CHAPTER 8

INTER-PROCESS COMMUNICATION

8.1 WHY USE MULTI-TASKING?

Every real time application will have at least two devices to deal with, as the minimum function will be to take in data in some form, process the data, and output the results. Most applications will have more than two devices, perhaps many more. In addition, the application may have some non-real-time devices to deal with, such as an operator keyboard and graphics display.

The direct approach to such an application is to poll each device in turn. If the device needs servicing an appropriate function is called, and then polling continues. However, this approach causes serious problems in most real time applications. The handler function for one device may take some time to execute (for example, the update of a graphical display), so that the real time response of another device is not met.

Again, the direct approach would be polling. A handler function that takes some time to execute can frequently poll the devices that need rapid response, call the appropriate handler if a device needs servicing, and then continue its own function. This produces an increasingly complex program that wastes a great deal of processor time in polling. The tortuous complexity of the program makes maintenance, improvement, and customization of the application very difficult. The processor time wasted in polling forces the use of a much faster processor, or may make the application impossible.

The solution is to use interrupts from external devices rather than polling, and to have a separate program handling each device. The program can sleep, waiting for its device to generate an interrupt, then handle the

interrupt and go back to sleep. Because the operating system will automatically share the processor time between the currently active processes (and give no processor time to the sleeping processes), a program does not have to worry that the time it is taking to service its device might cause an unacceptable delay in servicing another device.

One program - the parent - is called by the operator, or started automatically by the system on startup. It has the job of forking the other programs - the children - that make up the application. Of course, any of the programs may themselves fork other programs. Such secondary programs may execute for as long as the application continues, as the original parent and children normally will, or may be called for a transitory purpose.

But by splitting the application into separate programs, executing as separate processes, another problem has been introduced. A process must be able to exchange data with other processes, and must be able to activate a sleeping process when it has data ready for that process. This is the purpose of inter-process comunication.

8.2 WHAT IS INTER-PROCESS COMMUNICATION?

Almost all real time applications require the use of multiple concurrently executing processes. Multiple processes executing together to produce a combined result need mechanisms to:

- Pass data between processes.
- Synchronize one or more processes with each other.

Frequently the two needs are combined - a process requires data from another process, and must be made to wait until the data is available. In addition to processes communicating with each other, interrupt handler functions must be able to communicate with processes. In particular, an interrupt handler must be able to activate (wake up) a sleeping process.

It is also sometimes desirable for a process or interrupt handler to cause a temporary change of flow of control in a program, so that it can handle exceptional circumstances without waiting or polling.

Some inter-process communication is private - that is, between processes who know of each other's existence and wish to communicate directly between themselves. Other communications are public - the sending process is essentially broadcasting, and some or all other processes can receive the communication.

Because an interrupt can occur while a system call is being executed, the operating system must mask all interrupts during system calls that can be called from an interrupt handler (to prevent concurrent use of the same operating system memory structures). To avoid masking interrupts unnecessarily (because it would adversely affect real time response), OS-9 specifies that only certain system calls are allowed in an interrupt handler. Therefore not all of the inter-process communication mechanisms can be used by an interrupt handler communicating with a process.

8.3 OS-9 INTER-PROCESS COMMUNICATION FACILITIES

Multiple inter-process communication mechanisms are required to efficiently service the various circumstances that require data transfer or inter-process synchronization. OS-9 provides several inter-process communication mechanisms. Deciding which to use to solve a particular problem is part of the application design task. The table in figure 6 highlights the important differences between the mechanisms. The columns of the table are headed as follows:

DAT Data can be passed.
SYN The mechanism provides synchronization (it can wake up a sleeping process).
PUB The mechanism is public, as opposed to private.
INT An interrupt handler function can use this mechanism.

Mechanism	DAT	SYN	PUB	INT
Wait for child	Exit status	Yes	No	No
Signal	Signal code	Yes	No	Yes
Event	Event value	Yes	Yes	Yes
Alarm	No	Yes	No	Yes[1]
Unnamed pipe	Yes	Yes	No	No
Named pipe	Yes	Yes	Yes	No
Disk file	Yes	Yes	Yes	No
Data module	Yes	No	Yes	Yes
Shared memory	Yes	No	Yes	Yes
External memory	Yes	No	Yes	Yes

[1]Generated by clock tick interrupt.

• **Figure 6. Inter-process communication mechanisms**

8.4 FORKING A PROCESS

The forking of a child process may not seem like a form of inter-process communication, but in fact it is the most fundamental form of inter-process synchronization and communication. Unless additional processes are forked no inter-process communication can take place (almost by definition!). Forking provides synchronization (the forked process starts when the fork request is made) and communication (the parameter string passed to the child). Lastly, waiting for a child to die is a basic form of synchronization and communication (by the exit status of the child).

This form of inter-process communication is used every time a command line is entered through **shell**. **shell** forks the requested program, and then waits for it (or any other of its child processes) to die. When it dies, **shell** reports its exit status to the user if it is not zero. **shell** provides variations on the use of this basic mechanism: concurrently executing processes ('&' and '!'), implicit forking of another shell (parentheses), and waiting for one or all children to die ('w' and 'wait').

A process is forked using the **F$Fork** system call. The parent process specifies the name of the program module, or the name of the file from which the program module is to be read (relative to the parent process's current execution directory). The parent also passes a pointer to a parameter string and the length of the string, the number of paths the child should inherit from the parent (usually three), and (optionally) an additional static storage size and process priority. The kernel adds the parameter string length to the data storage size specified in the program module header, the stack size in the program module header, and the additional static storage specified in the fork request, to make the total static storage size to allocate for the child. The kernel copies the parameter string to the top of the child process's static storage.

In effect, the parameter string is a message passed from the parent to the child process. The parameter string can be any byte string, although an ASCII text string or sequence of strings is usually used. The Microware C library functions **os9fork()** and **os9forkc()** allow the **F$Fork** system call to be made from C. The difference is that **os9forkc()** allows the programmer to specify the number of paths to inherit, while **os9fork()** implicitly asks for three paths to be inherited. However, the library also provides a higher level function - **os9exec()** - that converts an array of text strings together with the environment parameter strings of the parent into a single parameter string compatible with **F$Fork**. The 'cstart' function, which is the startup

function of a compiled C program, converts this string back into an argument array and environment parameter array, as expected by a C program.

The forked process is immediately put in the active queue, and so will be allocated processor time in its turn. Therefore the parent cannot assume that it can execute further instructions between the fork request and the child's first slice of processor time. It is quite possible that the parent process will finish its time slice during the fork system call, and that the new child process will start execution immediately the call is finished.

Below is an example forking of a child program, specifying three paths (0, 1, and 2) to be inherited and no additional static storage memory. The process priority value of zero specifies that the child process should have the same process priority as the parent. Note that by convention (for C programs) the first argument string is the name of the program, although this has no effect on the **F$Fork** system call or the **os9exec()** library function.

```
char *args[]={
    "tmode",        /* program name */
    "nopause",      /* argument strings */
    "noecho",
    NULL            /* NULL pointer terminates the list */
};
int child_pid,      /* child process ID */
    dead_id,        /* process ID of dead child */
    status;         /* child exit status */

int os9fork();
extern char **environ;  /* environment parameter array */

/* Fork the child (could be one of many): */
void fork_child()
{
    child_pid=os9exec(os9fork,args[0],args,environ,0,0);
    printf("Forked process ID %d\n",child_id);
    /* Wait for child (could be any child) to die: */
    dead_id=wait(&status);
    printf("Process ID %d died with exit status %d\n",dead_id,status);
}
```

The **F$Wait** system call, called by the **wait()** C library function, waits for any process that is a child of this process to die. The system call returns the process ID of the dead child, and its exit status code. A non-zero exit status is usually considered an error, but as this is only interpreted by the parent it can be used instead to return a result value. Note that the parent cannot specify which child it is waiting for, and that the parent can be woken instead by a signal, in which case the reported child ID and exit status are zero. Therefore if the process has forked more than one child it will need to

check which child has died, and if it has installed a signal intercept handler (see the section on Signals) it will need to check whether it was woken by the death of a child or by a signal.

A process cannot "miss" the death of a child, even if it is not waiting when the child dies, or more than one child dies simultaneously. Although all of the resources of the child are de-allocated by the kernel when the child dies, the child's process descriptor is retained until the parent process executes an **F$Wait** system call and is returned the exit status of the child, or the parent process itself has died. This guarantees that this form of inter-process communication cannot fail.

The **F$Fork** system call can be used from system state. For example, a device driver can fork a process. In this case the parent of the new process is the current process, that is, the process that made the system call as part of which the child was forked. As the device may later be used by other processes it may be desirable to disinherit the process (make it an orphan) by cutting the links to the parent process. This is described in the chapter on Multi-tasking.

8.5 SIGNALS

An OS-9 signal is a small transitory message sent from one process to another. It can be viewed as the software equivalent of a one word telephone call - it wakes you up if you were asleep, interrupts what you were doing, and gives you a small piece of information as to the reason for the call. It is very important to distinguish between signals and interrupts (a common source of confusion). A signal is purely a software mechanism, while an interrupt is a hard-wired response of the microprocessor to an external electrical signal. The confusion arises because, under OS-9, both cause an asynchronous change of flow of control of software, and both can be "masked". However, there the similarity ends, and the two mechanisms are completely separate.

The **F$Send** system call is used to send a signal from one process to another, or to all processes of that user (a broadcast signal). It requires knowledge of the process ID of the destination process, unless the signal is broadcast. The sending process must have the same user ID and group number as the receiving process, or be a super user (group zero). Sending a signal has two important effects on the destination process:

a) The process is made active if it is not already active, and so will become the current process at some time in the future (it may already be the current process).

b) The process's signal handler function is called when the process next runs in user state.

It is important to bear in mind that at the moment the signal is sent – either from another process or an interrupt handler – the destination process cannot be executing in user state, even if it is active, and even if it is the current process. Even if the process sends a signal to itself, the sending of the signal is carried out in system state by the **F$Send** system call. Therefore once the signal has been sent, the process's signal handler function will be the next part of the program to execute (after any system call the program is making finishes), even if the program was active and only part of the way through a subroutine.

Under OS-9, the signal mechanism is the only inter-process communication mechanism that can cause such an asynchronous change of flow of control. Also, signals are the only way of waiting for multiple sources of synchronization without polling, because a signal forces a process to become active. Once the signal handler function finishes, execution of the program continues as before. If the process was sleeping or waiting, execution continues with the instruction following the "sleep" or "wait" system call.

If the destination process was executing a system call, the signal handler function of the process is not called until the system call finishes (the process returns to executing in user state). If the process was sleeping within a system call (for example, in a device driver, waiting for an interrupt), the process is woken and continues execution after the sleep system call. A signal makes a process active irrespective of its previous state (and is the only way of waking a process from an untimed sleep). Therefore a signal will wake a process that is, for example, waiting for an event, or waiting for a child to die. So, on return from such "waiting" system calls it is important to check the reason for the return from the system call – did the system call finish for the intended reason, or was a signal received?

When a process is about to continue execution in user state (after completion of a system call or an interrupt) the kernel checks whether a signal is pending for the process. If a signal is pending, the kernel checks whether the process has installed a signal handler function. If so, the kernel builds an additional stack frame on the process's stack so that execution will be diverted to the signal handler function, exactly as if the program had made a

subroutine call to the signal handler. On return from the signal handler the program continues execution as before.

A process may receive multiple signals before becoming the current process. The signals are queued (within the recipient's process descriptor) in the order in which they have been received. The **P$Signal** field of the process descriptor contains the latest signal code received[1], or zero if no signals are pending. A signal handler function should end with an **F$RTE** system call, rather than a "return from subroutine" instruction. This system call does very little, but its exit causes the kernel to check again whether a signal is pending for the process, and call the signal handler function again if so. In this way all pending signals are handled before normal program execution recommences. Note: the "wakeup" signal **S$Wake** and the "kill" signal **S$Kill** are not queued - see below.

A process installs (or cancels) a signal handler function using the **F$Icpt** system call. If a program intends to handle signals, installing the signal handler function should be one of the first instructions in the program. The process will be aborted if it does not have a signal handler function installed when the process is about to execute in user state and a signal is pending (except for the "wakeup" signal **S$Wake** - see below). The "kill" signal **S$Kill** cannot be handled - it always aborts the receiving process.

The signal intercept routine is passed the signal code (in the low word of the **d1** register), and the number of signals queued including the current one (in the low word of the **d0** register). It is not passed the process ID of the sender.

The C library function **intercept()** is used to install a signal handler function in a C program. The signal handler is a normal C function. The **intercept()** function ensures that the **F$RTE** system call is made on exit from the signal handler function. Passing a null pointer (zero) instead of a function address cancels any currently installed signal handler for the process.

The signal codes 2 to 31 are known as the "deadly" signals[2]. Device drivers that are sleeping (waiting for completion of a device operation) will normally abort their operation and return to the caller with an error if woken by such a signal, unless this might be destructive to a filing system. Signals 2 to 32 can be ignored, by setting the corresponding bit in the **P$SigMask** field in the process descriptor (signal 32 is ignored by setting bit zero). This function

---

1 Before OS-9 version 2.4, the **P$Signal** field contained a copy of the first signal in the queue - the oldest pending signal - or zero if no signals were pending.

2 Before OS-9 version 2.4, only signal codes 2 and 3 were considered deadly.

is not available through a standard system call. Ignored signals are not queued, and do not cause the receiving process to become active.

Signal code 0 - **S$Kill** - is the "kill" signal. It is not queued. Instead, the kernel sets the "condemned" flag in the **P$State** field of the process descriptor. When the process is about to restart execution in user state it is unconditionally terminated. The name "kill" can cause confusion to C programmers, as the C library function **kill()** is the C function for sending signals. The C function is called **kill()** for compatibility with the UNIX standard C library, but it is used to send any desired signal code.

Signal code 1 - **S$Wake** - is the "wakeup" signal. It is specifically intended for use by an interrupt handler to wake up a sleeping device driver (or other operating system component). It should not be used to signal a user state program, as its special properties do not guarantee proper inter-process synchronization. This signal is not queued, and so does not cause the recipient's signal handler function to be called.

8.5.1 Masking Signals

In order to guarantee correct inter-process synchronization, avoiding the possibility of a timing "race" condition, it is important to be able to "mask" signals, in the same way that interrupts can be masked. That is, the response to the signal can be held off until the process is ready to respond. Signals - except the "kill" signal - can be masked using the system call **F$SigMask**.

This system call increments, decrements, or clears the signal mask field **P$SigLvl** in the caller's process descriptor. While this field is non-zero signals sent to the process are queued in the recipient's process descriptor (except the "kill" and "wakeup" signals) and force the process into the active queue as normal, but the process's signal handler function is not called until signals are unmasked.

The **F$SigMask** system call takes a single parameter, which must be -1, 0, or 1. If the parameter is zero the field **P$SigLvl** in the process descriptor is cleared (signals are unmasked). If the parameter is one **P$SigLvl** is incremented (unless it is already 255). If the parameter is minus one **P$SigLvl** is decremented (unless is is already zero). This approach allows calls to mask signals to be nested. The kernel masks signals (increments **P$SigLvl**) for the process before calling the process's signal handler function (the signal mask must have been clear for the kernel to decide to call the signal handler), and unmasks signals (decrements **P$SigLvl**) on return from

the signal handler function, as part of the **F$RTE** system call. This ensures that the signal handler function will not be called recursively.

If the signal handler function increments the signal mask, the mask will not return to zero when the kernel decrements it, so any other pending signals are not serviced until the program clears the signal mask. This permits the main body of the program to respond to signals one at a time[3].

Executing the system call **F$Sleep** (timed or indefinite sleep) or **F$Wait** (wait for child to die) from user state clears the **P$SigLvl** signal mask. Therefore the following sequence works correctly as a user state inter-process synchronization mechanism, without the risk of a "race" condition:

1) Mask signals using **F$SigMask**.
2) Check a flag set by the signal handler function indicating that a signal has been received.
3) Sleep using **F$Sleep**.

An example in C is shown below.

```
int got10,                  /* flag - signal 10 received */
    got_20,                 /* flag - signal 20 received */
    invalid;                /* flag - invalid signal received */

sighandler(s)               /* the signal intercept handler */
register int s;             /* signal code received */
{
    switch (s) {            /* record which signal has arrived */
        case 10:
            got_10=TRUE;
            break;
        case 20:
            got_20=TRUE;
            break;
        default:
            invalid=s;
            break;
    }
}
```

3 This facility was not available prior to OS-9 version 2.3 - the **F$RTE** system call called the program's signal handler function again if another signal was pending, without checking the signal mask.

```
main()
{
    while (TRUE) {
        sigmask(1);              /* mask signals */
        if (got_10 || got_20 || invalid) {
            sigmask(0);          /* unmask signals */
            if (got_10)
                printf("Received signal 10\n");
            else if (got_20)
                printf("Received signal 20\n");
            else exit(_errmsg(1,"Invalid signal %d received\n",
                invalid));
        }
        else
            tsleep(0);           /* sleep until woken */
    }
}
```

In the above example, signals are masked before checking whether a signal has already arrived. Once signals are masked the signal handler function will not be called even if a signal arrives, so the flags cannot be set between making the check and going to sleep. Calling the **F$Sleep** system call (via the **tsleep()** or **sleep()** C library functions) unmasks signals[4] and checks for any signal pending. If a signal is pending the process is not suspended – the system call returns immediately. Otherwise the process is put in the sleeping queue. The kernel performs these actions with interrupts masked, so even if the signal is to come from an interrupt handler function these steps are indivisible – a signal cannot arrive between checking for signals and going to sleep.

This check for a pending signal is not just a check of whether the **P$Signal** field (most recently received signal code) of the process descriptor is not zero. If this were done, a device driver, or other operating system component, woken from a sleep by a signal being used for inter-process communication, would not be able to go back to sleep again (waiting for a signal from an interrupt service routine), because the pending signal would cause the **F$Sleep** system call to return immediately. Therefore the kernel sets a flag in the process descriptor (bit 7 of the **P$SigFlg** field) whenever a signal is received when the process is active. User state calls to "sleep", "wait for child", or "wait for event", clear this flag, but calls made in system state do not. Before suspending the process these calls check whether this flag is set. If so, the flag is cleared, but the process is not suspended – the system call returns immediately. In effect, in system state this flag indicates that the process has received a signal since the last "sleep"; "wait for child", or "wait for event" system call. Thus a system state function is only woken once by

4 Unless the system call is made from system state.

each signal, and can go back to sleep even though a signal is pending in the process descriptor.

If the "sleep", "wait for child", or "wait for event" system call is made from user state, the system call clears bit 7 of the **P$SigFlg** field immediately. However, it then checks whether there is a signal pending (the **P$Signal** field of the process descriptor is not zero). If so, it sets the bit. This is detected by the main body of the system call routine (common to calls from system state and user state), which - as described above - returns immediately to the caller. Thus any pending signal causes the process to continue execution. Note, however, that the "wakeup" signal (**S$Wake** - code 1) is not queued, nor is it put in the **P$Signal** field - it just causes bit 7 of the **P$SigFlg** field to be set. Therefore a call to "sleep", "wait for child", or "wait for event" made from user state cannot detect this signal. The process will be suspended even if this signal was received since the last "sleep" call, unless the call is made in system state and there has been no intervening call made from user state. Thus the "wakeup" signal is not suitable for inter-process communication - it is intended only to be used by an interrupt service routine waking up an operating system component.

This somewhat complex distinction between the effect of these calls in user and system state reflects the fact that to the user state program these are inter-process communication mechanisms, but to an operating system component (such as a device driver) they are mechanisms for communication between an interrupt service routine and a process.

As described above, the **F$Wait** system call performs the same operations as the **F$Sleep** system call with regard to pending signals, but note that prior to OS-9 version 2.3 the **F$Wait** system call tested for a pending signal by checking the **P$Signal** field directly, rather than using bit 7 of the **P$SigFlg** field, so it was not suitable for use from system state.

Note that the "wait for event" system call does *not* clear the signal mask, but it does check for a signal pending[5]. If a signal is pending, the system call returns immediately to the caller (in which case the returned event value is the current event value). Note also that if a process waiting for an event receives a signal while it has signals masked (the **P$SigLvl** field is not zero), the process is still forced active, but the signal handler function will not be called - the process continues execution with the instruction following the "wait for event" system call. This means that if a process makes a "wait for event" system call with signals masked, but a signal is pending or is received before the event changes to the desired range, the "wait for event" terminates

5 From OS-9 version 2.3 onwards.

and the process is returned the event value at the time it made the "wait for event" call (which will be outside the desired range), but the program's signal handler function is not called (until signals are unmasked).

Before OS-9 version 2.3 there was no C library function for the **F$SigMask** system call, so one is given in assembly language below:

```
#asm
sigmask:    move.l  d0,d1           copy mask value (0, 1, or -1)
            moveq   #0,d0           d0 must be zero
            os9     F$SigMask       execute the system call
            rts
#endasm
```

8.5.2 Signals - Cautions

This section covers particular details of the operation of signals that are most commonly the source of problems when using signals.

The "wakeup" signal **S$Wake** is not queued. If it is received while the process is in system state it is lost on return to user state, and so is not suitable for inter-process synchronization. It is primarily intended as a mechanism by which an interrupt handler function can wake up a sleeping device driver (or other system state component).

The **F$Sleep** and **F$Wait** system calls return to the calling process immediately if a signal is pending. In system state, a pending signal will only cause a process to wake up once - a further signal must be received to wake the process from a subsequent "sleep" system call made before returning to user state.

The signal intercept handler function is called when a process with a signal pending is about to return to user state (it becomes the current process, or - if it is already the current process - it returns from a system call or an interrupt service routine finishes). Therefore system state processes cannot make use of a signal intercept handler function.

8.6 EVENTS

OS-9 events are another mechanism for inter-process synchronization. OS-9 events are similar to signals in a number of ways:

a) An event is used for inter-process synchronization.

b) An event passes a small amount of data - the event value.

c) Events can be used from interrupt handler functions.

However, events differ from signals in some important respects:

a) An event cannot cause a temporary change of flow of control – there is no equivalent to the signal handler function.

b) A process cannot be woken by an event when waiting for something else – a process can only be woken by an event when waiting for that event to change.

c) Events are public – any number of processes can link to an event, and alter the event value or wait for it to change.

d) Events are more flexible than signals – the event value can be changed in a number of ways, and a process can wait for the event value to change to within a desired range.

e) Events are not transitory – an event exists even when not being changed or waited for, and its current value can be read.

To use an event it must first be created. A system call is used to create an event, giving a name (character string) for the event, and an initial value for the event. The name can be up to 12 characters, and is subject to the same restrictions as module names. Letter case is not significant. The kernel finds a free entry in the event table (checking that no event of the same name already exists), and initializes the entry.

The event is allocated an event ID. This is a long word, of which the high word is the event number from the high word of the **D_EvID** field of the System Globals (after it has been incremented), and the low word is the index (base zero) of the event entry within the event table. Before creating the event ID the kernel increments the high word of the **D_EvID** field. The kernel does not permit the high word of the **D_EvID** field to be zero (if it becomes zero it is set to one), so the event ID cannot be zero. The event table is dynamically extendible, so the number of events in existence at any one time is not limited. The caller creating the event is returned the event ID (or an error if an event of the same name already exists).

The event table entry also contains a link count (initialized to one when the event is created), a "signal increment" (also known as an "automatic increment"), and a "wakeup increment". The event value is a signed long word and the increment values are signed words. The increment values are specified when the event is created. The "signal increment" (nothing to do

with OS-9 signals) is added to the event value when the "signal event" call is made. This is a convenience, being simpler to use than an explicit change to the event value. The "wakeup increment" is added to the event value when a process is woken from waiting on an event because of a change to the event value.

Once an event has been created, other processes can get the event ID by making a "link to event" call, specifying the event name. Each such call increments the event link count, in a similar manner to the link count of an OS-9 module directory entry. Similarly, once a program has finished with an event it makes an "unlink from event" call, which decrements the link count for the event. Once the link count has been reduced to zero the event can be deleted by a "delete event" call, which frees the event table entry. Because an event could be unlinked more than once by the same process, reducing its link count to zero even though other processes are waiting on the event, the kernel will wake up any processes waiting on an event that is being deleted (returning them an "invalid event ID" error - **E$EvntID**). Note that when a process is terminated the kernel does not automatically unlink or delete any events a process may have linked to or created.

A process that has the event ID can make use of the event in three ways:

a) Read the current event value.

b) Change the event value.

c) Wait for the event value to fall within a specified range.

Whenever the event value is changed, the kernel function that makes the change checks whether the new event value falls within the specified range of any process waiting on the event. If so, the kernel wakes up the waiting process, returning it the new event value that caused it to be woken, and then adds the wakeup increment to the event value.

A flag (bit 15 of the event function sub-code) is passed with each call to change the event value. This flag indicates to the kernel if it should wake up all processes waiting on the event for which the new value falls in the process's specified range (group wakeup), or only the first such process in the queue of processes waiting on the event (individual wakeup). For an individual wakeup the kernel wakes up the first process in the queue for which the event value is now in range - this may not be the first process in the queue. Processes are queued on an event in the chronological order in which they made the "wait for event" calls.

Note that the wakeup increment is added to the event value as soon as a process is woken, changing the event value for the check of subsequent processes in the queue during a group wakeup. Also note that processes earlier in the queue are *not* rechecked if the wakeup increment causes a change to the event value. It is therefore possible for a process to remain in the queue even though the event value now falls within its range.

The event value can be changed in four ways:

a) Set a new absolute value.

b) Specify a signed integer to add to the event value – "set relative".

c) "Signal" the event – adds the "signal increment" to the event value.

d) Pulse the event temporarily to an a new absolute value.

Each call to change the event value takes the individual/group wakeup flag (as described above). The calls are returned the event value as it was before the call, in the **d1.l** register[6]. If the "pulse" feature is used, the kernel sets the new event value and checks the queue of waiting processes in the normal way, but restores the event value back to its previous value before returning to the caller.

A process can wait for an event in two ways. Both specify a range – signed minimum and maximum values. One method specifies the range as absolute values. The other specifies the range as values relative to the current event value. The process will be woken when the event value is changed to fall within the specified range, unless the change is made with the "individual wakeup" flag, and the waiting process is not the first in-range process in the queue on the event. Also, due to the sequential nature of the group wakeup (described above), and to the immediate wakeup (described below), if the wakeup increment is not zero it is possible for a process to remain in the event queue even though the event value now falls within the desired range.

If a process attempts to wait for an event that is already within the specified range, the system call returns immediately to the caller. In this case the wakeup increment is applied to the event value as usual, but the kernel does *not* check the event queue to see if the new event value (assuming the wakeup increment is non-zero) is now in range for any waiting process.

---

6 From OS-9 version 2.3 onwards.

A single system call **F$Event** is used for all the event functions. The required function is specified by a sub-code. The C library provides separate C functions for each event function. The following table shows the functions available, with the C function name and the assembly language sub-code name. The sub-codes are defined in the file 'DEFS/funcs.a'.

Sub-code	C function	Description
Ev$Creat	_ev_create	Create an event (must not already exist).
Ev$Delet	_ev_delete	Delete an event (link count must be zero).
Ev$Link	_ev_link	Link to an existing event.
Ev$UnLnk	_ev_unlink	Unlink from an event.
Ev$Wait	_ev_wait	Wait for event - absolute maximum and minimum values.
Ev$WaitR	_ev_waitr	Wait for event - maximum and minimum values relative to the current event value.
Ev$Set	_ev_set	Set new event value.
Ev$SetR	_ev_setr	Add signed quantity to event value.
Ev$Signl	_ev_signal	Add "signal increment" (set when event created) to event value.
Ev$Pulse	_ev_pulse	Momentarily change event value.
Ev$Read	_ev_read	Read current event value.
Ev$Info	_ev_info	Read (next) event table entry structure.

A process waiting on an event may be woken by a signal. This is not considered an error - the process is returned the event value at the time the signal is received, and can detect that it was woken by a signal (rather than by an appropriate event value) in one of two ways:

a) The program's signal handler function sets a flag.

b) The returned event value is not within the requested range (the program is returned the event value that existed when it made the "wait" call).

The "wait for event" function does *not* clear the process's signal mask (**P$SigLvl** field of the process descriptor). If the signal mask is not zero, a pending signal will keep the process active, or a subsequent signal will wake the process, but the process's signal handler function will not be called until the process clears the signal mask.

Note that prior to OS-9 version 2.3 the "wait for event" function did not check whether a signal was already pending - the process would be put to sleep even if a signal was received while the "wait for event" function was

being executed. A process wishing to wait for an event *or* a signal to occur might not have responded to the signal until the event occurred.

8.6.1 Using Events

It may be seen from the above description that events are very flexible, and can be used in many different ways. Indeed, the only problem with using events is in deciding what technique is appropriate to a given situation. This section aims to reduce the possible confusion by describing some typical techniques.

The features available with OS-9 events have been very carefully chosen[7]. A very wide range of simple and complex inter-process synchronization algorithms can be implemented by a simple use of events, if the method of use is carefully chosen. In general, only a very few event statements are needed for even very complex algorithms. Therefore if your implementation of the algorithm in your application appears to require a complex or convoluted use of events it is worth reconsidering your approach.

In choosing a particular technique the principal aim must be secure operation (as with all forms of inter-process communication). That is, there must be no possible condition under which the mechanism will fail and cause the application to lock up or lose data. The features of OS-9 events are designed to give this security, provided they are used correctly. As with all system calls, event functions are indivisible - another process cannot be scheduled in while the call is executing, unless the kernel explicitly goes to sleep (such as during a "wait for event" call). Also, the kernel masks interrupts during critical code fragments, so events can be used from interrupt handlers (but not "wait for event"!).

8.6.2 Pulsing an event

In its simplest form an event can be used in the same way as a signal, except that an event does not cause the asynchronous execution of a handler function, nor can it wake a process that is not waiting on the event. One process creates the event, and another process links to the same event. The initial event value is set to zero, and the wakeup increment is set to zero. The process "receiving" the event waits for the event to reach a value of one (minimum value of range is one, maximum is one). The process "sending" the event pulses the event value to one.

---

[7] Although one or two useful functions are not available, such as an indivisible "set event and wait".

In this simple technique the event is transitory - its value is always zero, except during the "pulse event" system call. In fact, this use of events is not secure - event changes cannot be made pending (equivalent to masking signals), so a process could decide to wait for an event that has already occurred. For this reason the "pulse event" technique should always be used with some form of handshaking, so that the "sending" process does not "send" the event until the "receiving" process is ready.

8.6.3 Interlocked handshake

A common use of events provides an interlocked handshake between two processes. For example, one process may place a data item in a data module, wake up another process by signalling an event, and then wait for the process to take the data. This can be done with complete security by using positive and negative event changes.

The event is created with a value of zero, and signal and wakeup increments of zero. The receiving process waits for the event to have a value of one (minimum is one, maximum is one). The sending process increments the event by one, changing the event value to one, which wakes up the receiving process. The sending process then waits for the event to have a value of zero.

When the receiving process has taken the data, it decrements the event value by one (adds minus one), changing the event value back to zero and waking up the sending process. There is no need for masking (which is not available with events) because the event value persists until changed, and a process attempting to wait on an event whose value is already in the desired range is immediately re-activated. Notice that this mechanism requires only two event statements in each program - a "wait for event" and a "set event relative". The equivalent algorithm implemented with signals would be significantly more complex.

8.6.4 Buffered handshake

The interlocked handshake described above is for the limited case of a single item of data being passed by the handshake. The event statements changing the event value could have explicitly set the event to one and zero instead of incrementing and decrementing it. It is in fact a subset of the more general case of a buffer of several items.

To enable a continuous flow of data, a buffer of two or more items may be used. As in the case of a single item, the buffer could conveniently be in a data module. For a multi-item buffer the sending process should only wait if

the buffer becomes full, and the receiving process should only wait if the buffer becomes empty. The mechanism is the same as for the single buffered case above. The event is created with a value of zero, and wakeup and signal increments of zero.

The sending process waits for the event to be in the range zero to "one less than the buffer size" – it will already be in this range unless the buffer is full. The sending process then adds the new item to the buffer and increments the event. The receiving process waits for the event to be in the range one to "the buffer size" – it will already be in this range unless the buffer is empty. The receiving process then takes the first item out of the buffer, and decrements the event.

Care must be taken in the use of variables for manipulating the buffer. An effective mechanism is to use a circular buffer in a data module, with next-in and next-out indices also in the data module. Only the sending process updates the next-in index, and only the receiving process updates the next-out index.

8.6.5 One to many synchronization

In this example one process writes data to a global pool (for example a data module), and multiple processes read the data. The sending process must not write new data until all receiving processes have read the data. The sending process wakes up all processes that were waiting for the data, and then itself waits until they have all accepted the data. The mechanism described below will work for any number of receiving processes, including zero (which can be useful for test purposes).

The event is created with a signal increment of minus one and a wakeup increment of one. The sending process writes new data to the data module, then sets the event to some large value (greater than the maximum number of waiting processes) – 1000 in this example. This wakes up all the waiting processes – they have been waiting for the event to have a value equal to or greater than 1000. Note that the call to change the event value specifies the **EV_ALLPROCS** group wakeup flag, to wake all waiting process for which the event value is now in range (a flag of zero would be specified for a single-process wakeup).

Because the wakeup increment is one, the event value is now equal to 1000 plus the number of processes that were waiting (and have now been woken). The sending process now subtracts the same large number (1000) from the event value. If the receiving processes have not yet changed the event value,

it will now be equal to the number of processes that were waiting. The sending process waits for the event value to be zero – which will already be true if no processes were waiting.

Each receiving process takes the data, and then decrements the event value – in this example by using the "signal increment" of minus one set when the event was created, for convenience. Once all the receiving processes have taken the data the event value is reduced to zero, and the sending process is woken. This final wakeup changes the event value to one, but this does not affect the mechanism, as the sending process will set it to an absolute value of 1000 when new data is ready.

This algorithm requires that the receiving processes be ready and waiting when new data is available. This is a common requirement where the sending process is gathering data at a fixed rate, and cannot delay if a receiving process is not ready. A check (such as a packet number in the data) would be used by the receiving processes to report an error if data is missed.

The sending process:

```
new_data();                          /* write the new data */
/* Wake up all waiting processes: */
_ev_set(event_id,1000,EV_ALLPROCS); /* value = 1000 */
_ev_setr(event_id,-1000,0);
/* Value now = number-who-were-waiting */
_ev_wait(event_id,0,0);              /* wait until value = 0 */
/* Event value is now one (after our wakeup increment) */
```

The receiving processes:

```
/* Wait until the value is set: */
_ev_wait(event_id,1000,5000);
take_data();                         /* get the new data */
_ev_signal(event_id,0);              /* decrement the value */
```

If the sending process must wait for all the receivers to be ready for the data, a modified form of the "interlocked handshake" described above can be used. The sender must know the number of receivers – stored in the variable **rx_num** in the example below. The event is created with a value of zero, a wakeup increment of zero, and a signal increment of one:

The sending process:

```
_ev_wait(event_id,rx_num,rx_num);    /* wait for all receivers*/
new_data();                          /* write the new data */
/* Wake up the receivers to take the data: */
_ev_set(event_id,0,EV_ALLPROCS);     /* value = 0 */
```

The receiving processes:

```
_ev_signal(event_id,0);          /* increment the event */
_ev_wait(event_id,0,0);          /* wait for the data */
take_data();                     /* get the new data */
```

In effect, this is a many-to-one synchronization, with many receivers indicating their readiness to one sender, followed by a one-to-many synchronization, with the sender broadcasting its readiness to all the receivers.

8.6.6 Rendezvous

Two or more processes may need to know that they are at a common point in their programs, waiting until all processes are at this "rendezvous". In the following example the variable **procs** holds the number of processes that wish to rendezvous. The event is created with a value of zero, a signal increment of one, and a wakeup increment of zero. Each program uses the same code fragment:

```
_ev_signal(event_id,EV_ALLPROCS);   /* increment the event */
_ev_wait(event_id,procs,procs);     /* wait for all processes */
```

After the rendezvous the event value must be reset back to zero by one of the processes:

```
_ev_setr(event_id,-procs,0);        /* reset the event */
```

There exists the possibility that this process, woken as a result of the event value change caused by the last process to come to the rendezvous, will reset the event value before that last process is able to execute its "wait for event" function. This problem exists because the OS-9 events system does not provide a single call to "change the event value and wait". This problem may be ameliorated by giving a low process priority to the process that resets the event value, so it is unlikely to "cut in" to the instruction sequence of the last process to join the rendezvous.

8.6.7 Semaphore

An event can be used to control access to a shared resource, such as a data module. A process wanting to use the resource must be made to wait until the resource is free. A process finishing with the resource must wake up the first process in the queue of processes waiting to use the resource. The event is effectively used as a "lock" or "semaphore" on the resource.

The event is created with a value of zero, a wakeup increment of one, and a signal increment of minus one. A process wanting to use the resource waits

for the event to have a value of zero. The wakeup increment automatically sets the event value to one, locking out any other process wanting to use the resource. When the process has finished with the resource it "signals" the event, setting it to one, using the "single process wakeup" mode so that only the first process in the queue is woken.

```
_ev_wait(event_id,0,0);              /* wait for the resource */
/* The event value is now one: */
use_resource();                      /* make use of the resource */
_ev_signal(event_id,0);              /* unlock the resource */
```

8.7 PIPES

A pipe is a "first-in-first-out" (FIFO) data store managed by the operating system. One or more processes write to the pipe using the standard I/O writing functions, and one or more processes read from the pipe using the standard I/O reading functions. The data is a byte stream - bytes are read from the pipe in the chronological order in which they were written to the pipe. Once one process has read a byte, the byte is lost from the pipe. When as many bytes have been read as were written, the pipe is empty - more bytes must be written before any more can be read. The pipe is of finite size. It becomes full if the number of bytes written exceeds the number of bytes read by the pipe size - no more bytes can be written until some have been read.

Because data is lost once read, and is read in strict chronological order, a pipe normally has only one reading process even if there are multiple writing processes. Multiple reading processes cannot know which process will read which data element, unless some other synchronization mechanism (such as an event) is used.

OS-9 pipes are memory buffers only (they are not held on disk). Pipes are managed by the **pipeman** file manager. Because pipes are held in memory there is no need for a device driver to manage an I/O interface. However, the OS-9 I/O system requires a device driver for every device. Therefore the device driver **null**[8] is needed in memory for pipes to operate, but its functions do nothing. The device descriptor for pipes is **pipe**. Pipes are created by creating a path to the device '/pipe' using the normal I/O path creation functions.

A pipe may be created as a "named" pipe or as an "unnamed" pipe. An unnamed pipe is created if the path is created on the device name alone, by a "create" or "open" system call:

---

8 **piper** prior to OS-9 version 2.3.

```
path=create("/pipe",S_IREAD|S_IWRITE,S_IREAD|S_IWRITE);
path=open("/pipe",S_IREAD|S_IWRITE);
```

A named pipe is created if the path is created with a second name element, using the "create" system call – similar to a single-level disk directory:

```
path=create("/pipe/fred",S_IREAD|S_IWRITE,S_IREAD|S_IWRITE);
```

Once a named pipe has been created it can be opened using the same path name:

```
path=open("/pipe/fred",S_IREAD|S_IWRITE);
```

By default the pipe file manager uses spare room in the path descriptor for the pipe buffer – 90 bytes. However, specifying an "initial file size" when creating the pipe (as would be done for a disk file) causes the file manager to allocate a separate buffer of the requested size, so pipes can be as large as needed. The actual size of the pipe buffer allocated may be greater than the requested size – the request is rounded up to the nearest multiple of the process minimum allocatable block size (16 bytes). Thus:

```
path=create("/pipe",S_IREAD|S_IWRITE|S_ISIZE,S_IREAD|S_IWRITE,1000);
```

will create a pipe buffer of 1008 bytes.

The pipe file manager connects multiple paths open on a named pipe so that they refer to the same memory buffer. In this way multiple processes can read and write the same pipe. The only way for multiple processes to access the same unnamed pipe is for the processes to inherit the path to the pipe, by being forked by a process that already has a path open to the pipe. That is, multiple paths cannot be open to an unnamed pipe – only multiple duplications of the same path permit multiple accesses to the pipe.

Implicit in this basic distinction between named and unnamed pipes are several differences in the details of operation, which are described below. Unnamed pipes are essentially a mechanism for connecting the standard output of one process to the standard input of another without the need for the processes to know that the path is not to a terminal. The features of pipes, and unnamed pipes in particular, reflect this requirement.

Pipes provide for data passing as well as inter-process synchronization. Reading from an empty pipe causes the process to be suspended until another process writes to the pipe, unless no other paths (or duplications of this path) have the pipe open for write, in which case the reader is returned an end-of-file error. This gives automatic synchronization between connected processes, with a proper end-of-file condition.

A process writing to a pipe when there is insufficient room in the pipe for the requested number of bytes is put to sleep until sufficient room becomes

available (because data has been read from the pipe by another process). To prevent a process "hanging up", writing to an unnamed pipe with no other paths (or duplications of this path) having it open for read returns a write error (**E_WRITE**). Thus if the connected reading process dies abnormally (for example, it is killed by the user), the writing process receives an indication of this condition, and can report an error or terminate itself.

This does not apply to named pipes. A named pipe can remain in existence even if there are no paths open to it, provided it contains data. Therefore a process attempting to write to a full named pipe is put to sleep even if no other process currently has the pipe open for reading, in the anticipation that another process will subsequently open a path to the pipe and read the data. A named pipe is automatically deleted if there are no paths open to it and it contains no data. It may also be deleted using the normal "delete" operating system call, provided no paths are open to it:

```
$ del /pipe/fred
```

The pipe file manager also permits the opening of a directory path on the '/pipe' device, allowing the single-level directory of named pipes to be read:

```
$ dir /pipe
```

If a process receives a "deadly" signal while waiting for a pipe operation to complete, **pipeman** will abort the operation, and return the signal code as an error code.

It may be seen that while pipes most resemble an **SCF** device (such as a terminal), named pipes have some of the properties of disk files. However, remember that the data has only a transient existence, and may only be read in the order in which it was written.

The "Get Status" call **SS_Ready** can be used to find out how many bytes of data are waiting in the pipe (just as for an **SCF** device). This call is made by the C function **_gs_rdy()**. Similarly, the "Set Status" call **SS_SSig** requests that the process be sent a signal when data is available in the pipe. This call is made by the C function **_ss_ssig()**.

8.7.1 Using Unnamed Pipes

As described above, because an unnamed pipe cannot be opened by name, only one path can be open on an unnamed pipe - the path created when creating the pipe. Therefore multiple processes can only access the same pipe by means of duplications of the path - that is, a process must inherit the path from its parent. If a process forks multiple children, or a child forks

another process (a "grandchild"), multiple processes can have access to the same pipe.

The **shell** uses unnamed pipes for piping the output of one process to the input of another. For example:

```
$ dir -ud ! grep -v "/$" ! del -z
```

would be used to delete all files in the current data directory, but not attempting to delete sub-directories. The following sequence of operations forks two processes, with the standard output of the first process redirected to a pipe, and the standard input of the second process redirected to the same pipe. The original standard input and output paths of the parent are restored to their original paths. To simplify the example all error handling has been omitted - in practice every function call should always be checked for an error being returned.

```
copy_in=dup(0);          /* duplicate standard input path */
copy_out=dup(1);         /* duplicate standard output path */
close(1);                /* close standard output path */

/* Open the pipe. It is guaranteed to be path 1 (standard output), as
    the kernel uses the lowest available path number: */
path=create("/pipe",S_IREAD|S_IWRITE,S_IREAD|S_IWRITE);

/* Fork the first process, passing three paths: */
pid_1=os9exec(os9fork,"prog1",args1,environ,0,0);

close(0);                /* close standard input path */
/* Duplicate the pipe. The duplicate is guaranteed to be path 0
    (standard input path): */
dup(1);

close(1);                /* close standard output path (the pipe) */
/* Duplicate the duplicate of the original standard output path,
    restoring the original standard output path: */
dup(copy_out);

/* Fork the second process, passing three paths: */
pid_2=os9exec(os9fork,"prog2",args2,environ,0,0);

close(0);                /* close standard input path (the pipe) */
/* Duplicate the duplicate of the original standard input path,
    restoring the original standard input path: */
dup(copy_in);

/* Close the duplicates of the standard paths: */
close(copy_in);
close(copy_out);
```

8.7.2 Using Named Pipes

Named pipes allow a public use of pipes, and remove the requirement for the pipe to be an inherited path. They can also be used in applications where an unnamed pipe cannot be used - for example, a program may take a path name as an explicit parameter, rather than sending output to the standard output path. Within a multi-tasking application a named pipe can be opened only as needed. For example, an error logging process may take input from a named pipe, which it creates and keeps open. Other processes needing to report an error can open the named pipe, write to it, and then close the pipe. To prevent an "end of file" error when attempting to read from the pipe, the error logging process must duplicate the path (**dup()** C library function), so that the local path number used for reading is not the only incarnation of the only path open to the pipe.

Named pipes can help in debugging a multi-tasking application. The programmer can display a directory listing of all named pipes to see how much data is in each pipe (this appears as the "file size" in the directory listing):

```
$ dir /pipe -e
```

The programmer can also insert data into the pipe, simulating information being sent from another process:

```
$ echo "action 2" >+/pipe/commands
```

(note the use of the '>+' redirection to send data to an already existing pipe or file). The programmer can also read the contents of a pipe (but remember that the data is then lost to the application):

```
$ dump /pipe/info
```

Both named and unnamed pipes can be created with an explicit buffer size, overriding the default of 90 bytes. This should be done with care. It is reasonable to use a large buffer if the data structures being passed are large, but it is generally inadvisable to create a buffer that can hold a large number of data structures with the aim of relaxing the response time requirement on the reading process. The reason is that if the reading process cannot keep up with a small buffer, a large buffer will only allow large processing delays and will not prevent the eventual failure of the reading process to respond in time. However, it is reasonable to use a large buffer if the average rate of data is low, but the peak rate can be high - the pipe will absorb the peaks.

One of the potential problems in using pipes in a multi-tasking application is the reading process not responding in time, so the pipe fills up and a writing process is put to sleep. This may be the desired operation, giving

inter-process synchronization, but in many applications it would destroy the real time response of the writing process, which must return to its task of data collection. Of course, if the reading process does not respond in time this may be considered a fatal error, but the writing process must know of the error and be able to report it.

To do this the writing process can use the **_gs_rdy()** function to determine how much data is already in the pipe. Subtracting this from the pipe buffer size gives the free space in the pipe. If this is less than the required amount the process reports the error and does not write to the pipe. Be aware that this sequence is not indivisible - if there are multiple processes writing to the pipe, a process may decide from its check that there is sufficient space to write to the pipe, but before it writes its data another process could write to the pipe and fill it up. (See the chapter on Multi-tasking for techniques on making a sequence of instructions indivisible).

The example below shows a named pipe being created with a defined size. The program checks that space is available in the pipe before writing:

```
#include <modes.h>
#define P_MODE (S_IREAD|S_IWRITE|S_ISIZE)
#define P_PERM (S_IREAD|S_IWRITE)
#define P_SIZE 1000          /* size of pipe buffer */
char *pipe="/pipe/fred";     /* name for named pipe */
char message[80];            /* buffer for data to write to pipe */
main() {
    int path;
    path=create(pipe,P_MODE,P_PERM,P_SIZE); /* create pipe */
    while (1) {
        get_data(message);  /* build message to send */
        if (_gs_rdy(path)>P_SIZE-strlen(message))   /* enough room? */
            _errmsg(0,"Pipe overflow\n");   /* no */
        else
            write(path,message,strlen(message));    /* write message */
    }
}
```

8.8 DISK FILES

Disk files provide a sophisticated form of inter-process communication. Large amounts of data can be passed, and inter-process synchronization is provided by record locking. The data is not transient[9] - contrast pipes - and is not lost on power-down, even if it occurs while the file is still open (the file structure maintained by the RBF file manager is very robust). The disadvantages are the lower reliability, slower access speed, and generally

9 Except a volatile RAM disk.

higher power consumption when compared to semiconductor memory (although the effective power consumption per bit stored is low for high capacity disk drives).

The record locking provided by RBF ensures inter-process synchronization. Note that record locking is not effective for C "file" operations - **fread()**, **fwrite()**, **fprintf()**, and so on - because these library functions maintain private buffers unknown to RBF. The name "record locking" derives from its principal application of databases. A database file (usually) holds an array of data structures known as "records". The aim of record locking is to prevent a process reading or - worse still - writing back stale data. For example, if record locking were not supported, the following disastrous sequence of events could take place:

1) Process A reads a record from the file.
2) Process B reads the same record.
3) Process A modifies the record and writes it back to the file.
4) Process B modifies the record and writes it back to the file, cancelling the modifications made by process A.

Record locking works as follows. Consider two processes which have the same file open, and one, which has opened the file in update (read and write) mode, performs a read. The other process will be queued if it attempts to read some or all of the same data, until the first process rewrites the data, or reads or writes a different part of the file (that is, reads or writes another record). Note that it is not sufficient for the first process to seek to another point in the file - it must read or write (or close the file) for the second process to be woken. Because RBF records the start position and length of the data read on each path, this record locking works correctly for any size of data structure ("record").

A potential problem with record locking is "deadlock". For example:

1) Process A reads a record from file 1.
2) Process B reads a record from file 2.
3) Process A attempts to read the same record from file 2, and is put to sleep by RBF.
4) Process B attempts to read the record from file 1 that was read by process A, and is put to sleep by RBF.

This would result in both processes sleeping forever, waiting for the other to release a record. RBF checks for this condition, and would return a deadlock error (**E_DEADLK**) to process B at step 4.

RBF implements another form of locking: end-of-file lock. If a process has a file open for write, and its file pointer is at the end of the file (so that the process is likely to be extending the file), another process reading the file will be suspended at end-of-file, rather than being returned an end-of-file error. The idea is that the second process should be made to wait until the first process has written more data to the file, rather than be told that there is no more data in the file. This gives a very useful inter-process synchronization during sequential file writing and reading, similar to a pipe. The sleeping process is woken with an end-of-file error if the first process closes the file.

8.8.1 RAM Disks

Because the OS-9 I/O structure separates the logical file management functions from the hardware control functions (into the file manager and device driver respectively), it is possible to use a "disk" device driver that actually manages an area of memory rather than a disk drive. The device driver considers the memory as an array of equal sized blocks - each block is the size of a "sector", as specified in the device descriptor. When RBF requests that a series of sectors be read or written, the device driver simply copies between the buffer supplied by RBF and the corresponding memory blocks in the "disk".

A "memory disk" (or RAM disk) has two important benefits:

a) Very fast access.

b) Provides all the functions of disk files (such as record locking) without the need for a disk interface or a disk drive (relatively unreliable, large, and power hungry).

The disadvantage of a memory disk is the relatively high cost per bit. Memory disks are therefore useful for providing inter-process communications facilities, the temporary storage of frequently required data, and the storage (in non-volatile memory) of permanent data in diskless systems.

The RAM disk driver provided by Microware (**ram**) supports both volatile and non-volatile memory disks. Volatile memory is memory that loses its contents when the power is removed. In general the main memory of a

computer is volatile - usually a type of memory known as "dynamic" RAM, which is low cost but uses too much current to be powered from a backup battery when the main power is removed. Non-volatile memory does not lose its contents when the power is removed. This is usually a special area of battery-backed, low power consumption "static" RAM, or ROM (ROM can be used for a read-only "memory disk").

The device driver decides that a volatile disk is required if the "port address" (**M$Port**) field in the device descriptor is less than 1024. In this case when the device driver is initialized it allocates memory from the main system memory, using the standard **F$SRqMem** memory allocation system call. The amount of memory is calculated from the parameters in the device descriptor - the number of sectors per track and the number of sectors on track zero added together, and multiplied by the sector size (fixed at 256 bytes). The device driver then initializes the memory as if the **format** utility had been used, creating an initialized empty "disk". When the device is terminated, the termination routine of the device driver de-allocates the memory.

Usually several volatile RAM disk device descriptors are provided (in the 'CMDS' or 'CMDS/BOOTOBJS' directory). All have the module name **r0**, with various memory sizes specified. The file is given a name indicative of the memory size - for example, 'r0_256k' would be for a RAM disk '/r0' with a size of 256k bytes. The choice of RAM disk size usually depends on how much memory can reasonably be set aside for this purpose. Typically the desired device descriptor would be loaded in the 'startup' file.

If the "port address" is greater than or equal to 1024, the device driver assumes it is the address of an area of non-volatile memory, not known to the operating system's memory allocation functions. An area of memory is not known to the operating system if it is not in the memory lists in the **init** configuration module. The device driver initialization function does not initialize the memory in any way if the device descriptor is marked as format protected (bit 0 of the **PD_Cntl** field is set) - the **format** utility must be used (provided the memory is writable) with an appropriate alias device descriptor that is not format protected.

If the device descriptor is not format protected, and the disk validation field (**DD_Sync**) in sector zero is not correct (it must be $4372757A), the device driver initializes the memory. So the first time the memory is used, the device driver initializes it, creating an "empty" disk. On subsequent uses the device driver does not initialize the memory (unless the disk validation field has become corrupted), so files are preserved.

A ROM disk is a useful way of providing fixed data to programs on a diskless system, where the programs normally expect to be reading disk files. For example, the 'termcap' file required by the **umacs** editor could be stored in a ROM disk. A ROM disk can be created with the following sequence of operations:

a) Load and initialize a volatile RAM disk device descriptor of the desired ROM disk size:

```
$ load BOOTOBJS/r0_64k
$ iniz /r0
```

b) Copy the desired files to the RAM disk:

```
$ dsave -eb100 /r0
```

c) Save the entire RAM disk to a disk file:

```
$ merge /r0@ -b100 >rom_disk
```

d) Program PROMs from the disk file.

e) Make a new device descriptor for the ROM disk, with the "port address" set to the address at which the PROMs are accessed. This can be done by using the **moded** utility on a copy of the RAM disk device descriptor file, changing the module name and port address.

Because I/O sub-systems are dynamically initialized and terminated under OS-9, a volatile memory disk must be explicitly initialized (**I$Attach** system call) to remain in existence with no paths open to it:

```
$ load r0_256k
$ iniz /r0
```

This is typically done in the 'startup' file. The RAM disk can be terminated (**I$Detach** system call) using the **deiniz** utility. Note that both attaching a device and changing directory to it increment the device use count. So to get rid of a RAM disk (return its memory to the free pool) it must be "detached" (**deiniz** utility) as many times as it was explicitly "attached" (**iniz** utility) *plus* as many times as **chx** and **chd** were used on it. Once the RAM disk has been terminated all the files in the RAM disk are lost. A non-volatile memory disk (battery-backed RAM, or ROM) does not need initializing or terminating.

The **ram** device driver provided by Microware performs no data integrity checks, other than a check of the validation word in sector zero. A battery-backed RAM disk can become corrupted due to power failure or program errors, particularly in systems that do not have the SSM for

inter-task memory protection. If the integrity of files in a battery-backed RAM disk is important, it may be advisable to protect against undetected corruption by modifying the **ram** device driver to write a CRC (Cyclical Redundancy Check) with each sector, and to check the CRC when a sector is read. The OS-9 module CRC system call (**F$CRC**) can be used for this purpose, in which case an additional three bytes must be allocated for each "sector" of memory. To make this modification requires the source code to the **ram** device driver.

8.9 DATA MODULES

All multi-tasking operating systems must address the need for memory areas accessible by multiple processes, for use as data pools, buffers, and common information structures. OS-9 offers an elegant solution by making use of the OS-9 memory module concept. Normally a module must be present at startup (in ROM, for example), or be loaded using the **F$Load** system call. However, in addition a module can be created in dynamically allocated memory, using the **F$DatMod** system call. Prior to OS-9 version 2.3 only a module of type "data" could be created in this way. The chapter on the OS-9 System Calls gives a detailed description of the **F$DatMod** system call, including how to create data modules in coloured memory, and how to create modules of other types.

The **F$DatMod** system call, available through the **_mkdata_module()** and **make_module()** C library functions, should be considered as a way of allocating a named memory area of any desired size. The module has a header, body, and CRC just like other modules, but it is the body only that is of interest to the programmer - this is the "allocated memory". The kernel creates the module such that the body is equal in size to the size requested by the **F$DatMod** system call (so the module in total is slightly larger), and clears the body to zeros, which can be a useful initialization aid. The CRC is initially correct, although it becomes invalid once data has been written to the module. This is of no importance unless the module is to be saved, and later loaded from disk or blown into ROM.

Because a data module is created from dynamically allocated system memory, the program cannot know the address of the memory at compile time - the address of the data module is returned by the **F$DatMod** system call. Therefore the memory must be addressed register indirect (assembly language), or by a pointer (C language). The program will usually maintain two pointers, one giving the address of the module header (for later

unlinking), and the other pointing to the body of the module – the "allocated common memory".

The creation of a data module for shared memory is more elegant and more public than the alternative of one process passing the address of a memory area to other processes. In addition, the data module approach is compatible with the System Security Module inter-task memory protection. If the SSM is being used on a system, a process cannot access memory that it has not itself allocated, even if it has the address. However, if it creates or links to a module the kernel adds the module's address space to the process's memory map, so the process can access it. Thus the use of a data module for shared memory is upwardly compatible with systems using the SSM, while passing the address of a memory area is not. Note that the permissions (public, group, and user) specified when the module was created control the access to the memory. If a process has only read or execute permission for the module, the memory management unit will be configured to give a bus error if the process tries to write to the module (provided the MMU has the capability, as is the case for the 68851, 68030, and 68040).

The most convenient way to create and use a data module is to define a single C structure containing all the elements required to be in the shared memory. The size value used to create the data module is then simply the size of the structure (**sizeof** keyword), and the pointer to the module body is a pointer to that type of structure.

While the **F$DatMod** system call returns both the address of the module header and the address of the module body, the C library function **_mkdata_module()** only returns the address of the module header. From this can be calculated the address of the module body, because the kernel initializes the "execution entry offset" of the extended module header with the offset to the module body. The example below shows such a calculation.

Once a process has created the data module, other processes can link to it, just as they would link to any other module in memory, using the **F$Link** system call, or the **modlink()** C library function. This returns the address of the module header. As with the **_mkdata_module()** function, the address of the module body is calculated by adding the "execution entry offset" in the module header to the address of the module header.

Data modules can also be created and linked to by operating system components, such as device drivers. This can be used to provide shared memory between device drivers, or between a device driver and a process. If the data module is to be used by a device driver, it is a useful technique to

build the data module name from the device "port address". For example, if the port address is $FC480000, the data module name could be 'dmFC480000'. This allows other incarnations of the device driver to be active at the same time, controlling other devices of the same type, without a conflict of names between the data modules.

When a process (or operating system component) has finished with the data module – for example, when the process or device driver is about to terminate – it should unlink from the data module, using the **F$UnLink** system call or the **munlink()** C library function. The creation of the data module sets the module's link count to one, and each link to the module increments the link count. Each unlink decrements the link count, just as with any memory module. Once the link count reaches zero the module is removed from the module directory and its memory is returned to the free pool. Again, the use of data modules should simply be seen as named memory allocation, with the memory being returned to the free pool when no longer needed.

The kernel does not keep track of the modules a process or operating system component has created or linked to. Therefore if a process does not unlink from a data module – perhaps because it has been killed by the kernel or another process – the data module will remain in existence. This is not fatal, as the data module can be identified by name, and unlinked by the user or another process. Alternatively, if the application is restarted it can detect that the previous incarnation was abnormally terminated, because it is returned a "module already exists" error (**E_KWNMOD**) when it tries to create the module. The new incarnation can either exit with an error, or link to the already existing module, and clear the module body to zeros (as if it had just been created).

The same mechanism can be used for communication between loosely bound processes. As each process starts up, it attempts to create the data module. If it succeeds, it knows that it is the first process to use the module, and so must initialize the module. If it fails (with the error **E_KWNMOD**), it knows that it is not the first process – it then links to the module, and does not initialize it. If this approach is used, then some synchronization mechanism – such as an event – is needed to prevent subsequent processes using the module body before it has been completely initialized by the first process. This precaution is not needed if the initialization is performed in system state, for example by a device driver, because rescheduling will not take place while execution is in system state.

A data module is only shared memory – it does not provide any inter-process synchronization. Therefore, unless access is always in system state, or the use of fields within the data module has been carefully designed to need no interlocks between processes, some independent synchronization mechanism such as an event or signals must be used. This is because a process's time slice can end at any time, including while it is reading or writing to a data module (or any shared memory), and another process that uses the data module could become the current process.

It is possible to alter the behaviour of the kernel's process scheduler (see the chapter on "Multi-tasking"), but in general such an approach is less flexible, more difficult to debug, and more likely to cause problems for future adaptations of the software under development. Certain 68000 instructions are indivisible, and these can be used in many applications to avoid the need for a synchronization mechanism. Reading and writing of words and long words, and the "bit change" instructions, are useful examples. The C compiler generates these instructions, as can be seen by inspecting the assembly language output of the compiler, or the necessary small functions can be written in assembly language (see the chapter on Microware C and Assembly Language).

The following example shows the creation of a data module, whose body is to contain a declared structure type. If the module already exists, the program links to it instead. The address of the body of the module is calculated, and a character string is copied to one of the structure elements. Note that the macro **mkattrevs** that builds the attribute and revision word for creating the module is defined in the file 'module.h'. As usual, for clarity all error handling has been omitted, except for the check that the reason for being unable to create the module was because a module of that name already existed:

```
#include <module.h>     /* module header structure declarations */
/* The module will have read and write permission for all processes: */
#define PERMS (MP_OWNER_READ+MP_OWNER_WRITE+MP_GROUP_READ
       +MP_GROUP_WRITE+MP_WORLD_READ+MP_WORLD_WRITE)
#define ERROR -1

/* These functions return a pointer to a module header: */
mod_exec *modlink(),_mkdata_module();

/* This is the structure the data module will contain: */
typedef struct {
    int msg_len;
    char msg_str[100];
} data_struct;
```

```
char *mod_name="data_module",    /* the name of the data module */
    *message;                    /* the message string to write */

main(argc,argv)
int argc;
char **argv;
{
    mod_exec *mod_ptr;           /* pointer to module header */
    data_struct *data_ptr;       /* pointer to module body */

    /* Try to create the data module: */
    if ((mod_ptr=_mkdata_module(mod_name,sizeof(data_struct),
        mkattrevs(MA_REENT,1),PERMS))==(mod_exec *)ERROR) {
        /* Couldn't create the data module: */
        if (errno!=E_KWNMOD)
            exit(errno);     /* fatal error */
        /* Module already exists - link to it: */
        mod_ptr=modlink(mod_name,0);
    }
    /* Calculate the address of the module body using the
        address of the module header, and the offset in the header: */
    data_ptr=(data_struct *)((char *)mod_ptr +mod_ptr->_mexec);
    /* Write the message to the structure in the data module: */
    strcpy(data_ptr->msg_str,message);
    /* And the length of the message: */
    data_ptr->msg_len=strlen(message);
}
```

8.10 SHARED EXTERNAL MEMORY

OS-9 only "knows" about memory areas specified to it in the memory search lists of the boot ROM and the **init** module. Therefore memory can be "hidden" from the operating system. Examples of memory which might usefully be excluded from the memory lists are:

a) Battery-backed memory for configuration parameters.

b) I/O memory, such as graphics display RAM.

c) Inter-processor communications mailboxes and buffers.

d) Fixed inter-process communication data space (not recommended).

When considering whether to "hide" an area of memory from the operating system you should first consider declaring it as coloured memory. If the memory area is given a priority of zero in the memory list, memory from that area can only be allocated by specific reference to its colour. This approach is more portable than creating a program that "knows" the absolute memory

address of a special memory area. However, the operating system will use the first few bytes of the memory to link it into the free memory lists, and this may be undesirable for certain types of special memory.

If the System Security Module is not used, there is nothing preventing a process directly addressing such memory (or any memory location). A process may also directly access the registers of an interface chip, such as a parallel port. This is perfectly acceptable provided you are sure it will not conflict with accesses from other processes, or a device driver. However, if the SSM is used these areas of "hidden" memory are not normally mapped in to a process's permitted memory map, and the process will generate a bus error if it attempts to read or write in that memory area. Such memory can normally only be accessed in system state, when the memory management unit's protections are suspended.

However, because this could be a serious restriction in certain applications, OS-9 allows a process to gain permission to access any memory area by using the **F$Permit** system call. This system call adds a memory area to the memory map of a process. The process can request any combination of read, write, and execute permissions (although the 68851 and the MMUs in the 68030 and 68040 only support read and read-and-write, so a request for execute permission gives read permission). The complementary system call **F$Protect** requests that the memory area be removed from the process's memory map.

Note that these system calls are actually installed by the SSM during its initialization, and are not part of the kernel. At coldstart the kernel installs handlers for these system calls that simply test the first byte of the memory area:

```
tst.b   (a2)
```

so this is the action taken if the SSM is not in use[10]. These system calls are only permitted from a process created by a super user (a member of group zero), or that has changed its user number to zero (using the **F$SUser** system call) - only permitted if the program module was created by a super user. These system calls are described in detail in the chapter on the OS-9 System Calls.

Be aware that the normal data and program caching hardware facilities of the processor (if any) will still be operational during accesses to memory revealed by **F$Permit**. Therefore accesses to I/O device registers are likely to cause problems if the processor has a data cache, as the processor may return

[10] Under OS-9 version 2.2 the kernel does not install default handlers for these calls, so if the SSM is not in use these calls return an "unknown service request" error (**E$UnkSvc**).

a value from the cache rather than re-reading the desired register, unless external address decoding circuitry inhibits caching during accesses to the I/O device registers. (The kernel disables the processor data cache during I/O system calls, so device driver accesses to I/O device registers are not cached).

There is no C library function to make the **F$Permit** system call, so the assembly language for a C-callable function is given below. The example shows a C program calling the assembly language function, followed by the function itself. The **F$Protect** system call (which is rarely needed) takes the same parameters, except that the **d1** register is not used.

```
map_in()
{
    /* Map in 64k of memory at address $FC840000, requesting
       read and write permission: */
    if (f_permit(0x10000,S_IREAD|S_WRITE,0xfc840000)==ERROR)
        exit(_errmsg(errno,"Can't access memory\n"));
}

#asm
*   f_permit(size,perms,address)
*   The F$Permit system call requires:
*       d0.l = size of memory area to map in
*       d1.w = access permissions
*       a2.l = start address of memory area to map in
f_permit:
            move.l  a2,-(a7)        save register
            movea.l 8(a7),a2        get start address parameter
            os9     F$Permit        map in memory
            bcc.s   f_permit10      ..success
            moveq   #0,d0
            move.w  d1,d0           copy error code
            move.l  d0,errno(a6)    save it
            moveq   #-1,d0          show error
            bra.s   f_permit20
f_permit10
            moveq   #0,d0           show no error
f_permit20
            movea.l (a7)+,a2        retrieve register
            rts
#endasm
```

8.11 ALARMS

Alarms are not strictly an inter-process communication mechanism, as they do not provide a means by which one process can communicate with another. Rather, they allow the clock tick interrupt handler function to communicate with a process.

A process installs an alarm using the **F$Alarm** system call. This requests that the kernel send a signal of a specified code to the process at a future time. Two types of alarm are available - single shot, and cyclic (periodic). The single shot alarm sends a signal after a specified number of ticks have elapsed (relative alarm), or at a specified date and time (absolute alarm), and then cancels (deletes) itself. The cyclic alarm sends signals repeatedly at the specified interval of ticks, until the process explicitly deletes the alarm, or the process dies. The kernel automatically deletes all outstanding alarms for a process when the process dies.

A single shot alarm allows a process to implement a timeout, for example when waiting for data to arrive on a serial port. A cyclic alarm is a useful means of getting a process to execute a sequence of instructions at strict intervals, independent of the time taken to execute the instructions (provided it does not exceed the alarm interval!). A cyclic alarm can also be used as a watchdog timer - the signal intercept routine of the process checks whether the main program body has set a flag in time, before the alarm signal was received.

A process can have any number of alarms installed at any one time. The **F$Alarm** system call returns a unique ID (actually the address of the alarm "thread execution block"), which is used to identify the alarm when deleting it. An alarm can be deleted using the **F$Alarm** system call, preventing any subsequent signals being sent by the alarm. Passing zero as the alarm ID when deleting alarms causes all alarms belonging to the process to be deleted (the kernel makes this call when a process dies). Only the creator of an alarm (same process ID) or a super user process (group zero) can delete an alarm.

The information about an alarm is held in a "thread execution block" allocated by the kernel when the alarm is created (see the section on the Process Descriptor in the chapter on the OS-9 Internal Structure). The thread block is linked into a linked list of thread blocks, rooted in the System Globals. The linked list is ordered by execution time - the first entry in the list will be executed first, and so on. Alarms are inserted in the list when they are created, and cyclic alarms are re-inserted in the list after every execution, ready for the next execution.

For an absolute alarm - set by date and time - the alarm date and time are stored in Julian format. Absolute alarms can therefore only be set to a resolution of one second. For a relative or cyclic alarm the alarm time given to the call is added to the current value of the **D_Ticks** field of the System Globals (ticks since system startup) before being stored in the thread block. Relative and cyclic alarms can therefore be specified to a resolution of one

tick. Note that if bit 31 of the time given to the call is set this indicates that the time value is given in 256ths of a second. The kernel clears bit 31, and converts the time to the nearest tick. This avoids the need for the programmer to know the tick period of the system. The minimum time that can be specified is one tick. This will cause a relative alarm to execute at the next tick, and a cyclic alarm to execute every tick.

Alarms are not directly acted on by the kernel's tick interrupt handler. Instead, the tick handler wakes up the System Process (see the chapter on Multi-tasking), and the System Process sends the alarm signals. The System Process has the highest possible process priority (65535), so it is sure to execute as soon as any currently executing interrupt service routines have finished, and any currently executing system call has finished or gone to sleep - that is, before any other program can continue execution in user state. This means that from a programming point of view the effect is exactly the same as the signals being sent from the tick interrupt handler, but because the tick interrupt handler does not have to handle the alarms it executes more quickly, and so allows other interrupts to be serviced with less latency.

The System Process, once activated, and having checked the timed sleep queue, checks every alarm in the queue for relative and cyclic alarms, comparing the alarm time in the thread block with the current value of **D_Ticks**. If the alarm time has been reached (or passed), the System Process executes the thread block function (sends the alarm signal), and removes the thread block from the queue. If the alarm is cyclic, the System Process adds the cyclic period to the alarm time in the thread block, and re-inserts it in the queue. Otherwise it de-allocates the thread block. Once all entries in the queue have been checked (stopping at the first entry that does not need execution, as the queue is in time order), the kernel calculates how many ticks must elapse before the first alarm still in the queue (if any) is to be executed. If this is less than the current value of **D_Elapse** (set by the check of the sleep queue), the System Process updates **D_Elapse** with the lesser value, so that it will wake up when necessary to execute the alarm.

Having checked the relative and cyclic queue, the System Process then checks the absolute queue, comparing each alarm date and time with the current date and time in the **D_Julian** and **D_Second** fields of the System Globals. If the alarm date and time have been reached (or exceeded) the System Process executes the thread block function, and deletes the thread block. Once all entries have been checked, the System Process checks the **D_Elapse** field, just as for the relative and cyclic alarms.

If the system date and time are changed (by the **F$STime** system call), the kernel forces the System Process to be active, causing a check of the alarm queues. Therefore any absolute alarms that have expired as a result of the change are immediately executed. Note, however, that as the **F$STime** system call does not update the **D_Ticks** field of the System Globals (number of ticks since system startup), relative alarms are not affected by the date and time change.

The **F$Alarm** system call is used for all the alarm operations, with a function code specifying which operation is required. Separate C library functions are provided for each of the alarm operations. The table below shows the function codes with their symbolic names from the file 'DEFS/funcs.a', the corresponding C library functions, and a brief description of each operation.

Code	Name	C function	Description
0	A$Delete	alm_delete	Delete an alarm, given the alarm ID (or zero to delete all alarms of a process).
1	A$Set	alm_set	Create a relative single shot alarm.
2	A$Cycle	alm_cycle	Create a cyclic alarm.
3	A$AtDate	alm_atdate	Create an absolute alarm, given a date and time in Gregorian format (YYYYMMDD, 00HHMMSS).
4	A$AtJul	alm_atjul	Create an absolute alarm, given a date and time in Julian format (date as days since 2nd January, year -4712, and time as seconds since midnight).

Alarms should be used with care. As described above, a process should if possible have only one sequence of instructions to execute, and so under normal circumstances it should not need the asynchronous change of flow of control provided by signals. A program that makes regular use of signals (rather than for exceptional conditions) is likely to be overly complex. Consider whether instead the program could be broken down into two or more separate processes using events, or using signals only as a synchronization mechanism.

8.11.1 System State Alarms

The System Process does not know implicitly what action to take when an alarm must be executed. Instead, it uses the register stack frame image that was built in the thread block by the **F$Alarm** system call. If the system call is made from user state, the **F$Alarm** handler routine builds an appropriate

register stack image for an **F$Send** (send a signal) system call - **d0.w** is the process ID (of the calling process), **d1.w** is the requested signal code, and the program counter is set to the address of the **F$Send** system call. This causes the **F$Send** system call to be made as the execution of the alarm. However, any subroutine can be called by the System Process.

When the System Process has determined that an alarm must be executed, it switches its user group and user number to those of the process that created the alarm (in the **P$User** field of the System Process process descriptor). It then sets the exception abort stack and return program counter (**P$ExcpSP** and **P$ExcpPC**) for a clean return to itself (because the execution is in system state, so without this provision an exception in the execution would cause a system crash). Finally, it takes the registers from the stack frame of the thread block (**d0** to **d7** and **a0** to **a3** - **a4** is the address of the System Process process descriptor, **a5** is the address of the stack frame in the thread block, and **a6** is the address of the System Globals), and calls the subroutine whose address is in the program counter field of the stack frame (**R$pc**).

When the subroutine returns, if the carry flag is set the System Process puts the error code in the **d1.w** register into the **d1.w** register of the stack frame (clearing the high word of **d1.l**). Lastly, it puts the returned Condition Codes register (**ccr**) in the stack frame (**R$ccr**), thus setting the carry flag in the stack frame if there was an error. This is a convenience for future uses of thread blocks, as alarm thread blocks are never returned to the caller. However, because the register stack frame is modified in this way, and also might be modified by the execution subroutine, when executing a cyclic alarm the System Process actually makes a copy of the stack frame in the thread block (on its stack) and uses that for the execution (**a5** points to it), so any changes to the stack frame used for the execution do not affect subsequent executions of the cyclic alarm.

The result is that alarms work differently when installed from system state, such as from a device driver or kernel customization module. Instead of sending a signal, the alarm execution uses a register stack frame given to the **F$Alarm** system call, which is copied to the thread block. The caller can therefore specify all the data and address registers used when the alarm is executed (except **a4**, **a5**, and **a6**, which are pre-defined - see above), and the address of the subroutine to call (in the program counter field of the stack frame - **R$pc**). This allows operating system components to "hook" subroutines into the clock tick interrupt service routine, providing watchdog, timeout, and polling functions independent of any calling process. Note that as with user state alarms the alarm is executed by the System Process, not

directly by the tick interrupt service routine, and interrupts are enabled when the alarm is executed.

For example, a single shot alarm can be used to turn off a floppy disk drive motor when the drive has not been used for a certain time, and a periodic alarm can be used to poll for input from a device that cannot generate interrupts. Because the caller specifies the processor register values to use when the installed routine is called, the routine can access the static storage of the caller (such as the device static storage used by a device driver), using the same symbolic names. In many ways an alarm routine installed from system state is very similar to an interrupt service routine – it is called asynchronously to the main body of (for example) the device driver, and can share static storage with the main body. However, because it is called from a process (the System Process), the alarm routine will not be called during the execution of a system call – it cannot break into the execution of the main body of a device driver, for example. The execution of the alarm routine will be deferred until the system call finishes or goes to sleep.

A system state alarm routine is called with the processor in supervisor state, and so has all the responsibilities of any system state routine. Although the System Process changes the group and user in its process descriptor to that of the creator of the alarm, the routine is still called as a subroutine of the System Process – the current process is the System Process. Therefore the routine must not sleep in any way (sleep, wait for event, make an I/O request that might sleep, and so on), because this would suspend the maintenance of the timed sleep queue and other alarms. However, other system calls that are forbidden in interrupt service routines can be used, because the System Process is scheduled in as the current process in the normal way, so there is no possibility of breaking into a system call being made by another process.

Similarly, the normal system state hardware exception recovery mechanism applies. If a bus error or other hardware exception occurs, control is transferred to the address given in the **P$ExcpPC** field of the process descriptor, with the stack pointer given in the **P$ExcpSP** field (see the chapter on Exception Handling). By default the System Process sets these before executing each alarm for a clean return to itself, ignoring any hardware exceptions.

When an alarm is created, using the **F$Alarm** system call, the thread block that is allocated is linked in to the linked list of thread blocks allocated by the calling process. When the process dies, the kernel deletes all outstanding alarms for the process. This applies whether the call is made from user or system state. However, this is normally undesirable in system state, as an

alarm installed by a device driver in response to an initialization caused by a path being opened by a process must not be deleted simply because that process has died - other processes may now have paths open on the device. This difficulty may be avoided by temporarily substituting the address of the System Process's process descriptor for the current process before making the **F$Alarm** system call. The thread block will then be allocated to the System Process. Also, the group and user for the alarm will be that of the System Process - 0.0. The following code fragment shows an example of this technique:

```
* Alarm time and date are in d3 and d4, function code is in d1
        move.l  D_Proc(a6),-(a7)    save current process descriptor
        move.l  D_SysPrc,D_Proc(a6) make System Process current process
        lea     AlarmHandler(pc),a0 point at alarm subroutine
        suba.w  #R$Size,a7          make room on stack for stack frame
        move.l  a0,R$pc(a7)         set routine address
        movem.l d0-d7/a0-a3,(a7)    set other registers for call
        movea.l a7,a0               copy stack frame address for call
        os9     F$Alarm             make system call
        move.w  sr,d2               save carry flag
        adda.w  #R$Size,a7          ditch stack frame
        move.l  (a7)+,D_Proc(a6)    restore current process
        move.w  d2,sr               restore carry (error flag)
* Alarm ID is in d0.l, unless carry is set.
```

Because the System Process never dies, the kernel will not automatically delete alarms that have been installed in this way. This is similar to other resources installed in system state. For example, a device driver that uses an alarm must make sure that the alarm is deleted as part of its termination routine, otherwise the System Process could attempt to call an alarm routine that is no longer in memory.

The C library functions mentioned above assume that the calls are being made from user state, and are not suitable for use from system state. Therefore if you are writing system state code (such as a device driver) in C, you will need to write your own C-callable alarm functions in assembly language. The writing of C-callable functions in assembly language is described in the chapter on Microware C and Assembly Language.