Interprocess Communications

This chapter describes the five forms of interprocess communication that OS-9 supports:

- Signals
- Alarms
- Events
- Pipes
- Data modules

Signals synchronize concurrent processes. ***Alarms*** send signals or execute subroutines at specified times. ***Events*** synchronize concurrent processes' access of shared resources. ***Pipes*** transfer data among concurrent processes. ***Data modules*** transfer or share data among concurrent processes.

Signals

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

Signals are also referred to as ***software interrupts*** because a process receives a signal similar to a CPU receiving an interrupt. Signals enable a process to send a "numbered interrupt" to another process. If an active process receives a signal, the intercept routine executes immediately (if installed) and the process resumes execution where it left off. If a sleeping or waiting process receives a signal, the process moves to the active queue, the signal routine executes, and the process resumes execution immediately after the call that removed it from the active queue.

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

Each signal has two parts: the process ID of the destination and a signal code. OS-9 supports the following signal codes in user-state:

Signal	Description
0	Unconditional system abort signal. The super-user can send the kill signal to any process, but non-super-users can send this signal only to processes with their group and user IDs. This signal terminates the receiving process, regardless of the state of its signal mask, and is not intercepted by the intercept handler.
1	Wake-up signal. Sleeping/waiting processes which receive this signal are awakened, but the signal is not intercepted by the intercept handler. Active processes ignore this signal. A program can receive a wake-up signal safely without an intercept handler. The wake-up signal is not queued if the process's signals are masked.
2	Keyboard abort signal. Typing control-E sends this signal to the last process to do I/O on the terminal. Usually, the intercept routine performs exit(2) upon receiving a keyboard abort signal.
3	Keyboard interrupt signal. Typing control-C sends this signal to the last process to do I/O on the terminal. Usually, the intercept routine performs exit(3) upon receiving a keyboard interrupt signal.

Signal	Description
4	Hang-up signal. SCF sends this signal when it discovers that the modem connection is lost.
5-31	These signal numbers are reserved for future use by Microware. Signals in this range are considered deadly to the I/O system.
32-255	These signal numbers are reserved for future use by Microware.
256-65535	User-defined signals. These signal numbers are available for use in user applications.

You could design a signal routine to interpret the signal code word as data. For example, you could send various signal codes to indicate different stages in a process's execution. This is extremely effective because signals are processed immediately upon receipt.

The following system calls enable processes to communicate through signals:

Name	Description
F$Send	Sends a signal to a process.
F$Icpt	Installs a signal intercept routine.
F$Sleep	Deactivates the calling process until the specified number of ticks has passed or a signal is received.
F$SigMask	Enables/disables signals from reaching the calling process.

For specific information about these system calls, refer to ***OS-9 System Calls***. The Microware C Compiler supports a corresponding C call for each of these calls, as well.

NOTE: Appendix A contains a program which demonstrates how you may use signals.

Alarms

User-state Alarms

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

OS-9 supports the following user-state alarm functions:

A$Delete	Remove a pending alarm request
A$Set	Send a signal after specified time interval
A$Cycle	Send a signal at specified time intervals
A$AtDate	Send a signal at Gregorian date/time
A$AtJul	Send a signal at Julian date/time

Cyclic Alarms

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

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

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

You could give each function the process must monitor a cyclic alarm, whose period is the desired refresh rate, and whose signal code identifies the particular display function. The signal handling routine might read an appropriate sensor and directly update the dashboard display. The system takes care of all of the timing details.

Time of Day Alarms

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

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

Relative Time Alarms

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

System-state Alarms

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

OS-9 supports the following system-state alarm functions:

A$Delete	Remove a pending alarm request
A$Set	Execute a subroutine after a specified time interval
A$Cycle	Execute a subroutine at specified time intervals
A$AtDate	Execute a subroutine at a Gregorian date/time
A$AtJul	Execute a subroutine at Julian date/time

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

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

One way to arrange for a persistent alarm is to execute the F$Alarm request on behalf of the system process, rather than the current I/O process. Do this by moving the system variable D_SysPrc to D_Proc, executing the alarm request, and restoring D_Proc. For example:

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

WARNING: If you use this technique, you must ensure that the module containing the alarm subroutine remains in memory until after the alarm has expired.

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

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

NOTE: Refer to Appendix A for a program demonstrating how you can use alarms.

Events

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

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

An OS-9 event is a 32-byte system global variable maintained by the system. Each event includes the following fields:

Event ID	This number and the event's array position create a unique ID.
Event name	This name must be unique and cannot exceed 11 characters.
Event value	This four-byte integer value has a range of 2 billion.
Wait increment	This value is added to the event value when a process waits for the event. It is set when the event is created and does not change.
Signal increment	This value is added to the event value when the event is signaled. It is set when the event is created and does not change.
Link Count	This is the event use count.
Next event	This is a pointer to the next process in the event queue. An event queue is circular and includes all processes waiting for the event. Each time the event is signaled, this queue is searched.
Previous event	This is a pointer to the previous process in the event queue.

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

You may use events directly as service requests in assembly language programs. The Microware C compiler supports a corresponding C call for each event system call.

The Wait and Signal Operations

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

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

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

When a process finishes with the printer, the process signals the event, that is, it applies the signal increment to the event value. Then, the event queue is searched for a process whose event value range includes the current event value. If such a process is found, the process activates, applies the wait increment to the event value, and uses the printer.

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

- ¿ Wait for the resource to become available.
- ¡ Mark the resource as busy.
- ¬ Use the resource.
- Ð Signal that the resource is no longer busy.

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

NOTE: Appendix A includes a program which demonstrates how you may use events.

The F$Event System Call

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

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

Currently, OS-9 has functions to allow you to create, delete, link, unlink, and examine events (listed below). It also provides several variations of the Signal and Wait operations.

The F$Event description in ***OS-9 System Calls*** discusses specific parameters and functions of each event operation. The system definition file funcs.a defines Ev$ function names. Resolve actual values for the function codes by linking with the relocatable library sys.l or usr.l.

OS-9 supports the following event functions:

Ev$Link	Link to an existing event by name.
Ev$UnLnk	Unlink an event.
Ev$Creat	Create a new event.
Ev$Delet	Delete an existing event.
Ev$Wait	Wait for an event to occur.
Ev$WaitR	Wait for a relative to occur.
Ev$Read	Read an event value without waiting.
Ev$Info	Return event information.
Ev$Pulse	Signal an event occurrence. Temporarily changes the event value.
Ev$Signl	Signal an event occurrence. Changes the event value.
Ev$Set	Set an event variable and signal an event occurrence.
Ev$SetR	Set a relative event variable and signal an event occurrence.

Pipes

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

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

A pipe may contain up to 90 bytes, unless a different buffer size was declared. Typically, a pipe is used as a one-way data path between two processes: one writing and one reading. The reader waits for the data to become available and the writer waits for the buffer to empty. However, any number of processes can access the same pipe simultaneously; pipeman coordinates these processes. A process can even arrange for a single pipe to have data sent to itself. You could use this to simplify type conversions by printing data into the pipe and reading it back using a different format.

Data transfer through pipes is extremely efficient and flexible. Data does not have to be read out of the pipe in the same size sections in which it was written.

You can use pipes much like signals to coordinate processes, but with these advantages:

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

Named and Unnamed Pipes

OS-9 supports both named and unnamed (anonymous) pipes. The shell uses unnamed pipes extensively to construct program “pipelines,” but user programs may use them as well. Unnamed pipes may be opened only once. Independent processes may communicate through them only if the pipeline was constructed by a common parent to the processes. Do this by making each process inherit the pipe path as one of its standard I/O paths.

Named and unnamed pipes function nearly identically. The main difference is that several independent processes may open a named pipe, which simplifies pipeline construction. The sections that follow note other specific differences.

Operations on Pipes

Creating Pipes

The I$Create system call is used with the pipe file manager to create new named or unnamed pipe files.

You may create pipes using the pathlist /pipe (for unnamed pipes, pipe is the name of the pipe device descriptor) or /pipe/<name> (<name> is the logical file name being created). If a pipe file with the same name already exists, an error (E$CEF) is returned. Unnamed pipes cannot return this error.

All processes connected to a particular pipe share the same physical path descriptor. Consequently, the path is automatically set to update mode regardless of the mode specified at creation. You may specify access permissions; they are handled similarly to RBF.

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

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

Opening Pipes

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

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

The only way for more than one process to share the same unnamed pipe is through the inheritance of the standard I/O paths through the F$Fork call. As an example, the outline on the following page describes a method the shell might use to construct a pipeline for the command dir -u ! qsort. It is assumed that paths 0,1 are already open.

```
StdInp =  I$Dup(0)              save the shell's standard input
StdOut =  I$Dup(1)              save shell's standard output
          I$Close(1)            close standard output
          I$Open("/pipe")       open the pipe (as path 1)
          I$Fork("dir","-u")    fork "dir" with pipe as standard output
          I$Close(0)            free path 0
          I$Dup(1)              copy the pipe to path 0
          I$Close(1)            make path available
          I$Dup(StdOut)         restore original standard out
          I$Fork("qsort")       fork qsort with pipe as standard input
          I$Close(0)            get rid of the pipe
          I$Dup(StdInp)         restore standard input
          I$Close (StdInp)      close temporary path
          I$Close (StdOut)      close temporary path
```

The main advantage of using named pipes is that several processes may communicate through the same named pipe without having to inherit it from a common parent process. For example, you can approximate the above steps with the following command:

```
dir -u >/pipe/temp & qsort </pipe/temp
```

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

Read/ReadLn

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

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

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

Write/WritLn

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

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

Close

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

Getstat/Setstat

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

GetStat Status Codes

Name	Description
SS_Opt	Reads the 128 byte option section of the path descriptor. You can use it to obtain the path type, data buffer size, and name of pipe.
SS_Ready	Tests whether data is ready. Returns the number of bytes in the buffer.
SS_Size	Returns the size of the pipe buffer.
SS_EOF	Tests for end-of-file.
SS_FD	Returns a pseudo-file descriptor image.

Other codes are passed to the device driver.

SetStat Status Codes

Name	Description
SS_Attr	Changes the pipe file's attributes.
SS_Break	Forces disconnection.
SS_FD	Does nothing, but returns without error.
SS_Opt	Does nothing, but returns without error.
SS_Relea	Releases the device from the SS_SSig processing before data becomes available.
SS_Size	Resets the pipe buffer if the specified size is zero. Otherwise, it has no effect, but returns without error.
SS_SSig	Sends a signal when the data becomes available.

Other codes are passed to the device driver.

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

Pipe Directories

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

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

The head of a linked list of named pipes is in the static storage of the pipe device driver (usually the null driver). If there are several pipe descriptors with different default pipe buffer sizes on a system, the I/O system notices that the same file manager, device driver, and port address (usually zero) are being used. It will not allocate new static storage for each pipe device and all named pipes will be on the same list.

For example, if two pipe descriptors exist, a directory of either device reveals all the named pipes for both devices. If each pipe descriptor has a unique port address (0,1,...), the I/O system allocates different static storage for each pipe device. This produces more predictable results.

Data Modules

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

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

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

Creating Data Modules

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

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

The Microware C compiler provides several C calls to create and use data modules directly. These include the _mkdata_module() call, which is specific to data modules, and the modlink(), modload(), munlink(), and munload() facilities which apply to all OS-9 modules. For more information on these calls, refer to the standard library sections of the ***OS-9 C Compiler User's Manual***.

The Link Count

Like all OS-9 modules, data modules have a link count associated with them. The link count is a counter of how many processes are currently linked to the module. Generally, the module is taken out of memory when this count reaches zero. If you want the module to remain in memory when the link count is zero, when you create the module make it "sticky" by setting the sticky bit in its attribute byte.

Saving to Disk

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

End of Chapter 4

NOTES