Supermax

C Programming Guide

Dansk Data Elektronik A/S
1 May 1984

1. Introduction.

This manual serves a twofold purpose: First, it gives an introduction to how to write programs in the C language for execution on the Supermax computer. Second, it describes how to compile and link C programs.

The reader must be familiar with the contents of the "Supermax System Operation Guide" and with Kernighan and Ritchie's "The C Programming Language".

The reader is referred to the Supermax Operating System User's Manual for detailed information about system calls and standard subroutines.

Danish readers should note that the following identity exists between the Danish national characters and the standard ASCII characters:

ASCII:	[	\	]	{	¦	}
Danish:	Æ	Ø	Å	æ	ø	å

A few paragraphs are marked with astisks (*) in the margin. These paragraphs describe features that have not yet been implemented, but will be in the near future.

Dansk Data Elektronik A/S reserves the right to change the specifications in this manual without warning. Dansk Data Elektronik A/S is not responsible for the effects of typographical errors or other inaccuracies in this manual and cannot be held liable for the effects of the implementation and use of the structures described herein.

2. Data Representation.

The C dialect running on the Supermax computer represents data in the following way:

Type	Representation
char	1 byte
short int	2 bytes, most significant byte in lowest address
long int	4 bytes, most significant byte in lowest address
int	same as long int
float	same as double
double	13 decimal digits and an 8-bit power of 10. the number is stored in 8 bytes in the following layout: Byte 0-6: 14 BCD digits. The first digit is the sign, 0 representing +, 9 representing -. The remaining 13 digits are the mantissa with negative numbers represented in 10's complement. The decimal point is assumed after the first of these 13 digits. Byte 7: The decimal exponent plus 128. Represented in binary form.
Pointers	4 bytes, most significant byte in lowest address.

Variables of type char may have values between -128 and +127.
Variables of type unsigned char may have values between 0 and +255.

Variables of type short int (or short) may have values between -32,768 and +32,767.
Variables of type unsigned short int (or unsigned short) may have values between 0 and +65535.

Variables of type long int (or long or int) may have values between -2,147,483,648 and +2,147,483,647.
Variables of type unsigned long int (or unsigned long or unsigned int or unisgned) may have values between 0 and +4,294,967,295.

Variables of type float or double may have values between -9.999999999999E+127 and -1.000000000000E-128 or the value 0 or values between +1.000000000000E-128 and +9.999999999999E+127.

The follwing are a few examples of the representation of various floating point numbers:

The number 3.141592653590 is represented in the following manner (the contents of all 8 bytes are given in hexadecimal):

03 14 15 92 65 35 90 80

The number -3.141592653690 is represented in the following manner:

96 85 84 07 34 64 10 80

Note that adding the 14 BCD digits of the mantissa and sign of the above two numbers will yield all zeroes.

The number 31.41592653590 is represented in the following manner:

03 14 15 92 65 35 90 81

The number 0.3141592653590 is represented in the following manner:

03 14 15 92 65 35 90 7f

The number 1 is represented in the following manner:

01 00 00 00 00 00 00 80

The number -1 is represented in the following manner:

90 00 00 00 00 00 00 7f

In this case normalization has caused the mantissa to be shifted one digit left.

2.1. Conversion of Parameters.

When a procedure is called, all its actual parameters are converted to 4-byte entities (except for floating point numbers, which remain 8-byte entities) before being passed to the procedure. There is therefore no difference between declaring a formal parameter of type char, short, or long, and in all descriptions in this manual and in the Supermax Operating System User's Manual the type int is used for such parameters.

3. Programs in General.

The C programming language differs from languages such as Pascal or Fortran in that there is no difference between a main program and a subroutine (procedure or function). The main program of a C program is merely a subroutine called main.

When a C program is executed, execution does not start directly at the entry point to main. First, a so-called C Run-time Startup module is executed. This module, which is supplied with the C compiler and which is linked to the C program, executes a few instructions that establish the proper environment for the C program. This includes constructing parameters for main and defining the so-called error variables, which are described below. When this has been done, the C Run-time Startup module calls main. The C Run-time Startup module is located in the file /lib/crt0.o.

But the programmer doesn't really "see" the C Run-time Startup module. The programmer must merely supply a routine called main; this routine will be the first one executed. If a return is made from main to the C Run-time Startup module, it will perform an exit(0) call.

3.1. The main Routine.

The routine called main is normally declared in the following manner:

```
main(argc, argv, envp)
  int argc;
  char *argv[], *envp[];

{
  ...
  /* program code */
  ...
}
```

The three parameters argc, argv, and envp are suplied by the operating system through the C Run-time Startup module. Often envp is omitted, as its value may be obtained in a different manner, as described below.

argc is the number of arguments passed from the command interpreter (vox or shell). argv is an array of pointers to the arguments passed from the command interpreter.

Let us, for example, assume that the program is stored in a file called prog and is started with the command

```
prog alpha beta gamma
```

to vox or shell. In this case argc will be 4 and argv[0] will be the address of a null-terminated string containing "prog", argv[1] will be the address of a null-terminated string containing "alpha", argv[2] will be the address of a null-termiated string containing "beta", and argv[3] will be the address of a null-terminated string containing "gamma".

Note that the name of the program is itself passed as a parameter.

envp is an array of pointers to strings describing the environment in which the program is executing. This environment is set up by vox or shell. For example, envp[0] may be the address of a null-terminated string containing "PATH=:/bin:/etc/bin", envp[1] may be the address of a null-terminated string containing "HOME=/usr/aragorn/ring", and envp[2] may be zero, which is an indicator that there are no more strings in the environment. The interpretation of these strings is a matter of convention, but it is customary that these strings contain an identifier, an equals sign, and a text, being the "value" of the identifier. By convention, for example, the value of the identifier HOME (being "/usr/aragorn/ring" in the above example) is the operator's home directory. The environment is described in greater detail in the Supermax Operating System User's Manual part 7.

If the envp parameter is not included in the declaration of main the environment may still be found by including the following declaration in the program:

```
extern char **environ;
```

This external variable has exactly the same value as the envp parameter.

3.2. The Standard Include File.

A file /usr/include/std.h is supplied with the C compiler. This file may be included in a C program in the following manner:

```
#include <std.h>
```

Including this file gives the programmer access to a number of definitions:

1) A few usefull constants are defined:
   TRUE and YES, both having the value 1.
   FALSE and NO, both having the value 0.
   NULL, having the value 0.

2) The standard I/O devices are named:
   STDIN, having the value 0.
   STDOUT, having the value 1.
   STDERR, having the value 2.
   STDLIST, having the value 3.

3) The maximum length of a pathname component is given as
   MAXSUBNAME, having the value 14.

4) The file /usr/include/fcntl.h is included. In this file constansts such as READ and WRITE are defined.

5) A number of words replacing {, |, and } are defined:

In the ASCII alphabet the characters [, \,], {, |, and } are reserved for national characters, and they are indeed used as extentions of the alphabet in German, French, Spanish, Danish, Norwegian, Swedish, and many other languages. Nevertheless, these characters are used in C, which means that C programs look very strange on non-English terminals and printers.

The standard include file contains the following definitions which help remedy this problem:

```
#define begin {
#define end }
#define then {
#define else_if } else if
```

```
#define otherwise } else {
#define end_if }
#define loop {
#define end_loop }
#define or ||
#define and &&
#define not !
#define forever for (;;)
```

These definitions enable the programmer to write pretty program structures such as:

```
main()
begin
  :
  :
end


if (a>8 or b<7) then
  :
  :
else_if (a>8 and b>=7) then
  :
  :
otherwise
  :
  :
end_if


while (not x) loop
  :
  :
end_loop


for (x=0; x<8; x++) loop
  :
  :
end_loop
```

```
do loop
  :
  :
end_loop while (x<8);


switch (x) begin
  :
  :
end


forever loop
  :
  :
end_loop
```

Although this greatly improves the legibility of C programs on non-English terminals and printers, the programmer should note that there is no check on the well-formedness of the structures. For example, the following structure is accepted by the compiler:

```
if (x>2) begin
  :
  :
end_loop
```

4. System Calls and Standard Subroutines.

A C program has at its disposal a number of routines in various libraries on the Supermax computer. Normally, the programmer uses these routines without worrying about how the routines perform their task.

However, a few general details about the routines may be helpful.

The routines may be devided into two groups: System calls and subroutines.

A system call is a routine that calls on the operating system to perform a certain task; for example, write a buffer to a file.

A subroutine is a routine that performs a certain amount of computation. It may or may not call on other subroutines or system calls to perform actions. Subroutines are, for example, the routine for computing the logarithm of a number and the routines for performing formatted input/output.

System calls are rather uniform in their behavior. For example, they all return the value -1 in case of an error. Subroutines are not so uniform.

4.1. The Supermax Operating System and Unix.

The Supermax Operating System is (with a few exceptions) a superset of the Unix System III operating system. Almost all the system calls and subroutines of Unix System III are found in the Supermax Operating System. However, the Supermax Operating System includes quite a few system calls and subroutines not found in Unix System III. Further, the Supermax Operating System gives the programmer a more detailed error reporting than Unix System III.

4.2. System Call Error Reporting.

All system calls return a value that informs the calling program about the success or failure of the operation. If the return value is non-negative, the system call was successful. If the return value is -1, an error occurred during the system call, and the operating system has stored additional error information elsewhere.

By including the following declarations in the C program, the programmer may access the error codes:

```
extern long errno;
extern long smoserr;
```

(These two long integers are located in the C Run-time Startup module.) If an error occured during a system call, the operating system stores the Supermax Operating System error code in smoserr (short for Supermax Operating System error) and the equivalent Unix error code in errno. If there is no equivalent Unix error code, -1 is stored in errno. The contents of these two variables are not changed if no error occurs.

A few system calls may return -1, even if they are successful. For example, the system call nice is used to set the priority of a process, and it returns the new priority. This priority may be -1. Therefore the return value -1 from nice may mean either that the call failed or that the new priority is -1. In order to find out which is the case, the programmer should set smoserr and/or errno to zero prior to executing nice. If no error occurred, the contents of the error variables will be unchanged.

Symbolic names for the Supermax Operating System error codes may be included in the program in the following manner:

```
#include <smoserr.h>
```

Symbolic names for the Unix error codes may be included in the program in the following manner:

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

The programmer should not include both these files, as some symbolic names are used in both, but with different values.

4.2.1. testerr and errout

Two standard subroutines exist that make error message handling easier. They are testerr and errout. Both these routines test the possible occurrence of an error, and if an error has indeed occured, the Supermax error number and a corresponding text is output. The text is taken from a set of texts stored in a partition created by the makeerr program (see the Supermax Operating System User's Manual part 8).

testerr is called with the result of a system call as its parameter. If this result is -1, the value of smoserr is inspected and an error message is output.

errout is called with a Supermax Operating System error code as its parameter. If this value is not zero, an error message is output.

Both routines return the value of their parameter.

testerr may, for example, be used in the following manner:

```
n=testerr(write(....));
```

If the write system call is successful, the value returned by write will be stored in n.

Normally, execution will continue after the occurrence of an error, but if the programmer includes the statement

```
extern short exonerr;
```

and sets this external variable to TRUE, testerr and errout will cause the process to exit after displaying the error message.

4.3. Privileged System Calls.

A number of system calls are "privileged". These system calls may only be performed by the super-user (user ID 0).

Even the super-user should be careful when using these system calls, as improper use may have disastrous results.

5. I/O Management.

5.1. Standard I/O Devices.

When a process is started, it normally inherits 4 open iounits from its parent process. These four iounits are known as

- the standard input device, whose iounit descriptor is 0
- the standard output device, whose iounit descriptor is 1
- the standard error device, whose iounit descriptor is 2
- the standard list device, whose iounit descriptor is 3

In the standard include file the symbolic names STDIN, STDOUT, STDERR, and STDLIST are found with the values 0, 1, 2, and 3, respectively. They may be used for the standard iounit descriptors.

Very often all of these iounits are the terminal at which the user is working, although vox or shell commands may redirect the standard i/o devices to other iounits.

It should be noted that the standard list device is not found in standard Unix systems.

Often, the standard i/o devices are used in the following manner: The standard input and output devices are used for communication with the terminal operator, the standard list device is used for program output that is to be saved for later use (typically redirected to a printer), and the standard error device is used for diagnostic messages.

5.2. Working with the Standard I/O Devices.

In this section we will discuss how i/o is normally performed on the standard i/o devices. The most common i/o device is a terminal. Therefore we will primarily consider how i/o to a terminal is performed. But what is said below applies to a very large extent to any kind of iounit. The reader should have no difficulty in applying the following information to other kinds of iounits.

The most common way to do i/o is to use the so-called 'standard i/o library'. This is a collection of subroutines that perform formatted i/o. Most of these routines are operating system independent in the sense that they are found in most systems that support C.

Before you use the standard i/o library, you should include the statement

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

in the program. This defines a set of symbolic values and macroes.

The function getchar() reads one charater from the standard input and returns its value. If the standard input device is a terminal, lines will be read from the terminal as required. If, for example, the terminal operator enters the text 'hello', the first call to getchar will return 'h', the second call will return 'e', the third and fourth call will both return 'l', the fifth call will return 'o', and the sixth call will return '\n' (new-line or line-feed), indicating the end of the line. A seventh call to getchar will cause another line of text to be read from the terminal.

If end-of-file is reached on the standard input (CTRL-D pressed on a terminal), getchar will return EOF. EOF is a symbolic constant defined in stdio.h with the value -1.

The function putchar(c) writes the character c to the standard output.

The following program reads characters from the standard input, converts lower case characters to upper case, and outputs the converted characters:

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

main()
begin
    char c;

    while ((c=getchar()) != EOF) loop
        if (c>='a' and c<='z') c += 'A'-'a';
        putchar(c);
    end_loop
end
```

To do i/o on data of other types than characters, the functions printf and scanf may be used. Both these functions take as parameters a character string containing a format specification, and a set of variables or values to be read or written.

The simplest case is where printf is used to output a character string:

```
printf("Hello, there!\n");
```

This statement outputs the text 'Hello, there!' and moves the cursor to the beginning of the next line. If the standard output is a file, the final new-line character will be stored in the file.

Now let us output integers:

```
int a,b;

a=21;
b=4;
printf("a is %d, b is %d, the sum is %d\n",a,b,a+b);
```

Here the character % is found three times in the first parameter to printf. These are format specifications. The three format specifications match the three additional parameters given to printf. The character following the % indicates the format used when converting the parameter value to a string of characters. In this case %d indicates decimal representation using as many character positions as required.

The above code will output

```
a is 21, b is 4, the sum is 25
```

and then move the cursor to the beginning of the next line.

We will not here go into detail with the different format specifications that can be used with printf. They are found in the Supermax Operating System User's Manual part 3. Let us mention just one more format specification, namely %s that is used to output a null-terminated string of characters:

```
char *str;

str="abcdefg";
printf("<X>str is %s",str);
```

This code will attempt to output '<X>str is abcdefg'. However, in the Supermax Operating System, the sequence <X> in the beginning of the buffer is an indication that the terminal screen should be cleared. See section 5.3.1 of the Supermax System Operation Guide. The above

code will therefore clear the terminal screen and output

```
str is abcdefg
```

leaving the cursor after the g. If the standard output device is a file, the entire text '<X>str is abcdefg' will be stored in the file.

scanf is used to do formatted input from the standard input device. The following code will read two integers and a string of characters into the variables a, b, and c, respectively:

```
int a,b;
char c[80];

if (scanf("%d%d%s",&a,&b,c)!=3) printf("Error in input\n");
```

The call scanf("%d%d%s",&a,&b,c) contains a format specification string that looks much like the string used in printf.

This call will read from the standard input, skipping spaces and new-lines, and interpret what it finds according to the format %d, that is, as a decimal integer. If a decimal integer is indeed found, its value is stored in the address &a, that is, in the variable a.

The scanning of the standard input will continue, skipping spaces and new-lines, and the next data found will be interpreted, if possible, as a decimal integer and stored in the variable b.

After this, the scanning of the standard input will continue, skipping spaces and new-lines, and the next data found will be interpreted as a string of characters that terminates at the next space or new-line. This string of character will be stored starting at the address c, that is in the character array c. (We here use the convention that the name of an array is the same as its address.)

If the terminal operator types

```
12 45
hello
```

the value 12 will be stored in a, the value 45 will be stored in b, and the string hello will be stored in c followed by a null-character.

scanf returns the number of format specifications matched. In this case all three were matched, so scanf returns 3. If, however, the terminal operator had typed

```
12
hello
```

it would have been impossible to find a match for the second %d, and scanf would have returned 1.

scanf will return EOF if the end-of-file is reached on the standard input device.

Note that scanf differs from printf in that the parameters for printf were the values to be written, whereas the parameters for scanf are addresses of variables, in which the data read is to be stored.

Now, suppose we want to write to the standard error or standard list device instead of the standard output device. For this purpose the subroutines putc and fprintf are used instead of putchar and printf.

The following calls writes the character 'z' to the standard error and standard list devices, respectively:

```
putc('z',stderr);
putc('z',stdlist);
```

The following calls writes the value of the integer variable i to the standard error and standard list devices, respectively:

```
fprintf(stderr,"i is %d\n",i);
fprintf(stdlist,"i is %d\n",i);
```

stderr and stdlist are defined in stdio.h. They are pointers to a data structure called FILE, of which more will be said later. For the standard input and output devices, pointers stdin and stdout are defined, so the calls printf("Hello\n") and fprintf(stdout,"Hello\n") are identical.

For the sake of completeness, let us add that there exists two functions, getc and fscanf, whose relationship to getchar and scanf is the same as the relationship of putc and fprintf to putchar and prinf. So the calls

```
c=getc(stdin);
fscanf(stdin,"%d",&a);
```

are identical to

```
c=getchar();
scanf("%d",&a);
```

Let us digress here slightly. When talking about printf and scanf, we might as well mention sprintf and sscanf. These two functions perform no i/o, instead they operate on strings. The code

```
char str[80];
int a;

a=4;
sprintf(str,"a is %d",a);
```

will place the characters 'a is 4' followed by a null-character in the character array str.

The code

```
int a,b;
char *c;

c="123 456";
sscanf(c,"%d%d",&a,&b);
```

will place the vaue 123 in a and 456 in b.

Back to i/o: The above-mentioned routines are not system calls. When printf is called, it constructs a character string and then invokes the system call write to do the actual writing. We will here consider five important system calls that perform terminal i/o.

The system calls write and writev both output a string of characters to an i/o device. Consider the following code:

```
char *c;

c="abc";
write(STDOUT,c,3);
writev(STDLIST,c,3);
```

Both system calls have three parameters:

- The iounit descriptor
- The address of the buffer to be written
- The number of bytes (characters) to be written

The difference between the system calls is that write outputs its buffer as it is, whereas writev outputs a 'variable length record', that is, a line of text. For terminals this amounts to terminating the output by moving the cursor to the next line. So the call write(STDOUT,c,3) will write 'abc' to the standard output device and leave the cursor after the c. The call writev(STDLIST,c,3) will write 'abc' to the standard list device and move the cursor to the beginning of the next line. If the standard list device is a file, the text 'abc\n' will be written to the file by writev.

Of course, the task performed by writev might equally well have been performed by write if the buffer had contained a new-line character. Why, then, have two system calls? The main reason is that in the Mikfile file system, lines of text are not seperated by new-line characters. Applying writev to an iounit always writes a line of text in the format required by that kind of iounit.

Here we must again digress and say a few words about using printf or fprintf on Mikfile files. These two routines use the write system call to perform their task. They will generate illegal file contents if they are applied to Mikfile files. The following code will work on any kind of iounit except Mikfile files:

```
int a;

a=4;
printf("a is %d\n",a);
```

The following code will work on any kind of iounit including Mikfile files:

```
int a;
char c[80];

a=4;
sprintf(c,"a is %d",a);
writev(STDOUT,c,strlen(c));
```

The subroutine call strlen(c) returns the number of non-null-characters in c.

The routines write and writev, like all other system calls, return -1 if an error occurs, in which case an error code is left in smoserr and errno.

The system calls read and readv both read a string of characters from an iounit. Consider the following code:

```
char c[10];
int n;

n=read(STDIN,c,10);
```

Let us suppose that the standard input device is a terminal. Associated with each terminal is an input buffer of 253 bytes. When the read system call is executed, the terminal operator is allowed to enter up to 253 bytes. The input operation terminates when the return key is pressed. If the operator typed, for example,

abcdefghijklmn

the terminal input buffer will contain these 14 characters followed by a new-line character. The system call read(STDIN,c,10) will take the first 10 of these 15 characters and store them in c. So c will contain 'abcdefghij'. read will return the number of characters transferred to c, in this case 10.

If the system call read(STDIN,c,10) is executed a second time, no i/o will be done, because there is already data in the terminal input buffer. The remaining 5 characters in the input buffer will be transferred to c, which will then contain 'klmn\n'. read returns 5.

A subsequent read call will cause terminal i/o to take place again.

If the standard input device is a file, read(STDIN,c,10) will read 10 bytes at a time until end-of-file is reached.

Now, let us look at readv:

```
char c[10];
int n;

n=readv(STDIN,c,10);
```

readv does not use the terminal input buffer. Instead, when the readv system call above is executed, the terminal operator will be allowed

to enter 10 characters - no more. If the characters typed are

```
alpha
```

these five characters will be stored in c with no final new-line character, and readv will return the value 5.

When operating on other kinds of iounits, readv reads a line of text in the format used on that kind of iounit. It is required that the line of text be not longer than the number of bytes requested.

Just as printf should be replaced by sprintf and writev when operating on Mikfile files, so scanf should be replaced by readv and sscanf.

If end-of-file is reached, read returns 0, indicating that nothing has been read. If readv returns 0, this indicates that an empty line of text has been read, which is not the same as an end-of-file. In case of end-of-file, readv returns -2. Both system calls return -1 in case of error.

Let us finally consider the system call edit. When applied to terminals this is a combined output and input operation. A line of text is output, the operator is allowed to edit it, and the modified text is returned to the program. Consider the following code:

```
char str[80];
int n;

strcpy(str,"alpha beta");
n=edit(STDIN,str,strlen(str),6);
```

The subroutine strcpy copies the string "alpha beta" to str.
edit takes four parameters:

- The iounit descriptor
- The buffer used in the operation
- The length of the buffer
- The cursor offset at the beginning of the operation

When edit is invoked, the old contents of str is presented on the terminal screen and the cursor is left at the b, which is the 6th character. The operator may now, if desired, change the contents of the string "alpha beta" and press RETURN. The modified buffer is then returned to str, and edit returns the number of charcters in the buffer, up to the last printable character.

For another example consider the following code:

```
char str[80];
int n;

strcpy(str,"<X'Yes or no? '>y");
n=edit(STDIN,str,strlen(str),0);
```

Here the buffer contains a control sequence. As described in section 5.3.1 of the Supermax System Operation Guide, such sequences cannot be modified by the edit operation. The X will cause the terminal screen to be cleared and the text

```
Yes or no? y
```

will appear. The cursor will be left at offset 0 relative to the first modifiable character, that is, on the y. The operator may now, if desired, modify this y and press RETURN. No other characters can be modified. edit will return the number of characters following the >.

The edit operation, of course, works only on character strings. To edit an integer the following code may be used:

```
char str[80];
int i;

sprintf(str,"%5d",i);
edit(STDIN,str,strlen(str),0);
sscanf(str,"%d",&i);
```

The format %5d used in the sprintf routine causes the converted integer to be stored right-justified in a 5 character field. If i has the value 25, sprinf(str,"%5d",i) will leave ' 25\0' in str.

If edit is applied to a file, it is converted into an equivalent readv operation. Consider the code

```
char str[80];
int n;

strcpy(str,"<X'Yes or no? '>y");
n=edit(STDIN,str,strlen(str),0);
```

If the standard input device is a file, the edit call above will be equivalent to readv(STDIN,&str[16],1);

5.2. I/O to Other Devices.

If a process wants to work with iounits that are not one of the four standard i/o devices inherited at the birth of the process, it must open or create these iounits itself.

If you want to use the standard i/o library (fprintf, fscanf, etc.), the iounit should be opended or created using fopen:

```
FILE *fp;

fp=fopen("/usr/bilbo/merry","r");
```

The first parameter to fopen is the pathname of the iounit to be opened. The second parameter specifies the desired open mode. The open modes allowed here are:

"r"	open for reading
"w"	open or create for writing
"a"	append; open for writing at end of file, or create for writing
"r+"	open for reading and writing
"w+"	open or create for reading and writing
"a+"	append; open or create for reading and writing at end of file

If the iounit is a file in the Unix file system, it is opened without reservation.

The data type FILE is defined in stdio.h. It should not be confused with an iounit descriptor which is a mere integer.

Once the iounit has been opened or created by fopen, data may be written to or read from the iounit using, for example, putc, fprintf, getc, or fscanf:

```
putc('z',fp);
fprintf(fp,"i is %d\n",i);
c=getc(fp);
fscanf(fp,"%d",&i);
```

When there is no need for the iounit any longer it should be closed in the following manner:

```
fclose(fp);
```

This call is automatically performed if the process terminates normally. If the process is aborted (for example, because of division by zero), the iounit is closed, but because some buffering takes place in these i/o routines, there is no guarantee that everything written by, say, fprintf will have been stored in the iounit.

If you want to use the basic system calls (read, write, etc.), the iounit should be opended or created using open or creat:

```
int ioud1, ioud2;

ioud1=open("/usr/bilbo/merry",O_RDONLY);
ioud2=creat("pip",0644);
```

open normally has two parameters, the pathname and the desired access mode. Occasionally, a third parameter is needed, but we will not go into this here. The access mode is in the above example 'read with no reservation'. All the possible access modes may be found in the Supermax Operating System User's Manual chapter 2.

creat has two parameters, the pathname and the protection bits. In the above example, the protection bits 644 (octal), which means read and write access for owner, read access for others. If creat is applied to an existing file, its length is truncated to zero. Except as noted in the Supermax Operating System User's Manual chapter 2, creat always opens the iounit for 'write with no reservation'.

Both open and creat return, if successful, an iounit descriptor, that is, an integer in the range 0 through 31. This integer is a unique identification within the process of the open iounit. The operating system guarantees that the number returned by open or creat is the lowest integer not currently used as an iounit descriptor.

Note that it is customary in Unix to refer to iounits as 'files' and iounit descriptors as 'file descriptors'. We feel that this terminology is misleading because it gives the uninitiated user the impression that i/o can be performed only on disk files.

Once the iounit has been opened or created, system calls such as read and write may be used:

```
n=read(ioud1,buf,10);
write(ioud2,buf,n);
```

The input or output operation is performed from the 'current position' in the iounit. The operating system maintains an iounit pointer associated with each open file or disk. This pointer indicates the position on the iounit where i/o is to take place. Immediately after the opening of an iounit the iounit pointer points to byte 0 of the iounit. Each successive read or write moves the iounit pointer forwards by the number of read or written bytes.

The iounit pointer may be moved by means of the lseek system call, thus giving direct access to the iounit:

```
newpos = lseek( ioud, count, mode );
```

ioud is the iounit descriptor associated with the iounit whose pointer is to be changed. mode may have the values 0, 1, or 2 and the interpretation of count depends on this value: If mode is 0, the iounit pointer is moved to byte number count. If mode is 1, the value count is added to the iounit pointer, thus causing a positioning within the iounit relative to the current position. If mode is 2, the iounit pointer is set to the value of the size of the iounit plus count, thus causing a positioning within the iounit relative to the current end of the iounit. If mode is 1 or 2, count may be negative.

The value returned by lseek is the new value of the iounit pointer.

Note the following useful special cases:

`lseek( ioud, 0, 0 )`	will move the iounit pointer to the beginning of the iounit.
`lseek( ioud, 0, 1 )`	will return the value of the current iounit pointer without changing it.
`lseek( ioud, 0, 2 )`	will move the iounit pointer to the end of the iounit.

lseek is meaningful only when applied to disk files and to disks seen without a file system.

When there is no more need for an iounit it should be closed:

```
close(ioud);
```

This is done automatically when a process terminates (normally or abnormally).

6. Process Management.

Processes may be started by means of several different system calls and subroutines.

6.1. Fork and Exec.

The only way in which a process may start another process in standard Unix systems is by way of the fork and exec system calls.

The system call

```
pid=fork();
```

will cause the calling process to fork. This means that an identical copy of the process will be created. The new process will be a child process of the calling process. The value returned by fork can be used to distinguish in the program between code that is to be executed in the parent process and code that is to be executed in the child process.

In the parent process fork returns the process ID of the child process (or -1 in case of error). In the child process fork returns zero.

The system call

```
execl("/bin/pip","pip","abc","xyz",0);
```

will cause the calling process to metamorphose into the program found in the file /bin/pip. The main subroutine of the new program will be called with the arguments "pip", "abc", and "xyz". The final zero indicates that no more arguments are given.

If the execl call is successful, no return will be made.

Normally, these two system calls are combined in the following manner:

```
pid=fork();

if (pid==0) then  /* this is the child process */
    execl("/bin/pip","pip","abc","xyz",0);

    /* this point is reached only if execl failed */
    fprintf(stderr,"cannot execute /bin/pip\n");
    errout(smoserr);
else_if (pid== -1) then  /* the fork failed */
    fprintf(stderr,"cannot fork\n");
    errout(smoserr);
end_if
:
: /* this is the parent process */
:
```

6.2. Makeproc.

The Supermax Operating System system call makeproc is not found in standard Unix systems. It provides an alternative means of creating new processes. Whereas fork always spawns new processes, makeproc may spawn, gemmate, or produce the new process.

The following example illustrates the use of makeproc:

```
short uv[4];

uv[STDIN]  =open("infile",O_RDONLY);
uv[STDOUT] =STDOUT;
uv[STDERR] =STDERR;
uv[STDLIST]=STDLIST;

pid=makeproc("/bin/pip",0,12,"pip\0abc\0xyz\0",4,uv,0,SPAWN);
```

Makeproc has 8 parameters:

The first parameter is the name of the file containing the program to be executed.

The second parameter is the name to be assigned to the process. In this example 0 has been specified, which is an indication that the operating system should itself assign a name to the process.

The third parameter is the number of characters in the fourth parameter.

The fourth parameter is the arguments passed to the new process. These arguments have been put together to form one string of characters, with null-characters seperating each argument. The format of this string is discussed in greater detail in section 6.4.

The fifth parameter is the number of elements in the sixth parameter.

The sixth parameter is an array of short integers. Each integer specifying an iounit that is to be inherited be the new process. In this example uv[STDIN] is the iounit descriptor returned from the opening of the file "infile". This means that the standard input device of the new process will be "infile". The standard output, error, and list devices of the new process will be identical to those of the calling process.

The seventh parameter is the address of a block of data specifying the priority, user and group IDs, current directory and a lot of other properties of the new process. In this example the parameter is zero, indicating that all these properties are to be inherited from the calling process.

The eighth parameter specifies whether spawning, gemmation, or production of the new process is to take place. The symbolic names SPAWN, GEMMATE, and PRODUCE are defined by including

```
#include <makeproc.h>
```

in the program.

The value returned by makeproc is the process ID of the new process, or -1 in case of error.

6.3. Imspawn.

A special feature of the Supermax Operating System is the ability to spawn a subroutine as a process.

The following example illustrates how this is done:

```
typedef struct begin
          long a, b;
          char *c;
        end             arg;

long stack[1000];
char xx[10];

newproc(parm)
  arg *parm;
begin
  long errors[2];

  my_err(errors);
  :
  :
end

main()
begin
  short pid, uv[4];
  arg   parm;

  parm.a=7;
  parm.b=8;
  parm.c="Hello!";

  uv[STDIN]  =STDIN;
  uv[STDOUT] =STDOUT;
  uv[STDERR] =STDERR;
  uv[STDLIST]=STDLIST;

  pid=imspawn(newproc,0,nice(0),sizeof(parm),&parm,
                                    4,uv,stack,sizeof(stack));
  :
  :
end
```

Calling imspawn causes the subroutine newproc to start execution in parallel to the continued execution of the main process.

Imspawn has 9 parameters:

The first parameter is the address of the subroutine.

The second parameter is the name to be assigned to the process. In this example 0 has been specified, which is an indication that the operating system should itself assign a name to the process.

The third parameter is the priority of the new process. In this case nice(0) has been used. nice(0) is a system call that returns the priority of the calling process. So the new process will have the same priority as the calling process.

The fourth parameter is the number of bytes in the data structure that is to be passed as a parameter to the new process.

The fifth parameter is the address of the data structure that is to be passed as a parameter to the new process. Once the new process is started it will be called with a parameter which is a copy of this data structure.

The sixth parameter is the number of elements in the seventh parameter.

The seventh parameter is an array of short integers. Each integer specifying an iounit that is to be inherited be the new process. In this example the standard input, output, error, and list devices of the new process will be identical to those of the calling process.

The eighth parameter is the address of the data area reserved as stack for the new process.

The ninth parameter is the number of bytes in the data area reserved as stack for the new process.

A few things should be noted about in-memory processes:

As seen in the above example a stack must be supplied by the calling process. If stack overflow occurs, the stack will grow into the global variables declared before the stack.

Any data structure may be passed as parameter to the new process.

The new process shares with the original process all memory segments allocated before the imspawn call. In particular this means that the

old and the new process share the global varibles. If both main and newproc refer to the xx array, they will refer to the same memory locations.

This sharing of global data is, of course, an advantage in many applications, but there are drawbacks. Many standard subroutines, for example printf, use global variables. If both the old and the new process calls printf, inconsistency in the global variables will arise, and printf will not behave properly.

All system calls return their error code in global variables smoserr and errno. The new process, of course, does not want the error codes from its system calls to destroy or be destroyed by error codes from system calls performed by the old process. Therefore newproc calls my_err(errors), which is a system call that tells the operating system that error codes from system calls performed by newproc should be returned in errors[0] and errors[1] rather than in smoserr and errno.

If and when a return is made from newproc, exit(0) will be called. *

Why would you use in-memory processes? Two possible reasons are listed below, sevel other possibilities exist.

1) You have some task to perform while at the same time you want to be able to read a user command typed to a terminal. In this case you can let an in-memory process read the user command while your main process continues with its task.

2) You want to be able to catch an attention exception, but you do not want the occurence of this exception to abort on-going terminal i/o operations. In this case you can let an in-memory process set up an attention exception handler and then wait indefinately. When the exception occurs, the calling of the attention exception handler will abort the in-memory process's indefinate waiting rather than an on-going terminal i/o operation.

6.4. Process Arguments.

Although processes see their arguments as a set of character strings this is not the way in which arguments are really transferred from a starting process to a started process. What is really transferred is a data structure of a given size. In the in-memory process example of section 6.3 we saw how a data structure was passed from the parent process to the in-memory child process. In reality this also takes place when other processes are started.

Let us consider first the starting of a new process by means of a makeproc system call. Two of the parameters to this system call indicate the length and the address of a data structure to be transferred to the new process. In the example in section 6.2 this data structure was the string "pip\0abc\0xyz\0". This data structure is passed to the new process. The C Run-Time Startup module of the new process converts this string to the standard argument format for the main subroutine.

If the C Run-Time Startup module had been different it would have been possible to access the passed data structure without modification, and in this case any data structure might have been passed to the new process, not just character strings.

The standard C Run-Time Startup module expects the passed data structure to have the following format: It must be a character array containing the arguments and the environment to be passed to the main subroutine. The arguments must be seperated by null-characters. The environment strings must be seperated by null-characters. Two null--characters must seperate the arguments from the environment string. If no environment is passed, these two null-characters need not be present.

Suppose, for example, that we want to pass the arguments "alpha", "beta", and "gamma" to a new process. The environment strings "TZ=DNT-1DST" and "PATH=:/bin:/usr/bin" should also be passed. The data structure given to makeproc should be the character array

```
alpha\0beta\0gamma\0\0TZ=DNT-1DST\0PATH=:/bin:/usr/bin\0
```

The three subroutines spto0, arto0, and comb0 may help in the creation of this character array. They are described in chapter 3 of the Supermax Operating System User's Manual.

Now let us consider the execl system call used in the example of section 6.1. Execl is not a genuine system call. Execl calls the arto0 and comb0 subroutines to create the character array to be passed to the new process. After this, execl calls the system call metamorph that performs the actual metamorphosis, passing the contents of the character array to the new program.

6.5. Wating for Child Process Death.

A process may issue a system call that causes it to wait for its child processes to die and informs the process of the death reason of the child.

Two forms of this system call exist. The most informative - but not standard Unix - call is wait2 which is used as follows:

```
int excepno, cc;
char name[8];

pid=wait2(&excepno,&cc,name,TRUE);
```

Wait2 has four parameters. The first three are addresses of locations in which it stores information about the dead process. In the above example, wait2 will store the death reason, that is, if the child died normally or because of an exception. In cc wait2 will store the condition code given by the child if it died normally. In the name array wait2 will store the process name of the dead child.

The fourth parameter is irrelevant if there already exists a dead child whose death information has not yet been requested. If, however, there is no dead child process the fourth parameter should be TRUE if the calling process should wait until one of its children dies. If the fourth parameter is FALSE an error condition will be returned from wait2 if none of the children of the calling process are dead.

Wait2 returns the process ID of the dead process, or -1 in case of error. A typical error is that the calling process has no child processes, neither dead nor living.

A variant of the wait2 system call is the standard Unix wait system call. The reader is referred to chapter 2 of the Supermax Operating System User's Manual for information about this system call.

6.6. Exit.

A process terminates - commits suicide - by calling the system call exit. Exit has one parameter, the so-called condition code, which is passed to a wait2 or wait system call in the parent process. By convention, a condition code of 0 indicates that the process succeeded in performing the task it was requested to do. A condition code different from zero indicates some kind of failure.

If a return is made from the main subroutine without calling exit, the C Run-Time Startup module will perform an exit(0).

6.7. Setting Up a Pipe.

Although pipe handling is really part of i/o management, it is treated here because some knowledge of how to start processes is necessary in order to understand pipes.

A pipe is an anonymous box. It is typically used for communication between a process, A, and a process, B, if B has been started by A or if A and B have been started by the same process.

The basis for pipe handling is the system call pipe:

```
int iouds[2];

pipe(iouds);
```

This system call creates an anonymous box. Two iounit descriptors are created, one for writing to the pipe and one for reading from the box. These iounit descriptors are stored in iouds[0] (reading) and iouds[1] (writing). The idea is now to start another process and let it inherit one of these descriptors, while the other iounit descriptor is used in the parent process.

Consider the following code:

```
int iouds[2], pid;

pipe(iouds);
pid=fork();
if (pid==0) then /* this is the child */
    close(iouds[1]); /* close the write end of the pipe */
    close(STDIN);    /* close iounit descriptor 0 */
    dup(iouds[0]);   /* create a new incarnation of the read
                        end of the pipe and make it STDIN */
    close(iouds[0]); /* close the old incarnation of the read
                        end of the pipe */
    execl(......);
end_if

/* this is the parent */
close(iouds[0]);     /* close the read end of the pipe */
write(iouds[1],......);
```

In the child process, STDIN is closed. The dup system call creates a new iounit descriptor referring to the same iounit as its parameter. The operating system guarantees that new iounit descriptors will have the value of the lowest unused iounit descriptor number. Since we have just closed STDIN (which is iounit descriptor 0), dup will make iounit descriptor 0 refer to the same iounit as iouds[0]. In other words STDIN now is the read end of the pipe. After the execl the new program may read from its standard input device and it will read what the parent process writes in the write statement in the above example.

If makeproc is used instead of fork and execl, things are a bit more easy:

```
int iouds[2], pid;
short uv[4];

pipe(iouds);

uv[STDIN]  =iouds[0];  /* the read end of the pipe should be
                          STDIN */
uv[STDOUT] =STDOUT;
uv[STDERR] =STDERR;
uv[STDLIST]=STDLIST;

pid=makeproc(.....,4,uv,...);

close(iouds[0]);     /* close the read end of the pipe */
write(iouds[1],......);
```

Here the use of the uv array to pass open iounits to the new process makes the complicated calls of close and dup superfluous.

7. Memory Management.

7.1. How to Get More Memory.

This section will deal with three ways in which a process may request more memory.

7.1.1. The sbrk and brk System Calls.

The address of the first byte following the data and bss part of a process's memory is called the 'break value'. A process may change its break value, thus allocating more memory. The system call sbrk increments the break value; the system call brk resets the break value to a previous one.

Consider the following code:

```
struct pip begin
            int a, b;
            char c[100];
          end              *abc, *def;

abc = sbrk(10*sizeof(struct pip));
def = sbrk(25*sizeof(struct pip));
:
:
:
brk(abc);
```

The first call of sbrk allocates 10*sizeof(struct pip) bytes of memory. abc is set to the old break value, that is, the address of the newly allocated memory. After this system call it is possible to refer to the structures abc[0], abc[1], ..., abc[9], because ten times the size of the structure have been allocated. The second call of sbrk allocates memory for another 25 elements of the same type, leaving the address of that memory in def.

The call brk(abc) resets the break value to the contents of abc. This means that both the 10 elements allocated in the first sbrk call and the 25 elements allocated in the second sbrk call are deallocated.

7.1.2. The malloc and free Subroutines.

malloc and free perform much the same task as sbrk and brk. In fact, malloc invokes sbrk. But whereas brk may deallocate memory allocated by several sbrk calls, free deallocates only what has been allcated by one malloc call.

Consider the following code:

```
struct pip begin
            int a, b;
            char c[100];
          end              *abc, *def;

abc = malloc(10*sizeof(struct pip));
def = malloc(25*sizeof(struct pip));
:
:
:
free(abc);
```

The first call of malloc allocates 10*sizeof(struct pip) bytes of memory. abc is set to the address of the newly allocated memory. After this call it is possible to refer to the structures abc[0], abc[1], ..., abc[9], because ten times the size of the structure have been allocated. The second call of malloc allocates memory for another 25 elements of the same type, leaving the address of that memory in def.

The call free(abc) deallocates the 10 elements allocated in the first malloc call, but not the 25 elements allocated in the second malloc call.

7.1.3. The par_cre and par_det System Calls.

The system calls par_cre and par_det are not found in standard Unix systems. par_cre creates a memory partition, and maps it to a specified logical address segment. par_det detaches and deletes the partition.

Consider the following code:

```
struct pip begin
            int a, b;
            char c[100];
          end              *abc=0x800000, *def=0x900000;

par_cre(0,10*sizeof(struct pip),abc);
par_cre(0,25*sizeof(struct pip),def);
:
:
:
par_det(abc);
```

The first call of par_cre creates a memory partition 10*sizeof(struct pip) bytes long. The MMU (Memory Management Unit) is instructed to map this partition to the logical address abc. abc has been intialized to 0x800000 (8 megabyte). Similarly, the second call of par_cre creates a memory partition 25*sizeof(struct pip) bytes long and maps it to logical address segment 9. par_cre has, in this case, three parameters:

- A zero indicating the creation of an anonymous partition (see below).
- The size of the partition to be created (at most 1 megabyte).
- The logical address to which the partition should be mapped (must be a multiple of 1 megabyte).

The system call par_det detaches the partition from the program and deletes it.

When a process dies, it is automatically detached from all its partitions, and the partitions are deleted, except as noted below.

7.2. Named Partitions.

A process may create a partition and assign it a name. In this case other processes running on the same MCU may attach to that partition and share the memory with other processes.

A process may create a partition called 'part' in the following way:

```
struct pip begin
            int a, b;
            char c[100];
          end              *abc=0x800000;

par_cre("part",10*sizeof(struct pip),abc,0644,FALSE);
```

The par_cre system call creates a named partition (the first parameter gives its name) with the size of 10 elements of type struct pip. The partition is mapped to logical address 8 megabyte. Two additional parameters are here given to par_cre. The first of these is a set of protection bits, in this case 644 (octal) giving the owner (creator) of the partition read and write access right, and all others read access right. The protection scheme is the same as with iounits. The final parameter specifies if the partition should be automatically deleted when no process uses it. The FALSE given in this example specifies that the partition should not be deleted when not used. An explicit delete request (the par_del system call) may delete the partition.

Let us now suppose that the process stores some interesting information in the partition. Let us further assume that another process running on the same MCU wants to access this information. The other process may 'attach' to the partition and access the information in the following way:

```
struct pip begin
            int a, b;
            char c[100];
          end              *abc=0x800000;

par_att("part",abc,READ);
```

The par_att system call attaches the calling process to the partition called 'part'. The partition is mapped to logical address 0x800000 and READ access is requested.

After this system call has been performed, the process may access the contens of the partition as abc[0], abc[1], ..., abc[9].

If the data in the partition contains pointers to other locations in

the partition it is important that the partition be mapped to the same logical address segment that was used when the partition was created.

8. Exception Handling.

An exception is an abnormal event in the execution of a process. In Unix it is customary to use the term 'signal'; we feel that this term is a misnomer, and prefer the word 'exception'. Exceptions may arise because of things done inside the process, such as a reference to an uanllocated memory cell, or because of things happening outside the process, such as the pressing of the attention key on a terminal.

Exceptions are discussed in section 3.2.6 of the Supermax System Operation Guide. We will therefore here confine ourselves to an example. We will write an exception handler that catches the attention exception:

```
#include <std.h>
#include <signal.h>

atthand()
begin
  write(STDOUT,"Attention\n",10);
end

main()
begin
  signal(XATTENT,atthand);
  :
  :
end
```

The system call signal informs the operating system that instead of performing the standard handling of an exception (in this case XATTENT, the attention exception), a subroutine should be called (in this case atthand).

If, during the further execution of the process, the attention key is pressed on the standard input terminal, atthand will be called and will write its text. Control will thereafter return to the main program. If we want to be able to catch several subsequent attentions, we must call signal inside atthand, for exception handling normally reverts to default handling after the occurrence of the exception.

What happens in the main program when the exception handler is called? Two possibilities exist:

1) If the main program is executing program code, atthand is called as if it were a simple subroutine, and the main program can detect nothing extraordinary.

2) If the main program is executing a system call that suspends the execution of the process (for example, reading from a terminal), the system call is aborted, atthand is called as a subroutine, and control returns to the main program as if the system call returned the Supermax Operating System error condition ESIGNAL (Unix error code EINTR).

As another example, let us consider a process that waits for one of two events to occur: The passing of 60 seconds or the pressing of the attention key. The process uses the system call suspend, that stops process execution for a given time:

```
#include <std.h>
#include <signal.h>

atthand()
begin
    /* don't do anything - we just want suspend to terminate
       abnormally */
end

main()
begin
    int i;

    signal(XATTENT,atthand);
    i=suspend(-1,60000); /* suspend for 60000 milliseconds */
    if (i==-1) then
      printf("Terminated by exception\n");
    otherwise
      printf("Terminated by time expiration\n");
    end_if
end
```

Two common exceptions are the bus error and address error. A subroutine exists that provide a standard handling of these two exceptions, its name is abehand (short for Address and Bus Error Handler). This routine is called in the standard C Run-Time Startup module. If an address or bus error occurs, information will be output to the stan-

dard error device, giving information about what address was accessed and where the access was made.

A bus error is caused by access to an unallocated memory location. A typical bus error is stack overflow. In this case, however, abehand cannot catch the bus error. The reason is that the calling of an exception handler requires that information be stored on the stack, but as the error was caused because of insufficient room on the stack, there is nothing to do about it. The process has no option but to die.

9. Interface to vox.

In many cases the vox operator communications program uses the environment strings as a means for transferring parameters to a program, because the environment gives the possibility of working on named parameters.

The follwing vox command will execute the program alpha with the environment string "unit=beta":

```
vox>alpha  unit=beta
```

Alternatively, the environment string may be given globally:

```
vox>unit=beta
vox>alpha
vox>alpha2
```

These three commands set up a global environment string "unit=beta" and then executes first alpha and then alpha2 with this environment string.

Two subroutines exist that can be used to fetch an environment value. They are getenv and env_prompt.

9.1. getenv.

getenv is called with a parameter that is the address of a null-terminated environment string name. getenv returns the address of the null-terminated value of this environment.

If a process is executed with the environment string "unit=beta" a call of getenv("unit") will return the address of "beta".

If no environment string exists with the given name, getenv returns 0.

9.2. env prompt.

env_prompt is, for example, called as follows:

```
env_prompt("unit","Which iounit? ","myfile",buf,FALSE);
```

This subroutine checks the occurence of an environment string, and if it is present its value is returned. Otherwise the terminal operator is prompted for its value.

env_prompt has five parameters:

1) The environment name, in this case "unit".

2) The prompt text.

3) A default value.

4) The address of a character buffer where the result is to be stored.

5) A boolean value that is TRUE if prompting is to take place even if the environment does exist.

In the above example the contents of buf will be set to the value of the environment "unit". If this environment does not exist, the text "Which iounit? " is output, and the operator is allowed to edit the text "myfile", leaving the result in buf.

env_prompt returns the number of characters in the environment value.

10. How to Compile.

The C compiler is called cc. It is described in detail in the Supermax Operating System User's Manual part 1. Here we will give a couple of examples of its use.

The source code for the C program is prepared by a text editor (for example, the Supermax editor edit). The source code must be stored in one or more files whose last two charcters are '.c'. Let us assume that we have a C program that consists of subroutines located in two files, abc.c and def.c. The two files are compiled by giving the following command to vox or shell:

```
cc -o abc  abc.c def.c
```

This command will compile abc.c and def.c, leaving the object code in files called abc.o and def.o. Further, these two object files will be linked together with the standard C Run-Time Startup module /lib/crt0.o and the standard C library /lib/libc.a. The final executable module is stored in a file called abc (this filename is given from the argument in the command that follows the -o option).

Now, suppose an error was detected in def.c. This file is modified and we want to compile again. There is, however, no need to recompile abc.c. The command

```
cc -o abc  abc.o def.c
```

will recognize abc.o as the name of an object file. So only def.c will be compiled, and it will be linked together with abc.o.

Suppose we to perform only the compilation. No linkage is to be done. By replacing the '-o abc' with '-c' only compilation takes place:

```
cc -c  abc.c def.c
```

This command will compile abc.c and def.c, leaving the object code in abc.o and def.o.

Assembly language modules may be included in the compilation. Assembly language source files should have '.s' as their last two characters. The command

```
cc -o abc  abc.c xyz.s def.c
```

will recognize abc.c and def.c as names of C source programs and compile these. The filename xyz.s will be recognized as the name of an assembly language source file, and it will be assembled. The object modules will be left in files abc.o, xyz.o, and def.o, whereupon they will be linked together.

If the C program contains #include statements, rules apply as indicated in the following examples:

1) If, for example, the command
```
cc -c abc.c
```
is given, and abc.c contains the statement
```
#include "hello.h"
```
the file hello.h will be taken from the current directory.

2) If the command
```
cc -c /usr/bilbo/abc.c
```
is given, and /usr/bilbo/abc.c contains the statement
```
#include "hello.h"
```
the file hello.h will be taken from the directory /usr/bilbo.

3) If a file contains the statement
```
#include <hello.h>
```
the file hello.h will be taken from the directory /usr/include.

10.1. Stack Size.

Local variables, parameters, and subroutine return addresses are stored on the process stack. There is currently no way for the system to extend the stack if required during the execution of a process.

Normally a load module (executable program) is created with a stack size of 4 Kbytes. Larger stacks may be created by including stack defining modules in the linking:

```
cc -o abc   abc.c def.c /lib/ssize16k.o
```

This command will create abc with a stack size of 16 Kbytes. The following stack size defining modules are available: /lib/ssize16k.o, /lib/ssize32k.o, /lib/ssize48k.o, and /lib/ssize64k.o.

The program setstack may be used to alter the stack size of a program. Giving the following vox command

```
setstack unit=abc stack=0x8000
```

or the shell command

```
env unit=abc stack=0x8000 setstack
```

will set the stack size of the program abc to 0x8000 bytes. Giving the vox or shell command

```
setstack
```

will cause setstack to prompt for a program name and allow the operator to edit the stack size.

Note: If the execl, execle, execv, or execve routines are used, at least 6 Kbytes of stack should be reserved.