



Easy C++

Tutorial and Reference Manual

ProAction

3QD
DEVELOPMENTS LTD

APDL

Easy C++

Copyright © ProAction & 3QD Developments Ltd 2026
All rights reserved.

No part of this product may be reproduced in whole or part by any means without written permission of the publisher. Unauthorised hiring, renting, lending, public performance or broadcasting of this product or its parts is prohibited.

While every care is taken, the publisher cannot be held responsible for any errors in this product, or for loss of data or consequential effects from the use of this package.

Easy C++ was written by David Allison.

SIXTH EDITION 2025 (Tutorial and Reference Manual)

Software and Documentation supplied by

APDL

a part of

3QD Developments Ltd

Single User version downloaded from the free of charge software section of
www.riscos.com

Please note that any free of charge downloads from RISCOS.com are supplied without any technical support package as per the licence agreement.

This document last edited on 29/09/26



If you have found this application and this manual useful please consider making a donation to help cover the costs of cataloguing, documenting, updating and hosting the APDL applications. Scanning the QR code will take you to our donations page.

Contents

Chapter 1 - C Tutorial	5
The basic elements	5
Iteration - the for loop	6
Optional parameters	7
Incrementing and decrementing	7
Assignment operators	7
The root of the problem	7
Character handling	9
Pointers and arrays	10
Files	12
Introducing structures	14
Formatted input and output	16
Formatted input - scanf()	18
Redirecting input and output	18
Command line arguments	19
Some useful functions	20
Bit variables	21
Using SWIs	22
 Chapter 2 - ANSI Functions	 25
Character handling <ctype.h>	25
Character testing functions	25
Mathematics <math.h>	27
Trigonometric functions	27
Hyperbolic functions	28
Exponential and logarithmic functions	28
Power functions	29
Miscellaneous functions	29
Input/output <stdio.h>	30
Operations on files	30
File access functions	31
Character input/output functions	37
Direct input/output functions	38
File positioning functions	40
Error handling functions	41
General utilities <stdlib.h>	42
String conversion functions	43
Pseudo-random sequence generation	46
Memory management functions	46
Communication with the environment	47
Searching and sorting utilities	48
Multibyte character functions	49
Multibyte string functions	50
String handling <string.h>	50
Copying functions	50
Concatenation functions	51
Comparison functions	52
Search functions	53
Miscellaneous string functions	54
 Chapter 3 - RISC OS functions	 55

Chapter 1 - C Tutorial

This introduction to C programming was originally written by Dec McSweeney and published in both BEEBUG and RISC User magazines. We've included it here as, despite our emphasis on C++, it's a good introduction to C programming in general.

The language C has been around for some time and has proved very popular with programmers, particularly for writing operating systems and other low-level applications. It can be used equally well in place of Basic and ARM assembler for almost any application. This chapter will introduce programmers familiar with Basic to the elements of the language, and give them a taste of its capabilities. All the program listings in this chapter may be found in the \$.Programs.Tutorial.c directory on the CD.

The basic elements

One fundamental difference exists between Basic and C. Programs consist of statements. Enhanced versions such as BBC Basic allow statements to be grouped in subroutines (PROC and FN), but these are optional. In C programs always consist of *functions*, referenced by name. Functions are self-contained sets of program instructions which usually take one or more parameters and generally return a value; much like the Basic functions: SQR(A) which returns the square root of A, LEFT\$(NAME\$,4) which returns the first 4 characters of NAME\$, and FNmyfunc(p1,p2,p3) which uses parameters p1, p2 and p3.

Let's take a look at a simple C program called *HelloW*:

```
/* HelloW - print hello, world */
#include <stdio.h>
int main(void)
{printf("hello , world\n");}
```

There are several points to note here:

- C does not use line numbers.
- Lower case characters are used for C keywords and (conventionally) for user-defined words such as variable names and function names.
- A number of non-alphabetic characters such as curly braces {}, hash sign (#) and asterisk (*) have special meanings in C.

Taking the program line by line, we find the following:

Line 1 in the source code above is enclosed in comment delimiters. Anything which appears between the */** and the **/* is ignored by the compiler. Good programmers use this to explain how their code works.

Line 2 starts with the # character. This makes it a *preprocessor directive*, and is the first phase of the compilation. It tells the preprocessor to find a header file called *stdio.h* and treat it as part of this program. This file contains information relating to the standard input and output routines, and hence its name *stdio*. On finding this directive, the C compiler looks for a file *stdio* in directory *h*. This naming convention is the opposite way round to normal Acorn conventions, but allows portability between C compilers running under different operating systems.

Line 3 is the name of the function. All C programs must have a function called **main()**, and execution of the program starts here. The words **int** and **void** will be explained in due course.

Line 4 is an opening brace { }, marking the actual start of the main function definition. Braces are used in C to allow a group of items, such as statements, variables or data, to be treated as a single item.

Line 5 is the only statement in our program. Note that **printf()** is NOT a C command, but a call to a library function called **printf()**, which prints the string enclosed in quotes. The character pair **\n** represents 'new line', and note how this is also included within the quotes. Note also how the statement ends with a semi-colon.

Line 6 marks the end of the function, and in this case the end of the program.

This source code is used as input to the C compiler. The preprocessor copies in any #included files, and the expanded file is passed to the compiler proper which (compilation errors permitting) produces an object file which must be linked to produce a program. To run the compiled program, double-click on *HelloW* to display the message hello, world.

HelloW is a very simple program; it uses no variables and there is no flow control. The expanded program *HelloW2* below, incorporates both of these features.

```
/* HelloW2 - hello, world expanded version */
#include <stdio.h>
int main(void)
{ x ;
  for (x = 0; x < 10; x = x + 1)
    printf("hello, world %d\n", x);
}
```

The first line after the opening brace, `int x;` defines a variable. In Basic, variables other than DIM declarations are called into being when they have a value assigned to them. All variables in C must be defined before they are used so the compiler can reserve space for them in the object program. The semi-colon terminates variable definitions as well as statements. As in Basic variable types include *float* (floating point), *int* (integer or whole number), *char* (a single character). Strings are dealt with as 'arrays of type *char*'. Functions must also have a type according to the nature of the data they return, just as `LEFT$( )` is a Basic string function, `instr()` returns an *int*, and `sqr()` a *float*.

The next line features a construction familiar to any Basic programmer, - the FOR/NEXT loop.

The `printf()` function now has a more complicated structure than hitherto with two arguments: the string *"hello, world %d\n"* known as the conversion string, and the variable *x*. The string contains *literal* text which is printed as it appears, plus *formatting information* to control the conversion (i.e. the format) of *x*. The `%d` tells the function to output its next argument as a decimal integer at this point on the printed line.

`printf()` is complicated and will be discussed in more detail as we proceed.

Iteration - the for loop

In C, `for()` works in very much the same way as the Basic equivalent, but is much more flexible. The general format is:

```
for ( expr1; expr2; expr3 ) statement
```

Note that no NEXT is required, and that statement could be a single statement which would then be followed by a semicolon, or a group of statements enclosed in braces. The brackets contain three expressions, separated by semi-colons: *expr1* sets up the starting condition(s), *expr2* is evaluated on each iteration, and *expr3* is the 're-initialiser' which is performed after each iteration. The loop terminates when the second expression evaluates to FALSE (i.e. the loop is performed while the second expression is TRUE). The Basic equivalent of the above is:

```
FOR N%=0 TO 9 STEP 1
PRINT "hello, world "N%
NEXT
```

The loop is performed while *x* (or *N%*) is less than 10, hence TO 9 is its Basic equivalent. *x* and *N%* will both be 10 when the programs exit from their respective loops. However, unlike a FOR/NEXT loop in Basic, a **for()** loop in C will never be executed (even the first time) if the terminating condition *expr2* is FALSE to start with. In C, expressions can be much more complicated than in Basic, for example:

```
for (x = 0, y = 1023; y >= 0 && point(x,y) == 0; x++, y--)
```

Here, there are two parts to the first and third expressions, and the central expression is a compound condition involving a call to a function. Note the lack of a terminating semi-colon; **for()** is a preamble to a statement or group of statements, rather than a statement in its own right. Let's look at this line in detail:

x = 0, y = 1023 assigns starting values to *x* and *y*. The comma separator may be used as both assignments form part of *expr1*.

y >= 0 && point(x,y) == 0 can be expressed as '*while y is greater than or equal to 0 and the function point(x,y) returns the value 0*'. The double ampersand `&&` is the logical AND operator. In C, `==` serves as the equality test (IS equal to), as opposed to the single `=` (BECOMES equal to), for assigning values. The function call **point(x,y)** is evaluated after the test on *y*, and only if it is TRUE.

Lastly, *x++*, *y--* means *add 1 to x and subtract 1 from y* (see below)

Optional parameters

While just the STEP clause is optional in Basic, everything is optional in **for()**. If you examine C programs you will often see 'empty' **for()** loops where all the work is done in the 'reinitialise' expression, for example:

```
for ( m = 1; n > 0; m *= n-- )
    ;
```

For clarity, the body of the loop, namely the semicolon which must be present, is detached from the closing parenthesis. Similarly, each of the three expressions of **for()** in parentheses is optional:

```
for( ; m < 10; ) /* no initialise or reinitialise */
for( ; ; )      /* start of an infinite loop */
```

Incrementing and decrementing

Incrementing and decrementing variables by one occurs so often it has its own operators; ++ and --. They can go before the variable (*prefix*) or after it (*postfix*). Prefix increment means 'add 1 to the current value, then use it'; postfix means 'use the current value then increment it'. For example:

```
x = 7;          /* x is initialised */
y = 4 * ++x;    /* x is incremented first, so y = 32 */
z = y--;        /* z becomes 32, y becomes 31 */
```

Assignment operators

The notation *= used above may be familiar to Basic V programmers who can use the related += and -=. In general, the notation:

A <op>= B

expands to:

A = A <op> B

where <op> can be almost any binary (two-operand) operation.

The root of the problem

This section describes a C program to calculate the square root of a number. We present the program here in its entirety before discussing its individual features. In particular, the example illustrates the use of some of C's flow control commands, and the use of variables.

```
/* Root - calculate the square root of a number */
#include <stdio.h>
#define ACCURACY 1.0E-6
double sqrt(float);
int main(void)
{ n ;
  printf("give me a number; ");
  scanf("%f", &n);
  while(n != 0)
  { entered %f", n);
    printf(" square = %f", n*n);
    if(n > 0)
      printf(" root = %f \n", sqrt(n));
    else
      printf("Negative - no real root\n");
    printf("give me a number; ");
    scanf("%f", &n);
  }
}
double sqrt(float num)
{ ans , acc;
  acc=ans=num/2.0;
  while(ans > 0 && acc > ACCURACY)
  { ans=(ans+(num/ans))/2.0 ;
    acc=num - (ans*ans);
    if(acc < 0)
      acc = -acc;
  }
}
```

```
    return ans;  
}
```

The first line is comment, the second is the `#include` preprocessor command we saw earlier, but the third line is new. It is another preprocessor command, `#define` which causes all occurrences of the first word (throughout the source program) to be expanded to the value of the rest of the line. This is useful for large programs using a constant value in many places which is subject to change, such as VAT rate, currency symbol, and so on. If the value changes, you only need to change it in one place in the program. Here we use it to define the accuracy required in our calculation. The notation might be familiar to you; it means '*1.0 times 10 to the -6th power*', or 0.000001.

The fourth line is a function prototype; it tells the compiler that the function `sqt()` exists in the program which returns a value of type `double`, and takes one parameter of type `float`. `double` stands for 'double precision floating point'. Specifying a prototype allows the compiler to check that all calls to the function have the right number and type of parameters. For the purpose of this example the function has been called `sqt()` to avoid a name clash with the function `sqrt()`, which is supplied as standard with all implementations of C.

As a function itself, `main()` has a return type of `int` (normally for returning an integer error code). It has no parameters, so you must put `void` where the arguments would normally go.

After the opening brace which marks the start of the program proper, the variable `n` of type `float` is defined. This is followed by a simple `printf()` to ask the user for input, and a call to `scanf()`, which is the C equivalent of Basic's INPUT. `scanf()` takes as its arguments a *conversion string* (`%f` in the above example) and the address(es) of where the inputted data is to be placed; `&n` means 'the address of `n`'. We'll be looking at addresses and pointers in detail later. Strictly speaking, `scanf()` fetches the data from the standard input stream, but this will normally be the keyboard.

Then a new iterative construct called `while()`, which is similar to WHILE in Basic. The expression in brackets after is evaluated and if it is TRUE (or non-zero) the group of statements is executed. This process repeats until the expression evaluates as FALSE (or zero). The line `while(n != 0)` means *while n is NOT equal to 0*, so the loop is only executed if a nonzero value is entered.

On the next two lines, the value entered and its square are printed. Then the value of `n` is tested using the `if()` statement. It works much like the Basic equivalent; if the expression evaluates as TRUE (non-zero), the statement (or group of statements) is executed, otherwise the statement(s) following the optional `else` are executed. Processing continues from the statement after the `else` statement.

In this case, if `n` is negative, a message is printed, otherwise we print the value returned by the function `sqt()` with `n` as its parameter (`%f` is the floating point equivalent of `%d`, by the way). To evaluate this, C passes a copy of the value held in `n` (call by value, an important property of C), executes the statements in `sqt()` and comes back with the calculated answer. The loop ends with another prompt for input, and processing continues until the user enters zero. The program terminates at the closing brace of `main()`.

`sqt()` is a separate function, and starts as all functions must, with its return type, its name, and any parameter declarations. Naturally, the definition of a function must agree with its prototype at the start of the program. In this case the return type is `double`, its name is `sqt()` and its one parameter `num` is of type `float`. In our example `num` will contain a copy of `n`. At the start of `sqt()` some more variables are defined and initialised. These variables are local to the function, and are called into existence for the duration of each call. The variable `ans`, for example, doesn't exist as far as `main()` is concerned. Nor does it retain its value between calls. Note the technique for assigning a value to more than one variable.

The first statement of `sqt()` is another while loop. In this case the loop will repeat as long as `ans` is greater than zero AND our accuracy requirements are not met. Note the `&&` for the logical AND in C as opposed to the single `&` for bitwise AND. The test for zero is to avoid a divide by zero within the loop. Unlike Basic C can't trap this for you tidily. At the end of the loop `acc` is made positive in an operation equivalent to Basic's ABS function.

The `sqt()` function uses an iterative method for obtaining a root, and this consists of refining a guess at the square root of `num` until the required accuracy is attained. Try out the arithmetic with a calculator if you like; the arithmetic operators and brackets within the loop mean the same as in Basic. When either condition is satisfied, the function sends the answer back with `return(ans)`. The value so obtained is passed to `printf()` for display upon the screen.

If you compile and run the program you will find a limitation C shares with Basic; floating point numbers are held to an accuracy of about nine digits, which causes inaccuracies in the least significant digits of large

numbers. This may be avoided by checking that the user's input is both positive and less than a certain threshold, by changing the if statement in **main()** to read:

```
if(n > 0 && n <= MAXNUM)
    printf(" root = %f \n", sqrt(n));
else
    printf("Negative or too big - can't do\n");
```

Where MAXNUM could be *#defined* like ACCURACY. You may be wondering why *#define* is used. Using 'magic numbers' in the source is bad for portability, as MAXNUM depends on the implementation and might be different on another machine. Hence we set these things out at the top of our program, or even in a header file. On another machine or with another compiler we could either amend the source or use the 'local' version of the header file.

We'll return to **sqrt()** in the future, and refine it to reduce its limitations.

Character handling

In Basic, there is a variable type string which holds 0 to 255 characters, preceded by a byte to hold the variable's length. In C, as we mentioned earlier, strings are dealt with as arrays of characters. The end of the string is marked by the character whose ASCII code is 0 (null). This simplifies things in some ways, while complicating them in others. In C, a variable of type char is defined as being at least big enough to hold any member of the machine's character set. The definition carefully avoids tying a char to a byte, and means you shouldn't make such an assumption in your programs. Since it is possible to do arithmetic on char's, it means we can do away with the basic functions ASC and CHR\$ at a stroke. Look at the following Basic fragment, and compare it with the C fragment which follows it:

```
REM Basic put "B" in new$
num1%=ASC("A"): num2%=num1%+1
new$=STR$(num2%)
/* C put "B" in new */
int num1, num2;
char new;
num1='A';
num2=++num1;
new=num2;
```

In the C program, 'A' is a character constant, defined as an int with the numeric value of the quoted character in the machine's character set. As a matter of fact, the use of int above is superfluous, given that arithmetic can be performed on char's. A C programmer might write:

```
/* put "B" in new - version 2 */
char new='A';
new++;
```

This sort of thing won't be completely new to you if you have used indirection in Basic.

It could be said that C's characters operate in much the same way as Basic's byte indirection while strings, being arrays of characters, are like Basic's \$ indirection. This is where an area is set aside in the program to hold (in our case) a string:

```
DIM mess 50      : REM reserve 50 bytes for string
$mess="Hello mum" : REM insert data
PRINT $mess      : REM behaves like a string variable
mess?5=13        : REM insert return at 6th char
PRINT $mess      : REM result would be "Hello"
```

In C, the code looks quite similar:

```
char str[50];      /* reserve 50 chars */
strcpy(str, "Hello mum"); /* copy message in */
printf("%s\n", str); /* print our message */
str[5]='\0';       /* insert null character */
printf("%s\n", str); /* print result, "Hello" */
```

The first line of the C example defines an array called str with 50 characters. Note that C uses square brackets for array subscripts. The second uses a library function strcpy to transfer the string into our array. Just as Basic

adds a return character, C adds a terminating null character to the source string Hello mum which is copied into str. The third line prints it; %s is the conversion code for strings, and \n adds the newline. Line 4 changes the contents of the sixth character (C arrays start at element 0). \0 is a special character constant, representing zero. Finally we print the altered string.

The nth element of an array is accessed as **str[n-1]**. However, there is another useful notation which exploits one of C's most powerful tools. C is a high-level language which allows complicated operations to be programmed in a small amount of human-readable code. However, it shares some characteristics with assembly languages, one of which is the ability to use the address of something, as well as its value. The address of an array is held in a special variable called a *pointer*. Pointers, like ordinary variables, must have types; in the example above the type is *char*. In fact, str itself is a pointer which holds the address of **str[0]**. The address of **str[1]** may be expressed as **str+1**, and so on.

The arguments passed to **printf()** and **strcpy()** are actually pointers to *char*. You can declare free-standing pointers like this:

```
int n;                /* ordinary variable */
int *ip;              /* pointer to int */
Pointers can hold the address of any variable of the same type:
ip = &n;              /* ip points to n,
                      & means 'the address of' */
```

You can then use the variable name and the pointer interchangeably:

```
*ip = 5;              /* the same as 'n = 5;' '*' means the
                      "value at the location pointed to by" */
nsquared = *ip * *ip; /* multiply 5 times 5 */
printf("%d", *ip);    /* print 5 */
```

Using the new notation with an array is quite logical:

```
str[5] = '\0';
is equivalent to:
*(str + 5) = '\0';
```

The brackets are needed to force the expression on the left of the equals sign to evaluate correctly; ***str + 5** is 'the contents of str[0], with 5 added' or in the above example M (the ASCII value of the H of Hello plus 5). You can do a limited amount of arithmetic with pointers. Remembering that a string is a zero-terminated character array, the following fragment will print the length of the string in str:

```
char str[512];
char *temp;                /* pointer to char */
scanf("%s", str);          /* str is the address of */
for(temp = str; *temp; temp++) /* the start of the array */
    ;
printf("this string is %d chars long\n", temp - str);
```

The for loop is empty because all its work is done in the re-initialiser. The loop will exit when the central expression ***temp** evaluates to zero i.e. when the character pointed to by temp is NULL, which is at the end of the string. The expression **temp++** simply increments temp so it points to the next element in the array.

The only legal arithmetic operation involving two pointers is subtraction and the result is only defined when they point to the same array. The result of such a subtraction is the number of elements lying between the two pointers, plus 1. So if **p1** and **p2** point to adjacent elements in an array, the expression **p2 - p1** evaluates to 1.

Pointers and arrays

One thing to strike the newcomer to C about arrays is the use of square brackets for the subscripts, for example when declaring an array:

```
int two_dim [12] [31];
```

Unlike Basic, Fortran, Cobol and so on, each subscript has its own pair of brackets. This allows us to think of the array as (in this case) twelve separate arrays, each of 31 integers. The following code illustrates one use of this concept:

```
int monthtotal(int daily_figs[])
```

```
{n , total;

    for(n = 0, total = 0; n < 31; n++)
        total += daily_figs[n];
    return total;
}
```

monthtotal() simply returns the sum of all the integers in a 31-element array; elements 0 to 30. There is no element 31! The array size is not required in the function definition as it is defined elsewhere. The function **monthtotal()** works quite happily in all the following cases:

```
int jan_figures[31];
int year_1990 [12] [31];
int decade_80_89[10] [12] [31];

jan_total=monthtotal(jan_figures);

for(n = 0; n < 12; n++)
    printf("Month %d total = %d\n", n,
monthtotal(year_1990[n]));

for(y = 0; y < 10; y++)
    {%d\n" , 1980 + y);
    for(n=0; n<12; n++)
        printf("Month %d total = %d\n", n,
            monthtotal(decade_80_89[y][n]));
    }
```

Notice how we send a 31-element array to the function by omitting the last set of brackets. As we suggested last time, a (one-dimensional) array name on its own is a pointer to the first element in the array. The designers of C are very fond of recursive definitions, so an n-dimensional array is the same as an (n-1)-dimensional array of one-dimensional arrays, and therefore **year_1990[n]** is a pointer to an array, as is **decade_80_89[y][n]**.

In passing, you may have spotted the arithmetic expression **1980 + y** in one **printf()** statement. As with Basic's PRINT, expression arguments are evaluated before the call to **printf()** actually takes place. In general, wherever C expects an object, it will be quite happy with an expression which has a result of that type.

The function **monthtotal** above could be made more useful if it didn't always have to add 31 elements. A more generally useful version would add as many elements as we liked, any time we called it, and I suspect it would use pointer notation. Before we start, however, some design considerations.

How do we specify which elements to add? The array could terminate with an agreed end-of-array value, just as strings always end in the NULL character. This would be awkward, as almost any value might reasonably appear in an integer array. One could use the implementation-defined maximum integer value, **INT_MAX** which is #defined in **limits.h**, but what if you want to add a part total, for example a weekly total within the monthly figures above? You would have to insert **INT_MAX** in your array, call the function and then restore the original value. A second approach is to specify how many elements to add, passing this as an argument to the function. This imposes fewer restrictions on the input, and is probably the best solution for our purposes. Observe:

```
/* arraytotal - accumulate n elements from an array of ints */
int arraytotal(int *a, int n)
{total=0 ;
    while( n-- > 0)
        total += *(a++);

    return total;
}
```

The function definition now contains two arguments, of which the first is the more interesting. The argument is of type '*pointer to int*', and will contain a copy of the address of the first element in the array. The second contains the number of elements to be accumulated. Since these are copies internal to the function, we can alter them safely in the while loop. As each element is added to total, **a** is incremented. This causes **a** to point to the next element in the array (not the next byte). **n** is decremented until it reaches 0 at which time the required total has been computed, so the function returns.

Calling the function is done in exactly the same fashion as before, except that we must specify the number of elements to add:

```
jan_total=arraytotal(jan_figures, 31);

for(n=0; n<12; n++)
    printf("Month %d total = %d\n", n,
        arraytotal(year_1990[n], 31));
week1_figs = arraytotal(jan_figures, 7);
```

Of course, we need not start at the beginning of an array:

```
week2_figs = arraytotal(jan_figures + 7, 7);
And we can treat a two-dimensional array as a one-dimensional array:
year_total = arraytotal(year_1990, 31*12);
```

Although this seems like bad programming as it relies on the 12 31-element arrays being contiguous in memory.

Files

Just like any other programming language, C has a set of functions allowing the user to open, read, write and close files.

A file is viewed as a stream of characters, hence my references to the standard input stream and so earlier in the chapter. All input and output is actually done through streams. When a C program is run, three streams are immediately opened - the standard input (stdin), standard output (stdout) and standard error (stderr) streams. stdin is normally assigned to the keyboard, stdout and stderr to the screen, but can be redirected, for example, to take input from a file. More of this later.

The Basic programmer will feel almost at home with the `fopen()` command:

```
fp = fopen(filename, mode);
```

fopen() is a function which opens a *stream* and assigns a *file* to it. **filename** is a character pointer to the string containing the filename, and **mode** is a character pointer to a string denoting the required access, such as **r** for read, **w** for write (create a file or open and clear an existing file), and **a** for append (create or open for writing at EOF; end-of-file). To these may be added **+** denoting read/write access, and/or **b** for binary files (as opposed to text files which, in some environments, are treated differently). For example:

```
infile = fopen("$scores", "r+");
/* open file '$scores' for update */
fopen() returns a pointer of type FILE:
FILE *infile;
```

which is typically a collection of data about the file including its name, status, location and so on. **fopen()** sets this data up and **infile** points to it, in much the same way as Basic's file handle. All other file-handling functions use the file pointer to access the file. If the file cannot be opened, the file pointer is set to NULL (zero). By definition a NULL pointer does not point to anything valid. Further information about the error is put in a variable **errno** declared in the header file *errno.h*.

Once the file has been opened, there are a number of ways of getting data into and out of it, including:

<code>fgetc()</code> , <code>fscanf()</code> , <code>fread()</code> , <code>fgets()</code>	various input functions
<code>fputc()</code> , <code>fprintf()</code> , <code>fwrite()</code> , <code>fputs()</code>	various output functions
<code>ftell()</code> , <code>fseek()</code> , <code>rewind()</code>	manipulate file pointer

Briefly, **fprintf()** and **fscanf()** work exactly like **printf()** and **scanf()**, except that they have an extra argument, a file pointer; **fread()** and **fwrite()** handle multiple reads and writes into/out of arrays; **fgets()** and **fputs()** read and write strings. For now we'll concentrate on **fgetc()** and **fputc()**.

fgetc() and **fputc()** are very much like **BGET#** and **BPUT#**, which you'll no doubt have come across in your Basic programming. We're going to use them to read and write strings in Basic file format. This will be useful if, like me, you have huge amounts of data generated by Basic programs and your future development strategy involves C.

Back to basics first. In Basic, as we have already discussed, a string consists of zero to 255 bytes preceded by two other bytes. The Basic file format reflects this:

Byte	0	1	2	3	4	5 6 7 8 9
Hex	00	08	64	72	61	77 6B 63 61 42
Alpha	-	-	d	r	a	w k c a B

First of all, there is a zero byte - indicating a string - followed by a count of the number of characters in the string, and then the string itself. For reasons which will soon become clear, the string is held in reverse order.

The object of the first exercise is to write a function which will read a string from a file into a character array. The sequence of operations is as follows:

1. Fetch the string length, l.
2. Fetch l characters into our array, last one first.
3. Terminate said array with a null character.

For now we'll assume the file is opened and closed in another function such as **main()**.

```
/* read_bstring - read a string from a Basic-format
   file and place in str */

void read_bstring(FILE *in, char *str)
{ n ;      /* for input (fgetc returns int) */

  n=fgetc(in); /* zero */
  n=fgetc(in); /* string length */
  str[n--]='\0'; /* insert terminator */
  while(n >= 0)
    *(str + n--) = (char)fgetc(in);
}
```

As you can see, reverse storage and using n as an offset which decrements to zero produces more efficient code. As a result, str will contain the data from the file, and the file pointer will be positioned in the file ready to read the next byte. Note the function type *void*, signifying no return data.

Of course, no physical operation such as reading files can be foolproof; faulty discs and corrupt data must be catered for. The function *feof()* returns TRUE if end-of-file is reached, and the function *ferror()* returns TRUE if a file error has occurred. These could be incorporated into the while statement thus:

```
while(n >= 0 && !feof(in) && !ferror(in))
```

Where && is AND and ! is NOT. It would be better to indicate to the calling function whether there has been any trouble. This could be achieved by returning the exit value of n (-1 if all is well) or by returning zero on successful termination, or by returning the number of bytes read i.e. the length of the string. This is largely a matter of taste; there is no real standard in operation among the other C I/O operations. The closest equivalent is **fgets()**, which returns a character pointer to the fetched string if OK, or a null pointer if unsuccessful.

This would appear at the end of our source as:

```
if(n == -1) /* valid exit from 'while':
  return str;
else
  return 0;
and the function type must be changed from void to char *.
Here is a complete program Bstring, which calls read_bstring():
/* Bstring - read and display a file full of Basic-format strings */

#include <stdio.h>
#include <stdlib.h>

char *read_bstring(FILE *, char *);

int main()
{ *in ;
  char b_string[256];
```

```
in=fopen("Bstrings", "r");
if(!in){open the file!\n");
    exit(1);
}
while(read_bstring(in, b_string))
printf("%s\n", b_string);
if(!feof(in))
    printf("Some sort of file error.");
fclose(in);
}
```

Not much here you haven't seen before. I've #included `stdlib.h` because that contains the prototype for `exit()`. The function prototype doesn't need variable names, just types (this can look odd with pointer variables), and you don't need function prototypes if all functions are defined before use i.e. no forward references. This can make your programs look odd, with **main()** at the bottom, but they'll work just as well and you won't need to keep the prototypes in line with the functions - a real problem when developing large programs. However, prototypes serve to document your program, so you should consider putting them in when the number and type of functions (and their arguments) settle down.

Other odd lines are the `if` after **fopen()** and the `while` beneath it. These both evaluate something which will be NULL if an error has occurred. The `if` reads as '*if file-pointer in is not set*', and the `while` reads as '*while read_bstring returns a non-null value*'. This is common in C programs.

Introducing structures

An important feature of most powerful programming languages, but sadly absent from Basic, is the facility to handle user-defined data structures. A structure is a collection of variables which relate to one another in some fashion. A classic example is the payroll record - name, hourly rate, tax code, etc. Each part of a structure can itself be a structure - the tax code consists of an *int* and a *char*, for example. The only restriction is that a structure cannot contain a structure of the same type. C provides mechanisms for handling structures which enable you to handle large chunks of related data as a single entity.

First of all, the notation: to define a structure you simply list all the constituent parts in braces, like this:

```
struct example {name[20] ;
    char address[4][30];
    int age;
    float height, weight;
};
```

Note the semicolon following the closing brace. This is necessary because the braces form part of the struct declaration. When the compiler meets this, it does NOT reserve any space; it has simply been told about a new data type called `example` (in this instance). Later in the same program, you might define a variable thus:

```
struct example details;
```

This is the point when space is reserved for a variable called `details` of type `example`. `details` contains enough space for all the individual data items. To access these items, you must use an expression of the form:

```
details.age
```

The full stop is known as the structure member operator, for example:

```
details.age = this_year - year_of_birth;
strcpy(details.name, "Peter Pan");
scanf("%s", details.address[i]);
```

In general, a structure member can be treated as a simple variable of the same type.

Structures are often used in arrays. The notation is a logical extension of the simple arrays we've already met:

```
struct example details[10]; /* defines 10 structures */
details[n].age = days_old / 365;
scanf("%s", details[n].address[i]);
```

As you can see, the subscript follows the structure name, and is itself followed by the rest of the member name. A common cause of compilation errors is misplacement of subscripts in complex structures.

Naturally enough, you can also use pointers to access structures. This leads to some fun and games in the notation department:

```
struct example details[10], *ex;
/* define an array and a pointer */
ex = &details[n];
(*ex).weight = weight_in_ounces / 16;
```

You can substitute `*ex` wherever `details[n]` can occur, but brackets are necessary here because the structure member operator (`.`) has a higher precedence than the indirection operator (`*`). The expression `*ex.weight` means *'the pointer weight within the structure ex'*. Again, this is a rich source of programmer error, which is a shame because this is an immensely useful aspect of C.

To get round this a special operator, known as the structure pointer operator is used. In our example:

```
(*ex).weight
```

could be written as

```
ex->weight
```

The new operator is represented by a minus sign followed by a right angle bracket.

Naturally, you can have pointers within a structure, but what surprises many people is that a structure can contain pointers to the structure itself. One important application of this idea is in the linked list. In essence, the structure has three components:

```
struct line_data{ *txt ;           /* char pointer to text */
  struct line_data *prev;         /* pointer to prev line */
  struct line_data *next;         /* pointer to next line */
};
```

Insertion and deletion are handled by linking and unlinking pointers. If `p` points to a line we wish to delete, the two instructions:

```
p->prev->next = p->next;
p->next->prev = p->prev;
```

achieve this (the first and last lines require slightly different treatment). It works because `p->prev` is a pointer to the `line_data` structure just as `p` is, so `p->prev->next` is *'the pointer to the line after the previous one'*, which should be equal to `p` itself. The first line's `prev` would be set to `NULL`, as would the last line's `next` pointer; by definition, any pointer with a value of `NULL` doesn't point to anything (`NULL`, by the way, is a symbolic constant #defined in `stdio.h`). To present the data in order, we must know where the first line is:

```
struct line_data *first; /* set and maintained
                        during the program */
```

or we could search from the current position until we find the first line:

```
while(p->prev != NULL)
  p = p->prev;

p=first;
do{printf("%s\n" , p->txt);
  p = p->next;
}
while(p != NULL);
```

This simple technique is used in many sophisticated word processors and editors. It can be adapted to maintain a sorted list by pointing to the next larger and next smaller items. The name/height/weight example above could be adapted as follows:

```
struct example {name[20] ;
  char address[4][30];
  int age;
  float height, weight;
  struct example *older, *younger;
```

```
};
```

As data is captured, its place in the list is found by recursive comparisons with older->age and younger->age. This leads to a tree structure which is discussed in detail in Kernighan and Ritchie's book, *The C Programming Language*.

Formatted input and output

C is supplied with a whole family of input and output routines, several of which have already been used quite extensively. Two of the most important functions (**printf()** and **scanf()**) have already been discussed briefly, but this section describes them in more detail and covers other functions in the same family. A full technical description of these functions is given in chapter 5.

The functions under discussion consists of **fprintf()** and **fscanf()**, which transfer data to and from files, and **sprintf()** and **sscanf()**, which format data within the program's memory. Variable argument variants of these (**vprintf()** and so on), are also available, but will not be discussed here.

In general, **printf()**, **scanf()** etc. take a variable number of parameters one of which is known as a format control string. This effectively tells the function how many arguments have been passed to it, and controls the way in which they are processed. Because of this, its important that the number and type of arguments agrees with the format string, or dire consequences may result. The prototypes for these functions are:

```
int printf(char *format, ...);
int fprintf(FILE *fp, char *format, ...);
int sprintf(char *str, char *format, ...);
int scanf(char *format, ...);
int fscanf(FILE *fp, char *format, ...);
int sscanf(char *str, char *format, ...);
```

The **ellipsis (...)** indicates that zero or more arguments follow. The character array format can be a quoted string, as used in previous instalments, or it can be a variable. The functions are all of type int. The **printf()** group returns the number of characters output (in the case of **sprintf()**, excluding the terminating **\0**). The **scanf()** group returns the number of input items assigned.

The format string may contain any sequence of characters. These are simply copied to the output stream until a **%** or **** is encountered. The backslash introduces an escape sequence - an otherwise unprintable character, such as *Newline* (**\n**), *Formfeed* (**\f**), and so on. The percent sign marks the beginning of a conversion string. When **printf()** encounters a **%** it converts the next argument according to the characters following the percent sign. You've already met **%s** for strings (strictly, pointers of type **char**) and **%d** for decimal integers.

Other conversion types are:

Character	Meaning
x	Hexadecimal integer, using lower case a - f
X	Hexadecimal integer, using upper case A - F
f	Floating point number
e	Floating point number in the form n.n...nEmm
g	Select %f or %e whichever gives the shortest output
%	Print a percent sign

The full list will be found in chapter 5. While the percent sign marks the beginning of the conversion, the letter marks the end. Between the two you can specify various other characteristics of the output such as field width, accuracy, zero suppression and so on. The general format of a conversion string is **%-n.mlx** where:

- n is the minimum field width. If n has a leading zero, zero suppression is NOT performed (strictly speaking, the field is padded with zeros rather than spaces);
- m is the precision (for floating point numbers) or the maximum number of characters to be printed from a string, or the minimum number of digits for an integer;
- l is the lower-case letter L, used to signify that the variable is of type long (when used with ints) or double (when used with floats).
- x is the conversion character as above.

The optional minus sign indicates left justification of the converted value. Even strings are usually right-justified within their field!

Here are a few examples; each group is preceded by the definition of the variable printed. The colons show the exact size of the field, an idea I've borrowed from the standard work on the language, *The C Programming Language* by Kernighan and Ritchie.

```
int x = 13579;
Conversion string      Result
1.  :%d:               :13579:
2.  :%06d:             :013579:
3.  :%-8.3d:           :13579 :
4.  :%08.13d:          :0000000013579:
```

Line 1 shows the default format. In line 2 the minimum field size is specified and zero suppression is cancelled. The field is left justified in line 3, and line 4 shows how the precision overrides the minimum field width.

```
float one_seventh = 100000.0 / 7.0
Conversion string      Result
1.  :%f:               :14285.714286:
2.  :%06.9f:           :14285.714285714:
3.  :%9.12f:           :14285.714285714286:
4.  :%-29.22f:         :14285.7142857142862000000000 :
5.  :%.4f:             :14285.7143:
```

The definition requires explanation here. The numeric constants must have the decimal point otherwise C treats the division as of type int, which would yield exactly 14285 - the result is truncated, not rounded. Note how the field in line 4 is 29 characters wide including the decimal point and the trailing space, and that rounding does take place at line 5.

The answers are accurate to 12 decimal places (a total of 17 significant figures) which is sufficient for most purposes. The symbolic constants *FLT_DIG*, *DBL_DIG* and *LDBL_DIG*, #defined in *float.h*, give the 'digits of precision' that can be relied on for types float, double and long double respectively, which allows you to check the value prevailing at run time, and act accordingly. In Easy C they are all reliable to 15 significant figures.

```
char str[13] = "Twelve chars";
Conversion string      Result
1.  :%s:               :Twelve chars:
2.  :%15.10s:          :Twelve cha:
3.  :%-15.10s:         :Twelve cha:
4.  :%10s:             :Twelve chars:
5.  :%.10s:            :Twelve cha:
```

Note the right justification at line 2, and the difference between line 4 (where the minimum field width is specified) and line 5, where the 'accuracy' is specified. If you want to truncate a string from the left (giving twelve chars, for example) you should pass the appropriate pointer to **printf()** - in this case (*str+2*).

The fact that C will not truncate a numeric field to a specified size is both a blessing and a curse; it can play havoc with tables of figures, for example, but it does mean that no data is lost. It is possible to align columns by exploiting the fact that **printf()** returns the number of characters it has output, and another conversion character, *, which may appear either in the width or precision area. This returns the value of the next argument which is then used for the specification. The examples above were tabulated using this facility.

The left-hand part was printed thus:

```
x=printf("%2d :%s:", n, fmt);
/* n=1 to 5, converted by '%2d'
fmt= (for example) "%10s" converted by '%s' */
```

After this call to **printf()**, x contains the number of characters output. The right-hand part followed:

```
printf("%*s:", 20-x, " ");
/* the correct number of spaces */
printf(fmt, str);
/* str contains "Twelve chars" */
printf("\n");
/* end of the line */
```

The first **printf()** outputs a field exactly 20-x characters wide, which contains a single space (right-justified, if anyone cares). Note how the variable `fmt` has been used both as a format control string and as a variable - the poacher turned gamekeeper!

Here's a routine which exploits the variability of the field width. The object was to print totals held in the array `total[]` at the foot of columns of figures; the offset of the right-most digit in each column was stored in array `wid[]`. `max_n` holds the highest subscript used in the arrays.

```
for(n=max_n; n >= 0; n-)  
    printf("%*d\r", wid[n], total[n]);
```

This prints the rightmost total first, space filled to the left. The Escape sequence `\r` denotes Carriage Return, without Line Feed. This allows the totals to appear on the same line.

Formatted input - scanf()

The formatted input command **scanf()** works along broadly similar lines to **printf()**. The major difference is that the arguments to **scanf()** must be pointers to the variables which will hold the data. This is a good place to start looking if your C programs don't work! Almost everyone forgets sooner or later. The address of a variable `var` is given by `&var` or by a pointer to which the address has been assigned. We could have used **scanf()** in **write_bstring()** in the previous section; the inner do loop would be replaced by:

```
scanf("%s", &bstring[0]);
```

Remembering that the name of an array is a pointer to element 0, this could be written:

```
scanf("%s", bstring);
```

One disadvantage of **scanf()** is that any 'white space' character (the general term for Space, Tab, Vertical Tab, Formfeed, Carriage Return and Newline) terminates a conversion. This means that **write_bstring** would write output one word at a time.

Any non-conversion characters (such as the colons in the **printf()** examples above) must be matched on the input stream. If such a match fails **scanf()** terminates. This can obviously cause problems with keyboard input.

scanf() returns the number of items successfully matched and assigned. If end-of-file is reached during a conversion, **scanf()** returns the value `EOF` - this is #defined in `stdio.h` as a value which cannot be confused with a character or item count (usually -1, but test for EOF for portability!)

Why should you expect EOF in keyboard input? In the first place, many operating systems allow the input stream to be redirected - that is, input may be coming from a file (see below). Naturally, when the input file is exhausted you would want the program to be able to detect it. Secondly, you can simulate EOF at the keyboard by entering **Ctrl-D**. This also works on Unix and MS-DOS by the way. Finally, this allows the whole **scanf()** 'family' to behave in similar ways under analogous circumstances - the designers of C have sought everywhere to exclude special cases as far as possible.

Redirecting input and output

A remarkably useful feature of C programs is their ability to redirect their input and output. This facility is borrowed from UNIX and MS-DOS and their obsolete forebears. Essentially, a program which is written to take input from the keyboard and output to the screen can take input from a file, write its output to a file or both. The syntax will be familiar to users of the above-mentioned operating systems. To run a C program called `myprog`, using input stored in keystrokes and writing output to `printout`, you would type the following at the command line:

```
*myprog <keystrokes >printout
```

The `<` controls redirection of the standard input stream (`stdin`), while the `>` controls the standard output stream (`stdout`). What about the standard error stream (`stderr`)? In the present case, any output sent to the standard error stream would still appear on the screen. To send the error stream to a file called `errors`, you would type the following:

```
*myprog <keystrokes >printout 2>errors
```

The basis for this notation lies in the conception of `stdin` as stream 0, `stdout` as stream 1 and `stderr` as stream 2. The zero and one are not usually specified though you can do so if you like. To redirect `stdout` and `stderr` to the

same file, you could type:

```
*myprog 0<keystrokes 1>printout 2>&1
```

or in a shorter form:

```
*myprog 0<keystrokes >&printout
```

which means 'send stream 2 to wherever stream 1 is going'. If the 2>&1 appeared before the >printout the stderr output would still appear on the screen as at that point that is the current destination of stdout.

It is useful, when developing programs, to have some way of following progress on screen, so I would not recommend redirecting stderr until you are satisfied your program works properly, or unless you want to preserve the output for a later postmortem.

Command line arguments

A popular way of invoking programs, of the sort you'll be creating initially, is:

```
*myprog arg1 arg2
```

This would cause the program myprog to be loaded and run (provided it existed in the current directory or the library). arg1 and arg2 are 'command line arguments', which are passed to the program by the operating system. But how does the program get hold of them?

Up to now, when example programs have appeared in this tutorial, the function **main()** has been followed by empty parentheses or the word void. However, **main()** can have two arguments. The first is an argument *count*, the second an array of char pointers, each pointing to a word on the command line. By convention, these two arguments are known as *argc* and *argv* respectively. The full prototype of main now reads:

```
<type> main(int argc, char *argv[])
```

where <type> would be void or int or whatever. The second argument, by the way, reads '*an array argv[] of type char **'. Mastering the art of interpreting complex declarations is an achievement of no small difficulty - take my advice and stick to the simple ones wherever possible.

argc will always be at least 1, in the case where the only word on the command line is the program name. Thus, the contents of **argv[0]** will always be the program name as entered at the keyboard. This allows you to detect if the directory path was used, if you really want to know (and I dare say there's scope for a spot of software protection here). A word is defined as a string of characters bounded by unquoted white space. Thus:

```
myprog hello mum      /* argc = 3 */
```

but:

```
myprog "hello mum"    /* argc = 2 */
```

Successive entries in **argv[]** will point to successive words on the command line. As with most things, C trusts the programmer to stay within the bounds of the array. And here we come to another rich source of errors in C programs; the number of arguments after the command line and subsequent program name (held in **argv[0]**) is thus one less than argc as : the first effective argument is **argv[1]** and the last is **argv[argc-1]**.

Here's a brief example of a program which takes arguments:

```
/* Calc.c - prints the result of arithmetic */
/* expression in arguments 1-3 e.g. 75 / 15 */
/* Written by Dec McSweeney 1991 */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{op1 , op2;

if(argc != 4){fprintf(stderr , "Wrong number of arguments\n");
  exit(1);
```

```
    }

    op1 = atoi(argv[1]);
    /* function atoi(str) returns int value of string */
    op2 = atoi(argv[3]);

    switch(*argv[2]){ '+' : printf("%d\n", op1 + op2);
        break;
        case '-':
            printf("%d\n", op1 - op2);
            break;
        case '*':
            printf("%d\n", op1 * op2);
            break;
        case '/':
            printf("%d\n", op1 / op2);
            break;
        default:
            fprintf(stderr, "Unknown operation");
            break;
    }
}
```

Note how the switch statement uses ***argv[2]**, that is, the character pointed to by **argv[2]** (each entry in the array **argv[]** is a pointer of type **char** remember), while **atoi()** takes as arguments the pointers themselves.

The arguments above are positional, that is they must appear in a fixed sequence and their meaning to the program is indicated by their position. More normally, at least some of the arguments will not be order-dependent and in the case of a program with many options could be quite numerous - think about writing a C version of the RISC OS ***COPY** command.

The usual way to handle such a situation is to process each argument in turn using a for loop:

```
for(n=1; n<argc; n++){... argv[n] processing .. */
}
```

Observe that **n** starts at 1, but the terminating condition is **n<argc**, so the last iteration processes **argv[argc-1]**. For a command like ***COPY** the first two arguments would be positional (and compulsory), but the remainder could appear in any order. In such a case the program would validate for at least two arguments (**argc >= 3**) and the for loop would start at the first optional argument. Optional parameters should be set to their defaults first, then reset if mentioned in the command invocation. Mandatory parameters can be set to invalid values first, then checked for valid values after all the arguments have been processed.

Some useful functions

The standard C library contains a large number of useful functions. Many, such as the input and output routines, have already been discussed. One group of functions that has not been covered is concerned with the dynamic allocation of memory.

In a multi-user operating system such as UNIX, or a multi-program environment such as RISC OS, it is important that programs do not take up more memory than necessary. If your word processor is to be used for a short letter, it doesn't need enough memory to store a novel. In the same way, it is frustrating if a program imposes unnecessary limitations on the user. One popular word processor runs on the IBM PC (a 640K machine), but will only cope with documents of up to 32K!

The easiest way to cope with this problem is to provide the program with memory when it needs it, and to claim it back when the program finishes with it. The functions **malloc()** and **free()** provide this facility. The full prototype of **malloc** is:

```
void *malloc(size_t s);
```

This means that **malloc()** returns a pointer of type **void**. A void pointer is simply a mechanism for converting pointers from one type to another using casting. So, if the memory is required for a character array, the function would be called thus:

```
char *str;
size_t array_size; /* set array_size to the required value
before malloc */
```

```
str = (char *)malloc(array_size);
```

Remember that `size_t` is an implementation-defined variable type which is used for variable sizes. The C function **sizeof()** is useful when using **malloc()**:

```
/* A portable call to claim space for 1000 ints */
/* The size of int varies from machine to machine */
int *numbers;
numbers=(int *)malloc(1000 * sizeof(int));
```

If **malloc()** cannot claim sufficient space from the operating system it will return a null pointer.

The structures discussed earlier in this chapter are ideally suited to the use of **malloc()**. A typical structure representing a block of text in a word processor might look like this:

```
struct block {*text ;
    struct *block prev;
    struct *block next;
};
struct *block bptr;
```

To claim more memory as extra lines are added:

```
bptr->next = (struct block *)malloc(sizeof(struct block));
/* link the new block to the chain */
bptr->next->prev = bptr;
bptr=bptr->next;
bptr->next = NULL;          /* the end of the chain */
```

Once the size of the block is established, **malloc()** can be used to allocate the text area:

```
size_t block_size;
bptr->text = (char *)malloc(block_size);
To release a block of memory, the function free() is used:
free(bptr->text);
/* unlink from the chain */
bptr->next->prev = bptr->next;
bptr->prev->next = bptr->prev; free(bptr);
```

If a block needs to be enlarged, the function **realloc()** will attempt to provide extra space. This may involve a **malloc()** for the new size followed by a copy, so the value of the pointer may change:

```
new_block = (char *)realloc(bptr->text, new_size);
if(new_block != NULL)
    bptr->text = new_block;
```

One limitation of these standard functions is that the spare memory can become fragmented, and while there may be sufficient memory there may not be a large enough contiguous chunk to service a request.

Furthermore, Acorn's implementation provides memory from the amount allocated in the program's original Wimp slot. Once this is used up, **malloc()** will return null pointers (implying no more memory), no matter how much there may actually be.

Bit variables

If you use flags extensively in your programs and you're coming from an assembler background, you might not like the idea of using a whole integer (4 bytes) to store a simple yes-no value. If your programs are huge, you might not have room! In a well designed language it should be possible to store a flag in a single bit, and this is true in C.

One approach is to use **#define** to set up bit-values. Returning to the *COPY example, options could be defined as powers of 2, corresponding to a bit:

```
#define FORCE_OVERWRITE 1
#define DELETE_SOURCE 2
#define VERBOSE 4
/* ... and so on */
int options;
```

To set a flag, write:

```
options |= VERBOSE;    /* '|' is the bitwise OR */
```

To unset it, use:

```
options &= ~VERBOSE;    /* '&' is the bitwise AND */
```

This uses a new operator: the tilde (~) which means NOT in bitwise operations. Note also the use of the <op>= shorthand.

To test such a flag, you would write:

```
if((options & VERBOSE) == 0)
    /* i.e. if VERBOSE is off */
```

However, C allows you to define a structure containing bit-fields:

```
struct {int force_overwrite : 1;
        unsigned int delete_source : 1;
        unsigned int verbose : 1;
    } copy_options;
```

The width of the field in bits is specified by the number after the colon. In this example, copy_options contains three one-bit fields which you can access in the same way as any other structure member. For example:

```
copy_options.verbose = 1;
if(copy_options.delete_source == 0)
```

Using SWIs

A feature of RISC OS is the well documented interface to operating system routines. In Basic this is achieved with the SYS command, in assembler, SWI. Of course, C also provides a way of accessing them. Naturally enough, what follows is specific to RISC OS machines.

A header file, kernel.h, contains function prototypes, #defines and other relevant data for making SWI calls, but there are possibly better alternatives in other libraries, such as *DeskLib* (a freeware WIMP library).

Information is passed to and from SWI routines in the registers 0-9. These are defined as a **structure** `_kernel_swi_regs` in the *kernel.h* file. In your program you should include the following lines:

```
#include <kernel.h>
_kernel_swi_regs my_regs;
```

Individual registers are treated as ints and accessed as in this example:

```
myregs.r[4] = 1099;
```

Registers are often used to hold the addresses of (or pointers to) character strings or other blocks of data. C is well-suited to this requirement. The address of any variable is given by the operator &, or in the case of arrays by the unsubscripted array name:

```
char message[50];
struct line_data edit_line;
my_regs.r[0] = (int)message;
my_regs.r[0] = (int)&edit_line;
```

Pointers are cast as integers, which is the type of the register array elements. You can even pass the address of a 'character constant' or 'string literal':

```
my_regs.r[0] = (int) "This is a literal";
```

To actually call a SWI, use the routine `_kernel_swi`:

```
_kernel_swi(int swi_number, _kernel_swi_regs *in, _kernel_swi_regs *out)
```

The SWI numbers are #defined in swis.h so you can refer to them by their names, for example:

```
#include <swis.h>
_kernel_swi(os_ReadVarVal, &my_regs, &my_regs);
The following short program uses SWI calls to display any portion of the time/date
display.
/* Date.c - print any portion of Time message in any      */ /* specified format (see PRM)
Example: date "%WE, %MO */
/* the %DY%ST" would print "Tuesday, December the 25th" */ /* Written by Dec McSweeney
*/

#include <stdio.h>
#include <stdlib.h>
#include <kernel.h>
#include <swis.h>

int main(int argc, char *argv[])
{pblock[5] ;
char outstring[50];
int n;
_kernel_swi_regs r;

/** First call OS_Word 14,3 - read clock as 5-byte val **/
/** r0=14, r1=address of parameter block to hold result**/
/** on entry 1st byte of r1 is set to 3,
        the "reason code"      **/

r.r[0]=14;
r.r[1]=(int)pblock;
pblock[0]= 3;
_kernel_swi(OS_Word, &r, &r);

/** on exit, pblock holds the 5-byte time - compare with a Basic command "SYS 07, 14,
pblock T0 , pblock" **/
/** first we show the contents of pblock...      **/

for(n=0; n < 5; n++)
    printf("%02X ", pblock[n]); printf("\n");

/** Now call SWI &C1 to convert. argv[1] is the char **/
/** array holding the format string. For this call **/
/** the 5-byte block is addressed via r0, the output **/
/** string via r1, the format string via r3. The **/
/** length of output array is in r2. **/

r.r[0]=r.r[1];
r.r[1]=(int)outstring;
r.r[2]=50;
r.r[3]=(int)argv[1]; _kernel_swi(OS_ConvertDateAndTime, &r, &r);
printf("%s\n", outstring);
exit(0);
}
```

As it stands, the program will fall over if there is no format string in **argv[1]**, and does no validation on the format, so it relies on **OS_ConvertDateAndTime** to cope with any rubbish the user might type. To test the program, run it from the command line as follows:

```
*date "%WE, %MO the %DY%ST"
```

Now you know how to do it, why not change date.c to report an error if no argument is present, or even to use a default string - you could use **OS_ReadVarVal** to access *SYS\$DateFormat*, or call **OS_ConvertStandardDateAndTime** (SYS &60) which uses *SYS\$DateFormat* instead of r3. A really user-friendly version would print helpful information if **argv[1]** was missing, invalid or set to ?.

Chapter 2 - ANSI Functions

All C programs linked to the library STDClib have access to the full range of standard ANSI functions. EasyC is supplied with all 15 header files which declare these ANSI functions. The most useful of these are declared in the five header files listed below, and are described in detail in this chapter:

ctype.h	stdlib.h
math.h	string.h
stdio.h	

Many more ANSI functions are declared in the 10 header files listed below. All of these may be used from EasyC, but due to their more advanced nature they are not documented in this user guide. However, the syntax of these functions can be found by studying the header files.

assert.h	setjmp.h
errno.h	signal.h
float.h	stdarg.h
limits.h	stddef.h
locale.h	time.h

Further information about ANSI functions can be found in the Tutorial in the previous chapter, and in most good books on C programming. For the definitive text on the C programming language and descriptions of all ANSI functions you should refer to the IOC/IEC international standard called Programming languages – C (ref. ISO/IEC 9899).

Character handling <ctype.h>

The header <ctype.h> declares several functions useful for testing and mapping characters. In all cases the argument is an int, the value of which shall be represented as an unsigned char or shall equal the value of the macro EOF. If the argument has any other value the behaviour of the functions will be undefined.

The behaviour of these functions is affected by the current locale. those functions that have implementation-defined aspects only when not in the "C" locale are noted below.

The term printing character refers to a member of an implementation-defined set of characters, each of which occupies one printing position on a display device; the term control character refers to a member of an implementation-defined set of characters that are not printing characters.

Character testing functions

Name	isalnum
Syntax	int isalnum(int c);
Description	Test for any character for which isalpha or isdigit is true.
Returns	Nonzero (true) if and only if the argument c conforms to that in the description of the function.
Name	isalpha
Syntax	int isalpha(int c);
Description	Test for any character for which isupper or islower is true, or any character that is one of an implementation-defined set of characters for which not of iscntrl, isdigit, ispunct, or isspace is true. In the "C" locale, isalpha returns true only for characters for which isupper or islower is true.
Returns	Nonzero (true) if and only if the argument c conforms to that in the description of the function.
Name	iscntrl
Syntax	int iscntrl(int c);
Description	Test for any control character.
Returns	Nonzero (true) if and only if the argument c conforms to that in the description of the function.
Name	isdigit
Syntax	int isdigit(int c);
Description	Test for any decimal-digit character.

Returns	Nonzero (true) if and only if the argument <code>c</code> conforms to that in the description of the function.
Name	isgraph
Syntax	<code>int isgraph(int c);</code>
Description	Test for any printing character except space (' ').
Returns	Nonzero (true) if and only if the argument <code>c</code> conforms to that in the description of the function.
Name	islower
Syntax	<code>int islower(int c);</code>
Description	Test for any character that is a lowercase letter or is one of an implementation-defined set of characters for which none of <code>isctrl</code> , <code>isdigit</code> , <code>ispunct</code> , or <code>isspace</code> is true. In the "C" locale, <code>islower</code> returns true only for the characters defined as lowercase letters.
Returns	Nonzero (true) if and only if the argument <code>c</code> conforms to that in the description of the function.
Name	isprint
Syntax	<code>int isprint(int c);</code>
Description	Test for any printing character including space (' ').
Returns	Nonzero (true) if and only if the argument <code>c</code> conforms to that in the description of the function.
Name	ispunct
Syntax	<code>int ispunct(int c);</code>
Description	Test for any printing character that is neither space (' ') nor a character for which <code>isalnum</code> is true.
Returns	Non-zero (true) if and only if the argument <code>c</code> conforms to to that in the description of the function.
Name	isspace
Syntax	<code>int isspace(int c);</code>
Description	Test for any character that is a standard white-space character or is one of an implementation-defined set of characters for which <code>isalnum</code> is false. The standard white-space characters are - space (' '), form feed ('\f'), new-line ('\n'), carriage return ('\r'), horizontal tab ('\t'), and vertical tab ('\v'). in the "C" locale, <code>isspace</code> returns true only for the standard white-space characters.
Returns	Non-zero (true) if and only if the argument <code>c</code> conforms to to that in the description of the function.
Name	isupper
Syntax	<code>int isupper(int c);</code>
Description	Test for any character that is an uppercase letter or is one of an implementation-defined set of characters for which none of <code>isctrl</code> , <code>isdigit</code> , <code>ispunct</code> , or <code>isspace</code> is true. In the "C" locale, <code>isupper</code> returns true only for the characters defined as uppercase letters.
Returns	Non-zero (true) if and only if the argument <code>c</code> conforms to to that in the description of the function.
Name	isxdigit
Syntax	<code>int isxdigit(int c);</code>
Description	Test for any hexadecimal-digit character.
Returns	Non-zero (true) if and only if the argument <code>c</code> conforms to to that in the description of the function.
Character case mapping functions	
Name	tolower
Syntax	<code>int tolower(int c);</code>
Description	Convert an uppercase letter to the corresponding lowercase letter.

Returns	If the argument is a character for which <code>isupper</code> is true and there is a corresponding character for which <code>islower</code> is true, the <code>tolower</code> function returns the corresponding character; otherwise the argument is returned unchanged
Name	<code>toupper</code>
Syntax	<code>int toupper(int c);</code>
Description	Convert a lowercase letter to the corresponding uppercase letter.
Returns	If the argument is a character for which <code>islower</code> is true and there is a corresponding character for which <code>isupper</code> is true, the <code>toupper</code> function returns the corresponding character; otherwise, the argument is returned unchanged.

Mathematics <math.h>

The header <math.h> declares several mathematical functions and defines one macro. The functions take double arguments and return double values.

The macro is `HUGE_VAL` which expands to a positive double expression.

For all functions, a domain error occurs if an input argument is outside the domain over which the mathematical input is defined. The description of each function lists any required domain errors; an implementation may define additional domain errors, provided that such errors are consistent with the mathematical definition of the function. On a domain error, the function returns an implementation-defined value; the value of the macro `EDOM` is stored in `errno`.

Similarly, a range error occurs if the result of the function cannot be represented as a double value. If the result overflows (the magnitude of the result is so large that it cannot be represented in an object of the specified type), the function returns the value of the macro `HUGE_VAL` with the same sign (except for the `tan` function) as the correct value of the function; the value of the macro `ERANGE` is stored in `errno`. If the result underflows (the magnitude of the result is so small that it cannot be represented in an object of the specified type), the function returns zero; whether the integer expression `errno` acquires the value of the macro `ERANGE` is implementation-defined.

Trigonometric functions

Name	<code>acos</code>
Syntax	<code>double acos(double x);</code>
Description	Compute the principal value of the arc cosine of <code>x</code> . A domain error occurs for arguments not in the range <code>[-1, +1]</code> .
Returns	The arc cosine in the range <code>[0, p]</code> radians.
Name	<code>asin</code>
Syntax	<code>double asin(double x);</code>
Description	Compute the principal value of the arc sine of <code>x</code> . A domain error occurs for arguments not in the range <code>[-1, +1]</code> .
Returns	The arc sine in the range <code>[-p/2, +p/2]</code> radians.
Name	<code>atan</code>
Syntax	<code>double atan(double x);</code>
Description	Compute the principal value of the arc tangent of <code>x</code> .
Returns	The arc tangent in the range <code>[-p/2, +p/2]</code> radians.
Name	<code>atan2</code>
Syntax	<code>double atan2(double y, double x);</code>
Description	Compute the principal value of the arc tangent of <code>y/x</code> , using the signs of both arguments to determine the quadrant of the return value. A domain error occurs if both arguments are zero.
Returns	The arc tangent of <code>y/x</code> in the range <code>[-p, +p]</code> radians.
Name	<code>cos</code>
Syntax	<code>double cos(double x);</code>
Description	Compute the cosine of <code>x</code> (measured in radians).
Returns	The cosine value.

Name	sin
Syntax	double sin(double x);
Description	Compute the sine of x (measured in radians).
Returns	The sine value.
Name	tan
Syntax	double tan(double x);
Description	Compute the tangent of x (measured in radians).
Returns	The tangent value.

Hyperbolic functions

Name	cosh
Syntax	double cosh(double x);
Description	Compute the hyperbolic cosine of x. A range error occurs if the magnitude of x is too large.
Returns	The hyperbolic cosine value.
Name	sinh
Syntax	double sinh(double x);
Description	Compute the hyperbolic sine of x. A range error occurs if the magnitude of x is too large.
Returns	The hyperbolic sine value.
Name	tanh
Syntax	double tanh(double x);
Description	Computes the hyperbolic tangent of x.
Returns	The hyperbolic tangent value.

Exponential and logarithmic functions

Name	exp
Syntax	double exp(double x);
Description	Compute the exponential function of x. A range error occurs if the magnitude of x is too large.
Returns	The exponential value.
Name	frexp
Syntax	double frexp(double value, int *exp);
Description	Break a floating-point number into a normalised fraction and an integral power of 2. It stores the integer in the int object pointed to by exp.
Returns	The value x, such that x is a double with magnitude in the interval $[1/2, 1]$ or zero, and value equals x times 2 raised to the power *exp. If value is zero, both parts of the result are zero.
Name	ldexp
Syntax	double ldexp(double x, int exp);
Description	Multiply a floating-point number by an integral of power 2. A range error may occur.
Returns	The value of x times 2 raised to the power exp.
Name	log
Syntax	double log(double x);
Description	Compute the natural logarithm of x. A domain error occurs if the argument is negative. A range error occurs if the argument is zero.
Returns	The natural logarithm.
Name	log10
Syntax	double log10(double x);
Description	Compute the base-ten logarithm of x. A domain error occurs if the argument is

Returns	negative. A range error occurs if the argument is zero. The base-ten logarithm.
Name	modf
Syntax	double modf(double value, double *iptr);
Description	Break the argument value into integral and fractional parts, each of which has the same sign as the argument. It stores the integral part as a double in the object pointed to by iptr.
Returns	The signed fractional part of value.

Power functions

Name	pow
Syntax	double pow(double x, double y);
Description	Compute x raised to the power y. A domain error occurs if x is negative and y is not an integral value. A domain error occurs if the result cannot be represented when x is zero and y is less than or equal to zero. A range error may occur.
Returns	The value of x raised to the power y.
Name	sqrt
Syntax	double sqrt(double x);
Description	Compute the non-negative square root of x.
Returns	The value of the square root.

Miscellaneous functions

Name	ceil
Syntax	double ceil(double x);
Description	Compute the smallest integral value not less than x.
Returns	The smallest integral not less than x, expressed as a double.
Name	fabs
Syntax	double fabs(double x);
Description	Compute the absolute value of a floating-point number x.
Returns	The absolute value of x.
Name	floor
Syntax	double floor(double x);
Description	Compute the largest integral value not greater than x.
Returns	The largest integral value not greater than x, expressed as a double.
Name	fmod
Syntax	double fmod(double x, double y);
Description	Computes the floating-point remainder of x/y.
Returns	The value $x - i * y$, for some integer i such that, if y is nonzero, the result has the same sign as x and magnitude less than the magnitude of y. If y is zero, whether a domain error occurs or the fmod function returns zero is implementation-defined.

Input/output <stdio.h>

The header <stdio.h> declares three types, several macros, and many functions performing input and output.

- The type **size_t** is the unsigned integral type of the result of the sizeof operator.
- The type **FILE** is an object type capable of recording the information needed to control a stream, including its file position indicator, a pointer to its associated buffer (if any), an error indicator that records whether a read/write error has occurred, and an end-of file indicator that records whether the end of the file has been reached.
- The type **fpos_t** is an object type capable of recording all the information needed to specify uniquely every position within a file.
- The macro **NULL** expands to an implementation-defined null pointer constant.
- The macros **_IOFBF**, **_IOLBF** and **_IONBF** expand to integral constant expressions with distinct values, suitable for use as the third argument to the setvbuf function.
- The macro **BUFSIZ** expands to an integral constant expression, which is the size of the buffer used by the setbuf function.
- The macro EOF expands to a negative constant expression returned by several functions to indicate end-of-file, ie. no more input from a stream.
- The macro **FOPEN_MAX** expands to an integral constant expression that is the minimum number of files that the implementation guarantees can be open simultaneously.
- The macro **FILENAME_MAX** expands to an integral constant expression that is the size needed for an array of char large enough to hold the longest file name that the implementation guarantees can be opened.
- The macro **L_tmpnam** expands to an integral constant expression that is the size needed for an array of char large enough to hold a temporary file name string generated by the tmpnam function.
- The macros **SEEK_CUR**, **SEEK_END** and **SEEK_SET** expand to distinct integral constant expressions with distinct values, suitable for use as the third argument to the fseek function.
- The macro **TMP_MAX** expands to a integral constant expression that is the minimum number of unique file names that shall be generated by the tmpnam function.
- **stderr**, **stdin** and **stdout** are expressions of type *"pointer to FILE"* that point to the **FILE** objects associated, respectively, with the standard error, input and output streams.

Operations on files

Name **remove**

Syntax int remove(const char *filename);

Description Cause the file whose name is the string pointed to by filename to be no longer accessible by that name. A subsequent attempt to remove that file using that name will fail, unless it is created anew. If the file is open, the remove function is implementation-defined.

Returns Zero if the operation succeeds, nonzero if it fails.

Name **rename**

Syntax int rename (const char *old, const char *new);

Description Cause the file whose name is the string pointed to by old to be henceforth known by the name given by the string pointed to by new. The file named old is no longer accessible by that name. If the file named new already exists, the rename function will fail.

Returns Zero if the operation succeeds, nonzero if it fails, in which case if the file existed previously it is still known by its original name.

Name **tmpfile**

Syntax FILE *tmpfile(void);

Description Create a temporary binary file that will automatically be removed when it is closed or at program termination. If the program terminates abnormally, whether an open temporary file is removed is implementation-defined. The file is opened for update with "wb+" mode.

Returns A pointer to the stream of the file that it created. If the file cannot be created, the tmpfile function returns a null pointer.

Name **tmpnam**

Syntax char *tmpnam(char *s);

Description	Generate a string that is a valid filename and that is not the same as the name of an existing file. The tmpnam function generates a different string each time it is called, up to TMP_MAX times. If it is called more than TMP_MAX times, the behaviour is implementation-defined. The implementation shall behave as if no library function calls the tmpname function.
Returns	If the argument is a null pointer, the tmpnam function leaves its result in an internal static object and returns a pointer to that object. Subsequent call to the tmpnam function may modify the same object. If the argument is not a null pointer, it is assumed to point to an array of at least L_tmpnam chars; the tmpnam function writes its result in that array and returns the argument as its value.

File access functions

Name	fclose
Syntax	int fclose(FILE *stream);
Description	Cause the stream pointed to by stream to be flushed and the associated file to be closed. Any unwritten buffered data for the stream are delivered to the host environment to be written to the file; any unread buffered data are discarded. The stream is disassociated from the file. If the associated buffer was automatically allocated, it is deallocated.
Returns	Zero if the stream was successfully closed, or EOF if any errors were detected.
Name	fflush
Syntax	int fflush(FILE *stream);
Description	If stream points to an output or an update stream in which the most recent operation was not input, the fflush function causes any unwritten data for that stream to be delivered to the host environment to be written to the file; otherwise, the behaviour is undefined. If stream is a null pointer, the fflush function performs the flushing action on all streams for which the behaviour is defined above.
Returns	EOF if a write error occurs otherwise, zero.
Name	fopen
Syntax	FILE *fopen(const char *filename, const char *mode);
Description	Open the file whose name is the string pointed to by filename, and associate a stream with it. The argument mode points to a string beginning with one of the following sequences: r w a rb wb ab r+ w+ a+ r+b or rb+ +b or wb+ a+b or ab+ open text file for reading truncate to zero length or create text file for writing append; open or create text file for writing at EOF open binary file for reading truncate to zero length or create binary file for writing append; open or create binary file for writing at EOF open text file for update (reading and writing) truncate to zero length or create text file for update append; open or create text file for update, writing at EOF open binary file for update (read and write) truncate to zero length or create binary file for update append; open or create bin file for update, writing at EOF. Opening a file with read mode ('r' as the first character in the mode argument) fails if the file does not exist or cannot be read. Opening a file with append mode ('a' as the first character in the mode argument) causes all subsequent writes to the file to be forced to the then current end-of-file, regardless of intervening calls to the fseek function. When a file is opened with update mode ('+' as the second or third character in the above list of mode argument values), both input and output may be performed on the associated stream. However, output may not be directly followed by input without an intervening call to the fflush function or to a file positioning function

	(fseek, fsetpos, or rewind), and input may not be directly followed by output without an intervening call to a file positioning function, unless the input operation encounters end-of-file.
Returns	A pointer to the object controlling the stream. If the open operation fails, fopen returns a null pointer.
Name	freopen
Syntax	FILE *freopen(const char *filename, const char *mode, FILE *stream);
Description	Open the file whose name is the string pointed to by filename and associates the stream pointed to by stream with it. The mode argument is used just as in the fopen function. The freopen function first attempts to close any file that is associated with the specified stream. Failure to close the file successfully is ignored. The error and end-of-file indicators for the stream are cleared.
Returns	A null pointer if the open operation fails. Otherwise, freopen returns the value of stream.
Name	setbuf
Syntax	void setbuf(FILE *stream, char *buf);
Description	Except that it returns no value, setbuf is equivalent to the setvbuf function invoked with the value _IOFBF for mode and BUFSIZ for size, or (if buf is a null pointer), with the value _IONBF for mode.
Returns	Void
Name	setvbuf
Syntax	int setvbuf(FILE *stream, char *buf, int mode, size_t size);
Description	May be used after the stream pointed to by stream has been associated with an open file and before any other operation is performed on the stream. The argument mode determines how stream will be buffered, as follows: _IOFBF causes input/output to be fully buffered; _IOLBF causes input/output to be line buffered; _IONBF causes input/output to be unbuffered. If buf is not a null pointer, the array it points to may be used instead of a buffer allocated by the setvbuf function. The argument size specifies the size of the array. The contents of the array at any time are indeterminate.
Returns	Zero on success, or nonzero if an invalid value is given for mode or if the request cannot be honoured.
	Formatted input/output functions
Name	fprintf
Syntax	int fprintf(FILE *stream, const char *format, ...);
Description	Write output to the stream pointed to by stream, under control of the string pointed to by format that specifies how subsequent arguments are converted for output. If there are insufficient arguments for the format, the behaviour is undefined. If the format is exhausted while arguments remain, excess arguments are evaluated but otherwise ignored. The fprintf function returns when the end of the format string is encountered. The format is composed of zero or more directives: ordinary multibyte 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 modifying the conversion specification - The conversion will be left-justified within the field. + A signed conversion begins with a plus or minus sign. space If the first character of a signed conversion is not a sign, or if a signed conversion results in no characters, a space will be prefixed to the result. # The result is to be converted to an "alternate form". 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) prefixed to it. For e, E, f, g, and G conversions, the result will always contain a decimal-point character, even if no digits follow it. For g and G conversions, trailing zeros will not be removed from the result. 0 For d, i, o, u, x, X, e, E, f, g, and G conversions leading zeros are used to pad to the

- field width; no space padding is performed. If the 0 and - flags both appear the 0 flag is ignored. For d, i, o, u, x, and X conversion, if a precision is specified the 0 flag is ignored.
- An optional minimum field width. If the converted value has less characters than the field width it will be padded with spaces on the left (or right, if the left adjustment flag, see later, has been given) to the field width.
 - An optional precision that gives the minimum number of digits to appear for the d, i, o, u, x and X conversions, the number of digits to appear after the decimal-point character for e, E, and f conversions, the maximum number of significant digits for the g and G conversions, or the maximum number of characters to be written from a string in s conversion. The precision takes the form of a period (.) followed either by an asterisk or a decimal integer.
 - 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; an optional h specifying that a following n conversion specifier applies to a pointer to a short int argument; 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; an optional l specifying that a following n conversion specifier applies to a pointer to a long int argument; or an optional L specifying that the following e, E, f, g, or G conversion specifier applies to a long double argument.
 - A character that specifies the type of conversion to be applied.
- A field width, or precision, or both may be indicated by an asterisk. In this case, an int argument supplies the field width or precision. The arguments specifying field width, or precision, or both, shall appear (in that order) before the argument (if any) to be converted.

The conversion specifiers and their meanings are:

d,i	The int argument is converted to signed decimal in the style [-]dddd.
o,u,x,X	The unsigned int argument is converted to octal (o), unsigned decimal (u), or unsigned hexadecimal notation (x or X) in the style dddd; the letters abcdef are used for x conversion and the letters ABCDEF for X conversion.
f	The double argument is converted to decimal notation in the style [-]ddd.ddd, where the number of digits after the decimal-point is equal to the precision. If the precision is missing, it is taken as 6; if the precision is zero and the # flag is not specified, no decimal-point character appears.
e,E	The double argument is converted in the style [-]d.ddde±dd, where there is one digit before the decimal-point character (which is nonzero if the argument is nonzero) and the number of digits after it is equal to the precision. If the precision is missing, it is taken as 6; if the precision is zero and the # flag is not specified, no decimal-point character appears.
g,G	The double argument is converted in style f or e (or in style E in the case of a G conversion specifier), with the precision specifying the number of significant digits. If the precision is zero, it is taken as 1. The style used depends on the value converted; style e (or E) will be used only if the exponent resulting from such a conversion is less than -4 or greater than or equal to the precision. Trailing zeros are removed from the fractional portion of the result; a decimal-point character appears only if it is followed by a digit.
c	The int argument is converted to an unsigned char, and the resulting character is written.
s	The argument shall be a pointer to an array of character type. Characters from the array are written up to (but not including) a terminating null character; if the precision is specified, no more than that many characters are written.
p	The argument shall be a pointer to void. The value of the pointer is converted to a sequence of printable characters, in an implementation-defined manner.
n	The argument shall be a pointer to an integer into which is written the number of characters written to the output stream so far by this call to fprintf. No argument is converted.
%	A % is written. No argument is converted.

Returns	The number of characters transmitted, or a negative value if an output error occurred.
Name	fscanf
Syntax	<code>int fscanf(FILE *stream, const char *format, ...);</code>
Description	Read input from the stream pointed to by stream, under control of the string pointed to by format that specifies the admissible input sequences and how they are to be converted for assignment, using subsequent arguments as pointers to the objects to receive the converted input. If there are insufficient arguments for the format the behaviour is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated (as always) but are otherwise ignored. The format shall be a multibyte character sequence, beginning and ending in its initial shift state. the format is composed of of zero or more directives: one or more white-space characters; an ordinary multibyte character (neither % nor a white-space character); or a conversion specification. Each conversion specification is introduced by the character %. After the %, the following appear in sequence: — An optional assignment-suppressing character *. — An optional nonzero decimal integer that specifies the maximum field width. — An optional h, l (ell) or L indicating the size of the receiving object. The conversion specifiers d, i, and n shall be preceded by h if the corresponding argument is a pointer to short int rather than a pointer to int, or by l if it is a pointer to long int. Similarly, the conversion specifiers o, u, and x shall be preceded by h if the corresponding argument is a pointer to unsigned short int rather than a pointer to unsigned int, or by l if it is a pointer to unsigned long int. Finally, the conversion specifiers e, f, and g shall be preceded by l if the corresponding argument is a pointer to double rather than a pointer to float, or by L if it is a pointer to long double. — A character that specifies the type of conversion to be applied.

The following conversion specifiers are valid:

d	Matches an optionally signed decimal integer. The corresponding argument shall be a pointer to integer.
i	Matches an optionally signed integer. The corresponding argument shall be a pointer to integer.
o	Matches an optionally signed octal integer. The corresponding argument shall be a pointer to unsigned integer.
u	Matches an optionally signed decimal integer. The corresponding argument shall be a pointer to unsigned integer.
x	Matches an optionally signed hexadecimal integer. The corresponding argument shall be a pointer unsigned integer.
e,f,g	Matches an optionally signed floating-point number. The corresponding argument shall be a pointer to floating.
s	Matches a sequence of non-white-space characters. The corresponding argument shall be a pointer to the initial character on an array large enough to accept the sequence and a terminating null character, which will be added automatically.
[	Matches a nonempty sequence of characters from a set of expected characters (the scanset). The corresponding argument shall be a pointer to the initial character of an array large enough to accept the sequence and a terminating null character, which will be added automatically. the conversion specifier includes all subsequent characters in the format string, up to and including the matching right bracket (]). The characters between the brackets (the scanlist) comprise the scanset, unless the character after the left bracket is a circumflex (^), in which case the scanset contains all characters that do not appear in the scanlist between the circumflex and the right bracket. If the conversion specifier begins with [] or [^], the right bracket character is in the scanlist.
c	Matches a sequence of characters of the number specified by the field width (1 if no field width is present in the directive). The corresponding argument shall be a pointer to the initial character of an array large enough to accept the sequence. No null character is added.
p	Matches an implementation-defined set of sequences, which should be the same as the set of sequences that may be produced by the %p conversion of the fprintf

	function. The corresponding argument shall be a pointer to a pointer to void.
n	No input is consumed. The corresponding argument shall be a pointer to integer into which is to be written the number of characters read from the input stream so far by this call to the fscanf function. Execution of a %n directive does not increment the assignment count returned at the completion of execution of the fscanf function.
%	Matches a single %; no conversion or assignment occurs. The complete conversion specification shall be %%.
Returns	The value of the macro EOF if an input failure occurs before any conversion. Otherwise, the fscanf function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.
Name	printf
Syntax	int printf(const char *format, ...);
Description	Equivalent to fprintf with the argument stdout interposed before the arguments to printf.
Returns	The number of characters transmitted, or a negative value if an output error occurred.
Name	scanf
Syntax	int scanf(const char *format, ...);
Description	Equivalent to fscanf with the argument stdin interposed before the arguments to scanf.
Returns	The value of the macro EOF if an input failure occurs before any conversion. Otherwise, the scanf function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.
Name	sprintf
Syntax	int sprintf(char *s, const char *format, ...);
Description	Equivalent to fprintf, except that the argument s specifies an array into which the generated output is to be written, rather than to a stream. A null character is written at the end of the characters written; it is not counted as part of the returned sum.
Returns	Return the number of characters written in the array, not counting the terminating null character.
Name	sscanf
Syntax	int sscanf(const char *s, const char *format, ...);
Description	Equivalent to fscanf, except that the argument s specifies a string from which the input is obtained, rather than from a stream. Reaching the end of the string is equivalent to encountering end-of-file for the fscanf function.
Returns	The value of the macro EOF if an input failure occurs before any conversion. Otherwise, the sscanf function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.
Name	vfprintf
Syntax	int vfprintf(FILE *stream, const char *format, va_list arg);
Description	Equivalent to fprintf, with the variable argument list replaced by arg, which shall have been initialised by the va_start macro (and possibly subsequent va_arg calls). The vfprintf function does not invoke the va_end function.
Returns	The number of characters transmitted, or a negative value if an output error occurred.
Name	vprintf
Syntax	int vprintf(const char *format, va_list arg);
Description	Equivalent to printf, with the variable argument list replaced by arg, which shall have been initialised by the va_start macro (and possible subsequent va_arg calls). The vprintf function does not invoke the va_end function.
Returns	The number of characters transmitted, or a negative value if an output error occurred.

Name	vsprintf
Syntax	<code>int vsprintf(char *s, const char *format, va_list arg);</code>
Description	Equivalent to <code>sprintf</code> , with the variable argument list replaced by <code>arg</code> , which shall have been initialised by the <code>va_start</code> macro (and possibly subsequent <code>va_arg</code> calls). The <code>vsprintf</code> function does not invoke the <code>va_end</code> function.
Returns	The number of characters written in the array, not counting the terminating null character.

Character input/output functions

Name	fgetc
Syntax	<code>int fgetc(FILE *stream);</code>
Description	Obtain the next character as an unsigned char converted to an int from the input stream pointed to by stream and advance the file position indicator for the stream.
Returns	The next character from the input stream pointed to by stream. If the stream is at end-of-file, the end-of-file indicator is set and fgetc returns EOF. If a read error occurs, the error indicator is set and it returns EOF.
Name	fgets
Syntax	<code>char *fgets(char *s, int n, FILE *stream);</code>
Description	Read at the most one less than the number of characters specified by n from the stream pointed to by stream into the array pointed to by s. No additional characters are read after a new-line character (which is retained) or after an end-of-file. A null character is written immediately after the last character read into the array.
Returns	If successful returns s. If end-of-file is encountered and no characters have been read into the array the array contents are unchanged and a null pointer is returned. If a read error occurs during the operation array contents are indeterminate and a null pointer is returned.
Name	fputc
Syntax	<code>int fputc(int c, FILE *stream);</code>
Description	Write the character specified by c (converted to an unsigned char) to the output stream pointed to by stream, at the position indicated by the associated file position indicator for the stream (if defined), and advances the indicator appropriately. If the file cannot support positioning requests, or if the stream was opened with append mode, the character is appended to the output stream.
Returns	The character written. If a write error occurs the error indicator is set and fputc returns EOF.
Name	fputs
Syntax	<code>int fputs(const char *s, FILE *stream);</code>
Description	Write the string pointed to by s to the stream pointed to by stream. The terminating null character is not written.
Returns	EOF if a write error occurs; otherwise a non-negative value.
Name	getc
Syntax	<code>int getc(FILE *stream);</code>
Description	Is equivalent to fgetc, except that if it is implemented as a macro, it may evaluate stream more than once, so the argument should never be an expression with side effects.
Returns	The next character from the input stream pointed to by stream. If the stream is at end-of-file, the end-of-file indicator is set and getc returns EOF. If a read error occurs, the error indicator is set and it returns EOF.
Name	getchar
Syntax	<code>int getchar(void);</code>
Description	Is equivalent to getc with the argument stdin.
Returns	The next character from the input stream pointed to by stdin. If the stream is at end-of-file, the end-of-file indicator for the stream is set and getchar returns EOF. If a read error occurs, the error indicator is set and getchar returns EOF.
Name	gets
Syntax	<code>char *gets(char *s);</code>
Description	Read characters from the input stream pointed to by stdin, into the array pointed to by s, until end-of-file is encountered or a new-line character is read. Any new-line character is discarded, and a null character is written immediately after the last character read into the array.
Returns	If it is successful it returns s. If end-of-file is encountered and no characters have been read into the array, the contents of the array remain unchanged and a null pointer is returned. If a read error occurs during the operation, the array contents are

indeterminate and a null pointer is returned.

Name	putc
Syntax	<code>int putc(int c, FILE *stream);</code>
Description	Is equivalent to <code>fputc</code> , except that if it is implemented as a macro, it may evaluate <code>stream</code> more than once, so the argument should never be an expression with side effects.
Returns	The character written. If a write error occurs, the error indicator for the stream is set and <code>putc</code> returns EOF.

Name	putchar
Syntax	<code>int putchar(int c);</code>
Description	Is equivalent to <code>putc</code> with the second argument <code>stdout</code> .
Returns	The character written. If a write error occurs, the error indicator for the stream is set and <code>putchar</code> returns EOF.

Name	puts
Syntax	<code>int puts(const char *s);</code>
Description	Write the string pointed to by <code>s</code> to the stream pointed to by <code>stdout</code> , and appends a new-line character to the output. The terminating null character is not written.
Returns	EOF if a write error occurs; otherwise it returns a non-negative value.

Name	ungetc
Syntax	<code>int ungetc(int c, FILE *stream);</code>
Description	Push the character specified by <code>c</code> (converted to an unsigned char) back onto the input stream pointed to by <code>stream</code>. The pushed-back characters will be returned by subsequent reads on that stream in the reverse order of their pushing. A successful intervening call (with the stream pointed to by <code>stream</code>) to a file positioning function (<code>fseek</code>, <code>fsetpos</code>, or <code>rewind</code>) discards any pushed-back characters for the stream. The external storage corresponding to the stream is unchanged. One character of pushback is guaranteed. If the <code>ungetc</code> function is called too many times on the same stream without an intervening read or file positioning operation on that stream the operation may fail. If the value of <code>c</code> equals that of the macro <code>EOF</code>, the operation fails and the input stream is unchanged. A successful call to the <code>ungetc</code> function clears the end-of-file indicator for the stream. The value of the file position indicator after reading or discarding all pushed-back characters shall be the same as it was before the characters were pushed back. For a text stream the value of the file position indicator after a successful call to the <code>ungetc</code> function is unspecified until all the pushed-back characters are read or discarded. For a binary stream its file position indicator is decremented by each successful call to the <code>ungetc</code> function; if it's value was zero before a call, it is indeterminate after the call.
Returns	The character pushed back after conversion, or EOF if the operation fails.

Direct input/output functions

Name	fread
Syntax	<code>size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);</code>
Description	Read into an array pointed to by <code>ptr</code> , up to <code>nmemb</code> members whose size is specified by <code>size</code> , from the stream pointed to by <code>stream</code> . The file position indicator for the stream (if defined) is advanced by the number of characters successfully read. If an error occurs, the resulting value of the file position indicator for the stream is indeterminate. If a partial member is read it's value is indeterminate.
Returns	The number of members successfully read, which maybe less than <code>nmemb</code> if a read error or end-of-file is encountered. If <code>size</code> or <code>nmemb</code> is zero, <code>fread</code> returns zero and the contents of the array and the state of the stream remain unchanged.

Name	fwrite
Syntax	<code>size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);</code>
Description	Write from the array pointed to by <code>ptr</code> , up to <code>nmemb</code> members whose size is specified by <code>size</code> , to the stream pointed to by <code>stream</code> . The file position indicator for the stream (if defined) is advanced by the number of characters successfully written. If an error

Returns	occurs, the resulting value of the file position indicator for the stream is indeterminate. The number of members successfully written, which will be less than nmemb only if a write error is encountered.
---------	---

File positioning functions

Name	fgetpos
Syntax	<code>int fgetpos(FILE *stream, fpos_t *pos);</code>
Description	Store the current value of the file position indicator for the stream pointed to by stream in the object pointed to by pos. The value stored contains unspecified information usable by the fsetpos function for repositioning the stream to its position at the time of the call to the fgetpos function.
Returns	If successful, the fgetpos function returns zero; on failure, the fgetpos function returns nonzero and stores an implementation-defined positive value in errno.
Name	fseek
Syntax	<code>int fseek(FILE *stream, long int offset, int whence);</code>
Description	Set the file position indicator for the stream pointed to by stream. For a binary stream the new position, measured in characters from the beginning of the file, is obtained by adding offset to the position specified by whence. The specified position is the beginning of the file if whence is SEEK_SET, the current value of the file position indicator is SEEK_CUR, or end-of-file if SEEK_END. A binary stream need not meaningfully support fseek calls with a whence value of SEEK_END. For a text stream, either offset shall be zero, or offset shall be a value returned by an earlier call to the ftell function on the same stream and whence shall be SEEK_SET. A successful call to the fseek function clears the end-of-file indicator for the stream and undoes any effects of the ungetc function on the same stream. After an fseek call, the next operation on an update stream may be either input or output.
Returns	Nonzero only for a request that cannot be satisfied.
Name	fsetpos
Syntax	<code>int fsetpos(FILE *stream, const fpos_t *pos);</code>
Description	Set the file position indicator for the stream pointed to by stream according to the value of the object pointed to by pos, which shall be a value obtained from an earlier call to the fgetpos function on the same stream. A successful call to the fsetpos function clears the end-of-file indicator for the stream and undoes any effects of the ungetc function on the same stream. After an fsetpos call, the next operation on an update stream may be either input or output.
Returns	If successful the fsetpos function returns zero; on failure, the fsetpos function returns nonzero and stores an implementation-defined positive value in errno.
Name	ftell
Syntax	<code>long int ftell(FILE *stream);</code>
Description	Obtain the current value of the file position indicator for the stream pointed to by stream. For a binary stream, the value is the number of characters from the beginning of the file. For a text stream, its file position indicator contains unspecified information usable by the fseek function for returning the file position indicator for the stream to its position at the time of the ftell call; the difference between two such return values is not necessarily a meaningful measure of the number of characters written or read.
Returns	If successful the ftell function returns the current value of the file position indicator for the stream. On failure, the ftell function returns -1L and stores an implementation-defined positive value in errno.
Name	rewind
Syntax	<code>void rewind(FILE *stream);</code>
Description	Set the file position indicator for the stream pointed to by stream to the beginning of the file.
Returns	Void.

Error handling functions

Name	clearerr
Syntax	void clearerr(FILE *stream);
Description	Clear the end-of-file and error indicators for the stream pointed to by stream.
Returns	Void.

Name	feof
Syntax	int feof(FILE *stream);
Description	Test the end-of-file indicator for the stream pointed to by stream.
Returns	A non-zero value if and only if the end-of-file indicator is set for stream.

Name	ferror
Syntax	int ferror(FILE *stream);
Description	Test the error indicator for the stream pointed to by stream.
Returns	Non-zero if and only if the error indicator is set for stream.

Name	perror
Syntax	void perror(const char *s);
Description	Map the error number in the integer expression errno to an error message. It writes a sequence of characters to the standard error stream thus : first (if s is not a null pointer and the character pointed to by s is not the null character), the string pointed to by s followed by a colon (:)and a space; then an appropriate error message string followed by a new-line character. The contents of the error message strings are the same as those returned by the strerror function with argument errno, which are implementation defined.
Returns	Void.

General utilities <stdlib.h>

The header <stdlib.h> declares four types, defines several macros and several functions of general utility, and defines several macros.

- The type `size_t` is the unsigned integral type of the result of the `sizeof` operator.
- The type `wchar_t` is an integral type whose range of values represent distinct codes for all members of the largest extended character set specified among the supported locales; the null character shall have the code value zero and each member of the basic character set shall have a code value equal to its value when used as the lone character in an integer character constant.
- The type `div_t` is a structure type that is the type of the value returned by the `div` function.
- The type `ldiv_t` is a structure type that is the type of the value returned by the `ldiv` function.
- The macro `NULL` expands to an implementation-defined null pointer constant.
- The macros `EXIT_FAILURE` and `EXIT_SUCCESS` expand to integral expressions that may be used as the argument to the `exit` function to return unsuccessful or successful termination status, respectively, to the host environment.
- The macro `RAND_MAX` expands to an integral constant expression, the value of which is the maximum value returned by the `rand` function.
- The macro `MB_CUR_MAX` expands to a positive integer expression whose value is the maximum number of bytes in a multibyte character for the extended character set specified by the current locale (category `LC_CTYPE`), and whose value is never greater than `MB_LEN_MAX`.

String conversion functions

Name	atof
Syntax	<code>double atof(const char *nptr);</code>
Description	Convert the initial portion of the string pointed to by nptr to double representation. Except for the behaviour on error, it is equivalent to: <code>strtod(nptr, (char **)NULL)</code>
Returns	The converted value.
Name	atoi
Syntax	<code>int atoi(const char *nptr);</code>
Description	Convert the initial portion of the string pointed to by nptr to int representation. Except for error behaviour it is equivalent to: <code>(int)strtol(nptr, (char **)NULL, 10);</code>
Returns	The converted value.
Name	atol
Syntax	<code>long int atol(const char *nptr);</code>
Description	Convert the initial part of the string pointed to by nptr to long int representation. Except for error behaviour it is equivalent to: <code>strtol(nptr, (char **)NULL, 10);</code>
Returns	The converted value.
Name	strtod
Syntax	<code>double strtod(const char *nptr, char **endptr);</code>
Description	Convert the initial part of the string pointed to by nptr to double representation. It decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by the <code>isspace</code> function), a subject sequence resembling a floating-point constant; and a final string of one or more unrecognised characters, including the terminating null character of the input string. Then it attempts to convert the subject sequence to a floating-point number. The expected form of the subject sequence is an optional plus or minus sign, then a nonempty sequence of digits optionally containing a decimal-point character, then an optional exponent part, but no floating suffix. The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is empty or consists entirely of white space, or if the first non-white-space character is other than a sign, a digit, or a decimal-point character. If the subject sequence has the expected form, the sequence of characters starting with the first digit or the decimal-point character (whichever occurs first) is interpreted as a floating constant according to the rules, except that the decimal-point character is used in place of a period, and that if neither an exponent part nor a decimal-point character appears, a decimal point is assumed to follow the last digit in the string. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated. A pointer to the final string is stored in the object pointed to by <code>endptr</code>, provided that <code>endptr</code> is not a null pointer. If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of <code>nptr</code> is stored in the object pointed to by <code>endptr</code>, provided that <code>endptr</code> is not a null pointer.
Returns	The converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, plus or minus <code>HUGE_VAL</code> is returned (according to the sign of the value), and the value of the macro <code>ERANGE</code> is stored in <code>errno</code> . If the correct value would cause an underflow, zero is returned and the value of the macro <code>ERANGE</code> is stored in <code>errno</code> .
Name	strtol
Syntax	<code>long int strtol(const char *nptr, char **endptr, int base);</code>
Description	Convert the initial portion of the string pointed to by nptr to long int representation. First it decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by the <code>isspace</code> function), a subject sequence resembling an integer represented in some radix determined by the value <code>base</code> , and a final string of one or more unrecognised characters including the

terminating null character of the input string. Then, it attempts to convert the subject sequence to an integer, and returns the result.

If the value of base is zero, the expected form of the subject sequence is that of an integer constant, optionally preceded by a plus or minus sign, but not including an integer suffix. If the value base is between 2 and 36, the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by base, optionally preceded by a plus or minus sign, but not including an integer suffix. The letters from a (or A) through z (or Z) are ascribed the values 10 to 35; only letters whose ascribed values are less than that of base are permitted. If the value of base is 16, the characters 0x or 0X may optionally precede the sequence of letters or digits, following the sign if present.

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is empty or consists entirely of white space, or if the first non-white-space character is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of base is zero, the sequence of characters starting with the first digit is interpreted as an integer constant according to the rules. If the subject sequence has the expected form and base is between 2 and 36, it is used as the base for conversion, ascribing to each letter its value as given above. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated. A pointer to the final string is stored in the object pointed to by endptr, provided that endptr is not a null pointer. If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of nptr is stored in the object pointed to by endptr, provided that endptr is not a null pointer.

Returns The converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, LONG_MAX or LONG_MIN is returned (according to the sign of the value), and the value of the macro ERANGE is stored in errno.

Name **strtoul**

Syntax unsigned long int strtoul(const char *nptr, char **endptr, int base);

Description Convert the initial portion of the string pointed to by nptr to unsigned long int representation. First it decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by the isspace function), a subject sequence resembling an unsigned integer represented in some radix determined by the value of base, and a final string of one or more unrecognised characters, including the terminating null character of the input string. Then, it attempts to convert the subject sequence to an unsigned integer, and returns the result.

If the value of base is zero, the expected form of the subject sequence is that of an integer constant, optionally preceded by a plus or minus sign, but not including an integer suffix. If the value base is between 2 and 36, the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by base, optionally preceded by a plus or minus sign, but not including an integer suffix. The letters from a (or A) through z (or Z) are ascribed the values 10 to 35; only letters whose ascribed values are less than that of base are permitted. If the value of base is 16, the characters 0x or 0X may optionally precede the sequence of letters or digits, following the sign if present.

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is empty or consists entirely of white space, or if the first non-white-space character is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of base is zero, the sequence of characters starting with the first digit is interpreted as an integer constant according to the rules. If the subject sequence has the expected form and the value of base is between 2 and 36, it is used as the base for conversion, ascribing to

each letter its value as given above. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated. A pointer to the final string is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer. If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of `nptr` is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

Returns

The converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, `ULONG_MAX` is returned, and the value of the macro `ERANGE` is stored in `errno`.

Pseudo-random sequence generation functions.

Name	rand
Syntax	int rand(void);
Description	Compute a sequence of pseudo-random integers in the range 0 to RAND_MAX.
Returns	A pseudo-random integer.
Name	srand
Syntax	void srand(unsigned int seed);
Description	Use the argument as a seed for a new sequence of pseudo-random numbers to be returned by subsequent calls to rand. If srand is then called with the same seed value, the sequence of pseudo-random numbers will be repeated. If rand is called before any calls to srand have been made, the same sequence shall be generated as when srand is first called with a seed value of 1.
Returns	Void.

Memory management functions

The order and contiguity of storage allocated by successive calls to the calloc, malloc and realloc functions is unspecified. The pointer returned if the allocation succeeds is suitable aligned so that it may be assigned to a pointer to any type of object and then used to access such an object or array of such objects in the space allocated (until the space is explicitly freed or reallocated). The pointer returned points to the start (lowest byte address) of the allocated space. If the space cannot be allocated, a null pointer is returned.

Name	calloc
Syntax	void *calloc(size_t nmemb, size_t size);
Description	Allocate space for an array of nmemb objects, each of whose size is size. The space is initialised to all bits zero.
Returns	Either a null pointer, or a pointer to the allocated space.
Name	free
Syntax	void free(void *ptr);
Description	Cause the space pointed to by ptr to be deallocated, that is, made available for further allocation. If ptr is a null pointer, no action occurs. Otherwise, if the argument does not match the pointer earlier returned by the calloc, malloc or realloc function, or if the space has been deallocated by a call to free or realloc, the behaviour is undefined.
Returns	Void.
Name	malloc
Syntax	void *malloc(size_t size);
Description	Allocate space for an object whose size is specified by size and whose value is indeterminate.
Returns	Either a null pointer or a pointer to the allocated space.
Name	realloc
Syntax	void *realloc(void *ptr, size_t size);
Description	Change the size of the object pointed to by ptr to the size specified by size. The contents of the object will be unchanged up to the lesser of the new and old sizes. If the new size is larger, the value of the newly allocated portion of the object is indeterminate. If ptr is a null pointer the realloc function behaves like the malloc function for the specified size. Otherwise, if ptr does not match the pointer earlier returned by the calloc, malloc or realloc function, or if the space has been deallocated by a call to the free or realloc function, the behaviour is undefined. If the space cannot be allocated, the object pointed to by ptr is unchanged. If size is zero and ptr is not a null pointer, the object it points to is freed.
Returns	Either a null pointer or a pointer to the possible moved allocated space.

Communication with the environment

Name **abort**

Syntax `void abort(void);`

Description Cause abnormal program termination to occur, unless the signal SIGABRT is being caught and the signal handler does not return. Whether open output streams are flushed or open streams closed or temporary files are removed is implementation-defined. An implementation-defined form of the status unsuccessful termination is returned to the host environment by means of the function call `raise(SIGABRT)`.

Returns Cannot return to it's caller.

Name **atexit**

Syntax `int atexit(void(*func)(void));`

Description Register the function pointed to by `func`, to be called without arguments at normal program termination.

Returns Zero if the registration succeeds, nonzero if it fails.

Name **exit**

Syntax `void exit(int status);`

Description Cause normal program termination to occur. If more than one call to the `exit` function is executed by a program the behaviour is undefined. First all functions registered by the `atexit` function are called, in the reverse order of their registration. Next all open output streams with unwritten buffered data are flushed, all open streams are closed, and all files created by the `tmpfile` function are removed. Finally control is returned to the host environment. If the value of `status` is zero or `EXIT_SUCCESS`, an implementation-defined form of the status successful termination is returned. If the value of `status` is `EXIT_FAILURE`, an implementation-defined form of the status unsuccessful termination is returned. Otherwise `status` returned is implementation-defined.

Returns Cannot return to it's caller.

Name **getenv**

Syntax `char *getenv(const char *name );`

Description Search an environment list provided by the host environment for a string that matches the string pointed to by `name`. The set of environment names and the method for altering the environment list are implementation defined. The implementation should behave as if no library function calls the `getenv` function.

Returns A pointer to a string associated with the matched list member. The string pointed to shall not be modified by the program, but may be overwritten by a subsequent call to the `getenv` function. If the specified name cannot be found, a null pointer is returned.

Name **system**

Syntax `int system(const char *string);`

Description Pass the string pointed to by `string` to the host environment to be executed by a command processor in an implementation-defined manner. A null pointer may be used for `string` to inquire whether a command processor exists.

Returns If the argument is a null pointer, the `system` function returns nonzero only if there is a command processor available. If the argument is not a null pointer, the `system` function returns zero for success or -2 for failure. Any other value is the return code from the executed command.

Searching and sorting utilities

Name	bsearch
Syntax	<code>void *bsearch(const void *key, const void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *));</code>
Description	Search an array of nmemb objects, the initial element of which is pointed to by base, for an element that matches the object pointed to by key. The size of each element of the array is specified by size. The comparison function pointed to by compar is called with two arguments that point to the key object and to an array element, in that order. The function shall return an integer less than, equal to, or greater than zero if the key object is considered, respectively, to be less than, to match, or to be greater than the array element. The array shall consist of: all the elements that compare less than, all the elements that compare equal to, and all the elements that compare greater than the key object, in that order.
Returns	A pointer to a matching element of the array, or a null pointer if no match is found. If two members compare as equal, which element is matched is unspecified.
Name	qsort
Syntax	<code>void qsort(void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *));</code>
Description	Sort an array of nmemb objects, the initial element of which is pointed to by base. The size of each object is specified by size. The contents of the array are sorted into ascending order according to a comparison function pointed to by compar, which is called with two arguments that point to the objects being compared. The function shall return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second. If two elements compare as equal, their order in the sorted array is unspecified.
Returns	Void.
Integer arithmetic functions	
Name	abs
Syntax	<code>int abs(int j);</code>
Description	Compute the absolute value of an integer j. If the result cannot be represented, the behaviour is undefined.
Returns	The absolute value.
Name	div
Syntax	<code>div_t div(int numer, int denom);</code>
Description	Compute the quotient and remainder of the division of the numerator numer by the denominator denom. If the division is inexact, the resulting quotient is the integer of lesser magnitude that is the nearest to the algebraic quotient. less than the magnitude of the mathematical quotient. If the result cannot be represented, the behaviour is undefined; otherwise quot * denom + rem shall equal numer.
Returns	A structure of type div_t, comprising both the quotient and remainder. The structure shall contain the following members, in either order: <code>int quot; /* quotient */int rem; /* remainder */</code>
Name	labs
Syntax	<code>long int labs(long int j);</code>
Description	Similar to the abs function, except that the argument and the returned value each have type long int.
Name	ldiv
Syntax	<code>ldiv_t ldiv(long int numer, long int denom);</code>
Description	Similar to the div function, except that the arguments and members of the returned structure (which has type ldiv_t) all have type long int.

Multibyte character functions

The behaviour of the multibyte character functions is affected by **LC_CTYPE** category of the current locale. For a state-dependent encoding, each function is placed into its initial state by a call for which its character pointer argument, *s*, is a null pointer. Subsequent calls with *s* as other than a null pointer cause the internal state of the function to be altered as necessary. A call with *s* as a null pointer causes these functions to return a nonzero value if encodings have state dependency, and zero otherwise. Changing the **LC_CTYPE** category causes the shift state of these functions to be indeterminate.

Name	mblen
Syntax	<code>int mblen (const char *s, size_t n);</code>
Description	If <i>s</i> is not a null pointer, the <code>mblen</code> function determines the number of bytes contained in the multibyte character pointed to by <i>s</i> . Except that the shift state of the <code>mbtowc</code> function is not affected, it is equivalent to: <code>mbtowc((wchar_t *)0, s, n);</code>
Returns	If <i>s</i> is a null pointer, the <code>mblen</code> function returns a nonzero or zero value, if multibyte character encodings, respectively, do or do not have state-dependent encodings. If <i>s</i> is not a null pointer, the <code>mblen</code> function either returns 0 (if <i>s</i> points to the null character), or returns the number of bytes that are contained in the multibyte character (if the next <i>n</i> or fewer bytes form a valid multibyte character), or returns -1 (if they do not form a valid multibyte character).
Name	mbtowc
Syntax	<code>int mbtowc (wchar_t *pwc, const char *s, size_t n);</code>
Description	If <i>s</i> is not a null pointer, the <code>mbtowc</code> function determines the number of bytes that are contained in the multibyte character pointed to by <i>s</i> . It then determines the code for the value of type <code>wchar_t</code> that corresponds to that multibyte character. (The value of the code corresponding to the null character is zero.) If the multibyte character is valid and <i>pwc</i> is not a null pointer, the <code>mbtowc</code> function stores the code in the object pointed to by <i>pwc</i> . At most <i>n</i> bytes of the array pointed to by <i>s</i> will be examined. The implementation behaves as if no library function calls the <code>mbtowc</code> function.
Returns	If <i>s</i> is a null pointer, the <code>mbtowc</code> function returns a nonzero or zero value, if multibyte character encodings, respectively, do or do not have state-dependent encodings. If <i>s</i> is not a null pointer, the <code>mbtowc</code> function either returns 0 (if <i>s</i> points to the null character), or returns the number of bytes that are contained in the converted multibyte character (if the next <i>n</i> or fewer bytes form a valid multibyte character), or returns -1 (if they do not form a valid multibyte character). In no case will the value returned be greater than <i>n</i> or the value of the <code>MB_CUR_MAX</code> macro.
Name	wctomb
Syntax	<code>int wctomb (char *s, wchar_t wchar);</code>
Description	Determine the number of bytes needed to represent the multibyte character corresponding to the code whose value is <code>wchar</code> (including any change in shift state). It stores the multibyte character representation in the array object pointed to by <i>s</i> (if <i>s</i> is not a null pointer). At most <code>MB_CUR_MAX</code> characters are stored. If the value of <code>wchar</code> is zero, the <code>wctomb</code> function is left in the initial shift state.
Returns	If <i>s</i> is a null pointer, the <code>wctomb</code> function returns a nonzero or zero value, if multibyte character encodings, respectively, do or do not have state-dependent encodings. If <i>s</i> is not a null pointer, the <code>wctomb</code> function returns -1 if the value of <code>wchar</code> does not correspond to a valid multibyte character, or returns the number of bytes that are contained in the multibyte character corresponding to the value of <code>wchar</code> . In no case will the value returned be greater than the value of the <code>MB_CUR_MAX</code> macro.

Multibyte string functions

The behaviour of the multibyte string functions is affected by the LC_CTYPE category of the current locale.

Name	mbstowcs
Syntax	size_t mbstowcs (wchar_t *pwcs, const char *s, size_t n);
Description	Convert a sequence of multibyte characters that begins in the initial shift state from the array pointed to by s into a sequence of corresponding codes and stores not more than n codes into the array pointed to by pwcs. No multibyte characters that follow a null character (which is converted into a code with value zero) will be examined or converted. Each multibyte character is converted as if by a call to mbtowc, except that the shift state of the mbtowc function is not affected. No more than n elements will be modified in the array pointed to by pwcs. If copying takes place between objects that overlap, the behaviour is undefined.
Returns	If an invalid multibyte character is encountered, the mbstowcs function returns (size_t)-1. Otherwise, it returns the number of array elements modified, not including a terminating zero code, if any.

Name	wcstombs
Syntax	size_t wcstombs (char *s, const wchar_t *pwcs, size_t n);
Description	Convert a sequence of codes that correspond to multibyte characters from the array pointed to by pwcs into a sequence of multibyte characters that begins in the initial shift state and stores these multibyte characters into the array pointed to by s, stopping if a multibyte character would exceed the limit of n total bytes or if a null character is stored. Each code is converted as if by a call to the wctomb function, except that the shift state of the wctomb function is not affected. No more than n bytes will be modified in the array pointed to by s. If copying takes place between objects that overlap, the behaviour is undefined.
Returns	If a code is encountered that does not correspond to a valid multibyte character, the wcstombs function returns (size_t)-1. Otherwise, the wcstombs function returns the number of bytes modified, not including a terminating null character, if any.

String handling <string.h>

The header <string.h> declares one type and several functions, and defines one macro useful for manipulating arrays of character type and other objects treated as arrays of character type. Various methods are used for determining the lengths of the arrays, but in all cases a **char *** or **void *** argument points to the initial (lowest addressed) character of the array. If the array is accessed beyond the end of an object, the behaviour is undefined.

- The type `size_t` is the unsigned integral type of the result of the `sizeof` operator.
- The macro `NULL` expands to an implementation-defined null pointer constant.

Copying functions

Name	memcpy
Syntax	void *memcpy(void *s1, const void *s2, size_t n);
Description	Copy n characters from the object pointed to by s2 into the object pointed to by s1. If copying takes place between objects that overlap, the behaviour is undefined.
Returns	The value of s1.
Name	memmove
Syntax	void *memmove(void *s1, const void *s2, size_t n);
Description	Copy n characters from the object pointed to by s2 into the object pointed to by s1. Copying takes place as if n characters from the object pointed to by s2 are first copied into a temporary array of n characters that does not overlap the objects pointed to by s1 and s2, and then the n characters from the temporary array are copied into the object pointed to by s1.
Returns	The value of s1.
Name	strcpy
Syntax	char *strcpy(char *s1, const char *s2);

Description	strcpy copies the string pointed to by s2 (including the terminating null character) into the array pointed to by s1.
Returns	the value of s1.
Name	strncpy
Syntax	char *strncpy(char *s1, const char *s2, size_t n);
Description	Copy not more than n characters (characters that follow a null character are not copied) from the array pointed to by s2 to the array pointed to by s1. If copying takes place between objects that overlap, the behaviour is undefined. If the array pointed to by s2 is a string that is shorter than n characters, null characters are appended to the copy in the array pointed to by s1, until n characters in all have been written.
Returns	The value of s1.

Concatenation functions

Name	strcat
Syntax	char *strcat(char *s1, const char *s2);
Description	Appends a copy of the string pointed to by s2 (including the terminating null character) to the end of the string pointed to by s1. The initial character of s2 overwrites the null character at the end of s1. If copying takes place between objects that overlap, the behaviour is undefined.
Returns	The value of s1.
Name	strncat
Syntax	char *strncat(char *s1, const char *s2, size_t n);
Description	Append not more than n characters (a null character and characters that follow it are not appended) of the array pointed to by s2 to the end of the string pointed to by s1. The initial character of s2 overwrites the null character at the end of s1. A terminating null character is always appended to the result. If copying takes place between objects that overlap, the behaviour is undefined.
Returns	The value of s1.

Comparison functions

The sign of a nonzero value returned by the comparison functions `memcmp`, `strcmp`, and `strncmp` is determined by the sign of the difference between the values of the first pair of characters (both interpreted as unsigned char) that differ in the objects being compared.

Name **memcmp****Syntax** `int memcmp(const void *s1, const void *s2, size_t n);`**Description** Compare the first *n* characters of the object pointed to by *s1* to the first *n* characters of the object pointed to by *s2*.**Returns** An integer greater than, equal to, or less than zero, accordingly as the object pointed to by *s1* is greater than, equal to, or less than the object pointed to by *s2*.**Name** **strcmp****Syntax** `int strcmp(const char *s1, const char *s2);`**Description** Compare the string pointed to by *s1* to the string pointed to by *s2*.**Returns** An integer greater than, equal to, or less than zero, depending accordingly as the string pointed to by *s1* is greater than, equal to, or less than the string pointed to by *s2*.**Name** **strcoll****Syntax** `int strcoll (const char *s1, const char *s2);`**Description** Compare the string pointed to by *s1* to the string pointed to by *s2*, both interpreted as appropriate to the LC_COLLATE category of the current locale.**Returns** An integer greater than, equal to, or less than zero accordingly as the string pointed to by *s1* is greater than, equal to, or less than the string pointed to by *s2* when both are interpreted as appropriate to the current locale.**Name** **strncmp****Syntax** `int strncmp(const char *s1, const char *s2, size_t n);`**Description** Compare not more than *n* characters (characters that follow a null character are not compared) from the array pointed to by *s1* to the array pointed to by *s2*.**Returns** An integer greater than, equal to, or less than zero accordingly as the string pointed to by *s1* is greater than, equal to, or less than the possibly null-terminated array pointed to by *s2*.**Name** **strxfrm****Syntax** `size_t strxfrm (char *s1, const char *s2, size_t n);`**Description** Transform the string pointed to by *s2* and places the resulting string into the array pointed to by *s1*. The transformation is such that if the `strcmp` function is applied to two transformed strings, it returns a value greater than, equal to, or less than zero, corresponding to the result of the `strcoll` function applied to the same two original strings. No more than *n* characters are placed into the resulting array pointed to by *s1*, including the terminating character. If *n* is zero, *s1* is permitted to be a null pointer. If copying takes place between objects that overlap the behaviour is undefined**Returns** The length of the transformed string (not including the terminating null character). If the value returned is *n* or more, the contents of the array pointed to by *s1* are indeterminate.

Search functions

Name	memchr
Syntax	<code>void *memchr(const void *s, int c, size_t n);</code>
Description	Locate the first occurrence of <code>c</code> (converted to an unsigned char) in the initial <code>n</code> characters (each interpreted as unsigned char) of the object pointed to by <code>s</code> .
Returns	A pointer to the located character, or a null pointer if the character does not occur in the object.
Name	strchr
Syntax	<code>char *strchr(const char *s, int c);</code>
Description	Locate the first occurrence of <code>c</code> (converted to a char) in the string pointed to by <code>s</code> . The terminating null character is considered to be part of the string.
Returns	A pointer to the located character, or a null pointer if the character does not occur in the string.
Name	strcspn
Syntax	<code>size_t strcspn(const char *s1, const char *s2);</code>
Description	Compute the length of the maximum initial segment of the string pointed to by <code>s1</code> which consists entirely of characters not from the string pointed to by <code>s2</code> .
Returns	The length of the segment.
Name	strpbrk
Syntax	<code>char *strpbrk(const char *s1, const char *s2);</code>
Description	Locate the first occurrence in the string pointed to by <code>s1</code> of any character from the string pointed to by <code>s2</code> .
Returns	A pointer to the located character, or a null pointer if no character from <code>s2</code> occurs in <code>s1</code> .
Name	strrchr
Syntax	<code>char *strrchr(const char *s, int c);</code>
Description	Locate the last occurrence of <code>c</code> (converted to char) in the string pointed to by <code>s</code> . The terminating null character is considered to be part of the string.
Returns	A pointer to the character, or a null pointer if <code>c</code> does not occur in the string.
Name	strspn
Syntax	<code>size_t strspn(const char *s1, const char *s2);</code>
Description	Compute the length of the maximum initial segment of the string pointed to by <code>s1</code> which consists entirely of characters from the string pointed to by <code>s2</code> .
Returns	The length of the segment.
Name	strstr
Syntax	<code>char *strstr(const char *s1, const char *s2);</code>
Description	Locate the first occurrence in the string pointed to by <code>s1</code> of the sequence of characters (excluding the terminating null character) in the string pointed to by <code>s2</code> .
Returns	A pointer to the located string, or a null pointer if the string is not found. If <code>s2</code> points to a string with zero length, the function returns <code>s1</code> .
Name	strtok
Syntax	<code>char *strtok(char *s1, const char *s2);</code>
Description	A sequence of calls to the <code>strtok</code> function breaks the string pointed to by <code>s1</code> into a sequence of tokens, each of which is delimited by a character from the string pointed to by <code>s2</code>. The first call in the sequence has <code>s1</code> as its first argument, and is followed by calls with a null pointer as their first argument. The separator string pointed to by <code>s2</code> may be different from call to call. The first call in the sequence searches the string pointed to by <code>s1</code> for the first character that is not contained in the current separator string pointed to by <code>s2</code>. If no such character is found, then there are no tokens in the string pointed to by <code>s1</code> and the <code>strtok</code> function returns a null pointer. If such a character is found, it is the start of the first token.

The strtok function then searches from there for a character that is contained in the current separator string. If no such character is found, the current token extends to the end of the string pointed to by s1, and subsequent searches for a token will return a null pointer. If such a character is found, it is overwritten by a null character, which terminates the current token. The strtok function saves a pointer to the following character from which the next search for a token will start.

Each subsequent call, with a null pointer as the value of the first argument, starts searching from the saved pointer and behaves as described above.

Returns A pointer to the first character of a token, or a null pointer if there is no token.

Miscellaneous string functions

Name **memset**

Syntax void *memset(void *s, int c, size_t n);

Description Copy the value of c (converted to an unsigned char) into each of the first n characters of the object pointed to by s

Returns The value of s.

Name **strerror**

Syntax char *strerror(int errnum);

Description Map the error number in errnum to an error message string.

Returns A pointer to the string. The array pointed to shall not be modified by the program, but may be overwritten by a subsequent call to the strerror function.

Name **strlen**

Syntax size_t strlen(const char *s);

Description Compute the length of the string pointed to by s.

Returns The number of characters that precede the terminating null character.

Chapter 3 - RISC OS functions

This chapter lists in alphabetical order all the machine-specific functions supplied with Easy C. The syntax of each function is given with a brief description of what it does and what it returns. In addition, there is a self-contained example program for each function which demonstrates how it is used. For your convenience all the examples are supplied with EasyC disc in the directory *Programs.OSExamples*.

RISC OS functions are contained in the library *ROSLib*, and you should `#include` the header file *roslib.h* in any program that uses them. For further information about the operating system calls made by these functions, please refer to the Acorn *Programmers Reference Manual* for your computer system.

Many of the functions described in this chapter are of type `os_error`; they return a pointer to an error if one has occurred, otherwise zero if the operation was successful. The example program below shows how to trap the error, and display the error number and error message.

```
/* Example of trapping an error */

/* Execute an illegal command to generate
   an error. */

#include <stdio.h>
#include "roslib.h"

void main(void)
{ *e ;
  char s[] = "rubbish";

  e = os_cli(s);
  if (e)
  {no : %d\n", e->errnum);
    printf("Err msg : %s\n", e->errmsg);
  }
  else
    printf("No error\n");
}
```

Name	os_adval																
Syntax	<code>int os_adval(int a);</code>																
Description	Return the current x or y position of the mouse, or the number of bytes in a particular input buffer, or the number of bytes free in a particular output buffer.																
Returns	The integer returned depends on the value of the parameter a, as follows: <table> <tr><td>a</td><td>action</td></tr> <tr><td>7</td><td>x coordinate of mouse</td></tr> <tr><td>8</td><td>y coordinate of mouse</td></tr> <tr><td>246</td><td>bytes in mouse buffer</td></tr> <tr><td>252</td><td>bytes free in printer buffer</td></tr> <tr><td>253</td><td>bytes free in RS423 output buffer</td></tr> <tr><td>254</td><td>bytes in RS423 input buffer</td></tr> <tr><td>255</td><td>bytes in keyboard buffer</td></tr> </table>	a	action	7	x coordinate of mouse	8	y coordinate of mouse	246	bytes in mouse buffer	252	bytes free in printer buffer	253	bytes free in RS423 output buffer	254	bytes in RS423 input buffer	255	bytes in keyboard buffer
a	action																
7	x coordinate of mouse																
8	y coordinate of mouse																
246	bytes in mouse buffer																
252	bytes free in printer buffer																
253	bytes free in RS423 output buffer																
254	bytes in RS423 input buffer																
255	bytes in keyboard buffer																

Example

```
/* Example of os_adval() */
/* Use os_adval() to print x and y coordinates of
   the mouse pointer. Press Esc to quit program */
#include <stdio.h>
#include "roslib.h"

void main(void)
{ x , y;
  while(1)
  { = os_adval(7);
    y = os_adval(8);
    printf("x=%d y=%d\n", x, y);
  }
}
```

```
}
```

Name **os_args**

Syntax `os_error *os_args(regset *r);`

Description Perform an OS_Args SWI call to read or write an open file's arguments, or to return the filing system type. The parameter *r* points to a structure of type `regset`, which contains an array of integers used to pass registers to OS_Args.

```
typedef struct  
{r[10] ;  
} regset;
```

Returns A pointer to an error if one has occurred, or zero if the operation was successful.

Example

```
/* Example of os_args() */  
/* Create file "temp" in current directory, set  
   set it's extent to 64, and close the file. */  
#include "roslib.h"  
void main(void)  
{r ;  
  int h;  
  
  r.r[0] = 0x83;  
  r.r[1] = "temp";  
  os_find(&r);  
  h = r.r[0];  
  
  r.r[0] = 3;  
  r.r[1] = h;  
  r.r[2] = 64;  
  os_args(&r);  
  
  r.r[0] = 0;  
  r.r[1] = h;  
  os_find(&r);  
}
```

Name **os_byte**

Syntax `os_error *os_byte(int a, int *x, int *y);`

Description Perform an OS_Byte SWI call. Parameter *a* is the reason code which determines the action of the call. Parameters *x* and *y* are passed to registers R0 and R1. Please refer to your computer user guides for a list OS_Byte reason codes and parameters.

Returns A pointer to an error if one has occurred, or zero if the operation was successful.

Example

```
/* Example of os_byte() */  
/* Read character at text cursor position,  
   and current screen mode using OS_Byte 135 */  
#include <stdio.h>  
#include "roslib.h"  
  
void main(void)  
{x , y;  
  os_byte(135, &x, &y);  
  printf("Character   = %d\n", x);  
  printf("Screen mode = %d\n", y);  
}
```

Name **os_circle**

Syntax `os_error *os_circle(int x, int y, int r);`

Description Draw a circle whose centre is at the coordinates specified by *x* and *y*, with a radius specified by *r*.

Returns A pointer to an error if one has occurred, or zero if the operation was successful.

Example

```
/* Example of os_circle() */  
/* Change to screen mode 12 and draw circle  
   of radius 300. */  
#include "roslib.h"
```

```
void main(void)
{x , y, r;

    os_mode(12);
    x = 500;
    y = 500;
    r = 300;
    os_circle(x, y, r);
}
```

Name **os_circlefill**

Syntax **os_error *os_circlefill(int x, int y, int r);**

Description Draw a solid circle whose centre is at the coordinates specified by x and y, with a radius specified by r.

Returns A pointer to an error if one has occurred, or zero if the operation was successful.

Example

```
/* Example of os_circlefill() */
/* Change to screen mode 12 and draw solid
   circle of radius 300. */
```

```
#include "roslib.h"
```

```
void main(void)
{x , y, r;

    os_mode(12);
    x = 500;
    y = 500;
    r = 300;
    os_circlefill(x, y, r);
}
```

Name **os_clg**

Syntax **os_error *os_clg(void);**

Description Clear the graphics window to the current graphics background colour.

Returns Void.

Example

```
/* Example of os_clg() */
/* Set up a graphics window (using os_vduq) then
   clear it to a specified background colour. */
```

```
#include "roslib.h"
```

```
void main(void)
{os_mode(12) ;
  os_vduq(24, 0, 1, 0, 1, 0, 2, 0, 2);
  os_gcol(0, 130);
  os_clg();
}
```

Name **os_cli**

Syntax **os_error *os_cli(char *s);**

Description Pass the string pointed to by parameter s to the command line interpreter to execute as a command. Please refer to your computer user guides for a list of commands available.

Returns A pointer to an error if one has occurred, or zero if the operation was successful.

Example

```
/* Example of os_cli() */
/* Execute the command '*Help'. */
```

```
#include "roslib.h"
```

```
void main(void)
{s[] = "Help";
  os_cli(s);
}
```

```
}
```

Name **os_cls**
Syntax **os_error *os_cls(void);**
Description Clear the text window to the text background colour, and moves the text cursor to its home position.
Returns Void
Example

```
/* Example of os_cls() */  
/* Set up a text window (using os_vduq) then  
   clear to a specified background colour. */  
#include "roslib.h"  
void main(void)  
{os_mode(12) ;  
  os_vduq(28, 5, 20, 45, 5);  
  os_colour(132);  
  os_cls();  
}
```

Name **os_colour**
Syntax **os_error *os_colour(int c);**
Description Set the text colour to the value c. If c is in the range 0 to 127, the text foreground colour is set. Adding 128 to this (i.e. 128 to 255), sets the text background colour.
Returns A pointer to an error if one has occurred, or zero if the operation was successful.
Example

```
/* Example of os_colour() */  
/* Display text in all the colours available  
   in a 256-colour screen mode (mode 15). */  
#include <stdio.h>  
#include "roslib.h"  
  
void main(void)  
{i ;  
  os_mode(15);  
  for(i = 0; i <= 63; i++)  
  {os_colour(i) ;  
   printf("Colour %d\n", i);  
  }  
}
```

Name **os_cursor**
Syntax **os_error *os_cursor(int n);**
Description Control the appearance of the cursor according to the value of n:

<i>n</i>	<i>meaning</i>
0	stops the cursor appearing
1	makes the cursor re-appear
2	steady cursor
3	flashing cursor

Returns A pointer to an error if one has occurred, or zero if the operation was successful.
Example

```
/* Example of os_cursor() */  
/* Step through the four cursor types. */  
  
#include <stdio.h>  
#include "roslib.h"  
  
void main(void)  
{i ;  
  
  for(i = 0; i <= 3; i++)  
  {os_cursor(i) ;  
   printf("Cursor %d\n", i);  
   printf("Press any key to continue.\n");  
   os_get();  
  }
```

```
    }  
}
```

Name **os_draw**
Syntax `os_error *os_draw(int x, int y);`
Description Draw a line in the current graphics foreground colour to the absolute coordinates specified by x and y.
Returns A pointer to an error if one has occurred, or zero if the operation was successful.
Example

```
/* Example of os_draw() */  
/* Draw a series of steps using os_draw(). */  
  
#include "roslib.h"  
  
void main(void)  
{ x , y;  
  
    os_mode(12);  
    x = y = 0;  
    while(x <= 1280)  
    { + = 40, y);  
      os_draw(x, y += 32);  
    }  
}
```

Name **os_drawby**
Syntax `os_error *os_drawby(int x, int y);`
Description Draw a line in the current graphics foreground colour to the position specified by the offsets x and y, relative to the current position.
Returns A pointer to an error if one has occurred, or zero if the operation was successful.
Example

```
/* Example of os_drawby() */  
/* Draw series of steps using os_drawby(). */  
  
#include "roslib.h"  
  
void main(void)  
{ i ;  
  
    os_mode(12);  
    for(i = 1; i <= 32; i++)  
    { os_drawby(40 , 0);  
      os_drawby(0, 32);  
    }  
}
```

Name **os_file**
Syntax `os_error *os_file(filestr *p);`
Description Perform an OS_File SWI call, which performs actions on whole files. The parameter p points to a structure of type filestr, which passes various arguments to OS_File.

```
typedef struct  
{ action ;  
  char *objname;  
  int loadaddr;  
  int execaddr;  
  int startaddr;  
  int endaddr;  
  int reserved[4];  
} filestr;
```

Returns A pointer to an error if one has occurred, or zero if the operation was successful.
Example

```
/* Example of os_file() */  
/* Create "temp" in current directory and read  
   it's catalogue information using OS_File 17. */  
#include <stdio.h>
```

```
#include "roslib.h"

void main(void)
{p ;

    os_cli("Create temp 100");

    p.action = 17;
    p.objname = "temp";
    os_file(&p);
    printf("Object type    = %d\n", p.action);
    printf("Object name    = %s\n", p.objname);
    printf("Load address   = %X\n", p.loadaddr);
    printf("Exec address   = %X\n", p.execaddr);
    printf("Obj length    = %d\n", p.startaddr);
    printf("Attributes    = %d\n", p.endaddr);
}
```

Name **os_fill**

Syntax **os_error *os_fill(int x, int y);**

Description Flood-fill an area in the current foreground colour, commencing at the coordinates specified by the parameters x and y.

Returns A pointer to an error if one has occurred, or zero if the operation was successful.

Example

```
/* Example of os_fill() */
/* Draw a series of steps using os_drawby(), then
   fill above with red and below with green. */
#include "roslib.h"

void main(void)
{i ;

    os_mode(12);
    for(i = 1; i <= 32; i++)
    {os_drawby(40 , 0);
      os_drawby(0, 32);
    }
    os_gcol(0, 1);
    os_fill(200, 500);
    os_gcol(0, 2);
    os_fill(500, 200);
}
```

Name **os_find**

Syntax **os_error *os_find(regset *r);**

Description Perform an OS_Find SWI call to open and close files. r points to a structure of type regset, which contains an array of integers used to pass registers to OS_Find.

```
typedef struct
{r[10] ;
} regset;
```

Returns A pointer to an error if one has occurred, or zero if the operation was successful.

Example

```
/* Example of os_find() */
/* Use OS_Find 0x83 to create file "temp" in
   current directory. Use OS_Find 0 to close it */

#include "roslib.h"

void main(void)
{r ;
  int h;

  r.r[0] = 0x83;
  r.r[1] = "temp";
  os_find(&r);
  h = r.r[0]; /* file handle in r.r[0] */
}
```

```

    r.r[0] = 0;
    r.r[1] = h;
    os_find(&r);
}

```

Name **os_gbp**

Syntax `os_error *os_gbp(os_gbpstr *p);`

Description Perform OS_GBP SWI call to read or write a group of bytes from or to an open file. The parameter p points to a structure of type gbpstr, which passes various arguments to OS_GBP.

```

typedef struct
{
    action ;
    int handle;
    void *startaddr;
    int number;
    int pointer;
    int buf_len;
    char *wildcard;
    int reserved[3];
} gbpstr;

```

Returns A pointer to an error if one has occurred, or zero if the operation was successful.

Example

```

/* Example of os_gbp() */
/* Create file "temp" in current directory, write
   write characters A to Z and close file. */

#include "roslib.h"

void main(void)
{
    r ;
    os_gbpstr p;
    int h;

    r.r[0] = 0x83;
    r.r[1] = "temp";
    os_find(&r);
    h = r.r[0];
    p.action = 2;
    p.handle = h;
    p.startaddr = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
    p.number = 26;
    os_gbp(&p);
    r.r[0] = 0;
    r.r[1] = h;
    os_find(&r);
}

```

Name **os_gcol**

Syntax `os_error *os_gcol(int a, int c);`

Description Set the graphics plot colour and action. If c is in the range 0 to 127, the graphics foreground colour is set. Adding 128 to this (i.e. 128 to 255), sets the graphics background colour. The graphics plotting action is specified by a as follows:

<i>a</i>	<i>action</i>
0	plot colour on screen with c
1	OR colour on screen with c
2	AND colour on screen with c
3	EOR colour on screen with c
4	Invert colour on screen
5	Leave screen unchanged
6	AND colour on screen with inverse of c
7	OR colour on screen with inverse of c

If 16, 32, 48 or 64 is added to a, the first, second, third or fourth colour pattern respectively is used instead of c. Adding 80 causes all four patterns to be used, placed side-by-side.

Returns A pointer to an error if one has occurred, or zero if the operation was successful. See page 124 for further details.

Example

```
/* Example of os_gcol() */
/* Plot a series of solid circles in different
   colours. */
#include "roslib.h"

void main(void)
{
    x ;
    os_mode(12);
    for(x = 0; x <= 1080; x += 180)
    {
        os_gcol(0 , x/180 + 1);
        os_circlefill(x + 100, 500, 85);
    }
}
```

Name **os_get**

Syntax int os_get(void);

Description Return a character from the input stream. The function does not return until a character is available, thus halting the program temporarily.

Returns The ASCII code for the character in the input stream, or the value 0x1xx if an escape condition exists.

Example

```
/* Example of os_get() */

/* Get character typed at keyboard and print
   ASCII code. Repeat until Esc pressed. */
#include <stdio.h>
#include "roslib.h"

void main(void)
{
    c ;
    printf("Type characters now\n");
    while((c = os_get()) != -27)
        printf("%d\n", c);
}
```

Name **os_inkey**

Syntax int os_inkey(int n);

Description Return a character from the input stream. If n is a positive integer, it represents the time limit in centi-seconds. If n is a negative integer in the range -255 to -1, it represents the negative INKEY code of the key being interrogated. Please refer to your computer user guide for a list of negative INKEY values.

Returns If n is positive, it returns the ASCII code for the character in the input stream, or -1 if there is no character within the time limit.

If n is negative, it returns -1 if the interrogated key is being pressed, or 0 if not.

If n is -256, it returns the version of the operating system in use.

Example

```
/* Example of os_inkey() */
/* Wait up to 5 seconds for a key to be pressed.
   If a key is pressed within the time limit print
   it's ASCII value, else print "Timeout!". */

#include <stdio.h>
#include "roslib.h"

void main(void)
{
    k ;

    printf("Press a key:\n");
    k = os_inkey(500);
    if(k == -1)
        printf("Timeout!\n");
    else
```

```
        printf("Key %d pressed.\n", k);
    }
```

Name **os_mode**

Syntax **os_error *os_mode(int m);**

Description Set the screen display to mode m.

Returns A pointer to an error if one has occurred, or zero if the operation was successful. See page 124 for further details.

Example

```
/* Example of os_mode() */
/* Change to mode 15 and draw circle. */

#include "roslib.h"

void main(void)
{os_mode(15) ;
  os_circle(640, 512, 250);
}
```

Name **os_move**

Syntax **os_error *os_move(int x, int y);**

Description Move the graphics cursor to the absolute coordinates specified by x and y.

Returns A pointer to an error if one has occurred, or zero if the operation was successful.

Example

```
/* Example of os_move() */
/* Draw a series of vertical lines using os_move()
   to position graphics cursor for each new line */

#include "roslib.h"

void main(void)
{x ;

  os_mode(12);
  for(x = 0; x < 1280; x += 32)
  {os_move(x , 0);
   os_draw(x, 1023);
  }
}
```

Name **os_moveby**

Syntax **os_error *os_moveby(int x, int y);**

Description Move the graphics cursor to the position specified by the offsets x and y, relative to the current position.

Returns A pointer to an error if one has occurred, or zero if the operation was successful.

Example

```
/* Example of os_moveby() */
/* Draw five rectangles using os_moveby() to
   shift the graphics cursor after each. */
#include "roslib.h"

void main(void)
{x ;

  os_mode(12);
  for(x = 1; x <= 5; x++)
  {os_drawby(0 , 100);
   os_drawby(100,0);
   os_drawby(0, -100);
   os_drawby(-100,0);

   os_moveby(150, 0);
  }
}
```

Name **os_origin**

Syntax	<code>os_error *os_origin(int x, int y);</code>
Description	Set the graphics origin to the position specified by the absolute the offsets x and y. The graphics origin is the screen position of point (0, 0), and is used for all subsequent graphics output.
Returns	A pointer to an error if one has occurred, or zero if the operation was successful.
Example	

```
/* Example of os_origin() */
/* Draw circle in centre of screen by setting
   os_origin() to the screen centre, then
   drawing circle at coordinates 0, 0. */

#include "roslib.h"

void main(void)
{os_mode(12) ;
  os_origin(639, 511);
  os_circle(0, 0, 100);
}
```

Name	os_palette
Syntax	<code>os_error *os_palette(int l, int m, int r, int g, int b);</code>
Description	Define the colour palette, where l is the logical colour to set, m is the physical colour or mode, and r, g and b represent the red, green and blue colour information.
<i>m</i>	<i>action</i>
0-15	logical colour l = physical colour m (r, g, and b are ignored)
16	logical colour l = physical colour specified by r, g, b
17	define first flash colour for l
18	define second flash colour for l
24	border colour = physical colour specified by r, g, b
25	mouse logical colour l = physical colour specified by r, g, b
Returns	A pointer to an error if one has occurred, or zero if the operation was successful. See page 124 for further details.

Example

```
/* Example of os_palette() */
/* Plot a solid circle in colour l then change
   palette for colour l to specified RGB value. */

#include <stdio.h>
#include "roslib.h"

void main(void)
{red , green, blue;

  os_mode(12);
  os_gcol(0,1);
  os_circlefill(639, 511, 200);
  printf("Press a key to change palette\n");
  os_get();
  red = 120;
  green = 128;
  blue = 186;
  os_palette(1, 16, red, green, blue);
}
```

Name	os_plot
Syntax	<code>os_error *os_plot(int p, int x, int y);</code>
Description	Perform an operating system plot function. Parameters x and y are the screen coordinates, and p is the plot number. Please refer to your computer user guide for a list of plot numbers.
Returns	A pointer to an error if one has occurred, or zero if the operation was successful.
Example	

```
/* Example of os_plot() */
/* Change to screen mode 12 and plot a circle
   segment using os_plot() with plot number 173 */

#include "roslib.h"

void main(void)
{os_mode(12) ;
  os_move(750, 0);
  os_plot(173, 0, 750);
}
```

Name	os_point
Syntax	int os_point(int x, int y);
Description	Return the logical colour of the pixel at the coordinates specified by x and y.
Returns	An integer in the range 0 to 63 representing the logical colour, or -1 if the specified point lies outside the current graphics window.

Example

```
/* Example of os_point() */
/* Change to screen mode 12, draw solid circle and
   use os_point() to return the colour. */

#include <stdio.h>
#include "roslib.h"

void main(void)
{os_mode(12) ;
  os_gcol(0, 3);
  os_circlefill(500, 500, 300);
  printf("Colr = %d\n", os_point(500, 500));
}
```

Name	os_pos
Syntax	int os_pos(void);
Description	Return the X coordinate of the text cursor.
Returns	An integer between 0 and the width of the current text window minus one.

Example

```
/* Example of os_pos() */
/* Print some text then return X coordinate
   of the text cursor. */

#include <stdio.h>
#include "roslib.h"

void main(void)
{x ;
  printf("Sample text.");
  x = os_pos();
  printf("\nX coordinate = %d\n", x);
}
```

Name	os_rectangle
Syntax	os_error *os_rectangle(int x, int y, int w, int h);
Description	Draw the outline of a rectangle at the coordinates specified by x and y, with a width and height specified by w and h.
Returns	A pointer to an error if one has occurred, or zero if the operation was successful.
Example	

```
/* Example of os_rectangle() */
/* Change to screen mode 12 and draw rectangle. */

#include "roslib.h"

void main(void)
{x , y, w, h;

    os_mode(12);
    x = 300;
    y = 300;
    w = 600;
    h = 400;
    os_rectangle(x, y, w, h);
}
```

Name	os_rectanglefill
Syntax	os_error *os_rectanglefill(int x, int y, int w, int h);
Description	Plot a solid rectangle at the coordinates specified by x and y, with a width and height specified by w and h.
Returns	A pointer to an error if one has occurred, or zero if the operation was successful.
Example	

```
/* Example of os_rectanglefill() */
/* Change to mode 12 and draw a solid rectangle */
#include "roslib.h"

void main(void)
{x , y, w, h;

    os_mode(12);
    x = 300;
    y = 300;
    w = 600;
    h = 400;
    os_rectanglefill(x, y, w, h);
}
```

Name	os_rnd
Syntax	unsigned os_rnd(int n);
Description	Return a random number, or re-seeds the random number generator.
Returns	If n is positive, it returns a random integer number between 0 and n-1. If n is negative, it re-seeds the random number generator and returns 0.
Example	

```
/* Example of os_rnd() */
/* Print 100 random numbers between 1 and 100. */
#include <stdio.h>
#include "roslib.h"
void main(void)
{ i ;
  for(i = 1; i <= 100; i++)
    printf("%d\n", os_rnd(100) + 1);
}
```

Name **os_sound**
Syntax `os_error *os_sound(int c, int a, int p, int d);`
Description Generate a sound. The parameters c, a, p, and d represent the channel, amplitude, pitch and duration.
Returns A pointer to an error if one has occurred, or zero if the operation was successful.
Example

```
/* Example of os_sound() */
/* Make a sound. */
#include "roslib.h"
void main(void)
{os_sound(1 , -5, 100, 100);
}
```

Name **os_swi**
Syntax `void os_swi(int n, regset *r);`
Description Performs the SWI call specified by n. The parameter r points to a structure of type regset, which contains an array of integers used to pass registers to the SWI routine. If an error occurs, an operating system error is raised. Please refer to your Programmers Reference Manual for a complete list of SWI calls available.

```
typedef struct
{r[10] ;
} regset;
```

Returns Void.

Example

```
/* Example of os_swi() */
/* Call OS_ConvertHex2 to convert a decimal number
to a 2-byte hex string. This example uses
os_swi(), so if an error occurs it is raised by
the operating system automatically. */
#include <stdio.h>
#include "roslib.h"
#include "swis.h"

void main(void)
{r ;
 int d;
 char s[3];

 d = 255;
 r.r[0] = d;
 r.r[1] = s;
 r.r[2] = 3;
 os_swi(OS_ConvertHex2, &r);
 printf("%d in hex = %s", d, r.r[0]);
}
```

Name **os_swix****Syntax** `os_error *os_swix(int n, regset *r);`**Description** Performs the X form of the SWI call specified by the parameter n. The parameter r points to a structure of type regset, which contains an array of integers used to pass registers to the SWI routine. If an error occurs, a pointer to the error is returned; an operating system error is not raised. Please refer to the Programmers Reference Manual for a complete list of SWI calls available.

```
typedef struct
{r[10] ;
} regset;
```

Returns A pointer to an error if one has occurred, or zero if the operation was successful.**Example**

```
/* Example of os_swix() */
/* Call OS_ConvertHex4 to convert a decimal number
   to a 4-byte hex string. This example uses
   os_swix(), so if an error occurs it may be
   trapped and reported by this program. */
#include <stdio.h>
#include "roslib.h"
#include "swis.h"

void main(void)
{r ;
  os_error *e;
  int d;
  char s[5];

  d = 65535;
  r.r[0] = d;
  r.r[1] = s;
  r.r[2] = 5;
  e = os_swix(OS_ConvertHex4, &r);
  if(e)
    printf("E%d%s\n", e->errnum, e->errmsg);
  else
    printf("%d converted to hex = %s", d, r.r[0]);
}
```

Name **os_tab****Syntax** `os_error *os_tab(int x, int y);`**Description** Position the text cursor at the coordinates specified by x and y. both coordinates must lie within the current text window, otherwise no cursor movement will take place.**Returns** A pointer to an error if one has occurred, or zero if the operation was successful. See page 124 for further details.**Example**

```
/* Example of os_tab() */
/* Print the horizontal tab position at every
   8 characters across the screen. */

#include <stdio.h>
#include "roslib.h"

void main(void)
{x ;

  os_mode(12);
  for(x = 0; x <= 72; x += 8)
  {os_tab(x , 0);
    printf("%d", x);
  }
}
```

Name	os_tint																				
Syntax	<code>os_error *os_tint(int c, int t);</code>																				
Description	Set the tint for a colour. The parameter <i>c</i> specifies which colour is set, as follows: <table><tr><td><i>c</i></td><td><i>colour</i></td></tr><tr><td>0</td><td>text foreground colour</td></tr><tr><td>1</td><td>text background colour</td></tr><tr><td>2</td><td>graphics foreground colour</td></tr><tr><td>3</td><td>graphics background colour</td></tr></table> The parameter <i>t</i> specifies the level of tint required, as follows: <table><tr><td><i>t</i></td><td><i>tint</i></td></tr><tr><td>0x00</td><td>darkest</td></tr><tr><td>0x40</td><td>dark</td></tr><tr><td>0x80</td><td>light</td></tr><tr><td>0xC0</td><td>lightest</td></tr></table>	<i>c</i>	<i>colour</i>	0	text foreground colour	1	text background colour	2	graphics foreground colour	3	graphics background colour	<i>t</i>	<i>tint</i>	0x00	darkest	0x40	dark	0x80	light	0xC0	lightest
<i>c</i>	<i>colour</i>																				
0	text foreground colour																				
1	text background colour																				
2	graphics foreground colour																				
3	graphics background colour																				
<i>t</i>	<i>tint</i>																				
0x00	darkest																				
0x40	dark																				
0x80	light																				
0xC0	lightest																				
Returns	a pointer to an error if one has occurred, or zero if the operation was successful.																				
Example	<pre>/* Example of os_tint() */ /* Plot a solid circle in 4 tints. */ #include "roslib.h" void main(void) {t ; os_mode(15); os_gcol(0, 25); for(t = 0; t <= 3; t++) {os_tint(2 , t); os_circlefill(500, 500, 200 - t * 50); } }</pre>																				

Name **os_vpos**
Syntax `int os_vpos(void);`
Description `Return the Y coordinate of the text cursor.`
Returns `An integer between 0 and the depth of the current text window minus one.`
Example

```
/* Example of os_vpos() */
/* Print some lines of text then return Y
   coordinate of the text cursor. */
#include <stdio.h>
#include "roslib.h"

void main(void)
{ i , y;

  for(i = 0; i <= 8; i++)
    printf("Line %d\n", i);
  y = os_vpos();
  printf("\nY coordinate = %d\n", y);
}
```

Name **os_vdu**
Syntax `os_error *os_vdu(int c);`
Description `Send the single byte c to the VDU drivers. Please refer to your computer user guide for a list of VDU commands.`
Returns `A pointer to an error if one has occurred, or zero if the operation was successful.`
Example

```
/* Example of os_vdu() */
/* Plot a circle segment using the os_vdu()
   function to send the single bytes to the vdu
   drivers. See also os_vduq() and os_vduw(). */

#include "roslib.h"

void main(void)
{os_mode(12) ;
  os_move(750, 0);
  os_vdu(25);
  os_vdu(173);
  os_vdu(0);
  os_vdu(0);
  os_vdu(238);
  os_vdu(2);
}
```

Name	os_vduq
Syntax	os_error *os_vduq(int c, ...);
Description	Send multiple bytes to the VDU drivers. The number of bytes sent is appropriate to the the VDU command c, and the function assumes the correct number of parameters have been passed. Please refer to your computer user guide for a list of VDU commands, and the number of parameters for each command.
Returns	A pointer to an error if one has occurred, or zero if the operation was successful. See page 124 for further details.

Example

```
/* Example of os_vduq() */
/* Plot a circle segment using the os_vduq()
   function to send multiple bytes to the vdu
   drivers. See also os_vdu() and os_vduw(). */

#include "roslib.h"

void main(void)
{os_mode(12) ;
  os_move(750, 0);
  os_vduq(25, 173, 0, 0, 238, 2);
}
```

Here are just a few of the other products you will be able to
download from RISCOS.com free of charge
(You can make a donation to help support future product releases but this is entirely voluntary)



This Documentation is © 3QD Developments Ltd 2026
Some of the products shown above might not be available immediately.
Not all products will run on all versions of RISC OS, check the product information page for details.

Other Products list last updated 29/2/26