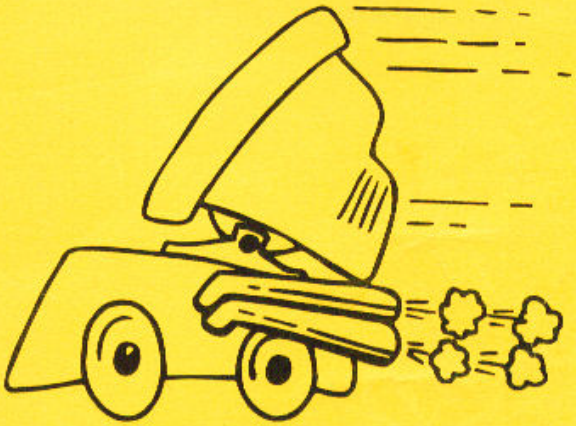
# SAM C



The professional programming language for the SAM.

FRED PUBLISHING

For SAM 512K

# SAM C

**User Manual**
**April 1995**

## CONTENTS

While every care has been taken in the production of this manual and the software it describes neither the author nor the publisher will be held liable for any mistakes or omissions nor be responsible or held liable for any damages that may arise from the purchasor's use of, or inability to use, this software and manual. No representations are made as to the suitability of the software for a particular purpose or for its use on any particular hardware

# SAM C

## Preface

C is one of the most powerful, yet usable, programming languages available for any computer. Combined with the portability that allows you to run programs on your SAM that were written on (and for) dozens of other machines (PC, Amiga, Atari, Unix ...) you have a very resourceful tool now running on your SAM. I hope this manual helps you to make the most of it, and that the development of C will help SAM programmers become even more prolific and proficient.

We would be very interested to see what you can do with SAM C - if you create a program, routine or library, please send it in whether it be in regard to commercial publication, inclusion on an issue of FRED so others can learn from your code, or simply to show us what SAM C is capable of!

We regret that we cannot take telephone questions relating to any problems you have when programming in SAM C, at this point no-one else is available for advice on the phone either. If you have a problem, send it to FRED with a printout of the program (and preferably a copy on disc) and a SAE. We will try to answer your query as quickly as possible but cannot make any guarantee as to how long this could take, or to whether we are actually able to solve the problem (we're just the publishers, you programmers are supposed to be the clever ones!). Unless you state otherwise we may use your problem / code in an issue of FRED magazine to demonstrate the solution to others.

Any product developed with the aid of SAM C by a purchasor of SAM C may be published independantly provided that it carries a prominent message stating that SAM C was used in its development.

The C compiler, libraries included, example programs and all related code are copyright 1995 FRED Publishing. Marian has worked extremely hard over the last few years to develop SAM C, please don't rip him off - let your friends buy their own copy.

SAM C and standard libraries by Marian Krivos.
Manual by Marian Krivos, Adrian Parker and Colin Macdonald.
Manual typesetting by FRED Publishing.

Those responsible for testing until all hours in the morning are : Bob Brenchley, Carol Brooksbank, Colin Macdonald, Adrian Parker, Chris White and Nev Young.

**SAM C Software and Manual © 1995 FRED Publishing. All Rights Reserved.**

## Introduction

Welcome to the world of 'SAM C'. You now have the facility to produce utilities, games, business software etc on your SAM in the most flexible, professional language available on any computer system today. C (and SAM C) gives you the flexibility (and much of the speed) of machine language, whilst allowing you the convenience of a high level language. However, before we get into the complexities of this program, let's try to get an idea of how it all started.

Way back in the mists of time (well, 1978 to be exact), when computers that could only do one hundredth of the capabilities of the one in front of you took up a room larger than the one you're sitting in, two system programmers called Brian Kemighan and Dennis Ritchie produced a new programming language. The language was based on an older language called BCPL (Basic Combined Programming Language), which had previously had a large influence on a language called 'B'. So, logically, the new language was called 'C'. The language was compiled - ie the program was converted, once only, to machine language BEFORE the program was run. The actual program that is written is known as the 'source file' and the final executable code as the 'object file'. Many versions of the language were produced on many computer systems and this led to other systems being produced. An interpreted version of C was produced in the early 80's, closely followed by a version that converted the high level C language to assembler mnemonics, ready to be assembled. Most versions of C will also assemble these mnemonics into ready-to-run machine code - as does SAM C!

Some of you might have heard of the C++ language - this is an object-orientated programming (OOP) version of C. SAM C is not object-orientated - it is a version of the conventional C, which is a programming style familiar to you all.

So now that we know where we are, the rest of this manual is devoted to explaining the different elements of this program that allow you to produce your own software.

### Getting Started

For your convenience, the disc enclosed has riot been protected. Because C will need to read files from this disc as well as save to it, it is very important that you make at least one working copy of the disc. You can save MasterDOS (you may encounter problems if you use MasterDOS and MasterBASIC - just use MasterDOS on its own) over the SAMDOS on your working copy to give you improved performance (particularly if you also have the 1 Megabyte memory expansion). You are now free to adjust the default settings files and save your source files to the disc without fear of losing any data. We would suggest producing at least one working copy for each type of programming you undergo (ie one disc for games, one for utilities, one for additional libraries etc).

If your master disc does become corrupted, send it along with £1 to cover costs and we will send you a replacement.

Any amendments to this manual will appear on the disc with the filename README.

# First Steps In C

I'll be honest. C didn't become one of the most popular programming languages in the world by being simple. And considering there are hundreds of books on how to learn C, it's a bit much to expect a complete beginner-to-expert guide here. What I will do is explain the very basics, and more importantly, provide you with the information about how to learn the rest. If you want to learn C, you must get access to a book on the subject - your local library is bound to have at least a couple, although from experience the following are recommended :

C : The Complete Reference by Herbert Schildt. Published by Osbourne McGraw Hill.
Teach yourself C by Herbert Schildt. Published by Osbourne McGraw Hill.
The C programming language by B.Kernighan and D.Ritchie. Published by Prentice Hall.
C for Dummies. Published by IDG Books.

If you can't find any of the books in your local shops / library, a reliable mail order book company is Derwent Bookshop, who can be reached on 01900-62503. They currently stock Teach yourself C at £ 19.99.

You should find it worthwhile reading the following few pages about some of the basics of C whether or not you intend to get a book on the subject.

C is a relatively simple language, it contains very few commands (only about two dozen in SAM C!) it doesn't even allow you to input anything from the keyboard or output anything to the screen. The power of C comes with the ability to make use of libraries. Fortunately, most C compilers these days come with a variety of standard libraries which allow other features like input and output. SAM C is no exception. Although using libraries themselves is very simple, it does take time to learn about the functions in a library - but this is something you can do as and when you need the extra functions.

I'll assume everyone is familiar with SAM BASIC and you should find you can relate a lot of the logic and methods in C to it.

Unlike BASIC, lines do not have such importance in C - a particularly long line can extend over several lines in the editor. For this reason, all instructions are separated by the semicolon (;).

All programs have a simple structure:

```
main()
{
    Declarations
    Instructions
}
```

It's not as tricky as it appears. C programs are made up of functions (sub-routines, like Procedures in BASIC), and the starting function is called main(). This is automatically called when you run the program. The parenthesis 0 after main() usually contain a list of parameters to be passed to the function, however, you normally won't be passing any to main().

The curly braces { } (press Symbol-F and G) are used to show the start and end of a compound statement block (see page 31) - in this case the block is a function.

All variables have to be declared before use. eg :

**int a,b;**

declares a and b to be integers. An integer is a whole number between -32768 and +32767 inclusive. For larger numbers you can use an unsigned integer which can hold whole numbers from 0 to 65535. The other variable type you will also be using a lot is char, which is simply a character. Because of the flexibility of C, the different variable types are fairly inter-changable. For example, you could define an integer, assign it a value, then simply print out it's ASCII equivalent as if it was a character!

Assignment is done with =, as in BASIC, but without the LET :-

**a=1;**

Sets a to 1.

The if statement is :-

if (condition) {action} else {other action}

Compared to BASIC which is :-

IF condition THEN action ELSE other action

C EXAMPLE - IF

**if (a==1) {b=3;} else {b=4;}**

Note that the else (and it's associated statement block) could be left out (as in BASIC) and that while the assignment operator is =, the equality operator is ==. Other comparison operators are != for not equal to, < and > for less than and greater than, <= and >= for less than or equal and greater than or equal.

Logical operators are :

| | | | |
|---|---|---|---|
| **BASIC** | **AND** | **OR** | **NOT** |
| **C** | **&&** | \|\| | **!** |

( the | symbol is produced by pressing Symbol-9 )

So, the BASIC line **IF a=3 AND b>4 THEN LET x=7**

in C, becomes **if (a==3 && b>4) {x=7;}**

In actual fact, if there is just one instruction as a result of the if (as in the above example) then the curly braces are not required.

The for...next loop is :-

for (Start condition ; End condition ; Increment) { actions }

where any number of instructions can be between the curly braces

The BASIC equivalent is:-

```
FOR x=Start TO End STEP increment
actions
NEXT x
```

C EXAMPLE - FOR

```
int n;
for (n=1 ; n<=10 ; n++) { printf ("hello"); }
```

Forget about n++ (it just increments n by one) and printf for the moment - the above loop will print hello on screen ten times. Because the condition you set does not have to be related to the start condition, you can easily do an infinite loop :

**for (n=1; ; n++) { } or even for( ; ; ) {}** Because the end conditions equate to 0

The While...Do loop is :-

```
while (condition) { actions }
```

The BASIC equivalent is :-

```
DO WHILE (condition)
actions
LOOP
```

C EXAMPLE - WHILE

```
while (7>6) { }
```

is, of course, another infinite loop because 7 will always be greater than 6.

The equivalent of the Repeat...Until loop also makes use of the while keyword :-

```
do { actions } while (condition);
```

So, the infinite loop above could also be done by :-

```
do {} while (7>6);
```

For something as simple as this, the outcome is identical, but the do. whil loop performs the actions first and then checks to see if it should exit. The normal while loop checks first and therefore need not do any iterations of the actions.

Creating your own functions is very easy - it's the same as writing the main() function.

ie :-

```
 main()
}
```

```
    int a,b,c;
    if (a==b && a!=6 && a>3) a=99;
    if (a==c && a!=6 && a>3) a=95;
}
```

This assigns a to 99 providing it was equal to b, wasn't 6 and was greater than 3. If a was then the same as c, not 6 and greater than 3 then a becomes 95.

If we wanted to use that small testing routine in other places, rather than typing it out several times we can create a function :-

```
int TESTTWO (int p, int q)
{
    if (p==q && p!=6 && p>3) { return 1; }
     else { return 0; }

main()
{
    int a,b,c
    if (TESTTWO(a,b)==1) a=99;
    if (TESTTWO(b,c)==1) a=95;
}
```

Which will produce exactly the same results except that the function TESTTWO is called twice. You will have noticed the way that parameters are passed to functions - in the calling statement they're simply put in parenthesis after the function name (the parenthesis must be there whether or not you're passing any parameters) and they're declared in a similar fashion in the functions itself - remembering of course to tell the compiler what type they are.

Although variables cannot be passed back in the same way they are passed to the function, by specifying the type of variable to be returned before the function name (use void if you don't want it to return any value. If no type is declared, the default is int) and then using return to select which variable you want returned, you can use functions just like functions in BASIC. Passing more than one value back is more complicated, and I suggest you look up 'pointers' in a C book.

The all-important comments can be placed inside the source using // to comment off just until the end of that line, alternatively, /* will start a comment and */ will end it - see page 27 for examples.

The usual + - * / are addition, subtraction, multiplication and division as expected. Although C has a few usual shorthand methods :-

instead of using a=a+1; you can use a++; which has the same effect. This can be used in the middle of calculations as well, and just for confusion's sake, you can put ++ either before or after the variable you want to increment depending on whether you want to increment before or after the value of the variable is read.

ie

```
a=3;
b=5;
c=b - a++;   // With parenthesis this should be c=(b - (a++)) for neatness
```

Will give c the value 2, and a the value 4. Because a was increment after it's value was used in the calculation.

Whereas with the same initial values (a=3 and b=5),

```
c=b - ++a;   // or c=(b - (++a))
```

Will finish with a being 4 but c being 1.

The operator -- will work in the same way, but decrementing the value instead.

I wasn't going to go into any library functions, but I feel obliged to help you write the typical first program - printing "Hello World" onto the screen. Admittedly, in BASIC this is a very simple task. Although C allows you much more flexibility, the complexities do bog down the very simple tasks.

```
#include "stdio   .h"       // note the three spaces

main()
{
    printf ("Hello World\n");
}
```

The **#include** line, lets you use the functions contained in that library - for details on the rest of the functions in stdio, see page 38.

**main()** as we know, defines the starting function and the code in it is contained within the curly braces.

**printf** Main calls the library function printf to output whatever follows. The **\n** is the newline (CR) character and tells printf to move to the start of the next line.

The above program will run on SAM C (you need to be careful of any spaces in between stdio and the .h). Just type it into the editor, press ESC to return to the main menu. Press F9 to compile it. The compiler will load in the libraries you've accessed, compile the code into assembler, then assemble the final code into an executable program. If either the compiler or assembler discovered any errors, go back and correct them, then try compiling it again.

Once the compilation has been completed (you'll be told the length of the object code), press a key to return to the main menu, then R to run the code.

With a bit of luck the words Hello World will be printed at the top left of your screen. Subsequent returning to the menu and running the code again will move each successive Hello World down the screen by a line - the result of your \n in the printf statement.

You've just written your first C program! Try experimenting with printing out variables using the %d operator (see page 39) and reading in values using scanf (see page 40)

You should be warned that C is an extremely powerful language, and you can do anything in C that is possible in machine code. You can assign memory directly and play around with the lowest level of your SAM, so any bugs that the compiler does not detect MAY CAUSE YOUR COMPUTER TO RESET - SAVE YOUR WORK REGULARLY. There is a switch in the Compile menu to save the source every time before compilation - it is worth using.

Although C will try to detect syntax errors in your code, it does assume that you know what you are doing and let you off with doing some unusual operations. Until you get used to knowing where to look for bugs, double check all of your code – nonobvious mistakes include forgetting the semicolon at the ends of instructions, getting the order of parameters mixed up, wrong assumptions about what C will do with calculations - check your precedence!

**Miscellaneous notes about C**

Instead of using a=a+5 or b=b-7, C allows you to use a+=5 and b-=7 instead. The same can also be done for multiply and divide by using *= and /=.

The bitwise operators are as follows :-

**&** **AND**
**|** **OR**
**^** **XOR**
**~** **NOT (One's complement)**
**>> x** **Shift right by x bits**
**<< x** **Shift left by x bits**

Declaring arrays is done by int a[10] which will give TEN variables - a[0], a[1], a[2] ... a[8] and a[9]. The dimensioning and accessing of the array must be done in square brackets [] ie.

```
int a(10];
a[0]=23;
a[9]=98;
```

The square brackets are obtained by pressing Symbol-R and Symbol-T.

Normal parenthesis () can be used to increase the precedence (see Appendix C) of part of a calculation, in the same way as BASIC does.

C applies boolean logic to relational operators ie if a is 5, a==5 will evaluate as TRUE, and return 1, whereas a==6 is FALSE and returns 0.

An array can be assigned a whole string (which must be done using strcpy - see page 47), with the string in double quotes ", and a string is automatically terminated by the NULL symbol \0. When assigning single characters to a char variable, single quotes should be used to avoid the compiler placing \0 after it in memory - which could be almost anywhere!

The **sizeof** unary operator returns the size of the variable (ie a char would return 8, an int would return 16). This can be useful for defining arrays whilst the code is running.

if statements are usually of the form

if (condition) action else other action

but if we are only using single expressions as the actions, we can use

condition ? action : other action

ie

**(a==b) ? printf("equal") : printf("not equal");**

In addition to the standard data types that come with C, you can make up your own. The two that SAM C allow are structures, unions and enumerations. Both are essentially just a combination of the already known types, but they can prove extremely useful.

Structures allow you to group together related pieces of information while still being able to reference them under one name - for example, it is possible to define variables called name, address, phone number and age but to keep things organised, we can define these variables as part of a larger data type - a struct. ie

```
struct person {
        char name[10];
        char address[50];
        char phone[10];
        int age;
        };
```

This has only defined the data type - no variables have been declared and no memory has been allocated. To do this, we set up a variable just like we set up any variable :

**struct person person_info;**

Which does define the variable person_info of type struct person. To access the information we use the dot (.) operator ie **person_info.age=21;** Of course, arrays of structs can be set up, and if you really get into things you can access a struct through a pointer to it via the **->** operator. The operators for normal pointers and address operators are * and & respectively - I recommend you read *a very* good book on C to get an understanding of such things – although when you do , you'll find SAM C copes admirably.

Unions and enumerations are defined in the standard way using the keywords union and enum.

Another simple program :-

To pause one second, print a message on screen, then wait until a key is pressed.

```
#include "system  .h"                          // include system library
extern int pause (int a);                      // prototype for pause function

ma() {
        pause (50);                            // call pause function for 50 50th s (1s)
        printf("\a\017\005Press any key");     // PRINT AT 17,5;"Press any key"
        while ( !kbhit() );                    // Loop (round an empty loop) while
}                                              // kbhit returns 0 (note the !)

#include "system   .c"
```

Although THERE ARE NO SITUATIONS WHERE GOTO IS REQUIRED - C is a structured language and goto should not be used (some versions of C actually prohibit the use of goto, but SAM C does allow it), the use of goto is :

```
label:
        goto label
```

ie. to change the above while loop to use goto, we can do:

```
restart: if ( !kbhit() ) goto restart;
```

You can have a look at the example programs on the SAM C disc, but because libraries are simply C code themselves, you can have a look at all the standard libraries on the disc as well (although there is a lot of assembly code used for optimum speed).

For performing numerous tests on a single variable, an if-then-else ladder will do the task, but a simpler way is to use the switch statement. A variable is successively compared to a list, when a match is found a set of instructions are executed.

```
switch(ch) {                    // ch is the variable to be tested
        case 1:                 // case statements without a break "drop through" so all
        case 2:                 // these cases have common statement sequences
        case 3:
                b=4;
                break;          // break terminates a sequence of instructions
        case 4:
                 b=2;
                 break;
        default:                  // default is used if no matches are found
                 b=17;
         }
```

In the above example, if ch is 1,2 or 3 then b becomes 4. If ch is 4, b becomes 2. If ch is anything else, b becomes 17. break is optional for "drop through" as above, and default is optional depending on whether you need a default value.

The general use of break is to force immediate termination of the current sequence. Here are two ways to force early exit from a simple loop :-

```
for (n=1 ; n<100 ; n++)                    for (n=1 ; n<100 ; n++)
        { if (n+a == 199) n=100; }                 { if (n+a ==199) break; }
```

Both programs do exactly the same, but the one with **break** is preferable because if the size of the loop is changed, nothing else needs changing. Whereas with the left example, if there were several 'early exit' tests and the length of the loop is changed, all the **n=100; statements** would also need to be changed.

continue forces the next iteration of a loop without executing any code - here is an example of a simple routine that only prints even numbers :-

```
for (n=1 ; n<100 ; n++)
        {
        if ((n/2)*2!=n)continue;
        printf("%d ",n);
        }
```

There are four basic storage class specifiers which can be applied to most variable types in C. These are : auto, extern, register and static.

auto is the default specifier and the one you will use most often - without even knowing it! When you assign a variable ie **int a;** it is, by default, an auto specifier.

extern is used when you are using separate modules and module. See the EXTERN MODIFIER section.

register makes use of the CPU's registers to speed up variables that are frequently accessed. See the REGISTER MODIFIER section.

static variables provide a more permanent storage method. Normally, every time a function is called, variables declared in that function are created. When the function ends, those local variables are destroyed. Declaring a variable to be static creates permanent storage space for it, meaning that every time you come back to the function, the variable is still there - along with the same value it had the last time you called that function!

While SAM C supports the normal use of pointers by * and &, it should be noted that double-pointers (a pointer to a pointer to a variable) are NOT allowed. Pointers allow more than one variable to be returned from a function as well as many other useful techniques like creating linked lists and binary trees of an unknown length. While pointers are a very important feature of C to more advanced programmers. as a beginner to C you will not need to use them until you start coding more complicated programs. The explanation behind pointers can be tricky, so I recommend getting hold of a good C book.

# Integrated Development Environment (IDE)

The IDE is the tool that you use to edit, compile and finally debug your software. This is where most of your work is done. The IDE is controlled from a group of menus from which options may be selected in several ways :

the underlined letter in the option name may be pressed;

the cursor may be positioned over the required option using either a joystick, keys or SAM Mouse;

Certain options (such as Directory) need room to give the user a list of alternatives to choose from. In this case a LIST BOX will appear and the alternatives will appear in it. The LIST BOX contents may be shifted up and down by using the UP/DOWN cursor keys, LEFT or ESC to exit, and RIGHT, RETURN or '0' to select the current (highlighted) item.

If you find yourself with an unwanted item, or wrongly selected option, pressing ESC at any point will take you back to the previous level of the menu system.

At some points you will find that the IDE requires you to type in a name (eg a filename to save or load). The text cursor position will be shown as  Enter your text and then press RETURN. Pressing ESC will remove the text request. If the option has been selected previously, the old text will re-appear when the option is next selected. The whole input line may be cleared by pressing the EDIT key. Hexadecimal numbers may be entered by prefixing them with '&', just as in BASIC.

When in the IDE main option menu, the screen shows a title bar at the top of the screen. This is the main menu. It has six headings - File, Editor, Debug, Compile, Options and Browser. At the bottom of the screen the name of the current source file is shown along with the size of the workspace for an asm file and the memory remaining for the source listing. The various sub-menus and headings will now be discussed in turn.

## File

### File

This option controls all access to the disc drive and file handling from the IDE.

### Load [F3]

As its name suggests, this option allows you to type in the name of a file to be loaded. This option may be selected quickly by pressing the 'hot-key' 'F3'. A dialogue box will appear for you to type in the required file name. If an error occurs in the attempt to load the file, an error message will be generated and the load aborted.

**New**

This command clears the edit buffer in readiness for the creation of a new source file. You will be prompted to input a name for this new source file. The source file is empty when first entering the IDE.

**Save [F2]**

Saves the current source file with its current name. The 'hot-key' F2 also has this effect. Again, any error will cause the save to be aborted with an appropriate error message

**Save as**

This is identical to the 'Save File' option above, except that you are prompted for a file name to save the source file under, instead of the source being saved as the current name

**Merge**

This option will append the chosen file to the end of your current source file. You are prompted for a file name.

**Print**

Print the source file to the printer. The printing is affected by various preset options (See page 23) and may be aborted by pressing ESC at any point.

**Get Info**

Displays version number, copyright and credit notices.

**Import**

This option will allow you to import an ASCII (Text only) file from any word processor (or C Compiler) that can produce such a file. This way you can choose to use your own word processor in preference to the built in editor supplied. N.B. The import process can take some time as the file is converted to the compiler's own internal format.

**Export**

The opposite of Import - allows you to save out your SAM C source file in standard ASCII format for porting to another C compiler or use in a word processor to get improved printer routines.

**Directory**

The current disc directory is displayed in a LIST BOX. By highlighting an item and pressing return, the chosen file can be loaded, merged, erased or imported.

**Drive**

This allows the specification of the current drive number. This option toggles the drive number between 1,2 and 3. Using the ram disc (Drive 3) saves considerable time when working.

**Quit**

This option quits the IDE and returns you to the calling program (usually BASIC). From BASIC the compiler may be restarted by pressing F7,

## Edit

The editor works just like a word processor in screen mode 3, with 64 characters per line. No line numbers are used in C, allowing you full use of all characters on each line. On the screen you will see a text cursor which indicates the position at which text will appear. Automatic scrolling of the screen is performed if a cursor movement would take it above or below the current portion of the source file that is being displayed. Various other keys also control the editing, and these will now be listed, along with their functions :

| | |
|---|---|
| ARROW KEYS | - Cursor Movement |
| CAPS | - Toggle CAPS lock |
| DELETE | - Backspace |
| SYM DELETE | - Delete current character |
| CTRL DELETE | - Delete a line |
| EDIT | - Clear current line |
| SYM EDIT | - Restore current line |
| TAB | - Tabulate to the right |
| SYM TAB | - Tabulate to the left |
| SYMBOL N | - Find next item |
| SYMBOL M | - Find next error |
| SYMBOL I | - Toggle Insert Mode |
| SYMBOL C | - Show Command Menu |
| SYMBOL S | - Swap location |
| CNTRL ESC | - As per SYMBOL C |
| SYM RIGHT | - Move text cursor to end of current line |
| SYM LEFT | - Move text cursor to start of current line |
| F0 | - Last Source page |
| F1 | - Page Down |
| F2 | - Set Block |
| F3 | - Insert Blank Line |
| F4 | - Page Up |
| F5 | - "{" left curly brace |
| F6 | - "}" right curly brace |
| F7 | - First source page |
| F8 | - "(' open parenthesis |
| F9 | - ")" close parenthesis |

In addition to these, you can set up a 'macro definition' on a particular key combination to achieve a desired effect. On startup, various combinations are already set for you in the 'CC.MAC' file on your C disc. Each of these requires the CNTRL key to be pressed along with the key indicated. These are, in alphabetical order by keyword, as follows :

CNTRL +

| | | | |
|---|---|---|---|
| = - #asm | 1 - #define | 7 - #else | " - #endasm |
| 9 - #endif | x - #ifdeclared | 2 - #ifdef | 0 - #ifndef |
| z - #include | m - #pragma argc | n -#undef | F0 - atoi( |
| b - break | q - case | c - char | k - continue |
| d - default : | a - do | j - else | e - extern |
| f - for | F4 - fprintf(" | F5 - fscanf(" | F9 - getche() |
| g - goto | o - if ( | F3 - input( | i - int |
| F2 - itoa( | F1 - itou( | - - NULL | F7 - printf(" |
| F6 - putch() | y - register | r - return | F8 - scanf(" |
| ; - sizeof | t - static | + - struct | s - switch ( |
| 1 - typedef | : - union | u - unsigned | v - void |
| w - while ( | | | |

TAB - #include "stdio .h"

CAPS - #include "graphics.h"

All other key combinations are available for you to set up for your own use.

**Entering Source**

After entering the editor, you may type your program (the source code). After pressing "RETURN on each line, any keywords on the line are automatically changed to lower case and redundant spaces are stripped from the code. N.B. In order for the keywords to be correctly recognised, there MUST be a space before and after each keyword.

a=sizeof(b); - This will NOT work!
a = sizeof(b); - Correct ! ! !

The last line of the screen is the status line, which displays the cursor position, CAPS lock, current mode, filename or block position.

**Block Settings**

To define a block (to use with the block commands) move the cursor to the first line and press F2 to select. Now move the cursor to the end line of the block and press F2 again to select the end of the block. At this point the status line will change to display the lines which are in the block : eg Block 0003 0007

**Block Operations**

First, you must have selected a block. Then enter the control menu by pressing SYMBOL C or CNTRL + ESC. The status bar will now display the control menu :

Copy Move Delete Find Replace Goto Start End Put Write cAlculate

To select an option from the control menu, press the key which is in upper case. When any of these options are selected, the current block settings are cancelled.

| | |
|---|---|
| **Copy** | Places an exact copy of the block selected at the current cursor position |
| **Move** | Moves the block to the current cursor position |
| **Delete** | Delete the block from the source |

**Find & Replace**

This option allows a search to be made of the source file for a search string which is input via a dialogue box (max 16 chars). You can use a wildcard '#' which matches any string, including an empty string.
eg POINTER# will match POINTER, POINTERS and POINTER2

To get the next item which matches the string, press SYM N. If you wish to change the string once found, the replace command may be used. This requires two parameters, the search string as before, and the replacement string. If the replacement string is empty, the first string will be removed from the source (ie replaced with an empty string). When the search string is found, the following message is displayed :

REPLACE(y/n/a/e) ?

Press y to replace the string, n to skip, a to replace all future strings which match and e to end the searching and replacement. If a block is selected, then this command only performs through the block, otherwise the whole source file is affected.

**Put**

This command takes the currently selected block and saves it to disc.

**Write**

Prints the current block on the printer.

**Goto**

To set the cursor at the line number specified.

**cAlculate**

Standard functions **+,-,/,*** may be used on decimal or hexadecimal numbers (preceded by &). Results are displayed in both decimal and hexadecimal forms.

**Inverse Colours**

Pressing the SPACE bar at this point changes the screen to display everything in inverse mode.

**Macros**

The editor supports text macro definitions. If a key, which does not have a macro already associated with it, is pressed, a macro recorder dialogue box is displayed. Pressing ESC aborts the option. The EDIT key cancels the current macro definition. Standard macro definitions are set up from the CC.MAC file on loading. To store your own macro definitions, select OPTION/EDITOR/MACRO/SAVE option from the main menu. NB : While defining the macro, pressing control keys (cursors, Delete, Return etc) will not have any action in the recorder and only be displayed as dots, however they will perform as expected when the macro is executed.

**Debugger**

The DEBUG option allows a monitor / debugger. BASIC memory allocation is as from &6180 to &7FFF which is saved into a buffer. Upon exit, BASIC is returned to its original state. Debug allows you to debug your program, setting breakpoints etc. Your program normally exists from &8000 to &FFFF in memory pages 1 and 2. The stack defaults to &4A00. The screen displays a control panel which shows the contents of various registers :

LMPR - Low memory pointer
HMPR - High memory pointer
VMPR - Video memory pointer
PRN yes/no - Printer
SCRN - User Screen page
OPTI - Work mode (later)

**Memory editing, arithmetic, text editing...**

CAPS - toggle upper/lower case characters
M - set memory pointer (MEM) to ....
UP KEY - Move to previous byte
DOWN KEY - Move to next byte
RETURNS - Insert byte to current location (MEM), if no current value exists, move to next MEM
SPACE -Insert word to current location, this writes bytes to MEM and MEM+1, if no value exists. move to MEM+2
F3 - Toggle between decimal/hexadecimal listing
A+SYM - Requires two parameters, shows result of +,-,*,/
T - Insert any ASCII text. When in text mode :-

UP = MEM-1
DOWN = MEM+1
ESC = End of command
DEL = MEM-1 and insert

ESC - Cancel current command
F9 - Set VMPR to user screen
F8 - Set VMPR to work screen (monitor use)
F7 - Disable/Enable printer output with listings
F6 - Set HMPR to new value
F5 - Toggle the debugger controlled part of the screen
F2 - Set OPTION bit : 0 - CR+LF after RET,JP...
: 1 - LF after CR to printer
: 2 - Formfeed after page

F1 - Set video page for debugger output
F0 - Show disassembly code
S - Set user screen video page value
SYM+S - Set video mode for user screen

**Listings, previews ...**

| | |
|---|---|
| F0 | - Disassembling from MEM address<br>ESC - cancel<br>LEFT - Return to start of listing<br>RIGHT - Jump to next listing page<br>UP - Step one line back<br>N - Find next item<br>Other - Next line of listing |
| V | - Disassemble from other address without changing MEM |
| O | - Text listing from MEM in ASCII form |
| O+SYM | - As for 'O' but for other address without changing MEM |
| L | - Hexdump from MEM as per FO |
| L+SYM | - As for 'L' but for other address without changing MEM |
| RIGHT | - Save current MEM to buffer and set to address |
| F4 | - As per RIGHT but set address to MEM |
| LEFT | - POP last address from buffer to MEM pointer |
| Q | - Show the MEM buffer content |
| U | - Show last 5 addresses on the Z80 stack |

**Block moving and filling**

| | |
|---|---|
| I | - Copies the block between FIRST and LAST to address |
| I+SYM | - Copies the block from FIRST of size LENGTH to address |
| P | - Fills the block between FIRST and LAST with byte |
| P+SYM | - Fills the block from FIRST of size LENGTH with byte |
| J | - Compares the block between FIRST and LAST with the memory from address TO |
| J+SYM | - Compares the block from FIRST of size LENGTH with the memory from address TO |

**Disc commands**

| | |
|---|---|
| SYM+R | - Read drive 1 at TRACK,SECTOR to address FIRST |
| SYM+W | - Write drive 1 at TRACK,SECTOR from address FIRST |

**Searching**

| | |
|---|---|
| G | - Up to a 5 byte string may be entered, terminated with RETURN. Pressing Z allows input in ASCII form. |
| N | - Find next item. |
| G+SS | - Show search string, debugger memory and current breakpoint address. |

**Trace, breakpoint**

Z+SYM - Execute one instruction (from current MEM) Increment by register R.
X+SYM - As per Z+SYM, but step over instruction.
V+SYM - Execute Z+SYM continuously until ESC pressed.
F toggles between slow/fast trace.
B+SYM - Execute X+SYM continuously until ESC pressed.
F toggles between slow/fast trace.
N+SYM - Setup registers
UP - previous register
DOWN - next register
RET - change register contents
RET - toggle flags
ESC - cancel command
W - Set start of program to execute.
U+SYM - Set breakpoint to current MEM and execute
program from address as set by W command.
E+SYM - Execute an EXX (as in machine code) command.
C+SYM - Execute an RET (as in machine code) command.

**Execute program**

H+SYM - Input address TO and this address is CALLed. Press 'Y' to execute.
T+SYM - As for H+SYM but the input address is executed by a JP command.

**Exit from debugger**

Q+SYM - Return to IDE.

**Compile**

This command converts your source code into assembly code. Errors are displayed and are included into the source file. If a fatal error occurs, compilation will be terminated without the production of the assembly code. From the main menu, the hot-key F9 performs the compilation using the current settings. ESC aborts the compilation. Once compilation is complete, a dialogue box appears with the following information :

Errors - number of errors
Symbols - number of global objects declared in source
Document size - size of compiled source file (bytes)
ASM file size - size of produced file
Compile lines - number of lines which are compiled
ASM file lines - number of lines in the assembly code file
Elapsed time - compilation time

Save source file - saves C source file before compilation (default - off)
Save ASM file - saves the assembler file as 'filename'.S (default - on)
Assembler pass - assembles the compiled code (default - on)
Save object code - saves object code as 'filename'.BIN (default - off)

Auto Run - run the code after compilation and assembly
Do it - Compile the file with the above settings

Using a RAMdisc for the include files saves considerable time during compilation.

The saved assembler file can be loaded into Comet assembler for hand optimisation and subsequent assembly. Comet costs £24.99 and is available from either FRED Publishing or Revelation Software.

# Options

**Compiler**

Any customisation that you set up for the compiler may be saved to disc (CC.INI) and will be automatically loaded the next time you load SAM C.

**Merge start code**

This option toggles between the inclusion/non-inclusion of the automatic inclusion of assembler directives at the start of the C program. These directives are stored in the CC.INIT file and the default is to be included in files.

**Stack pointer**

Start value of the Z80 stack pointer on program entry. The default value is &4A00 (18944 decimal). You may change this value to within the ranges &4200 to &4A00 and &6000 to &8000. The minimum size of the stack is 512 bytes. DO NOT CHANGE THIS VALUE UNLESS YOU ARE EXPERIENCED IN MACHINE CODE.

**ORG address**

Sets the address for the DUMP and ORG directives at the start of the final executable code. The default is &8000, but may be changed within the range &6500 to &FA00. DO NOT CHANGE THIS VALUE UNLESS YOU ARE EXPERIENCED IN MACHINE CODE.

**East code**

This option causes the code to be optimised for speed, but at the expense of being extremely large. (default - off)

**C src. as remark**

To allow remarks to be included in the assembler line preceded by ; (default - off)

**Label prefix**

The compiler creates its own internal labels. These consist of two characters and a number (CC0,CC1..CC65535). If two C programs are merged together, then duplicate labels cause a clash. The default labels are as shown with the letters 'CC and a number. You can use this option to define your own letters as the start of the labels in the second file. eg. CC for main file, C1 for first library, C2 for second library.
The library files are compiled once only and merged with the main file when needed. The prefix may be any alphanumeric characters, provided the first is a letter.

**Beep if error**

This self-explanatory option causes a beep to be generated if an error occurs. (default - off) **Pause if**

**error**

Waits for a key to be pressed if an error occurs. (default - off)

**Optimise size**

Two methods of code optimisation are available. The first is for size and the second is for speed. Some tips for faster programs are:
1. Use int, not char variables.
2. Use unsigned char, not signed.
3. Global variables are three times faster than local variables. This option works only if FASTCODE option is OFF only.

**Relative jumps**

This option toggles between the code producing absolute (JP) or relative (JR) jumps. The default is absolute jumps (off). Please note that if the relative jump option is selected, this can cause 'Displacement out of range' errors from the assembler.

**Unsigned char**

This option toggles between char declarations being automatically signed or unsigned. (default - on)

**Disc for output**

This option sets the disc drive that the assembler source will be saved to. The best drive to use is the RAM drive (if available). The files CCINIT.S and RUNTIME.S should also be copied here. The compilation time will be greatly speeded up.

**Printer**

SAM C has the facility to print out your programs via a comprehensive printer driver. This enables you to set many ways in which the printed output is formatted in ways which will now be described. The printing may be aborted at any point by pressing ESC.

**Header**

You may specify a header for each page up to a maximum of 35 characters long. The default is 'SAM C - Compiler Version 3.1'.

**Page length**

This specifies how many lines may fit on one page ( +2 for the header ). (default - 78)

**Margin**

The number of spaces which are printed at the start of each line. The default value is 8, up to a maximum of 32.

**Compressed**

Specifies whether the printed output is to be in normal or compressed form (10 or 12 characters per inch). Default is normal (off).

**Left feed**

This allows the sending of a Line Feed code after CR, the default is ON (see your printer manual for details of this setting).

**Form Feed**

This allows the sending of a Form Feed code after CR, the default is ON (see your printer manual for details of this setting).

## Preferences

**Mouse**

The graphics cursor (pointer) is normally controlled by the arrows keys or the joystick. If you have a SAM MOUSE, you can set this option to allow you to use it. DO NOT use this option if you do not have a mouse as your computer will lock up. (default - off)

**Text size**

You may change the maximum size of the source file in steps of 16K upto a maximum of 114K (9 pages) (16384 bytes = 1 page). The default value is 1 page (16384 bytes).

**Workspace**

As for the previous option but for the size of the output assembler file.

**Ink**

The colour of the ink (from 0 to 15), (default - 15).

**Paper**

As before, but for the paper colour. (default - 0)

**SVision**

If this option is ON, the programs will be compiled using the SAM Vision library. This uses the SV40.LIB file as the runtime file. The SAM Vision library is a separate library providing a more complex set of functions including a graphical user interface. It is still in development and will hopefully be released later in 1995.

**Ramdisc**

Number of tracks formatted in Ramdisc, upto 100. (default - 30)

**Font no.**

Font used (1-4) (default - 1)

**Editor**

**Autoindent**

If this option is ON (default) the editor will set the cursor to the same indentation as that of the last line. All spaces before the first letter on a line are compressed to one byte in the source file.

**Tab size**

This sets the size of the tab setting (4 to 16) default 4.

**Macros**

**Save**

Saves the current macro definitions from the editor. The definitions stored in the file CC.MAC are loaded on startup.

**Load**

Reload pre-stored macros into the editor.

**Reset**

Removes all macros except for CNTRL-C and CNTRL-I.

**Save**

This command stores the state of the switches into the configuration file CC.INI which is loaded on startup. All options set on the following menus are stored :

OPTIONS/COMPILER; OPTIONS/PRINTER; OPTIONS/PREFERENCES;

along with the EDITOR autoindent, ins/over flag and TAB size. This option should be used each time an option is changed.

**Browser**

This option is only accessible after compilation has been completed. It shows details of the global objects in the current program (variables, functions, etc) on the printer.

**Objects**

Shows a list box which contains the names of all of the global objects. The selected item will be displayed with the following information :

| | |
|---|---|
| Name | - Name of object |
| Ident | - Identifier :- VARIABLE, POINTER, ARRAY, FUNCTION |
| Type | - Type of object :- CHAR, INT, STRUCT, ENUM, UNION |
| Class | - Storage class :- STATIC, REGISTER, EXTERN, AUTOEXTERN |
| Sign | - Signed/Unsigned |
| Defined | - if defined |
| Used | - if used |
| Params | - Numbers of parameters or size of array size |

**Protocol**

The complete list of global objects and macros (#defines) are printed on the printer.

**Symbols**

The complete list of all symbols used in the compiled code

**CodeView**

A listing of the assembly code

# The C Language

**Introduction**

Because of the huge number of C implementations around, the American National Standards Institute (ANSI) introduced their definition of C in 1988 in order to maintain portability between different versions.

Although SAM C is not quite ANSI C, it is based more the original K&R C (Kernighan and Ritchie) but leans toward the ANSI standard and most features are the-same. The most important differences from ANSI C in SAM C are that long, double and float types are not implemented.

**Remarks**

The /* and */ sequence for the inclusion of remarks in standard C source is accepted by SAM C, but nested remark identifiers are not allowed. Remarks may be spread over several lines. eg

```
/*      This is a standard
              C remark block                    */
```

An extension in ANSI C is to allow single line remarks to be entered after a double forward slash sequence. eg

```
// This is a single line remark
// and this is another!
```

**Identifiers**

The name of each object must start with a letter, followed by any alphanumeric characters. Keywords (see below) may NOT be used as identifiers. The first 14 characters are examined by the compiler. N.B. Do NOT start identifiers with the'_' character.

**Keywords**

The following are recognised keywords in SAM C and may not be used in any other than the recognised context.

| | | | |
|---|---|---|---|
| auto | break | case | char |
| continue | default | do | else |
| enum | extern | for | goto |
| if | int | _interrupt | register |
| return | short | static | struct |
| switch | typedefstruct | union | unsigned |
| while | | | |

**Constants**

Integer constants - These are numbers in the range -32768 to 32767 inclusive, or 0 to 65535 if declared as unsigned. The number specified is assumed to be in decimal, or, in octal if the number is preceded by a zero, hex if preceded by '0x' (zero, x) and binary if preceded by '0b' (zero, b). Thus the following initialisations are equivalent :

int a=255, 0377, c=0xff, d=b11111111;

Explicit type is not implemented.

Character constants - any alphanumeric character placed between apostrophes (eg 'x') may be used as a character constant. Character constants are stored internally as unsigned ints. In addition to alphanumeric characters, the following are also valid constants :

| | | | |
|---|---|---|---|
| '\t' | - tab; | '\\' | - \ (backslash); |
| '\b' | - backspace; | '\'' | -' (apostrophe); |
| '\n' | - carriage return; | '\"' | - " (speechquotes); |
| '\a' | - AT(print at); | '\DDD' | - binary template; |
| '\f' | - form feed; | '\xHHH' | - hex byte; |

16 bit character constants are allowed but are treated as integer constants. eg int a='AB'; (=0x4142).

Floating point constants are not implemented.

Literal constants - These are implemented as standard except that if a string is longer than a current line length is required, placing a backslash at the end of a line will allow the continuation of the string on the next line. Eg

```
char *array = "this is a string \
on two lines";
```

**The size of standard types**

| Type | Size (bits) | Range |
|---|---|---|
| unsigned char | 8 | 0 to 255 |
| signed char | 8 | -128 to 127 |
| unsigned short | 16 | 0 to 65536 |
| signed short | 16 | -32768 to 32767 |
| unsigned int | 16 | 0 to 65536 |
| signed int | 16 | -32768 to 32767 |
| enumerated | 16 | -32768 to 32767 |

### Register modifier

The type 'register' may be applied to an identifier as per the ANSI specifications, if CPU registers are available. Variables declared of type register are stored in CPU registers rather than main memory, where normal variables are stored. This means the variables are accessed much faster because the CPU does not require memory access. Only two register variables are available at any one time.

### Automatic type conversion

Character expressions are represented internally as short integers. Integer and pointer representations are identical. An expression which involves integers and unsigned variables will return an unsigned type.

### Expressions

```
int c;
c = *(char *)12345;   //This is as per BASIC -   PEEK(12345)
c = *(int *)12345;    //This is as per BASIC -   DPEEK(12345)
```

The shift operators << and >> perform a logical (NOT arithmetic) shift.

## Declarations

### Variable

There are several variable types available each of which can be modified by the modifiers : typedef, auto, static, extern and register. Typedef and extern do not allocate memory for the associated variables. Exceptions to this are function prototypes. eg

```
static int num=1;       // static is implicit for globals
extern int num2;
char *string, *sfnc();  // string is pointer to char,
// sfnc is extern function gets a char pointer

char (*fncl)()[l0];     // fnc1 is an array of function
                        // pointers which gets a char

int main()
{
register unsigned a,b,c;
```

In the above program fragment, variable a will be placed in IX, b in IY, c and other variables will be allocated as auto variables on the CPU stack.

External variables do not allow direct initialisation. eg

```
extern string[] = "abcd";   // Causes an error
```

All of the SAM C library files are composed of two files, eg. graphics.h and graphics.c. The .h file contains all of the function prototypes of the functions which are defined in the .c file. At the start of your program you should use the #include directive to include any libraries which contain the functions that you wish to use, eg #include "graphics.h". Just after this you should type in the prototypes of the functions you will be using. At the end of your program, you should then include the corresponding .c file. SAM C will only include and compile those functions which are declared AND USED by your program. This keeps the final executable code as small as possible.

**Function modifier _interrupt**

The modifier _interrupt allows functions to be defined as interrupt handlers. The modifier causes code to be generated which saves all used registers upon entry and those registers to be POPed back from the stack upon exit. WARNING : THIS FUNCTION IS FOR ADVANCED USERS ONLY, USED INDISCRIMINATELY IT WILL ALMOST CERTAINLY CRASH YOUR MACHINE!

eg : A mouse handler fragment.

```
void (*old_mouse)();
_interrupt void mouse_handle( void )
{
        // your handler routine
if (old_mouse) (*oldmouse)();                   // call old vector
}

int main( void )
{
        old_mouse = *(int*)0x5AFC;              // save old vector
        *(int*)0x5AFC = mouse_handle;           // set new vector
        . . . .
        *(int*)0x5AFC = old_mouse;              // restore old vector
}
```

**Initialisation**

When a static object is declared, it may be initialised at that time using the '=' operator in the standard C way.

```
char space = ' ', *string = "sentence", *p_c = "next";
int number = -1, array[4] = { 1, 2, 3, 4 };
char *messages[3] = { "first", "second", "third" };
int array2[2][2] = { 100,200,300,400 };
```

The direct initialisation of structures is not possible. The initialisation expression must be a constant. If an object is not initialised, then it will be assigned NULL. Register and auto variables are not allowed to be initialised, they will be undefined on entry. If an array is initialised with fewer entries than the length of the array, the remaining items are filled with NULLs. eg

```
char pole[6] = "abcd";
```

The contents of 'pole' in order will be : 'a', 'b', 'c', 'd', 0, 0 Extern variables are not allowed to be initialised at declaration time, but can be initialised immediately afterwards :

```
extern char name[] = "Message";          // Not allowed!

extern char name[];
name[] = "Message";                      // This is OK.
```

**Compound Statement**

This is a sequence of single or compound statements enclosed between curly braces { }. Declarations are allowed at the start of a compound statement.

```
main()
{
    char a,b;
    a=b=3;
    if(a==b)
    {
        char a;
        a=b+1;
    }
    printf("%d\n", a);        // Prints 3, not 4
}
```

Block locals may have the same name as those in the containing block.

**Extern modifier**

Functions with variable number of parameters :

1. If a function has a variable' number of parameters, eg void printf(...) the compiler will place the number of parameters at compile time into register A.

eg printf("Number %d", 1 );

is translated as :

```
LD HL,CC0
PUSH HL
LD HL,1
PUSH HL
LD A,2              // this is the number of parameters
CALL PRINTF
POP BC
POP BC
 RET
CC0: DEFB "Number %d",0
```

**2. How many parameters?**

```
printf( int *arg )
{
        int count;
        return ((_print ((count=CCARGC()) + &arg-1, count);
}
```

CCARGC is a built-in function that returns the parameters entered in reverse order (last one first). The function must be called at the entry to a function. Register A is copied to HL and the function is called repeatedly until the Accumulator holds zero.

**The pre-processor**

The following compiler directives may be entered in your source :

- Macro Definition
  #define macroname macrobody

- Macro Removal
  #undef macroname

- Conditional Compilation on Macro Definition
  #ifdef macroname, #else, #endif
  #ifndef macroname, #else, #endif

- Conditional Compilation of Library Functions
  #ifdeclared nameofobject, ..., #endif

- Include Source File
  #include "filename"

- Inclusion of Error Message into the Source
  #error message

- Inclusion of Assembler Instructions into the C Source
  #asm, #endasm

- Compiler Check and Controls
  #pragtma

**#define**

#define cannot overwrite an existing macro. If you need to redefine a macro, you may do so as shown :

```
#define     POCET     10      //macro definition
#undef      POCET             //macro removed
#define     POCET     20      //new definition
```

**#include**

#include simply takes another file and inserts it into the same position. #include may not be nested in another #included file. It is usually found at the start of a program. eg

```
#include "stdio   .h"
```

When **#includeing** a library of functions, it is also necessary to include the prototypes (as in the header file) of the functions used.

**#ifdeclared, #ifdef, #ifndef, #if**

The #ifdeclared directive allows conditional compilation of the library files according to which functions contain the named objects. Its usage is :

```
#ifdeclared objectl_name [, object2_name [, ...]]
```

where object_name is the name of a function, pointer, variable, etc

```
#ifdeclared plot, box          // if plot and/or box are defined
void plot( int x,int y)        // then compile this function
{….}
#endif
```

All of the libraries are formed like this. This allows only the used functions to be compiled.

The #ifdef and #ifndef directives are used for standard conditions, as follows :

```
#ifdef SAM512
        int page=32;
#else
        int page=16;
#endif
```

```
#ifndef SAM512
        int page=16;
#else
        int page=32;
#endif
```

These two routines have exactly the same effect.
The #, if directive can evaluate any constant expression :

```
#if LINE >= 99   // LINE is a macro definition
…
#endif
```

**#asm, #endasm**

These directives allow you to code time-critical routines in assembly code.

```
#asm
// assembler instructions, directives and remarks // (as
per Comet Assembler)
#endasm
```

These directives can be used inside and outside functions:

```
#asm
ORG 16384
DUMP $ #endasm

int main()
{
cls(1);
}

void cis( int a )
{
#asm
LD A,L
CALL &14e
#endasm
}
```

1. At entry the last parameter value is in HL (see Stack Pointer for more details)
2. On exit, HL can be returned as the function value
3. RET instruction is automatically inserted at #endasm

**#pragma**

This directive allows you to issue special commands to the compiler. There are three uses of #pragma :

#pragma argc

If you are going to use command line arguments with your program, then you MUST place this command at the start of your program. It instructs the compiler to insert code to place the number of command line arguments into the accumulator at the start of your program.

#pragma test

This allows you to switch on the advanced compiler checks (unused variables and parameters, implicit integer return values etc are indicated). To switch this option off, use the #pragma test command again.

#pragma option

This allows you to set the compiler options found in the menu OPTIONS/COMPILER from within your program. There are 9 switches with parameters. You may use the comma(,) separator to specify more than one switch at a time. Please note that some of these options will only have a meaning to you if you are familiar with machine code. The switches are as follows :

| | |
|---|---|
| a (const expression) | - set start address of program(ORG) |
| s (const expression) | - set address of Z80 stack pointer |
| r (2 char const) | - set prefix for assembler labels |
| c (switcher) | - source in assembler list |
| o (toggle) | - speed optimise |
| f (toggle) | - size optimise |
| p (toggle) | - prolog/epilog linking |
| v (toggle) | - SAM vision library linking |
| u (toggle) | - '+' for unsigned or '-' for signed default variables |

N.B. 'switcher' uses '+' or '-' after option specifications

#pragma option c+, a 30000, o-

This means that the C source will be placed as remarks into the assembler file, the assembler directive ORG will be placed at the start of the machine code to force the start address to be 30000 and optimisation is not in use.

**Low-level communication**

This information is not normally of interest to the general user - it can be important if you are hand optimising routines or interfacing with your own machine code. It is included here for completeness.

**The Stack Pointer and Function Interface**

The Z80 stack pointer is used at the function interface to store both formal parameters and local variables. The arguments to the function call are evaluated from left to right and stored onto the stack - ie the last argument is on the top of the stack. If a function prototype exists, then the number of parameters will be checked. After the arguments have been processed in this way, the function will be called. Once the function has been entered? space is allocated for the local variables (on the stack). At exit this space is reclaimed. The arguments are removed from the stack after returning from the function. The return value of the function is stored in the HL register pair (and in DE when needed). eg

```
int a1=2, a2=4;
int main()
{
      al = print( al, a2, 100 );
}
int print( a, b, c) { }
```

Generates :

```
a1:     DEFB    2
a2:     DEFB    4
main:
LD HL,(a1)          // first argument to SP
PUSH HL
LD HL,(a2)          // second argument to SP
PUSH HL
LD HL,100           // last argument to SP
PUSH HL
LD A,3              // number of parameters
CALL print          // call function
POP BC              // remove arguments from stack
POP BC
POP BC
LD (a1),HL          // save result
RET
print:
RET
```

**Using the registers**

The compiler does not use the I and R registers in the compilation process. The compiler uses BC and H'L' as working registers, DE for temporary results and HL for the results of operations. The index registers IX and IY are reserved for register variables. When the program is run, the interrupts are enabled. The B'C', A'F' and D'E' registers are unused.

**Memory Sharing**

SAM C is loaded at the end of the free memory in your SAM and allocates 6 RAM pages ( 96K) which the program uses as an allocation table (ALLOCT). After this, the compiler allocates the next 4 pages as 2 for source, 2 for workspace. In total, 10 RAM pages are used ( 160K ). An optional 3 pages may be used as a RAMdisc (if you have MasterDOS) and 1 page for the SAM Vision buffers (if this library is used).

The 'runtime' file that is attached to your program calls ROM routines ( RST 16 and jump table vectors only ). Space is set by the compiler to default to 'OPEN TO 3' and RAMTOP to 32767. Memory for static objects is allocated at compile time. The free memory 'behind' your program is pointed to by the global pointer MEMPTR_ in the runtime file. This is used by the functions alloc() and free() which are found in "stdlib". Normally the program is compiled to run at address 32768.

**Program arguments and the RUNTIME file**

You may pass arguments to your program in the following way :

CALL start_address+3, "first", "second", "third", "fourth"

The main function can accept two arguments. The first (argc) contains the number of arguments on the command line minus 1. The second argument is a pointer to an array of char pointers ( char *argv[]; ). A simple example of this may be found in the demo ECHO.C source file on your disc.

load "echo.bin"code
call 32768+3,"Hello","World"

When calling your compiled program, you should call the start address (32768 unless you changed it with the options) if you have no command line arguments, or start address+3 if you are using command arguments (and don't forget to use the #pragma argc at the start of your program!)

argv[0] contains the name of your program (+'\0'). The first argument is stored in argv[1], etc. The last argument is argv[argc-1]. If argc==1 then no arguments have been given.

Your C program is merged with the runtime module automatically. This file contains routines for integer arithmetic, built in functions etc. It is an assembler file that is attached to the start of your program. When you return from a function (including main() ), the HL register pair contains the return value.

# APPENDIX A

**The Standard SAM C Libraries**

Built in functions are stored in the runtime file which is automatically attached to your program output. These routines have been hand coded in machine code and are extremely fast. It is recommended that you use these routines wherever possible.

The libraries consist of two files each. The first is the header file which contains function prototypes and the second is for the body of the functions themselves. Each of the files are readable by the editor as they are standard C files. To use the library files, include the relevant header file and prototypes at the start of your program, and then include the functions file at the end of your program. The libraries will now be described, although fuller descriptions are available in C books.

## "stdio .h"

This header file is unusual in that there is no corresponding function file. "stdio .h" contains headers for all of the built in functions and also definitions of standard names such as NULL, EOF etc. It should ALWAYS be included at the start of your program as most library functions need it.

**int getc(), int getch()**

One character is read from the current stream and returned. EOF will be returned on end of file or error.

**void ungetc( int c, int strm )**

This will push c back onto the stream such that the next call to getc() will return this character.

**int getchar()**

As per getc() but stream 2 (screen) is opened first.

**int fgetc( int strm )**

As for getc() but from a specified stream which is opened first.

```
int fgetc( int strm );
{ stream(strm); return (getc()); }
```

This stream will be used with all subsequent functions in the range -5 to 16.

**void putc(c), void putch(c)**

Outputs one character to the current stream (via RST 16)

**int getche()**

Returns code of the key pressed with echo to screen.

**int kbhit()**

Returns 1 if keypressed, else returns 0.

**char *gets( char *s )**

Reads a line from the standard input into the buffer at 's', deleting the newline. The buffer is NULL terminated. Its return value is its argument, or NULL on end of file.

**char *fgets( char *s, int max, int strm )**

fgets() places in buffer 's' up to max-1 characters from the stream specified. If a newline occurs before the correct number of characters have been read, then fgets() will return immediately. The newline will be left in the buffer. The buffer is NULL terminated in any other case. A successful fgets( will return its first argument, NULL is returned on end of file or error.

**void stream( int strm )**

stream() opens stream strm for I/O functions. strm may be in the range -5 to 16. This function calls the JSETSTRM routine in ROM0.

**void print( char *s )**

print() sends the specified string to the current stream.

**void fputs( char *s, int strm )**

fputs() opens the specified stream and uses puts() on s.

**void puts( char *s )**

As for print() but sends a newline character at the end.

**void fputc( int c, int strm )**

fputc() opens stream strm and outputs character c.

**void printf( char *cntrl, ... )**

printf() is a formatted output routine to the standard output (stdout). It takes as arguments a format string (which may include text) followed by a list of zero or more arguments. In the format string are conversion specifiers, each of which MUST correspond to an argument after the string. Each conversion specifier is of the form %m.nc, where the % indicates a conversion, followed by an optional width specification m, n is an optional precision specification (introduced by the dot() ) and c is the letter specifying the type of conversion. (It's much easier than it sounds, honest!) A minus sign(-) preceding m indicates left rather than right alignment of the converted value in the field. Where the field width is larger than that required for the conversion, blank padding is performed to the left or right as required. Where right adjustment of a numeric conversion is specified, and the first digit of m is 0, the padding will be performed with zeros rather than blanks. The valid conversions are :

Integer conversions: b O x X u d

These are (in order) to base 2, 8, 16, 16, 10 and 10. The conversion is signed in the case of d, unsigned otherwise. the precision value is the total number of digits to print, and may be used to force leading zeros. eg 8.4x will print 4 hex digits in an 8 wide field.

Strings : s

The value argument is assumed to be a character pointer. At most n characters from the string will be printed in a field m characters wide (if specified ).

Character : c

The argument is assumed to be a single character and is printed literally. Any other characters used as conversion specifications will be printed. So %% will produce a single percent sign. eg

```
printf("Total = %4d%%",23)
// The above prints : Total = 23%
printf("Name = %.8s","a1234567890") //
The above prints ; Name = a1234567
```

**void fprintf( int strm, char *cntrl, ... )**

As per printf, but the output is directed to the specified stream.

**void sprintf( char *s, char *cntrl, ... )**

As per printf but the output is directed to the specified buffer. The output will be NULL terminated.

**int scanf( char *cntrl, ... )**

scanf() performs formatted input from stdin, in much the same way that printf() performs formatted output. The input conversions are performed according to the format string, in general a character in the format string must match a character in the input. However, a space character in the format string will match zero or more 'white-space' characters in the input ie spaces, tabs and(or) newlines. A conversion specifier takes the form of the character '%' as per printf, optionally followed by a numerical maximum field width specifier, followed by the conversion character (which specifies what type of input is expected). Each conversion specifier will assign a value to the variable pointed to by the next argument. The conversion characters are as follows:

b o x d
Skip white space, then read a NUMBER in base 2, 8, 16 or 10 respectively. If a field width is supplied, only accept that many characters from the input. A leading minus sign will be recognised. The pointer argument is assumed to be a pointer of type int (ie call with 'address of an integer variable eg. &i)

s
Skip white space, then read a sequence of non white-space characters. The pointer argument must be a pointer of type char. The field width will limit the number of characters read. The resultant string will be NULL terminated. The input can be terminated by a white space character.

c
Read a single character from the input. The pointer argument is assumed to be a pointer to char. If a field width is specified, then that many characters will be read. This differs from the %s option in that white space is stored and does not terminate the input.

scanf() returns the number of successful conversions, EOF is returned if an error occurs before all of the conversions are completed.

```
scanf("%d %s", &a, &s);
with input "21s"
will assign 12 to a, 's' to s.

scanf("%4cd", &c);
with input " abcd "          // note the space either side of abcd
will assign " abc" to c
```

**int fscanf( int strm, char *cntrl, .... )**

As per scanf(), fscanf() performs formatted input but this time from the specified s tream. See scanf() for full details of the format of this command.

**int sscanf( char *s, char *cntrl, ... )**

As per scanf(), sscanf() performs formatted input but, this time from the specified character array. See scanf() for full details of the format of this command.

N.B. For fprintf() and fscanf() :

the streams which are available for use by these commands are the standard BASIC streams -5 to 16 (see Technical Manual). Stream stdout(2) is the upper screen, stdin(0) is the lower screen and stdprn(3) is the standard printer output. If the OPEN #2,"P" command is executed from BASIC prior to running your program, then all output is redirected to the printer. If you have MasterDOS, you can OPEN #4,"filename" and then read or write to "filename" by fscanf(4,...) or fprintf(4,...).

**char *input( int line, int column, int max)**

input() is a non-standard function for convenient text string input. The cursor "|" will be displayed at position "column","line" and allows a maximum line input of "max" characters. The following keys may be used for editing the input : RIGHT, LEFT, DEL, SYM DEL, EDIT (Clear Line). The input is terminated with the RETURN key. The function returns a pointer to a buffer which contains the NULL terminated input, or NULL if the input is cancelled by the user pressing ESC.

**void at( int line, int column)**

Set the current print position to column,line.

**void exit( value )**

Terminates the program and returns "value".

**char *itoa( int value, char *string, int radix )**

The integer value stored in "value" which is of base "radix" is converted to a string which is placed in "string". Leading whitespace is skipped.

**char *itou( int value, char *string, int radix )**

As per itoa() but value is assumed to be an unsigned integer.

**int isdigit( char c )**

Returns c if this char is a decimal digit, otherwise returns 0.

**int isspace ( char c )**

Returns c if this char is white-space (TAB, CR, LF, SPACE) otherwise returns 0.

**unsigned strlen(s)**

Returns the Iength of the NULL-terminated string s, not including the NULL.

**void assert( e )**

This function is used for debugging purposes. This command is placed in strategic points throughout your code where the correct operation of your code depends on certain conditions being true. An assert() may be used to ensure that at run time that the assumption holds true. For example, the following statement checks that the pointer tp is non-null :

assert( tp );

If at run-time the expression evaluates to false, the program will abort with a message identifying the source file and the address of the assertion, and the expression used as an argument.

**int CCARGC()**

This is a function which, if called at the start of a function with a variable number of arguments ( eg printf() ), returns the number of parameters passed to your function.

**char *MEIVIPTR_;**

This is a global pointer to char, which points to the first free byte following immediately after your program in memory. This pointer is used by malloc() and free().

**void *calloc( unsigned cnt, unsigned size )**

calloc() attempts to obtain a contiguous block of dynamic memory which will hold 'cnt' objects, each of length 'size'. The block will be initialised with zeros. A pointer to the block is returned if successful, otherwise a NULL pointer is returned.

**void free( void *block )**

free() deallocates the block of memory at pointer 'block', which was previously allocated using malloc() or calloc().

**void *malloc( unsigned cnt )**

As for calloc() above, but for 'cnt' bytes of storage which is left uninitialised.

## "ctype .h"

These functions are exactly as specified in the ANSI C library CTYPE. They each test the supplied character for membership is one of several overlapping groups of characters. Note that all of these functions are defined using the preprocessor directive : #if isascii()

**isaipha(c)** // returns true if c is in P.-Z or a-z
**isupper(c)** // returns true if c is in A-Z
**islower(c)** // returns true if c is in a-z
**isxdigit(c)** // returns true if c is in 0-9, a-f or A-F
**ispunct(c)** // returns true if c is non-alphanumeric
**isalnum(c)** // returns true if c is alphanumeric
**isprint(c)** // returns true if c is a printable character
**isgraph(c)** // returns true if c is a non space character
**isentrl(c)** // returns true if c is a control character
**isascii(c)** // returns true if c is a 7-bit ascii character
**tolower(c)** // converts upper case to lower case
**toupper(c)** // converts lower case to upper case
**toascii(c)** // converts c to 7-bit ascii (0-127)

## "graphics.h"

**void moveto( int x, int y )**

Move the graphics pen position to x,y co-ordinates.

**void plot( int x, int y )**

Set the point at the co-ordinates specified to the current colour.

**void drawto( int x, int y )**

Draw a line from the current graphics pen position to position x,y with the current colour.

**void line( int x0, int y0, int x1, int y1 )**

Draw a line from position x0,y0 to x1,y1 in the current pen colour.

**void box( int x, int y, int width, int height )**

Draw an unfilled box (outline only) from position x,y (top left corner) with a horizontal 'width' and a vertical 'height'.

**void cis( int a )**

If a is NULL, the whole screen is cleared, otherwise only the upper screen is cleared.

**put( int x, int y, unsigned data, int mode, unsigned mask)**

Place the sprite data in position x,y (top left corner) in the mode set, and with an optional mask. The start address of the data must be in the ranges Ox4000-0x7FFF or 0xE000-0xFFFF. The range of modes are :

0 - INVERSE
1 - XOR
2 - OR

3 - AND
4 - OVERWRITE (Default - fastest)
5 - USING THE MASK

The mask parameter is only available in mode 5. It points to a block of data the same size as 'data' and specifies a mask which indicates which areas of the sprite will be 'seen'.

**unsigned grab( int x, int y, int width, int len)**

This function copies (grabs) a section of the screen and stores it in a buffer in memory at the end of screen memory. x,y specify the top left hand corner of the block, where width is the horizontal width of the block and len is the vertical height of the block. The function returns the size of the block in bytes and MUST NOT exceed 8192 bytes! For more details see the Technical Manual (jgrab and jput routines).

**void fill( int x, int y, int mode )**

This function fills the enclosed area starting at x,y with a pattern set previously by a call to fillpattern(). If the pattern is NULL, then a solid fill will commence. Also see Technical Manual (jfill routine).

**void mode( int x )**

Set the video mode to mode x, where x is in the range 1 to 4.

**void open_scr( int scr, int mode )**

Open screen number 'scr' in mode 'mode'.

**void display( int scr )**

Display to the user screen number 'scr'.

**void close_scr( int scr )**

Close screen number 'scr'.

**void palette( int pos, int col )**

Set position 'pos' (0-15) in the current palette (CLUT) with colour number 'col' (0-127).

**void allpalette( char pal[16] )**

Change all of the colours in the current palette at once. The argument is a pointer to an array of 16 colour items (8-bit values) in the range 0-127.

eg  pal[0] = 56; pal[l] = 23; ... pal[15] = 31; allpalette( pal );

**void triangle( int x0, int y0, int x1, int y1, int x2, int y2 )**

Draw a triangle between the given points in order.

**void setpattern( char *p )**

This function takes, as its argument, the address of some data for the fill pattern which will be used by subsequent fill operations. If p is NULL, a solid fill pattern will be used (this is the default). The pattern data requires 8 bytes for the top row, and is 8 rows deep. ie 128 bytes are required.

**void color( int i, int p )**

Set current colour of ink to 'i' and paper to 'p' (Note the spelling of colour as 'color' in this function name).

**void pen(int i )**

Pen colour is set to 'i'.

**void paper( int p )**

Set the paper colour to 'p'.

**void setover( int o )**

This is the graphics writing status. The options for 'o' are :

0 - NORMAL
1 - XOR
2 - OR
3 - AND

**int getx()**

Returns the x co-ordinate of the graphics pen.

**int gety()**

Returns the y co-ordinate of the graphics pen.

**void fatpix( int s )**

If 's' is NULL then if the screen is in MODE 3, there will be 512 points per horizontal line, otherwise there will be 256 pixels per horizontal line.

**void scroll( int x, int y, int width, int len, int direct, int size )**

This function moves part of the screen in a given direction. The screen area must be an even number of pixels across. x,y specifies the top left corner of the block to be moved of horizontal 'width' and height 'len'. The block will be moved 'size' pixels in direction 'direct', where direct may be :

1 - left : 2 - up : 3 - right : 4 – down

**void roll( int x, int y, int width, int len, int direct, int size )**

As for scroll() but this time with wrap-around.

**void border( int c )**

Set the screen border colour to colour 'c' (0-15).

## "system .h"

**void disable()**

Disable interrupts.

**void enable()**

Enable interrupts.

**unsigned inp( unsigned port )**

Read the CPU 'port' and return the value read.

**void outp( unsigned port, char data )**

Write the 'data' to the CPU 'port',

**int escape()**

Returns 1 if ESC is pressed, otherwise returns 0.

**int avail( int a )**

Returns the size of the free memory that may be used by alloc() and malloc(). If the argument is non-zero and there is no memory free, the program will be aborted, otherwise NULL is returned.

The following commands are identical to their BASIC counterparts.

void poke( unsigned addr, char data )
void dpoke( unsigned addr, char data )
char peek( unsigned addr )
int dpeek( unsigned addr )

**int callcode( char a, int bc, int de, int hl, unsigned addr )**

The subroutine at address 'addr' will be called. Prior to this, the registers A, BC, DE and HL will be loaded with their respective values specified in the function arguments.

**void pause( unsigned w )**

Wait until w frame interrupts have been received ie waits w/50$^{th}$s of a second. If SPACE is pressed, the function is aborted.

**int gettime()**

Returns the content of the system variable FRAMES (&5C78)

**int getsp()**

Returns the current stack pointer value.

## "string .h"

**void memcpy( char *d, char *s, unsigned n )**

Copy n bytes of memory starting at 's' to the block starting at 'd'.

**void memset( char *s, char c, unsigned n )**

Initialise n bytes of memory starting at 's' with character 'c'.

**int memcmp( char *sl, char *s2, unsigned n )**

Compare two blocks of memory, each of size n, byte by byte, block 1 starting at 's1', block 2 starting at 's2'. Returns 0 if blocks are equal.

There now follows a series of functions that operate on NULL-terminated strings.

**char *strcpy( char *s1, char *s2 )**

Copy the contents at 's2' into the buffer pointed to by 's1' including the terminating NULL.

**char *strncpy( char *s1, char *s2, int n )**

As above, but only copies 'n' characters. N.B. The resulting string may NOT be NULL terminated.

**char *strcat( char *s1, char *s2 )**

Appends the string pointed to by 's2' to the string 's2' and NULL terminates the new string.

**char * strncat( char *s1, char *s2, int n )**

As before, but only adds 'n' characters from 's2' to 's1'.

**char *strcmp( char *s1, char *s2 )**

Compare the two strings pointed to by 's1' and 's2' and returns a number greater than or less than zero if the two strings are different, according to whether the byte in 's1' at the point at which the two strings differ is numerically greater than or less than the byte at that point is 's2'. A zero being returned indicates that the two strings are identical.

**char *strncmp( char *s1, char *s2, int n )**

As before, but only compares the first 'n' characters in each string.

**char *strchr( char *s, int c )**
**char *strrchr( char *s, int c )**

These functions locate an instance of the character 'c' in the string 's'. In the case of strchr(), a pointer will be returned to the first occurrence of the character in the string, whereas strrchr() searches from the end of the string BACKWARDS towards the beginning for the character. A NULL pointer is returned if the character does not exist in the string.

## "string2 .h"

**int otoi( char *octstr, int *nbr )**

The octal number stored in its character form in the string pointed to by 'octstr' is stored in its numeric form in 'nbr' upon exit from this function.

**int utoi( char *decstr, *nbr )**

As above, but the string 'decstr' points to an unsigned decimal number in its character representation.

**int xtoi( char *hexstr, *nbr )**

As above, but the string 'hexstr' points to a hexadecimal number in its character representation.

**int dtoi( char *decstr, *nbr )**

As above, but the string 'decstr' points to a decimal number in its character representation.

**char *itoo( int nbr, char *str, int sz )**

This function is the reverse of otoi() above - the octal number 'nbr' will be placed in its character representation into the buffer pointed to by 'str', the buffer being of size 'sz'.

**char *itou( int nbr, char *str, int sz )**

As above, but operates on the unsigned integer stored in 'nbr'.

**char *itox( int nbr, char *str, int sz )**

As before, but operates on the hexadecimal number stored in 'nbr'.

**char *itod( int nbr, char *str, int sz )**

As before, but operates on the decimal number stored in 'nbr'.

**void reverse( char *str )**

Takes a NULL terminated string pointed to by 'str' and reverses the characters in the string, whilst leaving it NULL-terminated.

**void left ( char *s)**

Left adjusts and terminates a string

## "stdlib .h"

**void abort( int c )**

Terminates program and gives a BASIC error message c (0 = OK ).

**int abs(int n)**

Returns the absolute value of 'n' - ie abs(-5) == 5 ==abs(5)

**int atexit( void (*fnc)(void) )**

Determines a function 'fnc' which will be called at program termination only. Only one call to the function, atexit(), is allowed.

**int max( int a, int b )**

Returns the greater of the two integer arguments.

**int min( int a, int b )**

Returns the lesser of the two integer arguments.

**int rand()**

Returns a 16 bit pseudo-random number.

**void srand( int seed )**

Starts (re-seeds) the pseudo-random number generator with 'seed'.

**void swap( unsigned n, char *s1, char *s2 )**

Swaps the two memory blocks, each of size 'n' pointed to by 's1' and 's2'.

**void beep( int duration, int pitch )**

Make a sound (beep) of 'duration' length and frequency 'pitch'.

**void sound( ... )**

This is a variable parameter function which enables you to send values to the registers in the sound chip. The parameters are arranged in pairs, the first of each pair being the sound register (0 to 31) and the second being the data to send to it (0 to 255). For more details please see the Technical Manual.

eg sound(27,2, 0,10, 8,16); sound (27,0);

**void nosound()**

Resets the sound chip to silence.

**int outprn( char c )**

The character 'c' is send directly to the printer, ESC is tested for.

**void gdump()**

As for the BASIC command DUMP.

**void tdump()**

As for the BASIC command DUMPCHR$. Both tdump() and gdump() require the dumpld utility to be resident in memory.

# "conio .h"

**void flash( int. f)**

If 'f' is 0. then the flashing attribute is switched off, if 'f' is 1, then flashing colour will start.

**void bright( int b )**

As for the previous command, but for the BRIGHT attribute.

**void inverse( int i )**

As before but switches the INVERSE attribute.

**void over( int o )**

As before but switches the OVER attribute.

**void tab( int t )**

The print position on the current line will be set to the argument 't'.

**void csize( int x, int y )**

As for the BASIC command CSIZE.

**void blocks( int b )**

As for the BASIC command BLOCKS.

**void window( int a, int b, int c, int d )**

As for the BASIC command WINDOW.

# APPENDIX B

**Error Messages**

Upon compilation any errors in your source code are indicated in your source by the introduction of extra lines explaining the error, which will each start with the directive #error. However, after correction of these errors, these error messages should be removed before re-compilation. Pressing CNTRL+DEL on the #error line will remove that line from your source.

**The error messages and their meanings**.

**1. Missing Token ")"**

The compiler expects a token, probably a right parenthesis, at this point.

**2. No closing #endasm or #endif**

You have not matched the start of an assembly code or if block with an appropriate ending. This test occurs at the end of compilation.

**3. Include nested**

An include file contains a #include. This is not allowed.

**4. Open failure on include file**

The included file is not found or corrupt.

**5. Bad include name**

The name of the included file must be between quotes, <> are NOT allowed.

**6. Void must be function of pointer**

The void type must be a pointer or a function - not a variable.

**7. Can't sizeof function**

You cannot take the sizeof a function.

**8. Not allowed in switch**

You cannot declare a variable in the switch block.

**9.Not allowed with goto**

**10. Not allowed with block-locals**

The goto statement may only goto a label inside a block.

**11. Must declare first in block**

The block-local variables must be first in a block.

**12. Must assign to char pointer or array**

Can't initialise a variable or a pointer to int by a character string eg
int *pole = "asd"; // is not allowed!

**13. Must assign to pointer**

Cant initialise a pointer to hit by a number.

**14. Negative size illegal**

Arrays sizes must be in the range 1 to 32767.

**15. Wrong number of Arguments**

The number of arguments submitted to the function does not agree with the function prototype.

**16. Illegal Function or Declaration**

The compiler shows this error when a corrupt line is compiled. See that no illegal characters exist in names (!@#$%^&...) etc.

**17. No open paren**

SAM C expects the bodies of functions to start with "{".

**18. Illegal argument name**

Check the names of the formal parameters.

**19. No coma**

The compiler expects a coma at this point.

**20. Type not implemented - int used**

You tried to use one of the unimplemented types - float, long, double.

**21. No semi-colon**

The compiler expected a semi-colon at this point.

**22. No final }**

The function has not been terminated with "}"

**23. Not in switch**

You cannot use the case or default statement outside of a switch.

**24. Too many cases**

A maximum of 25 cases is allowed at once.

**25. Multiple defaults**

Only one: default per switch is allowed.

**26. Bad label**

The goto statement has been used with a non label parameter.

**27. Not a label**

This label is already used.

**28. Illegal symbol**

The symbol contains an incorrect character (@#$%^).

**29. Already defined**

A symbol is defined more than once.

**30. 1-val required**

An lvalue, ie something which can be assigned to, is required after '&' or on the left hand side of an assignment.

eg 0 - a+b; ++(i+b); --41; etc

**31. Global symbol table overflow**

Only 700 symbols are allowed in one program.

**32. Local symbol table overflow**

Only 60 local variables are allowed in one program.

**33. Too many active loops**

Too many nested whiles, repeats, etc

**34. Out of context**

Break, continue, while used in an inappropriate position.

**35. No quote, no apostrophe**

You have unmatched quotes or apostrophes on this line.

**36. Line too long**

The source line is too big after the macro expansions etc. You should split the line or remove remarks from it.

**37. No matching #if**

Check the #if usage.

**38. Macro name table full**

You may only have a maximum of 256 macros.

**39. Macro string queue full**

The body of the macros is too big - remove some macros.

**40. Void function can't return value**

You have placed a return() in a function declared as of type void.

**41. Illegal address**

You cannot take the address of a register variable.

**42. Can't subscript**

The object is not an array.

**43. Undefined symbol**

This object is not declared in this block.

**44. Invalid expression**

This expression is badly formed.

**45. Function or void pointer in expression**

Pointer to function or void may not be used with an expression.

**46. Must be constant expression**

The compiler expects a constant expression.

**47. Literal queue overflow**

Literal string definitions in this function exceed 800 characters. Split a string declaration.

**Out of workspace**

The space for the assembler file is full. You can adjust the size of this workspace by using the menu OPTIONS/PREFERENCES and re-starting the compiler. Don't forget to save your work to disc first!!!

**49. Division by zero**

An attempt to divide by zero has been made.

**50. Bad option switch**

The switcher in the #pragma option directive is not correct. See #pragma for correct values.

**51. '-' or '+' use only**

Only the sign characters are valid after this switcher.

**52. Too big struct/union**

Structures and unions may not be bigger than 255 bytes.

**53. Syntax error**

Probably due to a bad structure or union declaration.

**54. Undefined struct/union**

This structure/union has not been previously defined.

**55. Struct/union in itself**

Nested structures/unions are not allowed.

**56. '.' expected after '..'**

The ellipsis symbol must have three dots.

**57. Can't initialise**

The local variables do not agree with the declaration.

**58. Struct/union can't be parameter**

A structure or union can not be used as a parameter - try the address of one of them.

**59. Typedef on local not allowed**

The typedef class specifier is only allowed outside of a function

**60. Short name**

The name of a global object must be at least 2 characters long.

**61. Macro with parameters not allowed**

No description

**62. Must be pointer - pointer required**

A pointer is required after a '*' (indirection) operator.

**63. Struct/union required**

A structure or union identifier is required before a '.'

**64. Struct/union member required**

A structure or union member is required after '.' or '->'

**65. Typedef class in expression**

This identifier is for declarations only.

**66. Struct/union in expression**

A structure or union may not been used in an expression - only a member can be used.

**67. Function declared implicit int**

This function has been called without an explicit declaration.

**68. Unused variables**

This message is shown only if you have selected extended error checks. Unused variables will then be indicated.

**69. Pointer type mismatch**

You cannot mix the various pointer types together without type casting.

# APPENDIX C

**Precedence of all C operators**.

Most operators associate from left to right, although the unary operators (* , & and -) and ? associate from right to left.

| | |
|---|---|
| **Highest** | () [] -> . |
| | ! ~ ++ -- - (type) * & sizeof |
| | * / % |
| | + - |
| | << >> |
| | < <= > >= |
| | == != |
| | & |
| | ^ |
| | \| |
| | && |
| | \|\| |
| | ? |
| | = += -= /= |
| **Lowest** | , |

# APPENDIX D

**This is a list of the files on your SAM C Disc**.

| | |
|---|---|
| CC | - BASIC Loader |
| CC.BIN | - The Compiler |
| CC.INI | - Configuration file |
| CC.MAC | - Standard macros for the editor |
| CC.SCR | - Intro screens |
| runtime.s | - The source code of the runtime functions |
| CCINIT.S | - prologue - routine for starting the C programs |
| DEBUG | - debugger code |
| stdio .h | - general macro definitions and headers of built-in functions |
| graphics.h | - The headers file for graphics.c file |
| graphics.c | - See main text of manual for functions |
| ctype .h | - Header file for the ctype .c functions |
| ctype .c | - See main text of manual for functions |
| string .h | - Header file for string .c functions |
| string .c | - See main text of manual for functions |
| string2 .h | - Header file for string2 .c functions |
| string2 .c | - See main text of manual for functions |
| conio .h | - Header file for conio .c functions |
| conio .c | - See main text of manual for functions |
| system .h | - Header file for system .c functions |
| system .c | - See main text of manual for functions |
| stdlib .h | - Header file for stdlib .c functions |
| stdlib .c | - See main text of manual for functions |

**Some example programs** :

| | |
|---|---|
| HELLO .C | // Traditional "Hello world" a la Kernigan & Ritchie |
| HILBERT .C | // The recursion algorithm of the HILBERT CURVE |
| ECHO .C | // Usage of parameters passed from BASIC on the command line |
| PRIME .C | // Calculates prime numbers up to 200. |

Executable versions of these files are included too. Note that HELLO and HILBERT are auto starting files. You can load and execute these files by :

```
LOAD "HELLO" CODE
LOAD "HILBERT" CODE
```

The ECHO program requires parameters and is used thus :

```
LOAD "ECHO" CODE 32768
CALL 32768+3, "FIRST", "SECOND", "THIRD", ... etc
```

# APPENDIX E

**Library Header Files**

**CONIO .H**

```
extern void border(int b);
extern void paper(int p);
extern void pen(int i);
extern void flash(int f);
extern void bright(int b);
extern void inverse(int i);
extern void over(int o);
extern void tab(int t);
extern void csize(int x,int y);
extern void blocks(int b);
extern void window(int a,int b,int c,int d);
extern void cls(int a);
```

**CTYPE .H**

```
extern char isalpha(int c);
extern char isupper(int c);
extern char islower(int c);
extern char isxdigit(int c);
extern char ispunct(int c);
extern char isalnum(int c);
extern char isprint(int c);
extern char isgraph(int c);
extern char iscntrl(int c);
extern char isascii(int c);
extern char toupper(int c);
extern char tolower(int c);
extern char toascii(int c);
```

**GRAPHICS.H**

```
extern void moveto(int x, int y);
extern void plot(int x, int y);
extern void drawto(int x, int y);
extern void line(int x0, int y0, int x1, int y1);
extern void box(int x, int y, int w, int l);
extern void cls(int a);
extern void put(int x, int y, unsigned data, int mode);
extern unsigned grab(int x, int y, int w, int l);
extern void fill(int x, int y, int mode);
extern void mode(int m);
extern void open_scr(int scr, int mode);
extern void display(int scr);
extern void close_scr(int scr);
extern void palette(int pos, int col);
extern void allpalettte(int pal);
extern void triangle(int x0, int y0, int x1, int y1, int x2, int y2);
extern int setpattem(int p);
extern void color(int i, int p);
extern void pen(int i);
extern void paper(int p);
extern void setover(int m);
extern int getx();
extern int gety();
extern void fatpix(int a);
extern void scroll(int x, int y, int width, int length, int direct, int size);
extern void roll (int x, int y, int width, int length, int direct, int size);
extern border(int a);
```

**STDIO .H**

```
                1
#define EOF     (-1)
#define ERR     (-2)
#define YES       1
#define NO        0
#define NULL      0
#define CR       13
#define LF       10
#define SPACE    ' '
#define stderr    0
#define stdin     0
#define stdout    2
#define stdprn    3
 /*
** These function prototypes are built up in the runtime file
*/
```

```
extern void ungetc(int c);

extern void putchar(int c);
extem void putch(int c);
extern void fputs(char *s, int strm);
extern void puts(char *s);
extern void fputc(int c, int strm);

extern int getchar( void );
extern int getch( void );
extern int getche( void );
extern int fgetc(int strm);
extern char * fgets(char *string, int max, int stream);
extern char * gets(char *string);
extern int kbhit( void );

extern void stream(int stream);
extern void print(char *s);
extern void at(int line, int column);

extern int printf(...);                    // these six functions are variable
extern int fprintf(...);
extern int sprintf(...);
extern int scanf(...);
extern int fscanf(...);
extern int sscanf(...);

extern char * input(int line, int column, int max);

extern void * MEMPTR_;                  //pointer to first free byte

extern char * skip(char *string);
extern char * itoa(int value, char *string, int radix);
extern char * itou(int value, char *string, int radix);
extern int atoi(char *s, int radix);
extern int isdigit(int c);
extern int isspace(int c);
extern unsigned strlen(char *s);

extern void assert(int e);
extern void exit(int c);
extern void * calloc(unsigned n, unsigned size);
extern void * malloc(unsigned size);
extern void * free(unsigned block);
```

**STDLIB .H**

```
extern void abort(int c);
extern int abs(int x);
extern int atexit(int fnc);
extern void tdump();
extern void gdump();
extern int max (int a, int b);
extern int min (int a, int b);
extern void sort(char *base,int offset, int size, int items, int (*comp)(), int (*swap)());
extern int rand();
extern void srand(int seed);
extern void swap(unsigned width, unsigned from, unsigned to);
extern void beep(int duration, int pitch);
extern void sound(...);
extern void nosound();
extern int is512kb();
extern int outprn(int c);
extern void memset(unsigned from, unsigned size, unsigned val);
```

**STRING .H**

```
extern void * memcpy(char *a,char *b,unsigned c);
extern char * strcpy(char *a,char *b);
extern char * strncpy(char *a,char *b,unsigned c);
extern char * strcat(char *a,char *b);
extern char * strncat(char *a,char *b,unsigned c);
extern int memcmp(char *a,char *b,unsigned c);
extern char * strcmp(char *a,char *b);
extern char * strncmp(char *a,char *b,unsigned c);
// extern void * memchr(a,b);
// extern unsigned strspn(a,b);
// extern char * strstr(a,b);
// extern char * strtok(a,b);
extern void * memset(char *a,int b,unsigned c);
extern char * strchr(char *a,int b);
extern char * strrchr(char *a,int b);
```

**STRING2 .H**

```
extern int otoi(char *octstr, int nbr);
extern int utoi(char *decstr, int nbr);
extern int xtoi(char *hexstr, int nbr);
extern int dtoi(char *decstr, int nbr);

extern char * itoo(int nbr, char *str, int sz);
extern char * itou(int nbr, char *str, int sz);
extern char * itox(int nbr, char *str, int sz);
extern char * itod(int nbr, char *str, int sz);

extern void reverse(char *s);
extern void left(char *s);
```

**SYSTEM .H**

```
extern void disable();
extern void enable();

extern unsigned inp(unsigned p);
extern unsigned outp(unsigned p,int v);

extern int escape();
extern int avail();

extern void poke(unsigned a, char b);
extern void dpoke(unsigned a, int b);
extern char peek(unsigned a);
extern int dpeek(unsigned a);
extern int callcode (int a,int b,int d,int h,unsigned adr);

extern void pause(int c);
extern int gettime();
extern int getsp();
```

# NOTES

The readme file from then Sam C disk displays the following message;

In order to successfully compile the news.c and some other flies it may be necessary to turn the smart Compiling off. This is the first option in the Options – Compile menu.

We would also recommend using MasterDOS for programming in SAM C.

## NOTES

This Sam C Manual was OCRed with Omnipage Pro 14
& MS Word 2003. The PDF document was compiled with
JAWs Creator pdf version 6.3
by Steve Parry-Thomas 17 Jan 2005

For SAM Coupé uses everywhere.

www.samcoupe-pro-dos.co.uk

SAM C Users Manual  PDF version 1 – 17 Jan 2005

We plan to release (both commercially and in the public domain) a wide range of C libraries, sources and products developed in C. If you did not buy your copy of SAM C direct from FRED Publishing, please get in touch so we can keep you informed of new developments.

In addition to C, FRED Publishing produce a wide range of games and utilities - ranging from the arcade hit Lemmings to the exceptional art package SAMPaint. For details on these, or any of the dozens of other pieces of software we stock please send a SAE to the address below.

First produced in June 1990, FRED disc magazine has gone on to become the biggest and best disc magazine for the SAM. Still produced monthly with a huge variety of items each month (news, reviews, games, demos, utilities, screens, music ...) it is available for £2 an issue or £20 for 12 months subscription which entitles you to further discounts off FRED software.

Requests for details / orders to :-

**FRED Publishing,**
**40 Roundyhill,**
**Monifieth,**
**Dundee,**
**DD5 4RZ  Tel : 01382 535963**