68 Disk I/O

The disk I/O routines explained in this section allow disk files to be read from and written to during Run mode. "Streams" describes how most of the routines operate on a data stream rather than the actual file. Under "Routines," all the disk I/O routines are explained. These routines perform read and write functions as well as other file maintenance tasks in Run mode, such as creating directories, renaming files, and deleting files.

68.1 Streams

Most disk I/O routines are not executed on the actual disk file, but on a stream which includes a copy of the file's data. Opening a disk file for reading or writing associates a stream with the file. A stream may be input or output. Input streams are read-only. Output streams are write-only. In either case, the stream remains associated with a disk file until the file is closed.

You may have more than one stream associated with a given file. (A maximum of ten streams may be open at one time.) For example, to read from and write to an existing file, you must open the file twice, once to create an input stream and once to create an output stream.

(A) Stream Components

A stream contains everything needed to perform disk I/O functions on a file.

1. *Buffer.* A buffer containing a copy of the data in a disk file is part of the stream. When a disk file is opened for reading, sectors of the disk containing the file are copied to this buffer.

   Sometimes a file's size may exceed the maximum size (512 bytes) of the buffer. In this instance, as much data from the file as will fit in the buffer is copied. As each character is read from the input stream, it is removed. (The *ungetc* routine may temporarily return a removed character to an input stream.) Each call to *fread*, *fgetc*, or *fgets* further empties the buffer, while leaving the contents of the disk file unchanged. When the buffer is empty, the next sector (or sectors) of the disk file is (are) automatically copied into the buffer.

   Similarly, when a file is opened for writing, the empty buffer is filled as *fwrite* or other output routines are invoked. Characters written to the output stream are not transferred to the disk file until there is a call to *fflush*. *Fflush* is automatic in *fclose* or when the stream buffer is full.

2. *File-position indicator*. The file-position indicator keeps track of progression through the disk file. For files opened in read mode, the indicator is initially located at the first character (character zero) in the file. As characters are read from the input stream, the indicator advances through the file.

   For existing files opened in append mode, the indicator is positioned after the last character in the file. For newly created files or files opened in overwrite mode, it is located at the beginning of the file. Every time an output routine is executed, the file-position indicator is advanced by the number of characters successfully written to the stream.

3. *Buffer pointer*. The stream also contains a pointer into the associated buffer of a file. In input streams, it points to the next character to be read. In output streams, it points to the next empty byte.

4. *EOF indicator*. If the end-of-file (EOF) indicator is set in a input stream, it means that a read operation encountered the end of the file. The EOF indicator is cleared via calls to *fopen*, *fseek*, *rewind*, *clearerr*, or *ungetc*.

5. *Error indicator*. In input streams, this indicator gets set when an *fread*, *fgetc*, or *fgets* routine does not successfully execute. Attempting to execute these input routines (or *ungetc)* on an output stream sets the error indicator. In output streams, the error indicator gets set when the *fflush*, *fwrite*, *fputc*, *fputs*, or *fprintf* routine does not successfully execute, or when output routines try to execute on an input stream. A call to *fopen*, *clearerr*, or *fseek*, clears the error indicator in either input or output streams. A *rewind* operation on an input stream also clears the indicator.

(B) Stream Pointer

The *fopen* routine returns a pointer to the stream. Disk I/O routines which perform operations on a stream require the stream pointer as an argument. It has been named *stream_ptr* in the routines discussed below.

(C) Locking Streams

Each file stream is locked internally during operations on it. If the user program is executing different conditions on multiple processors and both actions require writing to the *same file stream*, internally the *stdio* library will allow the first task that requests to write to execute until completion and the second task will be locked out. All processes that are locked out are temporarily put to sleep and removed from the tasking queues for that CPU. When the first process completes its operations on the stream, the locked-out processes are woken up and may try to claim the lock. Deadlock or deadly embrace situations can never arise internally to the *stdio* library.

If two or more file streams are associated with a single *file*, processes on each stream may try to operate on the file concurrently. Internal locking does not apply in this situation, so use the locking routines.

68.2 Routines

Disk I/O routines fall into four categories. The first category includes routines valid for both input and output streams, including the two locking routines (not exclusive to disk I/O). The remaining groups are routines valid for input streams only, routines applicable to output streams only, and routines which handle other file maintenance functions.

The routines and their descriptions closely conform to the ANSI specification for the Programming Language C, as defined in the draft document published July 9, 1986. Discrepancies with the ANSI standard are noted. The document number is X3J11-86-098. Refer to pages 107-129.

Use the *#include <stdio.h>* preprocessor directive with all disk I/O routines. The *stdio.h* file contains type definitions and function prototypes, making declarations of the routines unnecessary.

When a filename is required as an argument, give the absolute pathname of the file, prefixed by the device name. Valid device names are FD1, FD2, or HRD. See Section 14.2(B) for a discussion of absolute pathnames. The disk filename is required as an argument for the *fopen* routine, which opens a file for reading or writing. From that point on, disk I/O routines relating to that file use the stream pointer, explained above, as input. File maintenance routines, such as *rename* or *remove*, use the filename as input.

> NOTE: A single program can perform disk I/O functions as well as data playback or recording. Disk I/O, however, must be suspended while disk recording (or playback) proceeds, and vice versa. RAM recording, on the other hand, may occur simultaneously with disk I/O operations. Refer to the *start_rcrd_play* and *suspend_rcrd_play* routines in Section 72 for more information on the interaction between disk I/O and recording/playback.

(A) Input/Output-Stream Routines

Several disk I/O routines may be executed on either input or output streams. *fopen* opens an existing disk file for reading or writing, or creates a new file. In each case, a stream is associated with the file until there is a call to *fclose*. *fclose* or a specific call to *fflush* delivers any output written to a stream to the host environment where it will be written to the disk file.

> NOTE: Always include a call to *fclose* in your program to make sure output is written to the file.

Test the end-of-file and error indicators with the *feof* and *ferror* routines, respectively. These same indicators may be cleared via the *clearerr* routine.

The *fseek* and *rewind* routines manipulate the file-position indicator and erase any memory of a character put into the stream via *ungetc*.

The *lock* and *unlock* routines prevent deadlock from occurring when processes on multiple streams try to operate concurrently on a single file.

fopen

Synopsis

```
#include <stdio.h>
extern FILE * fopen(filename_ptr, mode_ptr);
const char * filename_ptr;
const char * mode_ptr;
```

Description

The *fopen* routine opens a file for access. Depending on the open mode, a file can be opened for reading (via an input stream) or for writing (via an output stream). For existing files, this routine also clears the end-of-file and error indicators.

Inputs

The first parameter is a pointer to the file to be opened, represented as the name of the file, placed inside double quotation marks. The filename must be the absolute pathname, prefixed by the device name (HRD, FD1, or FD2).

The second parameter is a pointer to a string (represented as a character inside double quotation marks) which identifies the type of open to be performed. Of the ANSI standard open modes, the following are supported:

r — Open an existing file for reading only. The file-position indicator is located at the start (character zero) of the file.

w — Create a file, or open an existing file, for writing only. For an existing file, truncate its length to zero and discard the contents.

a — Create a file, or open an existing file, for writing only. For an existing file, retain the contents and locate the file-position indicator at the end of the file. Append new data to the end of existing data, unless a call to *fseek* or *rewind* has repositioned the file-position indicator. In this instance, overwrite existing data. (This implementation is different from the ANSI specification which appends new data to the end of existing data regardless of any previous calls to *fseek*.)

rb Currently implemented the same as "r." Use "rb" for the *fseek* routine.

wb Currently implemented the same as "w." Use "wb" for the *fseek* routine.

ab Currently implemented the same as "a." Use "ab" for the *fseek* routine.

Returns

This routine returns a pointer to the stream, with a type definition FILE (defined in the *stdio.h* file).

If the open fails (for example, the file does not exist), zero is returned.

Example

Open a file called "*buff01*" in the */usr* directory on a disk in floppy drive 2. Store the pointer to the stream in *stream_ptr*. Indicate whether or not the open is successful on the prompt line.

```
{
 #include <stdio.h>
 FILE * stream_ptr;
}
LAYER: 1
     STATE: open_a_file
        CONDITIONS: ENTER_STATE
        ACTIONS: PROMPT "Press O to open file.                                    "
        CONDITIONS: KEYBOARD "oO"
        ACTIONS:
        {
         if((stream_ptr = fopen("FD2/usr/buff01", "r")) == 0)
            display_prompt("Cannot open file.                                    ");
         else
            display_prompt("File opened.                                         ");
        }
```

fclose

Synopsis

```
#include <stdio.h>
extern int fclose(stream_ptr);
FILE * stream_ptr;
```

Description

All opened files must be closed. If the disk file to be closed is an input file, then any data remaining in the stream buffer is discarded. If the file is an output file, any data written to the stream is written to the file. (In other words, *fclose* automatically calls *fflush*.) The stream is freed from its association with the disk file, and the disk file is closed.

Inputs

The only parameter is the stream pointer.

Returns

If the stream is successfully closed, zero is returned. If errors are detected, or if the stream is already closed, a non-zero value is returned.

Example

Close the file that was opened in the *fopen* example. Indicate whether or not the close is successful on the prompt line.

```
{
 #include <stdio.h>
 FILE * stream_ptr;
}
LAYER: 1
     STATE: open_and_close_a_file
        CONDITIONS: ENTER_STATE
        ACTIONS: PROMPT "Press O to open file.                                    "
        CONDITIONS: KEYBOARD "oO"
        ACTIONS:
        {
         if((stream_ptr = fopen("FD2/usr/buff01", "r")) == 0)
            display_prompt("Cannot open file.                                    ");
         else
            display_prompt("File opened.                                         ");
        }
        CONDITIONS: KEYBOARD "cC"
        ACTIONS:
        {
         if(fclose(stream_ptr) != 0)
            display_prompt("Either file is already closed, or close cannot be executed.  ");
         else
            display_prompt("File closed.                                         ");
        }
```

fflush

Synopsis

```
#include <stdio.h>
extern int fflush(stream_ptr)
FILE * stream_ptr;
```

Description

If *stream_ptr* points to an output stream, the *fflush* routine causes any unwritten data for that stream to be delivered to the host environment where it will be written to the file. If *stream_ptr* points to an input stream, the *fflush* routine undoes the effect of any preceding *ungetc* operation on the stream.

Inputs

The only parameter is the stream pointer.

Returns

If a write error occurs, non-zero is returned and the error indicator is set.

Example

Assume the X.25 personality package has been loaded in at Layer 2. Whenever you receive a frame type "unknown," write the actual value of the control byte to an output file stream *and* to the disk file.

```
{
 #include <stdio.h>
 FILE * stream_ptr;
 extern volatile const unsigned char rcvd_frame_cntrl_byte_1;
}
LAYER: 2
     STATE: write_then_fflush
        CONDITIONS: ENTER_STATE
        ACTIONS:
        {
         if((stream_ptr = fopen("FD2/usr/frame_unkwn", "a")) == 0)
            display_prompt("Cannot open file.                                    ");
         else
            display_prompt("File opened.                                         ");
         pos_cursor(1,0);
        }
        CONDITIONS: RCV UNKNOWN
        ACTIONS:
        {
         if(fprintf(stream_ptr, "%02x\n ", rcvd_frame_cntrl_byte_1) < 0)
            displayf("Error in printing to stream.                           \n");
         else
            displayf("Print to stream completed.                             \n");
         if(fflush(stream_ptr) != 0)
            display_prompt("Write error.                                         ");
         else
            display_prompt("Write to file completed.  Press C to close file.     ");
        }
        CONDITIONS: KEYBOARD "cC"
        ACTIONS:
        {
         if(fclose(stream_ptr) != 0)
            display_prompt("Either file is already closed, or close cannot be executed.  ");
         else
            display_prompt("File closed.                                         ");
        }
```

feof

Synopsis

```
#include <stdio.h>
extern int feof(stream_ptr);
FILE * stream_ptr;
```

Description

This routine tests the end-of-file indicator for an associated stream.

Inputs

The only parameter is the stream pointer.

Returns

The *feof* routine returns a non-zero value if the end-of-file indicator is set for the stream.

Example

Get a character from a file. If it is not at the end of the file, display it; otherwise prompt with "End of file."

```
{
 #Include <stdio.h>
 FILE * stream_ptr;
 int character;
}
LAYER: 1
     STATE: test_for_eof
        CONDITIONS: ENTER_STATE
        ACTIONS: PROMPT "Press O to open file.                                    "
        CONDITIONS: KEYBOARD "oO"
        ACTIONS:
        {
         if((stream_ptr = fopen("FD2/usr/buff01", "rb")) == 0)
            display_prompt("Cannot open file.                                    ");
         else
            display_prompt("File opened.  Press G to get character.              ");
         pos_cursor(1,0);
        }
        CONDITIONS: KEYBOARD "gG"
        ACTIONS:
        {
         character = fgetc(stream_ptr);
         if(feof(stream_ptr) != 0)
            display_prompt("End of file.  Press C to close file.                 ");
         else
            displayf("%c", character);
        }
        CONDITIONS: KEYBOARD "cC"
        ACTIONS:
        {
         if(fclose(stream_ptr) != 0)
            display_prompt("Either file is already closed, or close cannot be executed.  ");
         else
            display_prompt("File closed.                                         ");
        }
```

ferror

Synopsis

```
#include <stdio.h>
extern int ferror(stream_ptr);
FILE * stream_ptr;
```

Description

This routine tests the error indicator for a stream.

Inputs

The only parameter is the stream pointer.

Returns

The *ferror* routine returns a non-zero value if the error indicator is set for the stream.

Example

Read a file called "*buff01*" from the */usr* directory on the disk in drive 2. If the number of elements read is less than the number designated to be read, determine whether an end-of-file was encountered or a read error occurred.

```
{
 #include <stdio.h>
 FILE * stream_ptr;
 char data [6091];
 size_t n;
}
LAYER: 1
     STATE: read_a_file
        CONDITIONS: ENTER_STATE
        ACTIONS: PROMPT "Press O to open file.
        CONDITIONS: KEYBOARD "oO"
        ACTIONS:
        {
         if((stream_ptr = fopen("FD2/usr/buff01", "r")) == 0)
            display_prompt("Cannot open file.                    ");
         else
            display_prompt("File opened.  Press R to read the file.  ");
        }
        CONDITIONS: KEYBOARD "rR"
        ACTIONS:
        {
         n = fread(data, 1, 6091, stream_ptr);
         if(n != 6091)
           {
           if(ferror(stream_ptr) != 0)
             display_prompt("Read error.                         ");
           else if(feof(stream_ptr) != 0)
             display_prompt("End-of-file encountered.            ");
           }
         else
            displayf("\n%.6091s", data);
         display_prompt("Press C to close the file.              ");
        }
```

```
CONDITIONS: KEYBOARD "cC"
ACTIONS:
{
 if(fclose(stream_ptr) != 0)
    display_prompt("Either file is already closed, or close cannot be executed.    ");
 else
    display_prompt("File closed.                                                   ");
}
```

clearerr

Synopsis

```
#include <stdio.h>
extern void clearerr(stream_ptr);
FILE * stream_ptr;
```

Description

This routine clears the end-of-file and error indicators for a stream. When an error occurs, no further operations are allowed until the error indicators are explicitly cleared. (These indicators are also cleared by a *fopen* or *rewind* operation.)

Inputs

The only parameter is the stream pointer.

Example

If a write error occurs, clear the indicators.

```
{
 #include <stdio.h>
 FILE * stream_ptr;
 int character;
}
LAYER: 1
     STATE: clear_indicators
        CONDITIONS: ENTER_STATE
        ACTIONS: PROMPT "Press O to open file.
        CONDITIONS: KEYBOARD "oO"
        ACTIONS:
        {
         if((stream_ptr = fopen("FD2/usr/buff01", "wb")) == 0)
            display_prompt("Cannot open file.                                  ");
         else
            display_prompt("File opened.  Press P to write character.          ");
        }
```

```
CONDITIONS: KEYBOARD "pP"
ACTIONS:
{
 character = fputc('h', stream_ptr);
 if(character == EOF)
    {
     display_prompt("Write error.  All indicators will be cleared.                ");
     clearerr(stream_ptr);
    }
 else
     display_prompt("Write completed.  Press C to close the file.                 ");
}
CONDITIONS: KEYBOARD "cC"
ACTIONS:
{
 if(fclose(stream_ptr) != 0)
     display_prompt("Either file is already closed, or close cannot be executed.  ");
 else
     display_prompt("File closed.                                                 ");
}
```

fseek

Synopsis

```
#include <stdio.h>
extern int fseek(stream_ptr, bytes, reference);
FILE * stream_ptr;
long int bytes;
int reference;
```

Description

This routine manipulates the file-position indicator, according to the ANSI specification for binary files. Future read operations will be referenced from that point. *fseek* clears the end-of-file indicator and resets the *ungetc* variable.

> NOTE: The ANSI specification for text files is not currently implemented. To ensure proper execution of *fseek* if future releases include the ANSI specification for text files, open files for *fseek* as binary ("rb," "wb," or "ab").

Inputs

The first parameter is the stream pointer.

The second parameter is the number of characters the file-position indicator should be moved from a specified position. A positive number advances the file-position indicator forward in the file; a negative number moves it backward.

The third parameter specifies the location of the file-position indicator. SEEK_SET will move the file-position indicator from the beginning of the file; SEEK_END will move the file-position indicator from the end-of-file; and SEEK_CUR will move the file-position indicator from its current position.

Returns

This routine returns non-zero for an improper request; otherwise it returns zero.

Example

Open a file and move the file-position indicator 4 characters from the beginning of the file. Each time the S key is pressed, move the indicator one character backward from its current position. After 4 executions, the indicator will be back at the beginning of the file.

```
{
 #include <stdio.h>
 FILE * stream_ptr;
 int character;
}
LAYER: 1
     STATE: move_indicator
        CONDITIONS: ENTER_STATE
        ACTIONS: PROMPT "Press O to open file.                                    "
        CONDITIONS: KEYBOARD "oO"
        ACTIONS:
        {
         if((stream_ptr = fopen("FD2/usr/buff01", "rb")) == 0)
            display_prompt("Cannot open file.                                    ");
         else
           {
            display_prompt("File opened.  ");
            pos_cursor(0,14);
            if(fseek(stream_ptr, 4, SEEK_SET) != 0)
              displays("Improper fseek request.                                  ");
            else
              displays("Fseek completed.  Press S to seek new position.          ");
           }
        }
        CONDITIONS: KEYBOARD "sS"
        ACTIONS:
        {
         if(fseek(stream_ptr, -1, SEEK_CUR) != 0)
            display_prompt("Improper fseek request.  Press C to close file.     ");
         else
            display_prompt("Fseek completed.  Press C to close file.            ");
        }
```

```
CONDITIONS: KEYBOARD "cC"
ACTIONS:
{
 if(fclose(stream_ptr) != 0)
    display_prompt("Either file is already closed, or close cannot be executed.      ");
 else
    display_prompt("File closed.                                                     ");
}
```

rewind

Synopsis

```
#include <stdio.h>
extern void rewind(stream_ptr);
FILE * stream_ptr;
```

Description

This routine returns the file-position indicator to the beginning of the file (i.e., it is equivalent to an *fseek* with the number of characters to move set as zero and the specified position SEEK_SET). The rewind operation also clears the end-of-file and error indicators and erases any memory of the character in a previous *ungetc* operation.

Inputs

The only parameter is the stream pointer.

Example

In this example, the first call to *fgetc* following the *rewind* operation will read the first character in the file.

```
{
 #include <stdio.h>
 FILE * stream_ptr;
 int character;
}
LAYER: 1
    STATE: move_indicator
       CONDITIONS: ENTER_STATE
       ACTIONS: PROMPT "Press O to open file.                                        "
       CONDITIONS: KEYBOARD "oO"
       ACTIONS:
       {
        if((stream_ptr = fopen("FD2/usr/buff01", "rb")) == 0)
           display_prompt("Cannot open file.                                         ");
        else
           display_prompt("File opened.  Press S to fseek.                           ");
       }
```

```
CONDITIONS: KEYBOARD "sS"
ACTIONS:
{
  if(fseek(stream_ptr, 4, SEEK_SET) != 0)
     display_prompt("Improper fseek request.                    ");
  else
     display_prompt("Fseek completed.  Press spacebar to rewind.  ");
}
CONDITIONS: KEYBOARD " "
ACTIONS:
{
 rewind(stream_ptr);
 display_prompt("Press G to get a character.                    ");
}
CONDITIONS: KEYBOARD "gG"
ACTIONS:
{
 character =fgetc(stream_ptr);
 display_prompt("Press C to close file.                         ");
}
CONDITIONS: KEYBOARD "cC"
ACTIONS:
{
 if(fclose(stream_ptr) != 0)
     display_prompt("Either file is already closed, or close cannot be executed.  ");
 else
     display_prompt("File closed.                               ");
}
```

lock

Synopsis

```
#include <stdio.h>
extern void lock(lock_variable_ptr);
int * lock_variable_ptr;
```

Description

The *lock* routine implements a lock using the integer variable pointed to by the routine parameter. If the lock variable is currently locked, the task goes to sleep. When an unlock on the same variable occurs (within an independent task), the task invoking the lock function will attempt to claim the lock. If successful, the task is executed; otherwise, it goes back to sleep until the next unlock.

NOTE: If locking is used at any place in the program, all related or possibly concurrent routines must also use the locking functions.

NOTE: The lock variable should always be defined as a global integer, never as local to a function. The lock variable should never be altered by the user program or deadlock can occur. Deadlock also results if the lock is invoked twice within the same task without an intervening unlock.

Inputs

The only parameter is a pointer to the lock variable.

Example

Two tasks concurrently write to their own file streams. The file streams are local to the routine *write_fox*, making them independent of each other even though both are referenced by *stream_ptr*. During the *fclose* operation (which automatically calls *fflush*), however, both tasks need to write to the same file. The locking routines ensure that the writes to the file occur sequentially, not concurrently.

```
{
#include <stdio.h>
const char data [] = "((FOX))\n";
int key;
void write_fox()
 {
  FILE * stream_ptr;
  size_t n;
  lock(&key);
  if((stream_ptr = fopen("FD2/usr/buff01", "a")) == 0)
    display_prompt("Cannot open file.                                    ");
  else
    display_prompt("File opened.                                         ");
  n = fwrite(data, 1, sizeof(data)-1, stream_ptr);
  pos_cursor(1,0);
  if(n != (sizeof(data)-1))
    displayf("Write error.                                             \n");
  else
    displayf("Write completed.                                         \n");
  if(fclose(stream_ptr) != 0)
    displayf("Either file is already closed, or close cannot be executed.  ");
  else
    displayf("File closed.                                               ");
  unlock(&key);
 }
}
LAYER: 1
  TEST: a
    STATE: write_and_signal
      CONDITIONS: RECEIVE STRING "THE QUICK BROWN FOX"
      ACTIONS: SIGNAL xyz
      {
       write_fox();
      }
```

```
TEST: b
  STATE: write_only
    CONDITIONS: ON_SIGNAL xyz
    ACTIONS:
    {
     write_fox();
    }
```

unlock

Synopsis

```
#include <stdio.h>
extern void unlock(lock_variable_ptr);
int * lock_variable_ptr;
```

Description

The *unlock* routine implements the inverse of the *lock* routine using the same integer variable. Sleeping tasks will be woken up to retry their attempt to claim the lock. One will succeed, and the rest will go back to sleep. See also *lock* routine.

Inputs

The only parameter is a pointer to the lock variable.

Example

See *lock* routine.

(B) Input-Stream Routines

The following routines are valid for input streams only. An attempt to apply them to output streams results in a read error. The error indicator for the input stream will be set.

Three routines read characters from the input stream. The *fread* and *fgets* routines transfer a specified number of characters from the stream buffer into a user-defined array. *fgetc* reads the next character from the input stream. The *ungetc* routine temporarily forces a designated character back into the input stream.

fread

Synopsis

```
#include <stdio.h>
extern size_t fread(data_ptr, size, number, stream_ptr);
void * data_ptr;
size_t size;
size_t number;
FILE * stream_ptr;
```

Description

This routine reads elements from the input-stream buffer and puts them into a user-defined buffer. The file-position indicator is advanced by the number of characters successfully read. The *fread* routine can read a file whose elements are more than eight bits each, 16-bit *shorts* or 32-bit *longs*, for example. The *fgets* routine is similar to *fread*. *fgets*, however, reads only 8-bit characters. The primary use of *fread* is to read the entire contents of a file, whereas the primary purpose of *fgets* is to read from a file one line at a time.

Inputs

The first parameter is a pointer to an array in which the incoming data should be placed.

The second parameter is the number of bytes in each element to be read. If the value of this parameter is zero, the contents of the array and the stream remain unchanged.

The third parameter is the number of elements to be read. If the value of this parameter is zero, the contents of the array and the stream remain unchanged.

The fourth parameter is the stream pointer.

Returns

The *fread* routine returns the total number of elements read. If the number of elements read is less than the number of elements designated to be read, an end-of-file has been encountered or a read error has occurred. Use the *feof* and *ferror* routines to distinguish an end-of-file from a read error. If an error occurs, the location of the file-position indicator is indeterminate.

Example

Read in a file called "*buff01*" from the */usr* directory on the disk in drive 2 and display it on the Program Trace screen. (See Section 64.4 for information on using trace buffers in C.) Determine the size of the array *data* from the file size indicated on the File Maintenance screen.

```
{
 #include <trace_buf.h>
 #include <stdio.h>
 FILE * stream_ptr;
 char data [6091];
 size_t n;
 extern struct trace_buf prog_trbuf;
}
```

```
LAYER: 1
    STATE: read_a_file
        CONDITIONS: ENTER_STATE
        ACTIONS: PROMPT "Press O to open file.                                    "
        CONDITIONS: KEYBOARD "oO"
        ACTIONS:
        {
         if((stream_ptr = fopen("FD2/usr/buff01", "r")) == 0)
            display_prompt("Cannot open file.                                    ");
         else
            display_prompt("File opened.  Press R to read the file.              ");
        }
        CONDITIONS: KEYBOARD "rR"
        ACTIONS:
        {
         n = fread(data, 1, 6091, stream_ptr);
         if(n != 6091)
            display_prompt("Either a read error has occurred, or an EOF has been
                  encountered.    ");
         else
           {
            tracef(&prog_trbuf, "%.6091s", data);
            display_prompt("Press C to close the file.                           ");
           }
        }
        CONDITIONS: KEYBOARD "cC"
        ACTIONS:
        {
         if(fclose(stream_ptr) != 0)
            display_prompt("Either file is already closed, or close cannot be executed.  ");
         else
            display_prompt("File closed.                                         ");
        }
```

fgets

Synopsis

```
#include <stdio.h>
extern char * fgets(string_ptr, max_number, stream_ptr);
char * string_ptr;
int max_number;
FILE * stream_ptr;
```

Description

This routine gets at the most one less than the specified number of characters from an input stream and puts them in an array. If an EOF, newline, or null is encountered in the stream, no more characters will be read, even if the specified number of characters has not yet been read. The newline will be retained. A terminating null character is written after the last character read into the array. The file-position indicator is advanced by the number of characters successfully read. The *fgets* routine is similar to *fread*. The *fread* routine can read a file

whose elements are more than eight bits each, 16-bit *shorts* or 32-bit *longs*, for example. *fgets*, however, reads only 8-bit characters. The primary use of *fgets* is to read from a file one line at a time.

Inputs

The first parameter is a pointer to the array into which the characters will be put.

The second parameter is the maximum number of characters (minus one) to be read.

The third parameter is the stream pointer.

Returns

If the routine is successful, a pointer to the array is returned. If end-of-file is encountered before any characters have been read into the array or if a read error occurs, a null pointer is returned. The contents of the array are indeterminate when a read error occurs.

Example

Five characters, at the most, from a disk file will be put into an array called *data* and displayed on the screen.

```
{
 #include <stdio.h>
 FILE * stream_ptr;
 char data [10];
}
LAYER: 1
     STATE: read_characters
        CONDITIONS: ENTER_STATE
        ACTIONS: PROMPT "Press O to open file.                                    "
        CONDITIONS: KEYBOARD "oO"
        ACTIONS:
        {
         if((stream_ptr = fopen("FD2/usr/buff01", "r")) == 0)
            display_prompt("Cannot open file.                                    ");
         else
            display_prompt("File opened.  Press G to get string.                 ");
        }
        CONDITIONS: KEYBOARD "gG"
        ACTIONS:
        {
         fgets(data, 6, stream_ptr);
         displayf("\n%.6s", data);
         display_prompt("Press C to close the file.                              ");
        }
```

```
CONDITIONS: KEYBOARD "cC"
ACTIONS:
{
 if(fclose(stream_ptr) != 0)
    display_prompt("Either file is already closed, or close cannot be executed.    ");
 else
    display_prompt("File closed.                                                   ");
}
```

fgetc

Synopsis

```
#include <stdio.h>
extern int fgetc(stream_ptr);
FILE * stream_ptr;
```

Description

The *fgetc* routine gets the next character (if present) from the input stream. The character is an *unsigned char* cast to an *int* (stored in the least-significant byte of the *int*). The file-position indicator advances by one character.

Inputs

The only parameter is the stream pointer.

Returns

This routine returns the next character in the input stream. EOF is returned if an end-of-file is encountered or if a read error occurs. The *stdio.h* file defines the macro EOF as -1. Use the *feof* and *ferror* routines to determine the reason for a returned EOF.

Example

In the following example, open an input file for reading. Each time the [g] key is pressed, display the next character in the file.

```
{
 #include <stdio.h>
 FILE * stream_ptr;
 int character, end;
}
LAYER: 1
    STATE: get_next_character
       CONDITIONS: ENTER_STATE
       ACTIONS: PROMPT "Press O to open file.
       CONDITIONS: KEYBOARD "oO"
       ACTIONS:
       {
        if((stream_ptr = fopen("FD2/usr/buff01", "r")) == 0)
           display_prompt("Cannot open file.                                   ");
        else
           display_prompt("File opened.  Press G to get a character.           ");
        displayf("\n");
       }
```

```
CONDITIONS: KEYBOARD "gG"
ACTIONS:
{
 character = fgetc(stream_ptr);
 if(character == EOF)
    {
    end = feof(stream_ptr);
    if(end != 0)
      display_prompt("EOF encountered.                                    ");
    else
      display_prompt("Read error.                                         ");
    }
 else
    displayf("%c", character);
}
CONDITIONS: KEYBOARD "cC"
ACTIONS:
{
 if(fclose(stream_ptr) != 0)
    display_prompt("Either file is already closed, or close cannot be executed.  ");
 else
    display_prompt("File closed.                                        ");
}
```

ungetc

Synopsis

```
#include <stdio.h>
extern int ungetc(character, stream_ptr);
int character;
FILE * stream_ptr;
```

Description

This routine temporarily forces a specified character into a variable associated with the input stream, overwriting the previous *ungetc* variable. The routine does not affect the location of the file-position indicator. The next *fgetc* will read the *ungetc* variable, not the stream. An intervening *fflush*, *fseek*, or *rewind* erases memory of the character. If the *ungetc* function is called too many times on the same stream without an intervening read, *fflush*, *fseek*, or *rewind* operation on that stream, the operation may fail. *Ungetc* also clears the end-of-file indicator.

Inputs

The first parameter is the character to be put into the input stream.

The second parameter is the stream pointer.

Returns

This routine returns the specified character. If the operation fails, EOF is returned and the input stream remains unchanged. It will fail if the values of the specified character and the macro EOF are equal.

Example

Read a character from the stream. Press the U key when you want to return the last character read to the stream. The next call to *fgetc* will read the returned character.

```
{
 #include <stdio.h>
 FILE * stream_ptr;
 int character;
}
LAYER: 1
     STATE: get_next_character
        CONDITIONS: ENTER_STATE
        ACTIONS: PROMPT "Press O to open file.
        CONDITIONS: KEYBOARD "oO"
        ACTIONS:
        {
         if((stream_ptr = fopen("FD2/usr/buff01", "r")) == 0)
            display_prompt("Cannot open file.                                    ");
         else
            display_prompt("File opened.  Press G to get a character.            ");
        }
        CONDITIONS: KEYBOARD "gG"
        ACTIONS:
        {
         character = fgetc(stream_ptr);
         if(character == EOF)
            display_prompt("End of file or read error.                           ");
         else
            {
            pos_cursor(0,0);
            displayf("character = %c   Press U to return character to stream.", character);
            }
        }
        CONDITIONS: KEYBOARD "uU"
        ACTIONS:
        {
         if((ungetc(character, stream_ptr)) == EOF)
            display_prompt("Character not returned.                              ");
         else
            display_prompt("Character returned.                                  ");
        }
        CONDITIONS: KEYBOARD "cC"
        ACTIONS:
        {
         if(fclose(stream_ptr) != 0)
            display_prompt("Either file is already closed, or close cannot be executed.  ");
         else
            display_prompt("File closed.                                         ");
        }
```

(C) Output-Stream Routines

The following routines are valid for output streams only. An attempt to apply them to input streams will result in a write error. The error indicator for the output stream will be set.

Four routines write to output streams. The *fwrite* and *fputs* routines transfer a specified number of characters from a user-defined array into the stream buffer. *fputc* writes a character to the next empty byte in an output-stream buffer. *fprintf* writes formatted output to an output stream similar to the way *displayf* writes output to the Display Window.

fwrite

Synopsis

```
#include <stdio.h>
extern size_t fwrite(output_ptr, size, number, stream_ptr);
const void * output_ptr;
size_t size;
size_t number;
FILE * stream_ptr;
```

Description

This routine writes elements from a user-defined array to the output-stream buffer. The file-position indicator is advanced by the number of characters successfully written.

Inputs

The first parameter is a pointer to an array from which the data should be taken. Declare it as *const* if it is read-only. In cases where the array will be written to, as in the example below, do not include *const* as part of the declaration.

The second parameter is the number of bytes in each element to be written.

The third parameter is the number of elements to be written.

The fourth parameter is the stream pointer.

Returns

The *fwrite* routine returns the total number of elements written. If the number of elements written is less than the number of elements designated to be written, a write error has occurred. If an error occurs, the location of the file-position indicator is indeterminate.

Example

Read the contents of a file, and write them to a new file.

```
{
 #include <stdio.h>
 FILE * read_stream;
 FILE * write_stream;
 char output [6091];
 size_t n;
}
LAYER: 1
     STATE: write_to_a_file
        CONDITIONS: ENTER_STATE
        ACTIONS: PROMPT "Press O to open files.
        CONDITIONS: KEYBOARD "oO"
        ACTIONS:
        {
         if((read_stream = fopen("FD2/usr/buff01", "r")) == 0)
            {
            display_prompt("Cannot open buff01.  ");
            pos_cursor(0,21);
            }
         else
            {
            display_prompt("Buff01 opened.  ");
            pos_cursor(0,16);
            }
         if((write_stream = fopen("FD2/usr/new_file", "w")) == 0)
            displays("Cannot open new_file.                                    ");
         else
            displays("New_file opened.  Press R to read buff01.                ");
        }
        CONDITIONS: KEYBOARD "rR"
        ACTIONS:
        {
         n = fread(output, 1, 6091, read_stream);
         if(n != 6091)
            display_prompt("Either a read error has occurred, or an EOF has been
                 encountered.     ");
         else
            display_prompt("Press W to write to new_file.                      ");
        }
        CONDITIONS: KEYBOARD "wW"
        ACTIONS:
        {
         n = fwrite(output, 1, 6091, write_stream);
         if(n != 6091)
            display_prompt("Write error.  Press C to close files.              ");
         else
            display_prompt("Write completed.  Press C to close files.          ");
        }
```

```
CONDITIONS: KEYBOARD "cC"
ACTIONS:
{
 if(fclose(read_stream) != 0)
    {
    display_prompt("Either buff01 is already closed, or close cannot be executed.  ");
    pos_cursor(0,0);
    }
 else
    {
    display_prompt("Buff01 closed.  ");
    pos_cursor(0,16);
    }
 if(fclose(write_stream) != 0)
    displays("Either new_file is already closed, or close cannot be executed.       ");
 else
    displays("New_file closed.                                                      ");
}
```

fputs

Synopsis

```
#include <stdio.h>
extern int fputs(string_ptr, stream_ptr);
const char * string_ptr;
FILE * stream_ptr;
```

Description

This routine writes a string of characters from an array, excluding the terminating null character, to the output stream. The file-position indicator is advanced by the number of characters successfully written.

Inputs

The first parameter is a pointer to the string to be written.

The second parameter is the stream pointer.

Returns

This routine returns zero if it is successful; it returns a non-zero value if a write error occurs.

Example

Write a fox message at the end of existing data in a file.

```
{
 #include <stdio.h>
 FILE * stream_ptr;
 char data [] = "((FOX))\n";
}
```

```
LAYER: 1
    STATE: write_a_string
        CONDITIONS: ENTER_STATE
        ACTIONS: PROMPT "Press O to open file.
        CONDITIONS: KEYBOARD "oO"
        ACTIONS:
        {
         if((stream_ptr = fopen("FD2!usr!buff01", "a")) == 0)
            display_prompt("Cannot open file.                          ");
         else
            display_prompt("File opened.  Press P to write string.     ");
        }
        CONDITIONS: KEYBOARD "pP"
        ACTIONS:
        {
         if(fputs(data, stream_ptr) != 0)
            display_prompt("Write error.  Press C to close file.       ");
         else
            display_prompt("Write completed.  Press C to close file.   ");
        }
        CONDITIONS: KEYBOARD "cC"
        ACTIONS:
        {
         if(fclose(stream_ptr) != 0)
            display_prompt("Either file is already closed, or close cannot be executed.  ");
         else
            display_prompt("File closed.                               ");
        }
```

fputc

Synopsis

```
#include <stdio.h>
extern int fputc(character, stream_ptr);
int character;
FILE * stream_ptr;
```

Description

This routine writes a given character (cast to an *unsigned char*) to an output stream. The file-position indicator advances one character.

Inputs

The first parameter is the character to be written to the output stream. It may be given as a hexadecimal, octal, or decimal constant; as an alphanumeric constant inside single quotes; or as a variable. A hexadecimal value must be preceded by the prefix 0x or 0X; an octal value must be preceded by the prefix 0. If no prefix appears before the input, the number is assumed to be decimal.

The second parameter is the stream pointer.

Returns

If the character is successfully written to the output stream, the routine returns that character. If a write error occurs, EOF is returned and the error indicator is set.

Example

Open the named file. If the file does not already exist, create it. If it does exist, truncate its length to zero, thereby deleting its contents. Put the character read from the input stream pointed to by *read_stream* into the output stream pointed to by *write_stream*. This example is similar to the one given for *fwrite*, except that in this case, each time the P key is pressed, only one character is copied, rather than the entire file.

```
{
 #include <stdio.h>
 FILE * read_stream;
 FILE * write_stream;
 int character;
}
LAYER: 1
     STATE: copy_a_character
        CONDITIONS: ENTER_STATE
        ACTIONS: PROMPT "Press O to open files.
        CONDITIONS: KEYBOARD "oO"
        ACTIONS:
        {
         if((read_stream = fopen("FD2/usr/buff01", "r")) == 0)
            {
            display_prompt("Cannot open buff01.  ");
            pos_cursor(0,21);
            }
         else
            {
            display_prompt("Buff01 opened.  ");
            pos_cursor(0,16);
            }
         if((write_stream = fopen("FD2/usr/buff01_copy", "w")) == 0)
            displays("Cannot open buff01_copy.                                    ");
         else
            displays("Buff01_copy opened.  Press P to copy a character.           ");
        }
        CONDITIONS: KEYBOARD "pP"
        ACTIONS:
        {
         character = fgetc(read_stream);
         if(character == EOF)
            {
            if(feof(read_stream) != 0)
              display_prompt("EOF encountered.  Press C to close files.           ");
            else
              display_prompt("Read error.  Press C to close files.                ");
            }
```

```
    else
        fputc(character, write_stream);
}
CONDITIONS: KEYBOARD "oC"
ACTIONS:
{
 if(fclose(read_stream) != 0)
    {
    display_prompt("Either buff01 is already closed, or close cannot be executed.  ");
    pos_cursor(0,0);
    }
 else
    {
    display_prompt("Buff01 closed.  ");
    pos_cursor(0,16);
    }
 if(fclose(write_stream) != 0)
    displayf("Either buff01_copy is already closed, or close cannot be executed.    ");
 else
    display("Buff01_copy closed.                                                     ");
}
```

fprintf

Synopsis

```
#include <stdio.h>
extern int fprintf(stream_ptr, format_ptr, ...);
FILE * stream_ptr;
char * format_ptr;
```

Description

The *fprintf* routine is similar to the *sprintf* routine, except that *fprintf* writes output to an output *stream*, while *sprintf* writes output to an *array*. The output is under control of the string pointed to by *format_ptr* that specifies how subsequent arguments are converted for output. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but otherwise ignored. The *fprintf* routine returns when the end of the format string is encountered. (*Sprintf* is documented in Section 67.3.)

Inputs

The first parameter is the stream pointer.

The second parameter points to the format string composed of zero or more directives: ordinary characters (not %), which are copied unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments. Each conversion specification is introduced by the character %. After the %, the following appear in sequence:

- Zero or more *flags* that modify the meaning of the conversion specification. The flag characters and their meanings are:

  - The result of the conversion will be left-justified within the field.

  + The result of a signed conversion will always begin with a plus or minus sign.

  space If the first character of a signed conversion is not a sign, a space will be prepended to the result. If the *space* and + flags both appear, the *space* flag will be ignored.

  # The result is to be converted to an "alternate form." For d, i, c, and s conversions, the flag has no effect. For o conversion, it increases the precision to force the first digit of the result to be a zero. For x (or X) conversion, a nonzero result will have 0x (or 0X) prepended to it.

- An optional decimal integer specifying a minimum *field width*. If the converted value has fewer characters than the field width, it will be padded on the left (or right, if the left adjustment flag, described above, has been given) to the field width. The padding is with spaces unless the field width integer starts with a zero, in which case the padding is with zeros.

- An optional *precision* that gives the minimum number of digits to appear for the d, i, o, u, x, and X conversions or the maximum number of characters to be written from an array in an s conversion. The precision takes the form of a period (.) followed by an optional decimal integer; if the integer is omitted, it is treated as zero. The amount of padding specified by the precision overrides that specified by the field width.

- An optional h specifying that a following d, i, o, u, x, or X conversion specifier applies to a *short int* or *unsigned short int* argument (the argument will have been promoted according to the integral promotions, and its value shall be converted to *short int* or *unsigned short int* before printing); or an optional l specifying that a following d, i, o, u, x, or X conversion specifier applies to a *long int* or *unsigned long int* argument. If an h or l appears with any other conversion specifier, it is ignored.

- A character that specifies the type of *conversion* to be applied. (Special AR extensions have been added.) The conversion specifiers and their meanings are:

d, i, o, u, x, X

The *int* argument is converted to signed decimal (d or i), unsigned octal (o), unsigned decimal (u), or unsigned hexadecimal notation (x or X); the letters abcdef are used for x conversion and the letters ABCDEF for X conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no characters.

c The *int* argument is converted to an *unsigned char*, and the resulting character is written.

s The argument shall be a pointer to a null-terminated array of 8-bit *chars*. Characters from the string are written up to (but not including) the terminating null character: if the precision is specified, no more than that many characters are written. The string may be an array into which output was written via the *sprintf* routine.

p The argument shall be a pointer to void. The value of the pointer is converted to a sequence of printable characters, in this format: *0000:0000*. There are always exactly 4 digits to the right of the colon. The number of digits to the left of the colon is determined by the pointer's value and the precision specified. Use this conversion to display 80286 memory addresses. The 16-bit segment number will appear to the left of the colon and the 16-bit offset to the right.

% A % is written. No argument is converted.

\n Writes hexadecimal 0A, the ASCII linefeed character. No argument is converted.

If a conversion specification is invalid, the behavior is undefined.

If any argument is or points to an aggregate (except for an array of characters using *%s* conversion or any pointer using *%p* conversion), the behavior is undefined.

In no case does a nonexistent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is expanded to contain the conversion result.

Returns

This routine returns the number of characters written, or a negative value if an output error occurred.

Example

Assume the X.25 personality package has been loaded in at Layer 2. When an unknown frame is received, copy the actual value of the control byte to an output stream.

```
{
 #include <stdio.h>
 FILE * stream_ptr;
 extern volatile const unsigned char rcvd_frame_cntrl_byte_1;
}
LAYER: 2
     STATE: save_unknowns
        CONDITIONS: ENTER_STATE
        ACTIONS: PROMPT "Press O to open file.                                    "
        CONDITIONS: ENTER_STATE
        ACTIONS:
        {
         if((stream_ptr = fopen("FD2/usr/frame_unkwn", "w")) == 0)
            display_prompt("Cannot open file.                                    ");
         else
            display_prompt("File opened.                                         ");
        }
        CONDITIONS: RCV UNKNOWN
        ACTIONS:
        {
         if(fprintf(stream_ptr, "%02x\n", rcvd_frame_cntrl_byte_1) < 0)
            display_prompt("Error in printing to stream.                         ");
         else
            display_prompt("Print to stream completed.  Press C to close file.   ");
        }
        CONDITIONS: KEYBOARD "cC"
        ACTIONS:
        {
         if(fclose(stream_ptr) != 0)
            display_prompt("Either file is already closed, or close cannot be executed.  ");
         else
            display_prompt("File closed.                                         ");
        }
```

(D) File Maintenance Routines

rename

Synopsis

```
#include <stdio.h>
extern int rename(oldfile_ptr, newfile_ptr);
const char * oldfile_ptr;
const char * newfile_ptr;
```

Description

This routine renames a specified file. A file can only be renamed if it resides on the active disk, indicated on the Current Directory line of the File Maintenance screen. Renaming an open file does not affect subsequent disk I/O operations on the stream. The stream is still associated with the same file, even though the filename has changed.

Inputs

The first parameter is a pointer to a string, the current name of the file. Give the absolute pathname of the file, prefixed by the device name (HRD, FD1, or FD2).

The second parameter is a pointer to a string, the new name to be given to the file. Give the absolute pathname of the file, prefixed by the device name.

Returns

If the rename operation succeeds, zero is returned. If it fails, a non-zero value is returned. If the renaming fails, the file will still be known by its original name.

Example

Change the name of a file from *old* to *backup*. Prompt whether or not the rename operation was successful.

```
{
 #include <stdio.h>
}
LAYER: 1
     STATE: rename
        CONDITIONS: ENTER_STATE
        ACTIONS: PROMPT "Press spacebar to rename file.                    "
        CONDITIONS: KEYBOARD " "
        ACTIONS:
        {
         if(rename("FD1/usr/old", "FD1/usr/backup") != 0)
            display_prompt("Rename failed.                                ");
         else
            display_prompt("File has been renamed.                        ");
        }
```

remove

Synopsis

```
#include <stdio.h>
extern int remove(file_ptr);
const char * file_ptr;
```

Description

This routine removes the named file from the disk. The file must be closed in order for the remove operation to succeed. Subsequent attempts to open the file will fail. Empty directories may also be removed with this routine.

Inputs

The only input is a pointer to a string, i.e., the filename. It must be the absolute pathname, prefixed by the device name (HRD, FD1, or FD2).

Returns

Zero is returned if the file is removed; non-zero if it is not (for example, the file does not exist in the specified location).

Example

Remove file *oldfile* from the */usr* directory on the disk in floppy drive 1. Prompt whether or not the remove operation was successful.

```
{
 #include <stdio.h>
}
LAYER: 1
     STATE: delete_a_file
        CONDITIONS: ENTER_STATE
        ACTIONS: PROMPT "Press D to delete file.                                    "
        CONDITIONS: KEYBOARD "dD"
        ACTIONS:
        {
         if(remove("FD1/usr/oldfile") != 0)
            display_prompt("File has not been deleted.                              ");
         else
            display_prompt("File deleted.                                           ");
        }
```

mkdir

Synopsis

```
#include <stdio.h>
extern int mkdir(directory_ptr);
char * directory_ptr;
```

Description

This routine creates a directory.

Inputs

The only parameter is a pointer to a string, i.e., the name of the directory to be created. The absolute pathname must be used, prefixed by the device name (FD1, FD2, or HRD).

Returns

If the directory is created, zero is returned; otherwise, a non-zero value is returned.

Example

Create a sub-directory called *disk_i_o* in the */usr* directory on the disk in drive 2.

```
{
 #include <stdio.h>
}
```

```
LAYER: 1
    STATE: make_directory
        CONDITIONS: ENTER_STATE
        ACTIONS: PROMPT "Press M to make a directory.
        CONDITIONS: KEYBOARD "mM"
        ACTIONS:
        {
         if(mkdir("FD2/usr/disk_i_o") != 0)
            display_prompt("Directory not created.                    ");
         else
            display_prompt("Directory created.                        ");
        }
```

_set_file_type

Synopsis

```
#include <stdio.h>
extern int _set_file_type(pathname_ptr, type_buff_ptr);
char * pathname_ptr;
char * type_buff_ptr;
```

Description

This routine determines the type identification of a specified file on the File Maintenance screen. If a file is created by a "w" or "a" open mode and a file type is not specified with the *_set_file_type* routine, it will be designated as an ASCII file. Note, however, that it is the file's contents, not its label, that determines which functions are valid for the file (see example).

Inputs

The first parameter is a pointer to a string, the name of the file. The filename must be the absolute pathname, prefixed by the device name (HRD, FD1, or FD2).

The second parameter is a pointer to a string, the file type. The type may be any of the following (upper or lower case is acceptable):

SYS	System
DIR	Directory
PRGM	Program
SETUP	Setup
OBJ	Object code
LOBJ	Linkable object
LPGM	Linkable program
ASCII	ASCII
BITIM	Bit-image data
CHDAT	Character data

Returns

If the operation succeeds, the routine returns zero; otherwise, it returns a non-zero value.

Example

The following example is almost the same one used for *fwrite*: read the contents of a program file and write them to a new file. The difference is that *new_file* is set to be a program file. In the *fwrite* example, the type designation in the file directory would default to "ASCII." It would still load and run as a program file, however, since the file's contents, not its type label, determine which operations are valid.

```
{
 #include <stdio.h>
 FILE * read_stream;
 FILE * write_stream;
 char output [6091];
 size_t n;
}
LAYER: 1
     STATE: write_to_a_file
        CONDITIONS: ENTER_STATE
        ACTIONS: PROMPT "Press O to open files.                                    "
        CONDITIONS: KEYBOARD "oO"
        ACTIONS:
        {
         if((read_stream = fopen("FD2/usr/buff01", "r")) == 0)
            {
            display_prompt("Cannot open buff01.  ");
            pos_cursor(0,21);
            }
         else
            {
            display_prompt("Buff01 opened.  ");
            pos_cursor(0,16);
            }
         if((write_stream = fopen("FD2/usr/new_file", "w")) == 0)
            displays("Cannot open new_file.                                        ");
         else
            displays("New_file opened.  Press "sS" to set the file type.           ");
        }
        CONDITIONS: KEYBOARD "sS"
        ACTIONS:
        {
         if(_set_file_type("FD2/usr/new_file", "PRGM") != 0)
            display_prompt("File type not set.  Press R to read buff01.            ");
         else
            display_prompt("File type set.  Press R to read buff01.                ");
        }
```

```
CONDITIONS: KEYBOARD "rR"
ACTIONS:
{
 n = fread(output, 1, 6091, read_stream);
 if(n != 6091)
    display_prompt("Either a read error has occurred, or an EOF has been
         encountered.     ");
 else
    display_prompt("Press W to write to new_file.                          ");
}
CONDITIONS: KEYBOARD "wW"
ACTIONS:
{
 n = fwrite(output, 1, 6091, write_stream);
 if(n != 6091)
    display_prompt("Write error.  Press C to close files.                  ");
 else
    display_prompt("Write completed.  Press C to close files.              ");
}
CONDITIONS: KEYBOARD "cC"
ACTIONS:
{
 if(fclose(read_stream) != 0)
    {
    display_prompt("Either buff01 is already closed, or close cannot be executed.  ");
    pos_cursor(0,0);
    }
 else
    {
    display_prompt("Buff01 closed.   ");
    pos_cursor(0,16);
    }
 if(fclose(write_stream) != 0)
    displays("Either new_file is already closed, or close cannot be executed.      ");
 else
    displays("New file closed.                                             ");
}
```

_get_file_type

Synopsis

```
#include <stdio.h>
extern int _get_file_type(pathname_ptr, type_buff_ptr);
char * pathname_ptr;
char * type_buff_ptr;
```

Description

This routine determines the type of a specified file.

Inputs

The first parameter is a pointer to a string, the name of the file. The filename must be the absolute pathname, prefixed by the device name (HRD, FD1, or FD2).

The second parameter is a pointer to an array in which the file type should be written. See *_set_file_type* for the different file types.

Returns

If the operation succeeds, the routine returns zero; otherwise, it returns a non-zero value.

Example

```
{
 #include <stdio.h>
 FILE * stream_ptr;
 char type [8];
}
LAYER: 1
     STATE: find_type
        CONDITIONS: ENTER_STATE
        ACTIONS: PROMPT "Press G to get file type.                                    "
        CONDITIONS: KEYBOARD "gG"
        ACTIONS:
        {
         if(_get_file_type("FD2/usr/new_file", &type[0]) != 0)
            display_prompt("File type not found.                                    ");
         else
            displayf("File type=%s.                                    ", type);
        }
```

69 Status

The structures and variables referenced in this section provide information about the current status of the programmer's INTERVIEW. This information must be accessed via C coding on the Protocol Spreadsheet since these structures and variables have no softkey equivalents.

69.1 Unit Configuration

Two structures presented in Table 69-1 may be accessed by the user to identify current features of the INTERVIEW. *unit_setup* variables reflect current Line Setup menu and FEB tick-rate selections. *unit_config* variables contain information about the user's INTERVIEW hardware and software.

69.2 Current Display Mode

The variables *display_screen_changed*, *crnt_display_screen*, and *prev_display_screen* track movement via softkey from one display screen to another. These variables also indicate transitions between Run mode and Freeze mode. They are documented in Section 64.1.

Table 69-1
Status Structures

Type	Variable	Value (hex/decimal)	Meaning
Structure Name:	**unit_setup**		Structure containing Line Setup and FEB tick-rate selections. Declared as type *extern struct*. Reference member variables of the structure as follows: *unit_setup.speed_dce*.
unsigned long	speed_dce		If *Clock Source* selection is *Internal*, this variable has *Speed* value entered on Line Setup. If *Clock Source* is *External*, this variable has *DCE* speed indicated under *Clock Source: Internal Split*.
unsigned long	speed_dte		If *Clock Source* selection is *Internal*, this variable has *Speed* value entered on Line Setup. If *Clock Source* is *External*, this variable has *DTE* speed indicated under *Clock Source: Internal Split*.
unsigned long	usec_per_tick		tick rate selected on FEB Setup
		a/10	10 useo
		64/100	100 usec
		3e8/1000	1 msec
		2710/10000	10 msec
		186a0/100000	1000 msec
		f4240/1000000	1 sec
unsigned char	bit_order_polarity	0	normal
		1	normal-inverse
		2	reverse-normal
		3	reverse-inverse
unsigned char	bits_per_byte	*5-8*	
unsigned char	clocking_type	0	internal
		1	external
		2	internal-split
unsigned char	data_source	0	disk
		1	line
unsigned char	format	0	sync
		1	bop
		2	async
		3	isoc
unsigned char	mode	0	automonitor
		1	monitor
		2	emulate dce
		3	emulate dte
unsigned char	parity	0	none
		1	even
		2	odd
		3	mark
		4	space
unsigned char	code_name [13]		ASCII, EBCDIC, etc.

Table 69-1 (continued)

Type	Variable	Value (hex/decimal)	Meaning
Structure Name:	**unit_config**		Structure containing unit configuration. Declared as type *extern struct*. Reference member variables of the structure as follows: *unit_config.floppy_exists_mask.*
unsigned char	floppy_exists_mask	1 2	floppy1 floppy2†
unsigned char	hard_disk	0 1	not present present
unsigned char	test_board	0 1	not present present
unsigned char	mux	0 1	not present present
unsigned char	modem	0 1	not present present
unsigned char	num_mpms	*0-4*	number of MPM boards present
struct mpm_info	mpm [4]		array of structures. Each element in the array is an instance of the structure *mpm_info* and corresponds to one of four MPM boards which may be present. Reference member variables of the structure elements in the array as follows: *unit_config.mpm[0].present.*
unsigned char	cpm_rev	0, 7f/0, 127 *1-1f/1-31* *20-7e/32-126*	original CPM board *TURBO*-compatible CPM board 4-Mbyte, *TURBO*-compatible CPM board
unsigned char	gbm_rev	0, ff/0, 256	original GBM board
unsigned char	pcm_rev	0, ff/0, 256 1	original PCM board 44-Mbyte hard disk compatible PCM board
unsigned char	modem_rev		reserved
unsigned char	mux_rev		reserved
unsigned char	tlm_type	f0/240 f1/241 f2/242 f3/243 f4/244 f5/245 *f6-fb/246-251* fc/252 fd/253 fe/254 ff/255	RS-232 X.21 V.35 RS-449 expansion adaptor RC-8245 reserved ISDN G.703 T1 none
unsigned long	last_ram_cpm		the value of this variable plus one yields the CPM memory size (in bytes)

(unit_config continued on next page)

† If *(unit_config.floppy_exists_mask & value)* == *value*, the drive is present.
For example, if *(unit_config.floppy_exists_mask & 2)* == 2, floppy drive 2 is present.

Table 69-1 (continued)

Type	Variable	Value (hex/decimal)	Meaning
unsigned long	self_test_errors	*(mask)*	self-test errors encountered during power-up††
		1	CPM DRAM error
		2	CPM 32-bit counter
		4	CPM System Timing Controller (9513a)
		8	CPM DMAC
		10/16	MPM0 DRAM (tested from CPM-global bus)
		20/32	MPM0 DRAM (tested from MPM0)
		40/64	MPM0 interrupt latch
		80/128	unused
		100/256	MPM1 DRAM (tested from CPM-global bus)
		200/512	MPM1 DRAM (tested from MPM1)
		400/1024	MPM1 interrupt latch
		800/2048	unused
		1000/4096	MPM3 DRAM (tested from CPM-global bus)
		2000/8192	MPM3 DRAM (tested from MPM3)
		4000/16384	MPM3 interrupt latch
		8000/32768	unused
		10000/65536	unused
		20000/131072	unused
		40000/262144	unused
		80000/524288	unused
		100000/1048576	unused
		200000/2097152	unused
		400000/4194304	unused
		800000/8388608	unused
		1000000/16777216	unused
		2000000/33554432	unused
		4000000/67108864	unused
		8000000/134217728	unused
		10000000/268435456	unused
		20000000/536870912	unused
		40000000/1073741824	unused
		80000000/2147483648	unused
unsigned long	version	9	current value for this version of *unit_config* structure
unsigned long	model_number	19c8/6600	INTERVIEW 6600
		1a90/6800	INTERVIEW 6800 *TURBO*
		1b58/7000	INTERVIEW 7000
		1c20/7200	INTERVIEW 7200 *TURBO*
		1d4c/7500	INTERVIEW 7500
		1e14/7700	INTERVIEW 7700 *TURBO*
unsigned char	feb_type	0	original version
		1	version with increased speed of software and faster access to ticks from FEB
		2	version which supports high-speed RAM recording, specifically aggregate T1 or G.703 data capture
		3	version which also supports INTERVIEW 7200 and 7700 *TURBO*s

(unit_config continued on next page)

†† If *(unit_config.self_test_errors & mask) == mask*, the error is present.
If *(unit_config.self_test_errors & 0xffffffff) == 0*, no errors encountered during power-up.

Table 69-1 (continued)

Type	Variable	Value (hex/decimal)	Meaning
unsigned char	ls_turbo	0 1	unit is not *TURBO* unit is *TURBO*
unsigned char	xdram_rev_num		XDRAM revision number
unsigned char	xdram_present	0 1	XDRAM board is not present XDRAM board is present
unsigned long	xdram_lo_addr		low end of memory range
unsigned long	xdram_hi_addr		high end of memory range
unsigned char	reserved		reserved
unsigned char	hard_disk_type	0, 2 3	20-Mbyte disk 44-Mbyte disk
unsigned char	xtim_installed	0 1	TIM-expansion shelf is not present TIM-expansion shelf is present
unsigned char	xsys_ram_present	0 1	additional system memory is not present additional system memory is presen
unsigned long	xsys_ram_lo_addr		low end of memory range
unsigned long	xsys_ram_hi_addr		high end of memory range
unsigned long	spare1		reserved/undefined
unsigned long	spare2		reserved/undefined
unsigned long	spare3		reserved/undefined
unsigned long	spare4		reserved/undefined
unsigned long	spare5		reserved/undefined
unsigned long	spare6		reserved/undefined
unsigned long	spare7		reserved/undefined
unsigned long	sw_version		software version†††
unsigned long	fw_version		firmware version†††
Structure Name: mpm_info			Structure containing information on specific MPM board. Instance of this structure for each MPM board is contained in array named *unit_config.mpm*. Declared as type *extern struct*.
unsigned char	rev_num		MPM revision number
unsigned char	present	0 1	specific MPM board (of four) not present specific MPM board (of four) present
unsigned long	lo_addr		low end of memory range
unsigned long	hi_addr		high end of memory range

†††To display the software version in the same format presented on the main menu screen, 5.00 for example, use the following format in a call to *displayf* (or *tracef*):

```
displayf("%lu.%02lu%c", ((unit_config.sw_version >> 8)/100), ((unit_config.sw_version >> 8)%100),
    (char)(unit_config.sw_version & 0xff));
```

The same format may be used for presentation of the firmware version.

70 Remote Port I/O

The REMOTE RS-232 port is a "spare" serial interface through which the programmer may communicate with other equipment. The remote port is located at the rear of the INTERVIEW next to the printer port. (The REMOTE LED on the front panel of the INTERVIEW is related to remote control of the unit, unimplemented at this time.)

Remote-port functions must be coded in C regions on the Protocol Spreadsheet. There are no spreadsheet-token equivalents of the C variables and routines described in this section. Use these variables and routines in either emulate or monitor mode to transmit and receive data through the remote port.

The remote-communications process on the CPM controls the flow of data between the user's program and the remote port. When data is received through the remote port, this process temporarily buffers it in a 2048-byte input queue. The user's program makes requests for data from the input queue via the *rmt_getc*, *rmt_getl*, and *rmt_gets* input routines discussed below. When the remote-communications process receives a request, it removes data from the queue and passes it to the task. If there are no outstanding requests at the time data is received, it is discarded from the input queue—i.e., data received between requests cannot be retrieved. This is the default condition of the input queue.

To "lock" all received characters in the input queue, call *rmt_lock*. When the input queue is locked, the remote-communications process removes data only when 1) a user task has requested data via the *rmt_getc*, *rmt_getl*, or *rmt_gets* routine, 2) the input queue is full and some data must be discarded in order for incoming data to be buffered, or 3) *rmt_flushi* is executed. "Unlock" the input queue with *rmt_unlock*. *rmt_unlock*, *rmt_flushi*, and *rmt_flusho* are automatically executed whenever the INTERVIEW returns to Program mode.

> NOTE: Although requests to receive (or transmit) data from more than one task are queued by the remote-communications process, a single task can have only one such request outstanding at a time.

Similarly, when the programmer wants to send data out the remote port, he calls *rmt_putc*, *rmt_puts*, or *rmt_putb*. The remote-communications process temporarily places these requests in an output queue before transmitting them through the remote port.

70.1 Structures

There are no structures associated exclusively with remote functions.

70.2 Variables

Table 70-1 lists the event variables specific to remote port I/O operations. Use most of these variables to detect changes in the status of the input and output queues.

As data is received through the remote port, the remote-communications process temporarily stores it in the input queue. Use *rmt_input_not_empty*, *rmt_input_almost_full*, and *rmt_input_overflow* to monitor how full the input queue is. When the input queue is "almost full," incoming data must be stopped in order to prevent the queue from overflowing.

rmt_input_almost_empty and *rmt_input_empty* are significant events as the remote communications process takes data out of the input queue. These events indicate that that the input queue is ready to accept more data.

Table 70-1
Remote Port I/O Variables

Type	Variable	Value (hex/decimal)	Meaning
extern event	rmt_break		True when a break (NULL with a framing error) is received through the remote port. Line Setup configured for emulate or monitor mode.
extern event	rmt_input_not_empty		True when remote input-queue transitions from empty to not empty. Beginning to receive characters. Line Setup configured for emulate or monitor mode.
extern event	rmt_input_almost_full		True when the remote input-queue transitions from less than 3/4 full to 3/4 full as data is being put into the queue. Line Setup configured for emulate or monitor mode.
extern event	rmt_input_overflow		True when remote input-queue transitions from not full to full. At this point, the oldest existing data in the queue is discarded to make room for new data coming in the remote port. Line Setup configured for emulate or monitor mode.
extern event	rmt_input_almost_empty		True when the remote input-queue transitions from more than 1/4 full to 1/4 full as data is being taken out of the queue. Line Setup configured for emulate or monitor mode.
extern event	rmt_input_empty		True when remote input-queue transitions from not empty to empty. All characters have been read or discarded. Line Setup configured for emulate or monitor mode.
extern event	rmt_output_empty		True when remote output-queue transitions from not empty to empty. All data output to the remote port has been transmitted. Line Setup configured for emulate or monitor mode.

70.3 Routines

Remote routines fall into three categories. *Input* routines are used to read data received from the remote port. Use *output* routines to transmit data through the remote port. The last category of routines reads or sets *parameters* for the remote port.

(A) Input Routines

Use *rmt_getc*, *rmt_getl*, and *rmt_gets* to read data received through the remote port. Use *rmt_lock* and *rmt_unlock* to control the flow of data from the input queue.

rmt_getc

Synopsis

```
extern int rmt_getc(wait);
int wait;
```

Description

The *rmt_getc* routine reads the next character (if present) from the remote port.

Inputs

If no character is available from the input queue when *rmt_getc* is called, this parameter determines when the routine will return:

- Specify a timeout value in the hexadecimal range $^{0}0^{1}$ through $^{F}F^{E}$ (decimal 1 through 65534) to indicate how long the routine should wait for a character to become available before returning. During this waiting period, *no other conditions and actions within the same state will be executed.* (The *extern event* variable *rmt_input_not_empty* in Table 70-1 can be used to indicate when data is received.)

  At the end of the timeout, the routine returns without a character if none is available. Timeout values represent tenths of a second. If another task has already requested data from the queue, this request will be queued.

- When the value is hexadecimal $^{F}F^{F}$, the routine does not return until a character becomes available. If another task has already requested data from the queue, this request will be queued.

- When the value is zero, the routine returns without a character if none is available. If there is already an outstanding request from another task, a zero value also causes the remote-communications process to return from the routine without checking the input queue.

NOTE: More than one test (task) may request data from the input queue. The remote-communications processes queues these requests as they are made. To ensure that requests are processed in turn, use this "wait" parameter consistently across tests. If you set the parameter to a non-zero value in a call to *rmt_getc* (or *rmt_gets*) in one test, do the same in all tests.

Returns

If a character is present in the input queue, this routine returns the character (as an *int*) read. If no character is present and the routine's "wait" parameter is zero or the timeout expires, a −1 is returned. When the parameter is zero, a −1 also is returned if there is already an outstanding request from another task.

Example

In the following example, the routine does not wait for a character to become available in the remote port before returning. Each time the C key is pressed, the next character, if present, is displayed. If a −1 is returned instead of a character, a message to that effect will be displayed on the prompt line.

```
LAYER: 1
    STATE: get_next_character
        CONDITIONS: ENTER_STATE
        ACTIONS:
        {
         display_prompt("Press C to get next character.                    ");
         rmt_lock();
        }
        CONDITIONS: KEYBOARD "cC"
        ACTIONS:
        {
         int character;
         character = rmt_getc(0);
         if(character == -1)
            display_prompt("No character available.                       ");
         else
            displayf("%c", character);
        }
```

rmt_getl

Synopsis

```
extern int rmt_getl(string_ptr, max_length);
char * string_ptr;
int max_length;
```

Description

rmt_getl reads from the remote port one line at a time. This routine gets at the most the specified number of characters from the remote port and puts them in an array. Unless a carriage return or linefeed is encountered, the routine does

not return until the specified number of characters has been read. A carriage return or linefeed causes the routine to return, even if the specified number of characters has not yet been read. The carriage return or linefeed is replaced by a terminating NULL character in the array.

Inputs

The first parameter is a pointer to the array into which the characters will be put.

The second parameter is the maximum number of characters to be read.

Returns

This routine returns the number of characters (preceding the terminating NULL) read into the array.

Example

Each time the L key is pressed, twenty characters, at the most, are read from the remote port, put into an array called *data*, and displayed on the screen.

```
LAYER: 1
    STATE: read_line
        CONDITIONS: ENTER_STATE
        ACTIONS:
        {
         display_prompt("Press L to get next line.                    ");
         rmt_lock();
        }
        CONDITIONS: KEYBOARD "IL"
        ACTIONS:
        {
         int number;
         unsigned char data [25];
         number = rmt_getl(data, 20);
         displayf("\n%u characters read:\n%.20s\n", number, data);
        }
```

rmt_gets

Synopsis

```
extern int rmt_gets(string_ptr, length, wait);
char * string_pir;
int length;
int wait;
```

Description

Similar to *rmt_getl*, this routine gets a specified number of characters from the remote port and puts them in an array. Unlike *rmt_getl*, characters continue to be read even if a carriage return or linefeed is encountered. The array is not NULL-terminated.

Inputs

The first parameter is a pointer to the array into which the characters will be put.

The second parameter is the number of characters to be read.

If the specified number of characters is not available from the input queue when *rmt_getl* is called, the third parameter determines when the routine will return:

- Specify a timeout value in the hexadecimal range $^{\circ}0^{\circ}1$ through $^{F}F^{F}E$ (decimal 1 through 65534) to indicate how long the routine should wait for the specified number of characters to become available before returning. During this waiting period, *no other conditions and actions within the same state will be executed.*

  At the end of the timeout, the routine returns with less than the specified number of characters if all are not available. Timeout values represent tenths of a second. If another task has already requested data from the queue, this request will be queued.

- When the value is hexadecimal $^{F}F^{F}F$, the routine does not return until the specified number of characters becomes available. If another task has already requested data from the queue, this request will be queued.

- When the value is zero, the routine returns with less than the specified number of characters if all are not available. If there is already an outstanding request from another task, a zero value also causes the remote-communications process to return from the routine without checking the input queue.

  NOTE: More than one test (task) may request data from the input queue. The remote-communications processes queues these requests as they are made. To ensure that requests are processed in turn, use this "wait" parameter consistently across tests. If you set the parameter to a non-zero value in a call to *rmt_gets* (or *rmt_getc*) in one test, do the same in all tests.

Returns

This routine returns the number of characters read from the remote port.

Example

When the [S] key is pressed, the INTERVIEW has a minute to read up to 4000 characters from the remote port. The program puts the characters into an array called *data*, displays them on the screen (until a NULL is encountered—see *%s* in *tracef* routine, Section 64), and writes them to a file named *echo_time*. This is the program that might be run to receive the file transmitted in the *rmt_putb* example.

```
{
 #define FILE_LENGTH 4000
 #define FILENAME "FD1/usr/echo_time"
 #include <stdio.h>
 #include <trace_buf.h>
 extern struct trace_buf ll_trbuf;
 FILE * stream_ptr;
 size_t n;
 unsigned char data [FILE_LENGTH];
 int count;
}
LAYER: 1
    STATE: get_string
        CONDITIONS: ENTER_STATE
        ACTIONS:
        {
         rmt_lock();
         if((stream_ptr = fopen(FILENAME, "w")) == 0)
            display_prompt("Cannot open file.");
         else
            {
             display_prompt("Press S to read string.");
             pos_cursor(1,0);
            }
        }
        CONDITIONS: KEYBOARD "sS"
        ACTIONS:
        {
         count = rmt_gets(data, FILE_LENGTH, 600);
         if(count != FILE_LENGTH)
            displayf("Could not read entire string.\n");
         tracef(&ll_trbuf, "%d characters read: \n%s\n\n", count, data);
         n = fwrite(data, 1, FILE_LENGTH, stream_ptr);
         if(n != FILE_LENGTH)
            displayf("A write error has occurred.\n");
         else
            displayf("File written. \n");
         if(fclose(stream_ptr) != 0)
            displayf("Either file is already closed, or close cannot be executed. \n");
         else
            displayf("File closed.\n");
        }
```

rmt_flushi

Synopsis

```
extern int rmt_flushi();
```

Description

If characters have been received in the input queue, but have not been read yet, this routine causes them to be discarded. Whenever the INTERVIEW enters or leaves Run mode, *rmt_flushi* is automatically executed. This ensures that the input queue is empty.

NOTE: A call to any of the routines which *set* the parameters of the remote port also causes *rmt_flushi* to be executed automatically. The routines which only *get* the current parameters of the remote port have no effect on the input queue.

When the programmer calls *rmt_flushi*, requests for data from the input queue are processed before the input queue is flushed. When a call to *rmt_flushi* is made from another test, however, input routines waiting for characters from the input queue are returned.

Returns

rmt_flushi returns a zero when the input queue is flushed successfully. Otherwise, it returns a non-zero value.

Example

This example is the same as that for *rmt_getc*. Notice that as the program enters the first state, the input queue is flushed.

```
LAYER: 1
    STATE: get_next_character
        CONDITIONS: ENTER_STATE
        ACTIONS:
        {
         display_prompt("Press C to get next character.                    ");
         rmt_lock();
         rmt_flushi();
        }
        CONDITIONS: KEYBOARD "cC"
        ACTIONS:
        {
         int character;
         character = rmt_getc(0);
         if(character ==-1)
             display_prompt("No character available.                       ");
         else
             displayf("%c", character);
        }
```

rmt_lock

Synopsis

```
extern void rmt_lock();
```

Description

Recall that in its default state, the input queue does not retain characters received through the remote port between requests from user tasks. Data in the queue must either be passed to a user task or be discarded. The *rmt_lock* routine "locks" all received characters in the input queue until they are requested. (Refer again to the beginning of this section.)

Example

The following example is the same as the one for the *rmt_getl* routine. Notice that a call to *rmt_lock* is made as the program begins. The operator makes a request for data from the input queue by pressing [L]. The next line of data in the input queue is removed and put in the array named *data*.

```
LAYER: 1
    STATE: read_line
        CONDITIONS: ENTER_STATE
        ACTIONS:
        {
         display_prompt("Press L to get next line.                    ");
         rmt_lock();
        }
        CONDITIONS: KEYBOARD "IL"
        ACTIONS:
        {
         int number;
         unsigned char data [25];
         number = rmt_getl(data, 20);
         displayf("\n%u characters read:\n%.20s\n", number, data);
        }
```

rmt_unlock

Synopsis

```
extern void rmt_unlock();
```

Description

The *rmt_unlock* routine implements the inverse of the *rmt_lock* routine. If characters are received in the remote port and there are no outstanding requests for data, the remote-communications process discards the characters. (Refer also to *rmt_lock* and to the beginning of this section.)

rmt_unlock is automatically executed when the INTERVIEW returns to Program mode.

Example

In the following example, the input queue is locked as soon as the program begins. It remains locked until the operator press [U] (or [PROGRAM]).

```
LAYER: 1
    STATE: read_line
        CONDITIONS: ENTER_STATE
        ACTIONS:
        {
         display_prompt("Press L to get next line.                    ");
         rmt_lock();
        }
```

```
CONDITIONS: KEYBOARD "IL"
ACTIONS:
{
 int number;
 unsigned char data [25];
 number = rmt_getl(data, 20);
 displayf("\n%u characters read:\n%.20s\n", number, data);
}
CONDITIONS: KEYBOARD "uU"
ACTIONS:
{
 rmt_unlock();
}
```

(B) Output Routines

Use the following routines to transmit data through the remote port.

rmt_putc

Synopsis

```
extern int rmt_putc(character, wait);
unsigned char character;
int wait;
```

Description

This routine sends a specified character to the output queue of the remote port for transmission.

Inputs

The first parameter is the character to be transmitted. It may be given as a hexadecimal, octal, or decimal constant; as an alphanumeric constant inside single quotes; or as a variable. A hexadecimal value must be preceded by the prefix 0x or 0X; an octal value must be preceded by the prefix 0. If no prefix appears before the input, the number is assumed to be decimal.

If space in the output queue is not available for the character when *rmt_putc* is called, the second parameter determines when the routine will return:

- Specify a timeout value in the hexadecimal range $^{0}0^{1}$ through $^{F}F^{E}$ (decimal 1 through 65534) to indicate how long the routine should wait for space in the output queue to become available before returning. During this waiting period, *no other conditions and actions within the same state will be executed.*

  If the character is successfully put in the queue, the routine returns zero. Timeout values represent tenths of a second. If there is already a request from another task, this request will be queued.

- When the value is hexadecimal FFFF, the routine does not return until space in the output queue becomes available. If there is already a request from another task, this request will be queued.
- When the value is zero and space in the output queue is not available, the routine returns -1. The character will not be in the queue. If another task is already waiting for access to the output queue, a zero value also causes the remote-communications process to return from the routine without checking for available space in the output queue.

> NOTE: More than one test (task) may request to send data to the output queue. The remote-communications processes queues these requests as they are made. To ensure that requests to output data are processed in turn, use this "wait" parameter consistently across tests. If you set the parameter to a non-zero value in a call to *rmt_putc* (*rmt_puts* or *rmt_putb*) in one test, do the same in all tests.

Returns

If the character is successfully written to the output queue, the routine returns zero. If no space is available in the output queue and the routine's "wait" parameter is zero or the timeout expires, a -1 is returned. When the parameter is zero, a -1 also is returned if another task is already waiting for access to the output queue.

Example

In the following example, the next character in a fox message is sent to the output queue of the remote port each time the operator presses C. As a character is successfully queued, it is displayed in the Display Window. If no space is available in the output queue for the character, -1 is returned and a message to that effect is displayed on the prompt line. No more characters will be sent.

```
{
 unsigned char data [] = "((FOX))↵";
 unsigned char character;
 int i, length, error;
}
LAYER: 1
     STATE: transmit_characters
        CONDITIONS: ENTER_STATE
        ACTIONS:
        {
         display_prompt("Press C to transmit character.                    ");
         length = sizeof(data) - 1;
        }
```

```
CONDITIONS: KEYBOARD "cC"
ACTIONS:
{
 for(i = 0; i < length; i++)
    {
     character = data[i];
     error = rmt_putc(character, 0);
     if(error == -1)
          display_prompt("No space available in output queue.                    ");
     else
          displayf("%c", character);
    }
}
```

rmt_puts

Synopsis

```
extern int rmt_puts(string_ptr, wait);
const char * string_ptr;
int wait;
```

Description

This routine outputs a NULL-terminated string to the output queue of the remote port.

Inputs

The first parameter is a pointer to the string to be transmitted.

If space in the output queue is not available for the string when *rmt_puts* is called, the second parameter determines when the routine will return:

- Specify a timeout value in the hexadecimal range $^{0}0^{1}$ through $^{F}F^{E}$ (decimal 1 through 65534) to indicate how long the routine should wait for space in the output queue to become available before returning. During this waiting period, *no other conditions and actions within the same state will be executed.*

  Before the timeout expires, as many characters as will fit are put into the output queue. Timeout values represent tenths of a second. If there is already a request from another task, this request will be queued.

- When the value is hexadecimal $^{F}F^{F}$, the routine does not return until space in the output queue becomes available. If there is already a request from another task, this request will be queued.

- When the value is zero and space is not available in the output queue, the routine returns the number of characters, if any, put into the queue. If another task is already waiting for access to the output queue, a zero value also causes the remote-communications process to return from the routine without checking for available space in the output queue.

NOTE: More than one test (task) may request to send data to the output queue. The remote-communications processes queues these requests as they are made. To ensure that requests to output data are processed in turn, use this "wait" parameter consistently across tests. If you set the parameter to a non-zero value in a call to *rmt_puts* (*rmt_putc* or *rmt_putb*) in one test, do the same in all tests.

Returns

This routine returns the number of characters put into the output queue.

Example

The following example is similar to the one given for *rmt_putc*. When the S key is pressed, the fox message is sent to the remote port. The difference is that the message is output to the remote port as a string (rather than character by character). If the output queue is full, the routine does not wait for space to become available before returning. The number of characters successfully queued is displayed in the Display Window. If the number of characters queued is less than the length of the string, a message to that effect is displayed on the prompt line.

```
{
 unsigned char data [] = "((FOX))\r";
 int count, length;
}
LAYER: 1
     STATE: transmit_string
        CONDITIONS: ENTER_STATE
        ACTIONS:
        {
         display_prompt("Press S to transmit string.                    ");
         length = sizeof(data) - 1;
        }
        CONDITIONS: KEYBOARD "sS"
        ACTIONS:
        {
         count = rmt_puts(data, 0);
         if(count !=length)
            display_prompt("Could not output entire string.             ");
         pos_cursor(1,0);
         displayf("%d characters transmitted.", count);
        }
```

rmt_putb

Synopsis

```
extern int rmt_putb(string_ptr, length, wait);
const char * string_ptr;
int length;
int wait;
```

Description

This routine sends a string of specified length to the output queue of the remote port.

Inputs

The first parameter indicates the string to be output.

The second parameter is the length of the string to be output.

If space in the output queue is not available for the string when *rmt_putb* is called, the third parameter determines when the routine will return:

- Specify a timeout value in the hexadecimal range $^{0}0^{1}$ through $^{F}F^{E}$ (decimal 1 through 65534) to indicate how long the routine should wait for space in the output queue to become available before returning. During this waiting period, *no other conditions and actions within the same state will be executed.*

  Before the timeout expires, as many characters as will fit are put into the output queue. Timeout values represent tenths of a second. If there is already a request from another task, this request will be queued.

- When the value is hexadecimal $^{F}F^{F}$, the routine does not return until space in the output queue becomes available and all characters in the string have been queued. If there is already a request from another task, this request will be queued.

- When the value is zero and space is not available in the output queue, the routine returns the number of characters, if any, put into the queue. If there is already an outstanding request from another task, a zero value also causes the remote-communications process to return from the routine without checking for available space in the output queue.

  NOTE: More than one test (task) may request to send data to the output queue. The remote-communications processes queues these requests as they are made. To ensure that requests to output data are processed in turn, use this "wait" parameter consistently across tests. If you set the parameter to a non-zero value in a call to *rmt_putb* (*rmt_putc* or *rmt_puts*) in one test, do the same in all tests.

Returns

This routine returns the number of characters put into the output queue.

Example

This is the program that might be run to transmit the file received in the *rmt_gets* example. The user specifies the filename and its size (shown in the directory listing on the File Maintenance screen) in the two *#define* preprocessor directives at the beginning of the program. When the program begins, the

contents of the file named *echo_time* are read into an array called *data*. When the operator presses the T key, the contents of the array are transmitted and displayed.

```
{
 #define FILE_LENGTH 4000
 #define FILENAME "FD1/usr/echo_time"
 #include <stdio.h>
 #include <trace_buf.h>
 extern struct trace_buf t1_trbuf;
 FILE * stream_ptr;
 size_t n;
 unsigned char data [FILE_LENGTH];
 unsigned char size [FILE_LENGTH+100];
 int count;
}
LAYER: 1
     STATE: transmit_string
        CONDITIONS: ENTER_STATE
        ACTIONS:
        {
         if((stream_ptr = fopen(FILENAME, "r")) == 0)
            display_prompt("Cannot open file.");
         else
           {
            pos_cursor(1,0);
            n = fread(data, 1, FILE_LENGTH, stream_ptr);
            if(n != FILE_LENGTH)
                 displayf("Either a read error has occurred, or an EOF has been
                     encountered.\n");
            if(fclose(stream_ptr) != 0)
                 displayf("Either file is already closed, or close cannot be executed. \n");
            else
                 displayf("File closed.\n");
            if(n == FILE_LENGTH)
                 display_prompt("Press T to transmit characters.");
           }
        }
        CONDITIONS: KEYBOARD "tT"
        ACTIONS:
        {
         count = rmt_putb(data, FILE_LENGTH, 0xff);
         if(count != FILE_LENGTH)
            displayf("Could not output entire string.\n");
         sprintf(size, "%d characters transmitted: %%.%dH", count, count);
         tracef(&t1_trbuf, size, data);
         tracef(&t1_trbuf, "\n\n");
        }
```

rmt_flusho

Synopsis

```
extern int rmt_flusho();
```

Description

If characters are queued to be output from the remote port, but have not been transmitted yet, this routine causes them to be discarded. This ensures that anything previously in the output queue port is deleted.

rmt_flusho is automatically executed when the INTERVIEW returns to Program mode.

> NOTE: A call to any of the routines which *set* the parameters of the remote port causes *rmt_flusho* to be executed automatically. The routines which only *get* the current parameters of the remote port have no effect on the output queue.

Returns

rmt_flusho returns a zero when the output queue is flushed successfully. Otherwise, it returns a non-zero value.

Example

This example is the same as that for *rmt_putc*. Notice that as the program enters the first state, the output queue is flushed.

```
{
 unsigned char data [] = "((FOX))";
 unsigned char character'
 int i, length, error;
}
LAYER: 1
     STATE: transmit_a_character
        CONDITIONS: ENTER_STATE
        ACTIONS:
        {
         rmt_flusho();
         display_prompt("Press C to transmit character.                    ");
         length = sizeof(data);
        }
        CONDITIONS: KEYBOARD "cC"
        ACTIONS:
        {
         for(i = 0; i < length; i++)
            {
             character = data[i];
             error = rmt_putc(character, 0);
             if(error ==-1)
                {
                 display_prompt("No space available in output queue.       ");
                 break;
                }
             else
                  displayf("%c", character);
            }
        }
```

rmt_suspendo

Synopsis

```
extern int rmt_suspendo();
```

Description

If characters are queued to be output from the remote port, but have not been transmitted yet, this routine causes transmitting to be suspended. The output queue is *not* flushed. Use this routine only when the remote port handshaking mode is full-duplex without flow control.

Returns

rmt_suspendo returns a zero when transmitting is successfully suspended. Otherwise, it returns a non-zero value.

Example

When the INTERVIEW receives an X-OFF as a signal to stop sending data, it suspends transmissions from the remote port.

```
{
 extern event rmt_input_not_empty;
 int character;
}
LAYER: 1
     STATE: suspend_output
        CONDITIONS: ENTER_STATE
        ACTIONS:
        {
         rmt_lock();
        }
        CONDITIONS:
        {
         rmt_input_not_empty
        }
        ACTIONS:
        {
         character = rmt_getc(1);
         if(character == 0x13)
            rmt_suspendo();
        }
        TIMEOUT ck_input RESTART 0.001
        CONDITIONS: TIMEOUT ck_input
        ACTIONS:
        {
         character = rmt_getc(1);
         if(character == 0x13)
            rmt_suspendo();
        }
        TIMEOUT ck_input RESTART 0.001
```

rmt_resumeo

Synopsis

```
extern int rmt_resumeo();
```

Description

This routine resumes transmission of characters from the remote port. Use this routine only when the remote port handshaking mode is full-duplex without flow control.

Returns

rmt_resumeo returns a zero when transmitting is successfully resumed. Otherwise, it returns a non-zero value.

Example

When the INTERVIEW receives an X-ON as a signal to send data, it resumes transmissions from the remote port.

```
{
 int character;
}
LAYER: 1
     STATE: resume_output
        CONDITIONS: ENTER_STATE
        ACTIONS:
        {
         rmt_lock();
        }
        TIMEOUT RESTART ck_input 0.001
        CONDITIONS: TIMEOUT check_input
        ACTIONS:
        {
         character = rmt_getc(1);
         if(character == 0x11)
            rmt_resumeo();
        }
        TIMEOUT ck_input RESTART 0.001
```

rmt_send_break

Synopsis

```
extern int rmt_send_break(wait);
int wait;
```

Description

This routine causes a break, queued as other transmits, to be transmitted.

Inputs

If space in the output queue is not available for the break when *rmt_send_break* is called, the only parameter determines when the routine will return:

- Specify a timeout value in the hexadecimal range $^{0}0_{1}$ through $^{F}F^{E}$ (decimal 1 through 65534) to indicate how long the routine should wait for space in the output queue to become available before returning. During this waiting period, *no other conditions and actions within the same state will be executed.*

  If the break is successfully put in the queue, the routine returns zero. Timeout values represent tenths of a second. If there is already a request from another task, this request will be queued.

- When the value is hexadecimal $^{F}F^{F}$, the routine does not return until space in the output queue becomes available and the break has been queued. If there is already a request from another task, this request will be queued.

- When the value is zero and space in the output queue is not available, the routine returns –1. The break will not be in the queue. If another task is already waiting for access to the output queue, a zero value also causes the remote-communications process to return from the routine without checking for available space in the output queue.

  NOTE: More than one test (task) may request to send data to the output queue. The remote-communications processes queues these requests as they are made. To ensure that requests to output data are processed in turn, use this "wait" parameter consistently across tests. If you set the parameter to a non-zero value in a call to *rmt_send_break* (*rmt_putc*, *rmt_puts* or *rmt_putb*) in one test, do the same in all tests.

Returns

If the break is successfully written to the output queue, the routine returns zero. If no space is available in the output queue and the routine's "wait" parameter is zero or the timeout expires, a –1 is returned. When the parameter is zero, a –1 also is returned if another task is already waiting for access to the output queue.

Example

In this example, a break is transmitted each time the operator presses the space bar.

```
LAYER: 1
    STATE: transmit_break
        CONDITIONS: KEYBOARD " "
        ACTIONS:
        {
         rmt_send_break(1);
        }
```

(C) Configuration Routines

The default configuration for the remote port at boot-up is the following:

Baud rate = 1200
Bits/character = 8
Parity = None
Mode = Full-duplex

Use the first four routines discussed below to change these settings. The programmer's reconfiguration of the remote port is not affected when the INTERVIEW exits or re-enters Run mode.

A call to any of these *set* routines causes *rmt_flushi* and *rmt_flusho* to be executed automatically before the parameter is set.

Use the remaining four routines to read the current parameter-settings for the remote port. These *get* routines have no effect on the input and output queues.

rmt_set_baud_rate

Synopsis

```
extern int rmt_set_baud_rate(speed);
int speed;
```

Description

This routine sets the baud rate for the remote port. The default value at boot-up is 1200.

> NOTE: A call to *rmt_set_baud_rate* causes *rmt_flushi* and *rmt_flusho* to be executed automatically before the baud rate is set.

Inputs

The only parameter is the desired baud rate. Values that are multiples of 300 in the range 300 through 19200 are valid.

Returns

If the specified baud rate is valid and successfully set, zero is returned. If the baud rate is valid, but not successfully set, -1 is returned. For an invalid baud rate, the routine returns -2.

Example

In order for two devices to communicate with each other, they must be using the same baud rate. When they are not the same, some devices send a break as a signal for the other to adjust its baud rate. If the following example, the INTERVIEW changes the baud rate for the remote port whenever a break is received.

```
{
 extern event rmt_break;
 int error;
 int speed = 300;
}
LAYER: 1
     STATE: adjust_baud_rate
        CONDITIONS:
        {
         rmt_break
        }
        ACTIONS:
        {
         error = rmt_set_baud_rate(speed);
         if(error != -1)
            {
             speed *= 2;
             if(speed > 19200)
                  speed = 300;
            }
         else
             displayf("Unable to set the baud rate to %d.", speed);
        }
```

rmt_set_bits

Synopsis

```
extern int rmt_set_bits(value);
int value;
```

Description

This routine sets the number of bits per character for the remote port. The default setting at boot-up is 8 bits/character.

> NOTE: A call to *rmt_set_bits* causes *rmt_flushi* and *rmt_flusho* to be executed automatically before the number of bits/character is set.

Inputs

The only parameter is the number of bits/character. Valid values are five through eight.

Returns

If the specified number of bits/character is valid and successfully set, zero is returned. If the number is valid, but not successfully set, -1 is returned. For an invalid value, the routine returns -2.

Example

In this example, the number of bits/character for the remote port is set to 7 and displayed on the Display Window screen.

```
LAYER: 1
    STATE: set_parameters
        CONDITIONS: ENTER_STATE
        ACTIONS:
        {
         displayf("Bits = %d ", rmt_set_bits(7));
        }
```

rmt_set_parity

Synopsis

```
extern int rmt_set_parity(parity);
int parity;
```

Description

This routine sets the parity for the remote port. The default setting at boot-up is no parity.

> NOTE: A call to *rmt_set_parity* causes *rmt_flushi* and *rmt_flusho* to be executed automatically before the parity for the remote port is set.

Inputs

The only parameter is a value designating the desired parity. Valid values are the following: none (0), odd (1), even (2), mark (3), or space (4).

Returns

If the specified parity value is valid and successfully set, zero is returned. If the value is valid, but not successfully set, -1 is returned. For an invalid parity value, the routine returns -2.

Example

In this example, the number of bits/character for the remote port is set to 7 and parity is even. Both settings are displayed on the Display Window screen.

```
LAYER: 1
    STATE: set_parameters
        CONDITIONS: ENTER_STATE
        ACTIONS:
        {
         displayf("Bits = %d  Parity = %d ", rmt_set_bits(7), rmt_set_parity(2));
        }
```

rmt_set_mode

Synopsis

```
extern int rmt_set_mode(mode);
int mode;
```

Description

This routine sets the handshaking mode for the remote port. The default setting at boot-up is FDX with no flow control.

> NOTE: A call to *rmt_set_mode* causes *rmt_flushi* and *rmt_flusho* to be executed automatically before the mode for the remote port is set.

Inputs

The only parameter is a value designating the mode. Valid values are the following:

0 = Full-duplex with no flow control (FDX)
1 = Half-duplex (HDX)
2 = Full-duplex with X-ON/X-OFF characters for flow control
3 = Full-duplex with DTR and CTS EIA leads for flow control. Use a special null-modem cable for direct connections. See Figure 70-1.

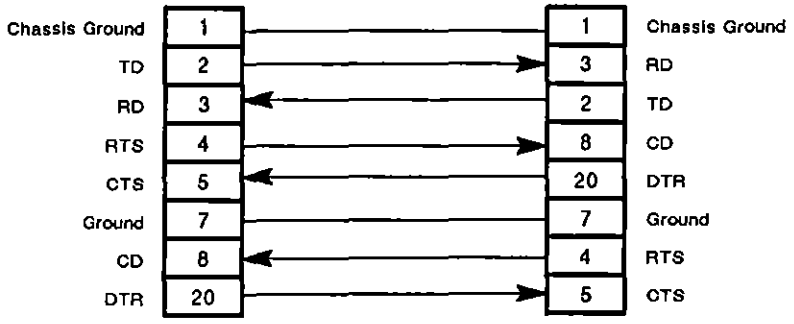

Figure 70-1 Null-modem cable connections.

Returns

If the specified mode value is valid and successfully set, zero is returned. If the value is valid, but not successfully set, -1 is returned. For an invalid mode value, the routine returns -2.

Example

In this example, the number of bits/character for the remote port is set to 7, parity is even, and the mode is set for FDX with X-ON/X-OFF. All three settings are displayed on the Display Window screen.

```
LAYER: 1
    STATE: set_parameters
        CONDITIONS: ENTER_STATE
        ACTIONS:
        {
         displayf("Bits = %d  Parity = %d  Mode = % d ", rmt_set_bits(7),
            rmt_set_parity(2), rmt_set_mode(2));
        }
```

rmt_get_baud_rate

Synopsis

```
extern int rmt_get_baud_rate();
```

Description

This routine gets the current baud-rate setting for the remote port.

Returns

The baud rate for the remote port is returned.

Example

As the program begins, the current baud-rate setting for the remote port is displayed on the Display Window screen.

```
LAYER: 1
    STATE: baud_rate
        CONDITIONS: ENTER_STATE
        ACTIONS:
        {
         displayf("Baud = %d ", rmt_get_baud_rate());
        }
```

rmt_get_bits

Synopsis

```
extern int rmt_get_bits();
```

Description

This routine tells how many bits there are per character. Possible values are five through eight.

Returns

The current number of bits per character for the remote port is returned.

Example

In this example, the current baud-rate setting and the number of bits/character for the remote port are displayed on the Display Window screen.

```
LAYER: 1
    STATE: current_parameters
        CONDITIONS: ENTER_STATE
        ACTIONS:
        {
         displayf("Baud = %d  Bits = %d ", rmt_get_baud_rate(), rmt_get_bits());
        }
```

rmt_get_parity

Synopsis

```
extern int rmt_get_parity();
```

Description

This routine gets the current parity setting for the remote port.

Returns

The current number of bits per character for the remote port is returned.

Example

In this example, the current baud-rate setting, number of bits/character, and the parity for the remote port are displayed on the Display Window screen.

```
LAYER: 1
    STATE: current_parameters
        CONDITIONS: ENTER_STATE
        ACTIONS:
        {
         displayf("Baud = %d  Bits = %d  Parity = %d ", rmt_get_baud_rate(),
             rmt_get_bits(), rmt_get_parity());
        }
```

rmt_get_mode

Synopsis

extern int rmt_get_mode();

Description

This routine gets the current handshaking mode for the remote port.

Returns

The current handshaking mode for the remote port is returned:

0 = Full-duplex with no flow control (FDX)
1 = Half-duplex (HDX)
2 = Full-duplex with X-ON/X-OFF characters for flow control
3 = Full-duplex with DTR and CTS EIA leads for flow control Requires a special null-modem cable for INTERVIEW-to-INTERVIEW direct connections. Refer to Figure 70-1.

Example

In this example, the current baud-rate setting, number of bits/character, parity, and handshaking mode for the remote port are displayed on the Display Window screen.

```
LAYER: 1
    STATE: current_parameters
        CONDITIONS: ENTER_STATE
        ACTIONS:
        {
         displayf("Baud = %d  Bits = %d  Parity = %d  Mode = %d ", rmt_get_baud_rate(),
             rmt_get_bits(), rmt_get_parity(), rmt_get_mode());
        }
```

71 AUX Port I/O

Bit Number	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Transmitter's AUX Port Lead Configuration	O	I	I	I	I	I	I	I/C	O	O	O	O	O	O	O	O
Pin Number	16	14	12	10	8	6	4	2	15	13	11	9	7	5	3	1
Bit will be used for	C	U	U	U	U	U	U	C	D	D	D	D	D	D	D	D

Bit Number	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Receiver's AUX Port Lead Configuration	I/C	I	I	I	I	I	I	O	I	I	I	I	I	I	I	I
Pin Number	16	14	12	10	8	6	4	2	15	13	11	9	7	5	3	1

O	Output/Non-control
I	Input/Non-control
I/C	Input/Control
C	Control
D	Data
U	Unassigned

Figure 71-1 Sample AUX port lead configurations for two INTERVIEWs connected by their AUX interfaces. Assume one-way data transmission (i.e., one device is controlling the other).

71 AUX Port I/O

The Auxiliary (AUX) port is a "spare" interface through which the programmer may communicate with other lab equipment. The AUX port is located at the rear of the INTERVIEW, between the printer and RGB connectors. It is controlled by a Zilog CIO (*C*ounter/Timer, Parallel *I*nput/*O*utput Unit) chip. The AUX port may be used as a serial or parallel interface. When it is operated as a parallel port, up to sixteen bits (one bit on each of sixteen leads) may be transmitted simultaneously.

AUX-port control must be coded in C regions on the Protocol Spreadsheet. There are no spreadsheet-token equivalents of the C variables and routines described in this section.

A normal configuration of equipment using the AUX port will involve two INTERVIEWs with AUX port setups that mirror each other to some extent, as in Figure 71-1. The transmitting INTERVIEW will use one of its output leads as a "strobe" to signal to the receiving INTERVIEW that an AUX word is available to be read. The receiver will detect this strobe as an *aux_change* event.

The receiving INTERVIEW will use one of its output leads to acknowledge each AUX word received. The transmitting INTERVIEW will detect this acknowledgment as an *aux_change* event.

> NOTE: The AUX port is not controlled by the same CPU that handles the user program. The need for interprocessor communication without data buffering makes rapid, successive transmissions difficult to handle. It is recommended, therefore, that control bits be set aside for flow control—a bit set by the transmitter as input/control is set by the receiver as output/non-control, and vice versa—and that every output word be acknowledged before a succeeding word is output.

71.1 Variables

Table 71-1 lists the variables specific to AUX I/O operations. The fast-event variable, *aux_change,* detects a change in a lead that has been configured as a control lead. Any or all of the sixteen leads in the interface may be designated control leads. Section 71.2 explains how to configure control leads.

aux_change does not establish which control lead(s) has changed. Two associated variables, *curr_aux_value* and *prev_aux_value*, indicate the status of all sixteen leads. These are two-byte (*short*) variables. Each lead is represented by a different bit in the *short*. If the bit-value of a given lead is zero, the lead is on. If the bit-value is one, the lead is off.

Whenever a control lead changes, the value in *curr_aux_value* is written to *prev_aux_value*. Then *curr_aux_value* is updated.

Table 71-1
AUX Port I/O Variables

Type	Variable	Meaning
extern fast_event	aux_change	True when the status of a lead designated as control (and input) changes. Is automatically made to come true by the CIO chip as soon as leads have been configured via *set_aux_direction* and *set_aux_ctl_leads* routines. Therefore, condition must be tested again in a different state. Line Setup configured for emulate or monitor mode.
extern volatile const unsigned short	curr_aux_value	Each bit designates a different lead. A bit-value of one indicates a given lead is on. When value of *curr_aux_value* is exclusive *ored* (^) with *prev_aux_value*, result indicates those leads whose status has changed. Updated when *aux_change* comes true. Line Setup configured for emulate or monitor mode.
extern volatile const unsigned short	prev_aux_value	Value of previous *curr_aux_value*. Updated when control leads change, but only after logic has had a chance to compare current and previous leads. Line Setup configured for emulate or monitor mode.

71.2 Routines

In the examples for the following routines, assume that two INTERVIEW's are connected and that data flows in one direction.

> *CAUTION: You may damage the AUX interface if the same lead is designated as output on both units. We suggest that you set the leads on each unit as input/output and control/non-control* before *you connect the AUX interfaces. See Figure 71-1.*

set_aux_direction

Synopsis

```
extern void set_aux_direction(input_or_output);
unsigned short input_or_output;
```

Description

This routine designates leads on the AUX port as input or output. Designated output leads for the transmitter are set as input leads by the receiver.

Inputs

The only input is a sixteen-bit variable. Each bit in the variable designates one lead and may be set to zero (output) or one (input).

Example

Both sides of the connection may be transmitter or receiver. But for simplification in examples, let's designate only one side as the transmitter and the other as the receiver. In this example, the transmitter sets all 8 bits of the low-order byte as output bits for data, the low-order bit of the high byte as input (for handshaking), the next 6 bits of the high byte as input (unused), and the high-order bit as output (the receiver will designate this bit as input for handshaking).

```
LAYER: 1
    STATE: set_input_leads
       CONDITIONS: ENTER_STATE
       ACTIONS:
       {
        set_aux_direction(0x7f00);
       }
```

The other (receiver) INTERVIEW sets a bit as input (for handshaking). It must be one that was designated as output by the transmitter, the highest-order bit of the high byte. The data bits set as output by the transmitter must be set as input by the receiver. The receiver's *set_aux_direction* routine would look like this:

```
LAYER: 1
    STATE: set_input_leads
        CONDITIONS: ENTER_STATE
        ACTIONS:
        {
         set_aux_direction(0xfeff);
        }
```

set_aux_ctl_leads

Synopsis

```
extern void set_aux_ctl_leads(ctl_or_not);
unsigned short ctl_or_not;
```

Description

This routine determines whether or not leads will be control leads. Control leads must also be input leads, but input leads do not necessarily have to be control leads. Output leads can never be control leads.

Inputs

The only input is a sixteen-bit variable. Each bit in the variable designates one lead and may be set to zero (non-control) or one (control).

Example

Assuming the input/output bits set in the previous example, the transmitter sets all 8 data bits (output) as non-control, the low-order input bit of the high byte as control (for handshaking), the next 6 input bits of the high byte as non-control (unused), and the high-order output bit as non-control (the receiver will designate this bit as control for handshaking).

```
LAYER: 1
    STATE: set_control_leads
        CONDITIONS: ENTER_STATE
        ACTIONS:
        {
         set_aux_ctl_leads(0x0100);
        }
```

The "receiver" INTERVIEW sets one input bit as control for handshaking purposes. It must be one that was designated as output by the transmitter, the highest-order bit of the high byte. The receiver's *set_aux_ctl_leads* routine would look like this:

```
LAYER: 1
    STATE: set_control_leads
        CONDITIONS: ENTER_STATE
        ACTIONS:
        {
         set_aux_ctl_leads(0x8000);
        }
```

write_aux

Synopsis

```
extern void write_aux(output_word);
unsigned short output_word;
```

Description

This routine sends a combination of data, control, and (perhaps) unused bits as output. Input bits are not transmitted by the CIO.

Inputs

The only input is a sixteen-bit variable. Each bit designates one lead and may represent data or control information, or be unused. If a given lead was designated as a control lead, it is an input lead and the CIO will not transmit the status of the bit in any case, so its setting of 1 or 0 does not matter. If the lead was designated as a non-control lead, it might contain data, be unused, or contain an alternating value to indicate acknowledgment (if the other side designated it as a control lead).

Example

The transmitting INTERVIEW is going to send data to the receiving INTERVIEW. Before the next transmission can be sent, an acknowledgment must be received. The acknowledgment is detected by the fast-event variable *aux_change*.

> NOTE: The CIO chip automatically generates a true *aux_change* condition when the *set_aux_ctl_leads* routine has been executed. The *aux_change* condition, therefore, should be placed in a separate programming state from the *set_aux_ctl_leads* routine.

The transmitter's program might look like this:

```
LAYER: 1
 {
  extern fast_event aux_change;
  extern volatile const unsigned short curr_aux_value;
  volatile unsigned short curr;
  unsigned short mask;
  unsigned char data;
 }
     STATE: configure_leads
        CONDITIONS: ENTER_STATE
        ACTIONS:
        {
         set_aux_direction(0x7f00);
         set_aux_ctl_leads(0x0100);
         curr = curr_aux_value;
         data = 0x01;
         mask = curr ^ 0x8000;
         display_prompt("Connect cable.  Press spacebar to transmit.                    ");
         pos_cursor(1,0);
        }
        NEXT_STATE: send_data
     STATE: send_data
        CONDITIONS: KEYBOARD " "
        ACTIONS:
        {
         if(data <= 10)
            {
             write_aux(mask | data);
             displayf("Transmission %d waiting for ACK.                          \n", data);
            }
        }
        NEXT_STATE: waiting
     STATE: waiting
        CONDITIONS: {aux_change}
        ACTIONS:
        {
         data++;
         mask = (mask ^ 0x8000);
         displayf("ACK received:  %04x  Press spacebar to transmit.          \n", curr);
        }
        NEXT_STATE: send_data
        CONDITIONS: {data > 10}
        ACTIONS:
        {
         display_prompt("End of test.                                                   ");
        }
```

The receiver's program would look like this:

```
LAYER: 1
 {
  extern fast_event aux_change;
  extern volatile const unsigned short curr_aux_value;
  volatile unsigned short curr;
  unsigned short mask;
  int count;
 }
     STATE: configure_leads
        CONDITIONS: ENTER_STATE
        ACTIONS:
        {
         set_aux_direction(0xfeff);
         set_aux_ctl_leads(0x8000);
        }
        CONDITIONS: {aux_change}
        ACTIONS:
        {
         curr = curr_aux_value;
         count = 1;
         mask = curr ^ 0x0100;
         display_prompt("Connect cable.  Ready to receive.                    ");
         pos_cursor(1,0);
        }
        NEXT_STATE: receive_data
     STATE:  receive_data
        CONDITIONS: {aux_change}
        ACTIONS:
        {
          displayf("Transmission %d received:  %04x    Press spacebar to send ACK.          \n",
                count, curr);
        }
        NEXT_STATE: send_ack
        CONDITIONS: {count > 10}
        ACTIONS:
        {
         display_prompt("End of test.                                         ");
        }
     STATE: send_ack
        CONDITIONS: KEYBOARD " "
        ACTIONS:
        {
         if(count <= 10)
           {
            write_aux(mask);
            count++;
            mask = (mask ^ 0x0100);
           }
        }
        NEXT_STATE: receive_data
```

NOTE: If you designate more than one lead as control, you might need to compare *prev_aux_value* with *curr_aux_value* to determine if the lead you are interested in is the one that changed. Here, since there is only one input-control lead on each side, the event *aux_change* is sufficient to signal and to acknowledge transmission. The value of *prev_aux_value* does not have to be checked.

set_aux_reg

Synopsis

```
extern void set_aux_reg(reg_value_word);
unsigned short reg_value_word;
```

Description

The CIO chip may be reconfigured by the user via the *set_aux_reg* routine.

NOTE: At present, the initial configuration of the Master Interrupt Control Register is (0x0082). The initial configuration of the Master Configuration Control Register is (0x0194).

Inputs

The only input is a sixteen-bit variable. The high byte is the CIO register number; the low byte is the value to store in the register number. For register numbers and their values, consult Appendix B in Zilog's *Z8036 Z-CIO/Z8536 CIO Counter/Timer and Parallel I/O Unit Technical Manual*, March 1982.

Example

The Master Configuration Control Register allows for selective enabling/disabling of the CIO ports. Port A's input/output is reflected in the least-significant byte of *reg_value_word*. Port B's input/output is reflected in the most-significant byte of *reg_value_word*.

NOTE: Port C of the CIO chip is used internally and is not available to the user of the INTERVIEW.

Suppose you want to disable port B input, output, and interrupts (ports A and C enabled) in one state, and in another state restore the original configuration (ports A, B, and C enabled):

```
LAYER: 1
    STATE: reconfigure_chip
        CONDITIONS: ENTER_STATE
        ACTIONS:
        {
         set_aux_reg(0x0114);
        }
    STATE: restore_original_config
        CONDITIONS: ENTER_STATE
        ACTIONS:
        {
         set_aux_reg(0x0194);
        }
```

72 Other Library Tools

The C structures, variables, and routines in this section provide additional programming tools not specific to any particular protocol. Most of these tools approximate layer-independent conditions or actions. Refer to Section 30 for more detailed explanations of the purposes of specific conditions and actions. Sometimes the name of the variable or routine is sufficient for identifying its related spreadsheet token. When this is not the case, the information is provided below.

72.1 Structures

Use the structures *tm*, *crnt_tm*, and *prev_tm* listed in Table 72-1 to monitor the current and previous date and time. Each minute the values in *crnt_tm* are copied to *prev_tm*. Then *crnt_tm* is updated. These structures are used to produce the date/time displays at the top of Run-mode screens and the Date/Time Setup screen.

The variables *flag_struct.prev*, *flag_struct.current*, and *flag_struct.old* (in the *flag_struct* structure) are used each time a flag is incremented, decremented, or set to a particular value. The current, previous, and old values these variables represent work the same way as their counterparts in the counter structure, discussed fully in Section 65.1(A).

> NOTE: The purpose of flags is to make it easy for the user to isolate selected bits in a variable. The translator does most of the work of flags by taking the user's flag masks and coding them in C. Flags constructed entirely in C bypass the translator and require the programmer to create the flag-mask code normally generated by the translator.

Before using the timeout routines included in this section, declare an instance of the *timeout* structure shown in Table 72-1. Refer to the *timeout_restart_action* and *timeout_stop_action* routines for examples of how to use this structure.

The *keyboard* structure stores the value of the most recent ASCII key used. The structure variable *keyboard.value* is updated only by the fast-event variable *keyboard_new_key*.

Table 72-1
Structure Fields—Other Library Tools

Type	Variable	Value (hex/decimal)	Meaning
Structure Name: keyboard			Declared as type *extern struct*. Declared automatically if program KEYBOARD condition is used. Updated by *keyboard_new_key* event variable. Reference the structure variable as follows: *keyboard.value*.
char	value		ASCII value of key just executed.
Structure Name: tm			Structure of time of day. Declared as type *extern struct*. Reference a structure variable as follows: *tm.tm_sec*.
int	tm_sec	*0-3b/0-59*	Seconds after the minute. Not currently updated; always set to -1.
int	tm_min	*0-3b/0-59*	Minutes after the hour.
int	tm_hour	*0-17/0-23*	Hours since midnight.
int	tm_mday	*1-1f/1-31*	Day of month.
int	tm_mon	*0-b/0-11*	Months since January.
int	tm_year		Years since 1900.
int	tm_wday	*0-6*	Days since Sunday. Not currently updated; always set to -1.
int	tm_yday	*0-16d/0-365*	Days since January 1. Not currently updated; always set to -1.
int	tm_isdst		Daylight Savings Time flag. Not currently updated; always set to -1.
Structure Name: crnt_tm			Structure of current time of day. Updated every minute. Declared as type *extern struct tm*.
Structure Name: prev_tm			Structure of previous time of day, one minute ago. Declared as type *extern struct tm*.
Structure Name: flag_struct			Structure of a flag. Declared as type *struct*. Declared automatically if a program flag is used. Program flags assigned to structure as follows: *struct flag_struct flag_name*. Reference a structure variable as follows: *flag_name.current*.
unsigned short	prev		When converting a flag action to C, the translator compares *prev* with *current* to determine whether flag has changed. Then *prev* is updated to *current* and *flag_name_change* is signaled.
unsigned short	current		This value of flag is acted on directly by program actions.
unsigned short	old		When converting a flag condition to C, the translator compares *old* with *current* to determine whether true condition is new (transitional). After program logic has examined flag, *old* is updated to *prev*.

Table 72-1 (continued)

Type	Variable	Value (hex/decimal)	Meaning
Structure Name:	timeout		Structure of a timeout. Declared as type *struct*. Declared automatically if a program timeout is used. Program timeouts assigned to structure as follows: *struct timeout* name. Reference a structure variable as follows: *timeout*_name.*event_id*.
unsigned long	event_id		Four bytes of a 6-byte timeout, containing the segment number and offset. *Timeout*_name_*stop* routines set this event id to zero.
unsigned short	event_id_uid		Two bytes of a 6-byte timeout which uniquely identify (uid) the timeout. Do not try to assign a value to this variable.

72.2 Variables

All of the variables in Table 72-2 are valid in either emulate or monitor mode.

(A) Monitoring Events

The event variables in Table 72-2 are *fevar_time_of_day*, *flag_name_change*, *timeout_name_expired*, *signal_name*, *keyboard_new_key*, and *keyboard_new_any_key*.

Event variable *fevar_time_of_day* comes true once a minute. An example of how to use this variable is provided in Section 57.1. This event variable is part of the spreadsheet TIME condition.

The event variable *keyboard_new_key* is used by the translator in a spreadsheet KEYBOARD condition. It comes true when any ASCII key is pressed. The event *keyboard_new_any_key*, on the other hand, comes true when an ASCII or other keyboard key is pressed. The only keys which will not trigger this event are [EXIT], [CAPS LOCK], and [DONE].

(B) Status Variables

Status variables are those in Table 72-2 that do not include *event* in the Type column. Their associated event variables guarantee that they are updated and tested.

Time and date variables are updated by *fevar_time_of_day*. Variables *crnt_time_of_day*, *prev_time_of_day*, *crnt_date_of_day*, and *prev_date_of_day* are older versions of variables that belong to the *crnt_tm* and *prev_tm* structures. The C translator uses these older versions when it construct time-of-day conditions (e.g., CONDITIONS: TIME 1614).

The status variable *keyboard_any_key* is updated by the fast-event variable *keyboard_new_any_key*.

Table 72-2
Other Library Variables

Type	Variable	Value (hex/decimal)	Meaning
extern fast_event	fevar_time_of_day		True once per minute. Line Setup configured for emulate or monitor mode.
extern event	flag_*name*_change		This event must be signaled by the program itself; it is not "external" to the program. The translator signals this event as part of the FLAG increment, decrement, or set action. Line Setup configured for emulate or monitor mode.
extern event	timeout_*name*_expired		This event must be signaled by the program itself. It is not "external" to the C program. The translator signals this event as part of the *timeout_restart_action* routine. Line Setup configured for emulate or monitor mode.
extern event	signal_*name*		True when the named signal is the argument in a *signal* routine. Spreadsheet-token equivalent is ON_SIGNAL name. Line Setup configured for emulate or monitor mode.
extern volatile unsigned short	crnt_time_of_day	*0-9370-2359*	Current time is stored in this variable. Updated as soon as time changes. Line Setup configured for emulate or monitor mode.
extern volatile unsigned short	prev_time_of_day	*0-9370-2359*	Current time is stored in this variable. Updated when time changes, but only after logic has had a chance to compare current and previous time. Line Setup configured for emulate or monitor mode.
extern volatile const unsigned char	crnt_date_of_day	*1-1f/1-31*	Current date is stored in this variable. Updated as soon as date changes. Line Setup configured for emulate or monitor mode.
extern volatile const unsigned char	prev_date_of_day	*1-1f/1-31*	Current date is stored in this variable. Updated when date changes, but only after logic has had a chance to compare current and previous date. Line Setup configured for emulate or monitor mode.
extern fast_event	keyboard_new_key		True when any ASCII key is pressed. Line Setup configured for emulate or monitor mode.

Table 72-2 (continued)

Type	Variable	Value (hex/decimal)	Meaning
extern fast_event	keyboard_new_any_key		True when any key is pressed. The only exceptions are [EDIT], [CAPS LOCK], and [DONE]. Line Setup configured for emulate or monitor mode.
extern volatile unsigned short	keyboard_any_key		Identifies last key or key-combination executed. Line Setup configured for emulate or monitor mode.
		0-7f/0-127	ASCII keys
		80-17f/ 128-383	not used
			Field entry keys:
		180/384	[HEX]
		181/385	[NOT EQUAL]
		182/386	[BIT MASK]
		183/387	[FLAG]
		184/388	[DON'T CARE]
		185/389	[CLEAR]
		186/390	[SHIFT]-[CLEAR]
		187/391	[CTRL]-[CLEAR]
		18a/394	[TAB]
		18b/395	[SHIFT]-[TAB]
		18c/396	[SHIFT]-[FLAG]
		18d/397	[CTRL]-[FLAG]
		18e/398	[SHIFT]-[BIT MASK]
		18f/399	[CTRL]-[BIT MASK]
		190/400	[SHIFT]-[DON'T CARE]
		191/401	[CTRL]-[DON'T CARE]
		192/402	[CTRL]-[TAB]
		193/403	[SHIFT]-[NOT EQUAL]
		194/404	[CTRL]-[NOT EQUAL]
		195/405	[SHIFT]-[HEX]
		196/406	[CTRL]-[HEX]
		197/407	[F1]
		198/408	[F2]
		199/409	[F3]
		19a/410	[F4]
		19b/411	[F5]
		19c/412	[F6]
		19d/413	[F7]
		19e/414	[F8]
		10d/269	[RETURN]
		1a0/416	[CTRL]-[RETURN]
		1a1/417	[SHIFT]-[RETURN]

(keyboard_any_key variable continued on next page)

Table 72-2 (continued)

Type	Variable	Value (hex/decimal)	Meaning
	(keyboard_any_key continued)		*Editing Keypad Keys (cont):*
		1a2/418	INSERT LINE
		1a3/419	INSERT CHAR
		1a4/420	DELETE LINE
		1a5/421	DELETE CHAR
		1a6/422	SHIFT-INSERT LINE
		1a7/423	SHIFT-INSERT CHAR
		1a8/424	SHIFT-DELETE LINE
		1a9/425	SHIFT-DELETE CHAR
		1aa/426	CTRL-INSERT LINE
		1ab/427	CTRL-INSERT CHAR
		1ac/428	CTRL-DELETE LINE
		1ad/429	CTRL-DELETE CHAR
			Utility Keys:
		1b0/432	PROGRAM
		1b1/433	RUN
		1b2/434	LOAD
		1b3/435	SAVE
		1b4/436	FREEZE
		1b5/437	RECORD
		1b7/439	PRINT
		1b8/440	XEQ
		1ba/442	□
		1bb/443	SHIFT-□
		1bc/444	CTRL-□
		1bd/445	SHIFT-PROGRAM
		1be/446	CTRL-PROGRAM
		1bf/447	SHIFT-RUN
		1c0/448	CTRL-RUN
		1c1/449	SHIFT-LOAD
		1c2/450	CTRL-LOAD
		1c3/451	SHIFT-SAVE
		1c4/452	CTRL-SAVE
		1c5/453	SHIFT-FREEZE
		1c6/454	CTRL-FREEZE
		1c7/455	SHIFT-RECORD
		1c8/456	CTRL-RECORD
		1c9/457	SHIFT-EDIT
		1ca/458	CTRL-EDIT
		1cb/459	SHIFT-PRINT
		1cc/460	CTRL-PRINT
		1cd/461	SHIFT-XEQ
		1ce/462	CTRL-XEQ

(keyboard_any_key variable continued on next page)

Table 72-2 (continued)

Type	Variable	Value (hex/decimal)	Meaning
	(keyboard_any_key continued)		*Pure Cursor Keys (cont):*
		1d0/464	↑
		1d1/465	SHIFT-↑
		1d2/466	CTRL-↑
		1d3/467	←
		1d4/468	SHIFT-←
		1d5/469	CTRL-←
		1d6/470	HOME
		1d7/471	SHIFT-HOME
		1d8/472	CTRL-HOME
		1d9/473	→
		1da/474	SHIFT-→
		1db/475	CTRL-→
		1dc/476	↓
		1dd/477	SHIFT-↓
		1de/478	CTRL-↓
			Cursor Keypad Keys:
		1e0/480	RUB OUT
		1e1/481	MARK
		1e2/482	SHIFT-MARK
		1e3/483	BRK
		1e4/484	SHIFT-BRK
		1e5/485	ROLL BACK
		1e6/486	PREV PAGE
		1e7/487	ROLL FWD
		1e8/488	NEXT PAGE
		1e9/489	CTRL-MARK
		1ea/490	CTRL-BRK
		1eb/491	SHIFT-ROLL BACK
		1ec/492	CTRL-ROLL BACK
		1ed/493	SHIFT-ROLL FWD
		1ee/494	CTRL-ROLL FWD
		1ef/495	SHIFT-PREV PAGE
		1f0/496	CTRL-PREV PAGE
		1f1/497	SHIFT-NEXT PAGE
		1f2/498	CTRL-NEXT PAGE
		1f3/499	SHIFT-RUB OUT
		1f4/500	CTRL-RUB OUT

(keyboard_any_key variable continued on next page)

Table 72-2 (continued)

Type	Variable	Value (hex/decimal)	Meaning
	(keyboard_any_key continued)		*Other Keys:*
		1f5/501	CTRL-1
		1f6/502	CTRL-2
		1f7/503	CTRL-3
		1f8/504	CTRL-4
		1f9/505	CTRL-5
		1fa/506	CTRL-6
		1fb/507	CTRL-7
		1fc/508	CTRL-8
		188/392	CTRL-9
		189/393	CTRL-0
		1fd/509	CTRL--

72.3 Routines

timeout_restart_action

Synopsis

```
extern void timeout_restart_action(timeout_name_ptr, value, function);
struct * timeout_name_ptr
  {
   unsigned long event_id;
   unsigned short event_id_uid;
  };
unsigned short value;
void function ();
```

Description

This routine starts a named timeout timer running down, starting at a specified value. When the timer reaches zero, a named function is called. The *timeout_restart_action* routine, preceded by a call to the *timeout_stop_action* routine, is the equivalent of the softkey TIMEOUT name RESTART action on the Protocol Spreadsheet.

Inputs

The first parameter is a pointer to the timeout structure. See Table 72-1 for further explanation of the *timeout* structure.

The second parameter is the starting value of the timeout timer in milliseconds.

The third parameter is the name of a routine to be called when the timeout expires. The routine may include the following statement: *timeout_name.event_id = 0;*. Timeout-stop actions set this event ID to zero. This action is not strictly necessary here, since the timeout has already expired; but the action may make the processing of subsequent stop actions slightly more efficient.

The body of the routine to be called may also include this statement: *signal(timeout_*name*_expired);*. In a softkey-entered TIMEOUT RESTART action, both statements are included in a routine called *timeout_*name*_isp*.

> NOTE: The routine named in the third parameter is an interrupt service process (isp). A long definition for this routine makes the processing of *timeout_restart_action* unpredictable.

Example

When a frame is sent, start a timeout timer at 2 seconds. When it expires, sound the alarm. If another frame is sent before the 2 seconds expires, stop the current timer and restart the timeout.

```
{
 struct timeout
  {
   unsigned long event_id;
   unsigned short event_id_uid;
  };
 struct timeout timeout_example;
 extern event timeout_example_expired;
 void timeout_example_isp ()
  {
   timeout_example.event_id = 0;
   signal(timeout_example_expired);
  }
}
LAYER: 2
     STATE: example_of_timeout
        CONDITIONS: FRAME_SENT
        ACTIONS:
        {
         timeout_stop_action(&timeout_example);
         timeout_restart_action(&timeout_example, 2000, timeout_example_isp);
        }
        CONDITIONS:
        {
         timeout_example_expired
        }
        ACTIONS: ALARM
```

Here is a version of the program that accomplishes the same result without an action to signal the timeout event:

```
{
 struct timeout
  {
   unsigned long event_id;
   unsigned short event_id_uid;
  };
 struct timeout timeout_example;
 extern void sound_alarm();
}
LAYER: 2
      STATE: example_of_timeout
         CONDITIONS: FRAME_SENT
         ACTIONS:
         {
          timeout_stop_action(&timeout_example);
          timeout_restart_action(&timeout_example, 2000, sound_alarm);
         }
```

timeout_stop_action

Synopsis

```
extern void timeout_stop_action(timeout_name_ptr);
struct * timeout_name_ptr
  {
   unsigned long event_id;
   unsigned short event_id_uid;
  };
```

Description

This routine stops a named timeout timer, preventing it from expiring. The softkey equivalent of this routine is the TIMEOUT name STOP action on the Protocol Spreadsheet. *timeout_stop_action* also precedes the call to the *timeout_restart_action* in the spreadsheet TIMEOUT name RESTART action.

Inputs

The only parameter is a pointer to the *timeout* structure. See Table 72-1 for further explanation of the timeout structure.

Example

In this example, if the user presses the [S] key, the timeout timer will not expire and the alarm will not sound (until another frame is sent and the timeout is restarted).

```
{
 struct timeout
  {
   unsigned long event_id;
   unsigned short event_id_uid;
  };
 struct timeout timeout_example;
 extern void sound_alarm();
}
```

```
LAYER: 2
     STATE: stop_a_timeout
        CONDITIONS: FRAME_SENT
        ACTIONS:
        {
         timeout_stop_action(&timeout_example);
         timeout_restart_action(&timeout_example, 2000, sound_alarm);
        }
        CONDITIONS:  KEYBOARD "Se"
        ACTIONS:
        {
          timeout_stop_action(&timeout_example);
        }
```

Index

Synopsis

```
extern char * index(string, character);
char * string;
char character;
```

Description

This routine searches for an instance of a character starting at the beginning of a specified list. The routine is used by the C translator to convert CONDITIONS: KEYBOARD softkey entries into C. This routine must be declared.

Inputs

The first parameter is a list of characters to be searched.

The second parameter is the character to be searched for in the list.

Returns

This routine returns a pointer to the first instance of the specified character, or zero if it does not occur.

Example

In the example below, the following test is established: when a key is pressed on the keyboard, search for a match to the keyboard character in the string " abc ". If it is found, sound the alarm.

```
{
 extern char * index();
 extern fast_event keyboard_new_key;
 extern struct keyboard
  {
   char value;
  };
 extern struct keyboard keyboard;
}
```

```
LAYER: 1
    STATE: Index_example
        CONDITIONS:
        {
         (keyboard_new_key && index(" abc ", keyboard.value))
        }
        ACTIONS: ALARM
```

Let's suppose that the user presses the space bar. In this case, the returned pointer will be pointing to the blank preceding the "a." If *rindex* had been used, the returned pointer would be pointing to the blank following the "c." As long as any non-null character is returned, the condition is *true*.

rindex

Synopsis

```
extern char * rindex(string, character);
char * string;
char character;
```

Description

This routine searches for an instance of a character starting at the end of a specified list. This routine must be declared.

Inputs

See *index*.

Returns

See *index*.

Example

See *index*.

load_program

Synopsis

```
extern void load_program(filename_ptr)
const char * filename_ptr;
```

Description

The *load_program* routines allows you to link programs together while the unit is in Run mode. When a call to *load_program* is encountered in a spreadsheet program, the current program is exited. The program named as the argument in the routine is

loaded and run. When you return to Program mode, the program displayed on the Protocol Spreadsheet will be the one just loaded. If *load_program* fails, you are returned to the main menu screen in Program mode.

Inputs

The only input is the absolute pathname, prefixed by the device name, of the file to be loaded. Valid device names are "HRD," "FD1," and "FD2."

Example

In the example below, at the successful conclusion of the last of a series of tests in module 18, a program for module 19 will be loaded and run.

```
LAYER: 3
    STATE: test_26
       CONDITIONS: ENTER_STATE
       ACTIONS: SEND DIAG
       CONDITIONS: RCV CLEAR_CONF
       ACTIONS: TRACE "Test_26 passed"
       {
        load_program("FD1!usr!module_19");
       }
```

lock

Synopsis

```
#include <stdio.h>
extern void lock(lock_variable_ptr);
int * lock_variable_ptr;
```

Description

The *lock* routine implements a lock using the integer variable pointed to by the routine parameter. If the lock variable is currently locked, the task goes to sleep. When an unlock on the same variable occurs (within an independent task), the task invoking the lock function will attempt to claim the lock. If successful, the task is executed; otherwise, it goes back to sleep until the next unlock.

> NOTE: If locking is used at any place in the program, all related or possibly concurrent routines must also use the locking functions.

> NOTE: The lock variable should always be defined as a global integer, never as local to a function. The lock variable should never be altered by the user program or deadlock can occur. Deadlock also results if the lock is invoked twice within the same task without an intervening unlock.

Inputs

The only parameter is a pointer to the lock variable.

Example

Two tasks concurrently write to their own file streams. (The file streams are local to the routine *write_fox*, making them independent of each other even though they have the same name.) However, during the *fclose* operation (which automatically calls *fflush*), both tasks need to write to the same file. The locking routines ensure that the writes to the file occur sequentially, not concurrently.

```
{
#include <stdio.h>
const char data [] = "((FOX))\n";
int key;
void write_fox()
 {
  FILE * stream_ptr;
  size_t n;
  lock(&key);
  if((stream_ptr = fopen("FD2/usr/buff01", "a")) == 0)
    display_prompt("Cannot open file.                                    ");
  else
    display_prompt("File opened.                                         ");
  n = fwrite(data, 1, sizeof(data)-1, stream_ptr);
  pos_cursor(1,0);
  if(n != (sizeof(data)-1))
    displayf("Write error.                                             \n");
  else
    displayf("Write completed.                                         \n");
  if(fclose(stream_ptr) != 0)
    displayf("Either file is already closed, or close cannot be executed.  ");
  else
    displayf("File closed.                                              ");
  unlock(&key);
 }
}
LAYER: 1
       TEST: a
    STATE: write_and_signal
       CONDITIONS: RECEIVE STRING "THE QUICK BROWN FOX"
       ACTIONS: SIGNAL xyz
       {
        write_fox();
       }
       TEST: b
    STATE: write_only
       CONDITIONS: ON_SIGNAL xyz
       ACTIONS:
       {
        write_fox();
       }
```

unlock

Synopsis

```
#include <stdio.h>
extern void unlock(lock_variable_ptr);
int * lock_variable_ptr;
```

Description

The *unlock* routine implements the inverse of the *lock* routine using the same integer variable. Sleeping tasks will be woken up to retry their attempt to claim the lock. One will succeed, and the rest will go back to sleep. See also *lock* routine.

Inputs

The only parameter is a pointer to the lock variable.

Example

See *lock* routine.

signal

Synopsis

```
extern void signal(signal_name);
```

Description

This routine conveys instructions to other tests and layers where conditions are monitoring the signal by name. The softkey equivalent of this routine is the SIGNAL action on the Protocol Spreadsheet.

Inputs

The only parameter is a name descriptive of the event being signaled.

Example

```
LAYER: 2
    STATE: signal_routine
        CONDITIONS: RCV FRMR
        ACTIONS:
        {
         signal(signal_link_down);
        }
        CONDITIONS: ON_SIGNAL link_down
        ACTIONS: ALARM
```

Here is a related example, this time with the signal detection also given in C. Note that a signal automatically generates an "event" that can be detected alone in a *waitfor* clause.

```
{
 extern event link_down;
}
LAYER: 2
     STATE: signal_event
        CONDITIONS: RCV FRMR
        ACTIONS:
        {
         signal(link_down);
        }
        CONDITIONS:
        {
         link_down
        }
        ACTIONS: ALARM
```

sound_alarm

Synopsis

extern void sound_alarm();

Description

This routine will sound the alarm. The softkey equivalent of this routine is the ALARM action on the Protocol Spreadsheet.

Example

When a bad BCC is detected on the DTE side of the link, sound the alarm.

```
LAYER: 1
     STATE: example
        CONDITIONS: DTE BAD_BCC
        ACTIONS:
        {
         sound_alarm();
        }
```

start_rcrd_play

Synopsis

extern void start_rcrd_play();

Description

Depending on the Line Setup configuration, this routine activates data recording or playback. If the Line Setup menu shows **Mode:** MONITOR, **Source:** DISK, the routine controls playback. In all other cases, it initiates recording.

Unless your recording source is RAM, make a call to *fclose* in programs containing disk I/O routines (Section 68) before you start to record (or resume playback). If you don't, the file will be closed automatically as soon as recording (or playback) begins, even if processes on the file have not been completed. (Using the RECORD key to activate recording or resume playback will have the same effect.)

Example

```
LAYER: 1
    STATE: example
        CONDITIONS: KEYBOARD " "
        ACTIONS:
        {
         start_rcrd_play();
        }
```

suspend_rcrd_play

Synopsis

```
extern void suspend_rcrd_play();
```

Description

Depending on the Line Setup configuration, this routine suspends data recording or playback. If the Line Setup menu shows Mode: MONITOR, Source: DISK, the routine controls playback. In all other cases, it suspends recording. Once recording or playback is suspended, resume it with a call to *start_rcrd_play*.

Unless your recording source is RAM, do not call disk I/O routines (Section 68) until you suspend recording (or playback). If you do, the disk I/O operation will fail.

> NOTE: Although playback is immediately suspended when *suspend_rcrd_play* is executed, the screen display continues until the character buffer's contents are fully displayed. (For bit-image data, the FIFO must empty.) At slower playback speeds, you may notice a slight delay before the display actually freezes.

Example

```
LAYER: 2
    STATE: example
        CONDITIONS: KEYBOARD " "
        ACTIONS:
        {
         suspend_rcrd_play();
        }
```

send_key

Synopsis

```
extern void send_key(number_of_keys, keys_ptr);
unsigned char number_of_keys;
unsigned short * keys_ptr;
```

Description

This routine sends a specific keystroke (or sequence of keys) during Run mode, as though the operator pressed the key. It also may be used to change the Run-mode display.

Inputs

The first parameter specifies the number of keys to be sent.

The second parameter is a pointer to an array of *shorts*. This array lists the keys to be sent. To send keyboard keys, use the values listed in Table 72-2 for the *keyboard_any_key* variable. To change the Run-mode display, send two keys. The first "key" always has a value of 0xff75. The second "key" identifies the desired display-screen. Use the values listed in Table 64-1 for the *crnt_display_screen* variable.

Example

For this example, assume you are playing back data from a disk and that the initial Run-mode screen is the dual-line data display. After a five-second pause, playback is slowed as though you pressed ⊡. As soon as a bad BCC is detected on the DTE side, the data display will change to the Layer 2 Protocol Trace screen.

```
{
 unsigned short keys [] = {0xff75, 0x42};
 unsigned short slow_down [] = {0x1dc};
}
LAYER: 2
     STATE: change_displays
        CONDITIONS: ENTER_STATE
        ACTIONS: TIMEOUT pause RESTART 5
        CONDITIONS: TIMEOUT pause
        {
         send_key(1, slow_down);
        }
        CONDITIONS: DTE BDBCC
        ACTIONS:
        {
         send_key(2, keys);
        }
```

surrender_cpu

Synopsis

```
extern void surrender_cpu();
```

Description

This routine surrenders the CPU, placing the calling task onto the end of the ready queue. If no other tasks are currently ready to run, this routine returns.

Use *surrender_cpu* only when executing C code which started as part of an ENTER_STATE condition. It is useful in programs containing a task that only performs computations (i.e., no I/O operations like disk accesses). Make a call to *surrender_cpu* to give other tasks on the same CPU a chance to run.

Example

In the following example, one task on a CPU waits on a *rcvd_frame* event variable in order to count frame types. For each of the different frame types, another task displays the value of the counter. Without a call to *surrender_cpu*, the display task would monopolize the CPU, preventing the frame-counting task from running.

```
{
 extern event rcvd_frame;
 extern volatile const unsigned char rcvd_frame_type;
 unsigned short frame_type_count[256];
 void display_frame_type_count()
  {
   pos_cursor(7,12);
   displayf("%3u", frame_type_count[0];
   pos_cursor(7,22);
   displayf("%3u", frame_type_count[1];

   /* ... Continue to position and display count for each frame type */

  }
}
LAYER: 2
   TEST: display_frame_types
      STATE: only
         CONDITIONS: ENTER_STATE
         ACTIONS:
         {
          pos_cursor(3,23);
          displays(" FRAME COUNTS BY TYPE");
          pos_cursor(5,11);
          displays("INFO      RR      RNR      REJ      ...);
          while(1)
           {
            display_frame_type_count();
            surrender_cpu();
           }
         }
```

```
TEST: count_frame_types
  STATE: only
    CONDITIONS:
    {
     rcvd_frame
    }
    ACTIONS:
    {
     ++frame_type_count[rcvd_frame_type];
    }
```