Embedded C

Copyright Embedded Results Ltd

Canolafan, Llanafan,
Aberystwyth, SY23 4AY

First Published 2007

TABLE OF CONTENTS

1 INTRODUCTION

This book is about applying the language C, originally developed for computer systems work, specifically to the field of Embedded Control, by using it to program micro controllers (MCUs).

C is a language that has rapidly evolved from its origins in the 1970's to become the `de-facto` standard for systems programming. It was developed by Dennis Ritchie at Bell Labs., from an earlier language `BCPL', which itself came from B (Cambridge University). The book *The C Programming Language,* by Kernighan and Ritchie, is still the most useful book available for C programmers, and every one attempting C should have a copy of it. This has recently been updated to cover the ANSI standard.

A new standard has been defined (ANSI) to which most C compilers adhere. For systems programming the language has recently been further extended to encompass Object Orientated Programming (OOP) and this version is now known as C++ with its own emerging ANSI standard.

C++ is now the standard language for most computer systems work and is available in a variety of forms such as TURBO C++, VISUAL C++ for Windows and many more. C++ tools are now also appearing for 32-bit MCUs.

WHY USE `C'?

The original language was developed to write most of the UNIX operating system for mainframe computers. As such, it had to be capable of low level functions and produce compact and efficient code from the compilation process. These are exactly the characteristics needed for micro controllers, that control at a low level and do not have much memory as a general rule (*although this to is rapidly changing*!)

The Object Orientated Programming extensions of C++ are not usually needed for embedded control work. (Although a C/C++ tool chain is now available to support the new NEC V850 32-bit RISC MCU). Consequently, two ANSI standards have emerged, C and C++, for different applications. You need to be able to program in C before learning C++.

Embedded 'C' is becoming *'flavour of the month*' with engineers. The reasons behind this sudden interest in 'C' for embedded control work are discussed below:-

- **POWER** - The advent of the latest super 8-bit MCUs such as the Arizona Microchip PIC, Atmel AVR, Hitachi H8, and many more, plus the appearance of 32-bit MCUs to replace Single Board Computers (SBC) in high end embedded applications, places huge power in the hands of the engineer. C is a very powerful, flexible and simple structured language, but this power comes at a price. The programmer is made responsible for a variety of aspects, which other language compilers would check. This allows you great flexibility in the design of the program, which means that you need to be more organised in your approach and application of the C rules. To program this power requires an efficient language. Assembly language is too unwieldy for this work. C is just right.

- **PORTABLE** - The C language is unusual in that it has no input/output or file access instructions, but can be extended by a set of standard libraries which contain the necessary functions. This makes the programs very portable, since C compilers for different hardware platforms, all have the same standard library functions, but the code for each will be unique to that platform. C Libraries for micro controllers have further extensions to access the special (INTRINSIC) functions of the device. If you wish to change the target processor, you recompile the program on a C compiler for the new device. This generates the new code for that device, making programs very portable, in theory at least. This is largely true for systems work, which have a very similar set of peripheral devices, even then the differing graphics handling makes for awkward conversion. For micro controllers it is unlikely to be that simple because each device will have its own unique set of intrinsic functions and different port connections! Even so. it should just be a matter of rearranging those sections of code to suit, and not a major rewrite. We will see how this works later because some of the example programs are tried on two 8-bit platforms, a CISC Von Neumann architecture MCU (Intel 8051 compatible) and a RISC Harvard MCU (Atmel AVR).

- **EFFICIENCY** - C is a medium level language, somewhere between the comprehensive instructions of high level languages (BASIC, FORTRAN etc.,) and the low level instructions required by the microprocessor. This is ideal for micro controllers, allowing fast, simple building of program structures, with the low level access needed for the special functions. You can concentrate on the structure of the program and leave the coding of the structures to the compiler. If your application requires 'number crunching' then 'C' will certainly be a lot easier to use than assembly language. Even 8-bit MCU platforms are now becoming so powerful, and capable of fast multi-byte

processing, that assembly language is becoming unwieldy. This is certainly true of the bigger 16 and 32-bit MCUs appearing on the market, where you need all the help you can get. Once the source code is produced, the compiler works on it, linking modules together, and then optimising the code. The end product is thus efficient object code.

- **SOFTWARE MAINTENANCE** - is much easier to perform, with helpful comments and reasonably obvious structures appearing in the C program. A decent C compiler will also format the code for you making the control structures easy to see. The C compiler also provides (mostly) useful error checking which is missing from assemblers. It will inform you if you try to do something illegal in your program, and warn you of any suspect areas of code, not just warn of bad syntax like assemblers. Any decent C development system, and IAR`C' is arguably the best, will also provide comprehensive project management facilities, enabling you to build a modular structure in your program development and build up libraries of routines.

- **MODULAR** - 'C' is designed to be built up from self-contained modules and then linked to form the complete system firmware. This is ideal for large projects, where several engineers can work on separate parts of the project firmware, test the modules and then link them all together for final testing.

****

At first glance, a C program looks a little strange, with the accent on terminal input and output. Any C book you pick up will concentrate on examples of input from the keyboard and output to the screen. This shows the upbringing of the language, it was after all created for systems work, where the major input and output device was a teletype terminal (TTY). Because of this, normal C books make little sense for MCUs, even though the language is the same. There was little point in writing yet another C book! Hence this book concentrates on the language applied to embedded control applications, with full language information, program structure and planning, and hardware and firmware examples. It takes a structured approach, using Design Structure Diagrams (DSDs) to illustrate program flow. (NOT the dreaded flowchart!)

The book is written assuming a prior knowledge of assembly language programming and of micro electronic systems in general. No attempt has been made to include basic systems work. I assume that you know all about logic operations, binary, hexadecimal, and interfacing techniques. If you need help on this I suggest you read the companion book to this called "*GET GOING WITH ... AVR MICROCONTROLLERS*". This not only introduces micro electronic system theory, it also describes the new RISC AVR micro controllers from Atmel which are used in some of the examples in this book. The instruction sets for these were designed with C in mind and produce very efficient code compilation!

The Arizona Microchip PIC range is, at present, a very popular 8-bit MCU for new designs. This is supported by IAR`C' (from late 1998), although with only 32 or so instructions to choose from, the code conversion is unlikely to be so compact as the AVR. (But I may be wrong, only time will tell!) The 8051 MCU is still around in huge numbers, multi-sourced and in current use. The change of platforms is a large step for a company, but here all you need to do is change software!

The book is targeted at Engineers and Students wishing to apply their expertise to embedded control applications. The micro controllers used in some examples are common ones available from a variety of manufacturers. Examples are also included using the latest RISC MCUs as well. You will need to modify the code to use the special functions needed

for your target MCU, if it is different. Examples of the kind of modifications required will be found in the last chapters, where we actually do it!

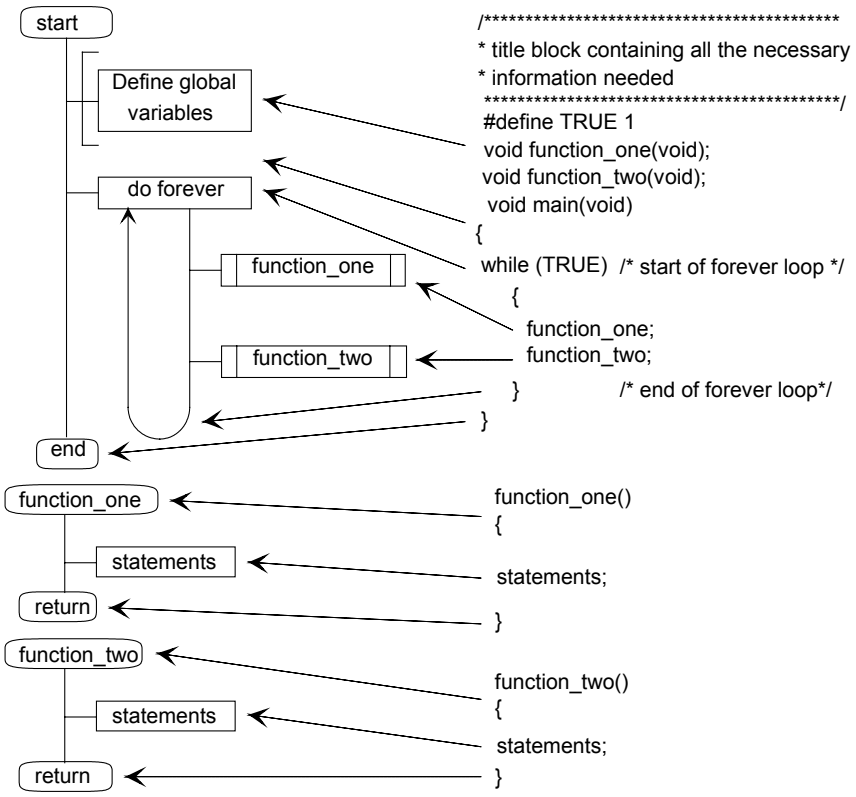
The IAR C employed in this book caters for a wide range of 8/16/32 bit microprocessors, and will come with the correct libraries for your chosen target when you install the software. (Assuming you ordered the correct version!)

The demo CD includes a wide variety of targets, so you may choose your favourite MCU. If you want to follow the examples exactly as in the book, you should install the Atmel manufacturer and the 8051 and/or AVR types. The Atmel 8051 derivatives are compatible with the Intel MCS-51™ devices and their clones but are In circuit Serial Programmable (ISP) which is very handy. The AVR devices are also ISP, having FLASH ROM, and are new RISC devices which are pin compatible (*except for the opposite polarity reset*) with the 8051 series but have a totally different instruction set! The AVR is very quick and easy to use and designed for 'C'. Consequently the AVR forms the target for many examples in this book.

AN OVERVIEW OF `C'

C is a very small language having very few keywords. This makes it very easy to learn (at first sight!). The format is very free, allowing the programmer great flexibility. It expects a BLOCK TYPE structure consisting of the main program (always called **main**) calling up a series of functions. The diagram below shows a typical program structure with DSDs and skeletal C language statements.

If you are new to Design Structure Diagrams (DSDs) then be patient, all will be explained.



In the source code, the title line shows how comments can be inserted in the program, either as title blocks, or as single line comments. Comments must start with a forward slash / and asterisk * and close with them in reverse order.
e.g. **/* comments */.** The comments can extend to many lines, you do not need to comment out every line, as you do with assemblers.

The global definition in this case is to define a symbol called TRUE = 1 for the WHILE loop test, which repeats forever, since it can never be false (0)! This is followed by the FUNCTION PROTOTYPES, which tell C the nature of the functions being called by the main program. (The type of variables being used and how much space to set aside for them.) In this case both functions take no parameters, and return no values and are hence declared as **void** type, requiring no memory space to be reserved.

The main program has its own DSD and code, with each function having its own DSD and code segment. Note that each segment of code, including **main** is surrounded by braces, **{** starts the code segment, **}** ends the code segment. The empty brackets **()** after each of the function names means that that code segment returns no value. ALL C functions must return a value, even if it is empty (void), as in this case.

Each C statement must terminate with a semi-colon **;**. This tells C when you have finished that statement (; = do it!). This is probably the most confusing area, what constitutes a statement? All will become clear as we progress.

Any variables defined in the functions (within the braces) will be assumed LOCAL to that function. That is, the variable name will only be known inside that function. This is essential because you will need to make code modules and fit them together later. It would be very difficult to ensure that you did not use the same variable name more than once over a large project.

ALL global variables must be defined at the start (outside the braces), and will be known to the main program and all the functions.

A good C compiler will format the source code to make it look like the DSD, with indents to show the loops and various coloured text for different types of words. e.g. Comments in one colour, keywords in another and different variable types in others. This makes a program very much easier to read, and hence debug. Good program documentation and easy program maintenance must be top of the list for any good software/firmware project design - IAR `C' provides all of this and more.

THE MAJOR DIFFERENCES BETWEEN TRADITIONAL `C' (K&R) AND ANSI `C'

For those of you who may have looked at C before, you might be interested in the improvements that have been made since the original C. The list below shows the major changes :-

- **entry** keyword is removed, allowing entry to be a user-defined symbol.
- **const** is added, an attribute indicating that a declared object is unmodifiable and hence may be compiled into a ROM segment.
- **volatile** keyword is added, an attribute indicating that the object may be modified by hardware and hence any access should not be removed by optimisation.
- **signed** keyword is added, an attribute indicating that an integer type is signed.

- **unsigned** keyword is added, an attribute indicating that an integer type is unsigned.
- **void** keyword is added, a type specifier that can be used to declare function return values, function parameters, and generic pointers as having no value.
- **enum** keyword is added that conveniently defines successive named integer constants with successive values.
- **DATA TYPES** available are now :-

  {unsigned/signed} **char**
  {unsigned/signed} **int**
  {unsigned/signed} **short**
  {unsigned/signed} **long**
  float
  double
  long double
  *- a pointer

These will be described as they are required in this book.

Getting the IAR C SOFTWARE

To try the examples you will need to get the IAR software. IAR produce a range of compilers called Embedded Workbench. These are available for a variety of platforms, including AVR and Atmel 89S range, which are 8051 based devices.

The software comes with a variety of license conditions, but for the purposes of this book, you should get the Free **Kickstart** editions. These are code size limited, but you just need to register to use them. They are available on IAR website (http://www.iar.com) or from these links – unless the site changes.

Atmel AVR Kickstart Edition of Embedded Workbench

http://supp.iar.com/Download/SW/?item=EWAVR-KS4

Atmel 89S Kickstart Edition of Embedded Workbench

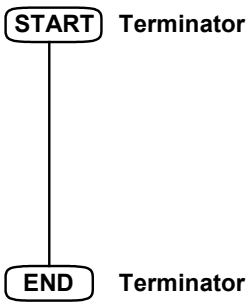
http://supp.iar.com/Download/SW/?item=EW8051-KS4

If you intend to use the Atmel AVR it would also be useful to look at AVR STUDIO software on this CD-ROM. This works with IAR 'C' and allows you to debug your programs in 'C' with a better presentation of the ports than C-SPY, the debugger which comes with IAR'C'.

You are now almost ready to follow the examples in this book and then try a few of your own ideas!

Before we do this you will need to know about Design Structure Diagrams (DSD) since these are used to explain the program flow in all the examples. I hope you will find them as useful as I do. One of the main problems with software development is getting the structure right. DSDs are a valuable tool for this, in any language.

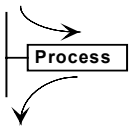
DESIGN STRUCTURE DIAGRAMS (DSDs) are a form of flowchart, but are very much more powerful and easy to follow. They provide a good, compact, picture of program flow and force you to structure the program. You prepare a picture, using DSDs, of what you want the program to do, and then code it into the language of your choice. The beauty of a DSD is that it is general purpose and relates to any language! A DSD is made up from a few simple constructs, very easy to learn (although, more complex symbols are available for system use): -



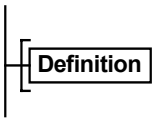
The start and finish of the program, indicated by round ended terminator boxes (as with flowcharts), are connected by a solid line (unlike flowcharts).
Program flow is downwards, no arrow is required.

The various processes now `bolt' onto the sides of this `stick'.

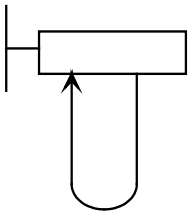
The processes are now fixed to the side of the sticks in the order of execution required. There is only one `NODE' which is the entrance AND the exit.



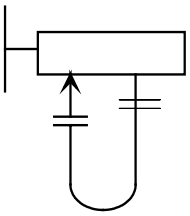
The dotted lines show the route taken, they are not included in the DSD, they are shown for explanation purposes only.
You place as much information in the box as you need to understand what the process has to do.



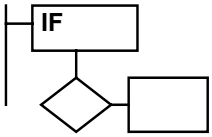
If the process is to declare or define variables, this is shown by adding a square bracket on the front.



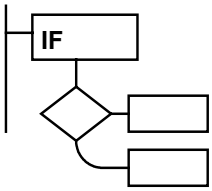
Loops are shown as a loop in a clockwise direction. The arrow shows this and is part of the symbol. The box will describe the type of loop and any terminating condition.
If the loop contains a test, to check for termination (escape), it appears as a capacitor type symbol.



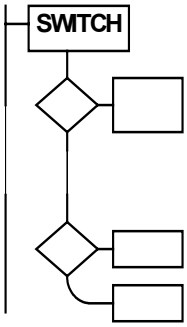
It can either be at the end of the loop (as shown) or at the head (shown dotted) depending on the type of loop.



Simple decisions are shown by a process box containing the test condition. If you only need that process to happen IF this condition is true it appears as one box on the diamond.

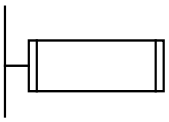


The ELSE process can be shown underneath, and will be done if the condition is not true.



A special multiple IF structure is sometimes useful called a SWITCH structure, which may or may not have an ELSE box.

The case steps up in integers (whole numbers) and may be as long as you like, with the conditions being tested following the word SWITCH.



Subroutines or functions are shown as a process with a double ended box.
The name of the subroutine or function will appear inside the box.
The function or subroutine will have its own DSD.

If the DSD should get too big to fit the page than the standard flowchart type connectors can be used :-

These are all that are needed to do most things in control work. We shall use these shortly in all the worked examples.

If you should wish to use more of these symbols for high level languages, for which they were designed, then please refer to the British Standard BS 6224 or later IEC standards which cover these symbols, and many more.

2. LANGUAGE STRUCTURE AND SYNTAX

GENERAL RULES

The introduction showed some basic rules of structure. We will now repeat and expand on these to give a workable understanding of the requirements. This chapter is for keyword and structure reference only and contains very few worked examples. It skimps over the instructions unlikely to be used for embedded control work and concentrates on those that are.

C has very few rules, but those that are there are strict. Here are the main ones just to get us started :-

BIG OR SMALL?

C is case sensitive and all keywords are **lower case**. You also usually keep your variable names in lower case but this is a matter of personal style. You can use this to advantage to make your program more readable, but it can be confusing to start with.

Beware! The compiler will treat variables such as **`variable`**, **`VARIABLE`**, **`Variable`**, and so on, as different variables. This may take you ages to spot when troubleshooting your program.

KEYWORDS

Keywords, the simple words available in the C language, are reserved words and must not be used as variable names (identifiers). They will only be recognised as keywords if they are in lower case.

RESERVED KEYWORDS (for ANSI `C') - *cannot be used as variable names or redefined :-*

auto	**default**	**float**	**register**	**struct**	**volatile**
break	**do**	**or**	**return**	**switch**	**while**
case	**double**	**goto**	**short**	**typedef**	
char	**else**	**if**	**signed**	**union**	
const	**enum**	**int**	**sizeof**	**unsigned**	
continue	**extern**	**long**	**static**	**void**	

In addition compilers will add new keywords for specific functions. IAR `C' adds the following keywords to access the functions of the MCU, these must also be avoided as variable names :-

C_task	**flash**	**interrupt**	**near**	**sfrb**	**tiny**
far	**huge**	**monitor**	**no_init**	**sfrw**	

These keywords may vary slightly between target processors and provide access to the special function registers, such as the control registers and ports. They also tell the compiler the memory size of the MCU, and control interrupts. The ones shown above are for the AVR processors, but the 8051 range is very similar. The help files will show you these within the package.

`C' also comes supplied with the standard libraries, which give you more words to use in the form of predefined functions. These provide input, output and mathematical functions. If you need to use one or more of these functions in your program, you will need to tell `C' to include the library files during the compilation process. This is done by adding lines at the start of the program such as :-

```
#include <header>  or
#include "myfile"
```

Where **<header>** is the name of a library file, stored in the predefined include directory, or **"myfile"** is a file you have created in your source code directory.

Different versions of C will all provide a standard set of libraries, but some will provide extra for other purposes. IAR`C', being designed for embedded micro. applications, comes with three distinct types of library file:-

- Standard Libraries, as found on all C compilers, which you can extend or modify.
- Start up modules for MCUs, which you can modify.
- Intrinsic functions, unique to the target processor.

The help files provide a summary of the example libraries available, with sufficient information to enable their use. The examples in this book show the most common functions applied to MCUs.

The manuals for the software will give you more detailed information. *(But you will need to buy the full version!)*

As an example the standard I/O library **<stdio.h>** contains :-

Function	Purpose
getchar	Gets character from the input stream
gets	Gets string from the input stream
printf	Writes formatted data to the output stream
putchar	Puts character to the output stream
puts	Puts string to the output stream
scanf	Reads formatted data from the input stream
sprintf	Writes formatted data to a string
sscanf	Reads formatted data from a string

CONSTANTS

Enable you to set an unchanging value to a symbol. The symbol name can be in upper or lower case, but upper case is usually employed to make them stand out as CONSTANTs. This is important because, for obvious reasons, you do not want to change constants. If you do, and you have defined it as a constant, `C' will tell you! If you adopt the use of upper case for constants you will not get confused.

A constant can be defined in the same fashion as a variable, but this is dangerous because it can then be changed! The C compiler will also not know that you intend it to be a constant. If you mean it to be a constant the last thing you want to do is to accidentally change it in your program! Consequently it is better to define your constants at the start of the main program using the #define preprocessor directive.

The format for this is :-

#define CONSTANT (**value**)

Where **CONSTANT** is any valid name (usually in capitals!).
The compiler will now use **value** everywhere it sees the symbol **CONSTANT**.

This also means that if you should need to change the value at any time you only have to change the definition once and recompile. This is similar to the **EQU**ate function in assembly language.

The other advantage is that you can do a selective compilation for different versions of your firmware, based on the value of one or more constants. Preprocessor directives are supplied for this purpose.

VARIABLES

Variables are place holders of a predefined size to store the results of computation**.** They **must** be declared as to the TYPE before they are used, so that `C' knows how much memory space to reserve for them. This is particularly important with MCUs because memory space is at a premium.

The usual rules about illegal characters hold true, i.e. names must start with a letter (or under score _), and must not contain mathematical operators or spaces. ANSI `C' allows identifiers (names) to be up to 255 characters long.

Variables declared at the start of the main program (outside the braces) are treated as GLOBAL, that is they can be `seen' by the main program and all the functions.

Variables declared at the start of functions (inside the braces) are treated as LOCAL and can only be seen within that function. This enables the same name to be used for different purposes in the main program and all the different functions with no confusion (to the program at least!).

Before a variable is used, you must declare the type so that C can allocate space for it. The most efficient type is the CHAR type intended for storage of ASCII codes for characters. This is a single byte storage space and should be used wherever possible. It is the default type in IAR `C' for this reason.

The table overleaf shows all the types available in ANSI `C'. Some of these are the same on MCU platforms and are kept for portability reasons. The word sizes relate to 8-bit MCUs and may be different for larger word size processors. The number range should be the same though, since this is defined in the ANSI standard.

TABLE OF DATA TYPES IN ANSI `C'

Data Type	Bytes	Range	Notes
char(unsigned)	1	0 to 255	Default type
signed char	1	-128 to +127	2's complement!
short	2	-2^{15} to $+2^{15}-1$	-32768 to 32767
int	2	-2^{15} to $+2^{15}-1$	-32768 to 32767
unsigned short	2	0 to $2^{16}-1$	0 to 65535
unsigned int	2	0 to $2^{16}-1$	0 to 65535
long	4	-2^{31} to $2^{31}-1$	-2147483648 to 2147483647
unsigned long	4	0 to $2^{32}-1$	0 to 4294967295
pointer	1,2 or 4		See special functions later.
float	4	+/-1.18E-38 to +/-3.39E+38	Floating point number
double	4	+/-1.18E-38 to +/-3.39E+38	Same as float
long double	4	+/-1.18E-38 to +/-3.39E+38	Same as float
sfrb	1	unsigned char in I/O registers	Resides at fixed location in I/O for MCU only.
sfrw	2	unsigned short in I/O registers	
enum	1,2 or 4	Creates each object with the shortest integer type required to contain its value.	

Variables can be declared by :- **`type identifier;`**

Where **`type`** is one from the list above and **`identifier`** is the variable name chosen.

e.g. **`int number`**;

Declares a variable called **`number`** as type **`int`**, an integer capable of housing whole 16-bit numbers in the range -32768 to +32767.

ANSI C also allows you to initialise the variable on the same line, to give it a starting value.

`type identifier = value`;

e.g. For the above it might be :-

```
int number = 468;
```

This assigns the number 468 to the variable **`number`** and allows two bytes of storage space for it. If the maximum number size was likely to be less than 255 then it would save memory space if you used the **`char`** type instead which only occupies one byte.

You can declare many variables on one line, separated by commas.

e.g. **`int number1, number2, number3;`**

This line declares all of these as integer type. Notice that each line ends in a semicolon since it is a statement.

Assigning values to a variable can be done at any time before it is used. The general format is similar to other languages :-

```
variable = value;
```

This may be combined, as shown previously, with the declaration. This may seem straightforward, but it is often confusing because of the type of the variable. For instance if a variable is declared as a **`char`** type, this can be an ASCII character or its number, such as :-

```
char letter = "a";
```

This declares the variable **`letter`** as a **`char`** type and assigns the number **97** to it, which is the ASCII code for "a". Since this works with numbers you can increment or decrement this variable to step through the ASCII character set. Very useful sometimes.

With MCU systems the **`char`** type is used as a normal variable for numbers to preserve space.

CODE SEGMENTS

Can be placed anywhere in a program, and allow you to make code blocks. Code segments must be bounded by braces **{ }**. Some constructs require them, as we shall see. The braces

can be on the same line as the code, but it is better to split them over separate lines and indent them to show the structure in more detail. It does not produce any more code and it makes debugging that much easier. IAR 'C' automatically follows your indents and all you need to do is press enter each time. This is of great help. It is the little things of this nature that separate a good package from a great package.

MAIN PROGRAM

Can be placed anywhere, the best position is subject to some conjecture! It makes sense to put the main function first, followed by all the function definitions. If you do this, you must declare the types of the function at the start of the program. This is commonly called the ***FUNCTION PROTOTYPE***. It MUST, however, be called **main.** This is not a real limitation, it makes sense since it is the main function that executes first.

Some programmers prefer to place their functions in the order of call from the main function, and put the main function last. This saves them declaring the types at the start, using the function definition as the declaration. The choice is yours and one of style/preference. I tend to prefer the obvious approach, though it forces you to declare your functions at the start. This makes you think about the program more, and also fits in better with any structure diagrams you may draw to plan the structure beforehand.

RETURN VALUES

All functions, including **main**, assume they pass and return a value. That is, the function is called and a parameter passed to it from the calling function, and then the result of the program is passed back to the calling function. If they do not take parameters the brackets must still be there, but empty (void), with no space between the function name and the brackets. The function return must also be declared as to type. Functions can be of any valid type except **`main,`** which must usually be type **`int`** in most compilers, although IAR `C' allows **`void`** as well! This makes more sense because the main function has nowhere to pass values to!

e.g. **`int main()`**

We will see later how to pass and return a value. ANSI `C' adds a **`void`** keyword which can be used to make this clearer and most programmers prefer to use this. You may even get a warning from some compilers if you do not!

e.g. **`int main()`** becomes
`void main(void)`

This is now very clear that the function **`main`** passes and returns no value, and that the compiler does not need to reserve any memory space for the value.

COMMENTS

Allow you to document the program for easier maintenance. The usual assembler method of adopting the semicolon (;) or backslash (\) does not work with C, nor does the // method of C++! (Except on IAR`C' where you can set an option to allow this). Comments must be surrounded by /* **`comments`** */ and can be as many lines as you like.

The formatting and use of comments is a matter of personal preference and style. All the examples in this book use the same style (I hope). This is another important point to note. Once you set a style that works, and you like it, stick to it. Some aspects of C are confusing enough without worrying about differences in layout. The important thing to ensure is that your program is easy to read and follow. Not just for you, but for anyone else who may be interested. Some programmers pride themselves on producing the most compact written source code possible, using all of C's more obscure operators. This is also usually impossible to read and debug. For instance, I like to start the program with a good title block. It might look something like this (please excuse my sense of humour, it comes from too many years stood in front of engineering students!):-

```
/*****************************************
*  Program to drive a wim wom for grinding *
*  smoke.   Written by Mike Ro.            *
*  Debug Version 0.2     November 5th 1998 *
*  Target Processor AT90S8515              *
*****************************************/
```

This should explain, at the very least, what the program does, the revision number (version), the date it was updated, and by whom, and the target processor. Other items may be added as you see fit. The only important thing, as far as `C' is concerned, is that it is surrounded by /* and */, so it does not try to compile it as code.
IAR 'C' detects that they are comments and shows them in blue italics (which you can customise if you wish). If your comments do not show up in italics you have not placed the comment operators in the right order. *(Yes, I have made that mistake too!)*

In-line comments should fully explain what that section of code does, in plain English, and will usually match the wording of the DSD, if you did that right. Time spent here will be worth its weight in chips!

The following lines show some examples of in-line comments:-

```
#include <stdio.h> /*includes standard I/O header
                                        file*/

total=count++;     /*running total, after increment*/
```

EXAMPLE PROGRAM

Now it is worth trying a simple program to illustrate the rules :

```
/*****************************************
*  Program to do what I said not to, i.e  *
* write a simple message to a non-existent*
* screen!   Written by Peter Sharpe       *
*  Release Version 1.0   March 1998       *
*  Target Processor Atmel AT89S8252       *
*****************************************/
#include <stdio.h>
void main(void)
{
   printf("It works, isn't it great!");
}
```

Not to break tradition, this is typical of the first `C' programs written in books. (Except they usually say "Hello World"). This program will actually run on IAR`C' CSPY debugger, regardless of target processor (but not on the MCU unless a terminal is connected) because it has a terminal window which acts like a screen. The important points to note are :-

- that the program starts with a title,
- the library file containing the **`printf`** function is included,
- the main function is declared type **`void`** and returns no value (void),
- the only function called, **`printf`**, has the correct data sent to it,
- and the main code block is surrounded by braces.

As I think you can guess, all this program does is print "It works, isn't it great!". Not terribly constructive or useful is it? Certainly not worth typing in yet.

NUMBER CRUNCHING

In C, there are many special characters with particular meanings, which most novices find very confusing. This is particularly true of the operators required to do some process on variables/constants.

ARITHMETIC OPERATORS

The main arithmetic operators are quite normal :-

+ = addition **-** = subtraction
***** = multiply **/** = divide
% = modulus (remainder after integer division)

The increment (++) and decrement (- -) operators are unary operators which can be applied to variables either before (prefix) or after (postfix) depending on whether the signs are before or after the variable. e.g `c = a++`; assigns the value of `a` to the variable `c` and then increments `a`, whereas `c = ++a`; increments the value of `a` and then assigns the value to the variable `c`. This is easier to type than the equivalent long hand method such as :-

```
a = a +1;
c = a;
```

These operators are very efficient since they have a direct equivalent in assembly language in the form of INC and DEC, but they can look strange in a program.

NOTE: *You cannot increment or decrement constants or expressions, only variables.*

COMPOUND OPERATORS

You will often have to change the values of your variables inside the program (*otherwise they should be constants!*). The increment and decrement operators allow this in steps of

ONE. To make other steps you can use the compound operators, which also allow you to use other processes apart from plus or minus.

COMPOUND ASSIGNMENT OPERATORS

Compound Operator	Example	Equivalent statement
*=	product *= 1.5	product=product*1.5
/=	quotient/=number	quotient=quotient/number
%=	pass%=4	pass=pass%4
+=	count +=2	count=count+2
-=	reduce-=5	reduce=reduce-5

These can look rather strange in programs until you get used to them. They are worth persevering with because they save a lot of typing.

SHIFT OPERATORS

Enable bytes or words to be shifted left or right a number of bits at a time. The end bits are replaced with zeroes. There are two shift operators :

OPERATOR	FUNCTION	EXAMPLE
<<	Shift left	X<<2 same as x*4
>>	Shift right	x>>1 same as x/2

ORDER OF OPERATORS

The mathematical process you may want to carry out can produce unexpected results. This is usually caused by an unexpected order of calculation based on precedence and associativity (*which one is done first and from which direction*)
For instance:-

```
sum = 8 + 3 * 4;
```

Would produce the answer 20, not 44, because C places a higher priority on multiplication than addition. If you are unsure, always use parentheses (brackets) and you cannot go wrong.

The priorities given by C to such operations are shown below (this is a complete list including some we have not explained yet) :-

Priority	Operator	Associativity
1	() parentheses, [] array element	Left to right
2	- (negative sign), ++, --, &, * (pointer), !	Right to left
3	* (multiply), / (division), %	Left to right
4	+ (addition), - (subtraction)	Left to right
5	<, <=, >, >=	Left to right
6	= = , ! =	Left to right

7	&&	Left to right
8	||	Left to right
9	? : (conditional operator)	Right to left
10	=, *=, /=, %=, +=, -=	Right to left
11	, (comma operator)	Left to right

LOGICAL OPERATORS

There are only three logical operators supported by C. These allow you to make decisions over two or more variables.

Operator	Operation
&&	AND
||	OR
!	NOT

Note the absence of Exclusive OR (EOR or XOR). This can of course be made from the ones above and defined as a function. Since the major usage of the EOR is to invert bits in embedded control work and the bitwise operators do this in C, then this is not a problem. These operators are intended for decision making work on expressions and not on a bit wise basis. (See Bitwise operators.)

RELATIONAL OPERATORS

These operators allow you to test for relationships between variables or constants. ANSI `C' has six operators for this purpose :-

Operator	Operation
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
==	equal
!=	not equal

BIT WISE OPERATORS

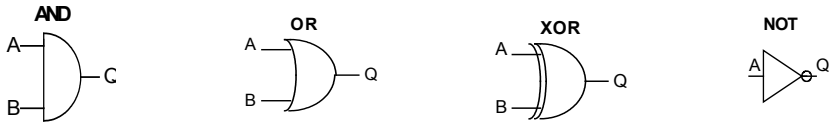
There are four bitwise operators, which are essential for any embedded control work.

Operator	Operation
&	bitwise AND
|	bitwise OR
^	bitwise XOR
~	1's complement (NOT)

These operators work with **char** and **int** types only, for obvious (*or it should be*) reasons. Just as a reminder, the truth tables for these functions are shown .

AND			OR			XOR			NOT	
A	B	Q	A	B	Q	A	B	Q	A	Q
0	0	0	0	0	0	0	0	0	0	1
0	1	0	0	1	1	0	1	1		
1	0	0	1	0	1	1	0	1	1	0
1	1	1	1	1	1	1	1	0		

These would be in each bit of course, normally eight bits wide (type - char). The rows in GREY background are the unique conditions. In the case of XOR there are two. That is you only get a one if both the input bits are different.

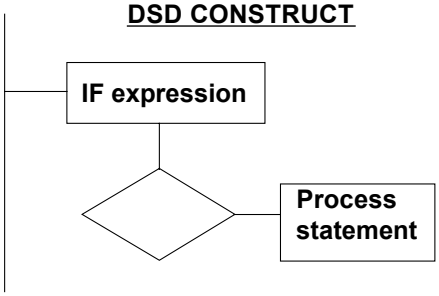


PROGRAM CONTROL STATEMENTS

A program needs to make decisions and form loops to be useful. C provides a number of control structures, which enable this to happen. We need to look at these before we can attempt any worthwhile embedded control work. In each example, I have included the DSD construct so that you can become familiar with them. If you are a real engineer you will be of a practical mind, and will need pictures. *(It is well known that software engineers are different, so I will not count these. Anyone that can follow Structured English or JSP in quantity must be different)*

if (expression) statement;

Is one of C's selection statements, (or conditional statements) and allows an expression to be tested for TRUE.



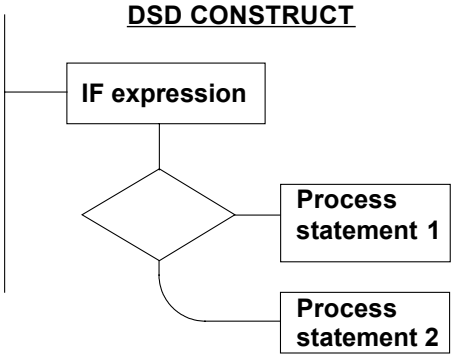
False equates to zero in C. If it is true (NON-ZERO) then the statement following is executed, otherwise execution continues with the next line. In the DSD the expression should contain the condition you are testing in plain English. The process statement will contain the process required if the condition is met.

e.g. **`if(number==0) string="zero.";`**
Only allocates the text **`"zero"`** to the variable **`string`** if **`number`** equals zero.

if(expression) statement1;

else statement 2;

Is an extension of **if** to allow one of two processes to be executed. IF the expression is true (non-zero) then statement 1 will be executed, ELSE (if false (zero)) statement 2 will be executed.



In the DSD the **`expression`** should state the condition being tested.
In **`process statement 1`** will be written the process to be executed if the condition is true.
In **`process statement 2`** will be written the process to be executed if it is not true.
This means that at least one process will be executed.

e.g
```
if(number==0) string="zero";
else string="not zero";
```

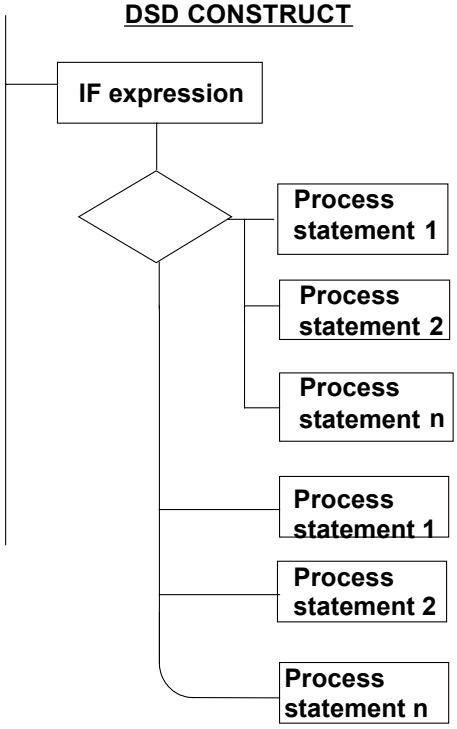
Allocates the text **"`zero`"** to the variable **`string`** only if **`number equals 0,`** else allocates the text **"`not zero`".**

Program flow then continues with the next line.

Note that with DSDs, program control always reverts to the same node as the input. There is only one door, which is both entry and exit. This is what makes it such a structured programming tool. It is very difficult, if not impossible, to write unstructured programs. (Although there is a GO TO symbol, but you will not need that will you!)

CODE BLOCKS

C allows a block of code to be created by linking two or more statements together. As an example, using IF, this may or may not include an ELSE condition as required, and looks like this :-




```
if(expression)
  {
  statement1;
  statement2;
  .
  .
  statement n;
  }
  else
  {
  statement1;
  statement2;
  .
  .
  statementn;
  }
```

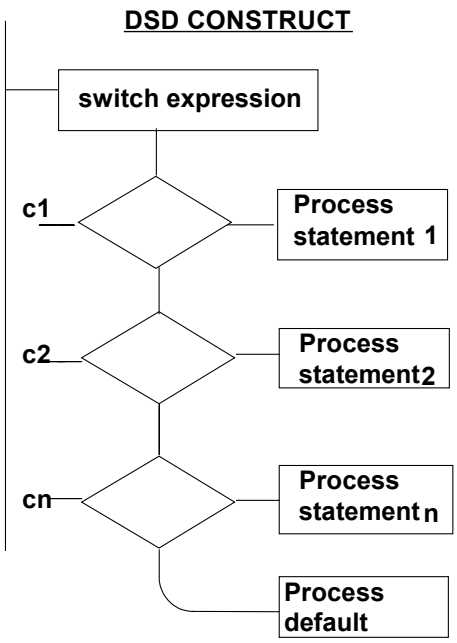
Note how the braces enclose the two blocks of code. These braces can be placed at any point in a C program to enclose a code block. In this case they differentiate between the IF and the ELSE code block. IF the expression is TRUE the top code block will be executed ELSE the bottom code block will be followed.

If you need to ask a series of questions about one variable then multiple IF structures can be formed, with an optional ELSE on the end for error trapping etc. (Often needed for such things as keyboard handling or other input devices.)
The structure could be formed with a series of IF statements and a final ELSE. In C though there is a better structure for this called the SWITCH structure.

SWITCH STRUCTURE

The switch structure tests for INTEGER values only and executes one of many possible processes. The coding format would look like this :-

```
switch(expression)
{
   case constant1:
      statement;
      break;
   case constant2:
      statement;
      break;
      .
      .
   case constantn:
      statement;
      break;
   default:
      statement;
      break;
}
```



The expression is evaluated, and must be an integer value. If a matching CASE is found, where the expression equals the constant, then that process is executed, ELSE the default process is followed (if there is one).
The **break** statement forces the switch structure to terminate and exit to the next C program structure if that case is found.

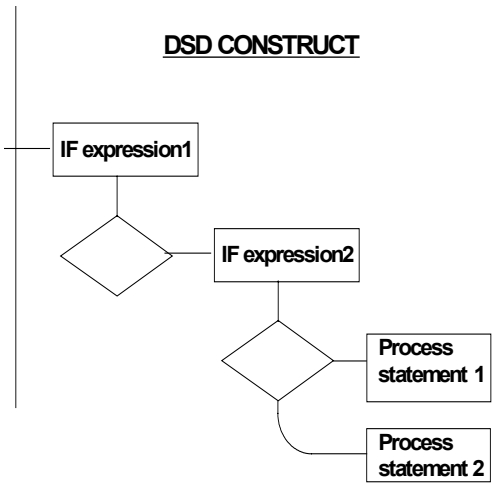
NESTING IF STATEMENTS

ANSI C allows IF statements to be nested up to at least FIFTEEN deep!

This kind of decision making is a common requirement, but not usually to that extent. As you can imagine, the DSD would take up a large width of page if you extend more than three deep.

The format for a two deep nesting, with else, would be :-

```
if(expression1)
   if(expression2)
   statement1;
   else
   statement2;
```



The DSD quite clearly shows the flow of this example. With the second question only asked if expression 1 is true, and then either one of the two processes will be executed depending on the condition of expression 2. The C coding is not quite so clear on this point.
The first expression may also have an ELSE condition and this would appear on the end.

Note how the text is indented to show the nesting. This form of structure rapidly gets out of hand. It often becomes difficult to see which relates to what. It is also possible to have ELSE conditions after each IF. The question arises as to which ELSE relates to which IF? The DSD will help you here. The golden rule here is that the last ELSE relates to the last IF that does not have an ELSE! You followed that of course?

It pays to keep your structures simpler than this, if possible, and there will usually be another way.

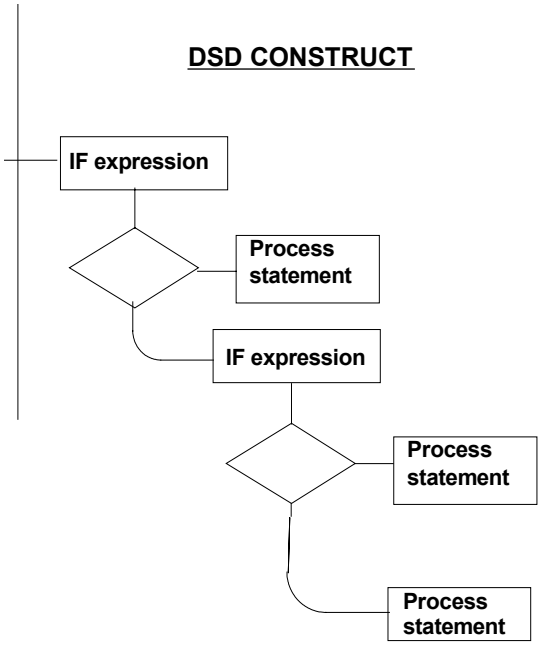
One possible way out is to use the **if-else-if ladder**, so called because of its appearance. This allows a nested IF to have as its target another IF and so on.

The written structure might look like this :-

```
if(expression) statement;
else
     if(expression) statement;
     else
          if(expression) statement;
.
          else
          statement;
```

Note how the text is indented to show the nesting. The conditions are evaluated from the top downwards. As soon as a true condition is found the statement associated with it is executed and the rest of the ladder bypassed.

If none of the conditions are met then the final else statement (if there is one) is executed.
This is best shown by referring to the DSD structure:-



Because of the potential for indentations getting out of hand, the format is often written thus:-

```
if(expression)
statement;
else
if(expression)
statement;
.
.
else
statement;
```

It should be quite clear now what a saviour DSDs are in tracing the flow of program control.

The importance of good program planning cannot be stressed enough, and DSDs allow you to visually plan and test your program structure BEFORE you waste time coding it.

Once you persevere with DSDs and use them a few times, you will NEVER go back to flowcharts or any other form of planning. Not willingly anyway.

At this stage, it is worth mentioning that there is an alternative method of doing IF-ELSE in C, which is easier to type and executes quicker.
The conditional operator, (**? :**), compiles to shorter code than IF-ELSE and although it looks pretty weird it is worth using for this alone.

CONDITIONAL OPERATOR

The format of this operator is:-

```
relation ? truestatement : falsestatement;
```

relation is any relational test e.g. size <=255,

truestatement is any valid C statement,

falsestatement is any valid C statement.

Although not strictly necessary, it is a good idea to surround each of these with parentheses. This helps separate each of the three parts so that you can see them easier.

Example :-

```
(limit==255) ? (limit-=1) : (limit+=1);
```

Asks if the variable limit is equal to 255. If it is it decrements it, if it is not it increments it. This is the same as :-

```
if (limit==255)
     {limit-=1;}
else
     {limit+=1;}
```

The DSD construct would also be identical. As you can see there is less likelihood of you leaving out semicolons and it is shorter.

LOOP STRUCTURES

Invariably a program requires at least one loop. In a control program, this would usually be a continuous loop to repeat the control cycle.

The rest of the program, and the functions, will also use various forms of loop to serve some function. This is where the program structure usually goes wrong because you lose track of where you are in the program, and your flowchart/structure method has become too unwieldy to be of use. The DSD comes to the rescue again, because loops show up as loops, always in a clockwise direction. The direction is always shown by an arrow on the loop exit. If the loop has a conditional test in it, it will be shown by a capacitor type symbol. This ensures that you are aware of whether the test is being done at the head, middle or tail of the loop. This can make a big difference.

C allows three loop constructs :-

- **for** loop - repeats the loop a set number of times in specified steps up or down. BASIC users will be familiar with this as the FOR-NEXT structure, only in C you do not need NEXT.

- **while** loop - repeats the processes within the loop while a condition is true. The condition is tested at the head of the loop and so the processes in the loop may not be done at all if the condition is true on the first pass.

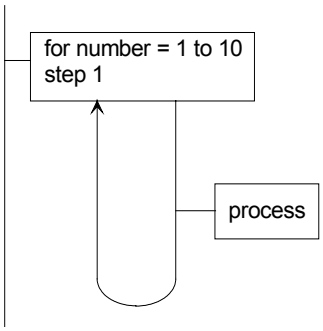
- **do** loop - does the processes while the condition is true, with the test at the tail of the loop. The process within the loop will thus be done at least once. BASIC users will know this as the REPEAT-UNTIL loop.

FOR LOOP

General form**:-**

```
for(initialisation; conditional test; increment) statement;
```

The **initialisation** gives an initial value to the loop control variable, starting the loop count at that value.



The **conditional test** is the reason for continuing the loop, with the loop exiting if the test fails. This is tested at the head of the loop.

The **increment** is the step size that the loop takes and may be positive (step up) or negative (step down), and is done at the tail of the loop. In the DSD the **step** may be omitted if it is one and positive.

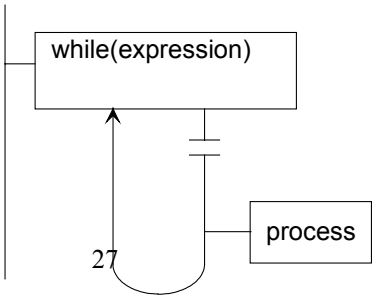
Example

```
for(num=1; num<11; num++)
{
process;
}
```

Repeats the **process** ten times with the variable **num** stepping from 1 to 10 in unit steps. Notice how **num** is incremented with the ++ operator. A step down loop can be constructed in a similar fashion using the - - operator.
The statement can equally well be a single statement on the end of the **for** line, if it is not too complicated.

WHILE LOOP

General form :-



```
while(expression) statement;
```

The expression is checked for true at the head of the loop and the statement executed only if it is true.

The DSD shows the structure better, with the capacitor type symbol showing the test point. Note the loop direction shown by the arrow head at the tail of the loop.

The statement (process) can be one statement or a code block between braces.

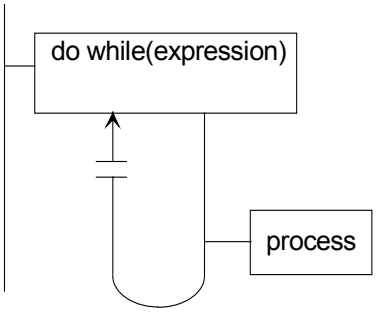
This loop can also be used as a DO-FOREVER (continuous) loop by making sure that the expression is always true.
e.g. **`while (1)`**

ALL control programs will have this because they will always be in control!

DO LOOP

General form:-

```
do{
   statements;
   }
while(expression);
```



Repeats the statements inside the braces while the expression is true, with the test at the tail of the loop (as shown on the DSD).

If only a single statement is required then the curly braces are not needed to show the code block (there will not be one!).

The process statements will be executed at least once because the escape condition is not tested until the end of the loop. Again the DSD shows this quite nicely with the capacitor type symbol.

BREAKING OUT OF A LOOP

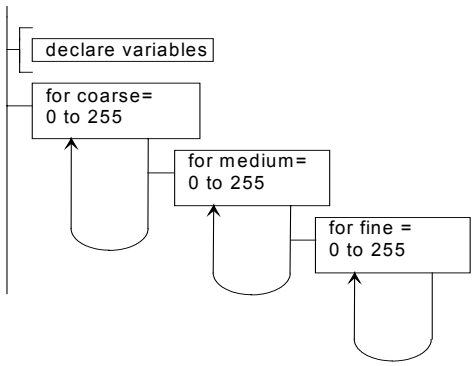
Sometimes you need to exit a loop outside of the control of the loop variable. This can be done in any three of the loops using the **`break`** statement. This causes the loop to terminate immediately the break is encountered. The use of break makes a program more difficult to follow but sometimes it is the only way to solve a problem. It is shown on a DSD by the break symbol :-

NESTED LOOPS

Nesting loops inside one another is a very common requirement. ANSI C specifies that loops can be nested up to FIFTEEN deep, although most compilers allow a lot more than this.

For instance, it is commonplace to generate time delays in control work using nested FOR loops as a time waste routine, particularly if the on-board timers are in use!

The example below show how a three nested loop time waste routine would appear in C, structured as a code block and set for maximum counts for an eight bit number (type char). The DSD shows the nesting well, as does the nesting of the C code :-




```
   {
   char coarse,

coarse==255;

   medium==255;

   fine==255; fine++)
   }
```

```
medium, fine;
   for(coarse=0;
coarse++)
   for(medium=0;
medium++)
   for(fine=0;
```

Note how the time waste variables are declared as type **`char'** for the 8-bit storage in the C code.
In practice, this would usually be defined as a function, and the time made adjustable by taking in the delay as a parameter.
In microcontrollers, it would also be more efficient if you could specify these variables for use in registers, instead of RAM. Especially with the AVR, which has 32 registers for this purpose. IAR 'C' will try to do this automatically if possible or you can force it to if you wish. (Using the **`register`** type.)

ARRAYS AND POINTERS

An array is a series of storage elements for the same type of data and is more commonly used as a look-up table in assembly language work. Arrays can be in several dimensions, but any more than one or two dimensions with micro controllers will consume more memory than you may have available.

To use arrays you must go through several stages :-

1. DECLARATION

They must be defined as to the type of data to be stored and the size:-

```
   type variable[size];
```

The **`type`** is any valid C type, the **`variable`** is the variable name and the **`size`** is the number of elements of storage needed to house the data.

e.g. **`char table[10]`**; - declares an array called table, to house ten words of 8-bit data.

`char grid[5][7];`- declares a two dimensional array 5 rows by 7 columns.

2. LOAD THE ARRAY WITH DATA

C stores the data in contiguous memory locations starting at the address of the first word of data. With microcontrollers, you would also have the choice of memory types to house the array in, and what the start address would be. This might be in SRAM, ROM or EEPROM. IAR `C' for embedded control allows this by providing an expanded set of keywords to access the special functions. Examples of this will be found in the next chapter. The loading of the array will depend on the target location, with EEPROM data loaded as a separate file in the MCU.

The standard C books often show arrays loaded with calculated data inside a **for** loop. This is fine for PC work in RAM but would be pointless with MCUs because it is very inefficient use of memory. The faster 8-bit RISC micro controllers, such as the PIC or AVR, are also more than capable of calculating such data at a fast speed. Arrays are thus reserved for data that have no mathematical relationship or need to be accessed very fast. Typical examples might be string arrays, or ASCII tables for printing to LCD displays, wave table data for music generation and look up tables for decoding/encoding work, or calibration data.

Arrays can be initialised in a similar fashion to normal variables by adding a value list after the type declaration thus :-

```
type variable[size]={value-list};
```

The value list is the data to be placed in the array separated by commas. The data type must of course match the declared type.

3. ACCESS

The array, to be used as a look-up table, needs to be accessed either using some kind of pointer, or directly by specifying the array cell.

To access the array elements directly you merely specify the cell address (beware the address starts from ZERO, so a ten element array will have addresses 0 - 9) e.g. If an array called **`table`** is set up and contains 12 pieces of data then **`table[11]`** used anywhere will produce the data in the last cell.

To access an array using a pointer is quite common in embedded applications and involves the use of a pointer which can be indexed to take the data out (or put it in) sequentially. In assembly language, this is usually achieved using indexed addressing, or indirect addressing.

In C, a pointer variable is employed. Two operators are employed for use with pointers.

These are :-

`&` **Address operator**

`*` **Dereferencing operator**

These need some explaining because they are very confusing, even to the hardened C programmer.

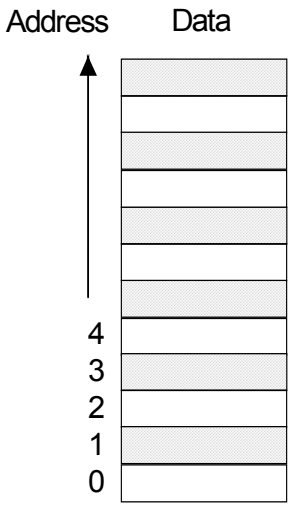
The pointer will be a variable used as the index and is defined in a special way. C knows it is a pointer variable because it starts with the asterisk:-

```
type *pointer_name;
```

Where type can be one from the list shown earlier. With most MCUs the types are usually extended to describe the nature of the memory it will be pointing to. (e.g. **`flash`** for flash ROM). We will meet this later.

The address of where the pointer is looking is given by :-

```
&variable
```



These two operators therefore provide you with a very flexible way of accessing arrays.

It is usual practice to start your pointer names with a "p", although there is no written law.

e.g **`int *pLot;`**

This defines a pointer variable named pLot as an integer. Note the capital L after the p to show up the "p". This makes the name standout as a pointer variable. The * is not part of the variable name, it is there to tell C that you are defining it as a pointer. That is why it is worth the effort to make it stand out.

If we now define a variable to store the data, and initialise it with say 37:-

```
char lot = 37;
```

You can now get to this data two ways :-

1. As a normal variable
   e.g **`printf(lot);`**

or

2. By using the pointer linked to the variable:-

e.g.

```
pLot=&lot   /*links the pointer to the data */
printf(*pLot)/*prints the value of lot (37) */
```

Pointers and arrays are one of the most confusing areas of C and take some explaining. It is recommended that you consult a C reference book related to PCs and apply the same principles, if you wish to use advanced features such as pointers.

In fact this chapter should be treated as a basic reference section. Most learning will be achieved by working through the examples in the next chapters and you trying your own examples. The debugger will tell you if they work!

STRUCTURES

Arrays can only hold data of the same type. This is very often the case with embedded work. For systems work, a database is often needed containing data of varying types.

Structures and unions were created for this application. Although it is not often required for embedded control, it can be handy to reduce memory space, if you have lots of single bit flags around the place. These can be placed in ***bitfields*** and accessed individually. It is also useful for sending data to output devices, such as multi-line LCD text displays, for various types of data.

WHAT IS A STRUCTURE?

A structure is a conglomerate or aggregate data type, which is composed of two or more related elements, not necessarily of the same size (type). Structures are defined using the general form:

```
struct tag_name
{
type element1;
type element2;
…
…
}
variable_list;
```

The keyword **`struct`** tells the compiler that a structure is being defined. The type is any valid C type such as char, int etc.,
Note - unlike arrays, the types may be all different.

`tag_name` is the type name of the structure. It is NOT a variable name, more of a self defined type.

`variable_list;` is where the variables of the structure are defined. The use of either *`tag_name`* or `variable_list;` is optional but one must be present.

The elements of the structure are normally called *fields* (or sometimes *members*).
As an example, a system might have to display the results of a data logger program on a LCD display, or some other output device. The display is to be updated 100 times a day, over a planned period of 100 years (built in centenary bug!), with the voltage and current having a range of integer steps from ±500 V/A. The variables to be displayed might be: -

Time - in 24 hr clock format e.g. 21:30, perhaps stored as hours and minutes, in two (char) bytes.

Date - in UK format e.g. 29/09/1999, perhaps stored as date, month and year in two (char) bytes and an int word.

Data - Integers, say voltage and current, in two words (signed int).

The structure would thus have the following members and size: -

Name	Type	Size (bytes)
hours	unsigned char	1
minutes	unsigned char	1
date	unsigned char	1
month	unsigned char	1
year	unsigned int	2
voltage	signed int	2
current	signed int	2

This will obviously require a fair amount of RAM space, or it may need to be housed in EEPROM to make it non-volatile. Speed of access is not a problem here, so this would be feasible if a large enough EEPROM was available, either internally or externally.

The structure would now be declared in this fashion: -

```
struct datalog
{
char hours[24];
char minutes[60];
char month[12];
unsigned int year[100];
signed int voltage[1000];
signed int current[1000];
}
data;
```

The structure is now called **data** and can have values assigned to the various elements by using the dot operator (**.**).

Each element is an array and can be accessed as such for read and write.

For example the assign the year 2000 to **year**, in the first element, the line would be :-

```
data.year[0] = 1999;
```

Note that the addresses start at zero.
To access the same date would be done, say using a predefined printf function for the LCD, by :-

```
printf("%d",data.year[0]);
```

This should display 1999 on the LCD in a previously sent cursor position.

To get data into the structure from a keyboard, for instance, can be done using the `scanf` function and the **`&`** operator.

It might look something like this :-

```
scanf("%u",&data.year[0]);
```

This would place the returned value of the scanf function in the first position in that field array. Note, the & goes in front of the structure name, NOT the field name.

You can also access structures via pointers. In this case you must use the arrow operator (->), not the dot (.) operator to access them, and declare the pointer variable as well. Again, C books for PCs will give you more details of how to do this.

For instance our datalog structure might now be declared thus :-

```
struct datalog
{
char hours[24];
char minutes[60];
char month[12];
unsigned int year[100];
signed int voltage[1000];
signed int current[1000];
}
data,*Pindex;
```

The addition of the pointer variable Pindex (note the convention of using capital P in front of pointer variables) means that you can use pointers in the following manner :-

```
Pindex=&data;
```

Assigns the address of data to the pointer variable Pindex

```
Pindex->year=2000;
```

Assigns 2000 to the year element pointed to by pointer Pindex.

BITFIELDS

Bitfields save memory space (but execute slower) by splitting up bytes or words into bits. In MCUs this can be very useful, if you can afford the drop in speed. Using bitfields, you can access, by name, one or more bits within a byte or word. In this way, for instance, you could store two nibble variables in one byte, saving a byte of memory.

Bitfields are defined by:-

```
type name:size;
```

Where `type` is either signed or unsigned char, short, long or int,
`name` is any valid variable name, and
`size` is the number of bits needed for storage.

The name and size must be separated by the colon (:).
If signed types are specified, the top bit is taken as the sign flag.

Bitfield variables are packed in elements of the specified type starting at the LSB position. Not all bits need be used and you can mix bitfields with normal variables in a structure.
You cannot obtain the address (&) of the variable though because the smallest addressable unit of memory is the byte.

As an example, this structure definition houses a title of 10 characters and two nibbles called left and right.

```
struct decades
{
char title[10];
unsigned right:4;
unsigned left:4;
}
nibbles;
```

This particular example is ideal for use with multi-digit display variables, where one nibble is used per decade.

TYPECASTING

You may be typecast, I am certainly, but C is not sure! When you do mixed calculations you can never be sure of the size of the result**.** For instance, if you add two **`char`** variables together the result may be larger than one byte resulting in the need for **`int`** type storage.

A C typecast temporarily changes the type of a variable for you prior to some process. The format is:-

`(type)value`

`type` is any valid C type

`value` is any variable, literal (number) or constant**.**

This can be placed anywhere in the program and ensures that the type is the same as the rest, for the process that follows the typecast only. All other uses elsewhere return to the original type declared.

Typecasting avoids some confusing answers that can occasionally result from mixed calculations. This normally results in lost accuracy or incorrect polarity with signed numbers.

3. THE IAR `C' DEVELOPMENT ENVIRONMENT

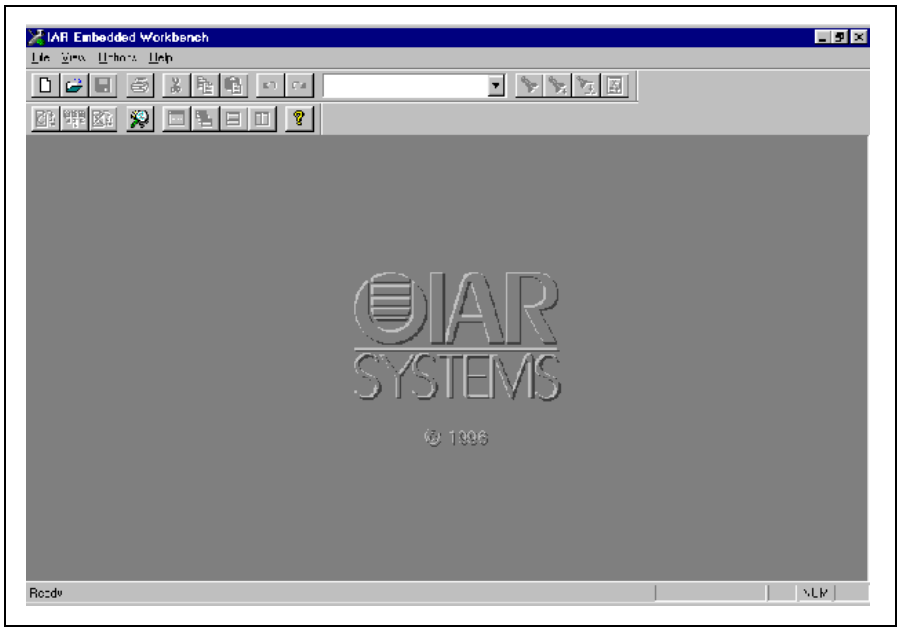
The time has come to try the program structures and keywords described in the previous chapter. Before we start programming and building up valuable experience, we must learn to drive the development system software. This chapter takes you through the software from beginning to end using a simple control example.

SOFTWARE OVERVIEW

IAR`C' is a complete integrated development system providing the following facilities :-

- EMBEDDED WORKBENCH Control shell providing access to all the tools, I/O and project management.
- Powerful, syntax highlighting editor to type your source code.
- Optimising C-compiler to build your program efficiently.
- Powerful Macro Assembler, for assembly language work.
- XLINK linker to link code from C compiler and Assembler to produce the target object code.
- XLIB librarian to allow you to build your own libraries of functions.
- C-SPY debugger to simulate your programs.

Locate the IAR Workbench Demo from your program selection, you will also see some *readme* files which are worth reading first. They tell you what version you have and what the limitations are. Click the mouse on the spanner/screwdriver icon when you are ready, you should see a screen similar to the one shown below:-

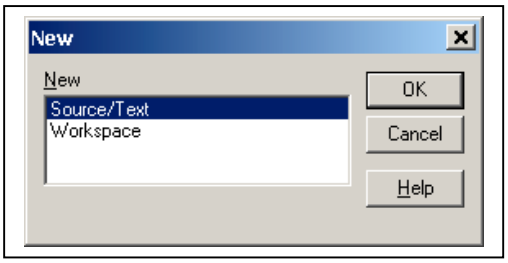


Experiment with the various icons by moving the mouse cursor over them and let the help balloons show what each does. Most of them are reasonably obvious such as new, print, save etc.,
At this stage, it is probably worth playing with the text selections as well to become conversant with where things are. The important ones will be described as we progress.

We are now ready to start our first project.

Select NEW and a window will appear like the one shown below.

Click on the Workspace item and select OK (or double click). Give the Workspace a name, e.g. alarm.eww, and a new window will now appear like this:-



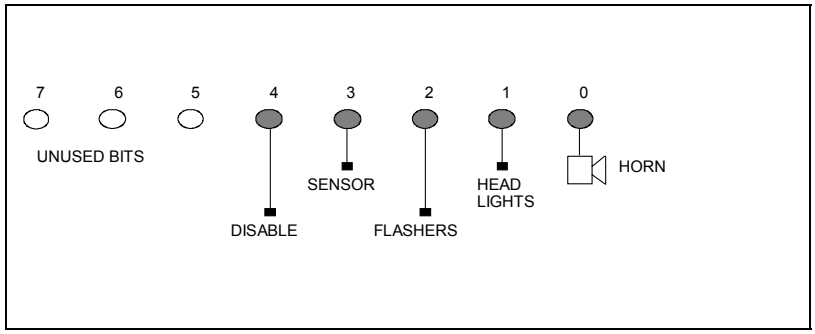
This is a workspace, and we need to create a Project. So, go to Project Menu > Create New Project and create a Project (alarm.ewp). This will appear in the Workspace Window.

If you chose to install the AVR version you can type in the example at the start of the next chapter, which is the same program adapted for the ATmega8515.

The project creates space for a group of files relating to both likely versions of your project. One for debugging purposes, which will have extra code so that you can see what is happening, and the final one with this removed ready for release. This is probably the most common arrangement. The drop down at the top will allow you to swap between Debug and Release versions. We now need to type the C source code using the editor before we can add it to the program group we have just created. The Project Group arrangement makes it easy to keep all your modules together which relate to the one project. You can even have several projects sharing the same modules. The linker can then link them together to build the final code. In our first example, we only need the one module so the approach is wasted here, but normal projects benefit greatly from this method of working. To start typing the source code select **New** again and this time select the Source/Text option. A blank editor window will appear headed **`Untitled1`** and you are ready to start typing.

Give it a name (alarm.c) and right click on the Project in the Workspace window, or select Project Menu > Add Files to put the file into the Project.

Project Code
The example uses a simple circuit connected to Port 1 (P1) of the target AT89S8252 MCU. We need not concern ourselves with the processor yet, only the port allocation is of interest:-



The output devices are all active high, that is a `1' turns them on and a `0' turns them off. The sensor is a tilt switch, which provides a logic `1' when the car is tilted. The disable input is from an ARM/DISABLE arrangement, a keyswitch or other type of facility, and is high for the ARM condition. This simple hardware configuration is now going to be programmed using the approach taken by all other examples in this book. This sets the pattern to follow, and also shows how the Embedded Workbench is used and how it fits in with the real world of embedded control.

Our first task is to devise a main specification for the system.
I have chosen the following :-

MAIN SPECIFICATION

1. Initialise port 1 - this sets the direction of data travel IN or OUT.
2. If ARM AND SENSOR then ALARM
ELSE clear port 1.
3. Go to 2.

This forms a continuous loop with an IF structure. The alarm only goes off if the arm switch is on AND the sensor is triggered. What it does not show is what the ALARM should do. The easiest way to do this is to make it a function, with its own specification :-

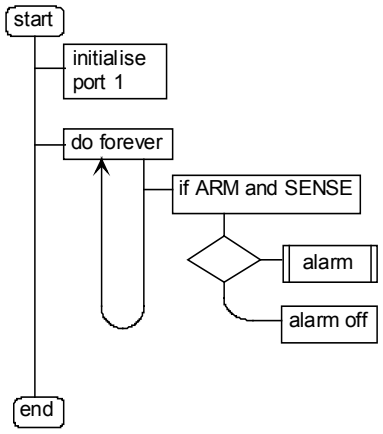
ALARM FUNCTION SPECIFICATION

1. *Turn ON horn,*
2. *Turn OFF flashers and headlights,*
3. *Wait a bit (0.5 second or so),*
4. *Turn OFF horn,*
5. *Turn ON flashers and headlights,*
6. *Wait a bit (0.5 second or so).*

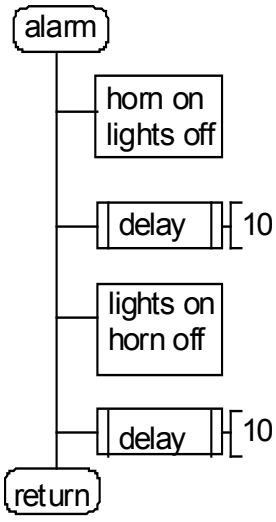
This will recycle all the time the sensor is tilted, causing the horn and lights to alternate with half second intervals. A proper system would latch of course and reset on time out (your job later perhaps!). The next problem is how do you generate a time waste delay in C. There is no wait instruction as found in some higher level languages. We will need to create another function.

So even in this simple example, we have three functions.

The specifications can now be converted into DSDs so that we can check the structure works:-

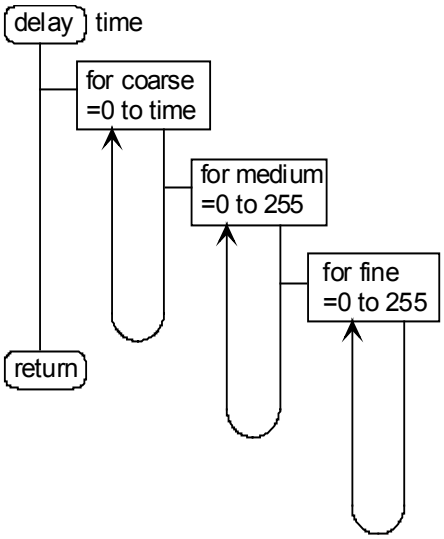


The main program initialises port 1 of the MCU and then goes into a continuous loop testing the ARM and SENSE inputs. IF both are high THEN the alarm function is executed ELSE the alarm is silenced by keeping all lights and horn off.
This will clearly work and now we must plan the two functions in the same manner. The port initialisation is an important part for embedded control, because, without it the port will not operate correctly. This is one area where your programs can differ depending on the target processor. For the 8051 type MCUs, port 1 is a bi-directional port capable of driving four TTL loads as an output. To set a bit as input you write a '1' to the bit(s) required which enables the pull-ups. It is most important that you do not disturb these bits once set. If you write a '0' to the bit it will stay low and the input will not respond.
If you are using a more modern MCU, with high current drive capability (20 mA), such as the PIC or AVR, you will need to write to a separate data direction register (examples later).



The alarm function is a simple two step sequence with a short delay in between each, to time the beep/flash period.

The delay function has been made versatile in that it allows a delay parameter to be passed into it, so that the delay can be modified on each function call. This is not necessary here, but has been done as an example of how to pass parameters into functions.
The DSD shows this by a square bracket symbol on the end of the double-ended function box. In this case, I have chosen the number 10 purely as a convenient round number.



With the simple approach we are using here, shown in the delay DSD, the delay time is uncertain, unless you calculate the number of clock cycles used for it. This means inspecting the object code after compilation and doing some sums. This rather defeats the object of using C.

To keep it simple to start with, the three nested FOR loop is used, with the outer coarse loop used as the time adjustment parameter. The middle and inner loop is run at the maximum **char** limit of 255 (0xFF). This provides a workable solution for approximate delays, and can be adjusted to suit on compilation/test. Three nested loops are required for most MCUs to give one or more seconds delay. In the next chapter, the use of MCU peripherals such as timers are investigated, which is a more predictable method.

With the structure planned, and tested by dry running the DSDs, we can now write the C source code. For most examples this will consist of the following sections :-

- **Title block** - to explain what the program does, who it was written by, when, the version number, and the target processor.
- **Pre-processor directives** - to include the special function register names, and intrinsic functions for the target processor and to tell C that they will be used.
- **Function prototypes** - to inform C how much space to reserve for any parameters passed into them and returned.
- **Global variable declarations** - to tell C what space to reserve for them. In this case we do not need any global variables.
- **Main program definition** - the main source code, calling up the other functions.
- **Function definitions** - containing local variable declarations and the source code.

The following program covers the above topics for the alarm example and may be typed in and saved as a source file with a .c extension. Each line is commented to explain the purpose.

The pre-processor directives will vary with the target processor family and match the Atmel 89S family of 8051 derivatives in this example.

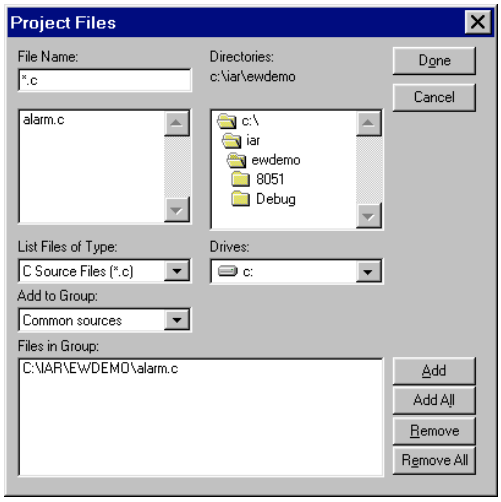
```
/*****************************************
 simple car alarm example
 for Atmel AT89S8252 (8051) microcontroller
 version 1.0 PJS Nov 1998.
*****************************************/

#pragma language=extended  /*use special functions*/
```

```
#include <io_a89.h>       /*include header file for
                          AT89S8252 */
#define TRUE (1)    /*CONSTANT for do forever loop*/
void alarm(void);
void delay(char);   /*function prototypes     */

void main(void)             /*main program    */
 {
 P1=0x18;                   /*initialise port 1*/

    while(TRUE)            /*do forever       */

    {
    if (P1 & 0x18)         /*if ARM and SENSE then*/
    alarm();               /*call alarm function  */
    else P1=0x18;          /*else disable outputs*/
    }
 }

/*********** ALARM FUNCTION ********************/

   void alarm(void)

    {
    P1=0x1A; /*headlights on, horn and flashers off*/
    delay(10); /*wait a bit */
    P1=0x1D; /*headlights off,horn and flashers on*/
    delay(10); /*wait a bit */
    }



/***********TIME DELAY FUNCTION****************/
void delay(char time)
 {
 char coarse, medium, fine;
    for (coarse=0; coarse!=time; coarse++)
    {
        for (medium=0; medium!=255; medium++)
        {
            for (fine=0; fine!=255; fine++)
            {
            }
        }
    }
 }
```

Do not try to compile this yet, we have some more work to do first. Ensure that you have saved this correctly. It must have a .c extension. You may noticed that the editor automatically highlights certain types of word to ease readability. If this does not appear, you have not saved the file with the correct extension. *(Bitter experience!)* The include file may be inspected by loading it in on top of your C source file. It is called `<io_a89.h>` and it is worth printing it to inspect the contents. You will find the names and addresses of all the special function registers for the MCU.

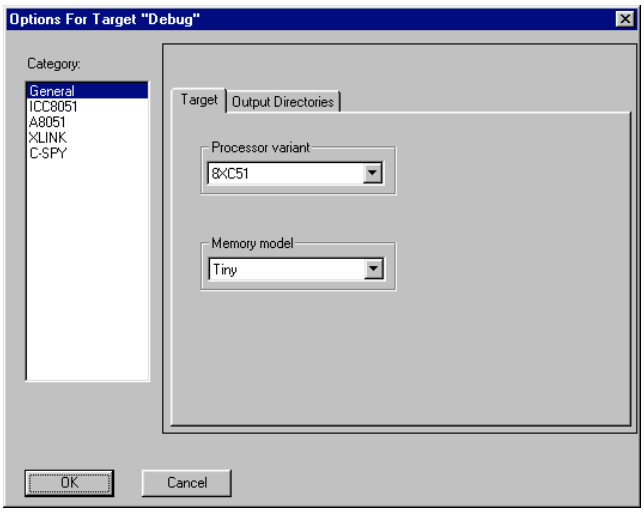
Having saved the file, you can now add it to your project file. This is done by :-

- Select the **debug** target in the project window,
- Select **project** and **files** from the pull down menu and a window should be displayed similar to the one shown overleaf. Select the name of the file to add to your project (alarm.c) and click the **add** box and finish by clicking the **done** box.

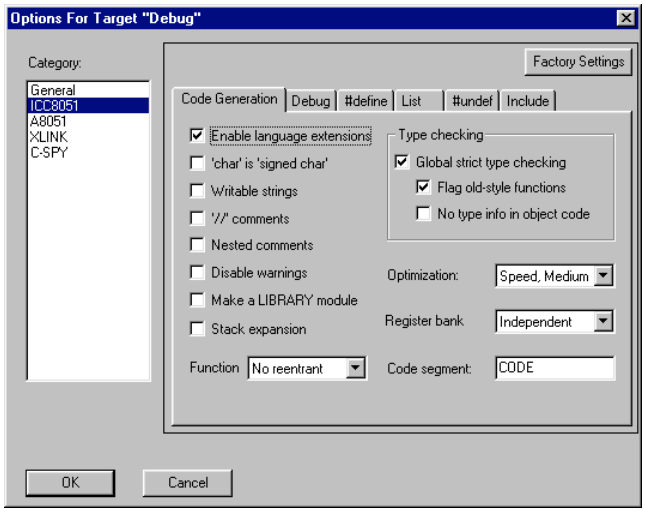


This window sets up the options for five areas. The general one sets up the target processor and the memory model. This should read 8xC51 and the tiny memory model.

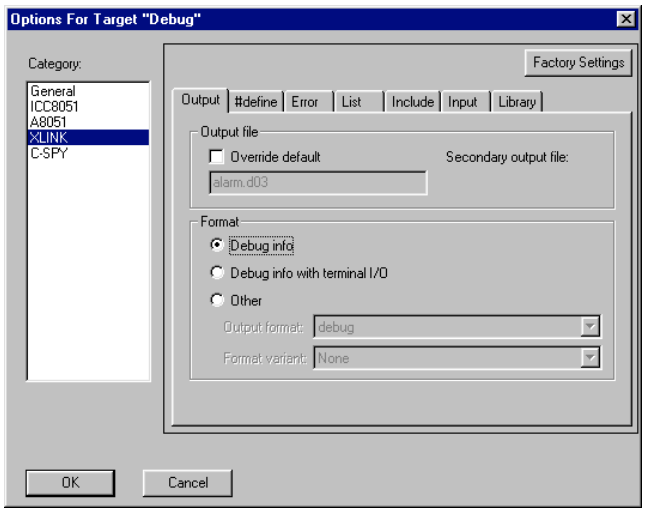
The project window should now have the file installed, click the + sign to expand the path and the file should be displayed. The project is now ready to build. Before compilation takes place the project options must be set up. This tells the compiler which target processor to compile for and what output options to use.
Select options from the project menu. This displays a window like this :-



The ICC8051 window sets up the compiler options. These should default OK but should look like this :-



The various tick boxes allow you to change the optimisation and other preferences.
The A8051 window sets up the assembler options, which we are not using in C . Although, it is a very powerful macro-assembler and ideal for the same approach in Assembly Language. It is also useful for fast bits of code to be included with the C source files.
The Assembler works in exactly the same way as the C compiler.

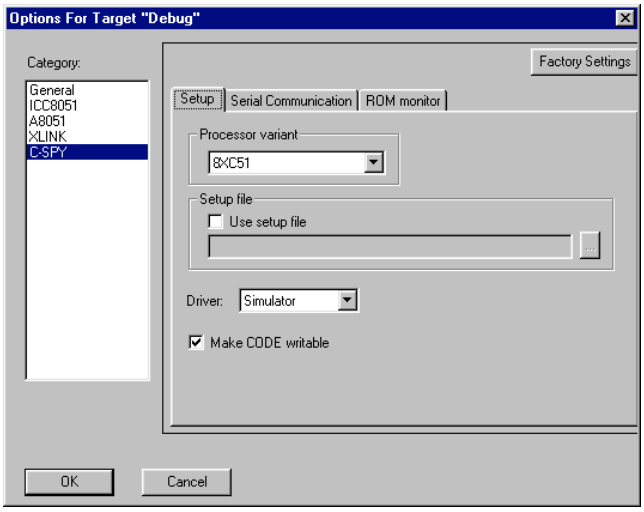


You can use Embedded Workbench as either system, or both, making a very versatile development system indeed. If you plan to include bits of assembly language in the C code then you will need to stick to a prescribed layout detailed in the manuals. The linker will then build a complete set of object code from both types of source code.

The XLINK window sets up the linker options and tells the system what output file to use. Again the default options should be OK but will appear thus:-

The file name should have a .d03 extension for the debugger/ simulator C-SPY format.

The CSPY window sets up the simulator options.
The options here dictate where the object code is sent. The driver should display **simulator** to supply CSPY with the object file in the correct format.

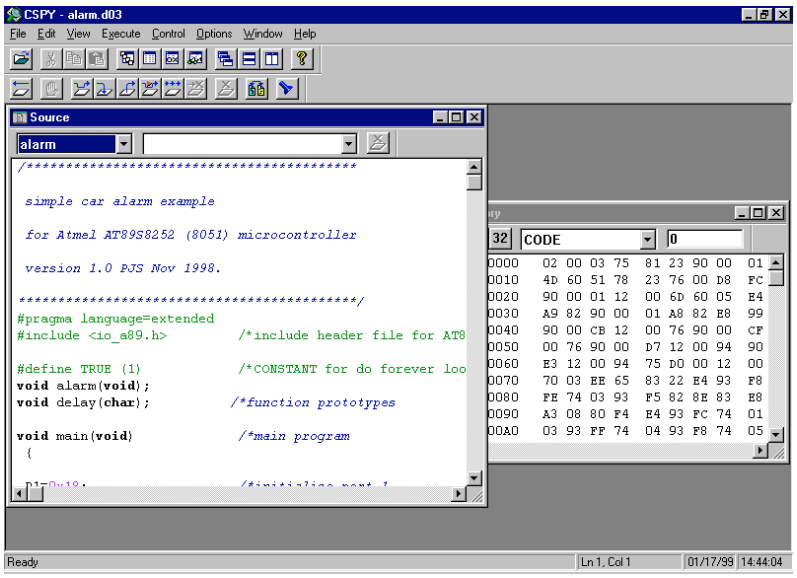


The set up is now complete and you are ready to build your first project. Select the **OK** box, followed by the project window. Now select **build all** or **make** from the projects menu. **Build all** rebuilds all files, whether they have changed or not, **make** only rebuilds those files that have changed. **Compile** only compiles the code but does not link it with other files. This is useful to check for local errors before you integrate your project with any other files.

A message window will display and, hopefully, it will show no errors. If any errors or warnings are displayed tile the windows vertical and click on the error line. The cursor will move to the source code line causing the problem and the error message will give you some idea of what is wrong. It will be a typing error!

Warnings are not fatal, in that they may not stop compilation and linking taking place. They should be taken seriously though because they are an indication that something may be wrong.

Errors are fatal and will normally stop the compilation and linking process. C may not build a file unless it is syntactically correct (spelt right and valid), unless you tell it to of course! If a whole string of errors are reported, it is usually because you have left out a brace (check that opening braces are matched with closing braces) or placed semi-colons where there



should not be.

Assuming all is well, you can now simulate the program in the debugger C-SPY. Click the C-SPY icon (which is a spanner moving a nut) and the C-SPY start up screen will appear. Click the source window and select your file and the source file will appear in a window as shown here. You may have to resize the window to see enough of the file.

Move the mouse cursor over the various icons to see what they do. The important ones for testing the program are the step icons in the bottom row.

From left to right they serve the following function:-

- **`Reset`** - resets the trace to the start of the program.
- **`Stop`** - stops the trace on a line during Autostep
- **`Step`** - fetches and executes one line at a time
- **`Step into`** - steps into a function, **`step`** now executes one line at a time in the function.
- **`Go out`** - exits a function and returns back the main
- **`Autostep`** - calls up small window that asks for a step time. The program then automatically animates the program at that speed.
- **`Go`** - runs the program at full speed. This is used in conjunction with breakpoints to investigate events at a point in the program.
- **`Go to cursor`** - runs the program to the cursor position.
- **`Toggle breakpoint`** - sets or clears a breakpoint at the current cursor position.
- **`Toggle C/Assembler`** - toggles the display between C source code and Assembly language.
- **`Find`** - searches for a given string and places the cursor on it.

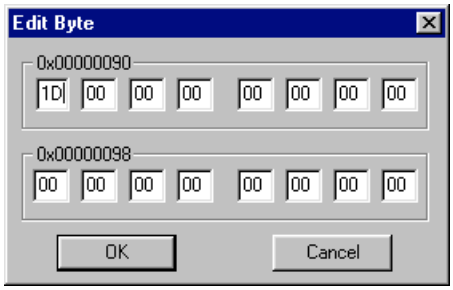
None of these are much good though unless you can see what is going on. This is achieved in one of several fashions. You can open up a watch window to trace variables, and you can view memory to inspect the I/O locations.

For our purposes we need to view port 1 and simulate the action of the sensors. The view windows are selected in the **`window`**
Menu bar.
Select memory and a window will appear as below.

Select the SFR block to view the Special Function Registers, which is where the ports and control registers reside.
Scroll down to memory location 0x00000090 which is the address of port 1 (from the data sheet). Resize the window, making this visible along with the code window, and we are ready to start trying the program.

Single step the program by clicking the step icon and watch the cursor move through the source code. Step into the while loop and simulate no sensor inputs by double clicking on the byte shown in the port 1address location (1D).

A small window will appear allowing you to change the data. Remove the data 1D and replace with 00 to simulate no sensor inputs and click OK.



The program should now continually circulate round the forever loop as you step through the code.
Now put 18 into port 1 in the same manner to simulate the ARM and SENSE input, and continue. The else process should now be entered and you can **`step into`** the alarm function. Single step through this and you will see the two sequences appear at the port address. Note that if you use **`step`** to step into a function, it will run the function at full speed, and then return. This should happen with the delay function. Try it now, the icons will grey out until control returns in a few seconds. This indicates that the delay function works.

C-SPY has proved that the program does what we want.

Congratulations, you have successfully completed your first embedded C program!

4 CHANGING TARGETS

What if you do not want to use an 8051? This deserves a chapter on its own. Every target will require a slightly different approach but this example will give you some idea of the items to change. We will use the preceding example and port it across to the very latest 8-bit RISC MCU from Atmel, the AVR.
We will use the smallest AVR with static RAM (*only devices with RAM can be programmed in C- are there any? Yes, the At90S1200 has no SRAM, only 32 registers which serve as RAM*), the ATtiny2313, a 20-pin version, although the larger ATmega8515, is pin compatible with the 8051, and could be used instead. The 2313 has two ports called port B and port D. We will use port B, which is the equivalent of the 8051 port 1.

When changing targets you may encounter several potential problems :-

- **PORT HARDWARE** – the ports may work differently. This is especially true of the modern MCUs, with high sink/source current drive capability. For instance :

  AVR - each port has three registers! A data direction register, to set the direction of data travel, a port register, for output and a pin register for input. Direction is set in the data direction register by loading each bit with a '0' for input or a '1' for output. The use of separate read and write registers overcomes the timing problems associated with read/modify/write applications (if you use the PIC you will know what I mean).

  PIC - each port has two registers, a tristate register for data direction and a port register for I/O. Direction is set in the tristate register by loading each bit with a '1' for input or a '0' for output.

- **SPECIAL FUNCTIONS** - if you are using port lines which have second functions, such as external interrupts, counter inputs, capture/compare pins etc., it is likely that these will differ. This will require a change of port allocation, which must be reflected in the program.

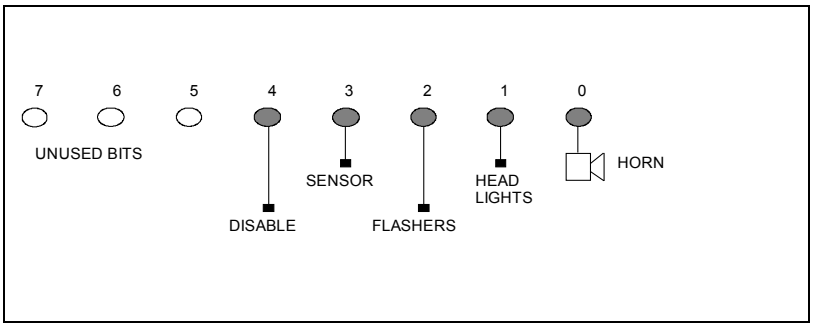
- **MEMORY MODELS** - devices, even the same device, can have a variety of internal or external memory configurations. 'C' must be told which you are using in the project options so that it can place the code in the correct places, set up the stack pointers correctly and generally provide good housekeeping.

- **SPECIAL FUNCTION REGISTERS** - will be different. The include files will need changing to suit the target device. Some devices also need other include files, in addition to the main one, to access the intrinsic functions.

- **RESOURCES** - each MCU will have a different set of peripheral resources. This may require a change in the structure of the program to take advantage of them.

This list sounds daunting, hardly portable is it? Let us try the previous example now and judge how difficult it is. The AVR has a totally different architecture to the Von Neumann CISC 8051, having a true RISC Harvard architecture and very fast single cycle pipelined execution. This would appear to be very different at first sight. It is certainly a lot faster and easier to program. The PIC is also Harvard architecture and semi-RISC.

Just as a reminder, the alarm example used a port allocation as follows :-
This will now be connected to port B of the AVR. The source code now changes as follows



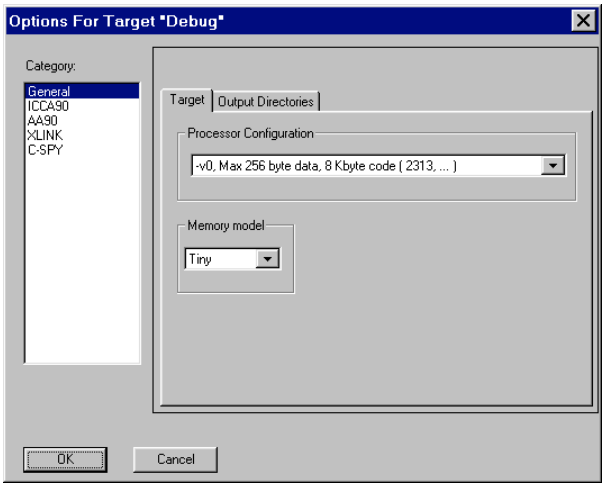

```
/*****************************************
 simple car alarm example
 for Atmel AT90S2313  microcontroller
 version 1.0 PJS Nov 1998.
*****************************************/
#pragma language=extended
#include <io2313.h> /*include header file for
      AT90S2313 */

#define TRUE (1) /*CONSTANT for do forever loop*/
void alarm(void);
void delay(char); /*function prototypes       */

void main(void)   /*main program             */
 {
DDRB=0xE7;       /*initialise port B        */

    while(TRUE)  /*do forever               */
    {
    if (PINB & 0x18) /*if ARM and SENSE then*/
    alarm();         /*call alarm function  */
    else PORTB=0;    /*else disable outputs */
    }
 }

/**********ALARM FUNCTION ****************/
void alarm(void)
{
PORTB=0x1A; /*headlights on, horn and flashers off*/
delay(10);  /*wait a bit                          */
PORTB=0x1D; /*headlights off,horn and flashers on */
delay(10);
}

/**********TIME DELAY FUNCTION*******************/
void delay(char time)
{
 char coarse, medium, fine;
    for (coarse=0; coarse!=time; coarse++)
    {
        for (medium=0; medium!=255; medium++)
        {
            for (fine=0; fine!=255; fine++)
            {
            }
        }
    }
}
```

The changes to the previous program are highlighted in bold text. As you can see, they are very few. The title block has been changed to revise the target processor. The include file has been amended to include the AVR special function register names.
The port initialisation now writes to DDRB (the name for the data direction register of port B) with 1's for outputs and 0's for input (0xE7). Unused bits are set as outputs to avoid them floating about and picking up noise.
The inputs are tested now in the PINB register, which is the input register for port B.
The outputs go to PORTB in the alarm function and the main function.

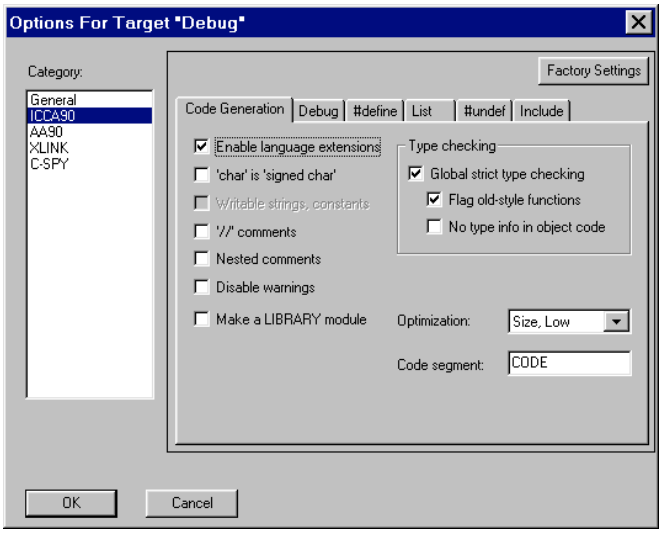
This was not too bad at all was it?

To try this for yourself you will need to install the KickStart version for the AVR range of processors. You will find a few things that are different. Because the AVR has been designed, in conjunction with IAR, to be programmed in 'C', and the AVR has different architecture, the options give you a different choice. The debug output can also be imported into the AVR simulator AVR STUDIO, which paints a 'prettier' picture of the ports than C-SPY, and directly drives the ICE system for hardware emulation.

To give you some idea of the range of options available, the screenshots that follow show how the workbench was set up this time, using the **Project > Options** menu. The general window now calls up the 2313 processor and the tiny memory model.



The C compiler, now called ICCAVR, has a similar set of default options set up. These are OK to use without modification.

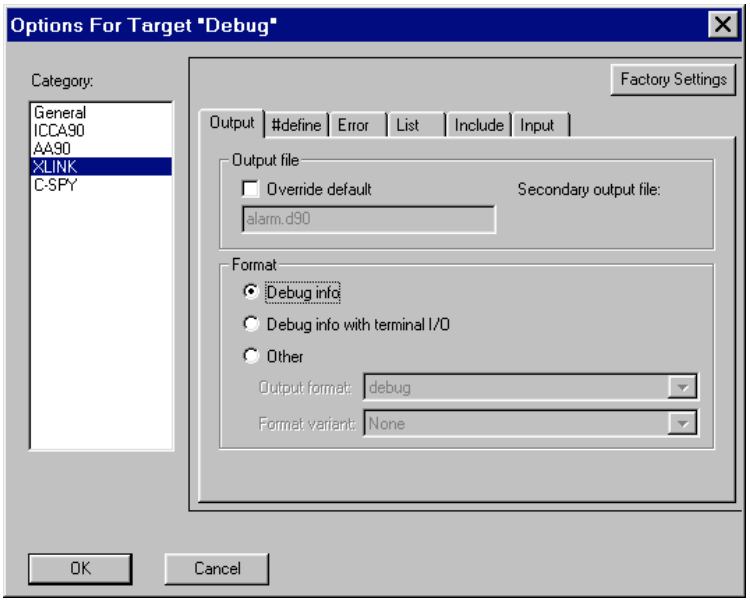


The optimisation can be set up in all versions to suit the method you wish it to use. You can adjust between code size, or speed as a priority, with various stages in between.

If you use C++, you might also like to change the comment style to the // version. Just tick the check box to do this and the compiler will be quite happy. All the versions have this capability and this is one example of how the environment enables you to tailor the package to suit how you want to work.

The linker, still called XLINK, is also set up to generate debug output and will produce a file with a .d90 extension for C-SPY.

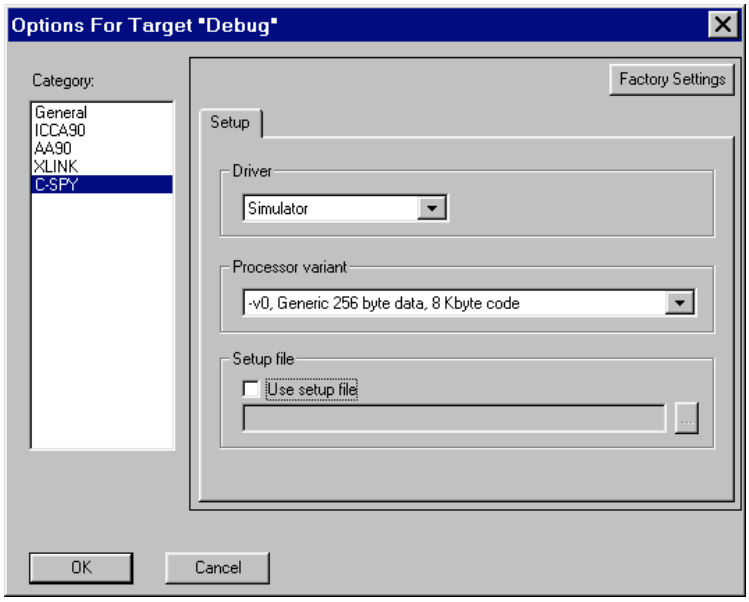
With the Atmel AVR this file can also be used as input to the Atmel debugging tool AVR STUDIO as shown in the screenshots that follow. STUDIO gives a better choice of port display.



In case you were wondering, this is also the place where you can make the linker produce ROM code for the actual device. Instead of selecting the **Debug** output you select **Release** instead. The output format box can then be adjusted to suit the file format needed. A wide range of formats are supplied, to drive most commercial ICE systems and ROM emulators. For the AVR, you would select **Intel extended,** to produce a file with a .a90 extension. This ROM image can then be loaded directly into a programmer and tried without emulation.

The C-SPY window defaults OK, and is set for the simulator driver, telling you the memory configuration of the processor.

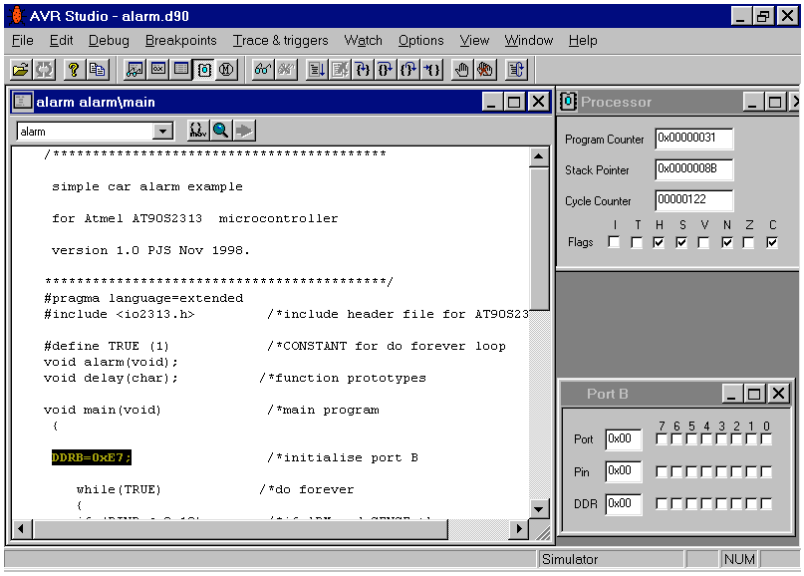
These memory models are supplied for common variants but can be modified to suit any other sizes.



A template is supplied for you to do this, which adds another memory model for you to use. The project is built in the same fashion as before.

C-SPY is used in exactly the same way, with the same user interface. In fact, most things are the same. This makes the Engineers life much easier when moving between processor platforms.

The screenshot below shows the same file loaded into the AVR STUDIO simulator. (Included on CD) As you can see, the ports are much easier to read and modify. This same package also drives the JTAG AVR ICE or Starter board directly.
Note the \include file should be changed to <avr/io.h>.



This is one example of how the workbench integrates with other development systems. This level of integration is a little unusual and if you think this package looks familiar, you would be right. The AVR development tools were produced in conjunction with IAR, as

was the processor. The AVR platform is thus ideal for C work, having been finely tuned by the professionals for C.

5 EXAMPLE APPLICATIONS

This chapter concentrates on using 'C' to do more typical embedded control work. It introduces interrupts, look-up tables and on-board peripherals, the workhorses of MCU applications, to show how they are accessed. The MCU target is mostly the Atmel AVR, rather selfishly, because it is so quick and easy to try the programs. It is what I use most, and it shows how up-to-date MCUs can be programmed in 'C'. Other targets can be used in a similar fashion, only the names of the include file and the special function registers need change, as we saw previously.

EXAMPLE 1 - TIMER ACCESS

Most MCUs have on-board peripherals such as timers. Accessing these can be achieved in one of two fashions, polling and interrupts. This example uses polling methods. The next example uses interrupts. This simple example, shows how a timer can be used to generate pulse width modulated (PWM) waveforms, a very common requirement for embedded controllers. The PWM is software generated, using the timer to time both the on pulse width (t) and the periodic time (T). In this example the duty cycle (δ) is taken from a variable **`delta`** , which would be loaded from another part of the application. Perhaps by some interrupt driven background process.

The AVRs generally have at least two timers, TIMER 0 an 8-bit timer/counter and TIMER1 a 16-bit timer counter. Both have pre-scalars to increase the range of the timer/counter. Timer 1 is used for hardware PWM generation, explored in the next example. We will use timer 0 for this exercise.
Firstly we need to choose the output bit to use for our PWM signal. The beauty of software control is that you have free choice of this. We will use the top bit of port B (PB7) in this example. We then need to look at the data sheet for the AVR and find the special function registers responsible for controlling and accessing timer 0. For the 2313 used previously these are :-

TCNT0 ($32)	-	**T**imer/**C**ou**NT**er contents, *A read/write register containing the 8-bit count.*
TCCR0 ($33)	-	**T**imer/**C**ounter **C**ontrol **R**egister, *Controlling how the counter functions.*
TIFR ($38)	-	**T**imer **I**nterrupt **F**lag **R**egister, *Triggering the internal interrupt*,
TIMSK ($39)	-	**T**imer **I**nterrupt **M**a**SK** register, *The ON/OFF switch.*

	7	6	5	4	3	2	1	0	
$33	-	-	-	-	-	CS02	CS01	CS00	TCCR0

PRE-SCALE SELECT

	STOP	0	0	0
	CLOCK	0	0	1
	CLOCK/8	0	1	0
	CLOCK/64	0	1	1
	CLOCK/256	1	0	0
	CLOCK/1024	1	0	1
External input	T0, falling edge	1	1	0
External input	T0, rising edge	1	1	1

Since we are not using interrupts we are only interested in the control register TCCR0 and the count value itself TCNT0.

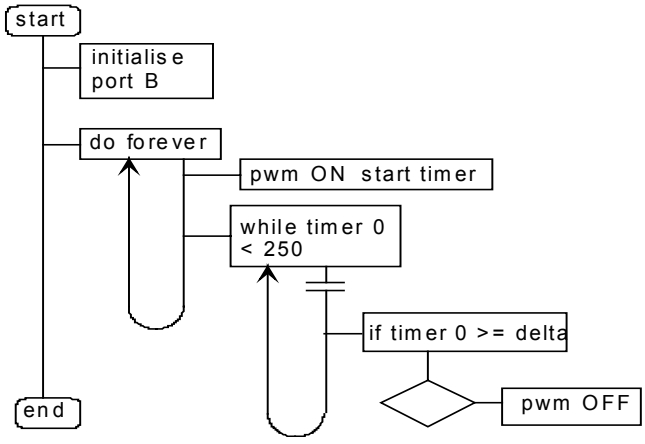
As shown above, TCCR0 turns on the timer and sets how it works, controlling the 10-bit pre-scalar and the clock source. We will use it connected directly to the system clock, with no pre-scalar selected, so a '1' is required in bit 0 of TCCR0.
The timer is an up timer, and starts counting as soon as a number is loaded into the timer register TCNT0.

	7	6	5	4	3	2	1	0	
$32	MSB							LSB	TCNT0

Most Significant Bit — Least Significant Bit

These are all the details necessary to write the program. The specification and DSD are shown below.

Specification DSD



1. Initialise port B.
2. PWM ON, and clear timer 0.
3. While timer 0 is less than 250; if timer 0 >= delta then turn PWM OFF.
4. Go to 2.

The names of these registers are already included in the include file we used previously, so it is a simple matter to call them by name in the program. Note the wording of the timer test. The timer is free running and, because of the instruction cycle time, it is possible to miss the exact time. Consequently the <= is used to ensure success and the maximum value is kept a few cycles below the 8-bit maximum of 255. Silly things like this catch you out in whatever language you use. It would be better to use interrupts of course, then this could be avoided. We will save that until the next example.

The matching C source code would then be written thus:

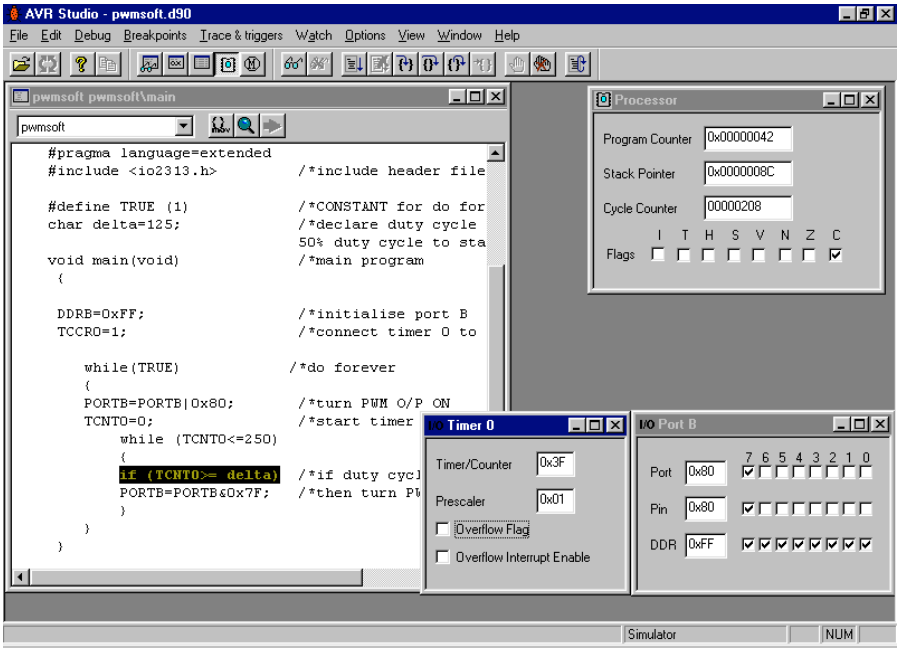
```
/*****************************************
```

```
 simple timer example
 for Atmel ATtiny2313  microcontroller
 version 1.0 PJS Nov 1998.
****************************************/
#pragma language=extended
#include <io2313.h> /*include ATtiny2313 header */

#define TRUE (1)     /*CONSTANT for do forever loop*/
char delta=125;      /*declare duty cycle and set it
                     at 50% duty cycle to start*/

void main(void)      /*main program            */
 {
 DDRB=0xFF;          /*initialise port B       */
 TCCR0=1;            /*connect timer 0 to clock*/
    while(TRUE)      /*do forever              */
    {
    PORTB=PORTB|0x80;   /*turn PWM O/P ON    */
    TCNT0=0;            /*start timer 0 at zero */
        while (TCNT0<=250)
        {
        if (TCNT0>= delta) /*if duty cycle reached */
        PORTB=PORTB&0x7F;  /*then turn PWM O/P off */
        }
    }
 }
```

Note that the duty cycle variable **`delta`** has been declared as an 8-bit CHAR and intialised at half way (125 is half of 250). This program would form part of the end application, which would update **`delta`** to change the duty cycle.

Since this is only a small program, running on the small AVR, the TINY memory model is still being used in the options set-up.

The program can then be simulated in C-SPY or AVR STUDIO after building. The screenshot below shows the STUDIO version, just prior to the pwm OFF point. As you can see, the peripherals are much easier to see in STUDIO. In C-SPY you would need to look up the address and monitor it in the memory I/O window.



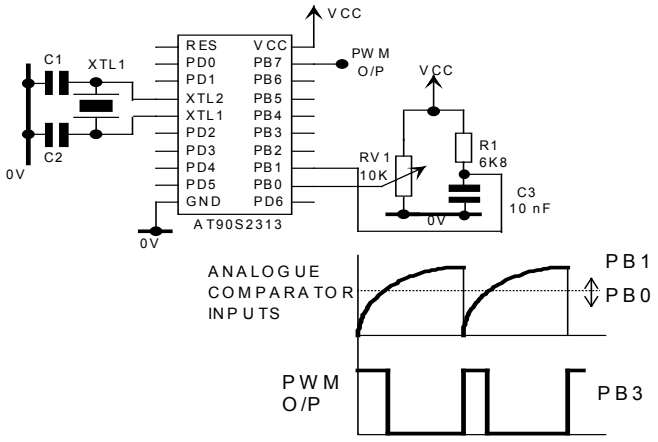
In STUDIO each peripheral has a separate window with the correct SFR names identified.

The problem with software generation of pwm is that the frequency is limited by the instruction cycle time. The resolution is also limited to that of the timer, in this case 8-bit, although this could be extended in software. In the case of the 2313, running at a cruising speed of 4 MHz, the program would produce a pwm, of nearly 8-bit resolution, at a frequency of :

1/(250 x 250 ns) = 16 kHz.

EXAMPLE 2 - INTERRUPTS

Most C compilers expect you to use assembly language routines for interrupt functions - not so with IAR 'C'. Extended keywords are provided along with, maybe, another include file to add functions to cope with intrinsic functions such as interrupts. This next example generates PWM in software by the same method as example 1, but employs interrupts, via the analogue comparator, built into the AVRs, to take in the duty cycle as an analogue input. This involves some cunning manipulation of the ports, using the high sink and source current capability of the AVR to reduce external components. This technique can also be used on other MCUs with high drive capability, such as the PIC. The circuit diagram below shows the hardware employed, with a 4 Mhz resonator used as the clock source.



The example makes a simple ADC to get input demand and produce PWM. The ON pulse is output at the start of the ADC cycle. The analogue interrupt provides the OFF output, when the two comparator inputs are equal. The cycle then completes under control of a timer. The circuit shows a simple arrangement using the ATtiny2313 AVR MCU from Atmel, but any AVR will do, or any MCU with comparator inputs and high current programmable I/O (PIC etc.,)

A simple analogue to digital converter is constructed to provide a means of control. This uses a simple R-C charging/discharging method, and the analogue comparator, to produce an exponential waveform on one input/output and compares it with the analogue voltage from a variable resistor on the other input. The capacitor is discharged at the start of the cycle by changing port configuration from input to output, and writing a '0' to that bit. Port configuration is then changed back to input, to allow the capacitor to charge up again for the rest of the cycle. Timer 0 is used to time the conversion process.

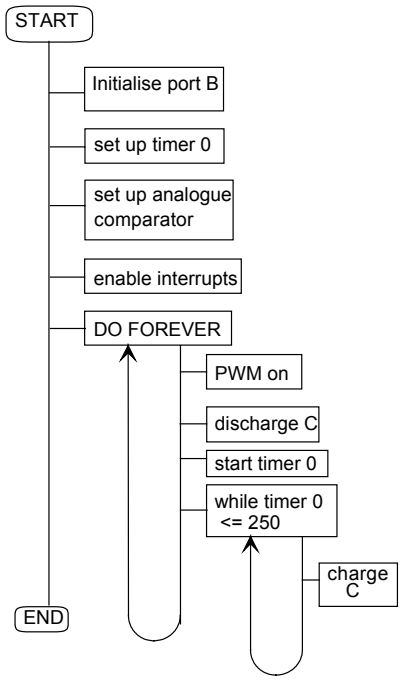
When the two inputs are equal, the analogue comparator toggles, interrupting the main timing process and turning off the PWM output to terminate the t pulse time. Control then returns to the main program, which continues timing the period time T. Since the periodic

time T is controlled by a hardware timer, minimal disturbance will be produced by the interrupt. The waveform produced is 'glitch' free and dependent on processor function. i..e. the processor is producing the PWM and any other process will disturb it.

The specification and DSD are shown overleaf. The important thing to note for this example is how interrupts are handled. For the AVR, you must include another file to handle the intrinsic functions. This is called <ina90.h>. The interrupt vectors form part of the interrupt service routine function.

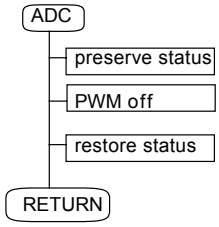
Specification

Design Structure Diagrams



Main program

1. Initialise port B.
2. Set up timer 0.
3. Set up analogue comparator.
4. Enable interrupts.
5. PWM on.
6. Discharge C.
7. Clear timer 0.
8. While timer 0 <=250, charge C.
9. Go to 5.



Interrupt Service Routine

1. Preserve registers
2. Turn off PWM
3. Restore registers
4. Return from interrupt.

The program can now be constructed as shown overleaf.

```
/*****************************************
 Simple interrupt example
 for Atmel AT90S2313  microcontroller
 version 1.0 PJS Nov 1998.
*****************************************/
 #include <io2313.h>     /* include register names  */
  #include <ina90.h>     /* and intrinsic functions */
  #define DISCHARGE (0x02)/* cap CHARGE on bit 1 */
  #define CHARGE (0xFD)   /* cap DISCHARGE on bit 1 */
```

```
#define PWMON (0x80)    /* pwm output on bit 7   */
#define PWMOFF (0x7F)   /* pwm output on bit 7   */
#define TRUE (1)

void C_task main (void)
{
   DDRB=0xFE;     /* initialise port B directions */
   TCCR0=1;       /* connect timer 0 to clock     */
   ACSR=0x0A;    /* set up analogue comparator   */
   _SEI();       /* enable interrupts            */
   while(TRUE)
       {
       DDRB=DDRB|DISCHARGE;/* discharge capacitor */
       _NOP();          /* allow time to discharge*/
       PORTB=PORTB|PWMON;  /* turn pwm output on */
       DDRB=DDRB & CHARGE; /* charge capacitor   */
       TCNT0=0;            /* start timer 0      */
           while (TCNT0<=250) /* time period T*/
           {
           }
       }                   /* and repeat cycle   */
 }

/*********************************************
Interrupt service routine to turn off PWM output
*********************************************/

  /*interrupt [ANA_COMP_vect] void adc(void)*/
#pragma vector = ANA_COMP_vect
__interrupt void ANA_COMP_interrupt( void )
  {
  PORTB=PORTB & PWMOFF;  /* turn off  PWM output   */
  }
```

This program contains a few additions of interest.

- The extra include file to access the intrinsic functions is needed for the AVRs (not all MCUs need this). This provides the enable interrupt function _SEI(), the no operation _NOP function to waste a clock cycle and many others.
- The bit masking numbers are defined as constants, making any port modifications easier and making the program easier to read.
- The main function has been declared as **`C_task`**. This is an extended keyword, which stops the compiler preserving vital registers on the stack when the function is called. This saves stack space. Since main will not be called by anything else there is no point in preserving registers! This is not available on all target versions. You will need to check the help files to see if it is available or remove it if the Compiler gives an error.
- Bit masking is employed in all the port manipulations to avoid disturbing any other bits. If the MCU has single bit instructions, like the AVR has SBI and CBI, this will be used automatically by the optimising routine instead of the AND and OR method.
- The interrupt function **`adc`** is declared in a special way using the **`interrupt`** type and has the interrupt vectors defined inside the square brackets. These are defined in the include file and the name must match that in the include file. With the analogue comparator of the AVR the vectors are called **`ANA_COMP_vect,`** and the same approach is adopted for all other interrupt sources.

- Registers are automatically preserved by 'C' on entry to a function and restored on exit. Consequently, you do not need to do this yourself, as you would in assembler. That is why the main function should use **`C_task`** if possible to save stack space.

Notes for WINAVR C Compiler included in AVR Studio
WINAVR has different syntax for some things and different include file names. This can be annoying. The above example will not compile in WINAVR because it uses different include files. To get this code to run, add
1)

```
#include <avr/interrupt.h> /* include ISR function */
```

2) `#include <avr/io.h>  /* instead of <io2313.h> */`
3) Change interrupt code to

```
/* Interrupt service routine to turn off PWM output    */

ISR(ANA_COMP_vect)
 {
 PORTB=PORTB & PWMOFF;  /* turn off  PWM output   */
 }
```

4) Remove `#include <ina90.h>` as WINAVR does not recognise it.
5) Find include file for SEI(0) and NOP() functions. I have no idea what this is I'm afraid, and Help files are limited! As an alternative use in-line assembler, like this:-

```
asm volatile ("SEI");       /* enable interrupts */
asm volatile ("NOP");       /* No Operation      */
```

These type of changes are needed every time you swap between different C Compilers, so you place your money and take your choice. WINAVR is free, has no linker and produces much larger code. IAR produces the best output but is expensive for the full version. The other two commonly available C Compilers are Code Vision and Image Craft. These cost a few hundred dollars and come somewhere between WINAVR and IAR for efficiency.

Anyway, back to the project.
Those of you who have not played with MCUs capable of driving high current loads (20 mA or so) will be impressed by the ability to use pins as I/O in this manner, without external buffering. The flexibility provided by the AVR and PIC (and any other device employing the same methods) port arrangement is unsurpassed for this type of work.

The analogue to digital converter is not the best in the world, but it works. It could be improved by replacing the resistor with a constant current source, producing a linear ramp instead of the exponential waveform. This would make the control more responsive and would mean that the timer value, at interrupt, would be directly proportional to analogue voltage input. The variable resistor input also causes jitter problems at the wiper extremes because of the comparator input limitations, i.e. the input range does not go right to the supply rails. This could be removed by placing small series resistors on each end of the wiper.

EXAMPLE 3 - HARDWARE PWM

Many MCUs have hardware support for PWM generation. This example is a modification of example 2, but uses the hardware circuits of the AVR instead. The circuit remains the

same except for the PWM output, which is now on port B bit 3 as dictated by the hardware circuits.

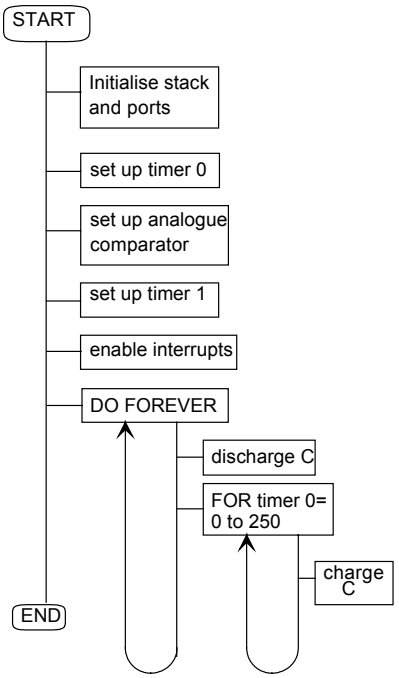
The analogue to digital conversion remains identical. The on and off control for the PWM output is now under control of timer 1, which does the PWM generation, independently of the processor. The resolution can also be improved from 8-bit because the hardware supports 8, 9 or 10-bit generation. The interrupt routine now transfers the timer 0 value, which contains a number proportional to input voltage, to the duty cycle control register OCR1L. This example selects 8-bit PWM mode.

Specification

Main program

1. Initialise port B.
2. Set up timer 0.
3. Set up analogue comparator.
4. Set up timer 1 in PWM mode.
5. Enable interrupts.
6. Discharge C.
7. Clear timer 0.
8. While timer 0 <=250, charge C.
9. Go to 6.

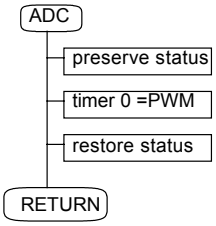
DSD



Interrupt Service Routine

1. Preserve registers
2. Transfer timer 0 to PWM
3. Restore registers
4. Return from interrupt.

INTERRUPT SERVICE ROUTINE



The program can now be constructed by modifying the previous one as shown in bold below.

```
/*****************************************
 Hardware PWM example
 for Atmel AT90S2313  microcontroller
 version 1.0 PJS Nov 1998.
*****************************************/
#include <io2313.h>  /* include register names  */
#include <ina90.h>   /* and intrinsic functions */
#define DISCHARGE (0x02)   /* capacitor mask 1  */
#define CHARGE (0xFD)      /* capacitor mask 2  */
#define TRUE (1)
void C_task main (void)
 {
```

```
    DDRB=0xFE;       /* initialise port B directions*/
    TCCR0=1;         /* connect timer 0 to clock    */
    TCCR1B=1;        /* connect timer 1 to clock    */
    ACSR=0x0A;       /* set up analogue comparator  */
    TCCR1A=0x81;     /* set 8-bit PWM mode          */
    OCR1L=127;       /* set initial 50% duty cycle  */
    _SEI();          /* enable interrupts           */
    while(TRUE)
        {
        DDRB=DDRB|DISCHARGE; /* discharge capacitor*/
        _NOP();          /* allow time to discharge*/
        DDRB=DDRB & CHARGE; /* charge capacitor    */
        TCNT0=0;            /* start timer 0       */
            while (TCNT0<250) /* time T period */
            {
            }
        }                    /* and repeat cycle   */
  }
   /*************************************************
  * Interrupt service routine to load PWM register with new duty cycle
*************************************************/

  #pragma vector = ANA_COMP_vect
__interrupt void ANA_COMP_interrupt( void )
  {
  OCR1L=TCNT0;   /* transfer timer 0 value to PWM  */
  }
```

Access to special functions is that simple with C.

EXAMPLE 4 - LOOK-UP TABLES

Enough of PWM, let us try to set up and access look-up tables, another very common thing to do in control work. This example, and the next, drives a single 7-segment display and uses look-up table, stored in ROM and then EEPROM, as the decoding medium.

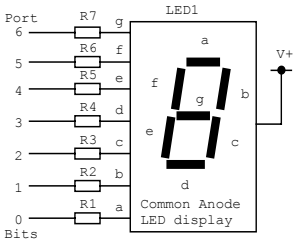
EXAMPLE 4 A - ARRAY STORAGE IN ROM

Arrays are convenient storage places for look-up tables and similar sets of connected data. They allow you to access data in a random, or sequential manner as you wish. The following example stores a seven segment decoding table, in an array in the ROM of the AVR. ROM is a convenient place to store fixed data because it is normally available in quantity and it will never be changed. MCUs normally have moderately large ROMs and very little RAM.

This example continually displays the numbers 0 to 9 on a single digit LED display directly driven from the port of the MCU. Modern MCUs like the AVR and PIC are capable of sinking 20 mA, or so at least, so the segments are operated in current sink mode, i.e. '0' = on. Other MCUs will need buffers.

The port allocation and interface circuit, are shown overleaf, along with the numbers required to display the digits 0 - 9. This example is kept simple, so only one digit display is driven. It is a simple matter to multiplex more digits using another port and use the same table for decoding purposes.

Connection diagram for single digit LED display (Common Anode).



The current limit resistors are calculated to limit the LED current to a safe level, typically 10 mA will produce a bright display. On a 5v supply, the values will thus be about,
(5v - 2v)/ 10 mA = 300 Ω.
Either a 270 or 330 Ω resistor would be employed.

Table of data required to drive display in active low mode (current sink).

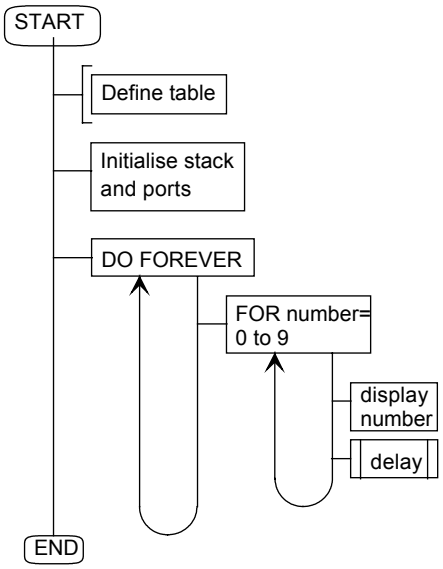
BIT	7	6	5	4	3	2	1	0	HEX
SEG	DP	g	f	e	d	c	b	a	Data
0	1	1	0	0	0	0	0	0	0XC0
1	1	1	1	1	1	0	0	1	0XF9
2	1	0	1	0	0	1	0	0	0XA4
3	1	0	1	1	0	0	0	0	0XB0
4	1	0	0	1	1	0	0	1	0X99
5	1	0	0	1	0	0	1	0	0X92
6	1	0	0	0	0	0	1	0	0X82
7	1	1	0	1	1	0	0	0	0XD8
8	1	0	0	0	0	0	0	0	0X80
9	1	0	0	1	0	0	0	0	0X90

The program structure can now be arranged to continually display the numbers 0 to 9 in short intervals using an array as the decoding medium.

Specification **DSD**

Main program

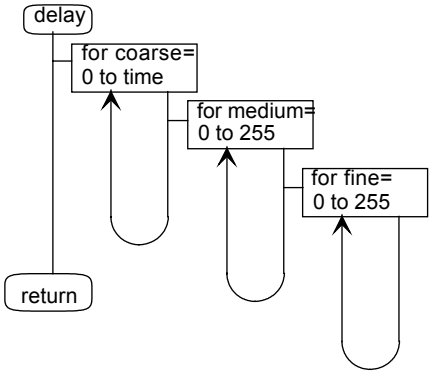
1. Define array.
2. Initialise ports.
3. For number = 0 to 9 find decoded value and display.
4. Delay a bit
5. Go to 3.



Delay function

1. For coarse = 0 to time,
2. For medium = 0 to 255,
3. For fine = 0 to 255,
4. Return.

This is the same delay function as used in a previous example.



Matching C program :-

```
/*****************************************
 Look-up table array example
 for Atmel AT90S2313  microcontroller
 version 1.0 PJS Nov 1998.
*****************************************/
#include <io2313.h>  /* include register names    */
#include <ina90.h>  /* and intrinsic functions   */
#define TRUE (1)
/*Put look up table in ROM*/
char flash LED[10] = {0xC0,0XF9,0XA4,0XB0,0X99,
0X92,0X82,0XD8,0X80,0X90};
char number=0;
void delay(char);
void main (void)
{
DDRB=0xFF;          /* initialise port B directions */
     while(TRUE)
     {
     for (number=0;number<10;number++)
         {
         PORTB=LED[number];  /* find decoded byte and
                  write to LED*/
         delay(10); /* wait a bit to show display  */
         }          /* find next number            */
        }           /* and repeat cycle            */
  }

void delay(char time)
 {
 char coarse, medium, fine;
    for (coarse=0; coarse!=time; coarse++)
    {
        for (medium=0; medium!=255; medium++)
        {
            for (fine=0; fine!=255; fine++)
            {
            }
        }
    }
 }
```

The important items of interest here are how the array is set up and defined, and how C is told to store it in Flash memory.

```
char flash LED[10] = {0xC0,0XF9,0XA4,0XB0,0X99,
0X92,0X82,0XD8,0X80,0X90};
```

This line uses the **`flash`** extended keyword, informing C that the variable should be stored in FLASH ROM. Other MCUs without flash ROM, will have similar extended keywords if ROM access is possible. (Not all MCUs allow this) The size of the array is defined as having ten elements, whose addresses will be 0 to 9. The contents are defined inside the braces, in whatever number system you like. The variable `number`, from the `for` loop, is used as the address and the result sent to port B by the line :-

```
PORTB=LED[number];
```

This is a very simple means of placing and using look-up tables.

If the data is likely to be changed at some point in time, and you do not want to disturb the main ROM, or it is One Time Programmable (OTP), then EEPROM would be a better

location. This process is complicated by the need for a separate file for EEPROM access. This file can be downloaded to the EEPROM at any time before or after the main ROM download. The beauty of this is that it can be downloaded at any time, without disturbing the main program.

Part B of this example shows one method of using the EEPROM to do the same job as the above example.

EXAMPLE 4 A - ARRAY STORAGE IN ROM

For MCUs with EEPROM, intrinsic functions are defined which access the EEPROM. For the AVR, these are called , _EEGET and _EEPUT, for read and write access respectively, and are defined in the `ina90.h` include file. The easiest way to create the table is to use the assembler. In this example, I have used the AVR assembler WAVRASM to create the EEPROM file, but any suitable assembler for the MCU would have this facility.
For the AVR the following source code in WAVRASM creates the necessary file for downloading into the EEPROM :-

```
;***EEPROM look-up table for C program**********
.ESEG
.ORG 0
.DB "0xC0,0XF9,0XA4,0XB0,0X99,0X92,0X82,0XD8,0X80,0X90"
```

This produces a file with a .eep extension when assembled, which can then be downloaded independently from the ROM image.
The structure remains almost identical but the C source code now changes to :-

```
/*****************************************
 Look-up table array example
 for Atmel AT90S2313  microcontroller
 version 1.0 PJS Nov 1998.
*****************************************/
 #include <io2313.h>  /* include register names  */
 #include <ina90.h>   /* and intrinsic functions */
 #define TRUE (1)
 char number=0;
 char value=0;
 void delay(char);
 void C_task main (void)
 {
 DDRB=0xFF;     /* initialise port B directions */
      while(TRUE)
      {
         for (number=0;number<10;number++)
         {
         _EEGET(value,number);
         PORTB=value;   /* find decoded byte and
                           write to LED*/
         delay(10); /* wait a bit to show display */
         }              /* find next number       */
      }                 /* and repeat cycle       */
  }

void delay(char time)
 {
 char coarse, medium, fine;
    for (coarse=0; coarse!=time; coarse++)
    {
        for (medium=0; medium!=255; medium++)
        {
            for (fine=0; fine!=255; fine++)
            {
            }
        }
    }
 }
```

The format of the intrinsic functions will be found in the include file. For the AVR the appropriate part of the include file is :-

```
#define _EEPUT(ADR,VAL)     while (EECR & 0x02); \
```

```
 EEAR = (ADR); EEDR = (VAL); EECR = 0x04; EECR = 0x02;

#define _EEGET(VAR, ADR) while (EECR & 0x02); \
        EEAR = (ADR); EECR = 0x01; (VAR) = EEDR;
```

This tells you that, for the write function, you need `_EEPUT(ADR,VAL)` where `ADR` is the address in EEPROM and `VAL` is the data to be written, and the read function uses `_EEGET(VAR, ADR)` where `VAR` is the variable to house the data, `ADR` is the address in EEPROM to be read.

This program used `_EEGET(value,number);` to do the job.

5 INTEGRATING DEVELOPMENT TOOLS IN THE DESIGN PROCESS.

We have already seen how some other software can be used in conjunction with 'C'. This chapter delves deeper into the ways of making the package part of your overall development system. It discusses how the software fits in with hardware programmers, ICE systems and other attachments commonly employed. It also discusses ways to configure the software to your own requirements.

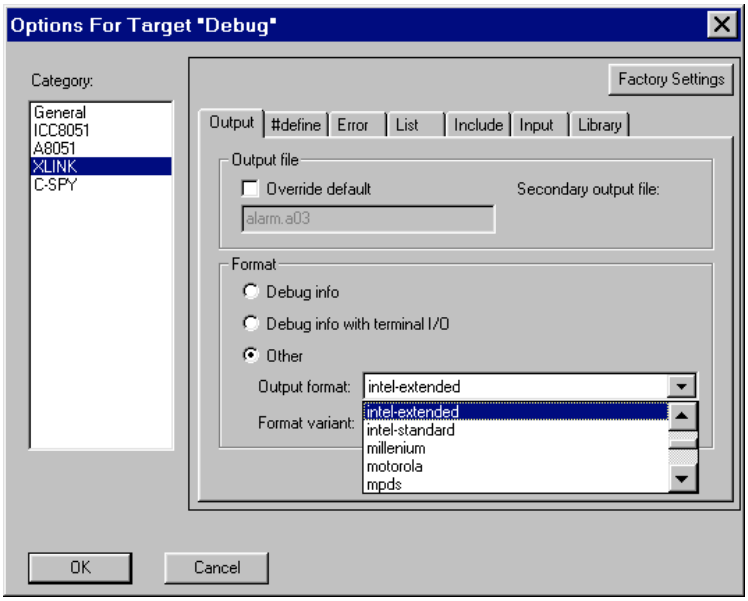
HARDWARE PROGRAMMING

The linker produces the final linked code ready for debugging or downloading to the target processor. In the previous chapters we have mainly generated debug output In the real world we would want to download the code to hardware, either directly as a ROM image, or to an In Circuit Emulator (ICE), Target board or ROM emulator.

The Embedded Workbench has a huge selection of file output formats to cover most requirements of this nature. The selection will depend on the type of target processor.

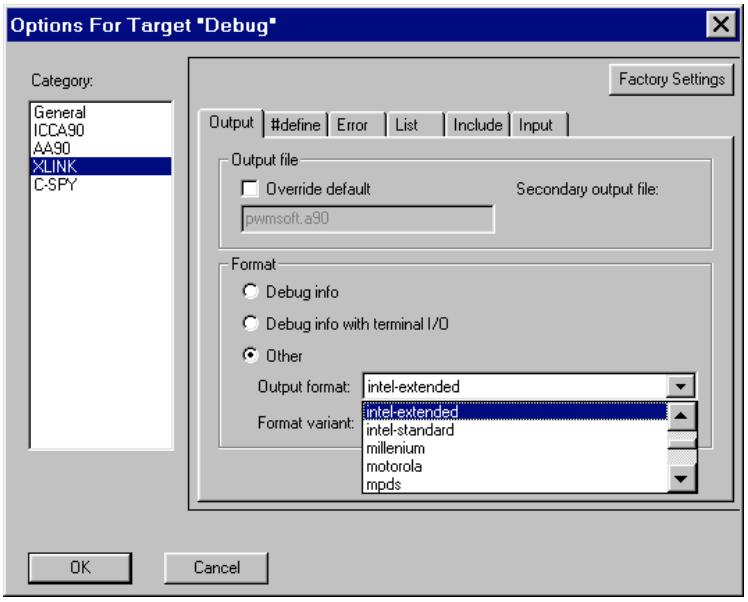
The screenshot overleaf shows just a few of the options available on the full version, in this case the 8051 system. The selection starts at Aom and Ashling and ends with Zax.

The selection shown highlighted is that required for the full version to generate ROM code.



The screenshot overleaf shows the same window in the AVR version. The only difference is the extension used on the output file.

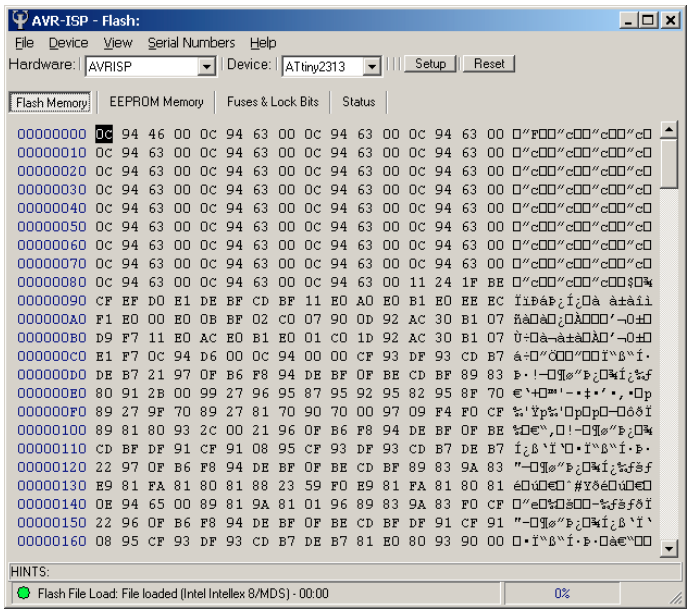
It is this common environment which makes the system so efficient for development work. Changing target platforms has minimal effect on the development environment, making the move from one system to another, a minor task of becoming familiar with the special



functions.

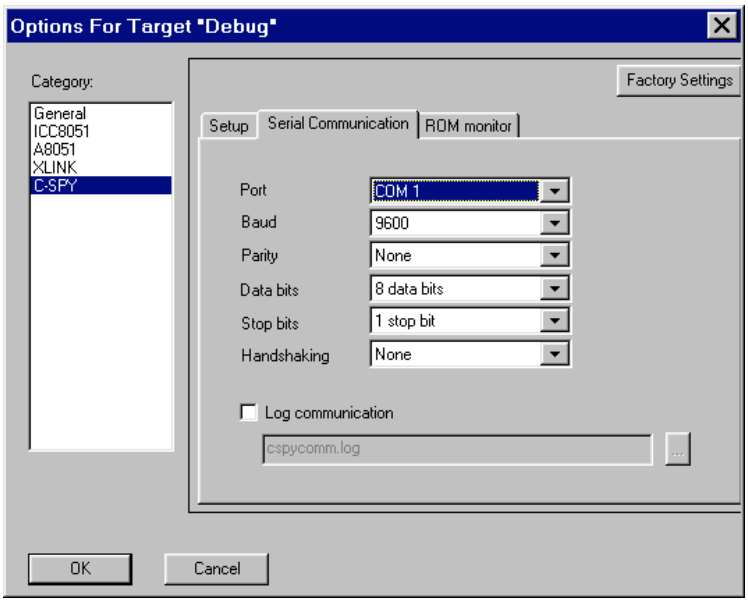
The file generated is then loaded into the appropriate download software for your development system. In the case of the AVR, and other MCUs with ISP FLASH, this can be downloaded directly into the target processor for testing, or into an ICE system for further debugging. With Windows 95/98/NT/XP you can have all your software in memory, and switch between tasks as you need them. Some software is even designed to work in conjunction with IAR 'C'. We have already seen how the AVR STUDIO debugger works in parallel with Workbench. If you have both in memory at the same time, it even reloads your corrected debug file automatically, after you amend it. This means that you can switch quickly between Workbench and STUDIO in an iterative manner until the program works to your satisfaction, with a minimum of fuss. Since STUDIO also directly drives the AVR JTAG ICE system or starter board, you can immediately see the effect of your changes, on the final hardware.

Even if you are using separate software, this software switching is still possible. For instance ,I use the parallel port programmer to program the AVRs.. The software for this runs in a small window, which you can have in memory all the time, making download very quick. This is equally applicable to any other programming system working in a Windows 95/98/NT environment. The screenshot below shows the ROM code, generated by 'C', loaded into the Kanda AVRISP software ready for download.



This kind of programming download software is available for most MCU platforms.

With targets that are OTP or EPROM based (i.e. not FLASH ROM, like the AVRs, some PICs, NEC 32-bit V850 and many more) the C-SPY Project Option windows gives you access to ROM monitors, which you may have already, via the serial port. The screenshot below shows how communication to this can be set up, prior to download.

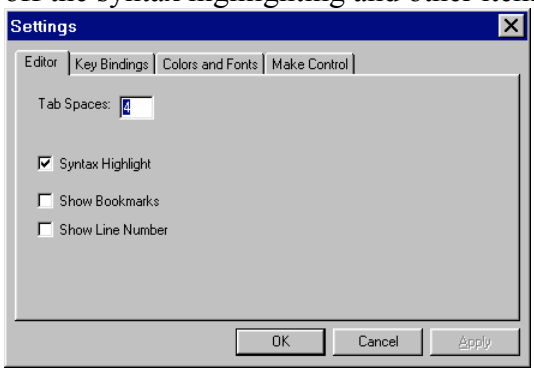


The tick box on the set up card needs to be ticked to produce ROM monitor code and the ROM driver selected for this window to become active. The ROM monitor card also gives you set up options for the ROM monitor such as target consistency checks and speed of download.

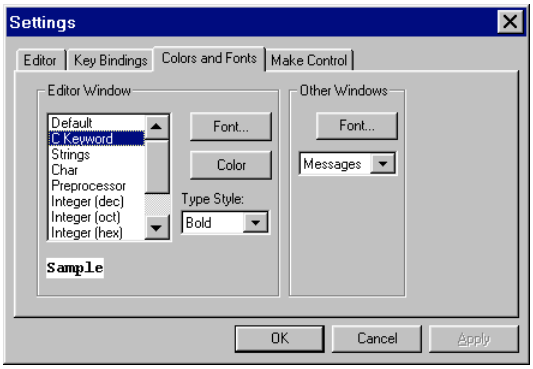
This arrangement enables you to try out your program directly from Workbench, before committing code to EPROM or OTP versions of the MCU.

SOFTWARE CONFIGURATION

We have already changed project options to amend how the code is built. Many other changes can be made to customise the user interface to your own requirements. For example, you can amend the editor colours, tab settings etc., from the main Options menu. The screenshot below shows how the editor may be amended to change the tab settings, turn off the syntax highlighting and other items.



The next window card 'Key Bindings' enables you to set up quick keys to speed up common functions you use. Most of the things you do can be speeded up by this method if you are a 'speed freak', rather than a 'mouse addict'. The Colors and Fonts card allow you to change the colour and fonts scheme to your own preferences. This screenshot shows part of the selection available.



The Make Control card allows you to control the action on Make and Build all. Whether you stop build on errors or warnings or not at all.

FUNCTION EXTENSIONS

What happens if you are not happy with the library functions supplied? The answer is to make your own. IAR even supply templates for some for you to modify. Small versions of **`printf`** and other common functions are included for the smaller MCU targets. You will most likely want to customise I/O functions such as **`putchar`** and **`getchar`** to suit your hardware environment. You can replace the ones supplied, which are for terminal I/O, in the library **`stdio.h`** or create your own function under the same or different name (probably safer) and add it to a library using XLIB.

As an example IAR supply a sample version of putchar in the icca90 directory which looks like this :-

```
/*            - putchar.c -
This sample version of putchar is written for the AT90S8515 and uses the
UART to transmit characters.
If you want to include your own version of putchar in your library, first
compile it, then insert it in the library file (cl??.r90)using the xlib
command 'replace-module'.

The following compiler options should be used when replacing a library
module:

  -b        Make object a library module
  -gA       Enable global typecheck, Depreciate K&R-style functions
  -m??      Select memory model ?? (eg. -ms means small memory model)
  -v??      Select processor option: ?? (eg. -v1 means 8515, -v0 means
2313)
  -z9       Maximum size optimization
```

```
  Version: 1.03 [ICAP]

*/

#include <stdio.h>            /* getchar prototype */
#include "io8515.h"            /* SFR declarations */

#define TXC 0x40

int putchar(int c)
{
  while (!(USR & TXC))  /* Wait until transmit is complete */
    ;
  UDR = c;
  return c;
}
```

This just places the character passed into the function in the UART data register of the AVR and returns with the character.
This may be added to the library <stdio.h> instead of the original.
It is important that you save the original first of course! You must compile it using the correct memory model options for the target MCU you are using.

Note that `putchar` acts as the low level part of the `printf` function. Note also that different memory models have their own library modules.

For instance, the AVR AT90S8515 using the -v1 model, has a file called cl1s.r90, whereas the -v0 model uses cl0s.r90. You can find all of the files for your particular MCU family in the compiler directory.

EFFICIENT CODING

When writing code for MCUs it is important that you appreciate the limitations of the target processor architecture and instruction set to avoid the use of inefficient language constructs. As an example the following list contains some ideas for use with 8-bit targets:-

- 8-bit data types (**`char, tiny`**) are handled more efficiently and should be used where possible.
- Avoid global and static variables, so that registers may be reused.
- For signed data types use x>>1 instead of x/2 which produces more efficient code.
- Bitfield types should be used only to conserve data memory space, as they execute slowly on some platforms.
- Use the smallest memory model possible.
- Declare **`main`** as **`C_task`** if possible. This prevents registers from being saved when entering **`main`**, conserving stack and program space.
- Use the **`?`** operator rather than the **`if`** construct because it generates more efficient code.

IN CONCLUSION

I hope this book has helped you get started with this powerful language and encouraged you to continue. You will need to read the help files of your version to find all the facilities available and the full version does come with the usual thick manuals! The book was never intended to be a comprehensive guide to Embedded C (the book would have to be four times this size) but it should be sufficient to 'Get you going…'.

Although the book includes a lot of examples using the Atmel AVR, this does not mean that the book is specific to that range of MCUs. It just happened to be convenient for me, because I have the full version. The same techniques and code can equally be applied to any other target MCU. I have not yet tried the newly released PIC version (it was in Beta testing when I wrote this book), but I have no doubt it will be very popular. Since the PICs have a similar port capability to the AVR all the examples will work on the PIC (assuming you use the correct PIC of course).
I am also hoping to look at the new NEC 32-bit version to support the new V850 MCUs. This competes with the Atmel 16/32 bit MCUs based on the ARM7 core. Both of these will need the support of C to program them! Wish me luck and happy programming!

APPENDIX A

GLOSSARY OF TERMS

ADC Analogue to Digital Converter - a circuit converting varying signals to binary digital data.

ANALOGUE A term which covers systems and devices which handle electrical signals that vary continuously with time.

ASSEMBLY LANGUAGE A simple collection of Mnemonics used to represent more complicated machine code instructions. Source code written in this can be compiled by simple assemblers to produce one-to-one conversion object code. The most code efficient programming language, but can be difficult to follow.

AVR **A**lf (Bogen) **V**ergard (Wollan) **R**isc microcontroller from Atmel Norway design team.

BIT Binary digIT, one logic level represented by a voltage level, `1' normally near the supply rail e.g. +5v, `0' near the 0v rail.

BITFIELD A byte or word used as one or more individual bits for variable storage in a structure.

BYTE Eight bits in parallel.

C A high level portable language producing efficient code when compiled into machine code.

CISC Complex Instruction Set Computer - a computer having a large instruction set capable of most things but resulting in slower execution times.

COMPILER A software application taking a high level language source file and creating a machine code object file capable of running on the target processor.

DAC Digital to Analogue Converter - a circuit to convert binary digital data to varying electrical signals.

DIGITAL A term which covers systems and devices which handle electrical signals that vary discontinuously with time (steps), usually binary 0 and 1.

DSD Design Structure Diagram, a modified form of flowchart showing program flow in a diagrammatic fashion and forcing a structured approach.

EEPROM Electrically Erasable Programmable Read Only Memory. A non-volatile ROM programmed by normal voltage levels and erased by normal voltage levels (usually). Normally slow to program, but can be reused many times.

EPROM Erasable Programmable Read Only Memory. A non-volatile ROM programmed by higher voltage levels than normal and erased by exposure to Ultra-Violet light. Most expensive ROM, but can be reused many times.

FIRMWARE Programs stored in ROM, usually the operating system, for embedded control work this is THE program.

FLASH A type of non-volatile memory ROM capable of fast read and write, usually at normal supply levels.

HARDWARE The components of a system. (Electronic and Mechanical).

HARVARD Computer architecture having separate address buses for memory devices, a wider word size for program memory, allowing single line programs.

INTERRUPT The action of a micro stopping execution of the main program and switching to another on instigation of a signal from elsewhere. (Hardware or Software.)

ISP In System Programming - the ability of a device to be programmed in circuit without disconnection.

LINKER Part of a compiler package, which links individual compiled files together to form a contiguous code block.

MACHINE CODE The language of the processor, a binary coderepresenting some instruction to the MCU. See also Object Code.

MPU Micro Processor Unit. A digital circuit to provide programmable logic functions, requiring other support ICs.

MCU Micro Controller Unit, a complete micro-based system on a single chip.

NIBBLE Four bits.

NON - VOLATILE
Refers to memory devices which DO NOT lose their contents when power is removed.

OBJECT CODE Machine code created by a compiler to run on a processor.

REGISTER A storage location inside a device, usually a micro. or support IC.

RISC **R**educed Instruction Set Computer - having a smaller but hardware efficient instruction set. Most commonly used instructions are provided, others are made from those available. Fast execution time.

SINK Current flow into the output of a device (from supply), usually at logic `0' state.

SOURCE Current flow out of a device (to 0v), usually at logic`1' state.

SOURCE CODE A text file containing program language and comments as input to assemblers or compilers, for translation to object code.

SPI Serial Peripheral Interface - a method of sending and receiving fast data in serial form between two or more devices.

STRUCTURE An array of dissimilar variable types.

VOLATILE Refers to memory devices which lose their contents when power is removed.

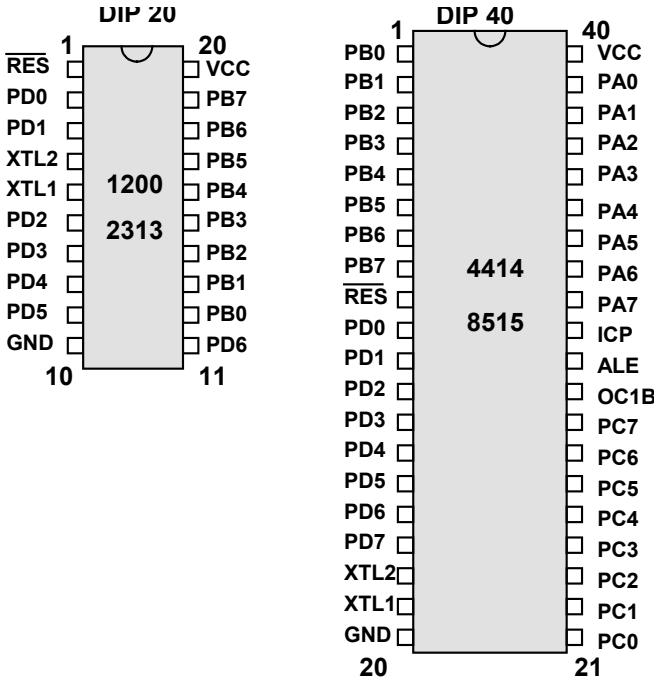
VON NEUMANN The original architecture of microprocessors having a common address bus for all memory devices.

WORD Either a sixteen bit number, or the number of bits used by the system for data, depending on context.

APPENDIX B

OVERVIEW OF TARGET MCUs USED

PINOUTS OF MCUs



The pinouts above show the 20-pin and 40-pin DIP versions of the AVR MCUs involved. The 40-pin version is also the same pinouts as the 8051 compatible MCU discussed AT89S8252. (Except Port A is called Port 0, Port B, Port 1 etc.,)

Atmel AT89S8252 40-pin MCU MCS-51 compatible

This device is a CISC type conventional MCU of the 8051 type, with the standard Von Neumann architecture. The I/O ports are TTL compatible.

Features

- 8k ISP FLASH ROM
- 2k EEPROM
- 2.7 - 6.0 v operating range.
- Fully static clock 0 -24 MHz range
- Three 16 bit timer/counters
- 256 x 8 internal SRAM
- Nine interrupt sources
- UART
- SPI interface
- Watchdog timer

The pinouts are as shown overleaf for the AT90S8515.

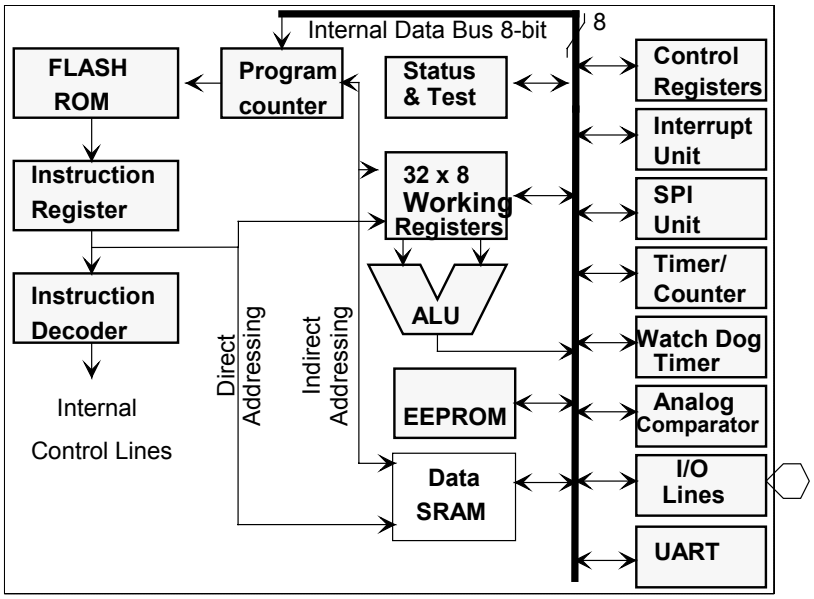
The 8051 is the workhouse of Embedded control and millions are in everyday use throughout the world. They execute quite slowly, typically taking twelve clock cycles to perform an instruction.

The wide range of instructions give a C compiler choice, so code efficiency is quite good, except that, having only two accumulators, RAM has to be mostly used for variable storage instead of registers. This will slow down execution somewhat.

Atmel RISC AVR MCUs CLASSIC range

These are newly introduced, true RISC MCUs with a Harvard architecture, with 8051 family pin compatibility for easy replacement, although the reset line is the opposite polarity. They all have 32 registers all of which can be used as accumulators. The instruction set and architecture have been designed for C programming and cycle time = clock time.

Sample structure of AVR device :-



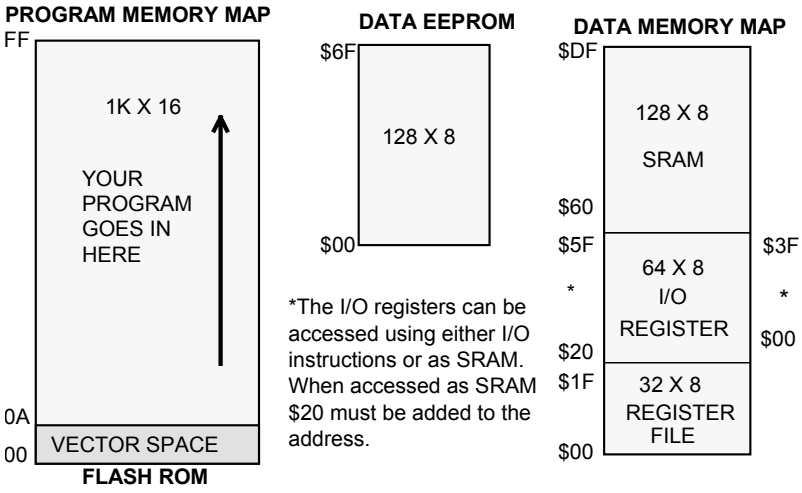
The two AVRs mentioned in the text are the 20-pin ATtiny2313 and the 40-pin ATmega8515.

Atmel Classic AVR ATtiny2313 20-pin MCU

Features

- 2k ISP FLASH ROM
- 128 bytes EEPROM
- 32 working registers
- 15 high current (20 mA) I/O lines
- 2.7 - 6.0 v operating range.
- Fully static clock 0 - 12 MHz range
- Two timer/counters (8 and 16-bit)
- 128 x 8 internal SRAM
- Ten interrupt sources
- Full duplex UART
- 8, 9 or 10-bit PWM
- Analogue comparator
- Watchdog timer

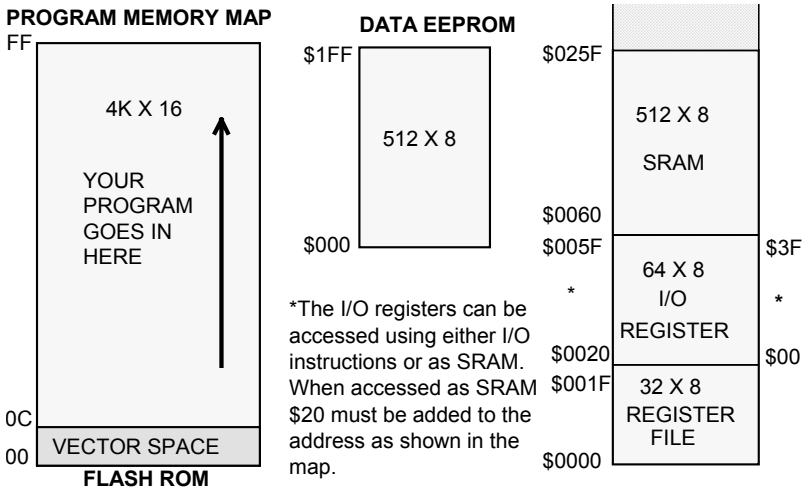
Each of these blocks are addressed separately (the Harvard approach).



Atmel Classic AVR ATmega8515 40-pin MCU

Features

- 8k ISP FLASH ROM
- 512 bytes EEPROM
- 32 working registers
- 32 high current (20 mA) I/O lines
- 2.7 - 6.0 v operating range.
- Fully static clock 0 - 8 MHz range
- Two timer/counters (8 and 16-bit)
- 512 x 8 internal SRAM
- Ten interrupt sources
- Full duplex USART and SPI port
- Dual 8, 9 or 10-bit PWM
- Analogue comparator
- Watchdog timer



External RAM up to 64k can also be accessed.

APPENDIX C

USEFUL ADDRESSES

Atmel UK Sales Offices & Operations

Atmel U.K., Ltd.
Coliseum Business Centre
Riverside Way
Camberley, Surrey GU15 3YL
England
TEL (44) 1276-686677
FAX (44) 1276-686697

AVR Data
Memory and other
MCUs

MCU SUPPORT PRODUCTS

Kanda.com
P.O. Box 200
Aberystwyth
Ceredigion
SY23 4AY
TEL +44 (0) 8707 446 807
FAX +44 (0) 8707 446 807