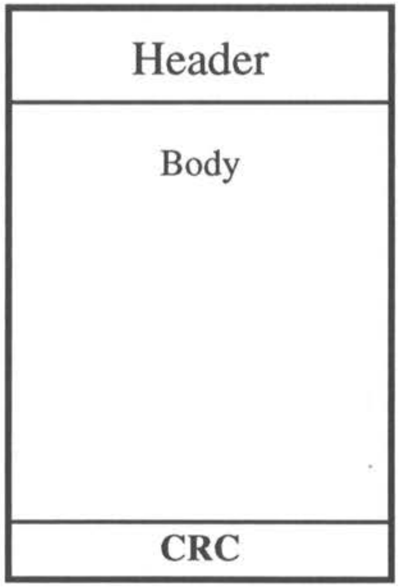
microware

Training and Education
Presents:

OS-9 System Overview

OS-9 Memory Modules

The OS-9 memory module is the basic unit of data within the system. All executables are placed within modules which the kernel can recognize by a unique name. The kernel can also identify any module based on its general type and purpose. All modules share a common architecture, as seen below.



Module Directory

All modules the kernel knows about are kept in the module directory on the system. This is just a table the kernel uses to keep track of all modules. When a module is needed and specified by name, the kernel searches this directory for the requested module. If it can not be found, the error E_MNF (module not found) is reported.

The module directory may be displayed by using the **mdir** utility. In the standard use, this program simply shows the names of all modules currently available in memory. The extended form of this command (mdir **-e**) shows much more information about each module, including its current address, size, type, language, attributes, revision, edition, and permissions.

```
$ mdir -e
   Addr     Size    Owner   Perm Type Revs Ed # Lnk  Module Name
________ ______  _____   _____ ____ ____ ____ ____ ____________
00009100  28292   0.0    0555 Sys  a000  244    2 kernel
000144d0    926   0.0    0555 Sys  8000   39    3 init
```

Items on the Mdir Line

The Lnk field specifies the link count of the module. This gives the system an idea as to how many times the module has been linked to. When you start the "dir" command, the dir module has its link count incremented by one. When the dir process completes, the link count of the dir module is decremented by one. Many modules will disappear when their link count reaches zero to make room for other modules on the system.

The Ed # field contains the edition number of the module. A module's edition number is a 16 bit value that means absolutely nothing to the system! This number allows for a convenient means of identifying one version of a module versus another. When a customer contacts Microware's technical support concerning a problem with a certain module, the support engineer can identify bug fixes that have occurred between the users edition and the current edition, possibly saving time in the debug process.

The field labeled Revs is actually two fields: an attributes section and a revision value. Each half of this field is eight bits, with attributes in the high-order section of the field. The revisions section is unstructured, but significant. When an explicit request to load the modules of a file is made, there is always the possibility that a module in memory shares the same name as one on disk. When this happens, the kernel compares the revision sections of each of the modules. If the module on disk has a smaller (or equal) value compared to the one in memory, the module in memory has its link count incremented. However, if the module on disk has a revision which exceeds that in memory, this module is considered to be more recent and replaces the older module in memory. (Processes currently using the older module continue to use it; any new requests for the module of that name are granted to the new module.)

The attributes section has three bits of significance currently, corresponding to the attributes of reentrancy, system state, and stickiness. Modules under OS-9 are generally reentrant, meaning they may be used simultaneously by several processes. To accomplish that, the module must not be self-modifying. (Microware's high level language compilers generate non-self modifying code.) If a module does not have the reentrant attribute, then when a link request is made of the module, that request will fail if the module already has a non-zero link count.

The sticky attribute determines what happens when the link count of the module reaches zero. If a module does not have the sticky attribute, it is generally removed from memory when the link count reaches zero. However, if the sticky bit is present, it will remain in memory with a zero link count.

Valid states are system and user state, corresponding to privileged versus normal cpu access. The state attribute defines what state a module executes in. System state is typically reserved for device drivers and other operating system extensions.

Type is another two item field; this one contains the modules type and language specifications.

The kernel uses the type section to determine which of the known module types it is looking at. These types include program modules (Prog), device drivers (Driv), file managers (FMan), system modules (Sys) among others.

The language field is mainly under-utilized. Currently, the only language used is "68k object." Since not all modules are executable code, there is also a wildcard language representing a don't care state.

The next field to the left is the module permissions. This field contains three sections: one for the owner of the module, one for members of the same group as the module owner, and one for all other users. Each section has three significant bits which are used to determine read, write, and execute access for the module.

The owner field simply identifies who owns the particular module.

The size field reports the length of the module, in bytes.

Finally, the address field tells the address of the first byte of the module in its current location. This address may change from one loading of the module to the next. (This is why OS-9 executables must be position independent.)

Modules vs. Files

Although it is a common mistake, the term module cannot be used interchangeably with the term file. A file is something that lives on disk, and may hold a module. A file also may hold more than one module, and can be completely devoid of any modules. When the "dir" command is typed into the shell, the system must be able to find the module called dir in memory to complete the request. If that module cannot be found, the system will search the disk for a file called dir. (The system will search the users current execution directory first, and then along the contents of the user's PATH environment variable.) If the module is not found but a file whose name is identical to the command is, the module(s) within that file are loaded into memory, and the system will execute the first module found in the file. This means that the name of the file to be executed does not have to be the same as the name of the file itself.

Resident Module Oriented Commands

mdir - display a list of all modules in memory

ident - display identifying information about certain modules

load - place modules from a file into memory

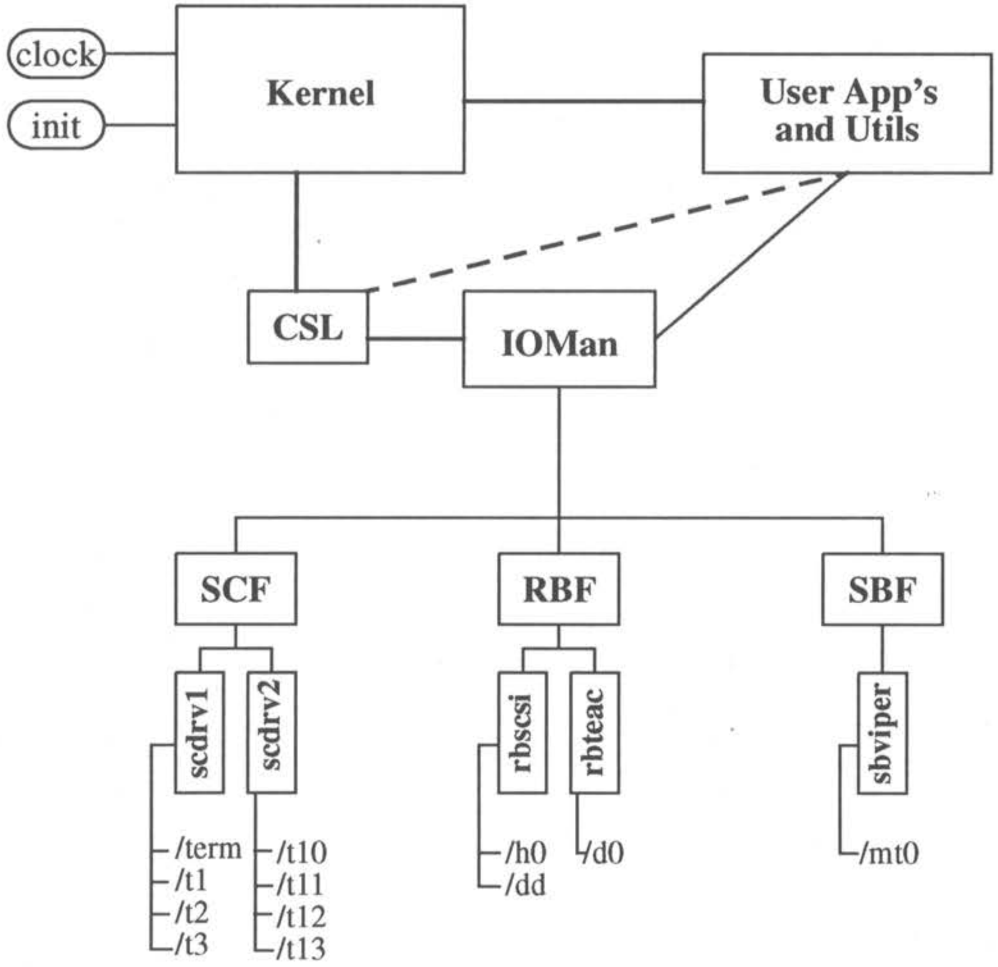
unlink - decrement the link count of a module, possibly removing it from memory

save - copy the contents of a module in memory to disk

fixmod - change certain pieces of information from the header of a module

moded - edit the body of certain types of modules (This command is not currently supported under OS-9000.)

OS-9 I/O Sub-System Structure



OS-9 Devices

A device under OS-9 is some entity, usually hardware, which requires some system level control. To satisfy this requirement, all devices under OS-9 should have a file manager, device driver, and device descriptor associated with them. For example, the ram disk on this system requires the **RBF** file manager, the **ram** device driver, and the **r0** device descriptor.

The file manager is sort of a generic device driver. These code modules are the interface between **ioman** and the device's driver. The file manager knows the generalities of a specific class of device, such as a sequential character device, or a random block device. They do not know how to control the actual hardware, but rather know how to perform some of the tasks that all drivers need to do and thus provide for more simplified device drivers. (The device driver writer can concentrate on the hardware and the interface to the file manager, without concern for higher level system activities.)

The device descriptor does not contain any executable code, but rather contains data (in a pre-defined format) describing the actual device of interest. Items in this description include the names of the file manager and device driver for the hardware, as well as the physical address for the device and customization details for the file manager and driver.

Using OS-9 Devices

Before a device may be used, it must be recognized by the system. This just entails initializing the device, which can be done with the **iniz** command. For example,

```
$ iniz r0
```

is a request to initialize the device represented by a descriptor named r0. (The **devs** command shows all devices that are currently initialized.) Once the device is initialized, it is accessed based on its name. For example, to create a file called *test* in the root directory of the r0 device, you would create the file named */r0/test*.

New devices may be added to OS-9 while the system is running, and devices may also be removed while the system is running. To add a new device, the descriptor, driver, and file manager must all be in memory. Once that is the case, simply using the device adds it to the system. (A device that is not initialized prior to use will become initialized, but it will become non-initialized when use of the device ends. This can lead to fragmented memory.)

Process Creation

View the output of the **procs -e** command and you will see many pieces of information concerning each of the processes currently on your OS-9 system. Each process on the system has a block of data called a process descriptor which is used to maintain the process throughout its lifetime. The information displayed by procs comes from the process descriptor.

In order to place a new process on the system, a process descriptor for the process must be created. The system has tools in place to make this task easy. Through Ultra-C, the *_os_exec()* function is used.

The **vos** utility has a command hook into the *_os_exec()* call through the command '_os_exec'; it may be used to create "child1", "child2", up to "child9". It also supports the command 'family' which shows parent and child information as related to the process under inspection. The "gen X" command, where X is an optional process id number, shows general information from the process' descriptor.

Using _os_exec()

This call takes the parameters needed to create the child process. Those parameters include a pointer to a function, the priority of the process, any additional stack space needed, the name of the module to execute, the process' environment and argument list, and the number of open paths to pass to the process. The basic syntax of the call is:

```
#include <cglob.h> /* to get the _environ variable */
error_code _os_exec(_os_fork, u_int32 priority,
  u_int32 pathcnt, (void *)argv[0], char **argv,
  char **envp, u_int32 stacksize, process_id *id, 0, 0);
```

As it happens, *_os_exec()* does not perform all of the work in creating the process descriptor. This is because there are a few ways the task can be performed. Shown above is the standard way, allowing one process to create another such that each can run in parallel. The first parameter, *_os_fork*, is the name of the function that completes the task for us. The next page shows a simple program which creates a child and waits for it to terminate.

fork.c

```
#include <stdio.h>      /* for printf */
#include <types.h>      /* good to have */
#include <process.h>    /* for _os_exec */
#include <errno.h>      /* for errno*/
#include <cglob.h>      /* for _environ */

char *argblk[] = {“dir”,”-e”,0};

main()
{
   process_id  id;
   status_code status;

   /* create the process */
   errno = _os_exec(_os_fork,0,3,(void*)argblk[0],argblk,
                   _environ,0,&id,0,0);

   /* did it work? */
   if (errno != 0) _os_exit(errno);
```

```
    printf("Process has been created with id %d\n",id);

    /* wait for the process to terminate */
    errno = _os_wait(&id, &status);

    /* any problems waiting? */
    if (errno != 0) _os_exit(errno);

    /* no. report child exit code and exit */
    printf("Child exited with code %d\n",status);
    _os_exit(0);
}
```

This program can be compiled with **cc fork.c**.

Paths

A path is used to perform device I/O on the OS-9 system. All paths are represented (to user state processes) by a number between 0 and 31 inclusive. This number is a table entry describing a system path number. This system path number describes a data structure called a path descriptor which is used by the kernel to maintain the I/O channel.

When most programs are started, they already have three open paths: standard input (0), standard output (1), and standard error (2). This being the case, the next paths opened will be numbers 3, 4, etc. The path number assigned to the creating process is always the lowest number which is not being used. Thus, if you were to close path 1 and open a new path, it would take position 1, the standard output path.

At the lowest level, paths are created with the C *_os_create()* call (or alternatively, the *_os_open()* call) and destroyed with the *_os_close()* call. They are read from and written to with the *_os_read()* and *_os_write()* calls respectively. Also, a path may be duplicated on another number with the *_os_dup()* call.

‘C’ Path Calls

All of these calls should have access to the `<modes.h>` and `<types.h>` header files. They also all return a variable of type `error_code`.

```
_os_create(char *name, u_int32 mode, path_id *path, u_int32 perm, ...)
Special modes: FAM_SIZE, FAM_NOCREATE;
Note: On OS-9, the _os9_create() function take identical parameters,
and is slightly faster.

_os_open(char *name, u_int32 mode, path_id *path);

_os_dup(path_id path, path_id *new_path);

_os_read(path_id path, void *buffer, u_int32 *count);

_os_write(path_id path, void *buffer, u_int32 *count);

_os_close(path_id path);
```

Modification of Child I/O

In order to modify the standard paths of a child process, careful planning and work must occur before the child is spawned. Assuming the pathcnt parameter to _os_exec() is three, the child will inherit three open paths from the parent, the three standard paths. If the parent is interested in modifying the standard paths received by the child, it must modify them for itself before creating the child. This perhaps sounds destructive to the parent, but with careful use of the _os_dup() call, the parent can save its state before making modifications and restore it afterwards. For example, pseudo-code for spawning a child with redirected standard output is as follows:

```
/* pseduo-code! */
sv_out = dup(1);     /* copy my standard output path */
close(1);            /* and destroy the real stdout */
open("new_output"); /* open childs output into my stdout path */
spawn_child();       /* kick child off running */
close(1);            /* I don't need this anymore */
dup(sv_out);         /* restore my stdout */
close(sv_out);       /* and get rid of saved path */
```

Training and Education
Presents:

OS-9 Inter-Process Communications

Inter-Process Communication Overview

OS-9 has six built-in methods of performing inter-process communications.

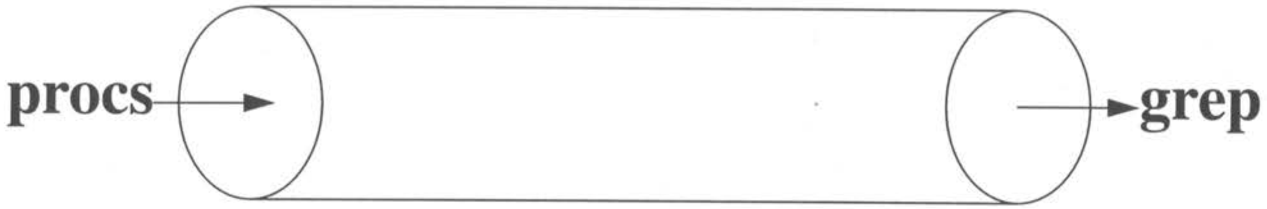
Pipes (data transfer):	sequential message passing (data queue)
Data Modules (data transfer):	shared memory implementation
Signals (synchronization):	software generated interrupts
Alarms (synchronization):	pre-arranged signals
Semaphores (synchronization):	logical binary flags (**V3.0 and later**)
Events (synchronization):	32 bit flags

Each of these methods of accomplishing IPC has its own advantages over others, but often, more than one form of IPC can be used to accomplish the desired effect.

Pipes

Pipes provide a method of placing data inside a first-in, first-out queue that is destroyed as soon as it is removed (read). There are two types of pipes supported under OS-9: **named** and **unnamed**. The pipe created when one process is "piped" into another is unnamed; it may only be accessed by the process that created it and processes that are started up with it being in an open path. Named pipes, on the other hand, may be accessed by any process that knows the name (and has permission based on the user and group number of the process.) When the command "a ! b" is entered, the standard output of process "a" is sent into a pipe, and the standard input of the "b" process is fed from the same pipe.

$ procs -e ! grep shell



When opening any path on the system there must be a name for that path. The name for a named pipe has the format of “/pipe/**name**”, where **name** is whatever name the pipe should have. (Although the pipe device appears as a directory type device, and in fact you can type the command “dir /pipe” to see the names of named pipes on the system, the system’s disk devices, if even present, are not used for pipes. The pipe directory is flat, that is, you can not have subdirectories within the pipe directory.) Even opening a path to an unnamed pipe requires some sort of name; the name for all unnamed pipes is “/pipe”. Every request to “/pipe” that succeeds is given a brand new unnamed pipe. (There may be a virtual unlimited number of unnamed pipes open at one time; all of them have the name /pipe.)

Pipe Activity

Table 5: What if you try to...

Action	Named Pipe	Unnamed Pipe
write to a pipe that is full?	Process trying to write to a full named pipe enter the I/O queue. (No error condition).	When all processes that have read access to a full unnamed pipe are in the I/O queue (trying to write), the first writer will receive an E_WRITE error.
read from a pipe that is empty?	When all processes with write access to an empty pipe are in the I/O queue (trying to read from it), the first reader will receive an E_READ error.	
close a pipe that is not empty?	A named pipe is deallocated when all processes close the pipe and no data remains.	Unnamed pipes are deallocated when there are no more processes with the pipe open, regardless of the amount of data remaining.
create a pipe whose name is taken?	Just as creating a file whose name is taken; the pipe is either truncated or the call fails, depending on the presence of FAM_NOCREATE.	Non-applicable.
open a pipe that doesn't exist?	Just as trying to open a non-existing file, the process will receive an E_PNNF error. (Path Name Not Found)	Every open to "/pipe" is the same as creating a new unnamed pipe.

“C” Calls for Pipes

These are some calls that, though not written specifically for use with pipes, lend themselves greatly towards dealing with pipes. These calls interface (through the kernel and ioman) to the pipe file manager to get and change information about a pipe. To use them, your program should include the `<types.h>` and `<sg_codes.h>` header files. They all return an error_code.

```
_os_gs_ready(path_id path, u_int32 *incount);
This call returns (in incount) the number of characters available.

_os_gs_eof(path_id path);
This call returns E_EOF if a read on this path would return an error
for some process.

_os_gs_size(path_id path, u_int32 *size);
Returns the maximum number of bytes the pipe can hold before blocking.

_os_ss_sendsig(path_id path, signal_code signal);
Installs a pending signal of the specified path, to be sent as soon as
data is available for reading.

_os_ss_relea(path_id path);
Remove a pending signal installed with _os_ss_sendsig().
```

Data Modules

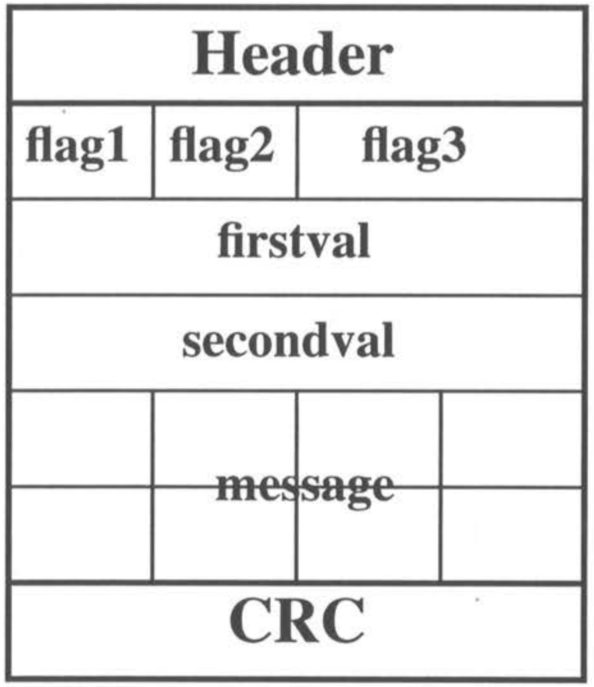
Data modules are memory modules that hold information rather than executable code.These modules are a method of sharing memory across multiple processes and they also provide a simple method for saving the state of a program for future recall.

Assume you would like two processes to share some data. You would place these shared "global" variables in a structure which resides within a data module. For example,

```
typedef struct {
  char flag1, flag2;
  short flag3;
  int firstval, secondval;
  char message[8];
} datastruct;
```

can easily be placed within a data module. The map of the module with this structure appears on the next page.

Data Module Map



Header

flag1 flag2 flag3

firstval

secondval

message

CRC

“C” Code for Data Modules

Before a data module may be used, it must exist in memory! If the module is already in memory, it can be linked to with the **_os_link**() function call. If it is not in memory but is inside of a file on disk, the contents of the file may be loaded into memory with either the **_os_load**() or **_os_loadp**() functions. If the module is to be created in memory, **_os_datmod**() will be necessary. All of these functions need access to the **<module.h>** header file.

```
_os_link(char **modname, mh_com **modptr, void **dataptr, u_int16
        *typlang, u_int16 *attrev);

_os_load(char *filename, mh_com **modptr, void **dataptr, u_int32
        mode, u_int16 *typlang, u_int16 *attrev, u_int32 color);

_os_loadp(char *filename, u_int32 mode, char *nameptr, mh_com
        **modptr);

_os_datmod(char *modname, u_int32 size, u_int16 *attrev, u_int16
        *typlang, u_int32 perm, void **dataptr, mh_data **modptr);
```

Two of the parameters for the above functions are pointers: one to point to the module header itself, and the other to point to the data area (body) of the module. The module pointer is of a standard variable type, but since the compiler writers could not anticipate the type of data you will place in your data module, the official type for the data pointer is void.

When an application completes using a data module, it has the responsibility of informing the kernel so the link count may be decremented. This action is performed through the **_os_unlink**() call. The function **_os_unload**() will also decrement the link count on a module.

```
_os_unlink(mh_com *modptr);

_os_unload(char *modname, u_int32 typlang);
```

u_int16?

The program on the next page describes the typical sequence of calls used when preparing to use data modules.

datmod.c

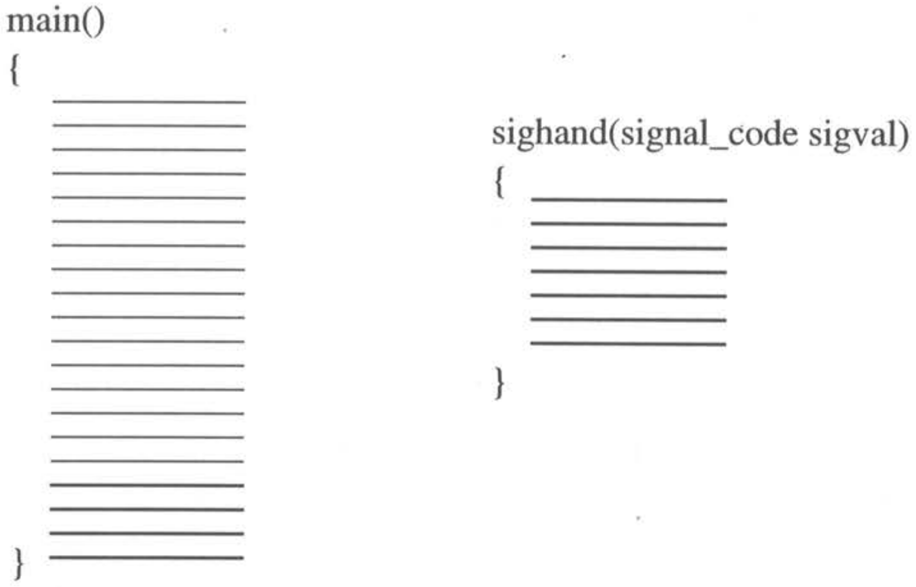
```
#include <stdio.h>
#include <module.h>
#include <modes.h>
#include <errno.h>

typedef struct {
   char flag1, flag2;
   u_int16 flag3;
   u_int32 firstval, secondval;
   char message[8];
} datastruct;

main()
{
   char *dm_name = "dm_name";  /* name of data module */
   mh_data *modptr;        /* point to the module header */
   datastruct *dataptr;    /* point to the data area */
   u_int16 attrev;         /* attribute/revision of module */
   u_int16 typlang;        /* type/language of module */
   u_int32 perm;           /* module permissions */
```

#include<types.h>

```
   /* prepare parameters */
   attrev = mkattrevs(MA_REENT, 0);
   typlang = mktypelang(MT_DATA, ML_ANY);
   perm = MP_OWNER_READ|MP_OWNER_WRITE|MP_GROUP_READ|MP_GROUP_WRITE;

   /* first try to create the module */
   errno = _os_datmod(dm_name,sizeof(datastruct),&attrev,
      &typlang, perm, (void**)&dataptr, &modptr);

   if (errno) {
      /* then try to link to it */
      errno = _os_link(&dm_name, (mh_com**)&modptr, (void**)&dataptr,
         &typlang, &attrev);

      if (errno) {
         fprintf(stderr,"Couldn't link or create! Error #%d\n", errno);
         _os_exit(errno);
      }
      fprintf(stderr,"Link was successful!\n");
   } else {
      fprintf(stderr,"Create was successful!\n");
   }

   /* use module to your heart's content */

   fprintf(stderr,"Exiting!\n");
   _os_exit(_os_unlink((mh_com*)modptr));
}
```

Signals

Signals are a form of software generated interrupts. When a signal is received by a process, that process puts what it is doing on temporary hold, executes a special signal handling routine, and then resumes to where it left off. (Sort of like a forced asynchronous subroutine call.)



```
main()
{

}

sighand(signal_code sigval)
{

}
```

To prepare to handle signals, the process must tell the kernel what routine will be the signal handler. If a process receives a signal and has not first informed the kernel about this routine, that process will be terminated. This handling routine will have access to the 16 bit signal_code that is sent to the process.

Table 6: Signal Definitions/Ranges

Signal	Description
0	Unconditional kill. The signal handler never sees this signal. **OS-9 only!**
1	Wake up signal. The signal handler never sees this either.
2	SIGQUIT (Ctrl-E)
3	SIGINT (Ctrl-C)
4	Modem hang up signal. **(OS-9000: Unconditional kill)**
5	**(OS-9000: Modem hang up signal)**
2-31	Deadly I/O signals. **(OS-9000: 2-5)**
0-255	Definition reserved by Microware.
256-65535	User defined signals.

The system is informed of a process' ability to handle signals via the **_os_intercept**() call. For example:

```
#include <signal.h>
#include <types.h>
#include <errno.h>
#include <cglob.h>              /* get access to _glob_data */

void sighand(signal_code sigval);  /* the signal handler */

main()
{
  errno = _os_intercept((void(*)())sighand, _glob_data);
```

The external variable `_glob_data` is a global that Ultra-C makes available to processes, which points to their static storage areas.

There is no typical signal handling routine, though most look at the signal that was sent to the process and take some action based on that signal. What is important, however, is that the signal handling routine use **_os_rte**() to exit the function at every possible exit. Not exiting in this manner will cause unpredictable (and usually bad) results to occur.

Actually, a signal handler may return to the main program through a long-jump. However, returning in this manner will leave the signal context on the stack; the **_os_sigreset**() function should be used to clear it. Also, signals will remain masked, but can be unmasked via **_os_sigmask**().

Signals are sent to a process through the function **_os_send**(), which takes a process id and signal as parameters. There are two idiosyncracies with this call. If the signal is 0, the unconditional kill signal, it will only be sent if the sending process and the receiving process are owned by the same user and group numbers or the sending process is a group zero process. If the process id is 0, the signal is sent to all other processes on the system owned by the same group and user as the sending process. (This is called broadcast signaling.)

Since there are times when it would be inconvenient to receive signals, each process has a signal masking level. This level is an unsigned value which, if non-zero, forces incoming signals to queue up waiting to be handled. If the level is zero, signals will be received by the signal handler. (The signal handler should increase the masking level upon starting and decrease it upon leaving.) The function **_os_sigmask**() takes as valid parameters 1, 0, and -1 to increase, set to zero, and decrease the masking level respectively. The signal masking level is an eight bit quantity; the system takes steps to insure that it does not wrap around in either direction. Note, that since signal values 0 and 1 do not queue up to be serviced, the signal mask has no effect on them. Also, a signal mask does not prevent the signal from waking a non-active process; it merely delays the calling of the signal handling routine.

Alarms

OS-9 alarms provide a method of sending yourself a signal that won't be received until some time in the future. Alarms may be installed for a specific time/date, at a specific time difference from "now", or repetitively every *n* clock ticks or seconds. With an alarm, you are requesting that some pre-defined signal be sent to your process later. The main C calls for signals are shown below. They all need access to the `<alarms.h>` header file and all return an `error_code`.

_os_alarm_set (alarm_id *alrm_id, signal_code sigval, u_int32 time): request *signal* to be sent when *time* clock ticks or, if appropriate, seconds have elapsed.

_os_alarm_cycle (): the same parameters as _os_alarm_set(), but the alarm is sent every *time* clock ticks / seconds. This function is often used to incorporate a watchdog timer.

_os_alarm_atdate (alarm_id *alrm_id, signal_code signal, u_int32 time, u_int32 date): send *signal* at the requested *time* and (if non-zero) *date*.

_os_alarm_delete (alarm_id alrm_id): remove the specified alarm before it is sent.

Semaphores

Semaphores are flags that may either be up or down; true or false; 1 or 0, etc. (I prefer to think of the semaphore states as free or used.) They are a basic tool used to synchronize shared resources that may only be used by one process at a time. When a process needs access to a resource, it will wait for the semaphore to be free, and once the semaphore is free, the waiting process returns from waiting and the semaphore is then marked as used. Thus, only one process at a time may own the semaphore. When the process that does own it completes using the resource, it must return the semaphore (mark it as free) so the next waiting process may take it.

Semaphores on OS-9 use very little overhead on the part of the system. So little, in fact, that the process that creates the semaphore must allocate memory for it! Since semaphores are generally used by multiple processes, the general place to place a semaphore is inside a data module. Doing this allows other processes to easily find the semaphore based on a pre-defined data module name and an offset into the data module for where the semaphore begins. To access a semaphore, a process must know the physical address of the space allocated for it; if a process knows the proper offset into a data module, that value can be added to the address of the module header, which is returned from _os_link().

“C” Semaphore Calls

All of these calls require to `<semaphore.h>` header file.

Every process using a semaphore must make a call to initialize it, **_os_sema_init**(). This function is passed the semaphore pointer which points to the semaphore of interest.

To compete for a semaphore, a process calls **_os_sema_p**() with the semaphore pointer as a parameter. This function will cause the calling process to wait until the semaphore is free before continuing.

When a process wishes to release the semaphore, it calls **_os_sema_v**() with its pointer. Note, that a process is not required to own the semaphore (through a previous **p** call) to release (**v**) it. This can provide some interesting, though esoteric situations.

When a process is through using a semaphore, it should call **_os_sema_term**() with the semaphore pointer.

Events

Events are a step higher than semaphores. When waiting for a semaphore, a process is waiting for one of two possible states to occur. Events are represented by a signed 32 bit value, so there are many more than two possible states! When waiting, a process chooses a range within the 32 bit realm which will satisfy the waiting process. The resources controlled by events age generally much more complex than those controlled by semaphores.

Before an event can be used, it must be created. The system will allocate the space for the event, but a process must make a call to **_os_ev_creat** (int32 winc, int32 sinc, u_int32 perm, event_id *ev_id, char *name, int32 value, u_int32 color) which builds an event with the given *name*, and assigns it an initial *value*. (OS-9 ignores the *perm* and *color* entries; these are used only by OS-9000.)

Basically, processes either wait for an event to occur or signal the fact that an event has occurred. While a process waits for an event to occur, it is waiting for the event value to come inside some wanted range. Once this has occurred, the waiting process will wake up and the event wait increment (*winc*) will be added to the event, allowing the value to immediately change. When a process signals the fact that an event has occurred, it causes the event signal increment (*sinc*) to be added to the event value. Each of these increments are signed values, thus the event value may rise or fall.

“C” Event Calls

Make sure all of these calls get the `<events.h>` and `<types.h>` header files included with them. They all return an `error_code`.

_os_ev_link (char *name, event_id *ev_id): link to the event based on its name. This is not necessary if you have done an _os_ev_creat() successfully.

_os9_ev_wait (event_id ev_id, int32 *value, int32 minval, int32 maxval): wait for the value of the event to be within the range defined by minval and maxval. The integer pointed to by the second parameter returns with the value of the event causing the process to wake.

_os_ev_signal (event_id ev_id, int32 *value, u_int32 actv_flag): cause the signal increment to be added to the event flag and then, if appropriate, wake up the first waiting process for the new value and if so, apply wait increment. Then, if actv_flag is non-zero, look for more processes to wake.

_os_ev_unlink (event_id): decrement the link count on the event. Use this call when you are through with the event.

_os9_ev_waitr(): the same parameters as _os9_ev_wait(), but the min and max values treat the current event value relative to zero. The returned integer is massaged to be relative to zero as well.

_os_ev_set (event_id, int32 *value, u_int32 actv_flag): change the event value to a specific value and wake up process(es) according to the same rules as for _os_ev_signal().

_os_ev_setr(): same parameters as for _os_ev_set(), but the value is added to the event value. Under OS-9, the value pointer will come back pointing to the value of the event before the call; under OS-9000 it will come back with the value of the event after the call.

_os_ev_read (event_id ev_id, int32 *value): determine the event value.

_os_ev_pulse (event_id ev_id, int32 *value, u_int32 actv_flag): similar to _os_ev_set(), but when all processes to wake up are awakened, this call restores the event value to its value prior to the pulse request.

_os_ev_delete (char *name): remove the event from the system event table if the event's link count is zero.

"C" Event Calls for Non-68K Versions of OS-9

_os_ev_setand(event_id ev_id, int32 *value, u_int32 mask, u_int32 actv_flag): logically ANDs the mask with the event value.

_os_ev_setor (): logically ORs the mask with the event value.

_os_ev_setxor(): logically XORs the mask with the event value.

_os_ev_tstset (event_id ev_id, int32 *value, signal_code *signal, u_int32 mask): logically ANDs the mask with the event value and blocks the calling process until all bits specified in the mask are cleared.

_os_ev_wait(event_id, ev_id, int32 *value, signal_code *signal, int32 min_val, int32 max_val): same as _os9_ev_wait(), except if awakened by a signal, the signal code is returned.

_os_ev_waitr(): the same parameters as _os_ev_wait(), but the min and max values treat the current event value relative to zero. The returned integer is massaged to be relative to zero as well.

Training and microware Education

I/O System Overview

MODULES

OS-9 manages both the physical assignment of memory to programs and the logical contents of memory by using ***memory modules***. A memory module is a logical, self-contained program, program segment, or collection of data.

Position Independent

OS-9 supports ten pre-defined types of modules and allows users to define their own module types. Each module type has a different function. Modules do not have to be complete programs. Modules simply have to be ***re-entrant, position-independent***, and conform to the basic module structure described in the next section.

Re-Entrant

The 68000 instruction set supports a programming style called ***re-entrant*** code. Re-entrant code is code that does not modify itself. This allows two or more different processes to share one copy of a module simultaneously. The processes do not affect each other, provided that each process has an independent area for its variables. Almost all OS-9 family software is re-entrant, and therefore uses memory efficiently.

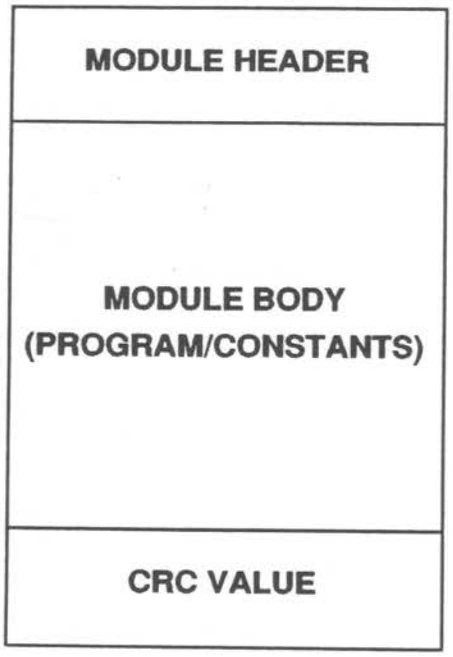
NOTE: Data modules are an exception to the no modification restriction. However, careful coördination is required for several processes to update a shared data module simultaneously.

A ***position-independent*** module does not know or care where it is loaded in memory. This allows OS-9 to load the program wherever memory space is available. In many operating systems, the user must specify a ***load address*** to place the program in memory. OS-9 determines an appropriate load address only when the program is run.

OS-9 compilers and interpreters automatically generate position-independent code. In assembly language programming, however, you must ensure position-independence by avoiding absolute address modes.

Basic Module Structure

Each module has three parts: a *module header*, a *module body*, and a *CRC (Cyclic Redundancy Check) value*.



The module header contains information (such as the module's name, size, type, language, etc.) that describes the module and its use. The linker creates the header at link-time.

The module body contains initialization data, program instructions, constant tables, etc.

The last three bytes of the module hold a CRC (Cyclic Redundancy Check) value used to verify the module's integrity. The linker creates the CRC at link-time.

Basic Memory Module Format

The CRC Value

The CRC is an error checking method used frequently in data communications and storage systems. The CRC is also a vital part of the ROM memory module search technique. It provides an extremely high degree of confidence that programs in memory are intact before execution and is an effective backup for the error detection systems of disk drives, memory systems, etc.

In OS-9, a 24-bit CRC value is computed over the entire module starting at the first byte of the module header and ending at the byte just before the CRC. OS-9 family compilers and assemblers automatically generate the module header and CRC values. If required, a user program can use the F$CRC system call to compute a CRC value over any specified data-bytes.

OS-9 cannot recognize a module with an incorrect CRC value. For this reason, you must update the CRC value of any module "patched" or modified in any way, or the module cannot be loaded from disk or found in ROM. Use the OS-9 fixmod utility to update the CRC's of patched modules.

ROMed Memory Modules

When OS-9 starts after a system reset, the kernel searches for modules in ROM. The kernel detects modules by looking for the module header sync code ($4AFC). When this byte pattern is detected, the header parity is checked to verify a correct header. If this test succeeds, the module size is obtained from the header and a 24-bit CRC is computed over the entire module. If the computed CRC is valid, the module is entered into the module directory.

OS-9 links to all of its component modules that were found during the search. All ROMed modules present in the system at startup are automatically included in the system module directory. This allows partially or completely ROM-based systems to be created. Any non-system module found in ROM is also included. This allows user-supplied software to be located during the start-up process and entered into the module directory.

Module Header Information

The following are the standard fields in the module header..

Offset	Name	Usage
$00	M$ID	Sync Bytes ($4AFC)
$02	M$SysRev	Revision ID
$04	M$Size	Module Size
$08	M$Owner	Owner ID
$0C	M$Name	Module Name Offset*
$10	M$Accs	Access Permissions
$12	M$Type	Module Type
$13	M$Lang	Module Language
$14	M$Attr	Attributes
$15	M$Revs	Revision Level
$16	M$Edit	Edit Edition
$18	M$Usage	Usage Comments Offset*
$1C	M$Symbol	Symbol Table
$20		Reserved
$2E	M$Parity	Header Parity Check
$30-up		Module Type Dependent
		Module Body
		CRC Check

* These fields are offset to strings

CRC Example Code

```
/***********************************************************************
 * Program:  compcrc.c                                                 *
 *                                                                     *
 * Purpose:  This program demonstrates how to compute the CRC value    *
 *           for an OS-9 module.                                       *
 *                                                                     *
 *  Update:  10/06/89 RRR    Created                                   *
 *           12/15/90 mja    Slightly Modified                         *
 *           01/03/91 mja    Modified Again                            *
 *                                                                     *
 *   Usage:  compcrc                                                   *
 *                                                                     *
 *    Note:  Compile with -DLIBCRC to use the library CRC function.    *
 ***********************************************************************/
@_sysedit:equ 10
#include <errno.h>
#include <module.h>
#define ERROR (-1)
main()
{   extern char *modlink();
    char  *p_mod;
    char mod_name[20];
    long  mod_crc,comp_crc;

    /* -- Get a pointer to the module name. */
    printf("Please input module name: ");
    gets(mod_name);

    /* -- Try to link to the named module. */
    if((p_mod=modlink(mod_name,mktypelang(MT_ANY,ML_ANY)))==(char*)(ERROR))
     exit (_errmsg (errno, "Unable to link to module [%s].\n", mod_name));

    /* -- Get the resident and computed module CRCs. */
    get_crcs(p_mod, &mod_crc, &comp_crc);

    /* -- Report the results. */
    printf ("The CRC stored in [%s] is %08x.\n", mod_name, mod_crc);
    printf ("The CRC computed for [%s] is %08x.\n", mod_name, comp_crc);

    /* -- Unlink the module and exit. */
    if (munlink (p_mod) == ERROR)
        exit (_errmsg (errno, "Error unlinking module.\n"));
}


get_crcs (modptr, modcrc, newcrc)
char *modptr;
long     *modcrc, *newcrc;
{   long accum;
    long mod_size;
```

```
        /* -- Get module size and initialize the CRC accumulator. */
    mod_size = ((mod_exec *)modptr)->_mh._msize;
    accum = -1;

    /* -- Compute the new CRC for the module. */
    calc_crc (modptr, mod_size - 4, &accum);
#ifdef LIBCRC
    crc (NULL, 0, &accum);
#else
    calc_crc ("", 1, &accum);
#endif
    accum = ~accum & 0xFFFFFF;
    *newcrc = accum;

    /* -- Get the old CRC from the module header. */
    *modcrc = *((long *)((char *)modptr + mod_size - 4)) & 0xffffff;
}


calc_crc(ptr,n,crcacc)
unsigned short *ptr;
long n;
unsigned long *crcacc;
{   register unsigned long tmp,tmp1,accum = *crcacc & 0x00ffffff;
    register short tmp2;

    if(n & 1) {
         /* special case for one zero byte*/
        if(n == 1 && *(char*)ptr == '\0') {
            tmp1 = tmp = (accum >> 16) & 0x0000ffff;
            accum <<= 8;
            tmp <<= 1;
            accum ^= tmp;
            tmp1 ^= tmp;
            tmp <<= 5;
            accum ^= tmp;
            tmp2 = tmp1;
            tmp2 <<= 2;
            tmp1 ^= tmp2;
            tmp2 = tmp1;
            tmp1 <<= 4;
            tmp2 ^= tmp1;
            if(tmp2 & 0x80) accum ^= 0x800021;
            accum &= 0x00ffffff;
            *crcacc = accum;
            return(0);
        } else  exit (_errmsg (1, "odd count for crc"));
            } else n >>= 1;             /* convert byte to word count */
    while(n-- > 0) {
#ifdef VAX
        tmp = *(unsigned char*)ptr;
        tmp = (tmp << 8) | *(((unsigned char*)ptr)+1);
        ++ptr;
#else
        tmp = *ptr++;               /* get (next) data word */
#endif
        tmp <<= 8;                  /* align data bit zero with CRC bit 0 */
        tmp ^= accum;               /* get new data-CRC difference */
        tmp &= 0xffffff00;          /* clear extraneous bits */
        accum <<= 16;               /* shift current CRC; strip all but 23:16 */
        tmp >>= 2;                  /* shift */
        accum ^= tmp;               /* add input bit 17 net effect (over 16 shifts)*/
        tmp >>= 5;
        accum ^= tmp;               /* add input bit 22 net effect */

        tmp2 = tmp;                 /* determine if even or odd number of bits */
        tmp >>= 1; tmp2 ^= tmp; tmp = tmp2;
        tmp <<= 2; tmp2 ^= tmp; tmp = tmp2;
        tmp <<= 4; tmp2 ^= tmp; tmp = tmp2;
        tmp <<= 8; tmp2 ^= tmp;
        if(tmp2 & 0x8000) accum ^= 0x800021;
        accum &= 0x00ffffff;        /* clear extraneous bits */
    }
    *crcacc = accum;
}
```

Levels of Modularity

OS-9 is a highly modular operating system. It is designed so that each module provides specific functions. OS-9's modularity allows individual modules to be included or deleted in the system when OS-9 is configured for a specific computer.

OS-9 has four levels of modularity:

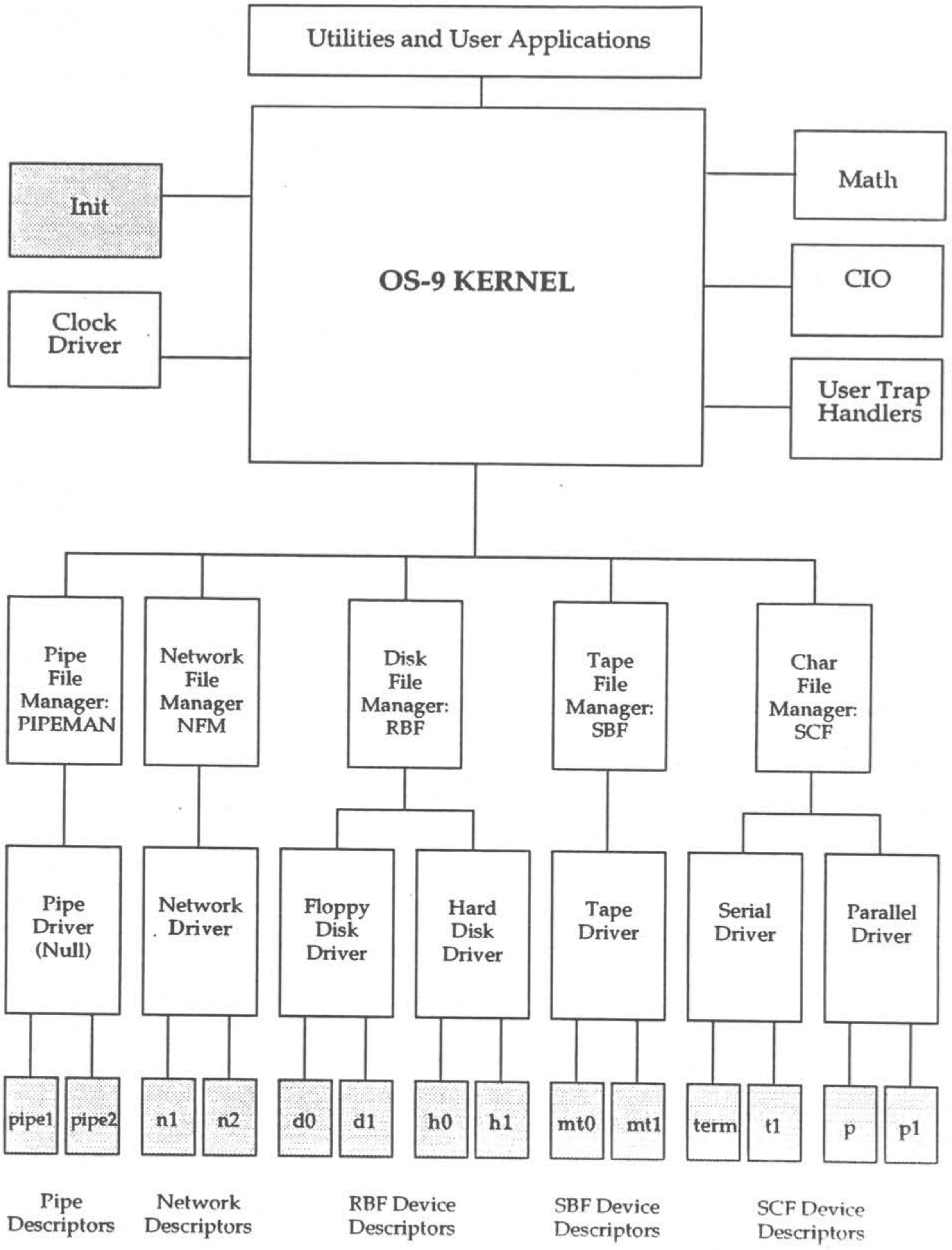
① **The Kernel, the Clock, and the Init Module**
The kernel provides basic system services. These include Input/Output (I/O) management, process control, and resource management. The clock module is a software handler for the specific real-time clock hardware. The Init module is an initialization table used by the kernel during system startup.

② **File Managers**
File managers process I/O requests for similar classes of I/O devices. File managers are hardware independent.

③ **Device Drivers**
Device drivers handle the basic physical I/O functions for specific I/O devices. Standard OS-9 systems are typically supplied with a disk driver, serial port drivers for terminals and serial printers, and a driver for parallel printers. Many users add customized drivers of their own design or purchase drivers from a hardware vendor. Device drivers are hardware dependent.

④ **Device Descriptors**
Device descriptors are small tables that associate specific I/O ports with their logical name, device driver, and file manager. These modules also contain the physical address of the port and initialization data. By use of device descriptors, only one copy of each driver is required for each specific type of I/O device regardless of how many devices the system uses.

OS-9 MODULE ORGANIZATION



For a list of the specific modules that make up OS-9 for your system, use the ident utility on the OS9Boot file.

Although all modules could be resident in ROM, the system bootstrap module is usually the only ROMed module in disk-based systems. All other modules are loaded into RAM during system startup.

I/O Overview

The OS-9 kernel does not process I/O requests directly. Instead, the kernel passes I/O requests to the appropriate file managers. Microware includes the following file managers in the standard professional distribution:

RBF The Random Block File Manager handles I/O for random-access, block-structured devices, such as floppy/hard disk drives.

SCF The Sequential Character File Manager handles I/O for sequentially character-structured devices, such as terminals, printers, and modems.

SBF The Sequential Block File Manager handles I/O for sequentially block-structured devices, such as tape drives.

PIPEMAN The Pipe File Manager handles I/O for interprocess communications through memory buffers called *pipes*.

Microware also supports the following file managers which are not included in the standard professional distribution:

PCF The PC File Manager handles reading/writing PC-DOS disks. PCF is sold separately.

NFM The Network File Manager processes data requests over the OS-9 network. NFM is included in the OS-9/NFM package.

ENPMAN The ENP10 Socket File Manager transfers requests to and from CMC ENP10 boards. ENPMAN is included in OS-9/ESP, the Ethernet Support Package.

SOCKMAN The Socket File Manager creates and manages the interface to communication protocols (sockets). SOCKMAN is included in OS-9/ISP, the Internet Support Package.

IFMAN The Communications Interface File Manager manages network interfaces. IFMAN is included in OS-9/ISP, the Internet Support Package.

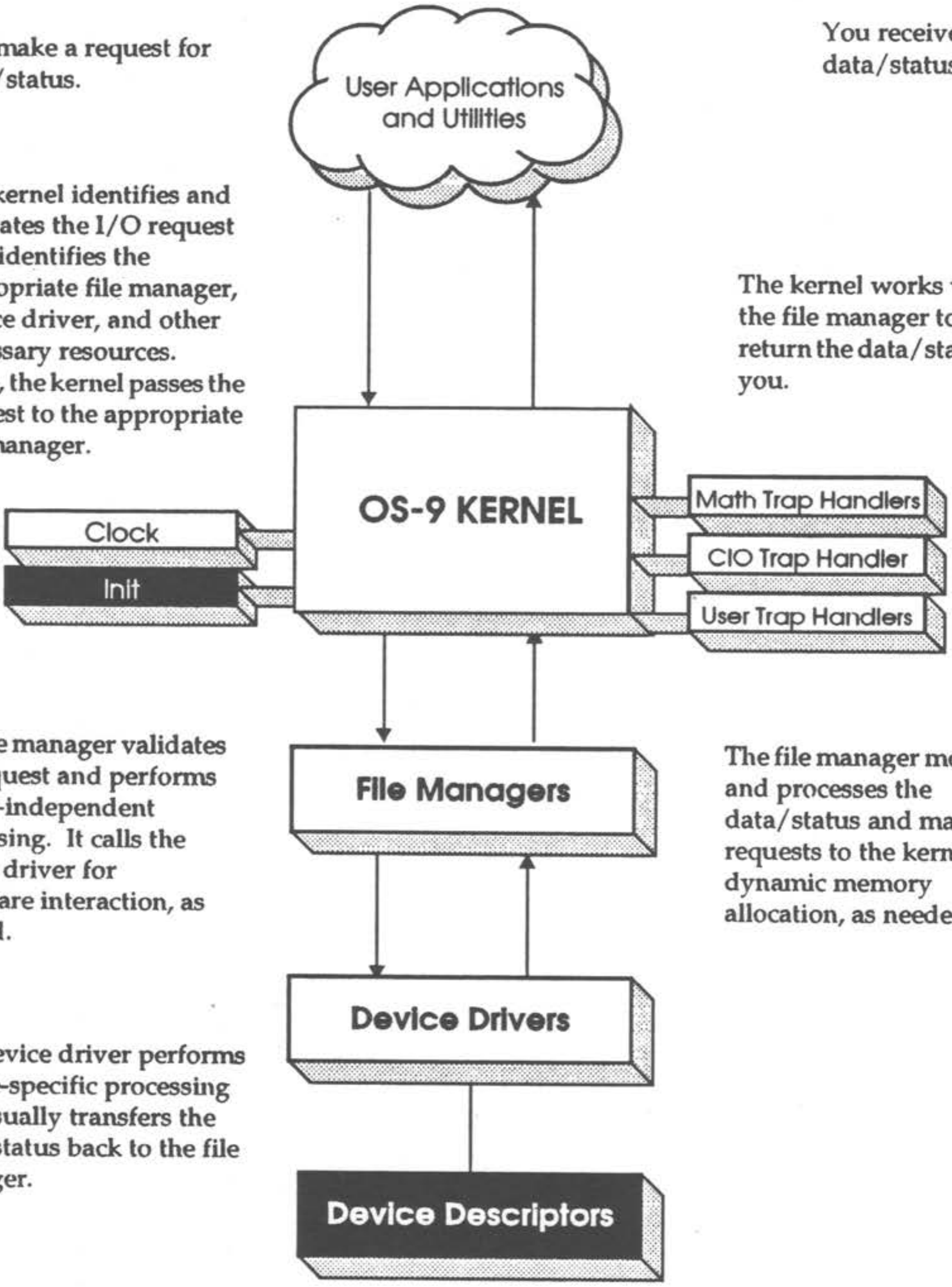
PKMAN The Pseudo-Keyboard File Manager provides an interface to the driver side of SCF to enable the software to emulate a terminal. PKMAN is included in the OS-9/ESP and OS-9/ISP Packages.

You make a request for data/status.

You receive the data/status.

The kernel identifies and validates the I/O request and identifies the appropriate file manager, device driver, and other necessary resources. Then, the kernel passes the request to the appropriate file manager.

The kernel works with the file manager to return the data/status to you.



The file manager validates the request and performs device-independent processing. It calls the device driver for hardware interaction, as needed.

The file manager monitors and processes the data/status and makes requests to the kernel for dynamic memory allocation, as needed.

The device driver performs device-specific processing and usually transfers the data/status back to the file manager.

The black boxes contain non-executable code. These modules are not "called," but are referenced. The descriptors are directly referenced by the kernel, file managers, and drivers. The Init module is directly referenced only by the kernel.

The Init Module

The Init module is sometimes referred to as the ***configuration module***. It is a non-executable module located in memory in the OS9Boot file or in ROM. The Init module contains system parameters used to configure OS-9 during startup. The parameters set up the initial table sizes and system device names. For example, the amount of memory to be allocated for internal tables, the name of the first program to be run (usually either SysGo or shell), an initial directory, etc. are specified. The system limits defined in the Init module may be examined at any time.

NOTE: The Init module MUST be present in the system in order for OS-9 to work.

The values in the Init module's table are the system defaults. These defaults can be changed in two ways:

- Modify the Init module with the moded utility.
- Edit the CONFIG macro in the systype.d file. The systype.d file is located in the DEFS directory. After systype.d is edited, the Init module is remade and placed in the new bootfile.

Both methods are discussed later in this section. Regardless of the method used, the changes made become the system defaults.

The following is a list of the system defaults located in the Init module. *Offset* refers to the location of a module field, relative to the starting address of the module. Module offsets are resolved in assembly code by using the names shown here and linking the module with the relocatable library: sys.l or usr.l.

Offset	Name	Description
$30	Reserved	Currently reserved for future use
$34	M$PollSz	Number of IRQ polling table entries
$36	M$DevCnt	Device table size
$38	M$Procs	Initial process table size
$3A	M$Paths	Initial path table size
$3C	M$SParam	Parameter string for startup module
$3E	M$SysGo	Offset to name string of first executable module
$40	M$SysDev	Offset to the initial default directory name string
$42	M$Consol	Offset to the initial I/O pathlist string
$44	M$Extens	Offset to a name string of customization modules
$46	M$Clock	Offset to the clock module name string
$48	M$Slice	Number of clock ticks per time-slice
$4A	Reserved	Currently reserved for future use
$4C	M$Site	Offset to the installation site code
$50	M$Instal	Offset to the installation name string
$52	M$CPUTyp	CPU type
$56	M$OS9Lvl	Level, version, and edition number
$5A	M$OS9Rev	Offset to the OS-9 level/revision string
$5C	M$SysPri	Initial system priority
$5E	M$MinPty	Initial system minimum executable priority
$60	M$MaxAge	Initial system maximum natural age
$62	Reserved	Currently reserved for future use
$66	M$Events	Initial number of entries allowed in the events table
$68	M$Compat	Compatibility flag one - Byte is used for revision compatibility
$69	M$Compat2	Compatibility flag two
$6A	M$MemList	Offset to the memory segment list
$6C	M$IRQStk	Size of the kernel's IRQ stack
$6E	M$ColdTrys	Retry counter if the kernel's initial chd fails

Name	Description
M$PollSz	Number of entries in the IRQ polling table. One entry is required for each interrupt-generating device. The IRQ polling table has 32 entries by default. Each entry in the IRQ polling table is 18 bytes long.
M$DevCnt	Number of entries in the system device table. One entry is required for each device in the system. The system device table has 32 entries by default. Each entry in this table is 18 bytes long.
M$Procs	Initial number of active processes allowed in the system. If this table becomes full, it automatically expands as needed. By default, 64 active processes are allowed. Each entry in the initial process table requires 4 bytes.
M$Paths	Initial number of open paths in the system. If this table becomes full, it automatically expands as needed. By default, 64 open paths are allowed. Each entry in the initial path table requires 4 bytes.
M$SParam	Offset to the parameter string (if any) passed to the first executable module. An offset of 0 indicates that no parameter string is required. The parameter string itself is located elsewhere, usually near the end of the Init module.
M$SysGo	Offset to the name string of the first executable module; usually SysGo or shell.
M$SysDev	Offset to the initial default directory name string; usually /d0 or /h0. The kernel does a chd and chx to this device prior to forking the initial device. If the system does not use disks, this offset must be zero.
M$Consol	Offset to the initial I/O pathlist string. This offset usually points to the /TERM string. This pathlist is opened as the standard I/O path for the initial process. It is generally used to set up the initial I/O paths to and from a terminal. This offset should contain zero if no console device is in use.

Name	Description
M$Extens	Offset to a name string of a list of customization modules (if any). A customization module can be used to complement or change OS-9's existing standard system calls. These modules are searched for during startup and are typically found in the bootfile. They are executed in system state if found. Modules listed in the name string are separated by spaces. The default name string to be searched for is OS9P2. If there are no customization modules, set this value to zero. **NOTE:** A customization module may only alter the d0, d1, and ccr registers.
M$Clock	Offset to the clock module name string. If there is no clock module name string, set this value to zero.
M$Slice	Number of clock ticks per time-slice. The number of clock ticks per time-slice defaults to 2.
M$Site	Offset to the installation site code. This value is usually set to zero. OS-9 does not currently use this field.
M$Instal	Offset to the installation name string.
M$CPUTyp	CPU type: 68000, 68008, 68010, 68020, 68030, or 68070. The default is 68000.
M$OS9Lvl	This four byte field is divided into three parts: `level: 1 byte   version: 2 bytes   edition: 1 byte` For example, level 1, version 2.3, edition 1 would be 1231.
M$OS9Rev	Offset to the OS-9 level/revision string.
M$SysPri	System priority at which the first module (usually SysGo or shell) is executed. This is generally the base priority at which all processes start. The default is 128.
M$MinPty	Initial system minimum executable priority. The default is 0.
M$MaxAge	Initial system maximum natural age. The default is 0.
M$Events	Initial number of entries allowed in the events table. If the table becomes full, it automatically expands as needed. The default is 0. Each entry in the events table requires 32 bytes. This value is no longer used.

Name **Description**

M$Compat Revision compatibility. The default is 0. If set, the following bits are currently defined:

Bit 0: Saves all registers for IRQ routines

Bit 1: Prevents the kernel from using stop instructions

Bit 2: Ignores the "sticky" bit in module headers

Bit 3: Disables cache burst operation (68030 systems)

Bit 4: Patternizes memory when memory is allocated or deallocated

Bit 5: Prevents kernel cold-start from starting system clock.

M$Compat2 Indicates the "absence/snoopiness" of the system caches

Bit#	Function	
0	0	External instruction cache is not snoopy*
	1	External instruction cache is snoopy or absent
1	0	External data cache is not snoopy
	1	External data cache is snoopy or absent
2	0	On-chip instruction cache is not snoopy
	1	On-chip instruction cache is snoopy or absent
3	0	On-chip data cache is not snoopy
	1	On-chip data cache is snoopy or absent
7	0	Kernel disables data caches when in I/O
	1	Kernel does not disable data caches when in I/O

*.snoopy = cache that maintains its integrity without software intervention.

M$MemList Offset to the memory segment list. The colored memory list contains an entry for each type of memory in the system. The list is terminated by a long word of zero. If this field contains a 0, colored memory is not used in this system.

M$IRQStk Size (in longwords) of the kernel's IRQ stack. The value must be 0 or between 256 and $ffff. If the value is zero, the kernel uses a small default IRQ stack. A larger IRQ stack is recommended. The default value is 256 longwords.

M$ColdTrys Retry counter if the kernel's initial chd to the system device fails. The default value is 0.

The following is a portion of the distributed init.a file:

```
_INITMOD equ 1 flag reading init module
CPUTyp  set 68000  cpu type (68008/68000/68010)
Level   set 1      OS-9 Level One
Vers    set 2      Version 2.4
Revis   set 4
Edit    set 1      Edition
IP_ID   set 0      interprocessor identification code
Site    set 0      installation site code
MDirSz  set 128    initial module directory size (unused)
PollSz  set 32     IRQ polling table size (fixed)
DevCnt  set 32     device table size (fixed)
Procs   set 64     initial process table size (divisible by 64)
Paths   set 64     initial path table size (divisible by 64)
Slice   set 2      ticks per time slice
SysPri  set 128    initial system priority
```

For more information on the Init module, refer to the ***OS-9 Technical Manual***.

Changing System Modules

System modules have been configured to satisfy the needs of the majority of users. However, you may want to change the existing modules or create new modules. New system modules and alterations to existing system modules are made by using the moded utility or changing the defaults in the systype.d file. The system modules most commonly altered are the device descriptors and the Init module.

Using the Moded Utility

The moded utility is used to edit individual fields of certain types of OS-9 modules. You can use moded to change a disk-based Init module and other disk-based OS-9 Device Descriptor modules.

To use the moded utility, type moded, the name of the desired device descriptor, and any options.

The moded: prompt shows that the editor's command mode has been entered.

When moded is invoked, it attempts to read the /dd/SYS/moded.fields file. Moded.fields contains module field information for each type of module to be edited. Without this file, moded cannot function.

The provided moded.fields file comes with module descriptions for standard RBF, SBF, SCF, PIPE, NETWORK, UCM, and GFM module descriptors. It also includes a description for the Init module.

To edit the current module, use the e command. If there is no current module, the editor prompts for the module name to edit. The editor then prints the name of a field, its current value, and prompt for a new value.

The following edit commands are available:

Command	Description
<expr>	A new value for the field
-	Re-display previous field
.	Leave edit mode
?	Print edit mode commands
??	Print description of the current field
<cr>	Leave current value unchanged

If the definition of any field is unfamiliar, use the ?? command to provide a short description of the current field.

Once all necessary changes are made to the module, exit edit mode by reaching the end of the module or by typing a period (.). At this point, the changes made to the module exist only in memory. To write the changes to the actual file, use the w command. This also updates the module header parity and CRC.

NOTE: moded is mainly used for editing existing descriptors. It is somewhat restrictive, and as a result, if you are building a device descriptor or changing a field so it has more characters than the current field, you may not want to use moded.

Editing the Systype.d File

The second method of changing system modules requires editing the systype.d file located in the DEFS directory. The systype.d file contains macros such as TERM, DiskH0, etc. for each device descriptor and the Init module. These macros contain basic memory map information, exception vector methods (for example, vectors in RAM or ROM), I/O device controller memory addresses and initialization data, etc. for each device descriptor and for the Init module.

The systype.d file consists of five main sections that are used when installing OS-9:

- Init module CONFIG macro
- SCF Device Descriptor macros and definitions
- RBF Device Descriptor macros and definitions
- ROM configuration values
- Target system specific definitions

The CONFIG macro is used when creating the Init module to determine six or more system dependent variables:

MainFram — A character string used by programs such as login to print a banner identifying the system. You may modify this string.

SysStart — A character string used by the OS-9 kernel to locate the initial process for the system. This process is usually the SysGo module. Two versions of SysGo are provided in the files, SysGo.a for disk-based OS-9 and SysGo_nodisk.a for ROM-based OS-9.

SysParam — A character string passed to the initial process. This usually consists of a single carriage return.

SysDev A character string containing the name of the path to the initial system disk. The kernel coldstart routine sets the initial execution and data directories to this device prior to forking the SysStart process. Set this label to zero for a ROM-based system. For example, SysDev set 0.

ConsolNm A character string containing the name of the path to the console terminal port. Messages to be printed during startup appear here.

ClockNm A character string containing the name of the clock module.

Other system parameters may be set in this macro to override the default values created by the init.a source file. This allows you to perform "system tuning" without modifying the generic init.a file.

The following is a portion of an example systype.d file:

```
CONFIG macro

  endm
  Slice set 2
  ifdef _INITMOD
 Compat set ZapMem patternize memory
  endc
```

When editing the Init module, constants may use either values or labels. CPUTyp set 68020 is an example of a constant that uses a value. These constants may appear anywhere in the systype.d file. Compat set ZapMem is an example of a constant that uses a label. These constants must appear outside the CONFIG macro and must be conditionalized to be invoked only when init.a is being assembled. If these values are placed inside the CONFIG macro, the old defaults are used. If a constant that requires a label is placed outside the macro and not conditionalized, illegal external reference errors result when making other files. Use the _INITMOD label to avoid these errors.

Interprocess Communications

OS-9 supports five forms of interprocess communication: signals, alarms, events, pipes, and data modules. ***Pipes*** transfer data among concurrent processes. ***Data modules*** transfer or share data among concurrent processes. ***Signals*** can be used to synchronize concurrent processes. ***Alarms*** send signals or execute subroutines at specified times. ***Events*** can be used to synchronize concurrent processes' access of shared resources.

Pipes

An OS-9 ***pipe*** is a first-in first-out (FIFO) buffer which enables concurrently executing processes to communicate data: the output of one process (the writer) is read as input by a second process (the reader). Communication through pipes eliminates the need for an intermediate file to hold data.

For example, assume that Process A creates a pipe named /pipe/temp to write to and read from it. Process B opens the same pipe to read from it:

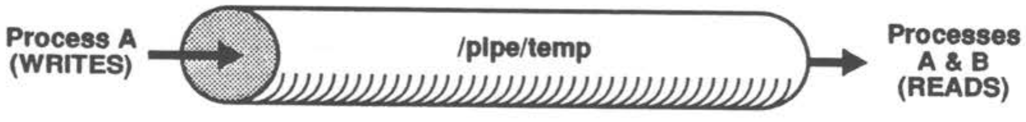


Figure 1

Process A writes Hello_There! into the pipe.

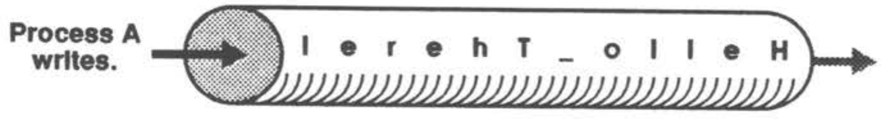


Figure 2

Process B reads six characters out of the pipe:

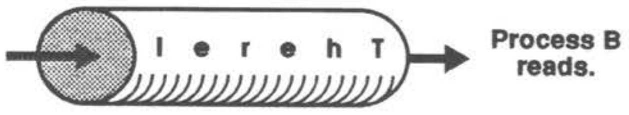


Figure 3

Process A reads the last six characters out of the pipe:

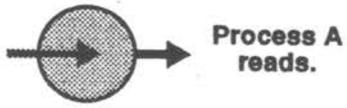


Figure 4

The previous example illustrates two important characteristics of pipes:

- Any process may read data out of a pipe, even the process that wrote the data.
- Data does not have to be read out of the pipe in the same size sections in which it was written.

A pipe may contain up to 90 bytes, unless a different buffer size has been declared. Typically, a pipe is used as a one-way data path between two processes: one writing and one reading. The reader waits for the data to become available and the writer waits for the buffer to empty:

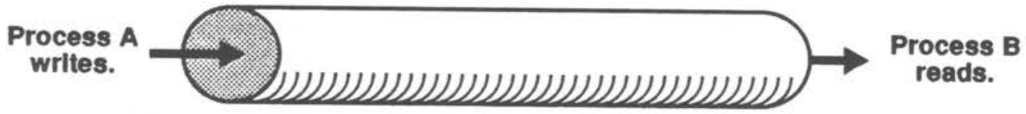


Figure 5

However, any number of processes can access the same pipe simultaneously: the pipe file manager coordinates these processes. A process can even arrange for a single pipe to have data sent to itself. This could be used to simplify type conversions by printing data into the pipe and reading it back using a different format.

```
/************************************ | /****************************************
* pipedemo1.c                       * | * pipedemo2.c                           *
*                                   * | *                                       *
* This program creates a pipe and   * | * This program opens the pipe that      *
* writes some text to it.  After    * | * pipedemo1.c created and reads some    *
* pipedemo2 is given adequate time  * | * of the text that is in the pipe.      *
* to read some of the text, this    * | *                                       *
* program reads the rest of the     * | *                                       *
* text.                             * | *                                       *
* Compile with:                     * | * Compile with:                         *
*    cc pipedemo1.c                 * | *        cc pipedemo2.c                 *
* Execute with:                     * | * Execute with:                         *
*    pipedemo1 & pipedemo2          * | *        pipedemo1 & pipedemo2          *
************************************/ | ***************************************/
                                      |
#include <modes.h>                    |#include <modes.h>
#include <errno.h>                    |#include <errno.h>
                                      |
#define OWNER   S_IWRITE | S_IREAD    |#define OWNER   S_IWRITE | S_IREAD
#define GRP     OWNER                 |#define GRP     OWNER
#define PUBLIC  S_IOREAD | S_IOWRITE  |#define PUBLIC  S_IOREAD | S_IOWRITE
#define ALL     OWNER | PUBLIC        |#define ALL     OWNER | PUBLIC
#define NAME    "/pipe/temp"          |#define NAME    "/pipe/temp"
#define ERR_MSG "Can't open Pipe"     |#define ERR_MSG "Can't open Pipe"
#define ERROR (-1)                    |#define ERROR    (-1)
#define NOERROR (0)                   |#define NOERROR  (0)
                                      |
main()                                |main()
{                                     |{
  int op,  x;                         |  int op, x;
  char msg[10];                       |  char msg[10];
                                      |
/* First try to create() the pipe,*/  | /* First try to create() the pipe, */
/* if it fails then some other     */ | /* if it fails then some other      */
/* process has created it.         */ | /* process has created it.          */
/* Therefore, all this process has*/  | /* Therefore, all this process has */
/* to do is open it.               */ | /* to do is open it.                */
/* If the open() fails, then exit */  | /* If the open() fails, then exit   */
/* with an error message.          */ | /* with an error message.           */
                                      |
 if((op=create(NAME,GRP,ALL))==ERROR)| if((op=create(NAME,GRP,ALL))==ERROR)
```

```
   if((op=open(NAME,GRP))==ERROR)      |  if((op=open(NAME,GRP))==ERROR)
     exit(_errmsg(errno,ERR_MSG));     |    exit(_errmsg(errno,ERR_MSG));
                                       |
  /* Now write text into the pipe   */| /* Sleep awhile waiting for       */
  if(write(op,"Hello There!",12) !=12)| /* pipedemo1.c to write some text */
     exit(_errmsg(errno,"OOPS\n"));   | sleep(1);
                                      |
  /* sleep awhile waiting for       */ | /* Now read text into the pipe   */
  /* pipedemo2.c to read some text */ | if((x=read(op,msg,6)) != 6 )
  sleep(1);                           |   exit(_errmsg(errno,"OOPS\n"));
                                      | msg[x] = '\0';
  /* Now read 6 remaining bytes     */ |
  x = read(op,msg,6);                 |
  msg[x] = '\0';                      |
                                      |
  /* Print the results              */ | /* Print the results             */
  _errmsg(NOERROR,"%s\n",msg);        | _errmsg(NOERROR,"%s\n",msg);
                                      |
  /* End of program                 */ | /* End of program                */
  exit(0);                            | exit(0);
}                                     |}
```

Pipes are commonly used to send and receive data between two processes. Two pipes are required in this situation because processes cannot determine who wrote the data in a pipe:

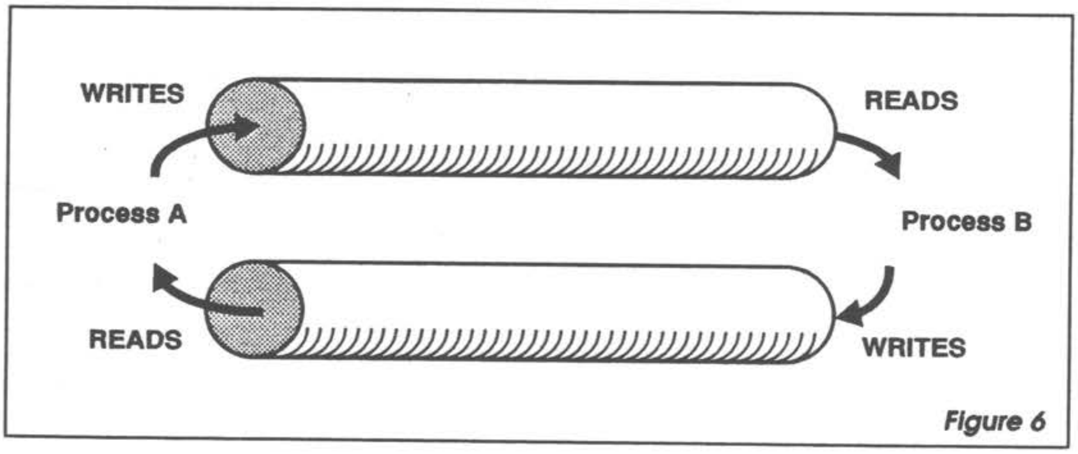


Figure 6

Another common use of pipes involves two processes sending data to a third process:

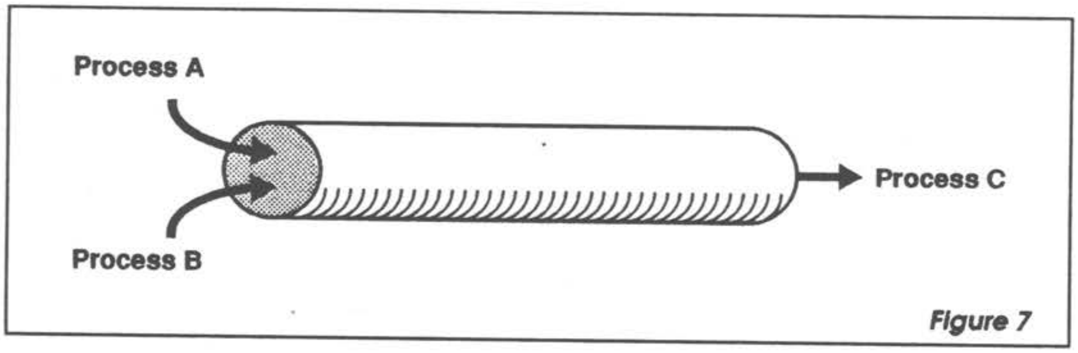


Figure 7

Pipes can be used much like signals to coordinate processes, but with these advantages:

- Longer messages (more than 16 bits)
- Queued messages
- Determination of pending messages
- Easy process-independent coordination (using named pipes)

Pipeman

Pipeman is the OS-9 file manager that supports interprocess communication through pipes. Pipeman is a re-entrant subroutine package that is called for I/O service requests to a device named /pipe. Although no physical device is used in pipe communications, a driver must be specified in the pipe descriptor module. The null driver (a driver that does nothing) is usually used, but only gets called by pipeman for GetStat/SetStat calls.

Named and Unnamed Pipes

OS-9 supports both named and unnamed (anonymous) pipes. Unnamed pipes are used extensively by the shell to construct program "pipelines," but may also be used by user programs. Unnamed pipes may be opened only once. Independent processes may communicate through an unnamed pipe only if the pipeline was constructed by a parent (or grandparent, great-grandparent, etc.) common to the processes. This is accomplished by making each process inherit the pipe path as one of its standard I/O paths.

Named and unnamed pipes function nearly identically. The main difference is that a named pipe may be opened by several independent processes, which simplifies pipeline construction. Other specific differences are noted in the following section.

Operations on Pipes:

Creating Pipes

The I$Create system call is used with the pipeman to create new named or unnamed pipe files. Pipes may be created using the pathlist /pipe (for unnamed pipes, "pipe" is the name of the pipe device descriptor) or /pipe/<name> (<name> is the logical file name being created). If a pipe file with the same name already exists, an error (E$CEF) is returned. Unnamed pipes cannot return this error.

All processes connected to a particular pipe share the same physical path descriptor. Consequently, the path is automatically set to "update" mode regardless of the mode specified at creation. Access permissions may be specified and are handled in the same manner as RBF's access permissions.

The size of the default FIFO buffer associated with a pipe is specified in the pipe device descriptor. When creating a pipe, you can override this default by setting the initial file size bit of the mode byte and passing the desired "file size" in register d2.

If no default or overriding size is specified, a 90-byte FIFO buffer inside the path descriptor is used.

Opening Pipes

When accessing unnamed pipes, I$Open, like I$Create, opens a new anonymous pipe file. When accessing named pipes, I$Open searches for the specified name through a linked list of named pipes associated with a particular pipe device. If the pipe is found, the path number returned refers to the same physical path that was allocated when the pipe was created. Internally, this works similarly to the I$Dup system call.

Opening an unnamed pipe is simple, but sharing the pipe with another process is more complex. If a new path to /pipe is opened for the second process, the new path would be independent of the old one.

The main advantage of using named pipes is that several processes may communicate through the same named pipe without having to inherit it from a common parent process.

NOTE: The OS-9 shell always constructs its pipelines using the unnamed /pipe descriptor.

Read & ReadLn

The I$Read and I$ReadLn system calls return the next bytes in the pipe buffer. If there is not enough data ready to satisfy the request, the process reading the pipe is put to sleep until more data becomes available.

The end-of-file is recognized when the pipe is empty and the number of processes waiting to read from the pipe is equal to the number of users of the pipe. If any data was read before end-of-file was reached, an end-of-file error is not returned. However, the byte count returned will be the number of bytes actually transferred, which will be less than the number requested.

NOTE: The I$Read and I$Write system calls are faster than I$ReadLn and I$WritLn because pipeman does not have to check for carriage returns and the loops moving data are tighter.

Write & WriteLn

The I$Write and I$WritLn system calls work in almost the same way as I$Read and I$ReadLn. A pipe error (E$Write) is returned when all the processes who have a full unnamed pipe open are attempting to write to the pipe. The first process attempting to write to the pipe receives the error and the pipe remains full.

When named pipes are being used, pipeman never returns the E$Write error. If a named pipe becomes full before a process that receives data from the pipe has opened it, the process writing to the pipe is put to sleep until a process reads the pipe.

Close

When a pipe path is closed, its path count is decremented. If no paths are left open on an unnamed pipe, its memory is returned to the system. With named pipes, its memory is returned only if the pipe is empty. A non-empty pipe (with no open paths) is artificially kept known to the system, waiting for another process to open and read from the pipe. This permits pipes to be used as a type of a temporary, self-destructing "RAM disk file."

The I$MakDir and I$ChgDir service requests are illegal service routines on pipes. They return E$UnkSvc (unknown service request).

The following chart summarizes the behavior of named and unnamed pipes in potential error situations:

	Named Pipes	Unnamed Pipes
Write to a Full Pipe	Processes that attempt to write to a full named pipe are queued. (No E$Write errors.)	When all processes with a full pipe open are attempting to write to the pipe, the first process will receive an E$Write error.
Read from an Empty Pipe (No data read before EOF)	When all processes with an empty pipe open are attempting to read from the pipe, the first process will receive an E$EOF error.	When all processes with an empty pipe open are attempting to read from the pipe, the first process will receive an E$EOF error.
Close a Non-Empty Pipe	A named pipe is deallocated only when the link count is zero and the pipe is empty.	An unnamed pipe is deallocated when its link count becomes zero, regardless of the pipe's contents.
Create a Pipe of the Same Name	Any process that attempts to create a named pipe with the same name as an existing pipe will receive an E$CEF error.	Any process that uses unnamed pipes exclusively cannot receive an E$CEF error.
Open a Pipe that does not Exist	Any process that attempts to open a named pipe that does not exist will receive an E$PNNF error.	For unnamed pipes, opening a pipe is the same as creating a pipe.

Figure 8 : Named vs. Unnamed Pipes

Pipe Directories

Opening an unnamed pipe in the "dir" mode allows it to be opened for reading. In this case, pipeman allocates a pipe buffer and pre-initializes it to contain the names of all open named pipes on the specified device. Each name is null-padded to make a 32-byte record. This allows utilities, that normally read an RBF directory file sequentially, to work with pipes as well.

GetStat & SetStat

Pipeman supports a wide range of status codes, to allow pipes to be inserted between processes where an RBF or SCF device would normally be used. For this reason, most RBF and SCF status codes are implemented to do something without returning an error. The actual function may differ slightly from the other file managers, but it is usually compatible.

GetStat Status Codes

Status Code	C Binding
SS_Opt	_gs_opt(path, buffer) Reads the 128-byte option section of the path descriptor. It can be used to obtain the path type, data buffer size and name of pipe.
SS_Ready	_gs_rdy(path) Tests whether data is ready. It returns the number of bytes in the buffer.
SS_Size	_gs_size(path) Returns the size of the pipe buffer.
SS_EOF	_gs_eof(path) Tests for end-of-file.
SS_FD	_gs_gfd(path, buffer, count) Returns a pseudo-file descriptor image.

SetStat Status Codes

Status Code	C Binding
SS_Opt	_ss_opt(path, buffer) Does nothing, but returns without error.
SS_Size	_ss_size(path, size) Resets the pipe buffer if the specified size is zero. Otherwise, it has no effect, but returns without error.
SS_FD	_ss_pfd(path, buffer) Does nothing, but returns without error.
SS_Attr	_ss_attr(path, attr) Changes the pipe file's attributes.
SS_SSig	_ss_ssig(path, sigcode) Sends a signal when the data becomes available.
SS_Relea	_ss_rel(path) Releases the device from the SS_SSig processing before data becomes available.

Other GetStat/SetStat codes are passed to the device driver.

NOTE: Remember that pipeman is not a true directory device, so commands like chd and makdir do not work with /pipe.

Data Modules

OS-9 data modules enable multiple processes to share a data area and to transfer data among themselves. A ***data module*** must have a valid CRC and module header, and can be non-re-entrant. That is, a data module can modify itself and be modified by several processes.

OS-9 does not have restrictions as to the content, organization, or usage of the data area in a data module. These considerations are determined by the processes using the data module.

OS-9 does not synchronize processes using a data module. Consequently, thoughtful programming, usually involving events or signals, is required to enable several processes to update a shared data module simultaneously.

Two data structures must be considered when referring to data modules: the structure that defines the module itself and the user-defined structure that the data module contains. The user-defined structure is simply a C language structure. For example, Figure 9 illustrates the data module used in the program described later in this section:

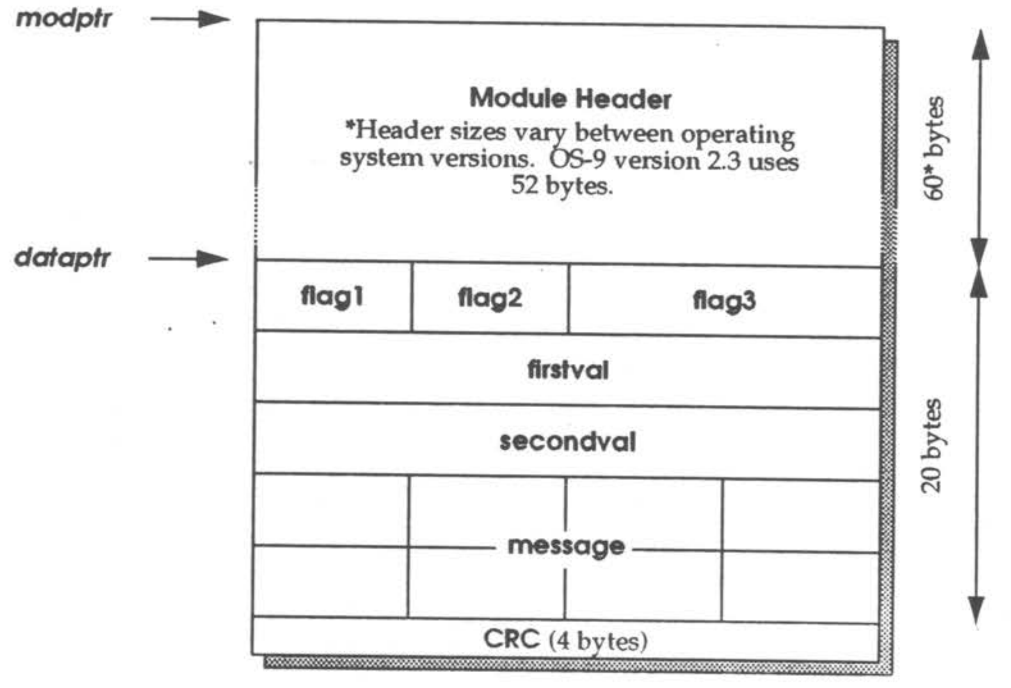


Figure 9: Example Data Module

NOTE: The _mkdata_module() C call returns a pointer to the beginning of the module header, ***not*** to the data section.

Creating Data Modules

The F$DatMod system call creates a data module with a specified data area size, module name, and set of attributes. The data area is cleared automatically. The data module is created with a valid CRC and entered into the system module directory.

NOTE: It is essential that the data module's header and name string not be modified to prevent the module from becoming unknown to the system.

The Microware C compiler supports several C calls to create and use data modules directly:

_mkdata_module(name,size,attr,perm)
: Creates a data module with the specified name, size, attributes and permissions.

make_module(name,size,attr,perm, typelang, color)
: Creates a memory module with the specified name and attributes in the specified memory.

modcload(modname, mode, memtype)
: Searches the module directory for a module with the same name as that pointed to by modname and links to it.

modlink(modname,typelang)
: Searches the module directory for the specified module and links to the module if the type and language match typelang.

modload(modname,accessmode)
: Searches the module directory for the specified module, loads the module, and links to the module.

modloadp(modname,mode, name)
: Loads and links a module. It uses the PATH environment variable to determine alternate directories to search for the named module.

munlink(module)
: Informs the system that the specified module is no longer required by the process. (The parameter module is a pointer to a module.)

munload(name,type)
: Informs the system that the specified module is no longer required by the process. (The parameter name is a pointer to the name of a module.)

The _mkdata_module() function is specific to data modules, while the modlink(), modload(), munlink(), and munload() functions apply to all OS-9 modules. For more information on these calls, refer to the standard library sections of the ***OS-9 C Compiler User's Manual***.

Link Count

Like all OS-9 modules, data modules have a link count associated with them. The link count is a counter of how many processes are currently linked to a module. Generally, a module is removed from memory when this count becomes 0. However, a "sticky" module is retained in memory until its link count becomes -1 or memory is required for another use. A module is "sticky" if the sixth bit of the module header's attribute byte is set.

Saving to Disk

If a data module is saved to disk, you can examine the module's format and contents with the dump utility. A data module can be saved to disk with the save utility or by writing the module image into a file. If the data module was modified since its creation, the saved module's CRC will be bad and it will be impossible to re-load the module into memory. To enable the module to be re-loaded, use the F$SetCRC system call or _setcrc() C library call before writing the module to disk. Or, use the fixmod utility after the module has been written to disk.

Data Modules: Example Program

Any number of independent processes can access a data module by using modlink() or modload().The example program illustrates how a data module is created and how individual elements of the module's data structure can be manipulated.

```
#include <stdio.h>
#include <module.h>
#include <errno.h>
#define ERROR -1
#define REVS 0x00
typedef struct datastr {
    unsigned char flag1;
    char flag2;
    short flag3;
    long firstval,
    secondval;
    char message[8];
    } dx;

main()
{
    char * _mkdata_module(),
         * modlink(),
           *modptr;
    dx   * dataptr;
```

Note that dataptr is declared as a pointer to an object of type dx, a previously defined data type. This could be any data structure that the user wishes to be internal to the data module.

```
    char * strcpy(),
         *modnam;

    unsigned modsize;
    short attributes,
          permissions;

    modnam = "demodule";
    modsize = sizeof(dx);

    permissions = MP_OWNER_READ | MP_OWNER_WRITE;
    attributes = (MA_REENT << 8) | REVS;
          /* or */
    /* attributes = ((MA_REENT | MA_GHOST) << 8) | REVS; */
```

The following section actually makes the data module. The modptr parameter will contain the starting address of the module or -1, if an error occurred.

```
if ((modptr=_mkdata_module(modnam,modsize,attributes permissions))
                                                  ==(char *)ERROR)
       exit(_errmsg(errno,"Unable to Create Data Module"));
else
       printf("Starting Address of Module %s = %x\n",modnam,modptr);
```

The following line calculates a pointer to the data portion of the module by adding the execution offset (an offset to the data) to the pointer to the beginning of the module header. Refer to Figure 9 for an illustration of these locations.

```
    dataptr = (dx *)(modptr + ((mod_exec *)modptr)->_mexec);
```

The following section puts string data into the module:

```
    printf("%c %c \n",dataptr->flag1, dataptr->flag2);
    dataptr->flag1 = 'R';
    dataptr->flag2 = dataptr->flag1;
    dataptr->flag3 = 0xfeed;
    dataptr->firstval = 0xabcd1234;
    dataptr->secondval = dataptr->firstval + 10;
    strcpy(dataptr->message,"FOOBAR");
```

When the process is finished with the data module, either munlink() or munload() should be executed to decrement the module's link count. This section has been commented out of the example program to enable the user to examine the module's contents.

NOTE: munlink() requires a pointer to the module, while munload() requires a pointer to the module's name.

```
    /* if (munlink(modptr) == ERROR)
          exit(_errmsg(errno,"Couldn't unlink from data module")); */
    /* or */
    /* if (munload("demodule",(MT_DATA << 8) ) == ERROR)
           exit(_errmsg(errno,"Couldn't unload data module")); */

}
```

Data Modules: Example Program

The example program is listed in its entirety on the following pages.

```
/*************************************************************
* Program: datmod.c                                          *
* Function: This program demonstrates how to use Data Modules. *
*************************************************************/

#include <stdio.h>
#include <module.h>                    /* This defines mod_exec and _mexec */
#include <errno.h>                                /* Allows the use of errno */

#define ERROR -1                               /* Errors are denoted by a -1. */
#define REVS 0x00                /* The revision number of created Data Module */


typedef struct datastr {                  /* The data elements that will be in the */
    unsigned char flag1;                          /* data module are defined here. */
    char flag2;
    short flag3;
    long firstval,
    secondval;
    char message[8];
    } dx;                             /* the name of this typedef structure is dx */

main()
{
    char *_mkdata_module(),           /* declare this function to return (char *) */
    *modlink(),
    *modptr;                          /* declare this variable to return (char *) */

    dx * dataptr;                 /* dataptr is a ptr to the defined structure dx */
    char * strcpy(),
    *modnam;                                       /* pointer to the module name. */

    unsigned modsize;                     /* size of the data area of data module. */
    short attributes,                  /* the attributes given to the data module. */
    permissions;                        /* the permissions given this data module. */

    modnam = “demodule”;        /* The name of the data module will be “demodule” */
    modsize = sizeof(dx);           /* The size is that of the defined structure. */

    permissions = MP_OWNER_READ | MP_OWNER_WRITE;      /* owner can read & write */
    attributes = (MA_REENT << 8) | REVS;    /*The Attribute/Revision definition */
          /* or */
    /* attributes = ((MA_REENT | MA_GHOST) << 8) | REVS;    /* for sticky module */
```

```
                                        /* Create the Date Module               */
                                        /* Note: _mkdata_module() will create   */
                                        /* a new data module even if there's one */
                                        /* already in memory by the same name   */
                                        /* and its Link Count = 0               */
if ((modptr=_mkdata_module(modnam,modsize,attributes permissions))==(char *)ERROR)

                                         /* if error, print error message */
        exit(_errmsg(errno,"Unable to Create Data Module"));
    else                                              /* else print some information */
        printf("Starting Address of Module %s = %x\n",modnam,modptr);

                                                   /* Figure dataptr from modptr */
    dataptr = (dx *)(modptr + ((mod_exec *)modptr)->_mexec);

                                 /* Now, use the data module elements as if they */
                                        /* were elements of any other structure. */
    printf("%c %c \n",dataptr->flag1, dataptr->flag2);
    dataptr->flag1 = 'R';
    dataptr->flag2 = dataptr->flag1;
    dataptr->flag3 = 0xfeed;
    dataptr->firstval = 0xabcd1234;
    dataptr->secondval = dataptr->firstval + 10;
    strcpy(dataptr->message,"FOOBAR");

                                          /* Don't forget to unlink from the */
                                      /* Data Module when finished using it. */
                                    /* munlink() requires the module pointer */
    /* if (munlink(modptr) == ERROR)
           exit(_errmsg(errno,"Couldn't unlink from data module"));
                                      /* MT_DATA is the type for data module */
                                             /* and is defined in module.h. */
    /* or */                                  /* It must be left shifted 8 bits */
                                    /* munload() requires a name and a type */
    /* if (munload("demodule",(MT_DATA << 8) ) == ERROR)
           exit(_errmsg(errno,"Couldn't unload data module"));                   */

}
```

Signals

In interprocess communications, a ***signal*** is an intentional disturbance in a system. OS-9 signals are designed to synchronize concurrent processes, but they can also be used to transfer small amounts of data. **Because they are usually processed immediately, signals provide real-time communication between processes.**

Signals are also referred to as ***software interrupts*** because a process receives a signal similarly to how a CPU receives an interrupt. Signals enable a process to send a "numbered interrupt" to another process. If an active process receives a signal, the intercept routine is executed immediately (if installed) and the process resumes execution where it left off (See Figure 10).

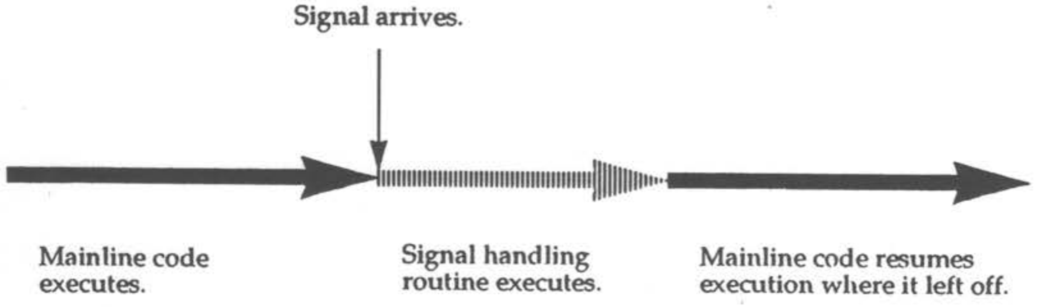

Signal arrives.
Mainline code executes.
Signal handling routine executes.
Mainline code resumes execution where it left off.

Figure 10: Signal arrives while process is executing.

If a sleeping or waiting process receives a signal, the process is moved to the active queue, the signal routine is executed, and the process resumes execution right after the call that removed it from the active queue (See Figure 11).

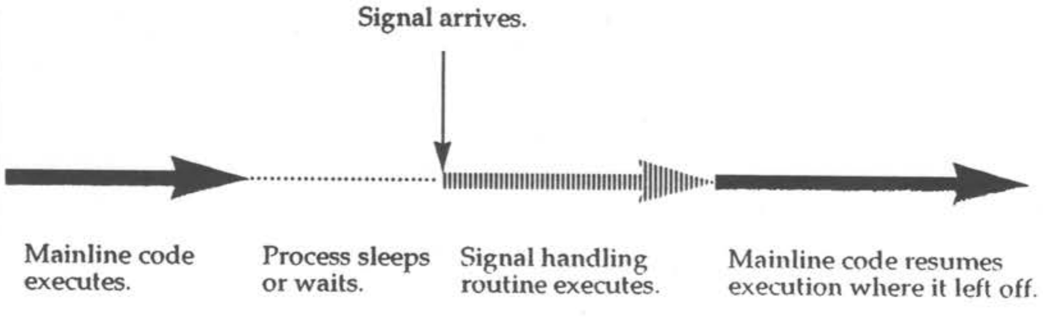

Signal arrives.
Mainline code executes.
Process sleeps or waits.
Signal handling routine executes.
Mainline code resumes execution where it left off.

Figure 11: Signal arrives while process is sleeping or waiting.

NOTE: A process which does not have an intercept routine will be killed if it receives a signal. This applies to all signals greater than 1 (wake-up signal).

Each signal has two parts: the process ID of the destination and a signal code. The following signal codes are supported:

Signal	Description
0	Unconditional system abort signal. The super-user can send the "kill" signal to any process, but non-super-users can send this signal only to processes with their group and user IDs. This signal terminates the receiving process, regardless of the state of its signal mask, and is not intercepted by the intercept handler.
1	Wake-up signal. Sleeping/waiting processes which receive this signal are awakened, but the signal is not intercepted by the intercept handler. Active processes ignore this signal. A program can receive a wake-up signal safely without an intercept handler. The wake-up signal will not be queued if the process' signals are masked.
2	Keyboard abort signal. When control-E is typed, this signal is sent to the last process to do I/O on the terminal. Usually, the intercept routine will perform exit(2) upon receiving a keyboard abort signal.
3	Keyboard interrupt signal. When control-C is typed, this signal is sent to the last process to do I/O on the terminal. Usually, the intercept routine will perform exit(3) upon receiving a keyboard interrupt signal.
4	Hang-up signal. This signal is sent by SCF when it discovers that the modem connection has been lost.
2-31	This signal is deadly to serial and pipe I/O system calls.
32-255	These signal numbers are reserved for future use by Microware.
256-65535	User-defined signals. These signal numbers are available for use in user applications.

A signal routine could be designed to interpret the signal code word as data. For example, various signal codes could be sent to indicate different stages in a process' execution. This is extremely effective because signals are processed immediately upon receipt.

The following system calls and C bindings enable processes to communicate through signals:

System Call	C Binding	Description
F$Send	kill(pid, signal)	Sends a signal to a process.
F$Icpt	intercept(icpthand)	Installs a signal intercept routine.

F$Sleep	sleep(seconds)
	tsleep(ticks)
		Deactivates the calling process until the specified number of seconds has passed or a signal is received.

F$SigMask	sigmask(level)	Enables/disables signals from reaching the calling process.

For specific information about these system calls, refer to ***OS-9 System Calls*** and the ***OS-9/68000 C Compiler User's Library***.

Setstat

System Call	C Binding	Description
SS_SSig	_ss_ssig(path,sigcode)	Sends a signal when data is available on the path.
SS_Relea	_ss_rel(path)	Releases the device from a previous SS_SSig call.

Signals: Example Program

The following program demonstrates a subroutine that reads a \n terminated string from a terminal with a 10 second time-out between the characters. This program is designed to illustrate signal usage; it does not contain any error checking.

The _ss_ssig(path, value) library call notifies the operating system to send the calling process a signal with signal code value when data is available on path. If data is already pending, a signal will be sent immediately. Otherwise, control is returned to the calling program and the signal is sent when data arrives.

```
#include <stdio.h>
#include <errno.h>

#define TRUE 1
#define FALSE 0

#define GOT_CHAR 2001
short dataready;      /* flag to show that signal was received */

/* sighand - signal handling routine for this process */
sighand(signal)
register int signal;
{
     switch(signal) {
            /* ^E or ^C? */
            case 2:
            case 3:
                 _errmsg(0,"termination signal received\n");
                 exit(signal);
           /* Signal we're looking for? */
           case GOT_CHAR:
                 dataready = TRUE;
                 break;
           /* Anything else? */
           default:
                 _errmsg(0,"unknown signal received ==> %d\n",signal);
                 exit(1);
     }
}

main()
{
     char buffer[256];               /* buffer for typed-in string */

     intercept(sighand);             /* set up signal handler */

     printf("Enter a string:\n"); /* prompt user */

     /* call timed_read, returns TRUE if no timeout, -1 if timeout */
     if (timed_read(buffer) == TRUE)
          printf("Entered string = %s\n",buffer);
```

```
    else
        printf("\nType faster next time!\n");
}
int timed_read(buffer)
register char *buffer;
{

    char c = '\0';              /* one character buffer for read */
    short timeout = FALSE;      /* flag to note timeout on read */
    int pos = 0;                /* position holder in buffer */
    /* loop until <return> entered or timeout occurs */
    while ( (c != '\n')  &&  (timeout == FALSE) ) {
        sigmask(1);               /* mask signals for signal setup */
        _ss_ssig(0,GOT_CHAR);   /* set up to have signal sent */
        sleep(10);              /* sleep for 10 sec or until signal */

/* NOTE: It is necessary to mask signals before executing _ss_ssig()
 to ensure that the signal does not arrive between the time when the
 _ss_ssig() is executed and the process actually goes to sleep. */

        /* Now we're awake, determine what happened */
        if (!dataready)
            timeout = TRUE;
        else {
            read(0,&c,1);           /* read the ready byte */
            buffer[pos] = c;        /* put it in the buffer */
            pos++;                  /* move our position holder */
            dataready = FALSE;      /* mark data as read */
        }
    }
    /* loop has terminated, figure out why */
    if (timeout)
        return -1;              /* there was a timeout so return -1 */
    else {
        buffer[pos] = '\0';     /* null terminate the string */
        return TRUE;
    }
}

/* OS-9 2.2 needs the following section in line assembly.
   This section is included in the OS-9 2.3 (or higher) library.    */

#asm
* C binding for sigmask(value)
sigmask: move.l d1,-(sp)    save d1 on the stack
 move.l d0,d1               get the passed parameter in the right place
 clr.l d0                   make d0 = 0
 os9 F$SigMask              make the system call to mask signals
 bcc.s ret                  if no error...
 move.l #-1,d0              return -1 to user
 move.l d1,errno(a6)        fill errno with error number
ret move.l (sp)+,d1         restore d1 from the stack
 rts                        return to user
#endasm
```

Alarms

OS-9 alarms enable programs to send signals or execute subroutines at specified times or at specified intervals.

User-State Alarms

The user-state F$Alarm request allows a program to arrange for a signal to be sent to itself. The signal may be sent at a specific time of day or after a specified interval has passed. The program may also request that the signal be sent periodically, each time the specified interval has passed.

The following user-state alarm functions are supported:

System Call	C Binding
A$Delete	alm_delete(alarmid) Removes a pending alarm request.
A$Set	alm_set(sigcode, time) Sends a signal after specified time interval.
A$Cycle	alm_cycle(sigcode, timeinterval) Sends a signal at specified time intervals.
A$AtDate	alm_atdate(sigcode, time, date) Sends a signal at Gregorian date/time.
A$AtJul	alm_atjul(sigcode, time, date) Sends a signal at Julian date/time.

Cyclic Alarms

A cyclic alarm is most useful for providing a time base within a program. This greatly simplifies the synchronization of certain time-dependent tasks. For example, a real-time game or simulation might allow 15 seconds for each move. A cyclic alarm signal could be used to determine when to update the game board.

The advantages of using cyclic alarms are more apparent when multiple time bases are required. For example, suppose that an OS-9 process is being used to update the real-time display of a car's digital dashboard. The process might want to:

- update a digital clock display every second
- update the car's speed display five times per second
- update the oil temperature/pressure display twice per second
- update the inside/outside temperature every two seconds
- calculate miles to empty every five seconds

Each function the process must monitor could be given a cyclic alarm, whose period is the desired refresh rate, and whose signal code identifies the particular display function. The signal handling routine might read an appropriate sensor and directly update the dashboard display. All of the timing details would then be handled by the operating system.

Time of Day Alarms

An alarm may be set to provide a signal at a specific time and date. This provides a convenient mechanism for implementing a "cron" type of utility, which executes programs at specific days and times. Another use would be to generate a traditional alarm clock buzzer for personal reminders.

A key feature of this type of alarm is that it is sensitive to changes made to the system time. For example, assume the current time is 4:00 and a program wants to send itself a signal at 5:00. The program could either set an alarm to occur at 5:00 or set the alarm to go off in one hour. Assume the system administrator discovers that the system clock is 30 minutes slow and resets the clock to the correct time. In the first case, the program would wake up at 5:00; in the second case, the program would wake up at 5:30.

Relative Alarms

A relative alarm can be used to set a time limit for a specific action. Relative time alarms are frequently used to cause an I$Read request to abort if it is not satisfied within a maximum time. This can be accomplished by sending a keyboard abort signal at the maximum allowable time, and then issuing the I$Read request. If the alarm arrives before the input has been received, the I$Read request returns with an error. Otherwise, the alarm should be cancelled. The example program deton.c demonstrates this technique.

System-State Alarms

A system-state counterpart exists for each of the user-state alarm functions. However, the system-state version is considerably more powerful than its user-state equivalent. When a user-state alarm expires, the kernel sends a signal to the requesting process. When a system-state alarm expires, the kernel executes the system-state subroutine specified by the requesting process at a very high priority.

The following system-state alarm functions are supported:

System Call	Description
A$Delete	Removes a pending alarm request.
A$Set	Executes a subroutine after a specified time interval.
A$Cycle	Executes a subroutine at specified time intervals.
A$AtDate	Executes a subroutine at a Gregorian date/time.
A$AtJul	Executes a subroutine at Julian date/time.

Currently, there are no C bindings for the system state calls.

NOTE: The alarm is executed by the kernel's process, not by the original requester's process. During execution, the user number of the system process is temporarily changed to the original requester. The stack pointer (a7) passed to the alarm subroutine is within the system process descriptor, and contains about 1K of free space.

The kernel automatically deletes a process' pending alarm requests when the process terminates. In some cases, this may be undesirable. For example, assume an alarm is scheduled to shut off a disk drive motor if the disk has not been accessed for 30 seconds. The alarm request will be made in the disk device driver on behalf of the I/O process. This alarm will not work if it is removed when the process exits.

One way to arrange for the alarm to have persistence is to execute the F$Alarm request on behalf of the system process, rather than the current I/O process. This may be accomplished by moving the system variable D_SysPrc to D_Proc, executing the alarm request, and restoring D_Proc. For example:

```
move.l D_Proc(a6),-(a7)              save current process pointer
movea.l D_SysPrc(a6),D_Proc(a6)      impersonate system process
OS9 F$Alarm                           execute the alarm request
/*  (error handling omitted)  */
move.l (a7)+,D_Proc(a6)                  restore current process
```

WARNING: If this technique is used, it is essential to ensure that the module containing the alarm subroutine remains in memory until after the alarm has expired.

An alarm subroutine must not perform any function that could result in any kind of sleeping or queuing. This includes F$Sleep, F$Wait, F$Load, F$Event(wait), F$IOQU, and F$Fork (if it might require F$Load). Other than these functions, the alarm subroutine may perform any task.

One possible use of the system-state alarm function might be to poll a positioning device, such as a mouse or light pen, every few system ticks. It is recommended to be conservative when scheduling alarms and to make the cycle as large as reasonably possible. Otherwise, a great deal of the system's available CPU time could be wasted.

The program listed in the following pages demonstrates how alarms may be used.

Alarms: Example Program

The following example program can be compiled with this command:

```
$ cc deton.c
```

```
/*---------------------------------------------------------------|
|     Psect Name:deton.c                                          |
|     Function: demonstrate alarm to time out user input          |
|---------------------------------------------------------------*/
@_sysedit: equ 1

#include <stdio.h>
#include <errno.h>

#define TIME(secs) ((secs << 8) | 0x80000000)
#define PASSWORD "Ripley"

/*---------------------------------------------------------------*/
sighand(sigcode)
{
          /* just ignore the signal */
}

/*---------------------------------------------------------------*/
main(argc,argv)
int     argc;
char    **argv;
{
    register int    secs = 0;
    register int    alarm_id;
    register char   *p;
    register char   name[80];

    intercept(sighand);
    while (--argc)
        if (*(p = *(++argv)) == '-') {
            if (*(++p) == '?')
                printuse();
            else exit(_errmsg(1, "error: unknown option - '%c'\n", *p));
        } else if (secs == 0)
                secs = atoi(p);
            else exit(_errmsg(1, "unknown arg - \"%s\"\n", p));

    secs = secs ? secs : 3;
    printf("You have %d seconds to terminate self-destruct...\n", secs);


    /* set alarm to time out user input */
    if ((alarm_id = alm_set(2, TIME(secs))) == -1)  /* sigcode must be 2 or 3 */
        exit(_errmsg(errno, "can't set alarm - ")); /* for OS-9 2.3          */

    if (gets(name) != 0)
        alm_delete(alarm_id);     /* remove the alarm; it didn't expire */
    else printf("\n");
    if (_cmpnam(name, PASSWORD, 6) == 0)
        printf("Have a nice day, %s.\n", PASSWORD);
    else printf("ka BOOM\n");

    exit(0);
}

/*-----------------------------------------------------------*/
/* printuse() - print help text to standard error           */
printuse()
{
    fprintf(stderr, "syntax: %s [seconds]\n", _prgname());
    fprintf(stderr, "function: demonstrate use of alarm to time out I/O\n");
    fprintf(stderr, "options: none\n");
    exit(0);
}
```

Events

OS-9 ***events*** are multiple-value semaphores. They are used to synchronize concurrent processes which are accessing shared resources such as files, data modules, and CPU time. For example, if two processes need to communicate with each other through a common data module, it may be necessary to synchronize the processes so that only one updates the data module at a time.

Events do not transmit any information, although processes using the event system may obtain information about the event, and use it as something other than a signaling mechanism.

An OS-9 event is a 32-byte system global variable maintained by the system. Each event contains the following fields, in this order:

Event ID — This number and the event's array position are used to create a unique ID. (2 bytes)

Event Name — This name must be unique and cannot exceed 11 characters. (12 bytes)

Event Value — This four-byte integer value has a range of 2 billion. (4 bytes)

Wait Increment — This value is added to the event value when a process waits for the event. This value is set when the event is created and does not change. (2 bytes)

Signal Increment — This value is added to the event value when the event is signaled. This value is set when the event is created and does not change. (2 bytes)

Link Count — This is the event use count. (2 bytes)

Next Event — This is a pointer to the next process in the event queue. An event queue is circular and includes all processes waiting for the event. Each time the event is signaled, this queue is searched. (4 bytes)

Previous Event — This is a pointer to the previous process in the event queue. (4 bytes)

If a Microware C compiler is installed on the system, the events.h file is included in the DEFS directory. This file lists the exact structure of an event.

The OS-9 event system provides the ability to create and delete events, to permit processes to link/unlink events and obtain event information, to suspend operation until an event occurs, and to implement various means of signaling.

Events may be used directly as service requests in assembly language programs. The Microware C compiler supports a corresponding C call for each event system call.

Wait & Signal Operations

The two most common operations performed on events are Wait and Signal. The Wait operation suspends the process until the event is within a specified range, adds the wait increment to the current event value, and returns control to the process just after the wait operation was called. The Signal operation adds the signal increment to the current event value, checks for other processes to awaken (depending on the all flag), and returns control to the process. These operations allow a process to suspend itself while waiting for an event and to reactivate when another process signals that the event has occurred.

For example, events can synchronize the use of a printer. You could initialize the event value to one, the number of printers on the system. You could set the signal increment to one and the wait increment to minus one (-1). When a process wants to use the printer, it checks to see if one is available. That is, it waits for the event value to be in the range (1, number of printers). In this example, the number of printers is one.

An event value within the specified range indicates that the printer is available; the printer is immediately marked as busy (the event value is incremented by -1, the wait increment) and the process is allowed to use it. An event value out of range indicates that the printer is busy and the process is put to sleep on the event queue.

When a process is finished with the printer, the process signals the event by applying the signal increment to the event value. Then, the event queue is searched for a process whose event value range includes the current event value. If such a process is found, the process is activated, the wait increment is applied to the event value, and the printer is used.

To coordinate sharing a non-sharable resource, user programs must:

① Wait for the resource to become available.

② Mark the resource as busy.

③ Use the resource.

④ Signal that the resource is no longer busy.

It is critical that the first **two steps** in this process are **indivisible**, because of time-slicing. Otherwise, two processes could check an event and find it free. Then, both processes would try to mark it busy. This would correspond to two processes using a printer at the same time. The F$Event service request prevents this from happening by performing both steps in the Wait operation.

F$Event System Call

The F$Event system call provides the mechanism to create named "events" for this type of application. The name "event" was chosen instead of "semaphore" because F$Event provides the flexibility to synchronize processes in a variety of ways not usually found in semaphore primitives. OS-9's event routines are extremely efficient and suitable for use in real-time control applications.

Event variables require several maintenance functions as well as the Signal and Wait operations. To keep the number of required system calls to a minimum, all event operations are accessible through the F$Event system call.

Currently, functions exist to allow events to be created, deleted, linked, unlinked and examined. Several variations of the Signal and Wait operations are also provided.

Specific parameters and functions of each event operation are discussed under F$Event in the ***OS-9 System Calls*** section of the ***OS-9 Technical Manual***. The "Ev$" function names are defined in the system definition file funcs.a. Actual values for the function codes are resolved by linking with the relocatable library sys.l or usr.l.

The following event functions are supported:

System Call	***C Binding***
Ev$Creat	_ev_creat(ev_value, wait_inc, signal_inc, ev_name) Creates new event.
Ev$Delet	_ev_delete(ev_name) Deletes existing event.
Ev$Info	_ev_info(ev_index, ev_buffer) Returns event information.
Ev$Link	_ev_link(ev_name) Links to existing event by name.
Ev$Pulse	_ev_pulse(ev_id, ev_value, allflag) Temporarily changes an event value and checks the event queue, then changes back to the original value.
Ev$Read	_ev_read(ev_id) Reads the event value without waiting.
Ev$Set	_ev_set(ev_id, ev_value, allflag) Sets the event variable and checks the event queue.

System Call	*C Binding*
Ev$SetR	_ev_setr(ev_id, ev_value, allflag) Sets relative event variable and checks the event queue.
Ev$Signal	_ev_signal(ev_id, allflag) Increments and event value by the signal increment and checks the event queue.
Ev$UnLnk	_ev_unlink(ev_id) Unlinks event.
Ev$Wait	_ev_wait(ev_id, ev_min, ev_max) Waits for event to occur.
Ev$WaitR	_ev_waitr(ev_id, ev_min, ev_max) Waits for relative to occur.

Events: Example Program 1

The following program uses a binary semaphore to illustrate the use of events. To execute this example:

- Type the code into a file called sema1.c.
- Copy sema1.c to sema2.c.
- Compile both programs.
- Run both programs with this command: sema1 & sema2.

The program creates an event with an initial value of 1 (free), a wait increment of -1, and a signal increment of 1. Then, the program enters a loop which waits on the event, prints a message, sleeps, and signals the event. After ten times through the loop, the program unlinks itself from the event and deletes the event from the system.

```
#include <stdio.h>
#include <events.h>
#include <errno.h>

char *ev_name = "semaevent";     /* name of event to be used */
int ev_id;     /* id that will be used to access event */

main()
{
     int count = 0;  /* loop counter */

     /* create or link to the event */
     if ((ev_id = _ev_creat(1,-1,1,ev_name)) == -1)
          if ((ev_id = _ev_link(ev_name)) == -1)
               exit(_errmsg(errno,"error getting access to event - "));
     while (count++ < 10) {
          /* wait on the event */
          if (_ev_wait(ev_id, 1, 1) == -1)
               exit(_errmsg(errno,"error waiting on the event - "));

          _errmsg(0,"entering \"critical section\"\n");

          /* simulate doing something useful */
          sleep(2);

          _errmsg(0,"exiting \"critical section\"\n");

          /* signal event (leaving critical section) */
         if (_ev_signal(ev_id, 0) == -1)
               exit(_errmsg(errno,"error signalling the event - "));

          /* simulate doing something other than critical section */
          sleep(1);
     }
     /* unlink from event */
     if (_ev_unlink(ev_id) == -1)
          exit(_errmsg(errno,"error unlinking from event - "));

     /* delete event from system if this was the last process to unlink
        from it */

     if (_ev_delete(ev_name) == -1 && errno != E_EVBUSY)
          exit(_errmsg(errno,"error deleting event from system - "));

     _errmsg(0,"terminating normally\n");
}
```

Events: Example 2

The following example uses events to synchronize two processes (task1 and task2), thereby regulating access to CPU time. Task1 executes test #1, and task2 executes test #2. To execute this example:

- Type the first program into a file called task1.c.
- Type the second program into a file called task2.c.
- Compile both programs.
- Run both programs with this command: task1 & task2.

This example executes as follows:

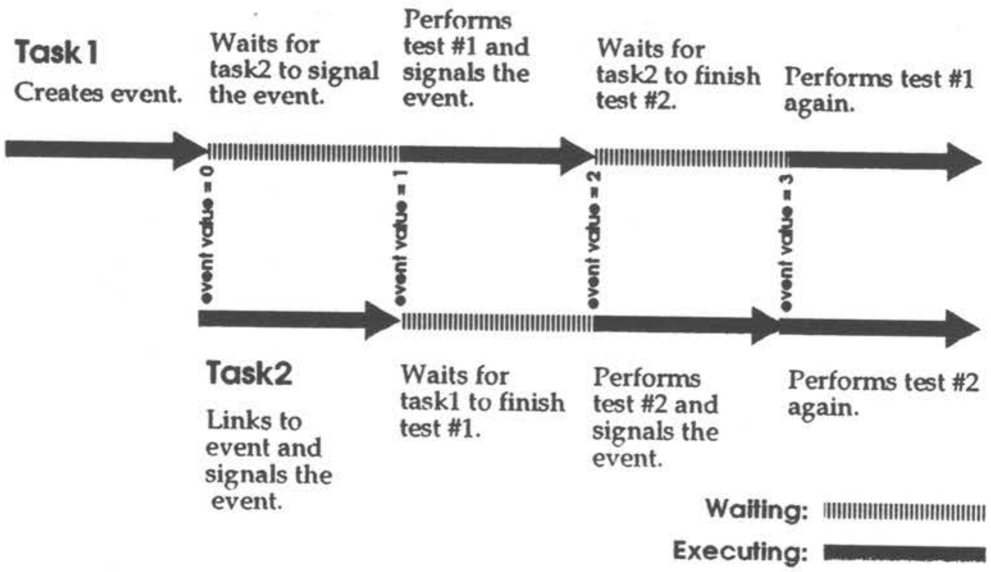
① Task1 creates an event with an initial value of 0, a wait increment of 0, and a signal increment of 1, and waits for the event value to become 1. **NOTE:** The signal increment is added to the event value each time the event is signalled.

② While task1 is waiting, task2 links to the event, signals the event (event value = 1), and waits for the event value to become 2.

③ While task2 is waiting, task1 executes test #1, signals the event (event value = 2), and waits for the event value to become 3.

④ While task1 is waiting, task2 performs test #2 and signals the event (event value = 3).

⑤ Task1 and task2 perform test #1 and test #2 concurrently.



Task1
Creates event.
Waits for task2 to signal the event.
Performs test #1 and signals the event.
Waits for task2 to finish test #2.
Performs test #1 again.
event value = 0
event value = 1
event value = 2
event value = 3
Task2
Links to event and signals the event.
Waits for task1 to finish test #1.
Performs test #2 and signals the event.
Performs test #2 again.
Waiting:
Executing:

Process Execution in Events Example 2

task1.c

```
/* task1.c */

#include <stdio.h>
#include <events.h>
#include <errno.h>

char *ev_name = "exmplevent";           /* event used by this program */
int ev_id;                              /* id for event */

main()
{
        /* create or link to the event */
        if ((ev_id = _ev_creat(0,0,1,ev_name)) == -1)
                if ((ev_id = _ev_link(ev_name)) == -1)
                        exit(_errmsg(errno,"error getting access to event - "));
        /* wait on the event (waiting from "task2" to signal event) */
        if (_ev_wait(ev_id, 1, 1) == -1)
                exit(_errmsg(errno,"error waiting on the event - "));
        _errmsg(0,"doing \"test #1\"\n");

        /* simulate doing a test */
        sleep(2);

        _errmsg(0,"finished with \"test #1\", signalling event\n");

        /* signal event, finished with test #1 */
        if (_ev_signal(ev_id, 0) == -1)
                exit(_errmsg(errno,"error signalling the event - "));

        /* wait for event, waiting for task2 to finish test #2 */
        if (_ev_wait(ev_id, 3, 3) == -1)
                exit(_errmsg(errno,"error waiting on the event - "));

        _errmsg(0,"doing \"test #1\" again\n");

        /* simulate doing test #1 */
        sleep(2);

        _errmsg(0,"finished with \"test #1\" again\n");

        _errmsg(0,"unlinking/deleting the event\n");

        /* unlink from the event */
        if (_ev_unlink(ev_id) == -1)
                exit(_errmsg(errno,"error unlinking from event - "));

        /* delete the event if necessary */
        if (_ev_delete(ev_name) == -1 && errno != E_EVBUSY)
                exit(_errmsg(errno,"error deleting event from system - "));

        _errmsg(0,"terminating normally\n");
}
```

task2.c

```
/* task2.c */

#include <stdio.h>
#include <events.h>
#include <errno.h>

char *ev_name = "exmplevent";           /* event name */
int ev_id;                              /* id used to access event */

main()
{
        /* create or link to the event */
        if ((ev_id = _ev_creat(0,0,1,ev_name)) == -1)
                if ((ev_id = _ev_link(ev_name)) == -1)
                        exit(_errmsg(errno,"error getting access to event - "));

        /* signal the event, tell task 1 it is okay to start */
        if (_ev_signal(ev_id, 0) == -1)
                exit(_errmsg(errno,"error signalling the event - "));

        /* wait on the event, wait for task1 to finish test #1 */
        if (_ev_wait(ev_id, 2, 2) == -1)
                exit(_errmsg(errno,"error waiting on the event - "));

        _errmsg(0,"doing \"test #2\"\n");

        /* simulate doing test #2 */
        sleep(2);

        _errmsg(0,"finished with \"test #2\", signalling event\n");

        /* signal event, tells task1 to do test #1 again */
        if (_ev_signal(ev_id, 0) == -1)
                exit(_errmsg(errno,"error signalling the event - "));

        _errmsg(0,"doing \"test #2\" again\n");

        /* simulate doing test #2 */
        sleep(2);

        _errmsg(0,"finished with \"test #2\" again\n");

        _errmsg(0,"unlinking/deleting the event\n");

        /* unlink from event */
        if (_ev_unlink(ev_id) == -1)
                exit(_errmsg(errno,"error unlinking from event - "));

        /* delete event if necessary */
        if (_ev_delete(ev_name) == -1 && errno != E_EVBUSY)
                exit(_errmsg(errno,"error deleting event from system - "));

        _errmsg(0,"terminating normally\n");
}
```