UNIVERSITY OF ILLINOIS LIBRARY
AT URBANA-CHAMPAIGN
ENGINEERING
NOTICE: Return or renew all Library Materials! The Minimum Fee for each Lost Book is $50.00.

The person charging this material is responsible for its return to the library from which it was withdrawn on or before the Latest Date stamped below.

Theft, mutilation, and underlining of books are reasons for disciplinary action and may result in dismissal from the University.
To renew call Telephone Center, 333-8400

UNIVERSITY OF ILLINOIS LIBRARY AT URBANA-CHAMPAIGN
Proposed Unix Modifications for a National Software Works Implementation

by

Steven F. Holmgren

September 20, 1976
SEP 13 1977

The person charging this material is responsible for its return to the library from which it was withdrawn on or before the Latest Date stamped below.

Theft, mutilation, and underlining of books are reasons for disciplinary action and may result in dismissal from the University.

UNIVERSITY OF ILLINOIS LIBRARY AT URBANA-CHAMPAIGN

ENGINEERING
CONFERENCE ROOM
Proposed Unix Modifications for a National Software Works Implementation

Prepared for the Defense Advanced Research Projects Agency

Under Contract ARPA/MDA 903-76-C-0345

Steven F. Holmgren
Center for Advanced Computation
University of Illinois
Urbana, Illinois 61801

September 20, 1976
Table of Contents

1. Introduction ................................................................................................................. 1
   1.1 The National Software Works ................................................................................. 1
   1.2 The UNIX System ................................................................................................. 2
   1.3 This Paper ............................................................................................................ 2

2. MSG ............................................................................................................................. 4
   2.1 Messages ................................................................................................................ 4
   2.2 Alarms .................................................................................................................... 4
   2.3 UNIX-MSG Considerations .................................................................................... 5
      2.3.1 Pipes .............................................................................................................. 5
      2.3.2 Signals ........................................................................................................... 6
   2.4 Proposed Solution ................................................................................................... 7
      2.4.1 Events ............................................................................................................ 8
         2.4.1.1 Wait_event ............................................................................................. 8
         2.4.1.2 Sndevent ............................................................................................... 8
         2.4.1.3 Snd_g_event .......................................................................................... 9
      2.4.2 Messages ....................................................................................................... 9
         2.4.2.1 Rcvseg .................................................................................................. 10
         2.4.2.2 Sndseg .................................................................................................. 10
         2.4.2.3 Snd_g_seg .............................................................................................. 10

3. Foreman ....................................................................................................................... 12
   3.1 Foreman-Unix Considerations .............................................................................. 12
      3.1.1 Unix-Foreman Process Control .................................................................... 13
      3.1.2 Unix-Foreman Encapsulation .................................................................... 14

4. File Package ................................................................................................................. 15
5. CLI-Frontend..................................................16
6. NSW IO Station..............................................17
6.1 UNIX-IOS Considerations.................................17
6.2 Simplified Netprinter Daemon............................19
7. Required Unix Modifications..............................20
7.1 Unix-IPC....................................................20
7.2 Addressing..................................................21
7.3 Events........................................................22
7.3.1 Sndevent and Snd_g_event..............................22
7.3.2 Wait_event................................................23
7.4 Messages....................................................24
7.4.1 Receiving a Message....................................26
7.4.2 Sending a Message......................................27
7.5 Message Primitive Definition............................27
7.5.1 GETSEA....................................................27
7.5.2 RCVSEG....................................................26
7.5.3 SNDSEG and SND_G_SEG.................................29
7.5.4 GETSEG and FREESEG....................................29
8. Non-Blocking I/O............................................31
8.1 Process Control.............................................32
1. Introduction

The construction of computer software is expensive. Highly trained and expensive programming talent is required for the construction of software development tools, viz., editors, debuggers, and compilers. The quality of their implementation defines in turn the quality of the environment in which later software construction will take place. In many cases the hardware, software, and human talent required for the construction of a particular piece of software are geographically dispersed throughout the nodes of a computer network. Distributed access to a single set of well-implemented development tools is an important economic goal.

1.1 The National Software Works

The National Software Works (NSW) is an attempt to provide access to just such a uniform, high-quality development environment.

The National Software Works EXEC (NSWXEXEC) is an operating system designed to support such an environment.

The NSWXEXEC is comprised of several well-defined components: A Works Manager (coordinates distributed NSW activities), A Front End (provides a uniform terminal environment), a Foreman (provides an interface between the executing tool and the NSW), and the NSW tools themselves.
The NSW components are designed to function on distributed nodes of the ARPA network. An NSW service called the MSG provides a common communications mechanism for these distributed components.

During execution, tools may require access to a mass of information storage. An NSW service called the File Package provides the transportation and local management for a distributed NSW file system.

1.2 The UNIX System

The Unix operating system is a full-scale general-purpose time-sharing system written for the PDP-11 family of minicomputers.

An ARPA Network Control Program (NCP) has been constructed for the Unix system. The AhPA Network hosts are accessed directly through the Unix file system. A program opens an ARPA connection by opening an existing named file. Communications with the foreign host are carried on by executing read and write file system calls. A close system call closes the connection.

1.3 This Paper

The ARPA network Unix system is a natural vehicle for the construction of an NSW Tool Bearing Host.
This paper will describe the Unix modifications required for the support of the major NSW components: MSG, Foreman, File Package, Front End, and I/O Station.

No specific design of the components themselves is presented. However, in order to synthesize the base operating system facilities required by a component, a rough design for the construction of a Unix version was carried out. These rough designs served as a basis for the discovery of applicable existing Unix facilities and the generation of required new mechanisms.

This paper is divided into two segments. The first segment quickly describes each component, the discovered applicable Unix facilities, and new mechanisms. The second segment describes more fully the mechanisms which must be constructed before an NSW implementation may take place.
2. MSG

MSG is the mechanism whereby distributed portions of the National Software Works communicate.

MSG must respond asynchronously to requests from local NSW components and foreign MSG counterparts. In general these requests ask either to create a communications path, or to pass data along either an implied or existing path. NSW processes may open or close specific communications paths.

Data is routed to NSW processes in two forms: messages and alarms.

2.1 Messages

Messages may be "addressed" to a specific NSW process or to a generic class of processes. When MSG accepts a generically-addressed message, it selects a specific destination process from a class that has indicated a willingness to receive such a message. If there is no such process, MSG may create one.

Primitives exist to send and receive both types of messages. Messages are not necessarily received in the order in which they were sent. Sequenced message communications are available.

2.2 Alarms

Alarms are specifically-addressed software events. NSW processes may send, receive, enable, and disable alarms. When a process enables alarms, it states an interest in using the alarm
form of communications. Such a process is expected to routinely envoke a "receivealarms" primitive and take appropriate action.

Message and Alarm calls are non-blocking. Each has a timeout and signal parameter. The timeout parameter specifies the amount of time MSG must wait before aborting the request. The signal parameter allows MSG to indicate that a response is available.

Complete MSG specification may be found in MSG Design Specification, Bolt, Beranek, and Newman Resegment No. 3237.

2.3 UNIX-MSG Considerations

MSG must ultimately rely on the local operating system for process-to-process message and alarm communication.

The Unix system supports two inter-process communication mechanisms: pipes and signals.

2.3.1 Pipes

Pipes are a means for two processes with a common parent process to transfer large amounts of data between one another. Pipes have been implemented within the I/O system. Their use is transparent to two processes expecting to read or write from a disk file. Read and write system calls are blocking. When a read is issued by a process, that process is de-scheduled until the read completes. When data is being read from a terminal, the completion time is user dependent. During this time data may become available from another source. The process is inhibited
from handling this alternate data until the primary read is satisfied. There are NSW components, the MSG, the File Package, and the Foreman, which require simultaneous access to input from several sources.

The implementation of the pipe mechanism is somewhat inefficient; the locking strategy is conservative, messages may be routed onto disk storage until called for. This inefficiency is unacceptable in an environment where all the communications between NSW entities rely on inter-process communication.

Pipes require a common parent for initial setup. NSW components require the dynamic construction of message paths. For example, MSG must continuously serve the needs of its associated NSW components. Many of those components will be under the parentage of other NSW components which also require direct parental control. The Foreman is a prime example.

Because pipes are blocking and require a common parent for creation, they are unsuitable for NSW communications.

2.3.2 Signals

The second form of process-to-process communication is called signals. Unix signals are principally used by the system to communicate error situations to a user process: illegal memory references, illegal instructions, etc. The user process may elect to intercept such occurrences and take corrective action. Designed for low-frequency usage, this mechanism is deficient as a basis for MSG alarms. Unix signals are not buffered, that is,
a process must constantly re-issue a system call enabling it to intercept a particular signal. Should a process be handling a signal, subsequent signals of the same type would be lost until the "intercept" could be re-issued. Intercepts should not be re-issued until processing of a signal is complete. When a Unix signal is generated, the program counter of the afflicted process is forced to the intercept procedure address and the user stack is made to "look like" a procedure was called. Control is then returned to the user level of the process. If an intercept procedure were to re-arm the signal with its own address, and another signal were then to come along, these procedures might interrupt upon themselves, thus leading to the unacceptable requirement for user-level intercept procedures to lockout intercepts during critical sections of their processing.

Both Unix inter-process communication mechanisms are unsuitable for NSW process-to-MSG and MSG-to-MSG communications.

2.4 Proposed Solution

Two new mechanisms are proposed: events and messages. Both are direct analogs of NSW alarms and messages and are designed to make an MSG implementation relatively easy. Events and messages are also sufficiently general to be useful throughout the rest of the standard Unix system.
2.4.1 Events

Three event primitives are defined: wait_event, sndevent, and snd_g_event.

Events have four properties: an event source, an event destination, an event type or opcode, and a word of data. The contents of the type and data fields are of no relevance to the system. Each process will own an event queue where events are buffered until requested.

2.4.1.1 Wait_event

Wait_event takes a parameter which may be zero or a specific event type. A parameter of zero indicates that the process is waiting for an event of any type. A non-zero parameter indicates that the process is waiting for an event of that specific type. The event source, type, and data fields are all available for inspection when control returns from a wait_event call. If there are no queued events of the type requested, wait_event may deschedule the process until an event is put into its event queue.

2.4.1.2 Sndevent

Sndevent sends an event to another process. It has three parameters: a destination, an event type, and a data word. Control is returned immediately. The destination is called a "specific destination", and will generally be obtained from the source field of a previously received event or message.
2.4.1.3 Snd_g_event

Like sndevent, snd_g_event sends an event to another process. It has the same three parameters with the exception that the destination is a null-terminated string of ASCII characters specifying what is called a "generic" destination. Receiving processes will be allowed to specify the generic names they want to be "known" by with the "generic-name" primitive. A call on snd_g_event will attempt to match the name specified in the destination parameter field with the names previously defined to the system. If a match is found, the destination associated with the previously-defined name will receive the event. A process may be known by several names or aliases. All processes known by a particular name will receive a generically-addressed event to that name.

2.4.2 Messages

Messages are created, transmitted, and received within segments. Segment Descriptors (SD) is a thinly-disguised term for the PDP-11 memory-relocation registers. At any time, a segment descriptor may have a certain amount of segment memory associated with it. The amount of segment memory associated with a segment is referred to as the size of the segment.
Three transmit and receive primitives are defined: rcvseg, sndseg, and snd_g_seg.

2.4.2.1 Rcvseg

Rcvseg is used to retrieve queued messages for a process. If there are no queued messages, rcvseg may de-schedule the process until a message is put into its queue. Rcvseg has two parameters, a segment base address and a source. When control is returned, the segment base address is associated with segment memory to the extent of the return value. The source variable contains the specific address of the sending process. The message data may then be directly manipulated through the segment base address.

2.4.2.2 Sndseg

Sndseg is used to transmit a length of segment memory to another process. Sndseg parameters include a specific destination and a segment base address. The specific destination is generally obtained from the source field of a previously received event or message. The segment base address has a length of associated segment memory containing the message to be transmitted.

2.4.2.3 Snd_g_seg

Snd_g_seg is to sndseg as snd_g_event is to sndevent. The destination specification is a null-terminated ASCII name defining a process or group of processes which have previously
defined themselves to the system by that name. The function and other parameters are the same.
3. Foreman

The Foreman is charged with providing the tool to the NSW component interface. Every executing tool has an associated Foreman. The Foreman has two interfaces. One interface is with the Works Manager and to some extent the Front End. The other interface is between the Foreman and its executing tools. This interface may take on various levels of involvement. On one hand, all system entries may be trapped by the Foreman either to be passed on to the system or to be translated into an NSW request. The trapping of every system call is termed encapsulation. The encapsulation technique is used by Foremen which want to execute a tool that has little built-in knowledge of the NSW components. At the other extreme, the Foreman may simply respond to requests from the Works Manager to force a change in a tools state. The tool itself has built-in NSW knowledge and makes the distinction between calls that should be passed directly to the local operating system and requests which should be made of other NSW components.

3.1 Foreman-Unix Considerations

Foreman functions may divided into two categories: process manipulation (begin, end, stop, and continue) and file manipulation (get and put global NSW files, provide file semaphore functions, import export etc.).
Combinations of standard Unix filing system functions and the IPC functions outlined in MSG will enable the kind of file manipulation required by a Foreman.

Non-blocking peripheral I/O is required for Import/Export functions.

3.1.1 Unix-Foreman Process Control

Process manipulation functions are an integral part of the standard Unix system. These functions will support the kind of controls needed by a Foreman. The process control strategy relies on Unix signals to force a process to stop itself, communicate memory contents (one word at a time), and continue from a stopping point. This strategy implies a form of inter-process communication. The IPC used in process control is a special cased mechanism for the passing of data a word at a time between a child and its immediate parent.

This strategy could be streamlined by taking advantage of the new inter-process communication mechanism outlined in MSG. A controlling process could ask the system to stop a process. Then messages and events could travel back and forth containing chunks of memory, contents of registers, and so forth. With such a scheme, a single Foreman might be able to manage many tools of differing classes and parentage.
3.1.2 Unix-Foreman Encapsulation

Each Unix system entry has an associated piece of user-level interface software. For example, when a user executes an 'open' system call, control is transferred to a small piece of code that does some pre-processing of the parameter list and traps into the system. When control returns, this interface code takes care of assigning the results to their defined locations. Control is then returned to the user software. These routines are bound into a user program by the loader from system libraries. Foreman encapsulation may be implemented by installing a set of interface routines that send generic messages containing the system call type and parameters to the Foreman. Messages could pass back and forth between the Foreman and the interface routine to accomplish the same function as the older interface routine, with the exception that a Foreman has control over actual system entry functions. Such a set of routines could be used to encapsulate a large class of standard Unix programs.
4. File Package

The File Package provides suitable copies of NSW global files to executing tools. Each Tool Bearing Host must have a File Package with the implied access to local NSW file storage space. The File Package communicates files with foreign File Packages under the direction of the Works Manager. The File Package must be able to translate an NSW-defined file format to and from local file system structure.

The File Package requires a local, directory-structured filing system with open, close, read, write, create, and delete capabilities. These functions are an integral part of the standard Unix file system. NSW cataloguing functions may be accomplished by keeping a master file of information on each locally-stored global NSW file. Further, all peripheral devices may be manipulated through the file system. For example, a card reader has a file name, /dev/cr, which may be opened and read from, just as standard data files are manipulated.

The File Package should use the MSG primitives previously described to communicate with other local and non-local NSW entities.

All standard Unix file system calls are blocking. It is not critical that the close, write, create and delete functions be non-blocking because of their direct file system manipulation and return. However, the read and open system functions should be modified so that they may be used in a non-blocking manner.
5. CLI-Frontend

The Command Language Interpreter (CLI) Frontend is designed to provide "intelligent" access to the NSW.

The basic thrust of the CLI is to separate the terminal communications section of an interactive tool so that it may reside on a front-end machine "close" to the user. This allows the computational section of a program to run on larger, more suitable machines. This strategy has several advantages: minimizing the number of network transmissions, minimizing system response time, and providing a uniform terminal environment to the user.

As its name implies, the CLI is an interpreter. The CLI interprets a Backus-Naur-like specification for a tool command language. This specification is termed a grammar.

The CLI-FE utilizes a local MSG to provide the necessary communication functions with other NSW components.

A reasonable amount of disk storage will be required to store the various grammars in use at any one time.

The process control functions required by the Foreman are needed here for the various CLI tasks.

The Unix modifications required for a CLI-FE are: a non-blocking inter-process communication facility, non-blocking process control functions, and non-blocking file system I/O. The standard Unix file system and terminal-control functions are general enough to meet CLI-FE requirements.
6. NSW IO Station

The NSW I/O Station (IOS) is a service designed to transfer internally-formatted NSW files to and from a small class of PDP-11 peripherals (cardreader, lineprinter, papertape, etc.) and NSW Tool Bearing Hosts. The IOS is expected to co-habitate with a CLI Front End.

The basic IOS strategy is to have various "well-known" ARPA network sockets managed by the IOS. When a direct connection is completed to a socket, an implied request is made of one of the peripheral types; socket 10 for lineprinter, socket 12 for papertape, and so on.

Local users may also manipulate peripherals with an Export Control Program (ECP). The ECP gathers a user request, attempts to make a network connection to the specified Tool Bearing Host and transfers data to or from the peripheral, causes cards to be read, etc.

6.1 UNIX-IOS Considerations

The Network Unix Operating system can support this type of activity with no modifications. A user process may issue a blocked "listen" on a specific network socket. When control returns, the program can execute standard system reads or writes to pass data between itself, the network, and the Unix I/O system.
Due to the blocking nature of standard Unix I/O, a process must be allocated to each managed socket. The IPC mechanisms described for MSG could be employed here to allow a single process to wait for a connection completion on all of the IOS sockets. As connections are opened, this single process might choose to handle the data transfers itself or spawn a sub-task to handle the duties.

The Network Unix system has a "netprint daemon" that waits for simplex receive connections on Socket 6. Various hosts throughout the network have user-level programs which take text files and send them across such a connection directly to the local lineprinter. Below is a simplified but usable version of the netprinter daemon.
6.2 Simplified Netprinter Daemon

```c
int fdi;
int fdo;
int nbytes;
char buf[512];

/* loop forever */
while( 1 )
{
    /* select net open type */
    openparam.type = DIRECT ; LISTEN;
    /* listen on local socket 6 */
    openparam.lskt = 6;

    /* wait for a network connection */
    fdi = open("/dev/net/anyhost", &openparams);
    /* open the line printer */
    fdo = open("/dev/lp", 1);

    /* while there is data from the net */
    while( (nbytes = read( fdi, buf, 512 )) > 0 )
    {
        /* write it to the lineprinter */
        write( fdo, buf, nbytes );
    }

    /* close the net and lineprinter files */
    close( fdi );
    close( fdo );
}
```
7. Required Unix Modifications

Three mechanisms require attention before the major NSW components may be constructed: inter-process communication (IPC), expanded process control, and non-blocking I/O.

The following sections describe a set of proposed Unix modifications for the NSW. The new inter-process communication facility is the most important and complex recommendation.

7.1 Unix-IPC

Taken together, events and messages comprise a completely new IPC scheme. The delays induced in the transmission of messages and to some extent events can have a significant effect on any upper-level implementations that rely on IPC for data transfer. For this reason, speed and efficiency have been held in fairly high regard throughout the new IPC design.

Simplicity has also been a major factor in the IPC design. There are few rules governing the use of the new mechanism. It is left to the communicating processes to implement their own protocols for making decisions such as when to send data, how much, etc. If a particular process needs to know when another process has finished transmitting data, they should agree to send a mutually-acceptable event indicating that the process has finished transmitting. Other communicating processes should not be forced to implement an "official" protocol to be used when a series of transmissions is completed if they are not concerned with such an event.
Generic addressing has been recommended because of the amount of overhead that can be bypassed with an implied communication path. Various error responses are not required, and a significant amount of 'open-a-path' software has been reduced to a success or error indication on the sending of a message.

7.2 Addressing

Event and message destinations may be addressed with a generic or specific form.

The generic form may be used to specify a process or group of processes that have "identified" themselves with a particular name. For example, the Unix MSG service may want to be known as "UNIX-MSG". All processes with the same name will receive generically-addressed messages or events to that name.

A primitive, 'generic_name', is defined to allow a process to specify the name by which it wants to be known.

Calling Sequence and Return Values

-1; 0 = generic_name( "myname" );

  0 = success
  -1 = out of name storage
7.3 Events

Unix signals are unsuitable for the implementation of MSG alarms. Three event primitives are defined: sndevent, snd_g_event, and wait_event. Each of the primitives involves the concepts of an event source (the process that sends the event), an event destination (the process or group of processes that is to receive the event), an event type or opcode (an arbitrary 16-bit type), and an event data (an arbitrary 16-bit data field). The contents of event type and event data are left to user discretion.

Each process will own an event queue where events are buffered until called for.

7.3.1 Sndevent and Snd_g_event

Sndevent and snd_g_event request that an event type and data field be transmitted to the destination process. Snd_g_event is used to send events to a generic destination of Addressing. Both return one of three values indicating the success or failure of a request. These values and calling sequences are detailed below.
Calling Sequence and Return Values

```
1 0 -1 = sndevent( dest, type, data );
1 0 -1 = snd_g_event( "dest", type, data );
```

1 = destination event queue full
0 = event transmitted successfully
-1 = invalid destination

### 7.3.2 Wait_event

Wait_event is used to retrieve a queued event. If there are no enqueued events, the process may be de-scheduled until one is stored into its queue. Wait_event may be used to wait for an event of any type (parameter of zero), or to wait for an event of a specific type (parameter of the type to be waited for). The second parameter is the address of a memory location in which the source of the event is stored upon return. The third parameter is the address of a memory location that will contain the data word when control returns. The specific event type is passed to the user as the return value of the call.

**Calling Sequence and Return Value**

```
eventtype = wait_event( type, &src, &data );
```

= the actual event type returned.
7.4 Messages

The Unix pipe mechanism is unsuitable for a large class of inter-process data transmissions. Six new message primitives are defined: sndseg, snd_g_seg, rcvseg, getsba, getseg, and freeseg.

As previously stated, efficiency plays a major role in the ultimate usefulness of an inter-process mechanism. Other forms of inter-process data transmission have required that data be copied from the sending process address space to system buffers, and when requested, copied from system buffers to the receiving process. The envisioned message implementation will provide the framework for the asynchronous transmission of large messages without traditional system copying.

Messages are communicated between processes via segments. Segments have two basic properties, a segment base address (SBA), and a length of physical memory. At any time a given segment may have a length of physical memory associated with it. The amount and location of a piece of physical memory associated with a segment is stored in a segment descriptor. The segment base address is a user-level virtual address. Each SBA corresponds to a segment descriptor within the operating system. The physical memory associated with a segment descriptor is called segment memory. Messages are transmitted and received within segment memory.
Segment base addresses are obtained from the getsba primitive. Getsba returns the SBA of a segment with no associated segment memory. Segment memory is associated with or dissociated from a segment by executing the getseg and freeseg primitives. The segment length attribute reflects the amount of segment memory presently associated with a segment. The length may range from 0 to 8192 bytes. Segment memory is directly referenced through the SBA. A segment-based memory reference greater than the current length of segment memory will generate an illegal memory trap.

Segment descriptors are a thinly-disguised term for the relocation registers of the PDP-11 Memory Management Unit. These registers will be used to map the address space of a receiving process and the address space of a sending process into a common piece of physical memory allowing the direct manipulation of message data by both entities.

When the PDP-11 memory management unit is enabled, 16-bit program-level addresses, called virtual addresses, are used in conjunction with the map registers to construct an 18-bit direct physical memory address, thereby introducing a level of indirection between user-level memory references and the physical memory itself. Thus, the same section of physical memory may be mapped into the virtual address space of several processes by loading a map register of each process with the same physical memory relocation constant.
The PDP-11 memory management hardware has a limited number of user-level map registers: 8 for 11/40 processors and 16 for 11/45 and 11/70 processors. The number of map registers available for IPC will depend on the size of a program's data, stack, and code segments.

The average size of Unix programs is small, about 6k bytes. This leaves 6 map registers free for IPC use. It is felt that 3 unused map registers are adequate for most uses. This means that large programs such as the Unix document formatting program, NROFF (20k bytes, 4 free map registers), could make use of this mechanism.

7.4.1 Receiving a Message

A process must execute the getsba primitive to obtain a valid segment and associated segment base address. To receive a message, a process passes a segment base address to the rcvseg primitive. Rcvseg searches a process message queue for a message. If one cannot be found, the process may be de-scheduled until one is stored in the queue. Once a message has been found, the segment map register is loaded with the segment memory relocation constant. The segment is now associated with a particular piece of segment memory containing message data. Data can then be directly manipulated through the segment base address. When processing of the message data is complete, the
freeseg primitive is executed to dissociate the segment base address and segment memory.

7.4.2 Sending a Message

A program must execute the getsba primitive to obtain a valid segment base address. The getseg primitive may then be used to allocate a section of segment memory to a segment. A message can be constructed within the segment memory through the SBA. Segment memory is transmitted to another process with either the sndseg or snd_g_seg primitive. The snd_g_seg primitive is used to send segment memory to a generic destination cf. Addressing.

Segment memory may be dissociated with a segment by executing the freeseg primitive.

7.5 Message Primitive Definition

Each of the message primitives will be described in turn. Each description will contain a short statement of the primitive function, a description of its calling sequence, and a statement concerning the potential return values. Primitive descriptions are grouped by function.

7.5.1 GETSBA

Getsba searches the process space for an unused map register. Getsba returns a Segment Base Address. Through this SBA any associated segment memory may be directly accessed as if it were an array.
Calling Sequence and Return Values

\[ SBA \mid -1 = \text{getsba()}
\]

\[ SBA = \text{Segment Base Address} \]

\[ -1 = \text{no segments available} \]

7.5.2 RCVSEG

Rcvseg is used to associate segment memory containing a message with a segment base address. Once an association has taken place, segment memory may be accessed through the SBA. Rcvseg may block until a message is placed in a processes message queue. An SBA with no associated segment memory is passed as the first parameter. The address of a memory word containing either zero (will accept messages from any source) or a specific source is passed as the second parameter. When control is returned, src contains the specific message source.

Calling Sequence and Return Value

\[ \text{len} \mid -1 = \text{rcvseg(SBA,&src)}; \]

\[ \text{len} = \text{segment memory size.} \]

\[ -1 = \text{Invalid SBA.} \]
7.5.3 SNDSEG and SND_G_SEG

Sndseg and snd_g_seg are used to transmit segment memory to specific and generic destinations respectively. Generic destinations are described in Addressing.

**Calling Sequence and Return Values**

- \(-1 \mid 0 \mid 1\) = sndseg( cest, sba );
- \(-1 \mid 0 \mid 1\) = snd_g_seg( "dest", sba );

- \(-1\) = invalid segment base address
- \(0\) = message successfully queued
- \(1\) = invalid destination

7.5.4 GETSEG and FREESEG

Getseg and freeseg are used to associate and dissociate segment memory with a segment. Getseg allocates segment memory, stores the physical memory relocation constant in the segment descriptor, and initializes the number of processes accessing the segment memory.

Freeseg invalidates the map register relocation constant and decrements the number of processes accessing the segment memory. If the number of processes becomes zero, the memory is released to a segment free pool.
FREESEG -- Calling Sequence and Return Values

freeseg( sba )

defreeseg has no return values

GETSEG -- Calling Sequence and Return Values

-1 ; 0 ; 1 = getseg( sba, len );

-1 = invalid segment base address

0 = memory associated successfully

1 = no segment memory available.
8. Non-Blocking I/O

The NSW File Package and I/O Station require non-blocking I/O functions. That is, a process must be able to issue an I/O request and not have to wait for completion. Completion is signaled at a later time with the return of an event.

Standard Unix I/O functions include open, close, read, write. These functions block until a request completes. Non-blocking analogs of each of these functions can be constructed by adding an event parameter to each of the calling sequences. Each of the non-blocking analogs will initiate the associated I/O request and return immediately. The specified event will be sent to the initiating process when the request has completed.

Each standard peripheral device driver must be modified to recognize a non-blocking function request and return an event type at the appropriate point.

Given that an event mechanism exists, non-blocking device I/O is relatively simple to implement.

Each standard peripheral device driver now issues a "wakeup" call to notify a blocked process that I/O has completed. All that is needed to add a non-blocking capability is to check the type of I/O requested by the initiating process. If a blocking request was made, a wakeup is performed. If a non-blocking request was made, an event is transmitted to the initiating process.
The Foreman, the File Package, CLI-Front End and IOS require non-blocking I/O.

8.1 Process Control

The Unix process control facility is sufficient for the requirements of the NSW Foreman, CLI-Front End, and MSG. However, given that an IPC mechanism is required for an MSG implementation, the process control facility could be streamlined nicely by taking advantage of the new mechanism.

Standard Unix process control allows an immediate parent process to start, stop, and continue a child. When a process has been stopped, a parent may read and write a child's memory one word at a time.

Presently all of the parent process control requests are blocking. The IPC event and message mechanisms could be employed to allow a parent to simultaneously manage several childrens' address spaces.

The IPC message primitives will allow the manipulation of a child's memory on a more efficient block-at-a-time basis.