HYDRA
Basic Kernel Reference Manual

Ellis Cohen
Bill Corwin
Dave Jefferson
Tom Lane
Roy Levin
Joe Newcomer
Fred Pollack
Bill Wulf

Department of Computer Science
Carnegie-Mellon University

November 4, 1976

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.

Statement of Content

This publication, a subset of the Hydra Reference Manual, is intended as a general introduction to the details of the Hydra operating system. The chapters selected were those considered relevant to a general understanding and appreciation of the capability-based protection system of Hydra. The material also includes the introductory and reference material on the mechanism for the switching of protection domains, the CALL mechanism.

This document includes the *complete* Table of Contents and Index from the Hydra Reference Manual as an outline of the total work. Omitted are the chapters on Paging, the Kernel Multiprogramming System (KMPS), the GST, the Policy System Interface, the interprocess communication system (the Port system), the physical device I/O support, and most of the appendices. A section dealing with the fine details of LNS context blocks has also been omitted.

For those who are actually programming Hydra, the Hydra Programmer's Supplement is available, which includes the material on LNS context blocks and nearly all of the appendices. The chapters on KMPS, the Port system, the Policy system interface and device I/O are also available as separate documents, since most users do not need to interact with the Kernel at those interfaces.

The Reference Manual was published in parts in order to most efficiently serve all classes of users. Reference copies of the complete manual have been made available locally.

1. Introduction

1.1. About the Reference Manual

This document is a reference manual for the Hydra Kernel. It contains a minimal amount of tutorial information. Readers who are not familiar with the ideas behind the capability-based protection systems so vital to Hydra are encouraged to read first the Hydra article in the CACM [Wu74], and the collection of papers given at the Fifth Symposium on Operating Systems Principles, particularly the paper by Cohen [Coh75].

We want to emphasize that Hydra is not by itself an operating system in the usual sense of the word: it does not perform the tasks of allocation, scheduling, accounting, and handholding that are typical of conventional operating systems. It is better to think of Hydra as an extension of the PDP-11 hardware, one which extends the instruction set of the underlying computer. Hydra and the host PDP-11 together form a virtual machine, and Hydra is the *kernel* of an operating system to run on that virtual machine. In fact, many different, competing "operating systems" can be running on Hydra simultaneously. There is a default user environment, consisting of the Hydra kernel and a collection of subsystems, that a user initially faces when he logs on to Hydra.

Hydra is thus a software virtual machine implemented on C.mmp, a closely-coupled PDP-11-based multiprocessor. The extended virtual machine instructions, i.e. those not provided by the underlying hardware but by Hydra, are called K-calls, or Kernel calls. Although these K-calls are technically machine instructions for the virtual Hydra machine, they were implemented with the understanding that their major use would be in BLISS programs, and they are all defined as macros for the BLISS-11 compiler. BLISS-11 runs on the PDP-10, and there is a file (KERKAL.REQ[N811HY97]) that contains all these macro definitions, and which may be included in any BLISS-11 compilation. Appendix K of this reference manual is a listing of these K-call macros.

This manual does not contain any information about the user environment. There are separate documents describing the various subsystems that form the the user environment. In particular, the reader is referred to the *Command Interpreter Reference Manual*, the *Directory Subsystem Manual*, the *Policy Module Manual*, and various pieces of user documentation. There is a document still in preparation that will describe the utility functions available in the user environment.

1.1.1 The Hydra user disk area

Since most of the development of software for Hydra has taken place on the PDP-10, nearly all of the files of interest to users exist there. The most important disk area is [N811HY97], on which all of the BLISS *require* files reside. Also on this area can be found documentation, the system news, various utility packages, and many other useful files. Most of these are described in the companion volume to this, The Hydra Songbook [Reid75]. The Songbook contains a user-level introduction suited to the user sitting at a terminal and attempting to interact with the system. It describes the command interpreter, interactive debugger, and utility programs which exist on Hydra. This document is primarily for use by those users writing programs (particularly BLISS code) and for those needing detailed information during debugging.

1.1.2 A Note on Notation

In this manual, fully capitalized words generally denote kernel operations (K-calls) or object types. A special font identifies other Hydra technical terms, e.g. *type*. Underlining indicates the defining occurrences of a non-Hydra technical term, or otherwise suggests importance. Initially capitalized words occasionally identify uses of technical terms in contexts where italics would be confusing or overwhelming in number.

Numbers will always be written in decimal, unless preceded by a hash mark (sharp sign, pound sign), i.e., #10 = 8.

1.1.3 A note on the revised edition

The Revised Edition represents a major change in the naming conventions used in previous versions. The proliferation of symbols had reached the point where users could not determine which symbol should be used for what purpose, and in many cases the symbols conflicted with user-defined symbols. The resultant difficulty in creating programs indicated that a drastic change was necessary. Therefore, the current manual has renamed every symbol in the Hydra interface code, albeit often in trivial ways. Every symbol brought in by a BLISS "require" file now begins with a "$" character. Furthermore, some of the naming conventions have been made more consistent. In response to user suggestions some of the names have been made more mnemonic or easier to remember, e.g., the old symbol UCNFRTS has been redefined to be $UNCFRTS because most users remembered that spelling rather than the former. Since the symbols used in the old reference manual were generated over a long period of time and by a number of people, the conventions and abbreviations used tended to be inconsistent. In the revised edition, a concerted attempt has been made to retain all vowels and consonants in symbols wherever this did not result in absurdly long symbols. In such cases, the abbreviations were standardized. Many new "require" files have been defined for obtaining useful symbols, and all of these have been documented as thoroughly as is feasible. The intent is that all of these changes, made when the system is still largely uncommitted, will make future use much more comfortable.

1.1.4 A note on the flags

Several sections of this manual are flagged with special symbols in the margin. These symbols represent the status of Hydra on the date this manual was compiled.

> An asterisk in the margin indicates that the feature described in the *
text was not implemented at the time this manual was compiled. If *
the feature is needed, you should check the Hydra news files or ask *
someone in the Hydra group to determine if it has been implemented. *

> The vertical bar symbol indicates a change from a previous version |
of the manual. Since the changes from the February 14, 1975 |
edition are so extensive it was impractical to flag all of them, but |
editions of the manual produced after November 1976 will have |
such sections flagged. |

> A question mark indicates a feature of the system which was ?
undergoing redesign at the time the manual was compiled and is ?
likely to change. This flag warns you that you should check the ?
Hydra news for the current specifications. ?

> An "X" indicates the feature is obsolete, but is maintained in the X
documentation to help people read existing code, and to encourage X
people to recode programs that use the obsolete features. Users are X
strongly discouraged from using obsolete features, since there is no X
maintenance commitment to these features and future compatibility X
is not guaranteed. X

1.1.5 The Hydra Data Base

All of the symbols defined in this manual and their definitions are kept in a large data base file. This file and a set of associated processing programs are used to create all of the user "require files" used by BLISS/11 users. It is recommended that designers of other systems use this data base facility to obtain definition files for their programs, systems, or users. In this way, they can be guaranteed that their files will always be current with the latest changes in the system and not made obsolete by upward-compatible extensions or rendered inoperable by possible non-compatible changes. Although the Hydra group recognizes a responsibility to make as few non-compatible changes as possible, Hydra must be considered a research system and compatibility may conflict with other goals.

Large sections of this manual, particularly in the appendices, have their contents generated entirely from this data base. Thus it will be the case that as extensions are added the manual will be able to reflect these with a minimum amount of effort.

1.1.6 Acknowledgements

As in all large systems, a great many people have participated in the development of Hydra. In addition to the primary contributors to this document, we must acknowledge the contribution of all those users who helped debug this manual and who provided the first user community for Hydra. The list of users includes, but is not limited to, George Robertson, Richard Suslick, Guy Almes, Rick Gumpertz, Philip Karlton, Peter Oleinick, David Lamb and Brian Reid.

In addition to the original implementors of Hydra, a number of people have contributed code to the Hydra Kernel. This list includes, but again is not limited to, Sam Harbison, Joe Newcomer and George Robertson. The awesome task of maintaining the Kernel is currently the responsibility of Hank Mashburn.

As an aside, Brian Reid produced a special version of PUB that could cope with the enormous strain placed on it by the inordinate amount of PUB hacking that was used to produce this document.

As in all cases of acknowledgements, there may have been omissions. The editor wishes to take this space to apologize to anyone whose name has been inadvertently omitted from the above list.

2. The Hydra Kernel

2.1. The Basic Kernel

2.1.1 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-10 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 an executing procedure has associated with it 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. These names are guaranteed unique for the entire existence of the system (or 36,000 years, should it live so long). Rather than showing 'real' unique names in diagrams, (represented internally by 64-bit strings), we will instead substitute unique alphanumeric names for pictorial clarity. In the actual Kernel, alphanumeric print names are only associated with classes of objects, called *types*. Users are cautioned, however, that the Kernel makes no attempt to guarantee uniqueness of these print names, and in fact there is nothing to prevent users from creating new object types (equivalence classes) with non-unique print names.

As indicated above, 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 are contained in pages represented by PAGE objects. Capabilities for these PAGE objects must appear in the C-list of the executing procedure. Section 4 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)[1] and differs from the type representing its static counterpart, a PROCEDURE. PROCESS objects are the scheduling entities provided by the Kernel. At any instant, each executing process has an LNS associated with it which determines the *environment* in which the process runs. Hydra provides a *call mechanism* which changes environments by associating a different LNS with a process.

2.1.2 Objects, Capabilities, and Paths

Every object has two parts, a *C-list* containing a list of capabilities, and a *data-part* containing arbitrary data represented as a vector of 16-bit words. 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 vary from type to type. Section A.4 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, let us 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. This data-part is protected against examination or alteration by the user program. The data-part of a DEVICE object contains some information 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* provided by 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

1 or, as has been suggested, Local Number Space.

could provide file access checking in a way similar to that of the PDP-10 TOPS monitor). Alternately, 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 ([Org72]).

C-lists and data-parts can only be accessed and manipulated through the Kernel via K-calls. Among the many operations provided by the Kernel are some very basic K-calls that provide *generic* operations on all types of capabilities. Typical operations allow a user to 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 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. In addition, as will be explained in the following sections, there may be additional restrictions on the operations.

Most K-calls require some arguments which specify capabilities. In the simplest case, these are denoted by *simple indices* into the C-list of the LNS. For example, there is a K-call, $DELETE, and $DELETE (3) calls the Kernel to eliminate the 3rd capability in the LNS executing that K-call. Of course, most programs should use symbolic names or other mnemonic means to create meaningful descriptors for slot indices. We use absolute numbers here to illustrate the simplest possible form in a K-call.

Often, the Kernel will allow a capability to be denoted by a *path index* (see Figure 1). 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 paths*.)

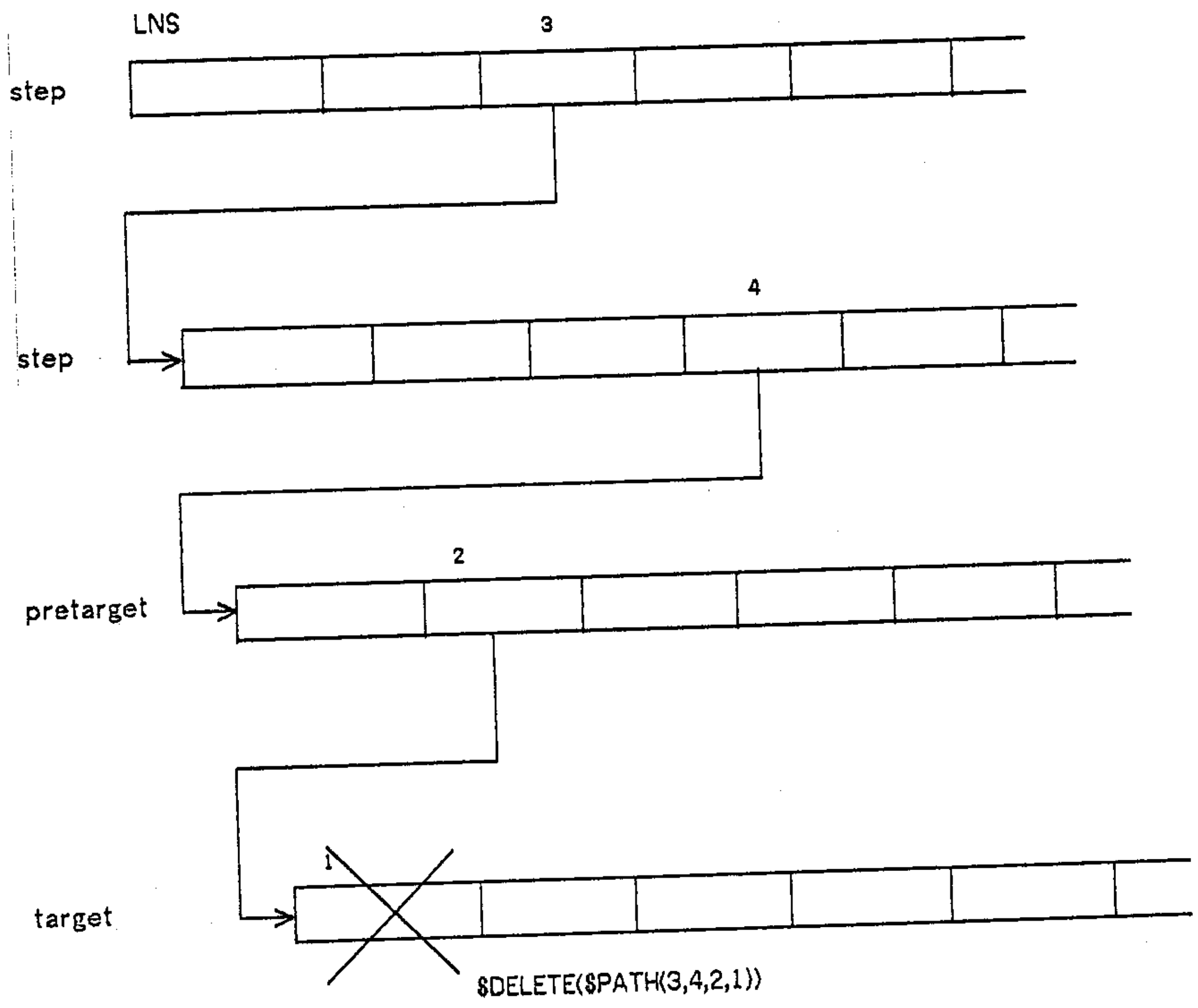


Figure 1: Steps, Pretarget and target in path walk

2.1.3 Generic 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 K-calls) with respect to a capability is determined by the presence of a particular bit in the *rights field* of a capability[2]. The following is a description of the rights relevant to basic Kernel K-calls. 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

$DELETERTS- Allows CAP to be deleted from the C-list which contains it; i.e., permits the $DELETE or $VACATE operation on CAP.

$ENVRTS - Allows CAP to be stored in some object; i.e., it may leave the environment.

C-list Rights

$GETCAPARTS[3]-Allows a capability to be extracted from OBJ's C-list; e.g., permits the $GETCAPA[4] operation on OBJ Note also that this right is always required on all steps of a path.

$PUTCAPARTS[5]-Allows a capability to be stored into OBJ's C-list; e.g., permits the $PUTCAPA[6] operation on OBJ.

$APPENDCAPARTS-Allows a capability to be appended onto OBJ's C-list; permits the $APPENDCAPA operation.

$KILLRTS - Allows a capability to be deleted from OBJ's C-list; permits the $DELETE or $VACATE operations on OBJ (as opposed to $DELETERTS which permits the operation on CAP).

---

2 There are 16 Generic Rights; thus the generic rights can be represented in a PDP-11 word. Only the more interesting rights are described here. For a listing of all rights and their respective bits, see section A.1.

3 Also known, for historical reasons, as $LOADRTS.

4 Likewise, also known as $LOAD.

5 Likewise, also known as $STORERTS.

6 Likewise, also known as $STORE.

Data-part Rights

$GETDATARTS-Allows data to be extracted from OBJ's data-part; permits the $GETDATA operation on OBJ.

$PUTDATARTS-Allows data to put into OBJ's data-part; permits the $PUTDATA operation on OBJ.

$APPENDDATARTS-Allows data to be appended onto OBJ's data-part; permits the $APPENDDATA operation on OBJ.

Restriction Rights

$MODIFYRTS- Allows modification of either OBJ's C-list or data-part.

$UNCFRTS - Allows OBJ to be 'UNConFined', that is, an object accessed through OBJ may be modified.

2.1.3.1 Some Examples

```
> $DELETE ( 3 )
   (The capability denoted by LNS slot) 3 requires $DELETERTS

> $DELETE ( $PATH(3,4) )
   3 requires $KILLRTS and $MODIFYRTS,
   $PATH(3,4) requires $DELETERTS

> $DELETE ( $PATH(3,4,2,1) )
   3 and $PATH(3,4) require $GETCAPARTS and $UNCFRTS,
   $PATH(3,4,2) requires $KILLRTS and $MODIFYRTS,
   $PATH(3,4,2,1) requires $DELETERTS
```

$GETCAPA(x,y) is a K-call which moves the capability at y to x, retaining the capability at y. x must be a simple index.

```
> $GETCAPA ( 5, $PATH(3,4,2) )
   3 requires $GETCAPARTS
   $PATH(3,4) requires $GETCAPARTS
   5 must be an empty slot
```

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

```
> $TAKE(x,y)
    like $GETCAPA but also deletes the capability at y.

> $TAKE ( 5, $PATH(3,4,3) )
    3 requires $GETCAPARTS and $UNCFRTS
    $PATH(3,4) requires $GETCAPARTS, $MODIFYRTS, and $KILLRTS
    $PATH(3,4,3) requires $DELETERTS
    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 K-call, $PUTCAPA(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 section A.2 for the format), by eliminating those rights not represented by a 1 in the mask. Note that it is always permissible to restrict the rights of a capability that one possesses.

```
> $PUTCAPA ( $PATH(3,4,3), 2, addr )
    3 requires $GETCAPARTS and $UNCFRTS
    $PATH(3,4) requires $PUTCAPARTS and $MODIFYRTS
    $PATH(3,4,3) must be an empty slot
    2 requires $ENVRTS
```

This call puts a copy of the capability in slot 2 in the destination specified by $PATH(3,4,3). The rights are then restricted by the contents of the two memory locations specified by 'addr' (the format of this is irrelevant at the moment, but readers with insatiable curiosity may find it in section A.2). It is possible to restrict rights without moving a capability by storing it into its own slot[7].

If the address designating the rights restriction mask is zero, no rights are restricted.

7 This latter case is the only case in which the result of a $PUTCAPA can be placed in a non-empty slot. Users are encouraged to use the $RESTRICTION K-call to perform this operation.

2.1.4 Auxiliary Rights

The rights we have seen so far are called *Generic rights* because they have meaning for any capability regardless of the type of the object it references. The interpretation of these rights is performed by the Kernel. In addition, each capability also contains a field of *auxiliary rights* that may be defined differently for each type of object[8]. Their use will become apparent in future examples. The assignment and interpretation of these auxiliary rights is left to the discretion of the subsystem designer; the Kernel performs no interpretation on the auxiliary rights fields of user-defined objects.

Note that although the Kernel does not *interpret* these fields, in the sense that it restricts the operations which may be performed on an object, it does perform the standard rights-checking which is performed when entering a new environment, and allows such operations as *rights amplification*; these concepts are explained in section 2.2.

2.1.5 Kernel-defined objects

The Kernel recognizes a basic set of types and treats them separately. Their auxiliary rights have predefined meanings and the Kernel also limits the Generic rights that any capability for an object of one of these types may have.

2.1.6 Empty slots

When a slot in a C-list is created, unless it is created by storing a capability into it, it is said to be *uninitialized*. With a few exceptions, any attempt to access the contents of an unitialized slot in a C-list will cause an error. An uninitialized slot is one form of an *empty* slot.

Objects of type NULL represent absolutely nothing. They are constrained by the Kernel to have neither a C-list nor a data-part[9]. An

---

8 There are eight auxiliary rights bits, which are stored in a single byte; for the exact format, see section A.2.

9 In fact, an object of type NULL never exists; only templates of type NULL can exist. Templates are not discussed until section 2.2.2. But we digress.

uninitialized slot contains a special kind of reference to something of type NULL[10]. A capability of any type may be stored in an uninitialized slot. Formally, we will say that an uninitialized slot is *unbound*, in the same way many programming language systems[11] mean that a value is unbound.

A slot in a C-list is also empty if it contains a reference to a NULL object[12]. In this case, the slot is said to be both *defined* and *empty*. The capability in such a slot may be operated upon as any other capability. In addition, the slot may be a target for any sort of store operation; a NULL capability may be replaced by any other capability.

If a slot in a C-list contains a non-NULL capability, it is said to be *defined* and *nonempty*. In most cases, an attempt to store a new capability into such a slot will cause an error.

A slot may be made empty by either the $DELETE or $VACATE K-calls. The $VACATE K-call deletes the capability at its target, leaving the target *empty* and *defined*. $DELETE deletes the capability at its target, leaving the target *empty* and *unbound*.

2.1.7 Empty slots and the C-list length

All C-lists have a *length* attribute. The *length* of a C-list is the index of the highest *defined* slot in the C-list. If a slot at the end of the C-list is made empty and unbound, the C-list size is reduced until it again reflects the index of the highest defined slot in the C-list; it will be at least one less than the size of the C-list before the last slot was unbound. Note that if the slot is simply made empty, but not unbound, no change in the C-list size will occur.

A C-list is extended implicitly as a side effect of operations which define the contents of a slot in the C-list. In general, any operation which causes a slot to be defined will cause the C-list size to be increased if the reference is to a slot outside the current C-list size.

In general, any attempt to access an unbound slot in a C-list will result in an error. Any attempt to access a slot in the C-list beyond the current C-

10 A NULL template with $UNBOUNDFLAG turned on.

11 E.g., LISP, APL, WATFOR, WATFIV.

12 Actually, a NULL template with $UNBOUNDFLAG turned off.

list size but less than the physical maximum permitted to that C-list will be treated as an attempt to access an unbound slot; if the user is concerned about the exact cause of the error, a check can then be made of the C-list size.

Each C-list has a maximum physical size, either an implementation upper bound determined by the Kernel or a specific upper bound $\leq$ the Kernel limit as determined when an object is created[13]. Any attempt to access a slot beyond the physical limit of the C-list will cause a "C-list bound exceeded" error.

2.1.8 Types DATA and UNIVERSAL

It is often convenient to be able to create a new object which simply encapsulates some data. The Kernel provides a K-call, $MAKEDATA, 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.

The Kernel also provides a UNIVERSAL object, one with both a C-list and a data-part. The K-call $MAKEUNIVERSAL creates just such an object.

2.1.9 K-call Values and Signals

Any K-call that executes successfully returns a non-negative value in BLISS register VREG (assembly language R0). K-calls that fail (e.g. for 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[14]. There is also a mechanism that can force signals to cause user traps (see section 2.3.1 for more details). The meaning of the various signals that can occur during basic Kernel K-calls can be found in section F.1[15]. It should be noted that the Kernel *signal mechanism* is distinct from (and incompatible with) a similarly-named facility in BLISS-11.

---

13 This is a simplified explanation; for more detail see section 2.2.5.

14 The fixed stack page locations are described by the file STKPAG.REQ[N811HY97] and in section F.5.

15 The Symbols for the Kernel Signals are defined in file SIGNLS.REQ[N811HY97]. See also appendix F.

2.1.10 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 executing simultaneously (on different processors). Consider, as an example, the case where a programming error results in both LNS's trying to $PUTCAPA 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 example above, one LNS (nondeterministically) will gain access to the object and $PUTCAPA a capability in it, while the other waits for the object to be unlocked. When the $PUTCAPA K-call completes, the other LNS will gain access to the object, but its $PUTCAPA K-call will fail (signal), since the slot in the shared object will no longer be empty.

For certain K-calls, if some referenced object cannot immediately be locked, the K-call will fail. To do otherwise in such cases would allow the possibility of deadlock. For the same reason, any K-call 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 K-call will fail.

In addition, it should be noted that there are some "composite" operations, e.g., $APPENDCAPA, $TAKE, $PASS, etc. (defined in section 2.1.15.3). These operations are defined in terms of simpler K-calls. Note, however, that because of the locking phenomenon, they are not precisely equivalent; if they were done as a series of K-calls, another process might be able to access, or alter one of the C-lists between the operations. In the composite operations these actions take place *indivisibly*, with both the source and target objects locked until the operations are complete.

2.1.11 Memory Addresses and 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 the 16 bit address designate 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 K-calls that allow the user to change other pages (virtual overlaying). Details can be found in section 4.7. Details on the C.mmp hardware may be found in a separate document ([WB72]).

Many K-calls require one or more arguments to be memory addresses.

Such memory addresses are expected to be the origin (low order address) of a block of memory from which the Kernel will retrieve information or into which the Kernel will store information. In most K-calls, these addresses can be anywhere in the user's immediately addressible page set[16] and are referred to as *legitimate memory addresses.* Historically, the Kernel required many of these addresses be on the stack page, and these are referred to as *legitimate stack memory addresses.* It is intended eventually that all multiword blocks of data (except for rights restriction masks) will be permitted to have legitimate memory addresses and not be confined to the stack; however, this document represents the restrictions as of November 4, 1976.

In the case where the Kernel demands that a legitimate stack memory address be used, it must have the following properties:

> The address must be in the stack page (high order 3 bits of the address must be 0).

> The block of memory to be accessed must lie within the *active region* of the stack. (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[17]. The region between the initial SP contents and the current contents of SP is called the *active region* of the stack). See figure 2. Further explanation of the stack limits given in figure 2 are given in section F.4.

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

In all other cases, the address must have the following properties:

> The page must be in the user's current RPS, i.e., directly addressible by the currently executing code.

> The page must be writable if data is to be written into it. (This is controlled by auxiliary rights on the page; see section 4.5).

> The address must be on a word boundary, unless a byte address is explicitly permitted for the K-call being specified.

---

16 Known as the relocation page set or RPS, discussed in sections 4.1 and 4.2.

17 Recall that the stack grows down, i.e., towards lower-numbered addresses.

> If the address specifies a block of memory, the block must not cross a page boundary.

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.

The user has no control over the value of the initial stack pointer when a procedure is invoked; the value depends upon the current call stack in the process and the amount of Kernel data placed in the stack. This property is true in the current stack-page-per-process implementation. Future revisions of the implementation may change this property. ? ? ? ? ? ?

There are some operations in the I/O system (chapter 8) which permit the user to specify an address which is not in the current RPS. Such addresses are used only in special-purpose I/O handling and cannot be specified in K-calls.

Locations #200-#377 comprise the *Kernel data area*. When signals, traps, and errors occur, certain additional information is placed in locations within this area. (Section F.5 lists these fields).

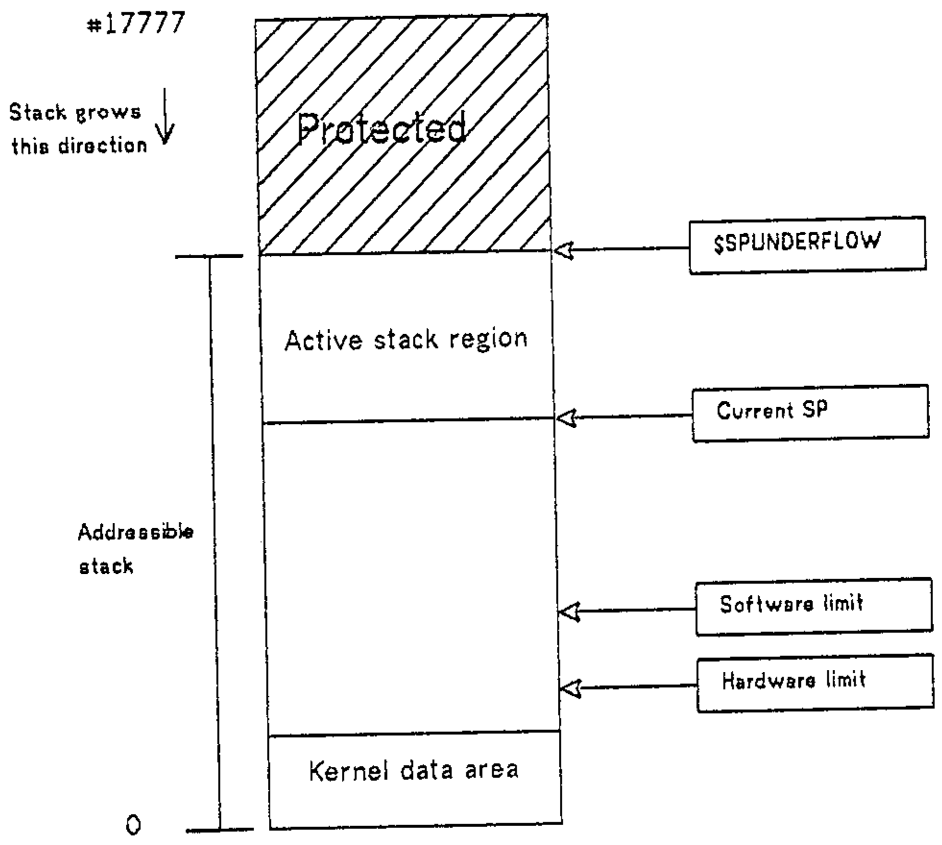


Figure 2: Stack page regions

2.1.12 Indirect K-calls

Often it is useful to be able to build up the argument stack for a K-call independently of the actual K-call itself (especially for interpretive and debugging programs).

The special K-call, $KALLINDIRECT provides this function. Its parameter specifies the beginning address of the argument stack and must be a legitimate stack memory address. The complete specification for this K-call can be found in section 2.1.15.5. Section I contains all details necessary for constructing the argument stack.

2.1.13 Conventions for K-call Specifications

There is a canonical format used in this manual to describe Kernel calls. Each K-call description consists of several components:

> The K-call name and its formal parameter list. K-calls are described in terms of Bliss macros. See appendix K.

> The 'parameters' section. Parameters to K-calls fall into three classes:

> An integer value, i.e., a 16-bit value, not to be confused with the integer used as a simple index to denote a capability (see below).

> A legitimate memory address. In some cases this is restricted to be a legitimate stack memory address; the differences and constraints are described in section 2.1.11. 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 sections 2.1.3 and A.2. As a notational convention, we will indicate memory locations with the symbols 'Mem' (for general memory locations) and 'Smem' (for stack memory locations). We will suffix these symbols with W or R to indicate that the location will be written into or only read, and further suffix the symbol with a number for the case where a block of memory is used. Thus 'SmemW16' indicates a 16-word block of data will be written into a valid stack memory address. When the amount of data transferred is a parameter to the K-call, the suffix 'Wn' or 'Rn' will be used, with a 'Count' parameter in the K-call specifying the actual amount of data transferred. Because of the frequency of rights-restriction masks, the symbol 'SMemrts' will be used to mean 'SmemR2' in such usage.

> A denotation for a capability - either a simple index, (sometimes negated[18] or 0 for a special effect) or a Path index, or a $CALL parameter (to be defined in section

[18] Of interest only to non-BLISS users and those using the Indirect K-call, see section 2.1.12.

2.2). We will also indicate necessary rights, type or kind (object reference or template) for the target capability, its steps, and its pretarget.

Again, as a notational convenience we will often indicate capabilities to be "source" capabilities (the targeted object is usually not altered[19]) and use the designators 'SPath' or 'SIndex', or "destination" capabilities (the targeted object is altered) and use the designators 'DPath' or 'Dindex'.

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 $GETCAPARTS.

We will not list restrictions on arguments that seem obvious or redundant and produce obvious signals if the restrictions are not met - most notably, indices into C-lists or data-parts less than 1 or greater than the maximum length (see section 2.1.7).

> 'Effect' is the effect of the K-call if no signal occurs. Except for a small subcase of LNS incarnation, K-calls that fail have no side effects.

> 'Signals' indicate that the K-call did not complete normally. 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 K-call and are described in section F.1. For a detailed description of how an LNS may handle signal conditions see section 2.3.2.

> 'Result' is the value of the K-call (returned in register VREG (R0)) assuming no signal occurred. (If a signal occurred, the value of the K-call is the signal value instead.)

2.1.14 Some undefined concepts

There will be a number of references in the forthcoming section (2.1.15) to concepts which, for the sake of exposition, have not yet been discussed. The comprehension of these concepts are not essential for

[19] Except in certain K-calls which transfer (i.e. delete from the source site) capabilities or exchange information; in the former case the source operand indicates where the capability is obtained; in the latter case both may be considered destination operands.

understanding the following section, but for completeness they must appear. References to *aliases*, *freezing*, *confinement*, and the associated rights, $REALLYRTS, $FREEZEFLAG, and $UNCFRTS may be ignored unless such matters are truly the concern of the moment. Such matters are dealt with in depth in section 2.2.4.

2.1.15 Specifications for Basic Kernel Calls

2.1.15.1 Informational K-calls

$LNSLENGTH ( )

<u>Parameters:</u>

None

<u>Effect:</u>

None

<u>Result:</u>

Length of the C-list of the executing LNS. The concept of "length" is fully described in section 2.1.7. Note that the length of the LNS may change dynamically during program execution.

$CLENGTH (SPath)

<u>Parameters:</u>

SPath	-	Path index; Pretarget: $GETCAPARTS; Target: object reference, $GETCAPARTS

<u>Effect:</u>

None

<u>Result:</u>

Length of the C-list of the object referenced by SPath's Target. The concept of "length" is fully explained in section 2.1.7. Intuitively it is the highest index of a defined capability.

$DLENGTH (SPath)

<u>Parameters:</u>

SPath	-	Path index; Pretarget: $GETCAPARTS; Target: object reference, $GETDATARTS

<u>Effect:</u>

None

<u>Result:</u>

Size, in words, of the data-part of the object referenced by SPath's Target.

$OBJINFO (SMemW16, SPath)

<u>Parameters:</u>

SMemW16-	Legitimate stack memory address
SPath	- Path index; Pretarget: $GETCAPARTS; Target: defined or unbound

<u>Effect:</u>

Information about the capability targeted by SPath is stored in the 16 word block of memory beginning at SMemW16. See Appendix B for the format. This K-call may be executed on a *unbound* target without causing an error.

<u>Signals:</u>

This K-call will *not* signal an unbound target.

<u>Result:</u>

0

$COMPARE (SPath, SIndex)

<u>Parameters:</u>

SPath	- Path index; Pretarget: $GETCAPARTS; Target: defined or unbound
SIndex	- Simple index, defined or unbound, or 0

<u>Effect:</u>

None

<u>Signals:</u>

This K-call will *not* signal if either or both of its targets are *unbound*.

<u>Result:</u>

A word of bits which indicate how the capabilities targeted by SPath and SIndex compare. If SIndex is 0, then just those bits pertaining to the capability targeted by SPath are set. See Appendix C for the meaning of each bit.

2.1.15.2 Simple DATA and UNIVERSAL Manipulation

$GETDATA (MemWn, SPath, Disp, Count)

<u>Parameters:</u>

MemWn	- Legitimate user address in the current RPS
SPath	- Path index; Pretarget: $GETCAPARTS; Target: $GETDATARTS
Disp	- Positive integer less than or equal to $DLENGTH(SPath)
Count	- Positive integer

<u>Effect:</u>

Moves up to Count words of data from the data-part of the object referenced by the Target of SPath to the block of memory beginning at MemWn. The data is copied beginning at the Disp'th word of the data-part and continuing for a total of Count words or until the end of the data-part is reached. Note that data parts are indexed with a 1-origin, i.e., the first word in the data part of an object is word 1.

<u>Result:</u>

Total number of words copied

$PUTDATA (DPath, MemRn, Disp, Count)

<u>Parameters:</u>

MemRn	- Legitimate user address in the current RPS
DPath	- Path index; Steps and Pretarget: $GETCAPARTS, $UNCFRTS; Target: $PUTDATARTS, $MODIFYRTS
Disp	- Positive integer
Count	- Positive integer

<u>Effect:</u>

Copies Count words of data beginning at MemRn into the data-part of the object targeted by DPath. The data is stored beginning at the Disp'th word of the data-part. The data-part will be extended if necessary to contain the data. If L_0 was the length (see $DLENGTH) before the $PUTDATA, then the new data-part length will be *max*(L_0,Disp+Count-1).

Signals:

$HySDBOUND-(Disp+Count) would exceed either the implementation-defined limit for a data part (section A.4) or the type-specific limit for the specific object involved (section 2.2.5).

Result:

0

$MAKEDATA (DPath, MemRn, Count, Smemrts)

Parameters:

DPath	- Path index; Steps: $GETCAPARTS, $UNCFRTS; Pretarget: $PUTCAPARTS, $MODIFYRTS; Target: empty
MemRn	- Legitimate user memory address in the current RPS
Count	- Non-negative integer
Smemrts	- Legitimate stack memory address, or 0

Effect:

Creates a DATA object and places a capability for it in DPath's Target. The data-part of the created object will contain the Count words of data copied from the block of memory beginning at MemRn. The capability will have all relevant rights except $REALLYRTS and $FREEZEFLAG and will be further restricted by the contents of SMemrts if SMemrts is non-zero.

Result:

0

$APPENDDATA (DPath, MemRn, Count)

Parameters:

DPath	- Path index; Steps and Pretarget: $GETCAPARTS, $UNCFRTS; Target: $APPENDDATARTS, $MODIFYRTS
MemRn	- Legitimate user memory address in the current RPS
Count	- Positive integer

Effect:

Copies the Count words of data from the block of memory beginning at MemRn onto the end of the data-part of the object referenced by DPath's Target. If L_0 was the length of the data-part before the operation, the new length is L_0+Count.

Signals:

See $PUTDATA.

Result:

Displacement in data-part of DPath's Target of first data word stored, i.e. one greater than the length of the data-part before the store occurred.

$SETDLENGTH (DPath, Count)

Parameters:

DPath	- Path index; Steps and Pretarget: $GETCAPARTS, $UNCFRTS; target: $MODIFYRTS, $PUTDATARTS
Count	- Positive integer

Effect:

Sets the length of the data-part of the target object to be Count. *This is the only way to reduce the length of a data-part.* The value may be any value between 0 and the maximum data-part length, either the implementation maximum defined in section A.4 or the type-specific limit as explained in section 2.2.5. If the data-part length was less than Count, the data-part is extended and filled with zeroes.

Signals:

$HySDBOUND-Count would exceed either the implementation-defined limit for a data part (section A.4) or the type-specific limit for the specific object involved (section 2.2.5).

Result:

$MAKEUNIVERSAL (DPath)

Parameters:

DPath	- Path index; Steps: $UNCFRTS, $GETCAPARTS; Pretarget: $PUTCAPARTS, $MODIFYRTS; Target: empty

Effect:

Creates a UNIVERSAL object and places a capability for it with all but $REALLYRTS and $FREEZEFLAG in DPath's Target.

Result:

0

2.1.15.3 Simple Manipulation of Capabilities

$PASS (DPath, SIndex, SMemrts)

Parameters:

DPath	- Path index; Steps: $GETCAPARTS, $UNCFRTS; Pretarget: $PUTCAPARTS, $MODIFYRTS; Target: empty
SIndex	- Simple index, $DELETERTS; if DPath is not simple, requires $ENVRTS as well
SMemrts	- Legitimate stack memory address, or 0

Effect:

Copies the capability in the SIndex'th slot of the current LNS to DPath's target, restricting rights (if SMemrts is nonzero) according to the contents of SMemrts. The capability at SIndex is then made *unbound*. Future attempts to access the target of SIndex will (generally) signal.

Result:

0

$TAKE (DIndex, SPath)

Parameters:

DIndex - Simple index, empty

SPath - Path index; Steps: $GETCAPARTS, $UNCFRTS; Pretarget: $KILLRTS, $GETCAPARTS, $MODIFYRTS; Target: $DELETERTS

Effect:

Copies the capability targeted by SPath to the DIndex'th slot of the current LNS. If any Step or Pretarget in SPath lacks $UNCFRTS, then DIndex will have $UNCFRTS, $MODIFYRTS and $REALLYRTS removed. If any capability in SPath lacks $ENVRTS, DIndex will have $ENVRTS removed. The capability targeted by SPath is then made *unbound*. Future attempts to access the target of Spath will (generally) signal.

Result:

0

$PUTCAPA [20] (DPath, SIndex, Smemrts)

Parameters:

DPath - Path index; Steps: $UNCFRTS, $GETCAPARTS; Pretarget: $MODIFYRTS, $PUTCAPARTS; Target: empty

SIndex - Simple index, Defined; If DPath is not simple, requires $ENVRTS as well. ?
If DPath and SIndex are the same, then none of the ?
above rights requirements holds, rather the ?
capability needs $DELETERTS. ?

Smemrts - Legitimate stack memory address, or 0

Effect:

Copies the capability in the SIndex'th slot of the current LNS to DPath's target, setting $DELETERTS, and (if Smemrts is nonzero) restricting rights according to the contents on Smemrts. ?

If DPath and SIndex are the same, however, the rights in the ?
target are simply restricted according to the contents of Smemrts ?
(if Smemrts is nonzero). ?

20 Also known, for historical reasons, as $STORE.

Result:

0

$GETCAPA [21] (DIndex, SPath)

Parameters:

DIndex	- Simple index, empty
SPath	- Path index; Pretarget: $GETCAPARTS; Target: Defined

Effect:

Copies the capability targeted by SPath to the DIndex'th slot of the current LNS, and sets $DELETERTS. If any capability in Target's Path lacks $UNCFRTS, DIndex will have $UNCFRTS, $MODIFYRTS and $REALLYRTS removed. If any capability in SPath lacks $ENVRTS, $ENVRTS will be removed from DIndex.

Result:

0

$PASSAPPEND (DPath, SIndex, Smemrts)

Parameters:

DPath	- Path index; Steps and Pretarget: $GETCAPARTS, $UNCFRTS; Target: $MODIFYRTS, $APPENDCAPARTS
SIndex	- Simple index, $DELETERTS, $ENVRTS; defined
Smemrts	- Legitimate stack memory address or 0

Effect:

Appends the capability in the SIndex'th slot of the current LNS onto the end of the C-list of the object referenced by DPath's target, restricting rights (if Smemrts is nonzero) according to the contents of Smemrts. The capability at SIndex is then made *unbound*. Future attempts to access the contents of Sindex will (generally) signal.

Result:

Slot number in Target's C-list which received SIndex.

21 Also known, for historical reasons, as $LOAD.

$APPENDCAPA (DPath, SIndex, Smemrts)

Parameters:

DPath	- Path index; Steps and Pretarget: $UNCFRTS, $GETCAPARTS; Target: $MODIFYRTS, $APPENDCAPARTS
SIndex	- Simple index, $ENVRTS
Smemrts	- Legitimate stack memory address or 0

Effect:

Appends the capability in the SIndex'th slot of the current LNS onto the end of the C-list of the object referenced by DPath's target, setting $DELETERTS, and restricting rights (if Smemrts is nonzero) according to the contents of Smemrts.

Result:

Slot number in Target's C-list which received SIndex.

$VACATE (DPath)

Parameters:

DPath	- Path index; Steps: $UNCFRTS, $GETCAPARTS; Pretarget: $MODIFYRTS, $KILLRTS; Target: $DELETERTS

Effect:

Deletes the capability targeted by DPath. See section 2.2.5 for other potential effects if this was the last capability referencing an object. The slot targeted by Dpath is *empty* but *defined* (see sections 2.1.7 and 2.1.6).

Result:

0

$DELETE (DPath)

Parameters:

DPath	- Path index; Steps: $UNCFRTS, $GETCAPARTS; Pretarget: $MODIFYRTS, $KILLRTS; Target: $DELETERTS

Effect:

Deletes the capability targeted by DPath. See section 2.2.5 for other potential effects if this was the last capability referencing an object. The slot targeted by Dpath is both *empty* and *unbound*. This may also affect the C-list length of the C-list containing the target. See sections 2.1.7 and 2.1.6.

Result:

0

$INTERCHANGE (DPath, DIndex, Smemrts)

Parameters:

DPath	- Path index; Steps: $UNCFRTS, $GETCAPARTS Pretarget: $MODIFYRTS, $KILLRTS, $GETCAPARTS, $PUTCAPARTS; Target: $DELETERTS
DIndex	- Simple index, $DELETERTS, $ENVRTS
Smemrts	- Legitimate stack memory address, or 0

Effect:

Interchanges the capabilities targeted by DPath and by DIndex. Restricts rights (if Smemrts is nonzero) of the capability placed into DPath's target according to the contents of Smemrts. If any Step in DPath lacks $UNCFRTS, DIndex will have $UNCFRTS, $MODIFYRTS and $REALLYRTS removed. If any Step lacks $ENVRTS, DIndex will have $ENVRTS removed.

Result:

0

$RESTRICT (DPath, Smemrts)

Parameters:

DPath	- Path index; Steps: $GETCAPARTS, $UNCFRTS; Pretarget: $GETCAPARTS, $PUTCAPARTS, $KILLRTS, $MODIFYRTS; Target: $DELETERTS; Target may be empty
SMemrts	- Legitimate stack memory address, or 0

Effect:

Restricts the rights at DPath's target according to the contents of SMemrts. If SMemrts is 0, no restriction is performed.

Result:

0

2.1.15.4 Date and Time K-calls

$GETGMT (MemW4)

Parameters:

MemW4 - Legitimate memory address, writeable.

Effect:

Returns the current time in microseconds in GMT[22] relative to system date 0. MemW4 is a four-word area which contains the result if no errors have occurred. This value may be subsequently converted to other formats by use of the $GMTTOLOCAL K-call. For the format of this area, see Appendix D.

Result:

0

$GNAMETOGMT (MemW4,MemR4)

Parameters:

MemR4 -	Legitimate memory address; contains the global name of an object
MemW4 -	Legitimate memory address, writeable; Note that MemW4 may overlap or be identical to MemR4.

Effect:

Converts the global name at MemR4 (such as is obtained from the $OBJINFO K-call, see section 2.1.15.1) to a corresponding time in microseconds in GMT and relative to system date 0. This value could be used in the $GMTTOLOCAL K-call, below, to obtain a creation date and time for the object.

Result:

0

[22] Greenwich Mean Time.

$GMTTOLOCAL (MemWn, Count)

Parameters:

MemWn -	Legitimate memory address, writeable; length is specified by the Count parameter
Count -	Integer, value ≥4.

Effect:

The first four words at the location specified by MemWn must contain a valid GMT time value (such as might be obtained from the $GETGMT K-call, above). The value is converted to local time and as many words as specified above the GMT date are filled in with this information. The exact format is given in Appendix D. Note that if a Count larger than the current implementation maximum is given, only as many words as the maximum specifies are produced; the balance of the area is unchanged. However, future extensions to this K-call may write in these words, so this behavior should not be depended upon. Note, however, that future extensions *will never write more words than Count specifies* so users are guaranteed compatibility between future extensions and existing code.

Signals:

$HySGMT	- The GMT value cannot be converted within the current implementation-defined limits (dates beyond the year 2060, approximately).

Result:

0

$GETUPTIME (MemW4)

Parameters:

MemW4 -	Legitimate memory address, writeable.

Effect:

Returns the time, in units of microseconds, of the GMT time that the system came up (relative to system date 0). MemW4 is a four-word area which contains the result if no errors have occurred. The value may be subsequently converted to other formats by use of the $GMTTOLOCAL K-call, or used directly to compare previous time stamps obtained by this K-call.

One use of this K-call is to allow the user to determine that the system has been taken down since the last time the time stamp was read. This permits the user to validate complex structures of objects and data which might have been damaged if the system was taken down when they were presumed to be in an inconsistent state.

Result:

0

$GETCLOCK (SMemW4)

Parameters:

SMemW4- Legitimate stack memory address

Effect:

Puts a reading of the system clock into the 4 word block of memory beginning at SMemW4. See Appendix E for the format.

Note: This K-call is obsolete and has been superceded by the GMT conversion K-calls given previously. It is included here so that existing code may be understood.

Signals:

Result:

0

2.1.15.5 Indirect K-call specifications

$KALLINDIRECT (SMemRn)

Parameters:

SMemRn- Address of the start of the K-call parameter list

Effect:

This K-call treats the data at SMemRn as if it were stacked as parameters to the K-call. For details on the format see section I.

Signals:

All signals are possible from this K-call. A signal may indicate an error in the parameter stack, the address passed, etc., or be generated by the K-call described at SMemRn.

Result:

Depends upon the K-call specified at SMemRn.

2.2. The Intermediate Kernel

2.2.1 Domain Switching

Let us consider some standard protection problems:

> 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.

> A program that manipulates a data base needs capabilities to access the data base but it should never be possible for callers of the program to have direct access to the data base.

- Since no program except those belonging to the data base system can ever access the data base representation directly, the system implementors may feel free to change internal representations at any point.
- Since no program except those belonging to the data base system can ever modify the data base, no user can inadvertently or maliciously alter the structure of the data base.
- Since no users can access the contents of the data base without using the data base system as an intermediary, the system can restrict what information will be passed out to various classes of users and what modifications it will accept from various classes of users.

In response to these issues, Hydra provides PROCEDURE objects. The K-call $CALL(Rtrn,Proc,A_1,...,A_k)$ creates a new LNS in which the procedure's code will execute and transfers control to it. (Proc denotes a capability for an object of type PROCEDURE, A_1 through A_k 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 K-calls $RETURN and $SUSPEND pass 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*[23] and can be used to solve the data base problem just mentioned. 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 check-rights field of the corresponding parameter template[24]. The initial value of the check-rights field in a parameter template is 0 for both Generic and Auxiliary rights, indicating that no rights-checking will take place. See section 2.2.7.2 and the $SETCHKRIGHTS K-call. A NULL parameter template may be used in cases where a number of different types of objects must be accepted as parameters. In such cases, the user must perform the type-checking explicitly, by using K-calls such as $COMPARE and $OBJINFO (section 2.1.15.1) or by more powerful operations, such as $MERGE, described below and in section 2.2.7.4.

Procedures can be used to construct systems of programs collectively known as <u>protected subsystems</u>. 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 the set of PROCEDUREs which comprise the 'Directory subsystem'. Therefore, the rights [in a capability] to a Directory object are *restricted* in such a way that no executing LNS outside the Directory subsystem can access or modify a Directory object. However, the Directory subsystem itself must be able to access and alter a Directory object; therefore it must gain the rights necessary to do this, even though the [capability for a] Directory object passed to it does not possess these rights. Hydra accomplishes this through a process called *rights amplification*. Capabilities passed as arguments in a $CALL need not have the same rights, either Generic or Auxiliary, 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

---

23 These are inherited from the PROCEDURE object which created the LNS, not the LNS which invoked the procedure. This point is sometimes unclear. The analogous feature in some programming languages is the ability to declare how "own" variables in a procedure or coroutine are to be initialized upon an invocation.

24 Note that both the Generic Rights and the Auxiliary Rights fields are included in the rights-checking.

the capability passed as an argument; in the incarnated LNS, the capability will have the new rights. Note that the new rights may, in some cases, be *less* than the rights passed; for example, an information-delivering procedure might, for the peace of mind and protection of its implementor, restrict all rights which would allow it to modify the "read-only" objects passed to it as parameters. This would prevent it from accidently modifying such objects, and furthermore might[25] prevent any procedures it calls from modifying the objects.

Figure 3 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 of 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'. 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 unrestricted creation of parameter templates which specify new-rights; 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 section 2.2.5), and since templates are capabilities, their dissemination 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 K-call $MAKELNS incarnates an LNS from a procedure and arguments, while the K-call $LNSCALL transfers control to the LNS. The advantages of having such pre-initialized LNS's are efficiency and the ability to build coroutine structures. Once an LNS $SUSPENDs, it may be $LNSCALLed again. Execution continues after the $SUSPEND. The LNS's pages, its C-list and registers R0 and PC will be retained; however, the rest of the registers will
? be destroyed
?
? and the stack will be reinitialized.

The same pre-initialized LNS may be used by more than one process at a

[25] Depending upon which rights were restricted.

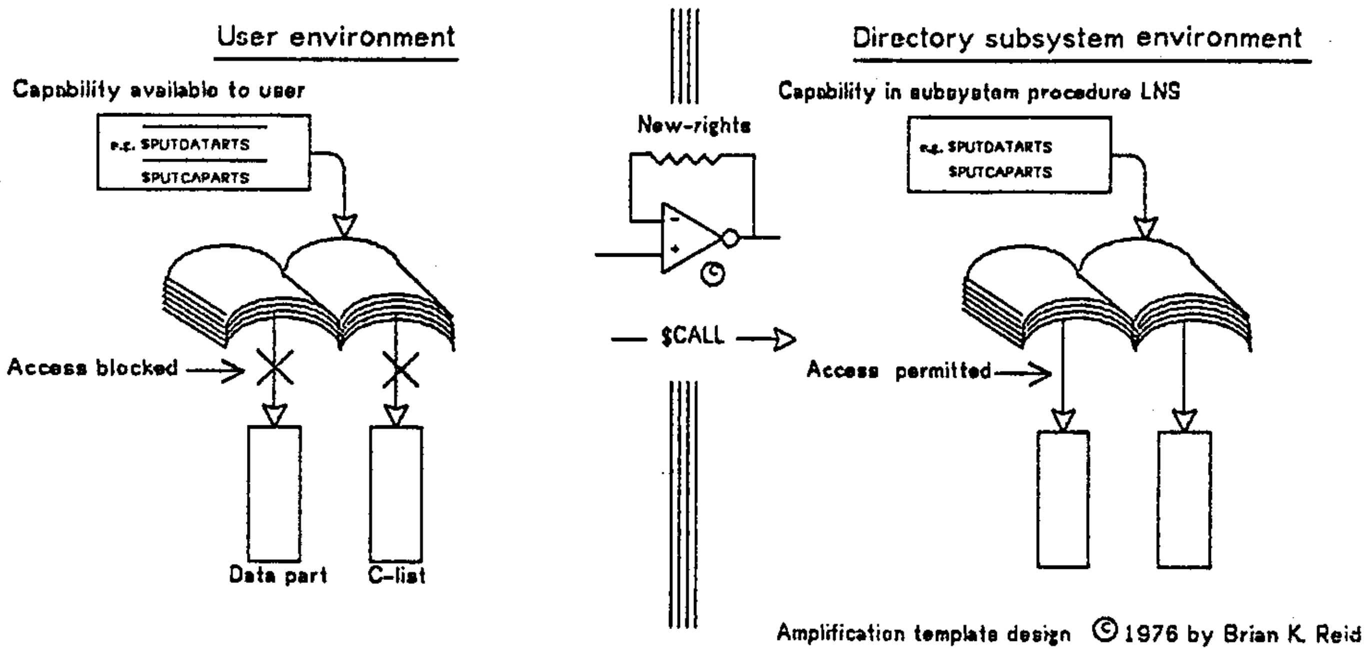

User environment
Capability available to user
e.g. $PUTDATARTS
$PUTCAPARTS
Access blocked
Data part
C-list
New-rights
$CALL
Directory subsystem environment
Capability in subsystem procedure LNS
e.g. $PUTDATARTS
$PUTCAPARTS
Access permitted
Amplification template design © 1976 by Brian K. Reid

Figure 3: Example of a subsystem

time, but not simultaneously[26]. It must $SUSPEND before it can be $LNSCALLed again. The creator of a procedure can control whether or not it can be passed to $MAKELNS, whether or not it can be re-$LNSCALLed, and whether or not it can be used to instantiate a process.

2.2.2 TEMPLATEs and 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

[26] The LNS is serially reusable but not re-entrant. Note that these properties apply to the LNS and not the code involved.

(for general merging by use of the $MERGE K-call described in section 2.2.7.4) as well as parameter templates (for $CALL-time merging). Templates contain two flags:

$TemplateFlag -	1 - Capability template 0 - Parameter template
$AmplifyFlag -	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 (see section A.2). 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, $UNCFRTS, $FREEZEFLAG, $MODIFYRTS, and $REALLYRTS. (Templates never have $REALLYRTS.) They will appear in the merged capability only if they appear both in the new-rights of the template as well as in the original capability.

2.2.3 NULLs Revisited

An *empty slot* has already been defined as a slot containing a NULL capability. In fact, it is impossible to create a NULL object, and empty slots actually contain NULL templates.

NULLs have one auxiliary right predefined, $UNBOUNDFLAG. We use the term *unbound* to mean a NULL template with both $UNBOUNDFLAG and $TemplateFlag set. When an object is initially created, its C-list is set to contain all unbound capabilities with all Kernel rights[27].

The *length* of a C-list is really the index of the last defined slot in the C-list. Hence NULL parameter templates or NULL templates lacking $UNBOUNDFLAG are included in the length.

[27] The initial size of the C-list of such objects is a parameter determined at the time the TYPE representative is created; see sections 2.2.5 and 2.2.7.3. For Kernel types, it is the implementation maximum given in section A.4.

2.2.4 Confinement, Freezing, and Revocation

A number of Generic rights are provided to solve some interesting protection problems. $ENVRTS, $MODIFYRTS, and $UNCFRTS 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 against whose leakage we wish to protect.

$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 except the LNS which received it as a parameter. It may be used as an argument to an LNS which the callee CALLs, but $ENVRTS cannot be gained through rights amplification. Additionally, whenever a capability is loaded into an LNS through any path in which some capability lacks $ENVRTS, the loaded capability will have $ENVRTS removed. This guarantees that if a capability without $ENVRTS is passed to a procedure neither the capability, nor any capability in the object it references, may escape from the caller's environment. Each inherited capability in an LNS incarnated from a procedure capability lacking $ENVRTS will have $ENVRTS removed as well.

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.

$MODIFYRTS and $UNCFRTS can be used to protect objects from modification through capabilities lacking those rights. If an LNS calls another LNS passing a capability lacking $MODIFYRTS, the callee cannot modify the accessed object through that capability regardless of amplification. This is assured because $MODIFYRTS cannot be gained through rights amplification and any K-call that modifies an object requires a capability for that object with $MODIFYRTS as well as other relevant rights.

$UNCFRTS 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 $UNCFRTS.

Users may wish to guarantee that information passed to an untrusted procedure will not be leaked to another user. The Generic right $UNCFRTS also provides this guarantee. Any LNS incarnated from a procedure capability lacking $UNCFRTS will be *confined*. Each capability in the LNS inherited from the Called procedure will lose $UNCFRTS, $MODIFYRTS, and

$REALLYRTS (explained later in this section). Confinement is then provided in the following way. The reader may note that any K-call which modifies an object requires that the capability for the object have $MODIFYRTS and that other capabilities in the *path* to the object have $UNCFRTS. Additionally, whenever a capability is loaded into an LNS through a path where some capability lacks $UNCFRTS, the loaded capability will have $UNCFRTS, $MODIFYRTS, and $REALLYRTS 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 $UNCFRTS 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 $UNCFRTS, 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 flag $FREEZEFLAG is used like a right to guarantee that an object is frozen. The K-call $FREEZE acts very much like a $COPY, except that it also sets $FREEZEFLAG and eliminates $UNCFRTS and $MODIFYRTS from the copy it creates. Since $UNCFRTS and $MODIFYRTS 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 $FREEZEFLAG can represent a guarantee of frozen-ness, it also cannot be gained through amplification.

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 aliasing object with $REALLYRTS. With $REALLYRTS, the aliasing object may be re-aliased with the $REALLY operation 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, $REALLYRTS are removed as well. *

2.2.5 Types, Creating, and 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. There is, of course, an equivalence class for objects of type TYPE represented by an object whose name is TYPE and whose type is TYPE. Some of these equivalence classes (for Kernel-defined objects) are illustrated in figure 4.

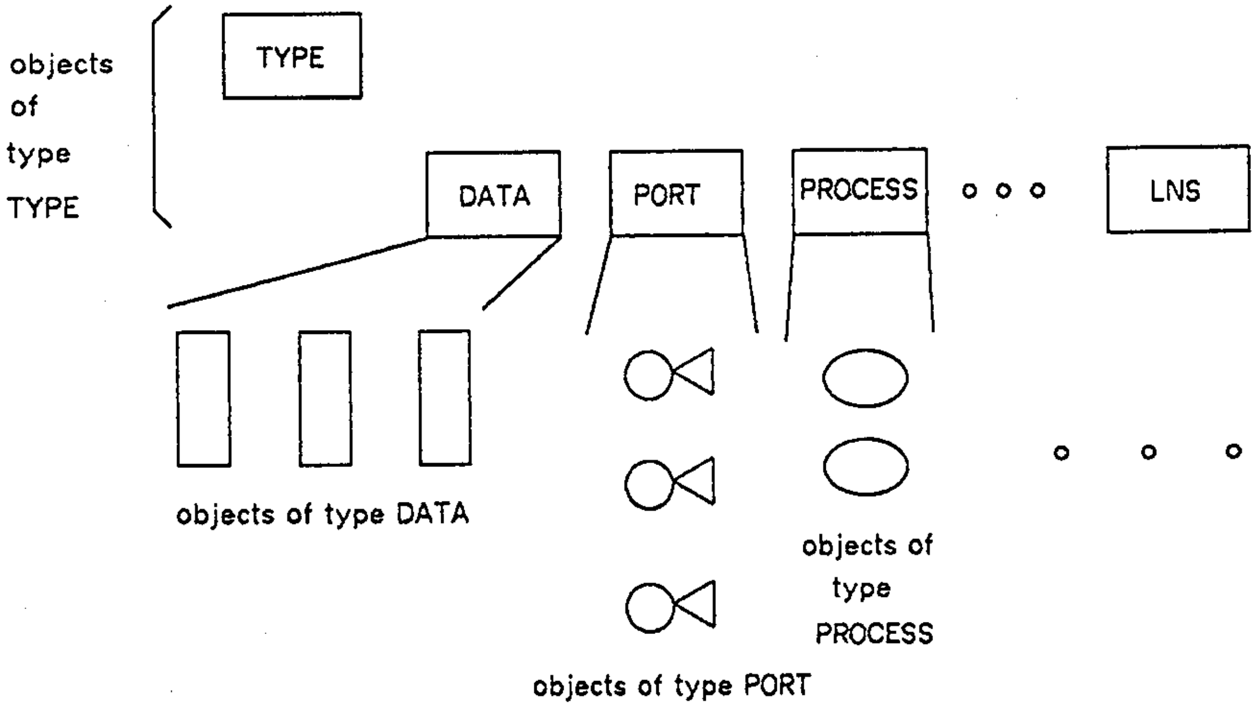


Figure 4: Some Generic-defined objects

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 Figure 4.)

The way to create a new object of some type, say FILE, is to use the K-call $CREATE, supplying as an argument a FILE template with $CREATERTS, called a *creation template*. A FILE template can first be gotten by using the K-call $MAKETEMPLATE, supplying a capability for the FILE TYPE object with $TEMPLATERTS.

Note that as a matter of policy, a subsystem may not allow users to have access to creation templates. Instead, a user who wishes to create an object must call upon a procedure in the subsystem which does have access to the creation templates. Thus, the subsystem is able to create an object and initialize it in a well-defined way, and then protect it from the user by restricting appropriate rights. The subsystem may also perform resource control or accounting, and refuse to create an object if no resources are available or the user has exceeded some quota.

Initially, Hydra provides templates for each Kernel type (though users may not directly be able to access these). These templates do not have all Generic rights, but rather a restricted set, depending on the type. For these rights limitations, see section A.5. Note that one does not need access to a Kernel TYPE object in order to obtain templates for Kernel types. See the $MAKETEMPLATE K-call in section 2.2.7.2.

$CREATE may expect some additional arguments when creating an object of a Kernel type[28]. For example, in creating a new TYPE object, $CREATE 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:

$CreatePNAME - word 0 of the type's *print name*. While all objects have a 64 bit bit unique name, TYPE objects also have a 10-byte print name. The K-call $OBJINFO, given a capability for an object, produces (among other information), the print name of its type.

$CreateCapaInit and $CreateCapaMax - 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. Either or both of these may be zero, but it must be true that $CreateCapaInit ≤ $CreateCapaMax.

$CreateDataInit and $CreateDataMax - the initial length of the data-part (zeroed) and the maximum length of the data-part of any object of the type created. As with the C-list limits, $CreateDataInit ≤ $CreateDataMax.

28 The complete set of Kernel types, their symbols, and their creation arguments, is given in section A.5).

$CreateTempFlag - an indication as to whether objects of this type are to be retained between activations of Hydra. If $CreateTempFlag is set in the TYPE representative of an object, then that object will (effectively) be destroyed whenever the system is shut down. Such objects are used to
contain transient status, e.g., open-file records. *
$CreateRetrieveFlag - 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 space *
occupied by the object is normally reclaimed. However, it is possible to *
retrieve such objects and prevent reclamation on a type-by-type basis (see *
RTRVFLAG above). The K-call $TYPERETRIEVE returns a capability for an *
object, all of whose references have been deleted (including aliases). To *
actually reclaim a retrievable object, the K-call $ERASE rather than *
$DELETE must be used to delete the last capability for the object. Aliasing *
objects are never retrieved. *

The fields for a type creation block are defined in CREATE.REQ[N811HY97].

2.2.6 Protected Subsystems

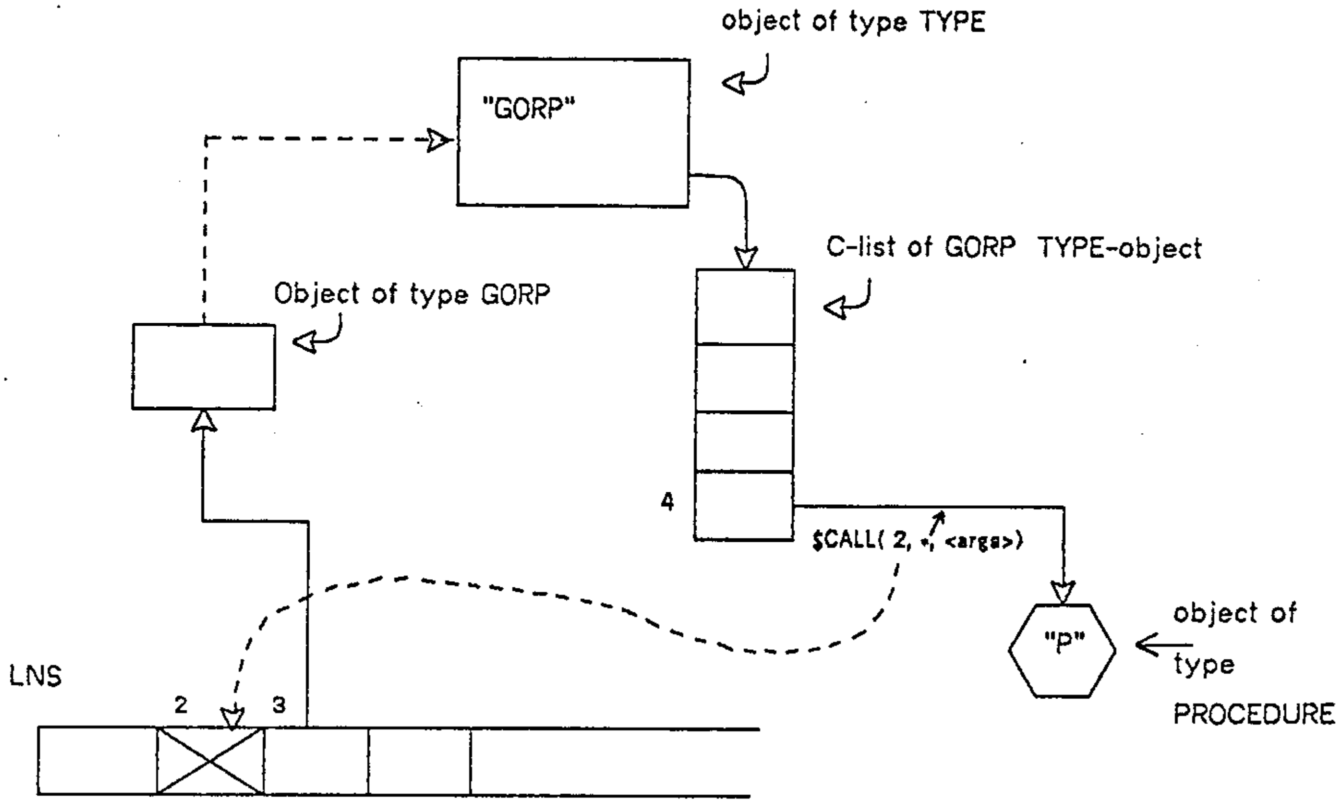
Since *protected subsystems* are generally built around a particular type of object (e.g. the Directory subsystem mentioned earlier), Hydra provides a way to use a subsystem without unnecessarily proliferating capabilities for the procedures which define it.

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 K-call $TYPECALL.

If the Ndx'th capability in the current LNS is of type T, and we use the notation T_j to indicate the j'th capability in the C-list of the T-TYPE object, then $TYPECALL (Rtrn,Ndx,j,$a_2,\ldots,a_k$) is the same as $CALL(Rtrn,$T_j,a_2,\ldots,a_k$). See figure 5.

There are several advantages to using the $TYPECALL mechanism. Two major advantages are:

> Users do not have direct access to capabilties for the subsystem procedures. Thus new procedures may be installed by changing the capabilities in the TYPE object. This makes management of complex systems much easier.



$TYPECALL(2,3,4,<args>)

Figure 5: A $TYPECALL example

> Standard interfaces can be provided. For example, the convention may be adopted that slot 2 of the TYPE object contains the creation procedure capability.

2.2.7 Specifications for Intermediate Kernel K-calls

2.2.7.1 Creation of TYPE objects

$CREATE (DIndex, SType, SMemR10)

<u>Parameters:</u>

DIndex	-	Simple index, empty
SType	-	Simple index, TYPE template, $CREATERTS.
SMemR10	-	Legitimate stack memory address. For details of contents, see section 2.2.5.

Effect:

Creates a TYPE object for a user-defined type. The type print name, C-list limits, data-part limits, temporary flag and retrievability flag are all set according to the contents of SMemR10.

Signals:

This K-call will signal if any of the parameters in the type creation block are out of range or inconsistent. The index of the offending word for the signal will be placed in $SIGDATA.

$HySTYPEBOUND - One of $CreateCapaInit, $CreateCapaMax, $CreateDataInit or $CreateDataMax larger than the implementation-defined maximum; or $CreateCapaInit > $CreateCapaMax or $CreateDataInit > $CreateDataMax.

Result:

0

$CHANGETYPESPECS (DIndex, SMemR10)

Parameters:

DIndex - TYPE object reference; Steps and Pretarget: $GETCAPARTS, $UNCFRTS; Target: $CHANGETYPERTS, $MODIFYRTS

SMemR10 - Legitimate stack memory address

Effect:

Allows the parameters set at creation of a TYPE object to be altered. Note that these changes will affect only future creations done for objects of this TYPE. The only change to existing objects is that the print name is changed. Note: if a -1 (#177777) is stored in a word of the type specification block, the contents of that field in the TYPE object is not altered.

Signals:

This K-call will signal if any of the parameters in the type specification block are out of range or inconsistent. The index of the offending word for the signal will be placed in $SIGDATA.

$HySTYPEBOUND - One of $CreateCapaInit, $CreateCapaMax, $CreateDataInit or $CreateDataMax larger than the implementation-defined maximum; or $CreateCapaInit > $CreateCapaMax or $CreateDataInit > $CreateDataMax.

Result:

0

2.2.7.2 TEMPLATE Manipulation

$MAKETEMPLATE (DPath, SIndex, Smemrts)

Parameters:

DPath	- Path index; Steps: $GETCAPARTS, $UNCFRTS; Pretarget: $PUTCAPARTS, $MODIFYRTS; Target: empty
SIndex	- Simple index, type TYPE, $TEMPLATERTS or a negative integer (see below)
Smemrts	- Legitimate stack memory address, or 0

Effect:

If SIndex is a simple index, then $MAKETEMPLATE places a template in DPath's Target whose type is the name of the SIndex'th capability in the current LNS. The template will only have $UNCFRTS set if SIndex has $UNCFRTS. The template will have all other flags and rights (both Generic and Auxiliary) set except for $REALLYRTS.

If SIndex is negative, then a template for the (-SIndex)'th Kernel type is placed in DPath's Target with $TemplateFlag set as well as various rights depending on the type. (See section A.5.) The first 13 types are the predefined Kernel types. The symbols which map these names into types are described in section A.5 and in the Bliss require file TYPES.REQ[N811HY97]. Note that the symbols given in this file are positive integers, and must be negated by the user when used in this K-call.

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

Result:

0

$SETCHKRIGHTS (DPath, SMemrts)

<u>Parameters:</u>

DPath	- Path index; Steps: $GETCAPARTS, $UNCFRTS; Pretarget: $GETCAPARTS, $PUTCAPARTS, $KILLRTS, $MODIFYRTS; Target: template, $DELETERTS
SMemrts	- Legitimate stack memory address

<u>Effect:</u>

Sets the check-rights of the template at DPath according to the contents of SMemrts.

<u>Result:</u>

0

2.2.7.3 Object Manipulation

$CREATE (DIndex, SIndex, <arguments>)

<u>Parameters:</u>

DIndex	- Simple index, empty
SIndex	- Simple index, template, $CREATERTS; must not be NULL; Also requires $UNCFRTS if the type is Retrievable
	- For description of additional <arguments> (only applicable when $CREATEing a Kernel object) see section A.5.

<u>Effect:</u>

Creates a new object of the same type as SIndex and places a capability for it in DIndex. The rights in DIndex are the same as those in SIndex except that $FREEZEFLAG will be removed, $DELETERTS, $ENVRTS, and $MODIFYRTS will be added, $UNCFRTS will be added unless SIndex is a procedure template and the current LNS is confined.

<u>Result:</u>

0

$COPY (DIndex, SIndex, <arguments>)

<u>Parameters:</u>

DIndex	-	Simple index, empty
SIndex	-	Simple index, object reference, $COPYRTS
	-	For description of additional <arguments> (only applicable when $COPYing a Kernel object) see section A.5.

<u>Effect:</u>

Creates a new object of the same type as SIndex and places a capability for it in DIndex. In addition, the contents of the C-list and data-part of the new object will be the same as those of the original.

The rights of the new capability in DIndex will be exactly the same as those for SIndex plus $DELETERTS, unless the object is of a Kernel type, in which case additional rights may be added. Note that some Kernel types cannot be copied. See section A.5 for details.

<u>Result:</u>

0

*
*
*
$DESTROY (DPath)

*
* <u>Parameters:</u>
*
* DPath - Path index; Steps and Pretarget: $GETCAPARTS,
* $UNCFRTS; Target: object reference,
* $MODIFYRTS, $OBJRTS
* <u>Effect:</u>
* Destroys the object referenced by the Target and deletes all
* capabilities in the object's C-list.
* Future accesses of the object will fail with either
* $HySCBOUND or $HySDBOUND signals.
* When the last capability for the object is deleted, the object
* will not be retrieved.

<u>Result:</u>

0

$SWITCH (DPath, DIndex)

Parameters:

DPath	- Path index; steps and pretarget: $GETCAPARTS, $UNCFRTS; Target: object reference, $MODIFYRTS, $OBJRTS, $UNCFRTS
DIndex	- Simple index, same type as DPath's target, $OBJRTS, $UNCFRTS,$MODIFYRTS

Effect:

Switches the C-list and data-part of the objects referenced by DPath's target and DIndex.

Signals:

$HySCANTLOCK-The object referenced by DIndex cannot be immediately locked.

Result:

0

*
*

$FREEZE (DIndex, SIndex, <arguments>) *

*

Parameters: *

*

DIndex	- Simple index, empty *
SIndex	- Simple index, object reference, $OBJRTS, $MODIFYRTS; SIndex must not be an alias; each capability in C-list of the object must have $FREEZEFLAG *
	- For description of additional <arguments> (only applicable when performing $FREEZE on a Kernel object) see section A.5. *

Effect: *

Performs a $COPY of the object, then sets $FREEZEFLAG and turns off $UNCFRTS and $MODIFYRTS in DIndex. Otherwise the same as $COPY. *

Signals: *

$HySFREEZE- Some capability in the object's C-list is not frozen. $SIGDATA indicates the index of the last such capability. *

$HySNOTUNIQUE-DIndex is not the only reference to the object. *

$HySALIAS - DIndex references an alias. *

Result: *

0 *

$MAKEALIAS (DIndex, SIndex)

Parameters:

DIndex - Simple index, empty

SIndex - Simple index, object reference

Effect:

Creates an object in DIndex of the same type as SIndex to act as an alias for the object referenced by SIndex. Any future references to to the new object (unless changed by $REALLY) will in fact access SIndex's Terminal object. DIndex will have the same rights as SIndex except $DELETERTS and $REALLYRTS will be added and it will not have $FREEZEFLAG.

Result:

0

$REVOKE (DIndex)

Parameters:

DIndex - Simple index, $REALLYRTS

Effect:

Revokes alias until re-ally-ing occurs. Future references through the alias will fail with signal $HySNOALIAS.

Result:

0

$REALLY (DIndex, SIndex)

Parameters:

DIndex - Simple index, $REALLYRTS (insures aliasing object)

SIndex - Simple index, must reference DIndex's original alias. SIndex, except for $DELETERTS and $REALLYRTS, must have at least all the rights that DIndex has.

Effect:

Re-allies the object referenced by DIndex to be an alias for the object referenced by SIndex.

Result:

0

$TYPERETRIEVE (DIndex, SIndex)

<u>Parameters:</u>

DIndex	-	Simple index, empty; or 0
SIndex	-	Simple index, TYPE object reference, $UNCFRTS, $RETRIEVERTS

<u>Effect:</u>

If DIndex is not zero, retrieves a capability for an object of the type named by SIndex, 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 except $FREEZEFLAG and $REALLYRTS (aliasing objects are not retrieved). If DIndex is zero, the K-call is executed for its result value only.

<u>Result:</u>

Number of objects in SIndex's type's retrieval queue, including object retrieved, if any. Note a result of 0 indicates no object was retrieved.

$ERASE (DIndex)

<u>Parameters:</u>

DIndex	-	Simple index, must be only reference to object, $OBJRTS, $DELETERTS

<u>Effect:</u>

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.)

<u>Signals:</u>

$HySNOTUNIQUE-DIndex is not the only reference to the object

<u>Result:</u>

0

2.2.7.4 The CALL Mechanism

$MERGE (DIndex, STempl, SPath)

<u>Parameters:</u>

DIndex	-	Simple index, empty
STempl	-	Simple index, template, $TemplateFlag
SPath	-	Path index; Pretarget: $GETCAPARTS; Target: defined, rights must contain all those specified by check-rights field of STempl. If STempl is not NULL, must be an object reference and must be of the same type as STempl. If STempl is NULL, may be of any type and may be either an object reference or a template.

<u>Effect:</u>

Copies the capability targeted by SPath to the DIndex'th slot of the current LNS and sets $DELETERTS. If SPath's Target is a capability for an aliasing object and STempl has $AmplifyFlag set, a capability for the alias's terminal object is copied instead.

If STempl has $AmplifyFlag set, STempl's (new-rights) are copied to DIndex, except for $ENVRTS, $UNCFRTS, $MODIFYRTS and $FREEZEFLAG which must appear in SPath's Target as well.

If any capability in the Path's steps or pretarget lacked $UNCFRTS, then $MODIFYRTS, $UNCFRTS and $REALLYRTS will be removed from DIndex. If any capability in the SPath lacked $ENVRTS, then $ENVRTS will be removed from DIndex.

<u>Signals:</u>

$HySCHECKRTS-Check-rights failure
$HySMERGE- STempl is not a template or does not have $TemplateFlag set.
$HySSTYPE - Types of SPath's Target and STempl are not the same.

<u>Result:</u>

0

$MAKELNS (DIndex, Nproc, <arguments>) *

Parameters:

DIndex - Simple index, empty

Nproc - Simple index, procedure object reference; $LNSRTS

The 0 or more <arguments> must each be of the following form:

> $PATH (SPath)
Path index; Pretarget: $GETCAPARTS;
Target: Requires $ENVRTS if Nproc has $PROCESSRTS

> $RESTRICTION (SPath, Smemrts)
SPath: as for $PATH
Smemrts: Legitimate stack memory address, or 0

> $TRANSFER (SPath, Smemrts)
SPath: Path index;
Steps: $UNCFRTS, $GETCAPARTS;
Pretarget: $MODIFYRTS, $GETCAPARTS, $KILLRTS;
Target: $DELETERTS, also requires $ENVRTS if Nproc has $PROCESSRTS.
Smemrts: Legitimate stack memory address, or 0

> $MEMDATA (MemRn, Count)
MemRn: Legitimate user memory address
Count: Positive integer

> $STKDATA (<data>)[29] X
<data>: 0 or more expressions, each of X
which generates one word of data X

[29] An obsolete contruct which is included here for completeness. Users may encounter it in older code and confuse it with $STACKDATA.

> $STACKDATA (<data>)
<data>: 0 or more expressions, each of which generates one word of data

> $LNS ()

> $LNSRESTRICT (Smemrts)
Smemrts: Legitimate stack memory address, or 0

Arguments $LNS and $LNSRESTRICT are permitted only if Nproc does not have both $PROCESSRTS and $ENVRTS. The capability denoted by each argument must also satisfy the requirements of its corresponding parameter template (see $MERGE).

* <u>Effect:</u>
* An LNS is incarnated from the procedure and arguments and
* a capability for it is placed in DIndex with $DELETERTS. In
* addition it will have $UNCFRTS and $FREEZEFLAG, and the
* auxiliary rights $LNSRTS and $PROCESSRTS if Nproc does. If
* Nproc lacks $PROCESSRTS or $ENVRTS, the Capability for the LNS
* will lack $ENVRTS.
* The LNS will be made confined if Nproc lacks $UNCFRTS.

All capabilities in the C-list of the PROCEDURE which are either object references or capability templates ($TemplateFlag set) are copied to the same slot in the C-list of the incarnated LNS. These are the *inherited capabilities*. If Nproc lacks $UNCFRTS, each of these will have $UNCFRTS, $MODIFYRTS and $REALLYRTS removed. If Nproc lacks $ENVRTS, each inherited capability will have $ENVRTS removed.

Parameter templates in the C-list of the PROCEDURE are capabilities specified by the arguments. Arguments are matched with parameter templates in descending slot number order. The "rightmost" parameter is merged to the highest-numbered parameter slot, the "next-rightmost" parameter to the next-highest-numbered parameter slot, etc., until all parameters have been merged. If insufficient parameters are provided, the remaining lower-numbered parameter slots are filled with *unbound* templates (see section 2.3.1).

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 'Effect' part of the $MERGE K-call. As noted, arguments come in 7 flavors. The capabilities they specify and additional side effects are as follows:

> $PATH: Capability is SPath's Target.

> $RESTRICTION: Capability is SPath's Target, restricted by the contents of Smemrts if Smemrts is non-zero.

> $TRANSFER: Capability is SPath's Target, restricted by the contents of Smemrts if Smemrts is non-zero. In addition, the capability at SPath's Target is deleted. (N.B. Use wisely, because if the K-call fails the capability may be lost.) This option is equivalent to the $PASS K-call.

> $MEMDATA: Capability is for a newly created DATA object with all rights but $FREEZEFLAG and $REALLYRTS. The data-part of the new object will contain the Count words of data copied from the block of memory beginning at MemRn.

> $STKDATA: Capability is for a newly created DATA object X
> with all rights but $FREEZEFLAG and $REALLYRTS. The X
> data-part of the new object will consist of '<data>' in X
> reverse order. Thus $STKDATA (11,22,33) produces a data X
> object containing words 33, 22, and 11 in positions 1, 2, X
> and 3, respectively. X

> $STACKDATA: What $STKDATA was really meant to be. Creates a capability for a newly-created DATA object with all rights but $FREEZEFLAG and $REALLYRTS. The data part of the new object will consist of '<data>' in the correct order. $STACKDATA (11,22,33) produces a data object containing words 11, 22, and 33 in positions 1, 2, and 3, respectively.

> $LNS: Capability is for the caller's LNS with $DELETERTS, $MODIFYRTS, $UNCFRTS, $GETCAPARTS, $PUTCAPARTS, $APPENDCAPARTS, $KILLRTS, $GETCBRTS, $SETCBRTS, $GETSTACKRTS, and $PUTSTACKRTS.

> $LNSRESTRICT: Capability is as in $LNS with rights additionally restricted by the the contents of Smemrts if Smemrts is non-zero.

Signals:

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

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

$HySCANTCONFINE-LNS is not allowed to be made confined. (See section 2.3.1.)

If an argument is bad or any merge failed, the usual signal will be generated with $HySMAKELNS or'ed in as well. In addition, the fixed location $SIGDATA in the stack page will contain 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.

Result:

0

*
*
* $LNSCALL (DIndex, NIns)
*
* Parameters:
*
* DIndex - Simple index, empty
*
* NIns - Simple index, LNS object reference, $LNSRTS;
*
* The LNS must be "usable" (see sections 2.3.2 and
* 3.1.1)
* Effect:
* The LNS is called and execution begins in its environment.
* When the called LNS $SUSPENDs, it may specify a capability to be
* returned. If DIndex is not zero, it designates the slot where that
* capability will be put. If DIndex is zero, a returned capability is
* simply discarded.
* Signals:
? $HySNOSTACK-Inadequate stack space available to run the LNS
? (see section 2.3.1). $SIGDATA contains amount
? of additional stack space needed.

$HySCONTROL-Callee returned by 'Punting a Control' rather than a $SUSPEND (see section 2.3.1).

$HySCANTLOCK-LNS is currently in use (see section 3.1.1).

$HySREUSE - LNS may not be reused (see section 2.3.3).

For paging-related signals, see section 4.7.1.

When the callee $SUSPENDs, it specifies a return value. If that value is negative, it is treated as a signal.

<u>Result:</u>

Value returned by the callee

$CALL (DIndex, Sproc, <arguments>)

<u>Parameters:</u>

DIndex	- Simple index, empty, or 0
Sproc	- Path, procedure object reference, $CALLRTS
	- Specifications for <arguments> are exactly as for $MAKELNS.

<u>Effect:</u>

The effect is almost equivalent to the sequence

```
$MAKELNS ( *, Nproc, <arguments> );
$LNSCALL ( DIndex, * ).
```

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 parameter template, an argument's target does not require $ENVRTS, regardless of whether or not Nproc has $PROCESSRTS. $LNS and $LNSRESTRICT are always allowed.

<u>Signals:</u>

See $MAKELNS and $LNSCALL

<u>Result:</u>

Value returned by callee

$SUSPEND (Value, SIndex, Smemrts)

<u>Parameters:</u>

Value	- Integer
SIndex	- Simple index, $ENVRTS, or 0
Smemrts	- Legitimate stack memory address, or 0

<u>Effect:</u>

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 Return slot and SIndex is non-zero (and the return slot has not otherwise had a capability stored into it), the capability denoted by SIndex is returned to that slot in the caller's LNS with rights restricted by the contents of Smemrts (if Smemrts is not zero) and with $DELETERTS added.

If the current LNS has no caller, the current PROCESS will be stopped. Attempts to restart it will be unsuccessful. See sections 3.1.7.4 and G.3.

Result:

Current value of R0. Control returns to caller (unless a signal occurs). Control only continues normally after a $SUSPEND if the current LNS is subsequently $LNSCALLed again.

$RETURN (Value, SIndex, Smemrts)

Parameters:

Value	- Integer
SIndex	- Simple index, $ENVRTS, or 0
Smemrts	- 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 in the caller's environment. If the caller specified a Return slot and SIndex is non-zero (and the return slot has not otherwise had a capability $PUTCAPAd into it), the capability denoted by SIndex is returned to that slot in the caller's LNS with rights restricted by the contents of Smemrts (if Smemrts is nonzero), and with $DELETERTS 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 R0. Control returns to caller (unless a signal occurs). Any attempt to resume execution via $LNSCALL will result in a signal $HySREUSE[30].

30 Actually, $RETURN is not a separate K-call; it is a macro which expands into a $SUSPEND, followed by a loop containing a $SUSPEND which signals $HySREUSE. There are several reasons for renaming the old $RETURN K-call to be $SUSPEND and creating this macro. They deal primarily with concepts of program clarity and ease of understanding; it is (or should be) perfectly obvious to both programmer and reader of code that a procedure is intended to resume after a $SUSPEND but not after a $RETURN. It also means that an LNS which was not intended to be re-used will not be inadvertently re-used because someone forgot to set the re-use flag.

2.2.7.5 Protected Subsystems

$TYPECALL (Dindex, SPath, TPath, <arguments>)

Parameters:

Dindex	- Simple index, empty or 0
SPath	- Path, defined
TPath	- Path beginning in the C-list of the TYPE object whose name is the type of SPath, PROCEDURE object reference, $CALLRTS
	- Specifications for <arguments> are exactly as for $MAKELNS.

Effect:

If we let TYP_{SPath} be the TYPE representative for the capability (object or template) targeted by SPath, then the effect is (roughly) equivalent to:

```
$GETCAPA (*,TYP_SPath);

$CALL ( Dindex, $PATH(*,TPath), <arguments>);
```

that is, the Kernel $CALLs the procedure in the type object without the caller getting a capability itself for the procedure. See section 2.2.6 and Figure 5.

Signals:

See $CALL

Result:

Value returned by callee

Appendix K: K-call macro summary

This appendix summarizes the Kernel calls and their parameters and provides a quick reference both to the calls and to their descriptions in the manual. The flags in the left column are those which appear with the Kernel call in its description in the manual, and are explained in section 1.1.4.

	$APPENDCAPA (DPath, SIndex, Smemrts)	2-25
	$APPENDDATA (DPath, MemRn, Count)	2-21
P	$ATTACHPOLICY (Nprcs, Npol, SBase)	3-15
	$BASECALL (DIndex, SPath, arguments)	3-10
	$CALL (DIndex, Sproc, arguments)	2-53
	$CHANGETYPESPECS (DIndex, SMemR10)	2-41
	$CLENGTH (SPath)	2-17
	$COMPARE (SPath, SIndex)	2-18
	$CONNECT (Port1, Outchan, Port2, Inchan, Connid)	6-9
P	$CONTROL (Nprcs, Code)	3-12
	$COPY (DIndex, SIndex, arguments)	2-44
	$COPY (DIndex, SPage, Ncps)	4-5
	$CPSLOAD (DIns, Pairlist)	4-6
	$CREATE (DIndex, SType, SMemR10)	2-40
	$CREATE (DIndex, SIndex, arguments)	2-43
	$CREATE (DIndex, SPrcs, SLns)	3-2
	$CREATE (DIndex, SPrcs, args, SProc)	3-3
	$CREATE (DIndex, SPol, Data)	3-14
	$CREATE (DIndex, SPort, NMsgSlots, NDummy, NOutchan, NResources)	6-8
	$CREATE (DIndex, SSem, Count)	7-1
	$CREATE (DIndex, SPage, Flags)	H-1
	$CREATE (DIndex, SPort, NMsgSlots, NDummy, NOutchan, NResources)	H-2
	$CREATE (DIndex, SSem, Count)	H-3
	$CREATE (DIndex, SPrcs, SLns)	H-3
	$CREATE (DIndex, SPrcs, args, SProc)	H-4
	$CREATE (DIndex, SProc)	H-5
	$CREATE (DIndex, SPol, Data)	H-6
	$DELETE (DPath)	2-25
	$DELETEMSGCAPA (Port, MsgSlot)	6-33
*	$DESTROY (DPath)	2-44

	$DISCONNECT (Port, Outchan)	6-11
	$DLENGTH (SPath)	2-17
*	$ERASE (DIndex)	2-47
*	$FREEZE (DIndex, SIndex, arguments)	2-45
	$GETCAPA[16] (DIndex, SPath)	2-24
X	$GETCLOCK (SMemW4)	2-29
	$GETDATA (MemWn, SPath, Disp, Count)	2-19
	$GETGMT (MemW4)	2-27
	$GETICB (SMemWn, SPath, Code)	2-63
	$GETLCB (SMemWn, SPath, Code)	2-64
	$GETMSGCAPA (Port, MsgSlot, DIndex)	6-33
P	$GETPCB (SMemWn, SPath, Code)	3-9
P	$GETPOLICY (SMemW16, Npol)	3-15
	$GETPROCESSID ( )	3-9
	$GETSTACK (SMemWn, SLns, MemLns, Count)	2-65
	$GETUPTIME (MemW4)	2-28
	$GMTTOLOCAL (MemWn, Count)	2-28
	$GNAMETOGMT (MemW4,MemR4)	2-27
P	$INFPOLICY ( )	3-17
	$INTERCHANGE (DPath, DIndex, Smemrts)	2-26
	$KALLINDIRECT (SMemRn)	2-29
*	$LNSCALL (DIndex, Nlns)	2-52
	$LNSLENGTH ( )	2-17
*	$MAKEALIAS (DIndex, SIndex)	2-46
	$MAKEDATA (DPath, MemRn, Count, Smemrts)	2-20
*	$MAKELNS (DIndex, Nproc, arguments)	2-49
	$MAKEMSG (Port, Buflen, Stkdep)	6-12
	$MAKEPAGE (DPath)	4-6
	$MAKETEMPLATE (DPath, SIndex, Smemrts)	2-42
	$MAKETEMPLATE (DPath, SIndex, Smemrts)	H-7
	$MAKEUNIVERSAL (DPath)	2-22
	$MERGE (DIndex, STempl, SPath)	2-48
	$NCONNECT (Port1, Outchan, Port2, Inchan, Connid, Disconnopt)	6-10
	$OBJINFO (SMemW16, SPath)	2-18
	$P (DIndex)	7-2
	$PASS (DPath, SIndex, SMemrts)	2-22
	$PASSAPPEND (DPath, SIndex, Smemrts)	2-24

[16] Also known, for historical reasons, as $LOAD.

[17] Also known, for historical reasons, as $STORE.

	$TYPECALL (Dindex, SPath, TPath, arguments)	2-
*	$TYPERETRIEVE (DIndex, SIndex)	2-
	$UPDATE (SPath)	5
*	$UPDATEN (SPath,Depth)	5
	$V (DIndex)	7
	$VACATE (DPath)	2-
	$VALL (DIndex)	7
P	$WHATPOLICY (SMemW16, Npol)	3-
*	$WORKSET (NIns, Size)	4
	$WRITEMSG (Port, MsgSlot, Pos, Len, MemRn)	6-

Appendix L: Bibliography

[Coh75] Cohen, E., et. al., "Protection in the Hydra Operating System, in **Proceedings of the Fifth Symposium on Operating Systems Principles**, (SOSP5), November, 1975.

[Org72] Organick, E. **The Multics System: An Examination of its Structure**, MIT Press, 1972

[Reid75] Reid, B. K., **The Hydra Songbook: A Vigilante Users' Manual**, Computer Science Department, Carnegie-Mellon University, 1975

[WB72] Wulf, W. A., and Bell, C. G., "C.mmp---a multi-mini-processor", Proc. AFIPS 1972, FJCC. Vol 41, AFIPS Press, Montvale, N.J., pp. 765-777

[Wu74] Wulf, W. A., et. al. "Overview of the Hydra Operating System Development", in **Proceedings of the Fifth Symposium on Operating Systems Principles**, (SOSP5), November, 1975.

[Wu74] Wulf, W. A. et. al. "Hydra: The Kernel of a Multiprocessor Operating System", CACM 17,6 (June 1974), pp. 337-345

INDEX