back to the Passive GST and each Capability in the Object's C-List will also be PASSIVATEd.

Result: 0

UPDATE (Path)

Parameters:

Path - Path index; Pretarget: LOADRTS; Target: Defined.

Effect: Has the same effect as PASSIVATE, except the Object will be updated in the Passive GST in any case. In addition, each Capability in the C-List of the Object referenced is UPDATEd.

Result: 0

↑L

.SEC |PORTS|

The documentation of the port system is being revised. Beware!

The Hydra Message System is the primary means of communication, synchronization and input/output for user PROCESSes. It consists of a set of primitive Kernel Kalls which allow PROCESSes to exchange "messages" with each other and with the input/output system via software switching and queueing centers called PORTS. Message transfers are fully synchronized so that other forms of synchronization, i.e., semaphores, mailboxes, etc. will often be unnecessary.

Two types of objects are handled by the Message System: PORTS and "messages". The characteristics of these objects will be discussed, followed by a discussion of the primitive operations on them.

.SUBSEC |WHAT IS A MESSAGE|

A message is basically a string of bytes attached to some routing and queueing information.

More concretely (but not right down at the nitty-gritty) a message has four parts:

1) A message "type",

2) A "reply stack" (possibly null) of places the message has been sent from and to which it might return as a reply, and

3) A text buffer of length >=0 which may be partially or completely filled with information.

4) An owner - i.e. the PORT in which the message was originally cre
**ated
and to which the (storage) resources used by the message are cha
**rged
until the message is destroyed.

The message type is an integer in the range 0-15 (decimal). It is not a static attribute fixed at the time of creation of the message. Instead it is set every time the message is sent (via SEND, RSVP, or REPL
**Y) which
may in general be many times before its destruction. When waiting for a message a PROCESS might choose to accept only those of a given type or a given set of types. Thus the programmer may encode some meaning or classification scheme into his use of the message type field as a convenience in structuring the communication among several PROCESSes.

He might, for example, use the type to distinguish "normal" messages from "exceptional" and "catastrophic", or to distinguish replies from non-replies.

Type 0 messages have a special meaning under certain circumstances which are discussed later under the description of REPLY. If the programmer is not interested in those circumstances he may use type 0 just as he would any other.

The "reply stack" of a message is employed when the programmer uses the RSVP or the REPLY command. It is a stack of places (i.e., PORT, input channel pairs) which are eligible to receive replies to this message. Basically, the RSVP operation causes a frame of data about the sender and the reply he wants to be pushed onto the message's stack w **hile the REPLY operation pops one (or more) frames from the stack and uses the information to return the reply. The use of this stack is described in greater detail under the descriptions of RSVP and REPLY. Here it is imPORTant only to note that the maximum stack depth (possibly zero) is set at the time of creation of the message and is static.

The text-buffer PORTion of the message is where the data (or text) is stored. It has a maximum length decided by the user at MCREATE-time and cannot be changed. The text buffer may be partially or completly filled using the MWRITE command so that the "length" of the message is always less than or equal to the length of the buffer. The contents of the text buffer of a message are, of course, completely uninterpreted by the Kernel. The "meaning" of the message is decided by the communicating PROCESSes.

It is perfectly legitimate to have a text buffer of length zero (no text buffer). If the programmer can communicate all he needs to in the type field then there is no need for text at all. The current maximum length of a text buffer is 1024 words (decimal).

The owner of the message is the PORT in which it was originally crea **ted. At the time a PORT is created it is given an allotment of storage to be u **sed for the creation of messages. When a message is created the amount of storage it uses is deducted from the resource account of the PORT. If t **he PORT has insufficient resources, the message cannot be created. The resources ar **e returned to the creating PORT whenever the message is destroyed. The pu **rpose of this feature is to limit the total number of messages outstanding in t **he system, thus preventing the disaster that might otherwise be caused if **a PROCESS tried to create an unbounded number of messages.

For efficiency reasons messages are not implemented as true Hydra objects with unique names and capability lists. Consequently there are no capabilities for them; they cannot be passed as parameters to PROCEDURES; and they cannot appear in DIRECTORYs. However they are similar to objects in that they can only be manipulated indirectly through Kernel Kalls and they reside in storage belonging to the Kernel.

.SUBSEC |WHAT IS A PORT|

A PORT is a software post-office where messages are queued, received, stored and dispatched. Messages may be routed from one PORT to another (or to the same PORT) or from a PORT to an I/O Device object, provided that a "connection" has been established first.

Unlike messages, PORTs really are full-fledged Hydra objects in the technical sense. Furthermore, they are predefined and understood directly by the Kernel in a way similar to objects of type PAGE, LNS, PROCEDURE, etc.

A PORT should be thought of as having five main parts:

1) A Resource Account - the total amount of storage (in words) allo **wed for outstanding messages created in this PORT.

2) Input Channel Section: 0-16 (decimal) "input channels" for queueing incoming messages.

3) Output Channel Section: A fixed number of "output channels" each of which may contain the name of (at most) one PORT or I/O Device object to which messages can be sent.

4) Local Name Section: A fixed number "local names". A local name is a slot for holding a message which has come to the attention of some PROCESS (i.e., a newly created or received message). A message can only be referred to by its local name.

5) Waiting PROCESS Section: a queue of suspended PROCESSes waiting for messages to arrive.

The actual capacity figures for a PORT are established when it is created and are fixed for its entire lifetime.

.SUBSUBSEC |OUTPUT CHANNELS, INPUT CHANNELS AND CONNECTIONS|

An output channel, when connected, holds a reference to an input channel of some PORT (possibly the same one the output channel is part of) or a reference to an i/o object. Whenever a message is sent it is sent via some output channel to the place that channel references, and ** thus at least one output channel is necessary if any messages are to leave the PORT (other than as a reply). Here is no simple upper limit to the number of output channels a PORT may have.

An input channel is simply a message queue. Since all incoming messages are received through an input channel, any PORT which is to receive messages must have at least one. A single PORT may have up to 16 input channels. Multiple input channels can be useful because the RECEIVE routine allows a PROCESS to wait for messages arriving on any subset of input channels. He can thus assign meanings to certain input channels as a convenience in his communication structure.

The CONNECT operation is used to "connect" an output channel to an input channel (or to an I/O Device object). Once a connection is made between two PORTs, messages can be sent between them in the direction of the connection. A connection may be broken using the DISCONNECT operation, and in general connections may be established, broken and then restablished to somewhere else many times during the lifetime of a PORT (although this is not expected to be frequent).

An output channel can be connected to at most one input channel at a time. However, many output channels may be connected to the same input channel. Thus, when a message is sent via an output channel it is always clear where it is going. But when a message is received from an input channel it is not in general clear which of several places it may have come from unless the programmer restricts himself to a one-to-one connection pattern or labels each connection with a "connection ID". (See CONNID parameter in the CONNECT operation.) It is not possible to tell how many, if any, output channels are connected to a particular input channel.

A brief bit of Hydra philosophy might be injected here. Notice first that messages are sent from PORT to PORT, not PROCESS to PROCESS. Therefore, one PROCESS need not know the name of (i.e., have a capability for) another PROCESS to get a message to it. This is expecially imPORTant in a system of several equivalent server PROCESSes which are sharing a message PROCESSing load. Merely sending a message to the PORT that they presumably share is sufficient to assure that one of them (and only one) will receive it. The number of server PROCESSes may change dynamically with time with no effect upon the action of the requesting PROCESSes.

Another consequence of this Message System design is that the programmer of a system using PORTs has strong control over the communication structure and can use the capability mechanisms of Hydra enforce that control. Messages cannot be sent arbitrarily between any two PORTs - only between PORTs that are connected. By appropriately controlling the flow of capabilities for PORTs, particularly those with right PCONNRTS of connection and disconnection, he can assure the integrity of the connection graph. He can further restricts his communication by limiting the distribution of the other auxilliary rights for the message-handling primitives, thereby achieving further protection. (See the list of auxilliary rights supPORTed by the Message System.)

.SUBSUBSEC |LOCAL NAMES|

Every PORT contains a set of message-holding pigeon-holes called "local names" which are numbered from 0. There is no simple upper limit to the numbernof local names a PwoRT may have. Each such local name can hold only one message at a time. In order for a PROCESS to perform any of the primitive operations upon a message, that message must be sitting in a local name of some PORT.

When referring to a message in order to perform an operation on it the user cannot simply give its address because he has no way of getting it (or accessing it even if he had it). Instead he refers to the message by specifying the pair (P,L) where P is the LNS index of a ca **pability for a PORT and L is the index of a local name within that PORT. (We will abbreviate from now on and say that L is a local name in some PORT, as opposed to the index of a local name.) Each of the primitive operations MREAD, MWRITE, SEND, RSVP and REPLY have just such a pair as their first two arguments.

A local name is in one of two states, "full" or "free", according to whether it holds a message at the moment or not. When a message is created via MCREATE the system searches for a "free" local name and allocates it to the new message, changing the state of the local name to "full". The user can then operate the message using MWRITE, SEND, RSV **p or REPLY. Once SEND, RSVP or REPLY is done, the local name becomes "free" again. Similarly, when a message is received via RECEIVE, the system has to search for a free local name to put it in before returning to the user, whereupon he may perform MREAD, etc., on the message.

If the Message System is unable to find a "free" local name an error condition is signaled. (NOTICE: The PROCESS is NOT suspended. This is to avoid deadlock in the case that only one PORT is using the PORT.) Thus, the local names of a PORT should be considered a valuable and scarce resource. If a PROCESS or group of PROCESSes uses the local names of a PORT unwisely it will require very complex algorithms to properly handle the error signals and get out of the jam without deadlock or other disaster. It may be advisable for PROCESSes sharing a PORT to control their use of local names via some kind of limit semaphore. However, any such arrangement is outside the Message System.

A single PORT may have up to 64 local names. the exact number is decided at the time the PORT is created and is static for the life of the PORT. Since, in order to do any message operations a local name is required, every PORT must have at least one. For simple message PROCESSing, where each PROCESS disposes of one message before beginning to PROCESS another, no more than one local name per PROCESS using the PORT is necessary.

"Local names" are so called because they are "local" to a single PORT. However, if several PROCESSes are using the same PORT it is possible for one PROCESS to interfere with another by operating on messages in local names that were never assigned to that PROCESS by MCREATE or RECEIVE. In that sense local names are really "common" or "global" to all PROCESSes using the same PORT. It is therefore very imPORTant that PROCESSes using the same PORT cooperate with one another in this respect.

.SUBSUBSEC |WAITING PROCESSES|

When a PROCESS does an unconditional RECEIVE operation for some class of messages and no message of that class has arrived, the PROCESS must be suspended. The identification of the PROCESS and the class of messages it is waiting for are placed on a queue associated with the PORT. Whenever a message arrives this queue is examined to see if any PROCESSes are waiting for it. Since messages-waiting-for-a-PROCESS and PROCESSes-waiting-for-a-message can both be queued, a PORT acts very much like a fancy semaphore.

When a message arrives at a PORT no more than one PROCESS is awakened for it. Two PROCESSes cannot receive the same message even if they are waiting for exactly the same class of messages.

.SUBSUBSEC |RELATION OF PORTS TO I/O OBJECTS|

As previously described, an output channel of a PORT may optionally be connected to an input/output device object instead of to an input channel of a genuine PORT. The device object, though technically not part of the Message System, acts abstractly as though it were really a PORT with one input channel and no output channels. An i/o request for the physical device associated with the device object is then implemented as a message sent to the device object. The result of the i/o operation is implemented as a reply to the request message. Exceptional and normal replies will generally have different types and thus return to different places according to the reply stack of the request message. (Historically, the requirement for exception handling in i/o was the primary model for the RSVP/REPLY mechanism of the Hydra Message System.)

The fact that a device object viewed as a PORT has no output channel
**s
means that there can never be a CONNECT operation between two device objects. It also means that the i/o system never creates or sends a message. It can only reply to messages that have been sent to it.

There is one departure from the abstraction that a device object acts like a PORT: only one output channel at a time can be connected to any particular device object. This corresponds to the notion that - at least at the lowest level - a hardware device belongs to only one PROCESS at a time.

CONNECT (Port1, Outchan, Port2, Inchan, Connid)

Parameters:

- PORT1 - Simple Index of PORT object reference;CNFRTS;PCONNRTS
- Outchan - Integer, either -1 or between 0 and N-1 inclusive, where N is the number of output channels in the first PORT.
- Port2 - Simple Index of PORT object or I/O Device object; PCONNRTS
- Inchan - Integer between 0 and N-1 inclusive, where N is the number of input channels in the second PORT. This parameter is ignored if Port2 refers to an I/O Device object.
- Connid - Any 16 bit pattern.

Effect: The output channel designated by Outchan in the first PORT is "connected" to the input channel designated by Inchan in the second PORT, thereby forming a path for messages to travel. The output channel is marked "connected" so that further CONNECT operations on the same output channel will fail until and unless it is DISCONNECTed first.

If Outchan is -1 the kernel selects a free output channel and makes the connection, signalling if there are no free output channels.

Connid is used as a symbol to identify the connection, and is part of the information stamped on every message that travels along the path made by the connection. It may be used for any purpose since it is completely uninterpreted by the kernel. (See RECEIVE for another reference to this feature.
**)

For purposes of the CONNECT operation an I/O Device object is identical to a PORT which is limited to only one input channel. However, there can be no more than one connection to an I/O Device object. A signal will be generated if an attempt is made to connect to an I/O Device object which is already connected. (There is no such restriction on connections to the input channels of a PORT.)

Signals: All signals from CONNECT will have SGPCONNECT in bits 6-10 and one of the following values in bits 0-5:

- SGPOCHANRANGE - Outchan is less than -1 or greater than highest output channel index of PORT1.
- SGPNOFREEOCHAN - Outchan is -1, but there are no free output channels available.
- SGPALREADYCONNECTED - Outchan specifies an output channel which is already connected.
- SGPICHANRANGE - Inchan is negative, or greater than the highest input channel index of in Port2.
- SGPIOERR - Attempted connection to an I/O Device object which is already connected.

Result: CONNECT normally returns the index of the output channel which was connected. This is either Outchan or, in the case Outchan is -1, the selected output channel.

DISCONNECT (Port, Outchan)

Parameters:

- PORT - Simple Index of a PORT object; CNFRTS;PCONNRTS
- Outchan - Integer index of the output channel to be disconnected.

Effect: The output channel Outchan of the specified PORT is logically "disconnected" from wherever it was "connected". The output channel may now be re-connected to somewhere else. There is no distinction between disconnecting from a PORT and disconnecting from an I/O Device object.

Signals: Signals from DISCONNECT have SGPDISCONNECT in bits 6-10 and one of the following values in bits 0-5:

- SGPOCHANRANGE - Outchan is negative or larger than the largest output channel index in the PORT.

- SGPUNCONNECTED - The output channel is not connected and thus cannot be disconnected.

Result: 0

MCREATE (Port, Bufflength, Stackdepth)

Parameters:

- PORT - Simple Index of a PORT object; CNFRTS; MCREATERTS
- Bufflength - Integer between 0 and #4000 (octal); specifies the length of the message buffer in bytes, i.e. the maximum length of the text of the message.
- Stackdepth - Integer between 0 and 10 (decimal) inclusive; specifies the maximum depth of the message's reply stack.

Effect: A new message is created according to the specification of of the Bufflength and Stackdepth parameters. A free local name is found in the PORT and the new message is assigned to that local name. The resources (storage) consumed by the message are deducted from the resource account associated with the PORT.

Signals: All signals from MCREATE have SGMCREATE in bits 6-10 and one of the following in bits 0-5:

- SGMBUFFLENGTH - Bufflength is negative or greater than the implementation defined maximum of #4000 bytes.
- SGMSTACKDEPTH - Stackdepth is negative or greater than the implementation defined maximum of 10.
- SGMRESOURCES - There are insufficient resources left in the resource account associated with the PORT to allow creation of this message.
- SGMNOFREELNAME - There are no unassignad local names to give to the message.

Result: MCREATE normally returns the local name assigned to the new message.

MREAD (Port, Lname, Pos, Len, Textadr)

Parameters:

- PORT - Simple Index of a PORT object; CNFRTS;MREADRTS
- Lname - Integer local name in the PORT
- Pos - Byte index (origin 0) of the section of the message buffer to be read.
- Len - Length in bytes of the section of the message buffer to be transferred.
- Textadr - Legitimate Stack Memory Address of an area at least Len bytes long.

Effect: The section of the message buffer designated by Pos and Len is copied into the stack area pointed to by Textadr.

Signals: All signals from MREAD have SGMREAD in bits 6-10 and one of the following in bits 0-5:

SGMLNAMERANGE - Lname is negative or out of range of the local names of the PORT.
SGMLNAMEFREE - Local name Lname is free, i.e. has no message assigned to it.
SGMBUFFBOUNDS - Pos and/or Len do not specify a segment wholly contained within the text of the message.
SGMTEXTADR - Textadr does not specify a Legitimate Stack Memory Address of an area at least Len bytes long (or the area is not wholly contained in the legitimate area of the stack).

Result: 0

MWRITE (Port, Lname, Pos, Len, Textadr)

Parameters:
PORT - Simple Index of a PORT object; CNFRTS;MWRITERTS
Lname - Integer local name in the PORT
Pos - Byte index (origin 0) of the section of the message buffer to be written.
Len - Length in bytes of the section of the message buffer to be written.
Textadr - Legitimate Stack Memory Address of an area at least Len bytes long containing the data to be written into the message.

Effect: The data in the area pointed to by Textadr is copied into the section of the message buffer specified by Pos and Len.

Signals: All signals from MWRITE have SGMWRITE in bits 6-10 and one of the following in bits 0-5:

SGMLNAMERANGE - Lname is negative or out of range of the local names of the PORT.
SGMLNAMEFREE - Local name Lname is free, i.e. has no message assigned to it.
SGMBUFFBOUNDS - Pos and/or Len do not specify a segment wholly contained within the message buffer.
SGMTEXTADR - Textadr does not specify a Legitimate Stack Memory Address of an area at least Len bytes long (or the area is not wholly contained in the legitimate area of the stack).

Result: 0

SEND (Port, Lname, Type, Outchan)

Parameters:
PORT - Simple Index of a PORT object; CNFRTS; SENDRTS
Lname - Integer local name of the message to be sent.

Type - Integer in the range 0-15 to become the new type of the message.

Outchan - Output channel index specifying the destination of the message.

Effect: The type indicator of the message with local name Lname is set to Type and the message is sent to the PORT or I/O Device to which output channel Outchan is connected. Local name Lname becomes free. There is no effect upon the other attributes of the message, i.e. its owning PORT, its message buffer, or its reply stack.

When the message arrives at the destination PORT and input channel it may satisfy the requirements of one or more PROCESSes that were blocked in a RECEIVE operation. If so, exactly one of the eligible blocked PROCESSes is awakened to receive the message; the other PROCESSes remain blocked. The longest blocked eligible PROCESS is always selected in order to enforce a policy of fairness. (Strictly speaking, the PROCESS is not awakened; rather the appropriate POLICY object is notified that it may schedule the selected PROCESS.)

If no PROCESSes are blocked at the destination PORT or if the incoming message does not satisfy the type or input channel criteria of any of the blocked PROCESSes, then the message is enqueued (in FIFO order) in the proper input channel and type queues. It will be received by the first PROCESS which does a RECEIVE operation on the same PORT for some class of messages to which this one belongs. Under no circumstances does ** the sending PROCESS get blocked.

If the destination of the message is an I/O Device (as opposed to a PORT) the I/O system immediately receives the message and begins to act on it.

Signals: All signals from SEND have SGSEND in bits 6-10 and one of the following in bits 0-5:

SGMLNAMERANGE - Lname is negative or out of range for this PORT.

SGMLNAMEFREE - Local name Lname is free, i.e. assigned to no message.

SGMOCHANRANGE - Outchan is negative or out of range for this PORT.

SGMUNCONNECTED - Output channel Outchan is not connected.

SGMTYPERANGE - Type is not in the range 0-15 inclusive.

Result: 0

RSVP (Port, Lname, Type, Outchan, Messid, Inchan, Replymask)

Parameters:

PORT - Simple Index of a PORT object; CNFRTS; SENDRTS

Lname - Integer local name of the message to be sent.

Type - Integer in the range 0-15 to become the new type of the message.

Outchan - Output channel index specifying the destination of the message.

Messid - 16 bit identifier for the message.

Inchan - Integer index of the input channel through which the reply (if it returns at all to this PORT) is to retur **n.

Replymask - 16 bit mask specifying (with 1-bits) which types of reply are to return to this PORT. Replies of other t **ypes will bypass this PORT.

Effect: RSVP does the same thing as SEND, but in addition requires that a reply be generated. The first four parameters to RSVP ar **e interpreted exactly like the four parameters to SEND. It is the last three parameters which provide the information necessary for ** the REPLY mechanism and which distinguish RSVP from SEND.

Just before doing the equivalent of a SEND operation, RSVP pushes a frame of information onto the message's reply-stack. This frame controls the action of the subsequent REPLY operation, and includes as data the last three parameters to RSVP: Messid, Inchan and Replymask.

Rsvp guarentees that a reply message will be generated by som **eone at some later time. But it does not guarentee that the reply will return to the PORT from which the corresponding RSVP was done. Whether or not a reply is ever received at the PORT wher **e the original RSVP was done depends on two things: 1) the Replymask parameter to RSVP, and 2) the type assigned to the message at the time the REPLY operation is done (usually by some other PROCESS.)

If the bit in Replymask corresponding to the type of the mess **age is 1, then the reply will be received at the PORT from which the **RSVP was done; if not, the PORT from which the RSVP was done will be bypassed during the REPLY operation and some other PORT (or none) will receive the reply. Thus, the only way to guarentee that a reply will be received at the PORT where the RSVP was done is to specify a Replymask of #177777 (octal). Then the PORT cannot be bypassed no matter what type is assigned to the message at the time the REPLY operation is done. (See REPLY for more details.)

A reply to an RSVP-message may or may not return to the originating PORT, but if it does, it must arrive through an input channel. The Inchan parameter allows the sender of an RSVP to specify which input channel any reply will return to. By turning on bit number Inchan in the channel-mask of a subsequent RECEIVE operation, the user can receive the reply.

In some applications it is essential to be able to keep track ** of individual messages and associate replies with the original rsvp.

The Messid parameter allows this bookkeeping to be done reliably.
Whatever argument is passed as Messid is used as a "name"
which stays with the message until the reply is received. When
**a
reply is received the original Messid is returned as part of the
message description. (See RECEIVE for more information.)
The Messid parameter is completely uninterpreted by the Kernel, s
**o the
user is permitted to devise any bookkeeping system he wishes (or
**none.)
There is no way that any subsequent handling of the message can
disturb this identification.

For more information related to RSVP, see the descriptions of
SEND, REPLY and RECEIVE.

Signals: All signals from RSVP have SGRSVP in bits 6-10 and one
of the following in bits 0-5:

SGMLNAMERANGE	- Lname is negative or out of range for this PORT.
SGMLNAMEFREE	- Local name Lname is free, i.e. assigned to no message.
SGMOCHANRANGE	- Outchan is negative or out of range for this PORT.
SGMUNCONNECTED	- Output channel Outchan is not connected.
SGMTYPERANGE	- Type is not in the range 0-15 inclusive.
SGMICHANRANGE	- Inchan is negative or out of range for this PORT.
SGMSTACKOVFL	- Reply stack overflow; no more room in the reply stack of this message.

Result: 0

REPLY (Port, Lname, Type)

Parameters:

PORT - Simple Index of a PORT object; CNFRTS; REPRTS
Lname - Integer index (local name) of the message to be REPLYed.
Type - Type to be assigned to the message.

Effect: The REPLY operation is used to delete a message or to return
**it
to some PORT where a previous RSVP operation was done to the mess
**age.
A record of those PORTs where an RSVP was done to the message and
** the
criteria for receipt of a reply at those PORTs
is carried around with the message in its reply-stack. Each tim
**e an
RSVP is done to the message one stack frame is pushed onto the
message's reply-stack, and each time a REPLY operation is done, o
**ne
or more frames are popped from the reply-stack. Thus, at any

given instant the reply-stack contains frames corresponding to exactly those PORTs which are be eligible to receive replies.

The REPLY operation proceeds in detail as follows:

1) The value of the parameter Type is assigned to be the type of the message with local name Lname.

2) Each reply-stack frame in the message is examined, starting naturally from the stack-top, to see if the c **urrent message is among those that were specified in the Repl **ymask parameter to the original RSVP operation. (See RSVP.)

3) If not, the reply-stack frame is popped and the examin **ation of frames continues. The PORT associated with the po **pped frame is "bypassed" and never receives a reply.

4) If so, however, the examination of frames stops. The ** message is "sent" to the PORT associated with the repl **y- stack frame through the input channel specified in the Inchan parameter to the original RSVP operation. (Se **e RSVP.) There the message will either be enqueued or will be immediately received by a blocked PROCESS, jus **t as if the message had been sent using SEND. (The last reply-stack frame examined is also popped.)

5) If all frames are popped without finding a PORT eligib **le to receive the reply, then the message is destroyed. Th **is is the only way a message can be deleted under Hydra; the **re is no MDELETE Kall.

Signals: All signals from REPLY have SGREPLY in bits 6-10 and one of the following in bits 0-5:

SGMLNAMERANGE - Lname parameter is negative or out of range for t **his PORT.
SGMLNAMEFREE - Local name Lname is free, i.e. is assigned to no message.
SGMTYPERANGE - Parameter Type is not in the range 0-15.

Result: 0

```
  RECEIVE ( Port, Cond, Waitclass, Mask, Descr )
Parameters:
        PORT      - Simple Index of PORT object; CNFRTS; MRECRTS
        Cond      - Boolean; true if RECEIVE is conditional, i.e. blockin
                              **g
                    not allowed; false if RECEIVE is unconditional and bl
                              **ocking
                    is permitted.
        Waitclass - Boolean; true if specifying messages by input channel
                              **s;
                    false if specifying messages by type.
        Mask      - 16 bit mask specifying either a set of input channels
                    or a set of types (depending on the Waitclass paramet
                              **er.)
                    Bits are numbered 0-15 from least to most significant
                              **.
        Descr     - Legitimate Stack Memory Address of an area at least s
                              **ix
                    words;  RECEIVE fills this area with a description of
                              ** the
                    received message.   (See format below.)

Effect:    RECEIVE is the basic message-receive primitive of the PORT sys
                              **tem.
        The user passes a description of the class of messages he wishes
                              **to
        receive, and the Kernel either immediately returns access to such
                              ** a
        message, or it blocks the PROCESS until such a message is availab
                              **le.
        If a message is received, a more detailed description of it
        is placed in the user's stack area at Descr so that
        he may know what kind of message he has received.
        The events in more detail are as follows:
            The two parameters Waitclass and Mask form the description of
                              **
        the class of messages the user wishes to receive.   He may either
        receive a message which has one of a set of message types, or he
        may elect to receive a message that arrives via any one of a set
        of input channels.   The choice between type-specification and ch
                              **annel-
        specification is made through the Boolean parameter Waitclass.
            The set of channels or types is specified by the parameter Ma
                              **sk.
        Bits 0-15 of the mask specify either channels 0-15 or types 0-15
        (depending on Waitclass.)   Thus, if Waitclass = 1 and Mask = #03
                              **0777
        then only a message which arrives through one of the channels 0-8
        or 12-13 will be received.    Any one-bits in Mask which correspo
                              **nd
        to channel indices greater than those allowed for the PORT in
        question are ignored.
```

The Waitclass and Mask parameters form a description of a cla
**ss
of messages but do not specify a particular message. Thus, ther
**e may be many
messages enqueued which fit the description at the time a
RECEIVE is done. The user has no control over which of the elig
**ible
messages will be received beyond what have already been described
** under
the Waitclass and Mask parameters. In particular, he has no way
**of
giving "priority" to certain channels or types. Messages are
selected by the Kernel for receipt subject to only two
restrictions:

1) Messages will be received in FIFO order within any giv
**en
type or any given input channel.
2) Type and channel queues will be scanned according to a
"fair" policy, so that no input channels or types will
be systematically ignored across many RECEIVE operatio
**ns.

The Cond parameter specifies whether or not the RECEIVE opera
**tion
is "conditional", i.e. whether or not the PROCESS doing the RECEI
**VE
is permitted to block. If Cond is true (odd) then no blocking i
**s
permitted. Thus, if a message fitting the Waitclass-Mask
description is available, it will be received; if not, no message
** will
be received, and a signal will be generated.
However, if Cond is false (even) then blocking is permitted.
If no satisfactory message is available the PROCESS will be suspe
**nded
until one arrives. (Actually the Kernel
doesn't "suspend" the PROCESS; it stops the PROCESS and
notifies the POLICY system not to reschedule it until further
notice. An erroneous POLICY system may schedule the PROCESS
anyway, but the Kernel will immediately re-stop it and once again
notify the POLICY system not to reschedule it.)
When a message is received a detailed description of the
message is is placed in the six-word area that the user provides
through the parameter Descr. The format of this
six word area, and the interpretation of the fields are as follow
**s:

```
----------------------------------
!          LNAME                 !
----------------------------------
!R!               ! TYPE  !INCHAN !
----------------------------------
!          LENGTH                !
```

```
                  ---------------------------------
                  !          BUFFLENGTH           !
                  ---------------------------------
                  !          MESSID               !
                  ---------------------------------
                  !          CONNID               !
                  ---------------------------------

                  LNAME      - The local name assigned to the received mess
                                  **age.
                  R          - Reply-bit: 1 if the message is a reply to an
                               earlier RSVP; 0 if it is a normal unsolicted
                               arriving message.  This field is the only
                               way to distinguish replies from non-replies.
                  LENGTH     - The length (in bytes) of the text in the mes
                                  **sage
                               buffer.
                  BUFFLENGTH - The length (in bytes) of the message buffer.
                                  ** Must
                               be greater than or equal to LENGTH.
                  MESSID     - If this message is a reply, MESSID contains
                               the message-id assigned to this message at t
                                  **he
                               time the RSVP was done.  (See RSVP.)
                  CONNID     - If this message is not a reply, CONNID conta
                                  **ins
                               the connection-id of the connection through
                                  **which
                               the message arrived.  This gives the receiv
                                  **er of
                               a message some idea of where the message cam
                                  **e
                               from.  (See CONNECT for a discussion of
                               the idea of a connection-id. )

Signals:    All signals from RECEIVE have SGMRECEIVE in bits 6-10 and
        one of the following vales in bits 0-5:

        SGMNOFREELNAME - Lname is negative or out of range for this PORT.
        SGMPACKADR     - Packadr is not a Legitimate Stack Memory Address
                                  ** of
                         a six word area.
        SGMCONDRECFAIL - The Cond parameter indicates a conditional recei
                                  **ve,
                         but no satisfactory message is available.

Result: RECEIVE normally returns the local name assigned to the received
        message.
↑L
```

.SEC |User I/O Operations|

.SUBSEC |Overview from a Subsystem Builder's Viewpoint|

In order to perform input/output operations, the subsystem must connect a port to an i/o device. This action is performed by means of the message system's PCONNECT operation, described in [ref]. After a connection has been established successfully, the i/o device identified by the specified object is available for exclusive use through the given port and output channel, and such exclusive access remains effective until disconnection (see PDISCONNECT). All future operations specify the i/o device indirectly, by way of the port and output channel to which it is connected, and the i/o device object is of no further use.

The i/o device object may also be used to request reconfiguration, but this is a specialized use which is documented in a separate section ([ref]).

.SUBSEC |Overview from a User Program's Viewpoint|

A user program performs i/o operations in exactly the same manner as it sends messages via the message system (see [ref]). In fact, there is no way to determine whether an output connection is to an i/o device or to another port. A user program merely sends messages of a prescribed format (see [ref]) and waits for a reply, if appropriate. The information in the message specifies the requested operation, and the reply type indicates the outcome of the request. All message system primitives for sending messages and obtaining replies are equally applicable to i/o requests.

.SUBSEC |Conventions|

All i/o messages (henceforth referred to as requests) contain at least an operation code indicating the specific action to be taken. Most requests also include a buffer, a byte count, and some device parameters (e.g. a sector address for a disk transfer). This section outlines the conventions which govern the format of i/o requests, leaving details of specific operations for the next section.

The operation code is the first word of every i/o message. It is subdivided into three fields: optype, opcode, and opformat. The optype places the request into one of four general categories. Immediate operations require no action by the device itself. Control operations affect the device, but no data transfer occurs (e.g. tape rewind). Input operations transfer one or more bytes of data from the device to memory; output operations transfer data from memory to the device.

The opcode field determines the particular operation to be performed within a given class. For many devices, only one operation of each class will be defined; however, some devices may have several. For example, a DECtape has two control operations, rewind and findblock. The optype and opcode fields together define a unique logical operation, which may correspond to zero, one, or more physical operations on the device.

The opformat field provides format information about the i/o request itself and does not directly influence the operation. It is broken down into individual bits which specify the existence or nature of other fields in the request. Not all of these bits may be relevant to a particular operation, and some operations may outlaw certain format settings -- consult the descriptions of the specific actions for details.

The general i/o request assumes the following form:

```
+-------------------------------+
!                               !
!           Operation           !
!                               !
+-------------------------------+
!                               !
!          Buffer Size          !
!                               !
+-------------------------------+
!                               !
!                               !
!                               !
+                               +
!                               !
!             Buffer            !
!                               !
+                               +
!                               !
!                               !
!                               !
+-------------------------------+
!                               !
!       Device Parameters       !
!                               !
+                               +
!                               !
!                               !
!                               !
+-------------------------------+
```

The operation field has already been discussed. The buffer size field is normally required only for transfer operations, and holds the number of bytes of data to be transferred. Some devices (e.g. teletype) allow the buffer size field to be omitted on some transfer operations; in such cases the omission is indicated by a bit in the opformat field. The buffer area is of the size specified by the byte count and is required for all operations which transfer data. The buffer is normally contained within the message itself, but may be specified indirectly as an address within the requesting lns's address space (cps -- see [ref]). In this case, a format bit is set in the opformat field and the buffer address is a two word quantity whose first word is a cps index and whose second word is a 13-bit displacement. The device parameters field is operation dependent and for sequential devices is usually omitted. It frequently contains positioning information for read/write heads, but may specify auxiliary information for any of the four optype classes.

The outcome of i/o requests is reported via the message system message type [ref], which summarizes the result of the operation. If additional information is necessary to define the outcome, it will be appended to the message following the last word supplied by the requestin
**g
process. No information in the message itself (except possibly the buffer during an input operation) is ever altered during an i/o operation
**.
Thus the contents of a failing request may be examined to determine the cause of the error. A single type, OPDONETYPE, indicates a successful completion, while other reply types are used to denote errors. The specific reply codes are discussed later.

.SUBSEC |Specific Device Operations|

This section describes the operations which are permitted for each of the several device classes supported. It should be noted that the values for specific fields are given symbolically rather than as absolute numeric quantities. The equivalences are established by use of the BLISS/11 "require" file UIO.REQ[N810HY00], which should always
** be used
by user programs.

1) Operations common to all devices

A limited number of operations are defined to have a common action for all devices.

a) DIDENTIFY

Class: Immediate
Format restrictions: not applicable
Byte Count: not used
Buffer: not used
Device Parameters: not used
Other Information: returns static information pertaining to the device in the words immediately following the operation code as follows:

```
+---------------+---------------+
!               !               !
!     PNUM      !     CTYPE     !
!               !               !
+---------------+---------------+
!                               !
!       Registers Address       !
!                               !
+-------------------------------+
```

```
!                               !
!   Interrupt Vector Address    !
!                               !
+---------------+---------------+
!/ / / / / / / /!               !
! / / / / / / / !  Unit Number  !
!/ / / / / / / /!               !
+---------------+---------------+
```

CTYPE   CONTROLLER TYPE
PNUM   PROCESSOR NUMBER

b) DSTATUS

Class: Control
Format restrictions: not applicable
Byte Count: not used
Buffer: not used
Device Parameters: not used
Other Information: returns device-specific dynamic status information in the word(s) immediately following the operation code

2) Line Frequency Clock

a) KWWAIT

Class: Control
Format restrictions: not applicable
Byte Count: not used
Buffer: not used
Device Parameters: a one-word count (treated as an unsigned integer) denoting the number of 1/60 second clock ticks ("jiffies") wwich are to elapse before a reply occurs.

b) DSTATUS

<< not yet specified>>

3) Line Printer

a) LPWRITE

Class: Output
Format restrictions: byte count is required
Byte Count: must be even - rounded up if not
Buffer: if last word is not full, high order (odd) byte should contain a pad of binary 0
Device Parameters: none

The data contained in the buffer are transferred to the line printer, with a reply occurring upon completion of the transfer. The buffer should normally end with a line terminating character

(e.g. line feed, form feed, vertical tab, carriage return, form feed, ↑S)

b) DSTATUS

<< not yet specified>>

4) Teletype

a) TTREAD

Class: Input
Format restrictions: none
Byte Count: optional, as per format specification
Buffer: required
Device Parameters: none

When a complete line of input is available in the terminal' **s input buffer, it will be copied into the user's buffer. A line is defined as a sequence of zero or more characters followed by a break character. Break characters are: line feed, carriage return, ↑A, ↑B, ↑C, ↑G, ↑K, ↑L, ↑Z, altmode. A t **yped carriage return causes both the carriage return and a generated line feed to enter the buffer.

Rubout, ↑U, and limited type-ahead are handled by the teletype support in a manner analagous to the PDP-10. No break character definition, image mode, or full character set mode is available, nor will any of the above be provided until the terminal front-end system is completed. The existing teletype support is an interim stopgap package.

If the user-supplied buffer is inadequate to hold an entire typed line, as much of the line as will fit is supplied and a special reply is used (OPDONETYPE + LOSTINFOTYPE). The remainder of the input line is retained in the terminal's input buffer, and is supplied on the next input operation.

The terminal input buffer has a capacity of approximately 120 characters.

If the user-supplied buffer resides within the i/o message itself, the size of the replied message can be used to determine the length of the line returned. If the buffer is specified indirectly, the break character which terminates the line is the only indication of line length.

b) TTWRITE

Class: Output
Format restrictions: none

Byte Count: optional, as per format specification
Buffer: required
Device Parameters: none

If the byte count is omitted, the buffer is assumed to contain an ASCIZ string to be transmitted to the terminal. An explicit byte count causes the specified number of characters to be transmitted, including nulls if present. It is important to note that if an indirect buffer specification is used, the buffer must not be changed while the i/o request is in progress, since the output data is taken directly from the user's buffer. An attempt to do so will result in indeterminate output. This presents no restriction if the buffer is contained within the message itself, since the user will be unable to alter the message while the i/o system is processing it.

c) DSTATUS

<< not yet specified>>

d) TTECHOCTL

Class: control
Format restrictions: not applicable
Byte Count: not used
Buffer: not used
Device Parameters: The low order bit of the word following the operation code determines whether echoing is performed (bit = 1) or not (bit = 0).

e) TTOUTRESET

Class: control
Format restrictions: not applicable
Byte Count: not used
Buffer: not used
Device Parameters: none

The successful execution of this operation causes all queued output requests, including the currently executing one, to be aborted (reply ERRTYPE). All program-generated output is thus canceled. Any pending echo characters are not affected by this operation.

f) TTINRESET

Class: control
Format restrictions: not applicable
Byte Count: not used
Buffer; not used
Device Parameters: none

The successful execution of this operation causes all pending input requests to be aborted (reply ERRTYPE). In addition, if any complete or partial lines are present in the terminal input buffer, they are deleted. However, any pending echo for characters in the input buffer will be allowed to proceed.

g) TTINCLEAR

Class: control
Format restrictions: not applicable
Byte Count: not used
Buffer: not used
Device Parameters: none

This operation causes any pending input requests to be aborted (reply ERRTYPE). In addition, an implied ↑U is issued -- any partial line in the input buffer is deleted. Complete lines in the input buffer will be preserved, as will any pending echo characters.

h) TTEXCP

Class: control
Format restrictions: not applicable
Byte Count: not used
Buffer: not used
Device Parameters: not used

Only one TTEXCP request may be pending on a terminal at a time; any attempt to issue a second one will cause an immediate reply of ERRTYPE. TTEXCP remains pending until an unusual condition occurs, at which time a successful reply occurs and a word of information is returned in the location immediately following the TTEXCP opcode. If an unusual condition is detected when no TTEXCP request is pending, it is ignored. The conditions are:

TTSAWBREAK - break key was hit
TTLOSTDATA - input rate too great
TTSAWCTLO - ↑O typed

5) DECTAPE

a) TCSETUNIT

Class: Immediate
Format restrictions: not applicable
Byte Count: not used
Buffer: not used
Device Parameters: a unit number between 0 and 7 inclusive in the word following the opcode.

If the specified unit number is available, it is allocated to the device, otherwise, the reply type REQILLDP is generated. When the DECtape connection is initially established (via PCONNECT), a unit number is allocated to it. Hence, TCSETUNIT

need not be issued unless the initial unit number is unsatisfact
**ory.
This initial value may be determined by using the DIDENTIFY operation.

b) TCREWIND

Class: control
Format restrictions: not applicable
Byte Count: not used
Buffer: not used
Device Parameters: none

The specified device is rewound to the forward end-zone, with the reply being generated upon detection of the end-zone.

c) TCFINDBLOCK

Class: control
Format restrictions: not applicable
Byte Count: not used
Buffer: not used
Device Parameter: a one-word value specifying the block at which the tape is to be positioned.

The tape is positioned so that an immediately following TCREAD or TCWRITE specifying the same block number will experience minimum positioning delay. If the block number cannot be found on the tape, an error reply will occur (reply type ERRTYPE).

d) TCREAD

Class: Input
Format restrictions: byte count required
Byte Count: should be even -- rounded up if not
Buffer: required
Device Parameter: a one-word value specifying the block at which reading is to begin

If the specified block cannot be found, an error reply occurs (ERRTYPE). Otherwise, input begins at the specified block and continues (in a forward direction) until the count is exhausted. Any "soft" error is retried five times before reporting the failure.

e) TCWRITE

Class: Output
Format restrictions: byte count required
Byte Count: should be even -- rounded up if not
Buffer: required
Device Parameter: a one-word value specifying the block at which writing is to begin

Identical to TCREAD, but performs output instead of input.

DECtape errors:

Reply type ERRTYPE causes a single word of error information to be appended to i/o message. This type can be generated for TCREWIND, TCFINDBLOCK, TCREAD, and TCWRITE **.

This word contains the value of the controller's status register (TCST) at the time the error occurred. Refer to Peripherals Manual for specific bit interpretations.

6) RP11 (moving head disk)

a) RPSEEK

Class: Control
Format restrictions: not applicable
Byte Count: none
Buffer: not used
Device Parameters: two words of disk addressing information, in a format described below

A seek operation is performed to position the read/write heads at a specified cylinder and track. No data transfer occurs. If the seek cannot be successfully performed, a reply with type ERRTYPE is generated, and error status information is returned in the message immediately following the device parameters.

b) RPREAD

Class: Input
Format restrictions: byte count required
Byte Count: should be even -- rounded up if not
Buffer: required
Device Parameters: two words of disk addressing information, in a format described below

The device parameters are used to seek the proper starting sector address. An input operation is then initiated which continues until the byte count has been exhausted. The transfer may involve more than one sector, and may cross track or cylinder boundaries. Error recovery is attempted, and "hard" errors are reported in the same way as for RPSEEK. See notes below for specifics.

c) RPWRITE

Class: Output
Format restrictions: byte count required
Byte Count: should be even -- rounded up if not
Buffer: required

Device Parameters: two words of disk addressing information in a format described below.

Identical to RPREAD except that output is performed instead of input.

d) RPWRITECHECK

Class: Output
Format restrictions: byte count required
Byte Count: should be even -- rounded up if not
Buffer: required
Device Parameters: two words of disk addressing information in a format described below

Identical to RPWRITE except that data from memory is compared to data at the specified disk address. No data is actually written on the disk. If a comparison error occurs, an error reply (ERRTYPE) will occur, as described below.

Notes on RP11 i/o programming:

1. Device parameters take the following form:

```
+--------------------------------+
!                                !
!            Cylinder            !
!                                !
+---------------+----------------+
!               !                !
!    Sector     !     Track      !
!               !                !
+---------------+----------------+
```

Sector is not required for RPSEEK.

2. When an unrecoverable error occurs, the reply is of type ERRTYPE, and two words of error status information are returned in the message. The first of these is the contents of RPER at the time of the error; the second contains the value of RPDS. Refer to peripherals manual for a description of the individual error bits.

3. Seek and transfer errors are retried five times by the disk software before they are considered "hard" and reported to the user program. Thus no further error recovery need be attempted upon receipt of an ERRTYPE reply.

7) ASLI Link (to another computer)

a) KLSETSPEED

Class: Control

```
          Format restrictions:  not applicable
          Byte count:  not used
          Buffer:  not used
          Device parameters:  one word containing line speed inform
                              **ation

   The parameter word contains a value in the range 0-7 in each of
                              **its
   bytes.  The even byte specifies the line input speed; the odd by
                              **te
   specifies the output speed.  The values have the following inter
                              **pretations:

          0                   110  Baud
          1                   134.5
          2                   300
          3                   600
          4                   1200
          5                   2400
          6                   4800
          7                   9600

   The line is initialized to 4800 baud in, 300 baud out.  These va
                              **lues
   are suitable for PDP-10 communication.

   b)  KLASCIIREAD

          Class:  Input
          Format restrictions:  byte count required
          Byte count:  required
          Buffer:  required
          Device parameters:  none

   An input line of ASCII characters is assembled and placed in the
                              ** buffer.
   If the buffer is of insufficient size to hold the entire line, t
                              **he number
   of characters specified by the byte count is returned and LOSTIN
                              **FOTYPE
   is indicated with OPDONETYPE.  No buffering is performed by the
                              **interrupt
   routine; hence, any characters which arrive when no i/o request
                              **is in
   effect will be discarded.  The line break characters are the sam
                              **e as for
   TTREAD.  If a hardware error is detected (break, lost data, etc.
                              **), ERRTYPE
   will be indicated in the reply code and the value of the input s
                              **tatus
   register will be returned in the word following the buffer.

   c)  KLBINARYREAD
```

```
                Class:  Input
                Format restrictions:  byte count required
                Byte count:  required
                Buffer:  required
                Device parameters:  none

         Identical to KLASCIIREAD except that 8-bit characters are return
                                **ed and
         no break character processing is performed.  Thus exhaustion of
                                **the byte
         count is the only terminating condition, and LOSTINFOTYPE is not
                                ** indicated
         with OPDONETYPE.  A request specifying KLBINARYREAD will remain
                                **pending
         until the specified number of characters have been input.

         d)  KLWRITE

                Class:  Output
                Format restrictions:  byte count required
                Byte count:  required
                Buffer:  required
                Device parameters:  none

         Outputs the specified number of 8-bit characters.  The character
                                **s are
         not interpreted in any way by the interrupt routine, so that any
                                ** 8-bit
         character is legal and will be transmitted unchanged.

.SUBSEC |Reply Codes|

     The i/o system generates a number of reply codes wich describe
the outcome of a request.  They are described in this section.

     REQDEVDOWN - The device is no longer on-line.

     REQTOOSMALL- The i/o request does not contain all of the
                    information required by the i/o system.

     REQBADBUF  - The buffer specification is illegal for one of
                    several reasons:
                       a)  illegal cps slot
                       b)  input operation and write-protected page
                       c)  zero or negative byte count
                       d)  buffer either crosses a page boundary
                           or is too large for message

     REQILLFMT  - Illegal format for specified opcode or unrecognized
     REQILLOP     opcode.

     OPDONETYPE    - Normal completion.

     ERRTYPE       - Error completion.
```

In the event that completion (normal or error) occurs but not all of the desired information can be supplied (e.g. ERRTYPE return, but request is too small to hold error information), the value LOSTINFOTYPE is added to either ERRTYPE or OPDONETYPE to warn the program that not all the expected information is present.

.SUBSEC |Format Modifiers|

Two format modifiers are defined, INDBUF and NOCOUNT. INDBUF specifies that the buffer is addressed indirectly, as described earlier. NOCOUNT is used to indicate that the byte count has been omitted. These modifiers are ignored when used with operations which do not require a buffer.

To use a format modifier, the user program employs the IOOPN macro to define a composite operation code, e.g.

```
IOOPN(TTREAD,INDBUF+NOCOUNT)
```

↑L

.SEC |THE APPENDIX|

Except where necessary, absolute values and locations for fields are not given in this manual. The bindings for all symbolics may be found in the file HYKALL.R11[N810HY00] @ CMU-10A.

.SUBSEC |HYDRA KERNEL RIGHTS|

In describing Hydra Kernel Rights, we consider the effect if Capability CAP has the right in question. If CAP is an Object Reference, we write OBJ as a shorthand for the Object Referenced by CAP:

LOADRTS - Allows a Capability to be Loaded from OBJ
STORTS - Allows a Capability to be Stored into OBJ
APPRTS - Allows a Capability to be Appended onto OBJ
KILLRTS - Allows a Capability to be Deleted from OBJ

GETRTS - Allows data to be gotten from OBJ
PUTRTS - Allows data to be put into OBJ
ADDRTS - Allows data to be appended onto OBJ

ALLYRTS - Allows OBJ to be Re-Allyed
OBJRTS - Allows OBJ to be Switced or Frozen

CREARTS - Allows an Object to be Created from CAP
COPYRTS - Allows a Copy to be made of OBJ

DLTRTS - Allows CAP to be Deleted
ENVRTS - Allows CAP to be Stored in some Object
MDFYRTS - Allows OBJ to be modified
UCNFRTS - Allows OBJ to be Unconfined, that is, an Object
  accessed through OBJ may be modified.
FRZRTS - Guarantees that OBJ is Frozen

Note that the last set of 5 rights cannot be gained through rights amplification. Note that whenever rights are restricted, ALLYRTS are always removed as well.

.SUBSEC |RIGHTS RESTRICTION FORMAT|

```
+--'--'--'--'--'--'--'--+--+--+--'--+--+--+--+
!                       !  !  !              !
!        AUXRTS         !NF!TF!    UNUSED    !
!         (8)           !  !  !     (6)      !
+--'--'--'--'--'--'--'--+--+--+--'--'--'--'--+
!                                            !
!                 KERNEL RIGHTS              !
!                     (1W)                   !
```

```
+--'--'--'--'--'--'--'--+--'--'--'--'--'--'--+
```

AUXRTS - Auxiliary rights
NF - NEWFLAG
TF - TMPLFLAG

Kalls that allow restriction of rights and flags (the flags fields, NF and TF are ignored in restricting an Object Reference) require an address that must point to a location in the active stack. That location is a two word area formatted as shown above. If the bit representing the particular Kernel or Auxiliary right or Flag is 0, the right or flag will be restricted.

Example, if the MUCH'th slot contained some Capability for a Procedure, to get a Capability for the same Procedure in the LESS'th slot having only CALLRTS, LNSRTS and DLTRTS, the following Bliss-11 code would do:

```
Begin
Local RESTR[2];
RESTR[0] ← CALLRTS or LNSRTS;
RESTR[1] ← DLTRTS;
Share ( LESS, MUCH, RESTR )
End
```

.SUBSEC |SIZE RESTRICTIONS|

The maximum size of a Data-Part is 1000 (#1750).
The maximum number of Capabilities in a C-List is 125 (#175).

.SUBSEC |KERNEL TYPES|

For each Kernel Type, we specify a number of things:

a) Defined Auxiliary rights
b) Initialization rights & flags - At system initialization, the initial Policy Subsystem has been provided with a Template with these rights and flags (NEWFLAG & TMPLFLAG).
c) Template rights and flags - The rights of a Template returned from the TEMPLATE Kall.
d) Copy rights - The rights added when a Capability of that type is copied.
e) Creation arguments - Additional arguments to the CREAT Kall.
f) Copy arguments - Additional arguments to the COPY Kall.

\- - - - - - -

1) Type TYPE

a) Auxiliary:
TMPLRTS - Allows Template of named Type to be made with all

rights and flags.
RTRVRTS - Allows TYPRETRIEVE Kall

b) Initialization:
LOADRTS, STORTS, APPRTS, KILLRTS, OBJRTS, CREARTS, COPYRTS, DLTRTS, ENVRTS, UCNFRTS, MDFYRTS, TMPLFLAG, All Auxiliary rights

c) Template:
DLTRTS, ENVRTS, TMPLFLAG

d) Copy:
DLTRTS

e) Creation arguments:
Address (in stack) of 16 word area containing
PNAME - words 1-5, Print Name
CAPINIT - word 6, Initial C-List size of CREATed Object
CAPMAX - word 7, Maximum C-List size
DATAINIT - word 8, Initial Data-Part size
DATAMAX - word 9, Maximum Data-Part size
RTRVFLAG - word 10, Retrievability flag in sign bit.

f) Copy arguments:
Same as Creation argument.

- - - - - - -

2) Type NULL

a) Auxiliary:
NULLRTS - Determines whether Capability is Truenull

b) Initialization:
All Kernel and auxiliary rights, TMPLFLAG. Note though that it is impossible to CREAT a Capability for a Null Object.

c) Template:
All Kernel and Auxiliary rights, TMPLFLAG.

d) Copy: May not be COPYed

e) Creation arguments: May not be CREATed

f) Copy arguments: May not be COPYed

- - - - - - -

3) Type PROCEDURE

a) Auxiliary:
GETCBRTS - Allow access to ICB
SETCBRTS - Allow modification of ICB
PRCSRTS - Allows LNS incarnated from Procedure to initialize a Process

LNSRTS - Allows LNS incarnated from Procedure to be LNSCALLed.
CALLRTS - Allows Procedure to be CALLed.

b) Initialization:
LOADRTS, STORTS, APPRTS, KILLRTS, OBJRTS, CREARTS, COPYRTS, DLTRTS, ENVRTS, UCNFRTS, MDFYRTS, TMPLFLAG, All Auxiliary rights

c) Template:
LOADRTS, STORTS, APPRTS, KILLRTS, OBJRTS, CREARTS, COPYRTS, DLTRTS, ENVRTS, MDFYRTS, TMPLFLAG, All Auxiliary rights

d) Copy:
DLTRTS

e) Creation arguments: None

f) Copy arguments: None

\- - - - - - -

4) Type LNS

a) Auxiliary:
GETCBRTS - Allows access to LCB
SETCBRTS - Allows modification to LCB
GSTKRTS - Allows access to LNS's active stack
PSTKRTS - Allows modification of LNS's active stack
PRCSRTS - Allows LNS to initialize a Process
LNSRTS - Allows LNS to be LNSCALLed.

b) Initialization:
DLTRTS, ENVRTS, TMPLFLAG.

c) Template:
DLTRTS, ENVRTS, TMPLFLAG.

d) Copy: May not be COPYed

e) Creation arguments: May not be CREATed (See MAKLNS)

f) Copy arguments: May not be COPYed

Note: LNS Capabilities created with MAKLNS have the following rights: DLTRTS as well as UCNFRTS, FRZRTS, LNSRTS & PRCSRTS only if the Procedure it was incarnated from had those rights.
LNS Capabilities created via the "Lns" argument specification for CALL have the following rights: LOADRTS, STORTS, APPRTS, KILLRTS, DLTRTS, MDFYRTS, GETCBRTS, SETCBRTS, GSTKRTS & PSTKRTS.

\- - - - - - -

5) Type POLICY

a) Auxiliary:
MAKERTS - Allows the MAKEPOLICY Kall
RCVRTS - Allows the RCVPOLICY Kall
POLRTS - Allows the POLICY Kall

b) Initialization:
LOADRTS, STORTS, APPRTS, KILLRTS, CREARTS, DLTRTS, ENVRTS, UCNFRTS, MDFYRTS, TMPLFLAG, All Auxiliary rights

c) Template
DLTRTS, ENVRTS, TMPLFLAG

d) Copy: May not be COPYed

e) Creation arguments:
One word indicating information about Policy Subsystem and its status

f) Copy arguments: May not be COPYed

\- - - - - - -

6) Type PROCESS

a) Auxiliary:
GETCBRTS - Allows access to PCB
SETCBRTS - Allows modification to PCB
STARTS - Allows the START Kall
STOPRTS - Allows the STOP Kall
CTLRTS - Allows the CONTROL Kall
SYNRTS - Allows the DESYNCH Kall
BASERTS - Allows association of Process Base in POLICY Kall
POLRTS - Allows association of Policy in POLICY Kall

b) Initialization:
CREARTS, DLTRTS, ENVRTS, UCNFRTS, MDFYRTS, TMPLFLAG, All Auxiliary rights

c) Template:
CREARTS, DLTRTS, ENVRTS, UCNFRTS, MDFYRTS, TMPLFLAG, All Auxiliary rights except BASERTS

d) Copy: May not be COPYed

e) Creation arguments:
Simple index denoting a Capability for an LNS Object with PRCSRTS. The LNS must be "useable" (See Subsection on PROCESS OBJECTS)

f) Copy arguments: May not be COPYed

\- - - - - - -

7) Type PAGE

a) Auxiliary:
CPSRTS - Allows Page to be loaded into CPS
PGWRTS - Allows Page to be written into
CACHRTS - Allows Page to be cached

b) Initialization:
OBJRTS, CREARTS, COPYRTS, DLTRTS, ENVRTS, UCNFRTS, MDFYRTS, TMPLFLAG, All Auxiliary rights

c) Template:
OBJRTS, CREARTS, COPYRTS, DLTRTS, ENVRTS, UCNFRTS, MDFYRTS, TMPLFLAG, CPSRTS, PGWRTS

d) Copy:
OBJRTS, DLTRTS, ENVRTS, UCNFRTS, MDFYRTS, CPSRTS, PGWRTS

e) Creation arguments: None

f) Copy arguments:
Index of a CPS slot. The COPYed PAGE will be CPSLOADed into that CPS slot.

\- - - - - - -

8) Type SEMAPHORE

a) Auxiliary: None

b) Initialization:
CREARTS, DLTRTS, ENVRTS, UCNFRTS, MDFYRTS, TMPLFLAG

c) Template:
DLTRTS, ENVRTS, TMPLFLAG

d) Copy: May not be COPYed

e) Creation arguments:
Initial value of Semaphore

f) Copy arguments: May not be COPYed

\- - - - - - -

9) Type POLSEM

a) Auxiliary:
PRTS - Allows the PPOLSEM Kall
VRTS - Allows the VPOLSEM Kall
CPRTS - Allows the CPOLSEM Kall

b) Initialization:
CREARTS, DLTRTS, ENVRTS, UCNFRTS, MDFYRTS, TMPLFLAG, All Auxiliary rights

c) Template:
DLTRTS, ENVRTS, TMPLFLAG

d) Copy: May not be COPYed

e) Creation arguments:
Initial value of the Policy Semaphore

f) Copy arguments: May not be COPYed

\- - - - - - -

10) Type DATA

a) Auxiliary: None

b) Initialization:
GETRTS, PUTRTS, ADDRTS, OBJRTS, CREARTS, COPYRTS, DLTRTS, ENVRTS, UCNFRTS, MDFYRTS, TMPLFLAG

c) Template:
GETRTS, PUTRTS, ADDRTS, OBJRTS, CREARTS, COPYRTS, DLTRTS, ENVRTS, UCNFRTS, MDFYRTS, TMPLFLAG

d) Copy:
GETRTS, PUTRTS, ADDRTS, OBJRTS, DLTRTS, ENVRTS, UCNFRTS, MDFYRTS

e) Creation arguments: None

f) Copy arguments:
Length of Data-Part of COPYed Object. The Data-Part of the COPYed Object will be expanded or contracted as necessary. If less than or equal to 0, the length will be as in the original.

\- - - - - - -

11) Type UNIVERSAL

a) Auxiliary: None

b) Initialization:
LOADRTS, STORTS, APPRTS, KILLRTS, GETRTS, PUTRTS, ADDRTS, OBJRTS, CREARTS, COPYRTS, DLTRTS, ENVRTS, UCNFRTS, MDFYRTS, TMPLFLAG

c) Template:
LOADRTS, STORTS, APPRTS, KILLRTS, GETRTS, PUTRTS, ADDRTS, OBJRTS, CREARTS, COPYRTS, DLTRTS, ENVRTS, UCNFRTS, MDFYRTS, TMPLFLAG

d) Copy:
LOADRTS, STORTS, APPRTS, KILLRTS, GETRTS, PUTRTS, ADDRTS, OBJRTS, DLTRTS, ENVRTS, UCNFRTS, MDFYRTS

e) Creation arguments: None

f) Copy arguments:
Same as for DATA.

\- - - - - - -

12) Type PORT

a) Auxiliary:
PCONNRTS - Allows PCONNECT and PDISCONNECT Kalls
MCREARTS - Allows MCREATE Kall
MWRITRTS - Allows MWRITE Kall
MREADRTS - Allows MREAD Kall
MSENDRTS - Allows MSEND Kall
MRSVPRTS - Allows MRSVP Kall
MRPLYRTS - Allows MREPLY Kall
MWAITRTS - Allows MWAIT Kall

b) Initialization:
CREARTS, DLTRTS, ENVRTS, UCNFRTS, MDFYRTS, TMPLFLAG,
All Auxiliary rights

c) Template:
CREARTS, DLTRTS, ENVRTS, UCNFRTS, MDFYRTS, TMPLFLAG,
All Auxiliary rights

d) Copy: May not be COPYed

e) Creation arguments:
To be specified

f) Copy arguments: May not be COPYed

\- - - - - - -

13) Type DEVICE

a) Auxiliary:
PCONNRTS - Allows PCONNECT and PDISCONNECT Kalls
Rest to be specified

b) Initialization:
CREARTS, DLTRTS, ENVRTS, UCNFRTS, MDFYRTS, TMPLFLAG,
All Auxiliary rights

c) Template:
DLTRTS, ENVRTS, TMPLFLAG

d) Copy:
DLTRTS, Rest to be specified

e) Creation arguments:

To be specified

f) Copy arguments:

To be specified

\- - - - - - -

↑L

.SUBSEC |FORMAT FOR WHAT|

The WHAT Kall provides a representation of a Capability. The format