The C51 Primer

An introduction to the use of the Keil C51 Compiler on the 8051 family

Edition 3.6 17 January 2006

by

First version by
Mike Beach

Editor for Edition 3.6.5
Chris Hills

PhaedruS SystemS

www.phaedsys.com

Contents

0 About The C51 Primer

If you've looked at a few 8051 datasheets, other 8051 books or flicked through the chapters in this guide, you may be left thinking that it is necessary to be an 8051 expert to produce workable programs with C51. This is not the case. In fact many hobbiests have turned out '51 systems at home using only basic tools. It is perfectly possible to write real commercial programs with nothing more than a reasonable knowledge of the ISO C language, the 8051 extensions and some appreciation of hardware.

However, to get the maximum performance from the 8051 family, knowing a few tricks is very useful. This is particularly true if you are working on a very cost-sensitive project where needing more memory can result in an unacceptable cost. After all, if cost was not a consideration, we would all be using ARM9 or PowerPC's!

Whilst the C51 Primer is really aimed at users of the Keil C51 Compiler, it is applicable in part to compilers such as IAR and Tasking.

This edition of the C51 Primer will use the Keil C51 PK51 package version 7.5, released in 2005.

The C51 Primer Will Help You

Find your way around the basic 8051 architecture.
Make a sensible choice of memory model and special things to watch out for.
Locate things at specific addresses.
Make best use of structures.
Use bit-addressable memory.
Think in terms of chars rather than ints.
Get the best out of the various pointer types.
Get a modular structure into programs.
Access on and off-chip ports and peripherals.
Deal with interrupts.
Use registerbanks.
Deal with the stack.
Understand RAM overlaying.
Interface C to assembler code.
Use some of the special versions.
Use efficient C.
Help the optimiser to produce the smallest, fastest code.

The C51 Primer Will Not Help You:

1 Program in ISO C - get a good reference. Look on the Association of C and C++ Users web site (www.accu.org) where they have independent reviews of several thousand C, C++ and SW Engineering book reviews includinig an embedded section. Take care as many "C " books are actually PC-C books and specific to (usually Microsoft C or C++ these days)

NOTE:-Whilst many swear by the Kernighan & Ritchie book it is not really the best book to learn C for embedded use. The K&R book is more of a language definition; it was written over 25 years ago for UNIX programmers. It has now been superseded by the International ISO C standards in 1989 and 1999. The syntax used in the K&R First Edition is now obsolete and should not be used. The K&R 2nd Edition followed the ISO C

1989 standard and was written in 1987 (published in 1988) so you can see how old it is now. The language has moved on.

2 Write portable code - simply use the compiler without using any extensions. NOTE:- 100% portable code is difficult to write for the 8051 and will be inefficient. Although C is widely claimed as "portable" the vast majority of embedded applications will never be ported (other than to another, usually more powerful, part in the same family). Also to make good use of the system many 8051 specific extensions must be used. If you write portable C program it will probably be much slower and larger than it need be.

3 Set-up each and every on-chip peripheral on all of the 400 plus… well it was 400 at the time it is now (2006) about 600 different 8051 variants! Some are, however, covered in the appendices. I don't expect the number to rise much now as other parts havebcome more popular.

This guide should be read in association with a good C reference and is not meant to be a definitive work on the C language. It covers most of the Keil 8051-specific language extensions and those areas where the CPU architecture has an impact on coding approach.

0.1 History

The C51 Primer was first concived and written by Mike Beach in 1989 as a guide to both the 8051 and the Keil compiler it ran to some 70 pages. Since it's initial publication it has been given away with all Keil C51 compiler sold by Hitex, put on the Hitex web site and the Phadsys web site since 2002 when it was expanded from 70 to nearly 200 pages. IT can be found on numerous other web sites and has become one of the standard texts on the 8051.

Issue I	1991	M Beach	
Issue II	*not issued*	M Beach	
Issue III	1994	M Beach	Based on Keil C51 V3.02
Issue 3.5	January 2002	Chris Hills Major re-write	Revised for Keil C51 V6
Issue 3.6	October 2003	Chris Hills	Revised for Keil C51 V7 (and academic year)

One of the main changes since Issue III is the change in C syntax between C51 V4 and C51 V5. The declaration for variables before Version 5 was:

```
code unsigned char name;
xdata int name;
```

This was changed for version 5 to

```
unsigned char code name;
int xdata name;
```

bl (banked linker) is now standard
floating point maths improved

The other major visable change is the uVision IDE. The uVuision 1 series IDE was a 16 bit system that ran under Win 3.1 (and 9*, NT) This was available with Version 5 compilers. This was replaced by uVision 2 which was a completely new 32bit native system. The current IDE , uVision3, is a major re-write of uV2 but externally it looks very similar. It is not until you use it that you realise that there are many enhancements. All the tools remain command line driven. This permits their use, as in the past, with most forms of make, other IDE's and script files.

Disclaimer and contact details

This book has been written by several humans and therefore may have errors and omissions. Should you find any errors and omissions please email the current editor, Chris Hills at chills@phaedsys.org

Eur Ing **Chris Hills** BSc (hons) C. Eng, MIEE, MBCS, MIEEE, FRGS
Technical Specialist
PhaedruS SystmS

chris@phaedsys.org
http://www.phaedsys.org
http://Quest.phaedsys.org

January 2006

The Quest series at http://QuEST.phaedsys.org contains this paper and papers on Embedded C in genreral, Embnedded Debuggers, testing strratergy etc.

1 Introduction

C was the great universal language that all software engineers, programmers and hackers had to learn. However it was designed when memory was tight and long names were not used (and keyboard buffers were short). It tended to be a very terse language. Full of short cuts.

Whilst it is widely quoted as being a high level language, C does contain many such features that are in used in HLL structured programming, defined procedure calling, parameter passing, powerful control structures etc. However, much of the power (and danger) of C lies in its ability to allow direct access to the actual bits, bytes and words of the hardware. To a great extent, C is a high-level assembly language. As Andrew from Manchester said, in a bar in Germany, "C is not a programmer's language, it is a Software Engineers language. Pascal is a programmer's language." In other words C is a tool for a disciplined expert and lets the inexperienced programmer fall into many traps without warning.

Most programmers who are familiar with C will have been used to writing programs within large machines running MS-Windows, Unix, Linux or other RTOS. Even in the now cramped 640KB of MSDOS, considerable space was available so that the smallest variable in a program will usually be an int (at least 16 bits). Most interfacing to the real world will be done via system interrupts and operating system function calls. Therefore the C that was written is concerned only with the manipulation and processing of variables, strings, arrays etc. It rarely has to manipulate hardware.

In the modern 8-bit microcontroller, however, the situation is somewhat different. Taking the 8051 as an example, the total program size might only occupy 4 or 8K bytes and use only 128 bytes of RAM. Real devices such as ports, special function registers that access peripherals and directly accessing the hardware must be addressed by the application, usually in C. Though some still try and do in-line assembler in the C more of which later.

Interrupts have to be written and serviced, which require vectors at absolute addresses. Special care must be taken with a routine's data memory allocation if over-writing of data is to be avoided. One of the fundamentals of C is that parameters are passed to a function and results returned to the caller via the stack. This means a function can be called from both interrupts and the background process without fear of its local data being overwritten. The ability to call a function from several, possibly overlapping places is called re-entrancy. Many 8051 compilers do not do re-entrancy or at least require it to be explicitly enabled on a per function basis.

A serious restriction with the 8051 family is the lack of a proper stack; typically with a processor such as the 8086, the stack pointer is 16 bits (at least). Besides the basic stack pointer, there are usually other stack relative pointers such as a base pointer etc. The 8051 only has one 8-bit stack pointer register and one 16-bit Data Pointer (*some* derivatives have up to 8 DPTR's but they are not easy to use and their overhead makes it more sensible to think only in terms of 1 DPTR).

With these extra demands on the stack control system, the ability to access data on the stack is crucial. As already indicated, the 8051 family has a stack system which is really only capable of handling return addresses. With a *maximum* of only 256, 8 bit, bytes of stack potentially available, and typically around 40 bytes in practice, it would not take too much function calling and parameter passing to use this up.

This would seem to indicate that implementing a stack-intensive language like C on the 8051 would be impossible. Well, it very nearly has been! While there have been 8051 C compilers around for many years now, they were not very effective in the early days in fact even now for many of them. Most, particularly the open source and low end or free compilers, have actually been adapted from generic compilers originally written for more powerful micros such as the 68000, x86 etc eg GNU .

The approach to the stack problem has largely been through the use of artificial stacks implemented by using 8051 opcodes. Typically, an area in external RAM is set aside as a stack; special library routines manage the new stack every time a function is called. While this method works and gives a re-entrant capability, the price has been very slow runtimes and larger code. The net effect is that the processor spends too much time executing the compiler's own code rather than executing your program!

Besides the inherent inefficiency of generating a new stack, the compiled program code is not highly optimised to the peculiarities of the 8051. With this overhead, the provision of banked-switched expanded memory, controlled by IO ports became almost a necessity for some! Whilst most compilers and debuggers and ICE can now handle bank-switched memory well, it is not a route you should really expect to go down. Therefore, with the 8051 in particular, the assembler approach to programming had been the only real alternative for small, time-critical systems. In the last few years Philips, Analog Devices and a few others have put FAR pointers in to the 8051 and permitted extended linear memory ie not bank -switched, out to 8MB but this is the exception not the rule.

However, as far back as 1980, Intel produced a partial solution to the problem of allowing high-level language programming on its new 8051 in the shape of PLM51. This compiler was not perfect, having been adapted from PLM85 (8085), but Intel were realistic enough to realise that a full stack-based implementation of the language was simply not on. Note Intel discontinued PLM51 in 1986 at Version 1.4. PhaedruS SystemS still has a copy of this we use for compatibility testing.....

The solution adopted was to simply pass parameters in defined areas of memory. Thus each procedure has its own area of memory in which it receives parameters and passes back the results. Provided the passing segments are internal the calling overhead is actually quite small. Using external memory slows the process but is still faster than using an artificial stack.

The drawback with this "compiled stack" approach is that re-entrancy is now not possible. This apparently "serious omission" in practice does not tend to cause a problem with typical 8051 programs though IT programmers used to megabytes of memory recoil in horror at the thought of no re-entrancy. The later Keil C51 versions do allow selective re-entrancy, so that permitting re-entrant use of a few critical functions does not compromise the efficiency of the whole program. C on a microcontroller is practical for (among other things):

Control of on and off-chip peripheral devices
Servicing of interrupts
Easily supporting different ROM/RAM configurations
A very high level of optimisation to conserve code space
Control of register-bank switching
Support of enhanced or special family variants.

The Keil C51 compiler contains all the necessary C extensions for microcontroller use. This C compiler builds on the techniques pioneered by Intel but adds proper C language features such as floating point arithmetic, formatted/unformatted IO etc. It is, in fact, an implementation of the ISO C standard specifically for 8051 processors. It should be noted that the Keil C51 in common with (at the time of writing in early 2006) virtually all other compilers are ISO9899:1990 compliment that is C90 plus the main Technical Amendment (A1) and the Technical Corrigendum. (TC1 and TC2). The Keil compiler is not C99 compliant nor is it, or any other 8 or 16 bit target compiler ever likely to be. Untill some of the bugs in the C99 are fixted it is unlikelyu that any compiler will be complient to the C99 or later ISO c standartd.

This does render the possibility of writing portable ISO C. However, the resultant application would be somewhat large and slow compared to one that used the 8051 specific extensions. It is also possible that it would be too slow and large to actuall work on an 8051. ***It is for this reason that generic compilers that are ported to the 8051 are usually a very poor substitute for one that has been written for 8051 from the start.***

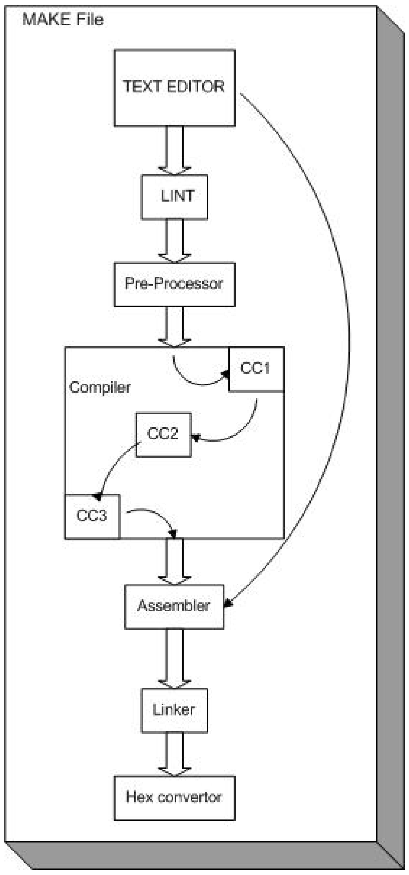
In some cases it has been found that the 2 or 4 or 8 K limited special version of the Keil C51 compiler can actually compile programs that are defeated by some of the unrestricted free, low end or generic compilers. One area in particular springs to mind. That of Data Overlaying. Data overlaying is the art of using the same physical memory to hold several different items of data on the grounds that the use of the data is mutually exclusive. This s something that should NEVER be done manually the slightest mistake causes data corruption. However by using full calling tree analysis it can be done. This is what the Keil compiler does, rigorously. Therefore the compiler can overlay a lot of the data. As the DATA space in an 8051 is limited this can make all the difference. It is often the DATA space that runs out before the code space. External data space is available but at a cost not only in speed of access but the hardware as well. At home it may not may much difference but in a commercial world a pound (or a couple of USD) may not seem like much but if your production run is over a thousand boards then it is a costly mistake on that one point alone.

The other point is the Keil C51 compiler, along with most other commercial compilers, are very well tested with industry standard test suites. Test suites that are well out of the reach of individuals and small companies. They are usually tested with the assistance of the chip manufacturers and other tool vendors.

2 Compiler Chain

There have been some changes in the compile process over the years Many years ago programmers used "terminals" these were simply display screens with a keyboard.. No intelligence, they certainly did not run programs. That is they could only display characters. Eg ASCII BAUDOT, EBSIDIC etc. They text based screen usually 32, 60 0r 80 characters by 25 or 40 lines. The 80 by 40 were "high resolution". As for colour, there was green text, or orange text or white text… that's right, monochrome! The highlight of graphics in those days was setting a character to reverse video, flash or bold! Wow! Sound? What do you think the "bell" character was for? The best you got was a keyboard "beep"

Editors were single window, ok everything was a single window and there was no multitasking on screen, but it was quite sophisticated. Some like VI and EMACS are still in widespread use today (2006). They had powerful, and hard to master, key bindings and macros whereby a Master could make multiple context sensitive replacements with a minimum of key strokes. An art that still amazes and users can still out-perform the average Windows user today.



Having edited and saved ones files the next step was to compile the. Usually by invoking:

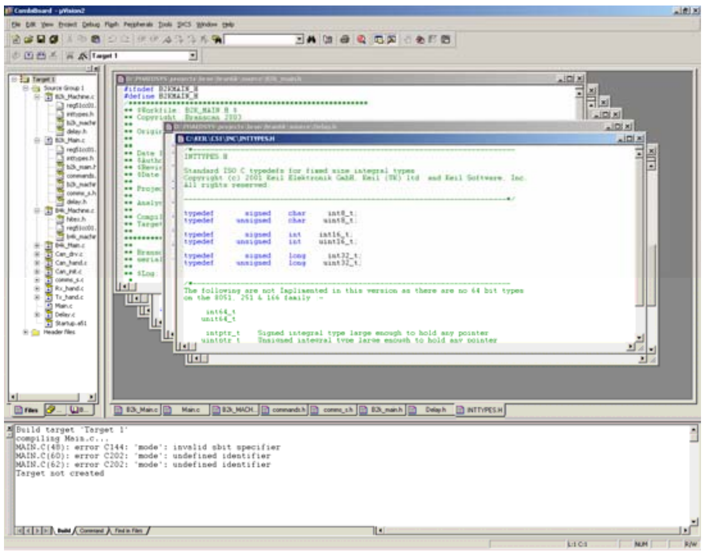
```
cc filename.c
```

on the command line... Windows users should think of a DOS-Box that fill the entire screen (no task bar and not mouse). Some of you may have noticed I missed out Lint and the pre-processor. Well yes and no. It depends on your system. In the very beginning CC was not the compiler. It was a script of batch file.

To build a program you would normally use MAKE: a program that would read a makefile for the project and process it. This would list all the c source files and the related header files for each. You did this manually. The project-build instructions would be included in the makefile. This would include Lint. It was assumed by Kernigan, Ritchie, Thompson and Johnson to be part of the compiler chain. NOTHING HAS CHANGED. You should ALWAYS use lint when compiling C or C++ for that matter.

The pre-processor whilst a separate module was usually included in the "cc" but as Lint has many more uses it was not. The "cc" compiler called the pre-processor and up to three compiler modules (i.e. two or three pass). These produced various intermediate files that were deleted during there run and not usually seen by the programmer. The compiler turned out assembly language. However some of these were two pass assemblers. This was required to work out the forward references. The assembly was assembled in to object code. The multiple object modules were then linked with the libraries to form an executable. The final line was a “clean” that removed all the intermediate files. In the very early days if it was a single file that was assembled, with no library files it did not need linking!

Whilst modern compiler systems seem very different they do have the same basic system under the IDE. It is the advent of modern GUI interfaces that brought about this change. Incidentally the modern GUI was Xwindows on Unix that led the way long before MS Windows. In fact the current Windows XP has features I used on Solaris over a decade before.



```
#ifndef B2KMAIN_H
#define B2KMAIN_H
/*****************************************************
** $Workfile: B2K_MAIN.H $
** Copyright  Brennan 2003
```

```
/*------------------------------------------------------------------
INTTYPES.H

Standard ISO C typedefs for fixed size integral types
Copyright (c) 2001 Keil Elektronik GmbH. Keil (UK) ltd  and Keil Software, Inc.
All rights reserved.

------------------------------------------------------------------*/

typedef       signed    char     int8_t;
typedef     unsigned    char    uint8_t;

typedef       signed    int     int16_t;
typedef     unsigned    int    uint16_t;

typedef       signed    long     int32_t;
typedef     unsigned    long    uint32_t;

/*----------------------------------------------------------------
The following are not Inplimented in this version as there are no 64 bit types
on the 8051, 251 & 166 family  -

   int64_t
  unit64_t

   intptr_t    Signed integral type large enough to hold any pointer
  uintptr_t    Unsigned integral type large enough to hold any pointer
```

```
Build target 'Target 1'
compiling Main.c...
MAIN.C(48): error C144: 'mode': invalid sbit specifier
MAIN.C(60): error C202: 'mode': undefined identifier
MAIN.C(62): error C202: 'mode': undefined identifier
Target not created
```

In a modern compiler system such as the Keil C51 the IDE is simply (?) an editor and project control system. In the case of the Keil system and many others it also has the simulator/debugger and handles a lot more. The compiler and linker etc are still called by command line or scripts though this is invisible to the user.

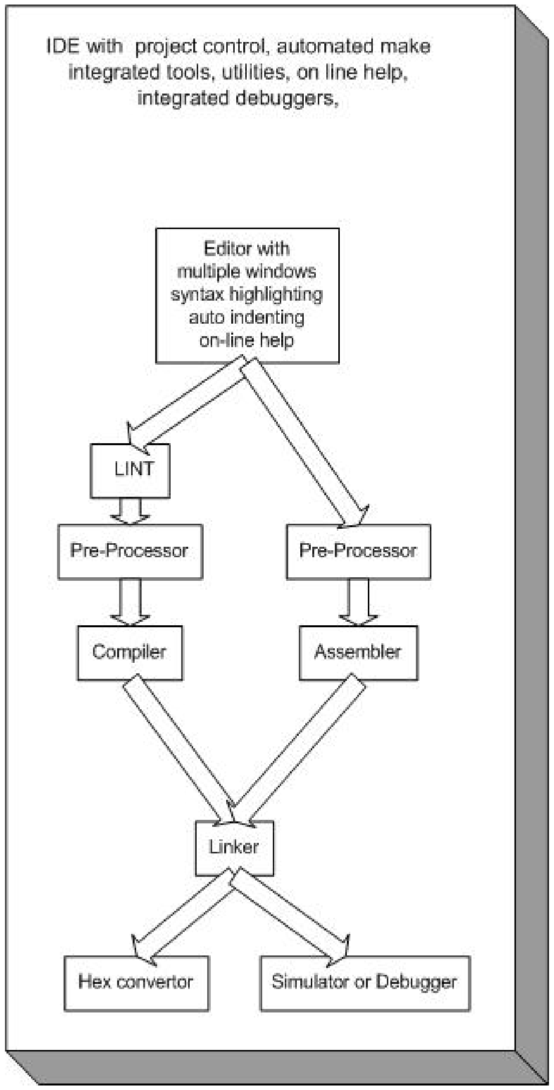
The compiler is a single pass compiler that outputs object code ready for linking. The C51 incorporates the pre-processor, cc1,2 and 3 in the same program. It also, normally skips the assembly phase producing object code ready for linking.

As we will see in later chapters it is still possible to get the compiler to turn out assembler but this is the exception rather than the rule. It is at the linker phase that the standard and user libraries are linked in to the program.

With the modern compiler system it is still easily possible to have both C and assembly modules in the same project. A project "Build" using the Keil uVision IDE is still a single mouse click whether it is all C, all assembler or mixed assembler and C modules. Indeed the Keil C51 compiler suite will still work with the Intel PL/M compiler making it possible to mix PL/M, C and Assembler modules. This makes the Keil Environment suitable for transitioning from PLM or assembler to C for legacy projects. However it should be noted that inline assembler should eb avoided where possible.

Students, and command line enthusiasts, should note that in the bottom window of the Target -> Options dialogues is the command line that the Keil IDE feeds to the compiler and linker. Students should try, at least once to use these strings in a batch file to build one program to see how the system works under the hood. Understanding the mechanisms will help you know what is happening and can assist when things don’t go as expected.

In short the IDE simple collects and automates many tools into one interface. A knowledge of what is under the hood is useful.


IDE with project control, automated make integrated tools, utilities, on line help, integrated debuggers,
Editor with multiple windows syntax highlighting auto indenting on-line help
LINT
Pre-Processor
Pre-Processor
Compiler
Assembler
Linker
Hex convertor
Simulator or Debugger

Do note that whilst there is a Lint on the compiler side there is no equivalent on the assembler side. This is because whilst there is a structure to C with if, else, while, do, for controls and matching between function prototypes, definitions, and uses there is nothing similar for assembler. The assembler can be syntactically correct and do nothing but it is impossible to spot mechanically.

As a final point ALL C compilers *require* the "Hello World" program to be run as an initialisation program. With Keil compilers it could be "Blinky" it depends if you have an LED or a serial port… actually it is probably easier to solder an LED on to a PCB than find a PC with an RS232 port on these days.

OK, so technically it is not true that you must run hello world. However it really is a Very Good Idea to run Hello World with printf, and that is the ONLY time you should use printf in an 8051 program). Why? It is because it makes sure the compiler, dongle/license, libraries etc are all correctly installed. If it compiles and an debug the examples you know that you have a good chance that the compiler is correctly installed.

Then do run a simple test project of your own just to make sure. There have been quite a few times where the problem has been an incorrect installation.

3 C51 Basics - The 8051 Architecture

The Keil C51 compiler has been written to allow C programmers to get code running quickly on 8051 systems with a short learning curve. However, to get the best from the 8051 family, some appreciation of the underlying hardware is desirable. The most basic decision to be made is which memory model to use. For general information on the C language, number and string representation, please refer to a standard C textbook.

3.1 8051 Memory Configurations

The physical memory layout of the 8051 is Harvard where most "normal" computers use Von Neuman. However, several silicon vendors have further confused this with on-chip "external memory" (ie internal external memory!). During 2001, which is “recent” in terms of 8051 history, several silicon vendors brought out a new memory configuration that can address up to 8M bytes of memory without bank-switching. We will start with the traditional 8051 memory map. All others are a superset of this one.

3.1.1 Physical Location Of The Memory Spaces

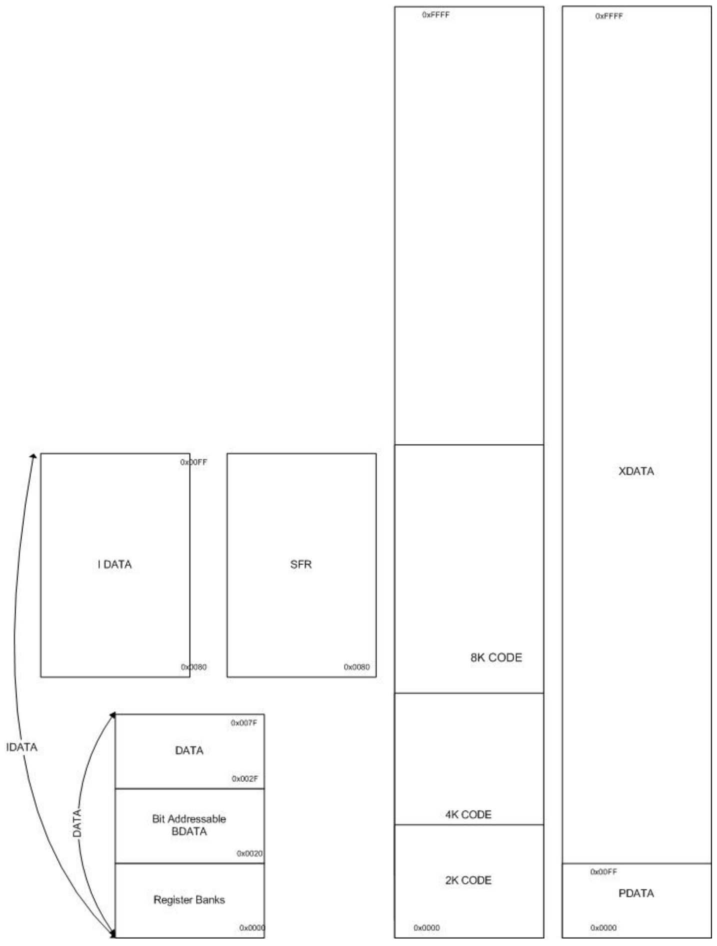


Fig.1. The 8051's Memory Spaces.

Initially perhaps the most confusing thing about the 8051 is that there are three (sometimes four) different memory spaces, all of which *appear* to start at the same address. The CODE and the DATA memory are distinct and separate. This is Harvard architecture. Most programmers are used to the Von Neuman memory configuration, used in most other microcontrollers, such as the 68HC11. This is a single plane memory where areas are located at sequential addresses.

Within the 8051 CPU the first area is the "DATA". This is on-chip RAM. This is box 1 in fig 1. This starts at D:0x00 (the ‘D:’ prefix implies DATA segment) and ends at 07fH (127 decimal). This RAM can be used for program variables. It is directly addressable, so that instructions like ‘MOV A,x’ are usable.

The problem is that as it is the fasted memory it is also used for other things.

The bottom 48 bytes of the DATA space are reserved. The first 8 bytes are registers R0-R7. These are all general purpose 8 bit registers. There are 4 identical sets of registers called Banks. Bank 0 -Bank3 from D:00 to D:1F These are automatically switched by the compiler unless manually over ridden by the user.

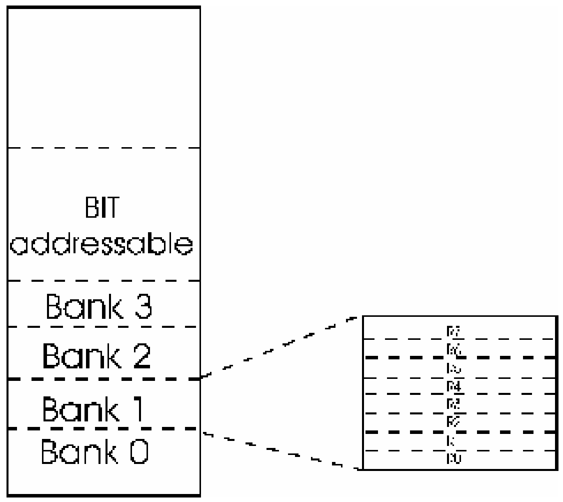


Fig 2. The DATA memory Space

There are 16 bytes of bit addressable RAM from D:20 to D:2F . This is where 8051 C starts to deviate from the ISO C standard. There is no type called "BIT" in C. There are bit fields in variables but no stand alone bit type. This is why truly portable ISO C is not efficient for 8051 and efficient 8051 code is not portable.

Above 80H the special function registers are located. These run from 80h to FFh. The SFR's, like the DATA block are directly addressable. This is the SFR box in fig 1.

The SFR's are usually addressed by symbolic names such as **SYSCON** (these names are set up in header or include files). Many of the SFR's are defined in the standard 8051 architecture. The spaces between the standard SFR's are used by 8051 manufactures for their own use such as CAN interfaces, USB, A to D and many other peripherals.

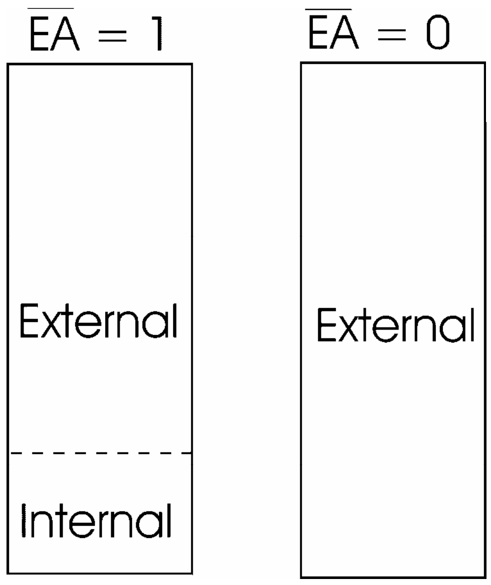
As the SFR block is not really conventional RAM but a series of hard wired registers. Where there is no SFR defined there is no empty register or memory byte for the user. Another "convention" is that if an SFR address ends in 0 or 8 the bits in the register are directly and individually addressable. This means that bits can be set without have to mask and write the whole byte.

However only the standard set of SFR's are fixed which means that whilst the individual silicon manufacturers have their own standards they are not portable. So the CAN registers in a Philips 8051 will probably not be in the same place as the CAN registers in an Atmel part.

A **second** memory area exists between 80H and 0FFH. This is the IDATA space. The IDATA is only indirectly addressable (MOV A,@Ri) and is prefixed by I: This is box 2 in fig1. and it effectively overlays the directly addressable SFR area. This constitutes an extended on-chip RAM area and was added to the 8051 design when the 8052 appeared. As it is only indirectly addressable, it is best left for stack use, which is, by definition, always indirectly addressed via the 8-bit stack pointer SP.

Just to confuse things, the normal directly addressable DATA RAM from 0-80H can also be indirectly addressed by the MOV A,@Ri instruction! Therefore the whole area from 0 to 0xff is IDATA but the bottom 128 bytes is also DATA and the bit it the middle of the DATA is BDATA (and the Register banks)

Having sorted out al that... There is a **third** memory space, the **CODE** segment, box 3 in fig1. This also starts at C:zero, but this is reserved for the program CODE. It typically runs from C:0000 to C:0FFFFH (65536 bytes). The CODE segment is accessed via the program counter (PC) for opcode fetches and by DPTR for data, both registers being 16-bit registers. Obviously, normally being ROM/FLASH/EEPROM etc, only constants can be stored here. However with the advent of FLASH it is possible to change data in the CODE Space. Some new parts permit the application to load new blocks of code via an ISP interface. The Atmel parts now have boot loaders that will work via the ISP, Serial or CAN interface.



In the original 8051 the CODE space on chip and was 4K of either ROM or EPROM. In the 8052 the on chip CODE space was 8K ROM. Though both of these parts could access additional off chip ROM. The 8031 and 8032 had off chip CODE space and no on chip code memory.

The modern 8051 variants, over 600 of them, have all manner of on chip ROM, OTP, EPROM, EEPROM and FLASH from 2k to 64K. In 2000 Philips announced plans for more than 64K of on chip FLASH CODE space and they have produced parts with 256K flash on them.

There are also many variants that only have off chip CODE space. Off chip memory is addressed by using ports 0 and 2 for data and address lines. With external memory these ports can not be used for any other purpose which rather restricts the capabilities of the part.

On parts that have internal or on-chip CODE space there is a way of selecting to use the internal or external memory. This is achieved by the EA line. When EA =1 the internal memory is used until the end of the internal memory is reached. If the internal memory is less than 64K external memory will be accessed above the internal space.

It is for this reason that if the internal memory is less than 64K and there is no external memory the last byte of internal CODE space should not be used. If it is the PC (program counter) will increment to the "next instruction" which is external. This will make ports 0 and 2 act as the address and data bus. This can wreak havoc if the ports are used as IO.

If the EA is set to 0 only the external memory is used. The EA pin is *usually* tied high or low and not toggled by the program.

It has generally been possible to have more than 64K of CODE space. This is done by using I/O lines from a port as additional adress lines to switch overlayed blocks of memory. Usually in the range 32k to 64K with common code in 0 to 32k

A fourth memory area is also off-chip, eXternal DATA, starting at X:0000. This is box 4 in fig1.This exists in an external RAM device and, like the C:0000 segment, can extend up to X:0FFFFH (65536 bytes). The ‘X:’ prefix implies the external XDATA segment. The 8051’s only 16-bit register, the DPTR (data pointer) is used to access the XDATA. When using off chip CODE and or XDATA ports 0 and 2 are used to provide the multiplexed address and data lines.

Finally, 256 bytes of XDATA can also be addressed in a paged mode This is box 5 in fig1. Here an 8-bit register (R0) is used to access this area, termed PDATA. When accessing PDATA only port 0 is used.

You may have noticed the **EDATA** block on the diagram. This is a new 8051 extension used by Philips. This is Extra **DATA**, it has also been known as AUX DATA. It is 256 bytes (and 768 bytes in some parts) This block of memory has been included here to illistrate the point that whilst the 8051 core memory, DATA, SFR, IDATA, XDATA and CODE are well defined the dozen or so 8051 manufacturers add their own extensions from time to time. Siemens (now Infineion) has some 8051's with on-chip (currently) XDATA.....

The obvious question is "*How does the 8051 prevent an access to D:00 resulting in data being fetched from C:0000?*" The answer is in the 8051 hardware. When the cpu intends to access D:00, the on-chip RAM is enabled by a purely internal READ signal - the external /RD pin is unchanged. The following examples are all in assembler as C hides this addressing process. Note the /RD and /WR pins are shared wirth port 3 pins 6 and 7 These can only have one purpose and may not chagne once their use has been established.

```
        MOV   A,40      ; Put value held in location 40 into the accumulator
```

This addressing mode (direct) is very fast and the basis of the SMALL memory model.

```
        MOV  R0,#0A0H   ; Put the value held in IDATA location 0A0H into
        MOV  A,@R0       ; the accumulator
```

This addressing mode is used to access the indirectly addressable on-chip memory above 80H and as an alternative way to get at the direct memory below this address.

A variation on DATA is **BDATA** (bit data). This is a 16 byte (128 bit) area, starting at 020H in the direct segment. It is useful in that it can be both accessed byte-wise bythe normal MOV instructions and addressed by special bit-orientated intructions, as shown below:

```
        SETB  20.0  ;
        CLRB  20.0  ;
```

The external CODE memory device at C:0000 is not enabled during data RAM access. In fact, the CODE memory is only enabled when a pin on the 8051 named the PSEN (program store enable) is pulled low. There is an internal equivelent when using on-chip CODE memory. The XDATA RAM and CODE EPROM do not clash as the XDATA device is only active during a request from the 8051 pins named READ or WRITE.

To help access the external XDATA RAM, special instructions exist, conveniently containing an 'X'....

```
        MOV    A,#40h          ; Direct internal data move of 0x40 into the A register
        MOV   DPTR,#08000H     ; Direct internal data move of 0x08000 into the 16-bit 
DataPoinTeR
        MOVX  A,@DPTR          ; "Put a value in A  (40h) in to the external RAM, whose 
address is
                                       ;   contained in the DPTR register  (8000H)".
                                       ;  ie  put 40h in to external data address 08000h.
```

The above addressing mode forms the basis of the LARGE model.

```
        MOVX  R0,#080H  ;
        MOVX  A,@R0     ;
```

This alternative access mode to external RAM forms the basis of the COMPACT memory model. Note that if Port 2 is attached to the upper address lines of the RAM, it can act like a manually operated “paging” control.

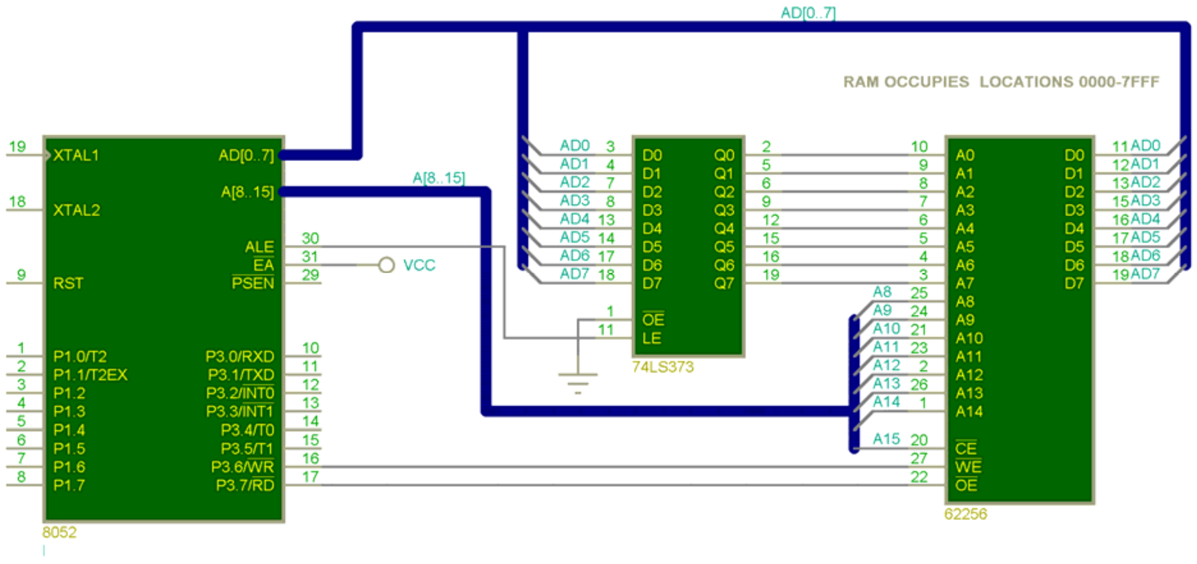
The important point to remember is that the PSEN pin is active when (CODE) instructions are being fetched; The external READ and WRITE are active when MOVX.... (“move external”) DATA instructions are being carried-out.

3.2 Hardware Memory Models

Although we are concerned with the software the hardware is never far away. This section will show the three basic hardware set ups. External XDATA, external CODE and the Von-Neumen method for being able to write to the (external) CODE space so that boot loaders can write code to memory. (Note this method is not used for internal FLASH or OTP code memory)

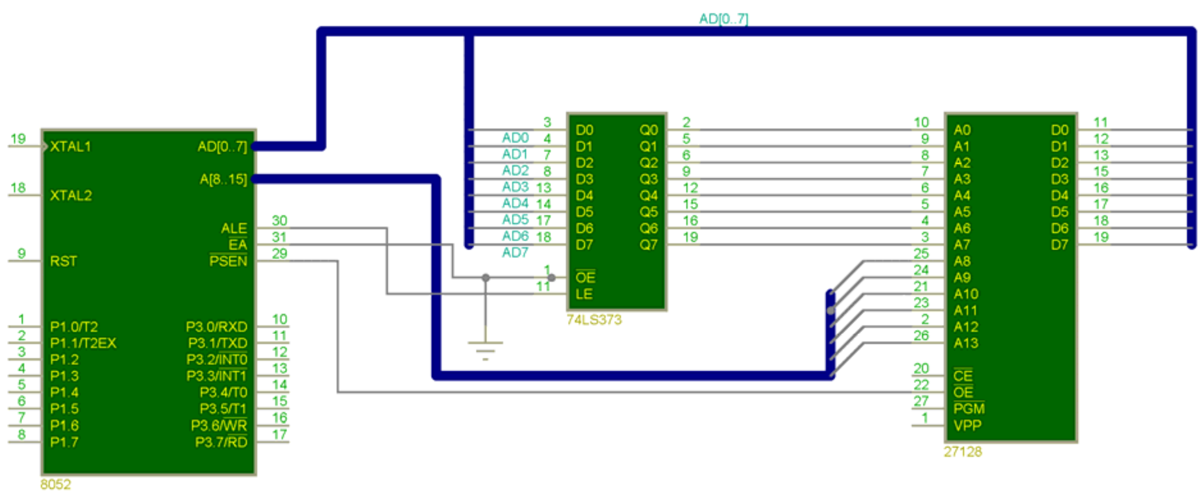
3.2.1 External DATA

This is the basic wiring for external data. Port 0 has 8 bits of the Data but also the lower 8 bits of the address bus. Thus to de-multiplex the bus a latch must be used. In this case a 74LS373. The ALE is used to latch the address. Thus Port 2 and the *output* of the latch make up the 16 bits of the address. Then Port 0 is the 8 bit data bus. Note in this case the CODE is internal as the EA is held high.



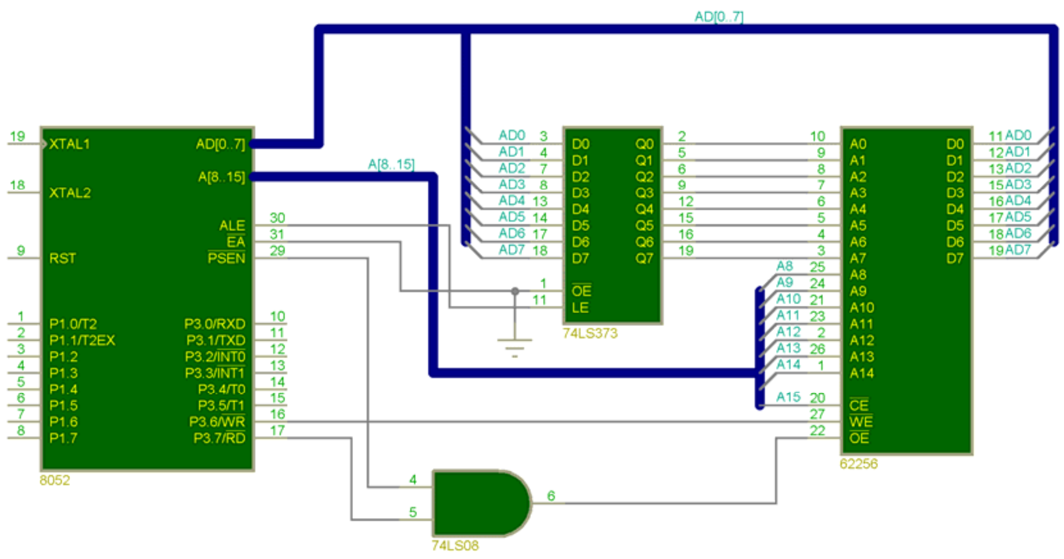
3.2.2 External Code

The diagram here is for external CODE in EPROM or other write only memory. This is very similar to the XDATA but the EA is held low. There is no Read or Write but PSEN is used for the chip enable or output. The address decoding is exactly the same as the external RAM or XDATA diagram. For External CODE and XDATA the two diagrams can be combined.



3.2.3 Write to CODE Space

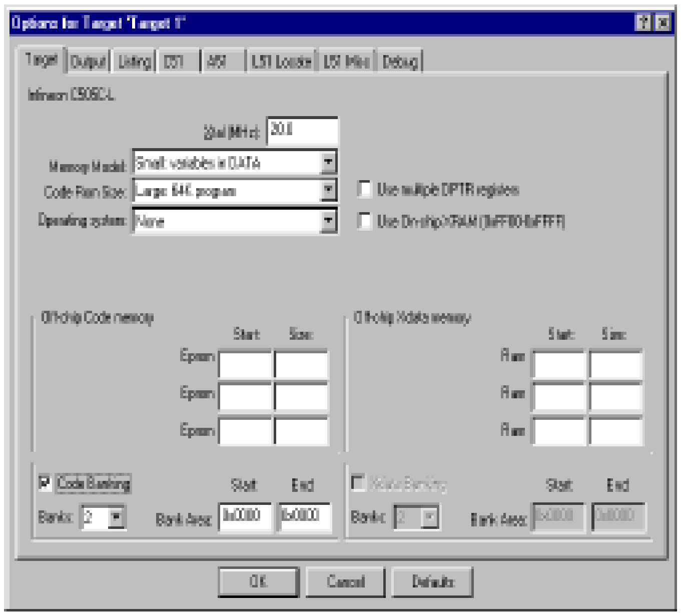
There are cases where you will want to write to CODE space. I.e. when a monitor is used or boot loading of CODE. In this case the memory has to be in both CODE and DATA space. This requires Von Neumen architecture. This is done by ANDing the PSEN and the WR lines.



3.3 Possible Memory Models

With a microcontroller like the 8051, the first decision is which memory model to use. Whereas the PC programmer, with a flat Von Neuman memory, chooses between TINY, SMALL, MEDIUM, COMPACT, LARGE and HUGE to control how the processor segmentation of the RAM is to be used (some may say "overcome!"), the 8051 user has to decide both the program and data models.

Not only, as in the PC, are address and pointer ranges considered but data locations and storage strategies for the several 8051 data memories. Some memories are best used for direct and others indirect addressing. Also frequently used variables will be better in ram that is accessed fast where as other data may be quite happy in slower memory and in the case of constants (eg look up tables) they can be put in to the ROM.



Infineon C505C-L

Xtal (MHz): 20.0

Memory Model: Small: variables in DATA

Code Rom Size: Large: 64K program

Operating system: None

☐ Use multiple DPTR registers

☐ Use On-chip XRAM (0xFF00-0xFFFF)

Off-chip Code memory

	Start:	Size:
Eprom		
Eprom		
Eprom		

Off-chip Xdata memory

	Start:	Size:
Ram		
Ram		
Ram		

☑ Code Banking — Banks: 2 — Bank Area: Start 0x0000 End 0x0000

☐ Xdata Banking — Banks: 2 — Bank Area: Start 0x0000 End 0x0000

This often seems somewhat overwhelming. However, if taken step at a time it is not. A basic understanding will cover most usage. It is really only when pushing the limits in-depth knowledge is required.

However, some thought before and durring programming will pay off in the long run. As with all the settings they can be obtained by point and click in the uVision IDE, by command line and, optionally, by `#pragma` in the source files themselves.

The memory model settings in the Keil uVision IDE are found by right clicking on the "Target" in the project window and selecting "options" onthe pop-up menu.

3.3.1 ROM Memory Models

Firstly we shall look at the ROM memory models. These are for the CODE space. This does not include any data (unless constants have been placed in to CODE space as follows:

```
        unsigned char code constant_1 =3;
        unsigned char code array_1[3] = {'1','2','3'}
```

Multi dimensional arrarys are also permited

```
        unsigned char  code array[3][5] = {
        {'a', 'b', 'c', 'd', 'e'}, {'1', '2', '3', '4', '5'}, {'A', 'B', 'C', 'D', 'E'}
```

```
	};
```

At run time you can not use array as an l-value, that is

```
	array[1][1] = 'X';
```

is illegal since it's in ROM. Do remember that for arrays in C the firsr ellerment is always 0. So the above array has elerments 0,0 to 2,4.

3.3.1.1 ROM SMALL

This is used where the total program CODE size is less than 2K In this mode all assembler CALL and JMP are coded as ACALL and AJMP. These are smaller and faster instructions that the LCALL and LJMP. Thus smaller and faster code is produced. For some of the smaller 8051 family members this is an ideal model.

3.3.1.2 ROM COMPACT

Compact is used where the the program CODE may be up to 64K but no function will be larger than 2K. In this modle all CALL instructions are coded as the longer LCALL but the JMP instructions remain as the shorter and faster AJMP.

3.3.1.3 ROM LARGE

The LARGE model sets both the CALL and the JMP at the longer and slower LCALL and LJMP. In this model the program may be 64K as in the compact but the functions can be over 2K (in fact they could be up to 64K) It is up to the programmer to work out if there is any saving to be gained by using the COMPACT model over the LARGE. It is not usually worth spending a great deal of time agonising over.

3.3.2 RAM Memory Models

Having chosen the ROM model for the CODE we shall move on to the RAM DATA memory models. These are a little more complex than the ROM models and will require a little more thought. Note that whilst these models are, like the ROM models global it is possible to locally place data in specific membory and specific C functions into a different model. This will be explained later.

The Source Browser (under the View menu on uV2) will be useful to see what variables are in which data space.

3.3.2.1 RAM SMALL

This is the fastest model. All variables will reside in the internal data. However internal data is often limited to 256 bytes, including register banks and stack. Of this 265 bytes only the first 128 bytes are directly addressable. The second 128 are only indirectly addressable. If all the CODE memory is on chip this is the smallest and fasted configuration. For the 8051/31 specifically there is no IDATA so the total amount available is 128 bytes of DATA. Only the 128 Bytes of directly addressable DATA will be available using this memory model even when using an 8052 derivative.

Note that when optimising the C51 compiler will use Register banks 1-3 as ordinary memory if they user is not explicitly using them, usually for inrterupts. The "Overlay Variables" compiler switch should be used if possible in this model. Note that the stack size is critical in this model and therefore will limit the nesting of C functions. Therefore few, larger functions may be better than lots of small ones that could lead to deeper nesting. Not only is this the fastest model it also results in smaller code, as the addessing is direct or 1-byte pointers.

This model also tends to lead to occurrences of the Linker warning L128 complaining about data segment overflow.

3.3.2.2 RAM COMPACT

The COMPACT model uses the PDATA bank for variables. This is the first 256 bytes of XDATA. It is addressed via Port 0 using indrect addressing through R0 and R1. Whilst this may seem to give no more memory than the SMALL model remember the stack will still be in IDATA and in the DATA memory the first 48 bytes are reserved for the register banks and bit addressable data. Also as we will see later specific data can be forced back into the DATA area.

3.3.2.3 RAM LARGE

The LARGE model puts all the data into XDATA and uses the 16-bit DPTR to access them indirectly. This is less efficient than the other forms of addressing and also uses longer, slower instructions. This makes the CODE larger as well as slower.

This mode gives up to 64K of data space. Though XDATA can be banked and is available in some 8051 derivatives with a greater range than 64K using special commands.

3.3.3 Choosing The Best Memory Configuration/Model

With the memory models, a decision has to be made as to which one to use. With the choice of three ROM or CODE sizes and three DATA models there are potentially 9 variations! In fact with the ability to change the model locally there are an infinite number of models but that will be covered later. IHowever, thiongs are not as bad as they first appear. The first choice is the ROM. This should be relatively simple. Especially as it is not a problem to change this, just point and click in the output tab on the KEIL IDE. The changes here are relatively minor. With the selection of the DATA models it is more problematic as two of the modes require XDATA which would require additional physical memory and the loss of ports 0 and 2.... not a minor change in the design! Some parts now have on-chip EDATA or XDATA which makes the dilema a little easier.

Single chip 8051 users may only use the SMALL model, unless they have an external RAM fitted which can be page addressed from Port 0 and optionally, Port 2, using MOVX A,@R0 addressing. This permits the COMPACT model. While it is possible to change the global memory model half way through a project, it is not recommended!

There are other versions of the 8051 family that now have large amounts of additional "Aux", "Extra" or additional "on chip Xdata"memory on chip that can be used with the other memory models so single chip may not always mean small memory modle these days.

As with the ROM model selection the data memory model can be selected in the IDE is is also possible to force individual modules, functions and variables into spesific data models. This will be covered later.

3.3.3.1 SMALL :- Total RAM 128 bytes (8051/31)

This model is rather restricting in the case of 8051/31 especially as they do not have IDATA, only the 128 bytes of DATA. The SMALL model will support code sizes up to about 4K but a constant check must be kept on stack usage. The number of global variables must be kept to a minimum to allow the linker OVERLAYer to work to best effect. With 8052/32 versions, the manual use of the 128 byte IDATA area above 80H can allow applications up to about 10-12K but again the stack position must be kept in mind.

Very large programs can be supported by the SMALL model by manually forcing large and/or slow data objects in to an external RAM, if fitted. Also variables which need to be viewed in real time are best located here, as dual-ported memory emulators, like the Hitex range, can read their values on the fly. This approach is generally best for large, time-critical applications, as the SMALL global model guarantees that local variables and function parameters will have the fastest access, while large arrays can be located off-chip.

3.3.3.2 COMPACT :- Total RAM 256 bytes off-chip, 128 or 256 bytes on-chip.

Suitable for programs where, for example, the on-chip memory is applied to an operating system. The compact model is rarely used on its own but more usually in combination with the SMALL switch reserved for interrupt routines. COMPACT is especially useful for programs with a large number of medium speed 8 bit variables, for which the MOVX A,@R0 is very suitable.

It can be useful in applications where large stack is required, meaning that data needs to be off-chip. Note that register variables are still used, so the loss of speed will not be significant in situations where only a small number of local variables and/or passed parameters are used.

3.3.3.3 LARGE :- Total RAM up to 64KB, 128 or 256 bytes on-chip.

Permits slow access to a very large memory space and is perhaps the easiest model to use. Again, not often used on its own but in combination localised use of the SMALL model. As with COMPACT, register variables are still used and so efficiency remains reasonable.

3.3.4 What data goes where?

In summary, there are five memory spaces available for data storage, each of which has particular pros and cons. Keywords can be used to place specific variables in specific memory locations overriding the global memory model. This gives the Software Engineer the ability to fine tune the porgram of a very high degree. Here are some recommendations for the best use of each:

DATA:- 128 bytes (SMALL model default location)
Best For:
Frequently accessed data requiring the fastest access. Interrupt routines whose run time is critical should use DATA, usually by declaring the function as "SMALL". Also, background code that is frequently run and has many parameters to pass. If you are using re-entrant functions, the re-entrant stacks should be located here as a priority.
Worst For: Any variable arrays and structures of more than a few bytes.

IDATA:- 256 Bytes indirectly addressed

Idata has a range of 256 bytes an can use instructions such as MOV @ Ri, A where Ri may be R0 or R1. Note that the lower 128 bytes of IDATA is the DATA space.

Best For:
Fast access data arrays and structures of limited size (up to around 32 bytes each) but not totalling more than 64 or so bytes. As these data types require indirect addressing, they are ideally placed in the indirectly addressable area. It is also a good place to locate the stack, as this is by definition indirectly addressed.

Worst For: Large data arrays, fast access words.

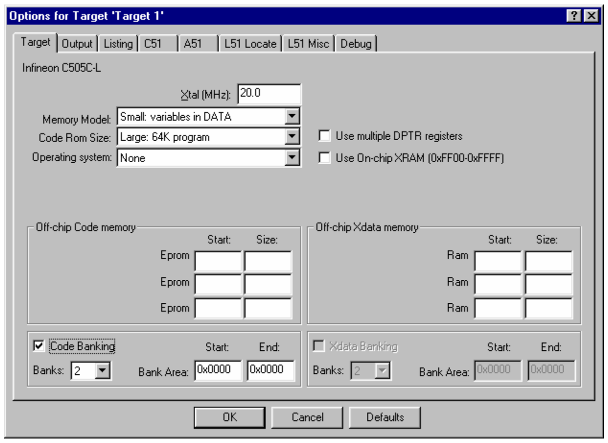
CODE :- 64K (or more with banking)
Best For: Constants and large lookup tables, plus opcodes, of course!
Worst For: Variables.

PDATA :- 256 bytes in paged XDATA (COMPACT model default area)
Best For:
Medium speed interrupt and fast background char (8 bit) variables and moderate-sized arrays and structures. Also good for variables which need to be viewed in real time using an emulator.
Worst For:

Very large data arrays and structure above 256 bytes. Very frequently used data (in interrupts etc..). Integer and long data.

XDATA :- 64K (LARGE model default area)
Best For:
Large variable arrays and structures (over 256 bytes). Slow or infrequently-used background variables. Also good for variables which need to be viewed in real time using an emulator.
Worst For:
Frequently-accessed or fast interrupt variables.

3.4 Setting The Memory Model



Infineon C505C-L

Xtal (MHz): 20.0
Memory Model: Small: variables in DATA
Code Rom Size: Large: 64K program
Operating system: None
☐ Use multiple DPTR registers
☐ Use On-chip XRAM (0xFF00-0xFFFF)

Off-chip Code memory	Start:	Size:
Eprom		
Eprom		
Eprom		

Off-chip Xdata memory	Start:	Size:
Ram		
Ram		
Ram		

☑ Code Banking — Banks: 2 — Bank Area: Start: 0x0000 End: 0x0000
☐ Xdata Banking — Banks: 2 — Bank Area: Start: 0x0000 End: 0x0000

The overall memory type is selected in the Keil IDE, uVision The memory model settings are found by right clicking on the "Target" in the project window and selecting "options" on the pop-up menu.

This menu will give you the "point and click" simplicity for setting the *global* ROM and RAM memory models.

This method is also possible in the older uVision 1 series IDE.

Before the uVision IDE became common the memory model was set in each file using a #pragma. This option is still permited for localised setting of the memory models.

#Pragma Usage

By including the line “#pragma SMALL” as the first line in the C source file. The (RAM) small memory model will be set for the whole source file the #pragma is in. See Section 2.1.3 for details on specific variable placement. SMALL is the default model and can be used for quite large programs, provided that full use is made of PDATA and XDATA memory spaces for less time-critical data.

Special note on COMPACT model usage
The COMPACT model makes certain assumptions about the state of Port 2. The XDATA space is addressed by the DPTR instructions which place the 16 bit address on Ports 0 and 2. The COMPACT model uses R0 as a 8 bit pointer which places an address on port 0. Port 2 is under user control and is effectively a memory page control. The compiler has no information about Port 2 and unless the user has explicitly set it to a value it will be undefined, although generally it will be at 0xff. The linker has the job of combining XDATA and PDATA variables and unless told otherwise it puts the PDATA (COMPACT default space) at zero. Hence, the resulting COMPACT program will not work.

It is therefore essential to set the PPAGE number in the startup.a51 file to some definite value - zero is a good choice. The PPAGEENABLE must be set to 1 to enable paged mode. Also, when linking, the PDATA(ADDR) control must be used to tell L51 where the PDATA area is, thus:

```
L51 module1.obj, module2.obj to exec.abs PDATA(0)XDATA(100H)
```

Note that the normal XDATA area now starts at 0x100, above the zero page used for PDATA. Failure to do this properly can result in very dangerous results, as data placement is at the whim of PORT2!

3.5 Local Memory Model Specification

3.5.1 Overview

From C51 version 3.20 it has been possible for memory models to be assigned to individual functions. Within a single module, functions can be declared as SMALL, COMPACT or LARGE thus:

```
#pragma COMPACT
/* This has set the whole file to COMPACT */
/* A SMALL Model Function */
void fsmall(void) small
{
       printf(“HELLO”) ;
}

/* This function has no memory modifier and is COMPACT
** As per the pragma at the top of the file
*/
void func(void)
{
       printf(“HELLO”) ;
}

/* A LARGE Model Function */
void flarge(void) large
{
       printf(“HELLO”) ;
}
/*Main  Caller */
void main(void)
{
       fsmall() ;  /* Call small function */
       flarge() ;  /*Call large function  */
       func();    /call to compact function */
}
```

In the example above the main() and any other function not specifically changed is COMPACT, fsmall is locally set to SMALL and flarge is locally set to LARGE

Note on main:- Whilst the ISO C standard defines 2 mains :-

```
int main (argv, argc)
```

and

```
int main (void )
```

In self hosted embedded systems i. e. where there is no operating system "void main (void)" is permitted.

3.5.2 Point To Watch In Multi-Model Programs

A typical C51 program might be arranged with all background loop functions compiled as COMPACT, whilst all (fast) interrupt functions treated as SMALL. The obvious approach of using the #pragma MODEL or command line option to set the model can cause odd side effects. The problem usually manifests itself at link time as a “MULTIPLE PUBLIC DEFINITION” error related to, for instance, putchar().

The cause is that in modules compiled as COMPACT, C51 creates references to library functions in the COMPACT library, whilst the SMALL modules will access the the SMALL library. When linking, L51 finds that it has two putchars() etc. The solution is to stick to one global memory model and then use the SMALL function attribute, covered in the previous section, to set the memory model locally. Example:

```
#pragma COMPACT
void fast_func(void) SMALL{
        /*function code*/
}
```

from two different libraries

4 Declaring Variables and Constants

4.1 Constants

The most basic requirement when writing any embedded program is to know how to allocate storage for program data. Constants are the simplest as they can reside in the code (EPROM,ROM,OTP etc) area or as constants held in RAM and initialised at runtime. Obviously, the former really are constants and cannot be changed.

While the latter type are relatively commonplace on big systems (Microsoft C), in 8051 applications the code required to set them up is often best used elsewhere. Also, access is generally faster to ROMmed constants than RAM ones if the RAM is external to the chip, as ROM “MOVC A,@DPTR” instruction cycle is much faster than the RAM “MOVX A,@DPTR”.

Examples of EPROMed constant data are:

```
     unsigned char code coolant_temp = 0x02 ;
     unsigned char code look_up table[5]=‘1’,’2',’3',’4''} ;
     unsigned int    code pressure = 4 ;
```

Note that “const” does not mean "code". Objects declared as "const" (but not CODE) will actually end up in the data memory area determined by the current memory model.

```
const unsigned char tens[] = { 1, 10, 100, 1000 };
```

will be stored in the heap, never on the stack if it is outside the scope of a function. It will exist in DATA if you use the small model and XDATA if you use the large model.

The keyword const is a compiler system prevents the data being changed by the program. The key word CODE stops the data being changed by the fact that it is physically impossible to write to CODE space. (We will see later that it is possible to wire CODE space so it can be written to. Also some FLASH parts do permit the changing of CODE space. However code space should not normally be changed.)

Tip:- Use "const" in function prototypes when passing in a value that should not be changed in the function. eg func(const char count).

An associated key word is "volatile". This is used where a variable is not changed by an application but by the hardware. Thus a declaration like

```
 char volatile variable_name;
```

will create a variable may be updated by "something" other than the application. This stops the compiler optimising out the variable reads where apparfently nothing has written since the last read.. The Special Function Registers (SFR) are by default "Volatile".

One, less obvious, case in point was a memory test routine that wrote to to the memory and then imediately read it back. The compiler, not a Keil one, "speeded up" the memory test by optimising out the write and subsiquent read to the memory location. The memory test was fast... but it never touched the physical memory. Defineing the variable as "Volatile" forced the compiler to give the behaviour required.

Note the declaration

unsigned char volatile const name;

is valid. This tells the compiler that the variable is a constant and *the application* can not change it but something else (i. e. the hardware) might change it.

Obviously, any large lookup tables should be located in the CODE area - a declaration might be:

```
 unsigned char code default_base_fuel_PW_map[] =
{
            0x08,0x08,
            0x00,0x00,0x00,0x09,0x41,0x80,0xC0,0xFF,
            0x00,0x00,0x13,0x1A,0x26,0x33,0x80,0xFF,
            0x00,0x00,0x00,0x09,0x41,0x80,0x66,0x66,
            0x00,0x00,0x00,0x05,0x4A,0x46,0x40,0x40,
            0x00,0x00,0x00,0x08,0x43,0x43,0x3D,0x3A,
            0x00,0x00,0x00,0x00,0x2D,0x4D,0x56,0x4D,
            0x00,0x00,0x00,0x00,0x21,0x56,0x6C,0x6F
  } ;
```

With large objects like the above it is obviously important to state a memory space. When working in the SMALL model in particular, it is very easy to fill up the on-chip DATA RAM with just a single table! RAM constants would be:

```
unsigned char scale_factor = 128    ;
unsigned int fuel_constant = 0xFD34 ;
```

These could, however, have their values modified during program execution. As such, they are more properly thought of as initialised variables - see section 3.2.2.

4.2 Variables

4.2.1 Uninitialised Variables

Naturally, all variables exist in RAM, the configuration of which is given in section 2.1.1. Note the MISRA-C guide (Rule 30) says that all variable shall have been asigned a value before use. It is always a good idea to set all variables to a known value. Ie set pointers to NULL and other variables to 0 or a set error condition. Thus if they are not set elsewhere before use it is would be possible to detect this. However as this takes time in startup it is some times not a good idea. Many embedded systems are required to come alive in a very short (and precise) time.

The setting in the uVision IDE will determine the overall memory model or the #pragma *memory_model* line in the file. In all examples in this guide the #pragma line will be used to highlight the global model in use. In this case, all variables are placed within the on-chip RAM. However, specific variables can be forced elsewhere as follows:

```
#pragma SMALL

unsigned char xdata engine_speed ;
signed char xdata big_variable_array[192] ;
```

This will have engine_speed placed in an external RAM chip. Note that no initial value is written to engine_speed, so the programmer must not read this before writing it with a start value! This xdata placement may be done to allow engine_speed to be traced “on the fly”, by an in-circuit emulator for example.

In the case of the array, it would not be sensible to place this in the on-chip RAM because it would soon get filled up with only 128 bytes available. In this case as the array is indirectly addressed the array could reside across DATA and IDATA. This is a very important point - never forget that the 8051 has very limited on-chip RAM.

Another example is:

```
#pragma LARGE
  .
  function(unsigned char data para1)
  {
       unsigned char data local_variable ;
       .
       .
  }
```

Here the passed parameters are forced into fast directly addressed internal locations to reduce the time and code overhead for calling the function, even though the memory model would normally force all data into XDATA.

In this case it would be better to declare the function as SMALL, even though the prevailing memory model is large. This is extremely useful for producing a few fast executing functions within a very big LARGE model program.

On a system using paged external RAM on Port 0, the appropriate directive is “pdata”. See notes in section 2.1.3 for details on how to best locate variables.

4.2.2 Initialised Variables

The MISRA-C guide (Rule 30) says that all variable shall have been asigned a value before use. To force certain variables to a start value in an overall system setup function, for example, it is useful to be able to declare and initialise variables in one operation. This is performed thus:

```
unsigned int engine_speed = 0 ;
function()
{
  .
}
```

Here the value “0” will be written to the variable before any function can access it. To achieve this, the compiler collects together all such initialised variables from around the system into a summary table. A runtime function named “C_INIT” is called by the “startup.obj” program which writes the table values into the appropriate RAM location, thus initialising them. startup.obj comes from startup.a51 and assembly file. This is automatically included by the linker unless a local startup.a51 is used to override it.

Immediately afterwards, the first C program “main()” is called. Therefore no read before write can occur, as C_INIT gets there first. The only point to note is that you must modify the “startup.a51” program to tell C_INIT the location and size of the RAM you are using. For the large model, XDATASTART and XDATALEN are the appropriate parameters to change.

```
;  User-defined Power-On Initialization of Memory
;  With the following EQU statements the initialization of memory
;  at processor reset can be defined:
; the absolute start-address of IDATA memory is always 0
IDATALEN       EQU    80H    ; the length of IDATA memory in bytes.
XDATASTART EQU 0H     ; the absolute start-address of XDATA memory
XDATALEN       EQU    0H     ; the length of XDATA memory in bytes.
PDATASTART EQU 0H     ; the absolute start-address of PDATA memory
PDATALEN       EQU    0H     ; the length of PDATA memory in bytes.
;  Notes:  The IDATA space overlaps physically the DATA and BIT areas of the
;          8051 CPU. At minimum the memory space occupied from the C51
;          run-time routines must be set to zero.
;------------------------------------------------------------------------------
;  Reentrant Stack Initilization
;  The following EQU statements define the stack pointer for reentrant
;  functions and initialized it:
;  Stack Space for reentrant functions in the SMALL model.
IBPSTACK       EQU    0      ; set to 1 if small reentrant is used.
IBPSTACKTOP    EQU    0FFH+1 ; set top of stack to highest location+1.
;  Stack Space for reentrant functions in the LARGE model.
XBPSTACK       EQU    0      ; set to 1 if large reentrant is used.
XBPSTACKTOP    EQU    0FFFFH+1; set top of stack to highest location+1.
;  Stack Space for reentrant functions in the COMPACT model.
PBPSTACK       EQU    0      ; set to 1 if compact reentrant is used.
PBPSTACKTOP    EQU    0FFFFH+1; set top of stack to highest location+1.
;------------------------------------------------------------------------------
;  Page Definition for Using the Compact Model with 64 KByte xdata RAM
;  The following EQU statements define the xdata page used for pdata
;  variables. The EQU PPAGE must conform with the PPAGE control used
;  in the linker invocation.
PPAGEENABLE    EQU    0      ; set to 1 if pdata object are used.
PPAGE          EQU    0      ; define PPAGE number.
```

4.3 Watchdogs With Large Amounts Of Initialised Data

In large programs the situation may arise that the initialisation takes longer to complete than the watchdog timeout period. The result is that the cpu will reset before reaching main() where presumably a watchdog refresh action would have been taken. To allow for this the INIT.A51 assembler file, located in the \C51p\LIB directory, should be modified.
A special empty macro named WATCHDOG is provided which should be altered to contain your normal watchdog refresh procedure. Subsequently, this is automatically inserted into each of the initialisation loops within the body of INIT.A51.

```
WATCHDOG       MACRO
               ;Include any Watchdog refresh code here
               P6 ^= watchdog_refresh  ;Special application code
               ENDM
;————————————————————————————————————
               NAME   ?C_INIT
?C_C51STARTUP SEGMENT CODE
?C_INITSEG     SEGMENT CODE  ; Segment with Initialising Data
               EXTRN CODE (MAIN)
               PUBLIC ?C_START
               RSEG   ?C_C51STARTUP INITEND: LJMP   MAIN
```

```
?C_START:
               MOV     DPTR,#?C_INITSEG
LOOP:
               WATCHDOG   ;<<— WATCHDOG REFRESH CODE ADDED HERE!
               CLR     A
               MOV     R6,#1
               MOVC    A,@A+DPTR
               JZ      INITEND
               INC     DPTR
               MOV     R7,A
               .
               . Large initialisation loop code
               .
               DJNZ    R7,XLoop
               DJNZ    R6,XLoop
               SJMP    Loop
               LJMP MAIN                ; C51 Program start

               RSEG    ?C_INITSEG
               DB      0
               END
```

4.4 C51 Variables

4.4.1 Variable Types

Variables within a processor are represented by either bits, bytes, words or long words, corresponding to 1, 8, 16 and 32 bits per variable. C51 variables are similarly based, for example:

```
bit            = 1-bit        0 - 1
signed char    = 8-bits       0 - +/- 127
unsigned char  = 8-bits       0 - 255
signed int     = 16-bits      0 - +/-32768
unsigned int   = 16-bits      0 - 65535
signed long    = 32-bits      0 - +/- 2.147483648x109
unsigned long  = 32-bits      0 - 4.29496795x109
float          = 32-bits      +/-1.176E-38   to +/-3.4E+38
pointer =24/16/8 bits  Variable address
```

There is also a type called short. However as, *in this implimentation*, short is the same size as an int it is not required nor used. There is a myth that short is always 16 bits and more portable than int. This is not correct and the size of short is not precisly defined. Short should not be seen in an 8051 program.

Misra-C rule 13 requires the use of typedefs for standard types. Further to this ISO C99 recommends a similar thing and specifies a header file called inttypes.h. This contains the following types int8_t, uint8_t, int16_t, uint16_t, int32_t and uint32_t that would be defined in C51 as:-

```
        typedef signed char      int8_t
        typedef unsigned char    uint8_t
        typedef signed  int      int16_t
        typedef unsigned int    uint16_t
        typedef signed  long    int32_t
        typedef  unsigned long  uint32_t
```

There are, in ISO C99, also two pointer types which could be used for generic pointer and signed and unsigned 64 bit types which are not appropreate to the 8051. For most embedded work, especially on the 8051 whether the variable is a character or integer is imaterial. What is often far more important is the size of the data.

Typical declarations would be:

```
        unsigned char xdata  battery_volts ;
        int idata correction_factor       ;
        bit flag_1 ;
```

```
Or using the ISO C   typedef's
```

```
        uint8_t  xdata  battery_volts ;
        int16_t idata correction_factor ;
        bit flag_1 ;
```

Note: There is no ISO C typedef for bits. Also bit variables are always placed in the bit-addressable memory area of the 8051 - see section 2.1.1 for this reason bits can not have a memory location modifier.

By definition they must be in BDATA.

With a processor such as the 8086, int is probably the commonest data type. As this is a 16 bit processor, the handling of 16 bit numbers is generally the most efficient. The distinction between int and unsigned int has no particular impact on the amount of code generated by the compiler, since it will simply use signed opcodes rather than the unsigned variety.

For the 8051, naturally enough, the char should be the most used type. Again, the programmer has to be aware of the thoroughly 8 bit nature of the chip. Extensive use of 16 bit variables will produce slower code, as the compiler has to use library routines to achieve apparently innocuous 16 by 8 divides, for example.

The use of signed numbers has to be regulated, as the 8051 does not have any signed arithmetic instructions. Again, library routines have to do the donkey work.

An interesting development has been the Siemens 80C537, 717, 509 abd the Dallas 390 which have an extended arithmetic instruction set. For instance, 32 by 16 divide and integer instructions. Indeed, this device might be a good upgrade path for those 8051 users who need more number crunching power and who might be considering the 80C196 or C16* family. A suite of runtime libraries is included with the Keil C51to allow the compiler to take advantage of the enhancements.

This uses the MOD517 directive to the compiler. In uVision2 this is an automatic selection whent he target choice is made. For older versions and command line use #pragma MOD517

4.4.2 Special Function Bits

A major frustration for assembler programmers coming to C is the inability of ISO C to handle bits in the bit-addressable BDATA area directly. Commonly bit masks are needed when testing for specific bits with chars and ints. From C51 version 3 however, it is possible to force data into the bit-addressable area (starting at 0x20) where the 8051's bit instructions can be used directly from C. An example is testing the sign of a char by checking for bit = 1. Here, the char is declared as "bdata" thus:

```
 char bdata test ; /* put char into bit addressable data */
sbit sign =  test^ 7 ;  /* the sign bit, 7,  is defined as  "sign" variable */
```

To use this:

```
char bdata test ;
sbit sign =  test^ 7 ;
void main(void)
{
        test = -1 ;
        if(test & 0x80) /* Conventional bit mask and & */
        {
                test = 1 ;     /* test was -ve */
                }
        if(sign == 1) /* Use sbit */
        {
                test = 1 ;     /* test was -ve */
                }
}
```

Results in the assembler:

```
RSEG  ?BA?T2
test:                 DS  1
sign    EQU    test.7
MOV     test,#0FFH
;     if(test & 0x80) { /* Conventional bit mask and & */
        MOV    A,test
        JNB    ACC.7,?C0001
;        test = 1 ;     /* test was -ve */
        MOV    test,#01H
;    }
?C0001:
;     if(sign == 1) {   /* Use sbit */
        JNB    sign,?C0003
        ;        test = 1 ;     /* test was -ve */
        MOV    test,#01H
;    }
?C0003:
        RET
```

Here, using the sbit, the check of the sign bit is a single JNB instruction, which is a lot faster than using bit masks and &'s in the first case!

The situation with ints is somewhat more complicated. The problem is that the 8051 does not store things as you first expect. The same sign test for an int would still require bit 7 to be tested. This is because the 8051 stores int's high byte at the lower address. Thus bit 7 is the highest bit of the higher byte and 15 is the highest bit of the lower.

	High byte	Low Byte
Bit locations in an integer	7,6,5,4,3,2,1,0,	15,14,13,12,11,10,9,8

Another useful place bit types can be used is in the SFR's that end on 0 or 8. For example the IO ports. Port 1 may be defined in a header file as:

```
        sfr P1   = 0x90;
```

this can then be bit used as follows

```
        #define MOTOR_ON   1;
        #define MOTOR_OFF  0;
        sbit Motor_Control  = P1^1;
        sbit Motor_State    = P1^2;

        Motor = MOTOR_ON;
        .
        .
        Motor = MOTOR_OFF;

        if (MOTOR_OFF == Motor_State)
        {
                .
                .
        }
```

4.4.3 Converting Between Types

One of the easiest mistakes to make in C is to neglect the implications of type within calculations or comparisons.

Taking a simple example:

```
        unsigned char x = 10;
        unsigned char y  = 5;
        unsigned char z ;
        z = x * y ;

Results in z = 50
```

However:

```
        unsigned char x = 10;
        unsigned char y  = 50;
        unsigned char z ;
        z = x * y ;
```

results in z = 244. The true answer of 500 (0x1F4) has been lost as z is unable to accommodate it. The solution is, of course, to make z an unsigned int. However, it is always a good idea to explicitly cast the two unsigned char operands up to int thus:

```
        unsigned char x ;
        unsigned char y ;
        unsigned int z ;

        z = (unsigned int) x * (unsigned int) y ;
```

C51, since Version 3, will automatically promote chars to int in it's natural state. This is called "Interget Promotion Rule" and is a requirement of the compiler for ISO comformance. It could be argued that on any small microcontroller you should always be aware of exactly what size data is at al times.

It is possible to disable this feature in most versions of the Keil C51 compiler. Why would you wanrt to do that? Well, assuming that you do *know* the maximum size of your data, and that there are suitable check for our of range data, it is faster and the code is smaller. In previous sections it was highlighted that the 8051 is an *8 bit* MCU. It can handle 8 bit chars far faster than 16 bit ints. So where you know the sizes of your data, and be very sure you do, switch off the Integer Promotion for faster smaller code. There will be no warnings unless you are using Lint.

4.4.4 A Non-ISO Approach To Checking Data Type Overflow

A very common situation is where two bytes are to be added together and the result limited to 255, i.e. the maximum byte value. With the 8051 being byte-orientated, incurring integers must be avoided if maximum speed is to be achieved. Likewise, if the sum of two numbers exceeds the type maximum the use of integers is needed.

In this example the first comparison uses a proper ISO approach. Here, the two numbers are added byte-wise and any resulting carry used to form the least significant bit of the upper byte of the notional integer result. A normal integer compare then follows. Whilst C51 makes a good job of this, a much faster route is possible, as shown in the second case.

```
; #include <reg51.h>
; unsigned char x, y, z ;
; /*** Add two bytes together and check if ***/
; /***the result has exceeded 255 ***/
;
; void main(void) {
        RSEG  ?PR?main?T
        USING   0
main:
;    if(((unsigned int)x + (unsigned int)y) > 0xff) {
        MOV     A,x
        ADD     A,y
        MOV     R7,A
        CLR     A
        RLC     A
        MOV     R6,A
        SETB    C
        MOV     A,R7
        SUBB    A,#0FFH
        MOV     A,R6
        SUBB    A,#00H
        JC      ?C0001
;        z = 0xff ;   /* ISO C version */
        MOV     z,#0FFH
;        }
```

In this case the carry flag, “CY”, is checked directly, removing the need to perform any integer operations, as any addition resulting in a value over 255 sets the carry. Of course, this is no longer ISO C as a reference to the 8051 carry flag has been made.

```
?C0001:
;    z = x + y ;
        MOV     A,x
        ADD     A,y
        MOV     z,A
;
;    if(CY) {
        JNB     CY,?C0003
;        z = 0xff ;   /* C51 Version using the carry flag  */
        MOV     z,#0FFH
;        }
```

```
?C0003:
        RET
```

The situation of an integer compare for greater than 65535 (0xffff) is even worse as long maths must be used. This is almost a disaster for code speed as the 8051 has very poor 32 bit performance, (excepting the C537, 517, 509, 390). The trick of checking the carry flag is still valid as the final addition naturally involves the two upper bytes of the two integers.

In any high performance 8051 system this loss of portability is acceptable, as it allows run time targets to be met. This again illistrates that good 8051 C may not be portable or pure ISO C. The alternative, using pure ISO C, might mean that a different, more powerful processor has to be used. Thus bringing up unit costs. This does not mean that solid good practice should be ignored. It is just that there are some non-standard architecture specific non-ISO-C extensions that can be used.

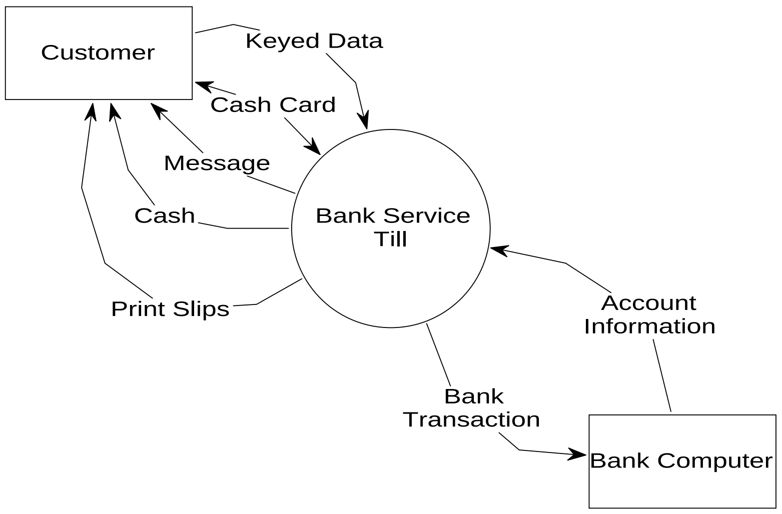
5 Program Structure And Layout

5.1 Modular Programming In C51

The first three editions of this work started this chapter with: "This is possibly not the place to make the case for modular programming, but a brief justification might be appropriate." However the interviening years have indicated that this *is* the place where the case for modular programming *must* be made!

In anything but the most trivial programs the overall job of the software is composed of smaller tasks, all of which must be identified before coding can begin. As an electronic system is composed of several modules, each with a unique function, so a software system is built from a number of discrete tasks. In the electronic case, each module is designed and perfected individually and then finally assembled into a complete working machine.

With software there are many similar modeling techniques used for designing the structure and modules of the code. A system, any system (properly applied) is better than no system. At one time graphical modeling was the answer and this spawned CASE, Computer Aided Software Engineering abd CAD, Computer Aided Design tools. They were expensive, promised the earth..... and with CASE largely failed to deliver. Things have improved greatly. Flow charts were in, then out and back in and some say never went away.

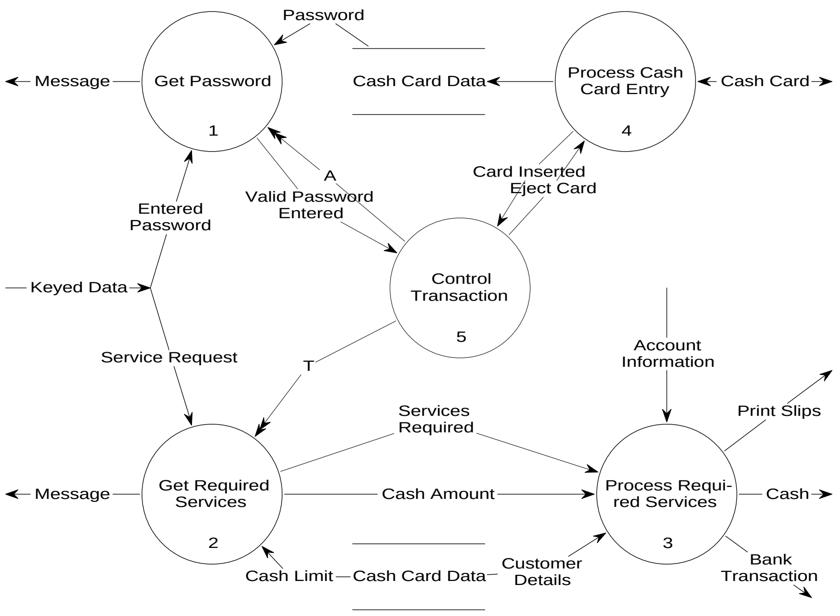


Firstly remember that any tool is only as good as the person using it. You can make a complete mess with or without the design tools tool (some would say that the automated tools help you make a mess more eficiently). For most 8051 designs the software structures can usually be drawn using paper and pencil. I have only used a tool in this case for clarity of the drawings.

For the 8051, using mainly C or assembler, the structured or modular design methods should be used. Structured or modular designed software is the C equivelent of the Object Orientated Programming used with C++, Ada and the like. Both Structured and OO programming share many concepts and good modular programming has many of the attributes of OOP but due to the differences in the way the languages work can not have all of them.

For the 8051 a structured method like Yourdon (one of the more popular) would be used. The Yourden method has several views of the software and it is worth looking at these as an example as to why modular programing should be used on all systems.

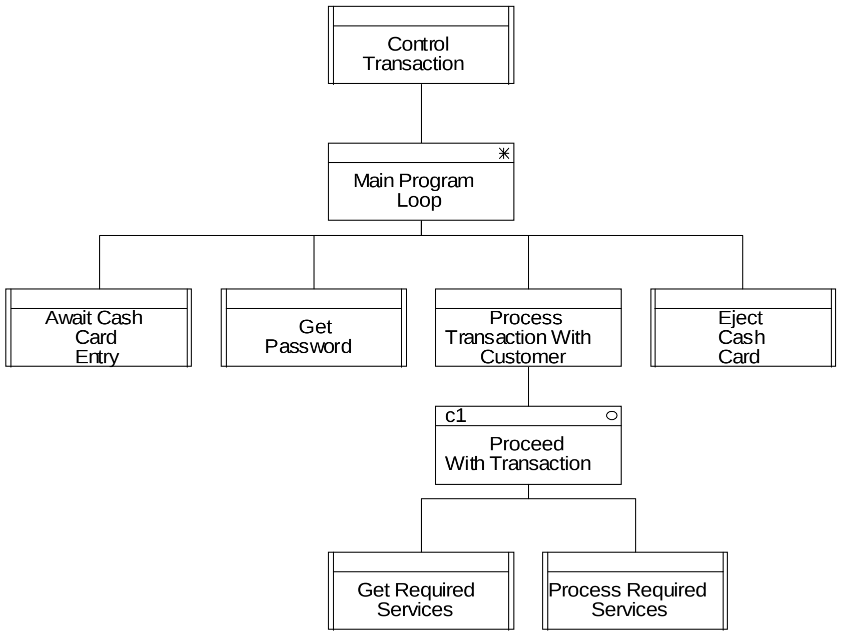
The top diagram shown here is a top level Context diagram. This shows all the inputs and outputs to a system. In this case an ATM. This defines the outer limits of the system. It could equaly be an Automitive engine control or a washing machine. In the case of my car there are mor similarites than I would care to acknowledge!



The next level down is what happens inside the main circle. As can be seen this has broken the system down in to descrete operations linked by data and control flows. This is top down modeling. Now the functionality and indeed functions start to appear.

The view changes slightly for the next diagram.

However, this clearly shows the structure of the software, as opposed to the system. It is not clear at this point if each box is a single function or a collection of functions, in effect a module.



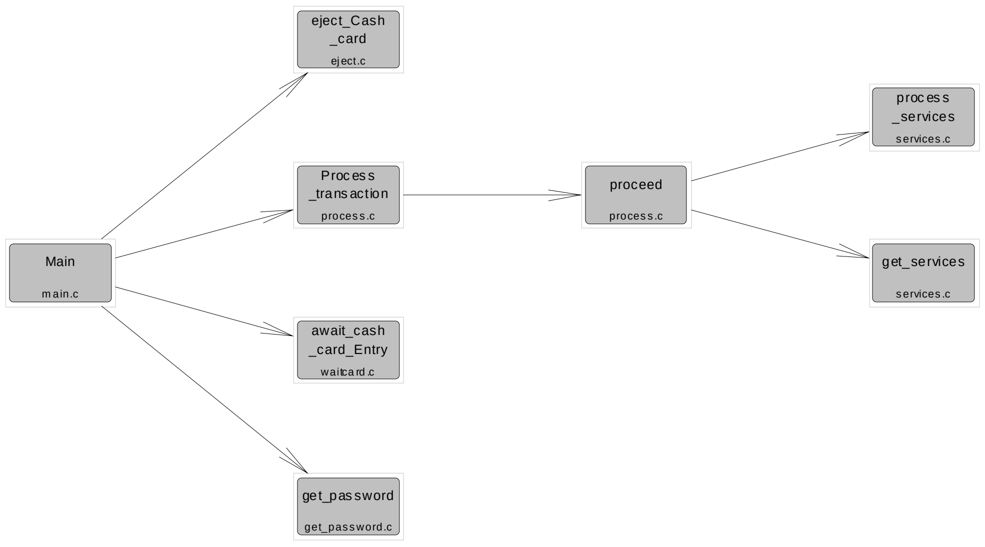
In some methods the actual data or control information passed between these functions would be shown. These functional boxes can be either single functions (is source code) or broken down further into similar charts containing several boxes (ie C functions). Some systems may have several layers of child charts such as the one shown.

The interesting thing is that there are no actual implimentation details shown. "Get Pasword" or "Await Cash Card Entry" are generic. This is where modular software containing more than one file starts to become the obvious answer.

In this diagram it is just a "Get password" box. However if all the source associated with this diagram were all in one file any change would result in a re-compilation of the whole file. By making the "Get Password" or "Await Cash Card Entry" a seperate file we get modularity and if there are any changes to the "Get Password" module only that file need to be re-compiled. The rest only needs to be re-linked. It also gives the added advantage that should a different password system be used only the one file requires modifications.

In this case there would probably be a selection of "Get Password" modules depending on the customer the algorithm or the hardware. Thus the main code loop can be reused. Note C had reuse long before the C++ and OOP crowd "invented" it! The diagrams so far were generated using the Select Yourden tool.

The next diagram is a little different to the one above. This was generated by the DA-C that reverse engineered the C source code. This example has gone a little to the extreme to demonstrate the point. As with all things common sense should be used.



This is the basic layout of the empty C functions. As can be seen there is one C function per file. The exceptions are the Process and Services files. This is because the Process file has the option to either proceed or not and the Services file would contine both the list of services availavle as well as the interface to them. Thus if the services were to change only the one module would need changing.

Apart from making the program more manageable in that file listings are shorter and functionality is contained there are other benefits. Data security. Many variables can be made local to the one file. With data overlaying enabled (see Linker manual) this saves much space on an 8051. Simply put the linker can "overlay" local parameters in the same memory space. As a General Rule only make a variable Global if you really have to. Global variables take up permanent residence in the (very limited) memory.

There are also similar arguments for functions. By making functions that are only used in one file static they will only be visible in the one file.

```
        static unsigned char proceed( unsigned char code);
```

This is rather like the encapsulation the OO programmers like.   It does help the C programmer because only the functions declared as extern will be available outside the file. This gives a defined interface to a file and permits internal changes to a file without having to make changes outside the file. For this to work well more thought must be given to the functions used as interfaces so that the possibility of changes to them is kept to a minimum. The use of static functions also makes smaller faster programs as the compiler knows that the function will only be used in the one file and smaller faster jumps can be used.

5.2 Accessibility Of Variables In Modular Programs

A typical C51 application will consist of several functional blocks each contained in their own source files. Each block will contain a number of functions which operate on and use variables in RAM. Individual functions will (ideally) receive their input data via parameter passing and will return the results similarly. Within a function temporary variables will be used to store intermediate calculation values.

At one time, as it used to be done years ago in assembler, all variables (even the temporary ones) would be defined in one place and will remain accessible to every routine. It also meant that very variable held a permanent place in memory. When memory was tight variables got reused by other functions for other purposes! Very dangerous as the programmer had to manually work out when a vartiable space was not going to be used. The closest you can get to that in C is a Union or explicit addressing of memory.

With the use of C data memory can be used far more effectively and safely. Data can be made local to a function, file or global. Data can be passed so that it is local to several linked functions. Further to this data can be dynamic or static. Ram space (not the variable name) used for local data that only has a life during a particular function can be reused by other temporary data. This is known as data overlaying and it can save a considerable amount of space. It is, however very carefully done by the compiler, not the programmer.

There is a view that there should be no global variables at all. Variables should always be passed as parameters in function calls. This is not possible in embedded systems because many variables will be registers for IO SFR's etc. or other peripherals and buffers. It is a good idea to keep global variable to a minimum. this is essential in applications that have more than one file as data control becomes impossible (and memory wasted).

With the Keil C51 compiler if the functions can be kept to three or fewer parameters the compiler will work faster and registers used rather than the compiled stack space (in data memory) . The rule is three chars, ints or pointers. However only two longs or floats may be passed. Though one char may be passed with the longs and floats. Incidentally the return parameter is always passed in a register. NOTE This does not apply if the first parameter is a bit. Therefore bits should be passed as the third parameter.

Thus it is a balancing act. It is faster, and consumes less memory if only a limited number of variables are passed as parameters.   However, globals always take space but may save space as they only appear once rather than in several guises if the data has to be passed through several functions there are more than three functions. This is becasuse if there will be several spaces allocated in the compiled stack for each function call.   I. E. if there are

three functions there will be three places in memory where space is provided for that parameter.
Globals are always available and visable. The use of local variables can aid encapsulation and data hiding (which is better for modula programming). this is because there is an easliy defineds interface between source files.

Pointers, of course, are the grey area as they can be used where only two or three bytes of a pointer need to be passed in order to give access to a large array of many bytes. The correct use of the pointer variable declaration can "hide" access to an array that is permanently in memory. Only the pointer to an array should be passed not the array itself.

Care should be taken with variables to conserve storage space. The code below is an example of scope (not how to write good code!). The use of blocks in the function Calculate_key below is quite useful, if an under used part of C, where a large number of temporary variables are required. A block may be defined anywhere in c simply by using a pair of {}.

NOTE that by default functions are "Extern" however it is good practice to use the extern (or static).

```
 /****** Start of code ********/

extern int system_flags;                    /*defined in another  module */
extern unsigned char get_password(unsigned char status);  /* function visable in other
modules */

/* three functions only visable in THIS file */
static long calculate_key(void);
static long gen_randon_prime();

long int password_flags;  /* variable visable in other files */

unsigned char get_password(unsigned char status)
{
        /* varuiables only visable in this fuuction */
        unsigned char pass_status;
        long int key_status;

        key_status = calculate_key();

        if (status > 0)
        {
                pass_status = key_status;
        }
        else
        {
                pass_status = 0;
        }

        return pass_status;
}


static long int calculate_key(void)   /* finction only visable in this file */
{
        int temp_1;       /* variables visable in this functiion only */

        static long int temp_key = 0;     /* static variable is only initialsed ONCE */
                                    /* at program start up. The variable is only visable*/
                                    /* in this function but is not destroyed at the end */
                                    /* of the function call and retauns its value for the */
                                    /* next time the function is called. */


        {
                /* four variables only visable in this BLOCK WITHIN the function. */
                /* note this can be done in a for, while, do or if block as wel as */
                /* a block simpley decalred betewwn two {} */
                long int prime_1;
                long int prime_2;
                long int key_a;
                long int key_b;

                prime_1 = gen_randon_prime();
                prime_2 = gen_random_prime();

                key_a = ((prime_1 -1)  * (prime_2 -1));

                key_b =         prime_1%temp_1 ;

                temp_key =      key_a + key_b;

        }

        /* statements */


        return temp_key;

}
/******* end of Code ******/
```

5.3 Building a C51 Modular Program

The text thus far has shown the architecture and the idiosyncrasies of the 8051 and the various ways of addressing the 8051 memory spaces. We have covered the various ways of allocating data memory in C. We have also covered the reasons for modular programming. Next we will cover the practicalities of writing a modular program.

5.3.1 The Problem

As explained in the ATM example previous section there are a lot of advantages in using multiple modules. Not least the fact that changing one line of code only requires re-compilation of that file (but not all the others) and just re-linking. The problem is how to ensure that the all the data required is passed cleanly and in a well defined manner between the files. Just as important is the need to ensure that no other data "leaks" between files. Some coding standards require that no variable name be used in more than one place in the entire application to ensure that there is no possibility of data leakage. Note this is not the same as memory leakage caused by dynamic allocation of memory.

5.3.2 Maintainable Inter-Module Links

The example ATM program is constructed in a modular fashion and we will use it for the example. It is modular in that the source is in separate modules. However, even with this small program a maintenance problem is starting to become apparent: The source of the trouble is that to add a new data item or function, at least two modules need to be edited. Not only, the module containing the data declaration but any other module which refers to the changed items. With long and meaningful names common in C and complex memory space qualification widespread in C51, much time will be wasted in getting external references to match at the linking stage. Simple typographic errors can waste huge amounts of time!

Function prototypes are one of the more helpful aspects that came in with the ISO-C89. This requires that any function:

```
unsigned char get_password(unsigned char status)
{


       return pass_status;
}
```

required a function prototype

Unlike the older “K*R” style where the parameters were placed after the function name but before the first opening brace the new style required that the parameters were placed between the brackets in the function name as they would be in a function call. Thus the prototype for the function above would be:

```
unsigned char get_password(unsigned char status);
```

Whilst the C-89 did require the parameter types it did not *require* that the parameters be names as well. So the prototype for the call above could be:

```
unsigned char get_password(unsigned char);
```

However his way lies madness.....! The parameter name should always be included and to be absolutely correct the new ISO-C99 typedefs (as per MISRA-C Rule 13) should be used as well giving a prototype of :

```
UINT8_T get_password(UNIT8_T status);
```

This prototype system could make matters worse if incorrectly used! I have seen the problem of inter-module maintenance on a grand scale. A multi-module program had some functions changed. All the programmer did was “copy” the new function prototype and recompile. Then in any module that did not compile, he pasted the new prototype. Next he did went to each error and pasted the new function call.... The problem was, in *most* cases, he forgot to remove the original (now incorrect) prototype. The program was littered with redundant prototypes and in some cases unused data. The other problem was that as the old function prototype was in some of the modules they did not complain where there were calls to the old function that had been missed. This only appeared at link time. We will shortly show how this can be avoided and maintenance greatly improved.

In large programs with many functions and global variables, the global area preceding the executable code can get very untidy and cumbersome. Though Global variables should be kept to a minimum. There is an argument that says it is good to have to have external references at the top of a module when first using a new piece of global data. This is because it means that you are always aware of exactly which items are used. This is fine in theory but rarely works in practice and ends up with a maintenance nightmare.

It is preferable to have a single include or header file incorporated as a matter of course in each source file, containing an external reference for every global item, regardless of whether the host file actually needs them all. However, this is not the best solution. It is an improvement that there is only one version of the definition.

A solution to this is to have “module-specific” include files. Basically, for each source module “.c” file, a second “.h” include is created. This auxiliary file contains both original declarations and function prototypes plus the external references. It is therefore similar in concept to the standard library files used in every C compiler. Stdio.h for example. NOTE: Many libraries were written a long time ago and may conform to “K&R” C not the ISO C89 or current C99.

For example the code in the top of the password.c file is:-

```
extern int system_flags;             /*defined in another  module */
extern unsigned char get_password(unsigned char status);  /* function visable in other
modules */

/* three functions only visable in THIS file */
static long calculate_key(void);
static long gen_randon_prime();

long int password_flags;  /* variable visable in other files */
```

the get password would go into the password.h file but the static (local) functions would not. This is for two reasons. Firstly they are only used in the password.c file and secondly that it will create a compile error when password.h is used in any other c file.

So we are looking at the following for our password.h header file.

```
extern unsigned char get_password(unsigned char status);
```

We also have the line:

```
long int password_flags;  /* variable visible in other files */
```

This gives us another problem that is connected with the line:

```
extern int system_flags;             /*defined in another  module */
```

These variables need to be "extern" in all files bar the one in which they are defined. The trick is, however, to use conditional compilation. This will prevent the original declarations and the external versions being seen simultaneously.

When included in their home modules, i.e. when password.h is included in password.c, the compiler only sees the original declarations. Whereas, when included in a anyother module, only the external form is seen. To do this each source module must somehow identify itself to the include file. This is very simply done with #define statements.

A #define is placed at the top of each module giving the name of the module.

Thus in the top of password.c the first line should be something like

```
#define  _PASSWORD_C

#include <inttype.h>          /*  <> denotes system or library header*/
#include “password.h”         /*  “”  denotes local or application header*/
```

Thus password.h will contain the following:-

```
#ifdnef _PASSWORD_H
        #define _PASSWORD_H

        #ifdef _PASSWORD_C
                UNIT8_T get_password(UNIT8_T status);
                INT32_T password_flags;
        #else
                extern UNIT8_T get_password(UNIT8_T status);
                extern INT32_T password_flags;
        #endif
#endif
```

When included in its "home" module, Password.c the #ifdef #else #endif will cause the preprocessor to see the "local" declarations. When placed in any other modules ie that do not have _PASSWORD_C defined, the preprocessor will see the external equivalents.

There are a couple of points with regard to the extern on the function and the check for the "home" file. Firstly that by *implication* all function prototypes are extern. As an embedded Engineer the word *implied* should not be in your vocabulary. Things should be explicit. This was discussed, extensively, in both the C and embedded circles in late 2000 and the view was it is better to be explicit than implied in this case.

The other point Keil supports __FILE__ and some may be tempted to use it in the header but it is not of practical to use in this context, as its “value” cannot be used for a #define name. Besides, even if it was possible it would not be Good Practice.

We have slipped another guard in here as well that requires explanation.

```
#ifdnef _PASSWORD_H
        #define _PASSWORD_H
        .
        .
#endif
```

This has the effect of including a header file only once. This is important. It does, marginally, cut down compile time but is not why it is done. It stops circular redefinition’s. I might put a define in my header file:

```
#define MAX = 128
```

but later in another header file

```
#define MAX = 256
```

this will cause problems. Worst still if I re-include my original header file. I will get the sequence

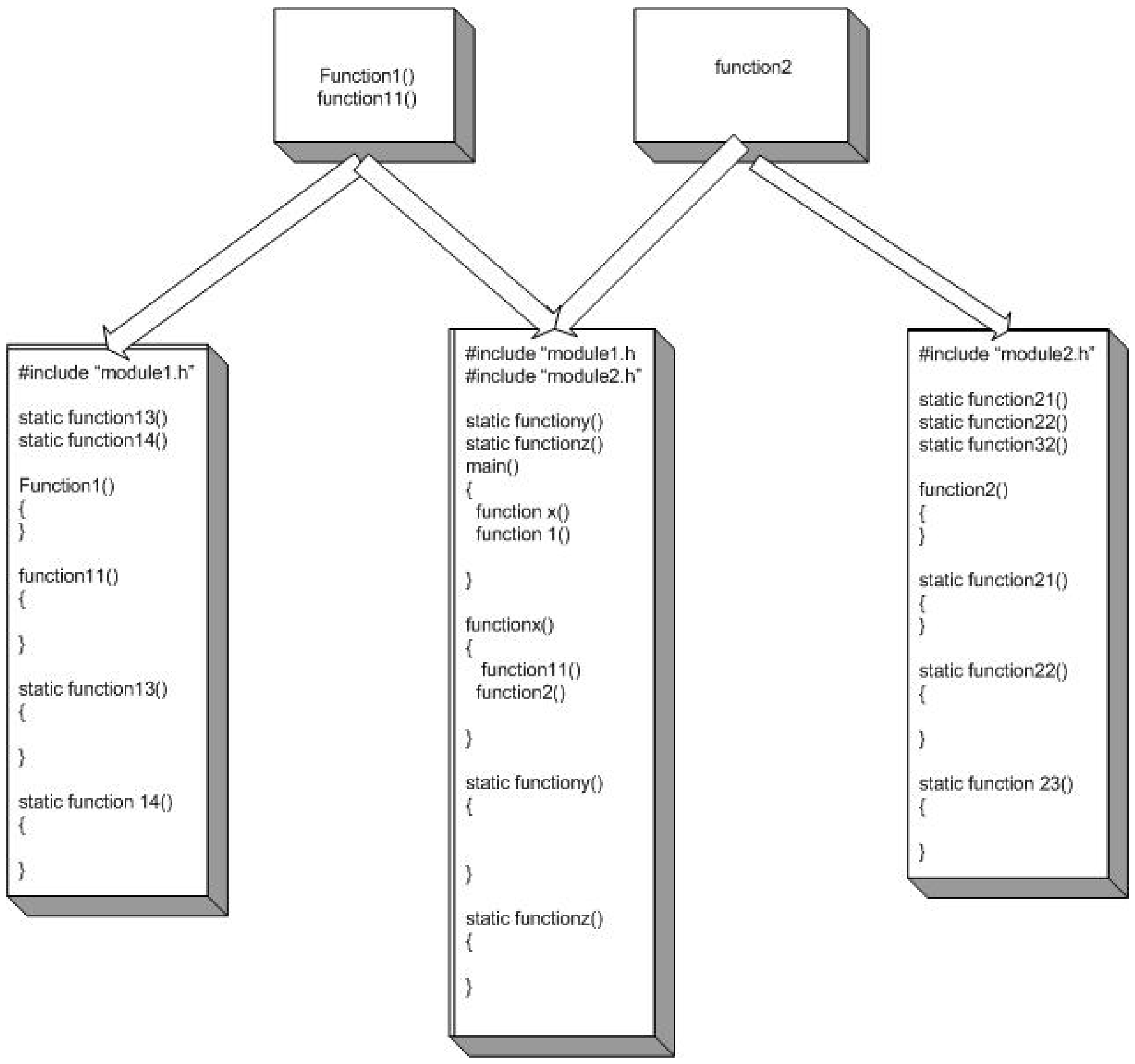
```
#define MAX = 128
#define MAX = 256
#define MAX = 128
```

The problem is that for a period MAX was 256 and may have had an influence of one, many or no other data or defines before MAX was reset to 128. A very difficult bug to find. It can also mask other serious problems. On one project when some multiply included header files were unwrapped the program would not compile. The multiple definitions were masking a very deep seated bug.

By only including module-specific header files in those modules that actually need to access an item in another module data is only visible to other parts of the program that need it. It also means that maintenance is easier. If the C file is changed the header is changed to match. The new changes are visible identically to all the modules that need it with no more editing. Where changes are required to the C file the header file can be relied on to be correct (or the home C file would not compile).

This is where good design comes in to make sure that only minimal changes are required to function prototype parameter lists. In most embedded C dialects this can be a major help in program development. For example, a change in a widely-used function’s memory model attribute, from small to large, can easily be propagated through an entire program; the change in the intelligent header file belonging to the function’s home module!

Here’s how it’s done in practice:



```
#include "module1.h"

static function13()
static function14()

Function1()
{
}

function11()
{

}

static function13()
{

}

static function 14()
{

}
```

```
#include "module1.h
#include "module2.h"

static functiony()
static functionz()
main()
{
  function x()
  function 1()

}

functionx()
{
   function11()
  function2()

}

static functiony()
{


}

static functionz()
{

}
```

```
#include "module2.h"

static function21()
static function22()
static function32()

function2()
{
}

static function21()
{
}

static function22()
{

}

static function 23()
{

}
```

5.4 Standard Templates (and Version Control)

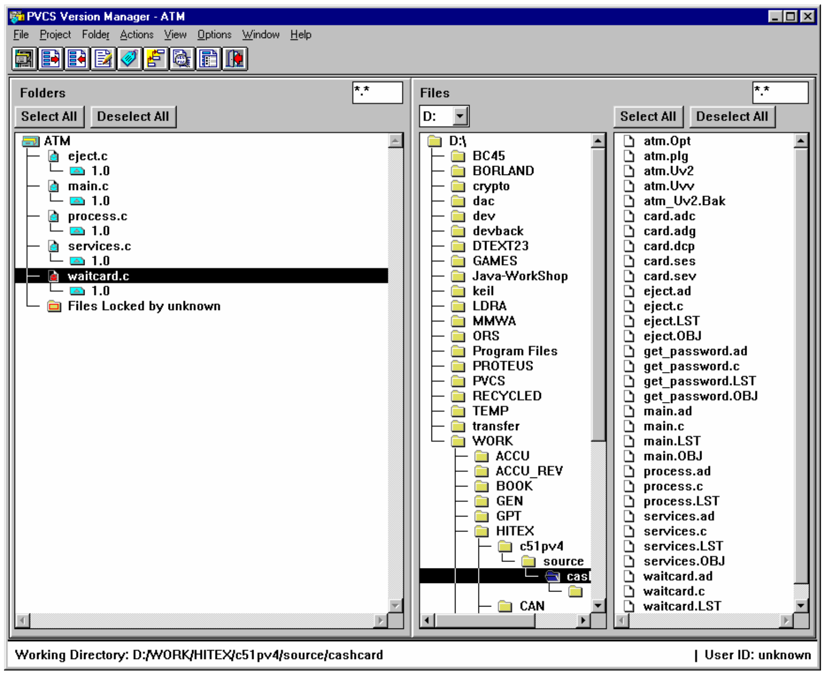
When constructing an application the source should always be thought through and the modules planned particularly the interfaces between modules. In order to keep track of the modules standard templates should be used. In fact the templates should be used for ***any*** code written event the "temporary" code. I once worked on a large project doing some maintenance. There were thirty of us do the maintenance! I noticed that one of the executables was called "FTB02". I could not work out why so I asked the team leader who asked the project leader who though back in to the mists of time (about 5 years) and explained: The code we were working on was the origional demo software constructed as part of the feasability study.... the FTB02 stood for Flying Test Bed 02!

You should always work assuming that the code will go on for ever. This may seem onourous but if standard templates are used it takes only a seconed to do. The appendix contains some suitable templates. Using the templates the majority of the work is done for you. However even more can be done automatically.

The templates in the appendaix have VCS keywords embedded in them. These keywords wil be expanded wil be expanded by the VCs software to automatically put in things like Author, file name, revision number, history etc. The format of the templates but generally all the vcs use similar key words so they should be easliy adaptable.

5.4.1 Version Control

The comments in the last section do, of course, assume that you are using a VCS... You are aren't you? Good! For those of you who do not: a VCS system is basically a application specific database. Most have an interface that looks like the MS Windows File Explorer Files are copied in to them. The file can be "checked out" and worked on and then "checked in" Most word processors will when saving a file move the origional to a *.bak or back up file. A VCS system will continue to do this such that all the old back ups are numbered and kept. This means that any version of the software can be retieved.


PVCS Version Manager - ATM
File Project Folder Actions View Options Window Help
Folders
Select All
Deselect All
Files
D:
Select All
Deselect All

```
ATM
├ eject.c
│ └ 1.0
├ main.c
│ └ 1.0
├ process.c
│ └ 1.0
├ services.c
│ └ 1.0
├ waitcard.c
│ └ 1.0
└ Files Locked by unknown
```

```
D:\
├ BC45
├ BORLAND
├ crypto
├ dac
├ dev
├ devback
├ DTEXT23
├ GAMES
├ Java-WorkShop
├ keil
├ LDRA
├ MMWA
├ ORS
├ Program Files
├ PROTEUS
├ PVCS
├ RECYCLED
├ TEMP
├ transfer
└ WORK
  ├ ACCU
  ├ ACCU_REV
  ├ BOOK
  ├ GEN
  ├ GPT
  ├ HITEX
  │ └ c51pv4
  │   └ source
  │     └ casl
  │       └
  └ CAN
```

```
atm.Opt
atm.plg
atm.Uv2
atm.Uvv
atm_Uv2.Bak
card.adc
card.adg
card.dcp
card.ses
card.sev
eject.ad
eject.c
eject.LST
eject.OBJ
get_password.ad
get_password.c
get_password.LST
get_password.OBJ
main.ad
main.c
main.LST
main.OBJ
process.ad
process.c
process.LST
services.ad
services.c
services.LST
services.OBJ
waitcard.ad
waitcard.c
waitcard.LST
```

```
Working Directory: D:/WORK/HITEX/c51pv4/source/cashcard          | User ID: unknown
```

VCS systems can do a great deal more than just store the versions. As mentioned they will update keywords in file information blocks. They can also assign lables to versions thus you can "check out" whole sets of files as "V1" or "Release 2" with a single command. Some VCS let you set up make and compilers so that you can issue a command such as "Make Release1" or "make v1.34" and you can isnstantly, well as fast as the compiler can run, produce V1.34. In other words you can almost instantly reproduce any version of the software you have created since you started to use the system. They can also "branch" that is hold several parallel version of a file. This Can be a god-send if you have to do maintain several different versions of an application at the same time. Most VC Scan also intergrate in to compiler IDEs, case tools , error reporting tools etc and can be used for most types of files not just source files.

VCS cost from free to the "standard" packages at about a third of the cost of a Keil DK51 and on to the rolls Royce packages at about the cost of a Series 3 BMW. Most, and certainly all 8051 embedded applications, will only require the "standard" packages. There are several free unix RCS on the internet. Whilst not essential a VCS is very useful and if you have many versions of source or many applications with lots of modules. VCS can be well worth the money. The Keil C51 will intergrate any VCS that has a command line interface. Keil currently supplies interfaces for MKS, PVCS and Source Safe systems.

Incidetally all VCs assume that you will be writing modular software.... In the ATM example shown I would use a VCS so I could maintain several different password modules, services modules etc. apart from using it to hold the other source modules. Thus I could almost instantly make a version od the software with any combination of password system and services. More to the point if I have to fix a bug in the password system or enhance it I onluy have to remake the one module and move the build labe on it to the newer version. I could then either compiler the one file and use the previous object files or rebuild the whiole project using the origional source.

Summary

If the Source templates are used with modular software and provided the necessary module name defines and globals are placed in the file or header as required the overall amount of editing required over a major project is usefully reduced. With the use of a VCS there is also a ful audit trail and instant recovery to older versions. Compilation and, more particularly, linking errors are reduced as there is effectively only one external reference for each global item in the entire program. For structures and unions the template only appears once, again reducing the potential for compilation and linking problems.

5.5 Task Scheduling

5.5.1 Applications Overview

When most people first started to learn to program, it was often using interpreted BASIC on a home computer or a PC. The programs are not usually too complicated; they start when you type “RUN” and finish at END or STOP. In between the start and stop, the computer appears to be totally devoted to executing your simple program. When it is finished you are simply thrown back to the BASIC editor or “operating environment”.

Most modern computers a use a multi-process operate system such as Unix, Linux or Microsoft Windows. This can appear to run many programs at once, your program is just one of many apparently running at the same time. In an 8051 system often there is no operating system to run the program. The other thing to realize, is that when you say “print file” on a PC (with an operating system e.g. MS Windows) is that the data is sent to another program called a [printer] device driver. In an 8051 program, you would have to right your own printer driver. Now you have two programs that need to run... apparently at the same time.

Who do you achieve this miracle? There are many, many ways of dong this but operating systems theory is well outside the scope of this publication! The simple answer is to use an operating system such as the Keil RTX51 or the CMX Rtos. The other more pragmatic approach for most programmers with simple systems is either to use interrupts or to write a simple scheduler. Interrupts could be used for the very short things such as servicing IO but are not really practical for running an application.

5.5.2 Simple 8051 multi-task Systems

The simplest approach is to put each "program" in to a separate functional block. Then call each major sub-function in a simple sequential fashion so that after a given time each function has been executed the same number of times. This is called a "Round-Robin". This constitutes a "background loop". In the foreground might be interrupt functions, initiated by real time events such as incoming signals or timer overflows. The Round-Robin is non-preemptive. That is it runs each task to completion or until the task relinquishes control. This is quite simply run as a loop as follows:

```
Main()
{
        while(1)
        {
                task1();
                task2();
                task3();
                task4();

        }

}
```

The problem here is that it is static in that all the tasks and their initial running order will be known at compile time.

The foreground interrupts are "pre-emptive" that is that the can preempt or override any task in the Round-Robin. Interrupts have priorities and may interrupt each other. For this reason, interrupt routines should be short and fast. These are now often written in C but it was quite common in the past to use assembler. Assembler is still used for interrupts so that they are as fast and small as possible.

NOTE interrupts can be set to use one of the four sets of register banks using the "using" keyword. This means that there does not have to be a register save and restore in the interrupt making the interrupt faster. By default the program uses bank0 so careful use of the 4 banks can greatly enhance performance. The problem is that like all these tips there is no hard and fast set of rules.

Usually data can be passed from background to foreground (or vice-versa) via global variables and flags. This essentially simple program model can be very successful if some care is taken over the order and frequency of execution of particular sections.

The background-called functions must be written so that they can either complete quickly or if to long for a single slice they run a particular section of their code on each successive entry from the background loop. Thus each function is entered, a decision is taken as to which section of code do this time, the code is executed and finally the "program" is exited. Usually with some special control flags set up to tell the routine program what to do next time. Thus each functional block must maintain its own control system to ensure that the right code is run on any particular entry.

```
Task1()
{
static uint8_t  flag;

        switch(flag)
        {
                case 1:
                        section1();
                        break;
                case2:
                        section2();
                        break;
                default:
                        error();
        }

}
```

Do remember though that this code is also taking up time and the scheduler itself will affect performance. Commercial RTOS will have figures for task, or context swapping etc. You will have to determine your own.

An alternative is to control overall execution from a real time interrupt so that each job is allocated a certain amount of time in which to run. If a timeout does occur, that task is suspended and another begins. This is more complex as the state of the tasks has to be saved somewhere (and quickly). It does give the flexibility that tasks can be allocated different run times. I.e. The task loads a “time” to the interrupt counter.

In the Round-robin system all functional blocks are considered to be of equal importance and no new block can be entered until its turn is reached by the background loop. Only interrupt routines can break this, with each one having its own priority. Should a block need a certain input signal, it can either keep watching until the signal arrives, so holding up all other parts, or it can wait until the next entry, next time round the loop. Now there is the possibility that the event will have been and gone before the next entry occurs. This type of system is OK for situations where the time-critical parts of the program are small.

In reality many real time systems are not like this. Typically, they will consist of some frequently used code, the execution of which is caused by, or causes, some real-world event. This code is fed data from other parts of the system, whose own inputs may be changing rapidly or slowly.

Code which contributes to the system’s major functionality must obviously take precedence over those sections whose purpose is not critical to the successful completion of the task. However most embedded 8051 applications are very time-critical, with such parts being attached to interrupts. The need to service as many interrupts as quickly as possible requires that interrupt code run times are short. With most real world events being asynchronous, the system will ultimately crash when too many interrupt requests occur per unit time for the cpu to cope with.

Fast runtimes and hence acceptable system performance are normally achieved by moving complex functions into the background loop, leaving the time-critical sections in interrupts. This gives rise to the problem of communication between background code and its dependant interrupt routine.

The simple system is very egalitarian, with all parts treated in the same way. When the CPU becomes very heavily loaded with high speed inputs, it is likely that major sub-functions will not be run frequently enough for the real-world interrupt code to be able to run with sufficiently up to date information from the background. Thus, system transient response is degraded.

5.5.3 Simple Scheduling - A Partial Solution

The problems of the simple loop system can be partially solved by controlling the order and frequency of function calling. One approach is to attach a priority to each function and allow the scheduler to decide the next task to be executed based on the flags set. The real-world driven interrupt functions would override this steady progression so that the most important (highest priority) jobs are executed as soon as the current job is completed. This kind of system can yield useful results, provided that no single function takes too long.

```
Main()
{
        while()
        {
                Switch(Priority)
                {
                        case 1:
                                task1();
                                task2();
                                break:
                        case 2:
                                task3();
                                break:
                        case 3:
                                task4();
                                task5();
                                break:
                        default:
                        task5();
                }
        }

}
```

Unfortunately all these tend to be bolt-ons, added late in a project when run times are getting too long. Usually what had been a well-structured program degenerates into spaghetti code, full of fixes and special modes, designed to overcome the fundamental mismatch between the demands of real time events and the response of the program. Moreover, the individual control mechanisms of the called functions generate an overhead which simply contributes to the runtime bottle-neck.

The reality is that real time events are not orderly and predictable. Some jobs are naturally more important than others. However inconvenient, the real world produces events that must be responded to immediately.

It is best to prototype first and look at “what if” scenarios. You will find that with a little thought a simple priority flag system can be developed. It is better to it in the initial stages where timing can be worked out. From experience I have found that in the case of a simple system that degenerates into bolt-ons and “quick” fixes it is better to re do the scheduler from scratch. Do not be tempted to re do the whole program. Just the scheduler not the tasks. This stops the system overheads becoming the problem.

Frequency: However, under normal conditions it is a useful way of ensuring that low priority tasks are not executed frequently. For example, there would be little point in measuring ambient temperature more than once per second. In a typical system, this measurement might be at level 100 in a switch scheduler.

To be able to make a judgement about how best to structure the program, it is vital to know the run times for each section.

Where this simple method falls down is when a low priority task has a long run time. Even though the interrupt has requested that the loop returns back to the top level to calculate more data, there is no way of exiting the task until completed. To do so requires a proper time-slice mechanism.

A useful dodge can be to utilize an unused interrupt to guarantee that high priority tasks will be run on time. By setting The most important factor overall is to keep run times as short as possible, particularly in interrupt routines. This means making full use of C51 extensions like memory-specific pointers, special function bits and local register variables.

6 C Language Extensions For 8051 Programming

Whilst there is an ISO standard for C and everyone tries to produce ISO C compliant compilers and tools it is not always very efficient to use pure ISO C in embedded work. On the larger 32 and 64 bit systems the vast majority of the embedded interfaceing is hidden by opperating systems. Even in 16 bit there is the space to insulate the programmer from the hardware. However, when one gets to the 8 bit systems and the 8051 in particular there is no room to hide the hardware. The programmer must directly interface to the hardware. 8051 programming is mainly concerned with accessing real devices at specific locations, plus coping with interrupt servicing.

In the past the interfacing to the hardware would have been done on assembler. In some cases it stil is. Keil however, along with most other 8051 compiler venders, has made extensions to the C language to allow near-assembler code efficiency when talking to the hardware. In many cases this is not just about speed but due tothe architecture of the 8051. For example the BDATA bit addressable area of memory. The main points are now covered.

6.1 Accessing 8051 On-Chip Peripherals

In the typical embedded control application, reading and writing port data, setting timer registers and reading input captures etc. are commonplace. To cope with this without recourse to assembler, C51 has the special data types sfr and sbit.

Typical declarations are:

```
sfr     P0  =   0x80
sfr     P1  =   0x81
sfr     SCON=   0x98
sbit    EA  =   0xAF
```

and so on.

These declarations reside in header files such as reg51.h for the basic 8051 or reg552.h for the 80C552 and so on. It is the definition of sfrs in these header files that customises the compiler to the target processor. Accessing the sfr data is then a simple matter:

```
{
ADCON = 0x08 ;   /* Write data to register */
P1 = 0xFF    ;   /* Write data to Port */

io_status = P0 ; /* Read data from Port */
EA = 1       ;   /* Set a bit (enable all interrupts) */

}
```

It is worth noting that control bits in registers which are not part of Intel's original 8051 design generally cannot be bit-addressed.

The rule is usually that addresses that are divisible by 8 are bit addressable. Thus for example, the serial Port 1 control bits in an 80C537 must be addressed via byte instructions and masking.

Always check the processor's user manual to verify which sfr register bits can be bit addressed.

6.2 Interrupts

Interrupts play an important part in most 8051 applications. There are several factors to be taken into account when servicing an interrupt:

(i) The correct vector must be generated so that the routine may be called. C51 does this automatically.

(ii) The local variables in the service routine must not be shared with locals in the background loop code: the L51
linker will try to re-use locations so that the same byte of RAM will have different significance depending on
which function is currently being executed. This is essential to make best use of the limited internal memory.
Obviously this relies on functions being executed only sequentially. Unexpected interrupts cannot therefore use
the same RAM.

6.2.1 The Interrupt Function Type

To allow C coding of interrupts a special function type is used thus;

```
timer0_int() interrupt 1 using 2
{
unsigned char temp1 ;
unsigned char temp2 ;
executable C statements ;
}
```

Firstly, the argument of the "interrupt" statement, "1" causes a vector to be generated at (8*n+3), where n is the argument of the "interrupt" declaration. Here a "LJMP timer0_int" will be placed at location 0BH in the code memory. Any local variables declared in the routine are not overlaid by the linker to prevent the overwriting of background variables.

Logically, with an interrupt routine, parameters cannot be passed to it or returned. When the interrupt occurs, compiler-inserted code is run which pushes the accumulator, B,DPTR and the PSW (program status word) onto the stack. Finally, on exiting the interrupt routine, the items previously stored on the stack are restored and the closing "}" causes a RETI to be used rather than a normal RET.

6.2.2 Using C51 With Target Monitor Debuggers

Many simple 8032 target debuggers place the monitor's EPROM code at 0, with a RAM mapped into both CODE and XDATA spaces at 0x8000. The user's program is then loaded into the RAM at 0x8000 and, as the PSEN is ANDed with the RD pin, the program is executed. This poses something of a problem as regards interrupt vectors. C51/L51 assume that the vectors can be placed at 0. Most monitors for the 8032 foresee this problem by redirecting all the interrupt vectors up to 0x8000 and above, i.e. they add a fixed offset of 0x8000. Thus the timer 0 overflow interrupt is redirected by a vector at C:0x000B to C:0x800B.

Before C51 v3.40 the interrupt vector generation had to be disabled and assembler jumps had to be inserted. However now the INTVECTOR control has been introduced to allow the interrupt vector area to be based at any address.

In most cases the vector area will start at 0x8000 so that the familar "8 * n + 3" formula outlined in section 5.2.1 effectively becomes:

8 * n + 3 + INTVECTOR

To use this:

```
#pragma INTVECTOR(0x8000)   /* Set vector area start to 0x8000 */

void timer0_int(void) interrupt 1 {

   /* CODE...*/

   }
```

This produces an LJMP timer0_int at address C:0x800B. The redirection by the monitor from C:0x000B will now work correctly.

6.2.3 Coping Interrupt Spacings Other Than 8

Some 8051's do not follow the normal interrupt spacing of 8 bytes - the '8' in the 8 * n + 3 formula. Fortunately the "INTERVAL #pragma" copes with this.

The interrupt formula is, in reality:

INTERVAL * n + INTVECTOR and so:

```
#pragma INTERVAL(6)   /* Change spacing */
```

7 Pointers In C51

Whilst pointers can be used just as in PC-based C, there are several important extensions to the way they are used in C51. These are mainly aimed at getting more efficient code.

7.1 Using Pointers And Arrays In C51

One of C's greatest strengths can also be its greatest weakness - the pointer. The use and, more appropriately, the abuse of this language feature is largely why C is condemned by some as dangerous!

7.1.1 Pointers In Assembler

For an assembler programmer the C pointer equates closely to indirect addressing. In the 8051 this is achieved by the following instructions:

```
MOV  R0,#40           ; Put on-chip address to be indirectly              MOV  A,@RO
addressed in R0

MOV  R0,#40           ; Put off-chip address to be indirectly
MOVX A,@RO             addressed in R0

MOVX A,@DPTR   ; Put off-chip address to be indirectly
                       addressed in DPTR

CLR  A
MOV  DPTR,#0040   ; Put off-chip address to be indirectly MOVC A,@A+DPTR    addressed in
DPTR
```

In each case the data is held in a memory location indicated by the value in registers to the right of the '@'.

7.1.2 Pointers In C51

The C equivalent of the indirect instruction is the pointer. The register holding the address to be indirectly accessed in the assembler examples is a normal C type, except that its purpose is to hold an address rather than a variable or constant data value.

It is declared by:

```
unsigned char *pointer0 ;
```

Note the asterisk prefix, indicating that the data held in this variable is an address rather than a piece of data that might be used in a calculation etc..

In all cases in the assembler example two distinct operations are required:

(i) Place address to be indirectly addressed in a register.
(ii) Use the appropriate indirect addressing instruction to access data held at chosen address.

Fortunately in C the same procedure is necessary, although the indirect register must be explicitly defined, whereas in assembler the register exists in hardware.

```
/* 1 - Define a variable which will hold an address */

unsigned char *pointer ;

/* 2 - Load pointer variable with address to be accessed*/
        /*indirectly */

pointer = &c_variable ;

/* 3 - Put data ‘0xff’ indirectly into c variable via*/
       /*pointer */

*pointer = 0xff ;
```

Taking each operation in turn...
1. Reserve RAM to hold pointer. In practice the compiler attaches a symbolic name to a RAM location, just as with a normal variable.

2. Load reserved RAM with address to be accessed, equivalent to ‘MOV R0,#40’. In English this C statement means:

“take the ‘address of’ c_variable and put it into the reserved RAM, i.e, the pointer”

In this case the pointer’s RAM corresponds to R0 and the ‘&’ equates loosely to the assembler ‘#’.

3. Move the data indirectly into pointed-at C variable, as per the assembler ‘MOV A,@R0’.

The ability to access data either directly, x = y, or indirectly, x = *y_ptr, is extremely useful. Here is C example:

```
/* Demonstration Of Using A Pointer */

unsigned char c_variable ;   // 1 - Declare a c variable unsigned char *ptr ;
                                     // 2 - Declare a pointer (not pointing at anything
yet!)
main() {

   c_variable = 0xff ;   // 3 - Set variable equal to 0xff directly

   ptr = &c_variable ;   // 4 - Force pointer to point at c_variable at run time

   *ptr = 0xff ;         // 5 - Move 0xff into c_variable indirectly

   }
```

Note: Line 4 causes pointer to point at variable. An alternative way of doing this is at compile time thus:

```
/* Demonstration Of Using A Pointer */

unsigned char c_variable;            //1-Declare a c variable
unsigned char *ptr = &c_variable;    //2-Declare a pointer, intialised to pointing at
                                     //c_variable during compilation

main() {
   c_variable = 0xff ;   // 3 - Set variable equal to 0xff directly

   *ptr = 0xff           // 5 - Move 0xff into c_variable ndirectly
   }
```

Pointers with their asterisk prefix can be used exactly as per normal data types. The statement:

```
x = y + 3 ;
```

could equally well perform with pointers, as per

```
char x, y ;
char *x_ptr = &x ;
char *y_ptr = &y ;
*x_ptr = *y_ptr + 3 ;
```

or:

```
x = y * 25 ;
*x_ptr = *y_ptr * 25 ;
```

The most important thing to understand about pointers is that

```
*ptr = var ;
```

means "set the value of the pointed-at address to value var", whereas

```
ptr = &var ;
```

means "make ptr point at var by putting the address of (&) in ptr, but do not move any data out of var itself".

Thus the rule is to initialise a pointer,

```
ptr = &var ;
```

To access the data indicated by *ptr ;

```
var = *ptr ;
```

7.2 Pointers To Absolute Addresses

In embedded C, ROM, RAM and peripherals are at fixed addresses. This immediately raises the question of how to make pointers point at absolute addresses rather than just variables whose address is unknown (and largely irrelevant).

The simplest method is to determine the pointed-at address at compile time:

```
char *abs_ptr = 0x8000 ;  // Declare pointer and force to 0x8000 immediately
```

However if the address to be pointed at is only known at run time, an alternative approach is necessary. Simply, an uncommitted pointer is declared and then forced to point at the required address thus:

```
Unsigned char *abs_ptr ;  // Declare uncommitted pointer

abs_ptr = (char *) 0x8000 ;  // Initialise pointer to 0x8000
*abs_ptr = 0xff ;            // Write 0xff to 0x8000

*abs_ptr++ ;                 // Make pointer point at next location in RAM
```

Please see sections 6.8 and 6.9 for further details on C51 spaced and generic pointers.

7.3 Arrays And Pointers - Two Sides Of The Same Coin?

7.3.1 Uninitialised Arrays

The variables declared via

```
unsigned char x ;
unsigned char y ;
```

are single 8 bit memory locations. The declarations:

```
unsigned int a ;
unsigned int b ;
```

yield four memory locations, two allocated to ‘a’ and two to ‘b’. In other programming languages it is possible to group similar types together in arrays. In basic an array is created by DIM a(10).

Likewise ‘C’ incorporates arrays, declared by:

```
unsigned char a[10] ;
```

This has the effect of generating ten sequential locations, starting at the address of ‘a’. As there is nothing to the right of the declaration, no initial values are inserted into the array. It therefore contains zero data and serves only to reserve ten contiguous bytes.

7.3.2 Initialised Arrays

A more usual instance of arrays would be:

unsigned char test_array [] = { 0x00,0x40,0x80,0xC0,0xFF } ;

where the initial values are put in place before the program gets to “main()”. Note that the size of this initialised array is not given in the square brackets - the compiler works-out the size automatically.

Another common instance of an array is analogous to the BASIC string as per:

```
A$ = “HELLO!”
```

In C this equates to:

```
char test_array[] = { “HELLO!” } ;
```

In C there is no real distinction between strings and arrays as a C array is just a series of sequential bytes occupied either by a string or a series of numbers. In fact the realms of pointers and arrays overlap with strings by virtue of :

```
char test_array = { “HELLO!” } ;
char *string_ptr = { “HELLO!” } ;
```

Case 1 creates a sequence of bytes containing the ASCII equivalent of “HELLO!”. Likewise the second case allocates the same sequence of bytes but in addition creates a separate pointer called *string_ptr to it. Notice that the “unsigned char” used previously has become “char”, literally an ASCII character.

The second is really equivalent to:

```
char test_array = { “HELLO!” } ;
```

Then at run time:

```
char arr_ptr = test_array ;  // Array treated as pointer
```

or;

```
char arr_ptr = &test_array[0] ; // Put address of first
                                       // element of array into
                                       // pointer
```

This again shows the partial interchangeability of pointers and arrays. In English, the first means “transfer address of test_array into arr_ptr”. Stating an array name in this context causes the array to be treated as a pointer to the first location of the array. Hence no “address of” (&) or ‘*’ to be seen.

The second case reads as “get the address of the first element of the array name and put it into arr_ptr”. No implied pointer conversion is employed, just the return of the address of the array base.

The new pointer “*arr_ptr” now exactly corresponds to *string_ptr, except that the physical “HELLO!” they point at is at a different address.

7.3.3 Using Arrays

Arrays are typically used like this:

```
/* Copy The String HELLO! Into An Empty Array */

unsigned char source_array[] = { “HELLO!” } ;
unsigned char dest_array[7];
unsigned char array_index ;
unsigned char

array_index = 0 ;

while(array_index < 7) {  // Check for end of array

dest_array[array_index] = source_array[array_index] ;
       //Move character-by-character into destination array

       array_index++ ;
   }
```

The variable array_index shows the offset of the character to be fetched (and then stored) from the starts of the arrays.

As has been indicated, pointers and arrays are closely related. Indeed the above program could be re-written thus:

```
/* Copy The String HELLO! Into An Empty Array */

char *string_ptr = { “HELLO!” } ;
unsigned char dest_array[7] ;
unsigned char array_index  ;
unsigned char

array_index = 0 ;

while(array_index < 7) {      // Check for end of array

dest_array[array_index] = string_ptr[array_index] ;  // Move character-by-character into
destination array.
array_index++ ;
   }
```

The point to note is that by removing the ‘’ on string_ptr and appending a ‘[ ]’ pair, this pointer has suddenly become an array! However in this case there is an alternative way of scanning along the HELLO! string, using the *ptr++ convention:*

```
array_index = 0 ;

while(array_index < 7) { // Check for end of array

 dest_array[array_index] = *string_ptr++ ; // Move character-by-character into
destination array.
 array_index++ ;
   }
```

This is an example of C being somewhat inconsistent; this *ptr++ statement does not mean “increment the thing being pointed at” but rather, increment the pointer itself, so causing it to point at the next sequential address. Thus in the example the character is obtained and then the pointer moved along to point at the next higher address in memory.

7.3.4 Summary Of Arrays And Pointers

To summarise:

Create An Uncommitted Pointer

```
unsigned char *x_ptr ;
```

Create A Pointer To A Normal C Variable

```
unsigned char x ; unsigned char *x_ptr = &x ;
```

Create An Array With No Initial Values

```
unsigned char x_arr[10] ;
```

Create An Array With Initialised Values

```
unsigned char x_arr[] = { 0,1,2,3 } ;
```

Create An Array In The Form Of A String

```
char x_arr[] = { “HELLO” } ;
```

Create A Pointer To A String

```
char *string_ptr = { “HELLO” } ;
```

Create A Pointer To An Array

```
char x_arr[] = { “HELLO” } ; char *x_ptr = x_arr
```

Force A Pointer To Point At The Next Location

```
*ptr++ ;
```

7.4 Structures

Structures are perhaps what makes C such a powerful language for creating very complex programs with huge amounts of data. They are basically a way of grouping together related data items under a single symbolic name.

7.4.1 Why Use Structures?

Here is an example: A piece of C51 software had to perform a linearisation process on the raw signal from a variety of pressure sensors manufactured by the same company. For each sensor to be catered for there is an input signal with a span and offset, a temperature coefficient, the signal conditioning amplifier, a gain and offset. The information for each sensor type could be held in “normal” constants thus:

```
unsigned char sensor_type1_gain = 0x30 ;
unsigned char sensor_type1_offset = 0x50 ;
unsigned char sensor_type1_temp_coeff = 0x60 ;
unsigned char sensor_type1_span = 0xC4 ;
unsigned char sensor_type1_amp_gain = 0x21 ;

unsigned char sensor_type2_gain = 0x32 ;
unsigned char sensor_type2_offset = 0x56 ;
unsigned char sensor_type2_temp_coeff = 0x56 ;
unsigned char sensor_type2_span = 0xC5 ;
unsigned char sensor_type2_amp_gain = 0x28 ;
unsigned char sensor_type3_gain = 0x20 ;
unsigned char sensor_type3_offset = 0x43 ;
unsigned char sensor_type3_temp_coeff = 0x61 ;
unsigned char sensor_type3_span = 0x89 ;
unsigned char sensor_type3_amp_gain = 0x29 ;
```

As can be seen, the names conform to an easily identifiable pattern of:

```
unsigned char sensor_typeN_gain = 0x20 ;
unsigned char sensor_typeN_offset = 0x43 ;
unsigned char sensor_typeN_temp_coeff = 0x61 ;
unsigned char sensor_typeN_span = 0x89 ;
unsigned char sensor_typeN_amp_gain = 0x29 ;
```

Where ‘N’ is the number of the sensor type. A structure is a neat way of condensing this type is related and repeating data.

In fact the information needed to describe a sensor can be reduced to a generalised:

```
unsigned char gain ;
unsigned char offset ;
unsigned char temp_coeff ;
unsigned char span ;
unsigned char amp_gain ;
```

The concept of a structure is based on this idea of generalised "template" for related data. In this case, a structure template (or "*component list*") describing any of the manufacturer's sensors would be declared:

```
struct sensor_desc {unsigned char gain ;
                         unsigned char offset ;
                         unsigned char temp_coeff ;
                         unsigned char span ;
                         unsigned char amp_gain ; } ;
```

This does not physically do anything to memory. At this stage it merely creates a template which can now be used to put real data into memory.

This is achieved by:

```
struct sensor_desc sensor_database ;
```

This reads as "use the template sensor_desc to layout an area of memory named sensor_database, reflecting the mix of data types stated in the template". Thus a group of 5 unsigned chars will be created in the form of a structure.

The individual elements of the structure can now be accessed as:

```
sensor_database.gain = 0x30 ;
sensor_database.offset = 0x50 ;
sensor_database.temp_coeff = 0x60 ;
sensor_database.span = 0xC4 ;
sensor_database.amp_gain = 0x21 ;
```

7.4.2 Arrays Of Structures

In the example though, information on many sensors is required and, as with individual chars and ints, it is possible to declare an array of structures. This allows many similar groups of data to have different sets of values.

```
struct sensor_desc sensor_database[4] ;
```

This creates four identical structures in memory, each with an internal layout determined by the structure template. Accessing this array is performed simply by appending an array index to the structure name:

```
/*Operate On Elements In First Structure Describing */
/*Sensor 0 */

sensor_database[0].gain = 0x30 ;
sensor_database[0].offset = 0x50 ; sensor_database[0].temp_coeff = 0x60 ;
sensor_database[0].span = 0xC4 ;
sensor_database[0].amp_gain = 0x21 ;

/* Operate On Elements In First Structure Describing */
/*Sensor 1 */

sensor_database[1].gain = 0x32 ;
sensor_database[1].offset = 0x56 ;
```

```
sensor_database[1].temp_coeff = 0x56 ;
sensor_database[1].span = 0xC5 ;
sensor_database[1].amp_gain = 0x28 ;
```

and so on...

7.4.3 Initialised Structures

As with arrays, a structure can be initialised at declaration time:

```
struct sensor_desc sensor_database = { 0x30, 0x50, 0x60, 0xC4, 0x21 } ;
```

so that here the structure is created in memory and pre-loaded with values.

The array case follows a similar form:

```
struct sensor_desc sensor_database[4] = {{0x30,0x50,0x60, 0xC4, 0x21 },

{ 0x32,0x56,0x56,0xC5,0x28 ; }} ;
```

7.4.4 Placing Structures At Absolute Addresses

It is sometimes necessary to place a structure at an absolute address. This might occur if, for example, the registers of a memory-mapped real time clock chip are to be grouped together as a structure. The template in this instance might be:

Contents Of RTCBYTES.C Module

```
        struct RTC { unsigned char seconds ;
                         unsigned char minutes ;
                         unsigned char hours   ;
                         unsigned char days    ;
} ;

struct RTC xdata RTC_chip ;  // Create xdata structure
```

There are two ways of doing this. The more common method now used is the "_at_" keyword. This was introduced in version C51 3.4

```
struct RTC xdata RTC_chip _at_ 0x200;  // Create xdata structure at X:0x0200
```

The AT keyword can be used to place data in any memory space. There is more information on the _at_ keyword in section 8.7

The other method is uses the linker. A trick using the linker is required here so the structure creation must be placed in a dedicated module. This module's XDATA segement, containing the RTC structure, is then fixed at the required address at link time. This can be used to place not only structures but modules containing anything else at a fixed point.

Using the absolute structure could be:

```
/* Structure located at base of RTC Chip */

MAIN.C Module

extern xdata struct RTC_chip ;
```

```
/* Other XDATA Objects */

xdata unsigned char time_secs, time_mins ;

void main(void) {

time_secs = RTC_chip.seconds ;
time_mins = RTC_chip.minutes;
}
```

Linker Input File To Locate RTC_chip structure over real RTC Registers is:

```
BL51 main.obj,rtcbytes.obj XDATA(?XD?RTCBYTES(0h))
```

NOTE: Older compilers may still use L51 and the newer (V6 and up) PK51 user may use LX51 instread of BL51

See section 7.6 for further examples of this placement method.

7.4.5 Pointers To Structures

Pointers can be used to access structures, just as with simple data items. Here is an example:

```
/* Define pointer to structure */

struct sensor_desc *sensor_database ;

/* Use Pointer To Access Structure Elements */

sensor_database->gain = 0x30 ;
sensor_database->offset = 0x50 ;
sensor_database->temp_coeff = 0x60 ;
sensor_database->span = 0xC4 ;
sensor_database->amp_gain = 0x21 ;
```

Note that the '*' which normally indicates a pointer has been replaced by appending '->' to the pointer name. Thus '*name' and 'name->' are equivalent.

7.4.6 Passing Structure Pointers To Functions

A common use for structure pointers is to allow them to be passed to functions without huge amounts of parameter passing; a typical structure might contain 20 data bytes and to pass this to a function would require 20 parameters to either be pushed onto the stack or an abnormally large parameter passing area. By using a pointer to the structure, only the two or three bytes that constitute the pointer need be passed. This approach is recommended for C51 as the overhead of passing whole structures can tie the poor old 8051 CPU in knots!

This would be achieved thus:

```
struct sensor_desc *sensor_database ;

sensor_database-> gain = 0x30 ;
sensor_database-> offset = 0x50  ;
sensor_database-> temp_coeff = 0x60 ;
sensor_database-> span = 0xC4 ;
sensor_ database- >amp_gain = 0x21 ;
```

```
test_function(*struct_pointer) ;

test_function(struct sensor_desc *received_struct_pointer) {
   received_struct_pointer->gain = 0x20 ;
   received_struct_pointer->temp_coef = 0x40 ;
   }
```

Advanced Note: Using a structure pointer will cause the called function to operate directly on the structure rather than on a copy made during the parameter passing process.

7.4.7 Structure Pointers To Absolute Addresses

It is sometimes necessary to place a structure at an absolute address. This might occur if, for example, a memory-mapped real time clock chip is to be handled as a structure. An alternative approach to that given in section 6.4.4. is to address the clock chip via a structure pointer.

The important difference is that in this case no memory is reserved for the structure - only an “image” of it appears to be at the address.

The template in this instance might be:

```
/* Define Real Time Clock Structure */

struct RTC {char seconds ;
               char mins ;
               char hours ;
               char days ; } ;

/* Create A Pointer To Structure */

struct RTC xdata *rtc_ptr ;  // ‘xdata’ tells C51 that this
                                   //is a memory-mapped device.


void main(void) {
   rtc_ptr = (void xdata *) 0x8000 ;  // Move structure
                                               // pointer to address
                                               //of real time clock at
                                               // 0x8000 in xdata

        rtc_ptr->seconds = 0 ;  // Operate on elements
        rtc_ptr->mins = 0x01 ;
   }
```

This general technique can be used in any situation where a pointer-addressed structure needs to be placed over a specific IO device. However it is the user’s responsibility to make sure that the address given is not likely to be allocated by the linker as general variable RAM!

To summarize, the procedure is:

(i) Define template
(ii) Declare structure pointer as normal
(iii) At run time, force pointer to required absolute address in the normal way.

7.5 Unions

A union is similar in concept to a structure except that rather than creating sequential locations to represent each of the items in the template, it places each item at the same address. Thus a union of 4 bytes only occupies a single byte. A union may consist of a combination of longs, char and ints all based at the same physical address.

The the number of bytes of RAM used by a union is simply determined by the size of the largest element, so:

```
union test { char x ;
             int y  ;
             char a[3] ;
             long z ;
} ;
```

requires 4 bytes, this being the size of a long. The physical location of each element is:

```
addr – 0   x byte  y high byte a[0]  z highest byte
       +1           y low byte a[1]  z byte
       +2                      a[2]  z byte
       +3                      a[3]  z lowest byte
```

Non-8051 programmers should see the section on byte ordering in the 8051 if they find the idea of the MSB being at the low address odd!

```
In embedded C the commonest use of a union is to allow fast access to individual bytes of
longs or ints.  These might be 16 or 32 bit real time counters, as in this example:

/* Declare Union */

union clock {long real_time_count ; // Reserve four byte
        int real_time_words[2] ;      // Reserve four bytes as
                                              // int array
        char real_time_bytes[4] ;     // Reserve four bytes as
                                            // char array
      } ;

/* Real Time Interrupt */

void timer0_int(void) interrupt 1 using 1 {

        clock.real_time_count++ ;       // Increment clock

        if(clock.real_time_words[1] == 0x8000) { // Check
                                      // lower word only for value

        /* Do something! */
        }

        if(clock.real_time_bytes[3] == 0x80) {  // Check most
                                 // significant byte only for value

        /* Do something! */
        }

        }
```

7.6 Generic Pointers

C51 offers two basic types of pointer, the spaced (memory-specific) and the generic. Up to version 3.00 only generic pointers were available.

As has been mentioned, the 8051 has many physically separate memory spaces, each addressed by special assembler instructions. Such characteristics are not peculiar to the 8051 - for example, the 8086 has data instructions which operate on a 16 bit (within segment) and a 20 bit basis.

For the sake of simplicity, and to hide the real structure of the 8051 from the programmer, C51 uses three byte pointers, rather than the single or two bytes that might be expected. The end result is that pointers can be used without regard to the actual location of the data.
For example:

```
 xdata char buffer[10] ;
 code char message[] = { “HELLO” } ;
void main(void) {
       char *s ;
       char *d ;

       s = message ;
       d = buffer ;

       while(*s != ‘\0’) {
          *d++ = *s++ ;
          }
   }
```

Yields:

```
        RSEG  ?XD?T1
buffer:                DS  10
        RSEG  ?CO?T1
message:
        DB  ‘H’ ,’E’ ,’L’ ,’L’ ,’O’ ,000H
;
;
; xdata char buffer[10] ;
; code char message[] = { “HELLO” } ;
;
;    void main(void) {
        RSEG  ?PR?main?T1
        USING   0
main:
                       ; SOURCE LINE # 6
;
;       char *s ;
;       char *d ;
;
;       s = message ;
                       ; SOURCE LINE # 11
        MOV     s?02,#05H
        MOV     s?02+01H,#HIGH message
        MOV     s?02+02H,#LOW message
;       d = buffer ;
                       ; SOURCE LINE # 12
        MOV     d?02,#02H
        MOV     d?02+01H,#HIGH buffer
        MOV     d?02+02H,#LOW buffer
?C0001:
;
;       while(*s != ‘\0’) {
                       ; SOURCE LINE # 14
        MOV     R3,s?02
        MOV     R2,s?02+01H
        MOV     R1,s?02+02H
        LCALL   ?C_CLDPTR
        JZ      ?C0003
;          *d++ = *s++ ;
```

```
                       ; SOURCE LINE # 15
        INC     s?02+02H
        MOV     A,s?02+02H
        JNZ     ?C0004
        INC     s?02+01H
?C0004:
        DEC     A
        MOV     R1,A
        LCALL   ?C_CLDPTR
        MOV     R7,A
        MOV     R3,d?02
        INC     d?02+02H
        MOV     A,d?02+02H
        MOV     R2,d?02+01H
        JNZ     ?C0005
        INC     d?02+01H
?C0005:
        DEC     A
        MOV     R1,A
        MOV     A,R7
        LCALL   ?C_CSTPTR
;          }
                       ; SOURCE LINE # 16
        SJMP    ?C0001
;       }
                       ; SOURCE LINE # 17
?C0003:
        RET
; END OF main
        END
```

As can be seen, the pointers '*s' and '*d' are composed of three bytes, not two as might be expected. In making *s point at the message in the code space an '05' is loaded into s ahead of the actual address to be pointed at. In the case of *d '02' is loaded. These additional bytes are how C51 knows which assembler addressing mode to use. The library function C_CLDPTR checks the value of the first byte and loads the data, using the addressing instructions appropriate to the memory space being used.

This means that every access via a generic pointer requires this library function to be called. The memory space codes used by C51 are:

```
CODE  - 05
XDATA - 02
PDATA - 03
DATA  - 05
IDATA - 01
```

7.7 Spaced Pointers In C51

Considerable run timesavings are possible by using memory spaced pointers. By restricting a pointer to only being able to point into one of the 8051's memory spaces, the need for the memory space "code" byte is eliminated, along with the library routines needed to interpret it.

DATA/BDATA/IDATA pointers are 1 byte long whereas XDATA/PDATA/CODE pointers are 2 bytes long.

A spaced pointer is created thus:

```
char xdata *ext_ptr ;
```

to produce an uncommitted pointer into the XDATA space or

```
char code *const_ptr ;
```

which gives a pointer solely into the CODE space. Note that in both cases the pointers themselves are located in the memory space given by the current memory model. Thus a pointer to xdata which is to be itself located in PDATA would be declared thus:

```
pdata char xdata *ext_ptr ;
  |              |
location         |
of pointer       |
               Memory space pointed into
               by pointer
```

In this example strings are always copied from the CODE area into an XDATA buffer. By customising the library function “strcpy()” to use a CODE source pointer and a XDATA destination pointer, the runtime for the string copy was reduced by 50%. The new strcpy has been named strcpy_x_c().

The function prototype is:

```
extern char xdata *strcpy(char xdata*,char code *) ;
```

Here is the code produced by the spaced pointer strcpy():

```
; char xdata *strcpy_x_c(char xdata *s1, char code *s2)  {
_strcpy_x_c:
        MOV     s2?10,R4
        MOV     s2?10+01H,R5
;—— Variable ‘s1?10’ assigned to Register ‘R6/R7’ ——
;   unsigned char i = 0;
;—— Variable ‘i?11’ assigned to Register ‘R1’ ——
        CLR     A
        MOV     R1,A
?C0004:
;
;   while ((s1[i++] = *s2++) != 0);
        INC     s2?10+01H
        MOV     A,s2?10+01H
        MOV     R4,s2?10
        JNZ     ?C0008
        INC     s2?10
?C0008:
        DEC     A
        MOV     DPL,A
        MOV     DPH,R4
        CLR     A
        MOVC    A,@A+DPTR
        MOV     R5,A
        MOV     R4,AR1
        INC     R1
        MOV     A,R7
        ADD     A,R4
        MOV     DPL,A
        CLR     A
        ADDC    A,R6
        MOV     DPH,A
        MOV     A,R5
        MOVX    @DPTR,A
        JNZ     ?C0004
?C0005:
;   return (s1);
; }
?C0006:
        END
```

Notice that no library functions are used to determine which memory spaces are intended. The function prototype tells C51 only to look in code for the string and xdata for the RAM buffer.

8 Accessing External Memory Mapped Peripherals

Commonly, extra IO ports are added to 8051s to compensate for the loss of Ports 0 and 2. This is normally done by making the additional device(s) appear to be just external RAM bytes. Thus they are addressed by the MOVX A,@DPTR instruction. Typically UARTS, additional ports and real time clock devices are added to 8031s as xdata-mapped devices.

The simplest approach to adding external devices is to attach the /RD and or /WR lines to the device. Provided that only one device is present and that it only has one register, no address decoding is necessary. To access this device from C simply prefix an appropriately named variable with “xdata”. This will cause the compiler to use MOVX A,@DTPR instructions when getting data in or out. In actual fact the linker will try to allocate a real address to this but, as no decoding is present, the device will simply be enabled by /WR or /RD.

In practice life is rarely this simple. Usually a mixture of RAM, UARTS, ports, EEPROM and other devices may all be attached to the 8031 by being mapped into the xdata space. Some sort of decoding is provided by discrete logic or (more usually) a PAL.

Here the various registers of the different devices will appear at fixed locations in the xdata space. With normal on-chip resources the simple “data book” name can be used to access them, so ideally these external devices should be the same.

There are three basic approaches to this:

(i) Use normal variables, char, ints etc, located by the linker
(ii) Use pointers and offsets, either via the XBYTE macros or directly with user-defined pointers.
(iii) Use the _At_ and _ORDER directives.

In detail, these may be implemented as shown in the following sections.

8.1 The XBYTE And XWORD Macros

To allow memory-mapped devices to be accessed from C, a method is required to effectively force pointers to point to fixed addresses. C51 provides many methods of achieving this, the simplest of which are the XBYTE[addr16] and XWORD[addr16] macros.

For instance:

The byte wide PORT8_DDI register of a memory mapped IO device is at 8000H. To access it from C it must be declared thus:

```
#include “absacc.h”;   /*Contains macro definitions */
#define port8_ddi   XBYTE[0x8000]
#define port8_data  XBYTE[0x8001]
```

To use it then,

```
port8_ddi = 0xFF       ;
input_val = port8_data ;
```

To access a word at an even external address:

```
        #define word_reg XWORD[0x4000]
        /* gives a word variable at 8000H */
```

Ignoring the pre-defined XWORD macro, the equivalent C line is:

```
        #define word_reg_ptr ((unsigned int *) 0x24000L)
        /*creates a pointer to a word (int) at address 8000H*/
```

To use this address then,

```
        *word_reg_ptr = 0xFFFF ;
```

Note that the address 8000H corresponds to 4000H words, hence the " 0x24000L ".

Here are some examples with the code produced:

```
#define XBYTE ((unsigned char volatile *) 0x20000L)
#define XWORD ((unsigned int volatile *) 0x20000L)

main() {

char x ;
 int y ;

x = XBYTE[0x8000]       ;

0000 908000         MOV     DPTR,#08000H
0003 E0             MOVX    A,@DPTR
0004 FF             MOV     R7,A
0005 8F00    R      MOV     x,R7

y = XWORD[0x8000/sizeof(int)] ;
}
0007 908000         MOV     DPTR,#08000H
000A E0             MOVX    A,@DPTR
000B FE             MOV     R6,A
000C A3             INC     DPTR
000D E0             MOVX    A,@DPTR
000E FF             MOV     R7,A
000F 8E00    R      MOV     y,R6
0011 8F00    R      MOV     y+01H,R7
}
0013         ?C0001:
0013 22             RET
```

However the address indicated by “word_reg” is fixed and can only be defined at compile time, as the contents of the square brackets may only be a constant. Any alteration to the indicated address is not possible with these macro-based methods. This approach is therefore best suited to addressing locations that are fixed in hardware and unlikely to change at run time.

Note the use of the **volatile** storage class modifier. This is essential to prevent the optimiser removing data reads from external ports.
See section 7.4 for more details.

Note: the header file “absacc.h” must be included at the top of the source file as shown above. This contains the prototype for the XBYTE macro. (see page 9-15 in the C51 manual)

8.2 Initialised XDATA Pointers

In many cases the external address to be pointed at is known at compile time but may need to be altered at some point during execution. Thus some method of making a pointer point at an intial specific external address is required.

Probably the simplest way of setting up such a pointer is to let the C_INIT program set the pointer to a location. However the initial address must be known at compile time. If the pointer is to be altered at run time, just equate it (without the “*” at run time) to the new address.

Note: this automatic initialisation was not supported on earlier versions of C51.

Simply do:

```
/* Spaced pointer */

  xdata char xdata *a_ptr = 0x8000 ;

/* Generic Pointer */

  xdata char *a_ptr = 0x028000L ;
```

Here the pointer is setup to point at xdata address 0x8000. Note that the spaced *a_ptr can only point at xdata locations as a result of the second xdata used in its declaration. In the generic *a_ptr case, the “02” tells C51 that an xdata address is intended.
An example might be:

```
   6            xdata char xdata *ptr = 0x8000 ;
   7
   8
   9            main() {
  11   1        char x ;
  13   1        ptr += 0xf0 ;

0000 900000  R    MOV     DPTR,#ptr+01H
0003 E0           MOVX    A,@DPTR
0004 24F0         ADD     A,#0F0H
0006 F0           MOVX    @DPTR,A
0007 900000  R    MOV     DPTR,#ptr
000A E0           MOVX    A,@DPTR
000B 3400         ADDC    A,#00H
000D F0           MOVX    @DPTR,A

  15   1        x = *ptr ;
  16   1
  17   1        }

000E E0           MOVX    A,@DPTR
000F FE           MOV     R6,A
0010 A3           INC     DPTR
0011 E0           MOVX    A,@DPTR
0012 F582         MOV     DPL,A
0014 8E83         MOV     DPH,R6
0016 E0           MOVX    A,@DPTR
0017 F500    R    MOV     x,A

  17   1        }

0019 22           RET
```

8.3 Run Time xdata Pointers

The situation often occurs that you need to point at addresses in the xdata space which are only known at run time. Here the xdata pointer is setup in the executable code.

The best way to achieve this is to declare an “uncommitted” pointer at compile time and to then equate it to an address when running:

```
char xdata *xdata_ptr ;   /* Uncommitted pointer */
                                     /* to xdata memory */
main() {

xdata_ptr=(char xdata*) 0x8000 ; /*Point at 0x8000 in */
                                                  /*xdata */
}
```

An alternative is to declare a pointer to the xdata space and simply equate it to a variable.

```
Here is an example:

   char xdata *ptr ; /* This is a spaced pointer!!! */

   main(){

   start_address = 0x8000 ;  /*Variable containing address*/
                                     /*to be pointed to */

0000 750080  R     MOV     start_address,#080H
0003 750000  R     MOV     start_address+01H,#00H

   ptr = start_address ;

000C AE00    R     MOV     R6,start_address
000E AF00    R     MOV     R7,start_address+01H
0010 8E00    R     MOV     ptr,R6
0012 8F00    R     MOV     ptr+01H,R7
0014         ?C0001:

   while(1) {

   x = *ptr++ ;

0014 0500    R     INC     ptr+01H
0016 E500    R     MOV     A,ptr+01H
0018 AE00    R     MOV     R6,ptr
001A 7002          JNZ     ?C0004
001C 0500    R     INC     ptr
001E         ?C0004:
001E 14            DEC     A
001F FF            MOV     R7,A

0020 8F82          MOV     DPL,R7
0022 8E83          MOV     DPH,R6
0024 E0            MOVX    A,@DPTR
0025 FF            MOV     R7,A
0026 8F00    R     MOV     x,R7
   }
0028 80EA          SJMP    ?C0001
002A         ?C0002:
  }
002A         ?C0003:
002A 22            RET
```

A variation of this is to declare a pointer to zero and use a variable as an offset thus:

```
char xdata *ptr ;

main() {

unsigned int i ;
unsigned char x ;

ptr = (char*) 0x0000 ;

for(i = 0 ;
i < 0x40 ;
i++) {
   x = ptr[i] ;
   }
}
```

This results in rather more code, as an addition to the pointer must be performed within each loop.

8.4 The "volatile" Storage Class

A common situation with external devices is that values present in their registers change without the cpu taking any action. A good example is a real time clock chip - the time changes continuously without the cpu writing anything.

Consider the following:

```
unsigned int xdata *milliseconds = 0x8000 ;  // Pointer to
                                                       // RTC chip

time = *milliseconds ;  -> (1)  // Get RTC register value

x = array[time] ;

time = *milliseconds ;  -> (2)  // Second register access
                                      // optimised out!

y = array[time] ;
```

Here the value retrieved from the array is related to the value of *milliseconds, a register in an external RTC.

If this is compiled it will not work. Why? Well the compiler's optimiser shoots itself in the foot by assuming that, because no WRITE occurred between (1) and (2), *millisec cannot have changed. Hence all the code generated to make the second access to the register is optimised out and so y == x!

The solution is declare *milliseconds as "volatile" thus:

```
unsigned int volatile xdata *milliseconds = 0x8000 ;
```

Now the optimiser will not try to remove subsequent accesses to the register.

8.5 Placing Variables At Specific Locations - The Linker Method

A final method of establishing external variables at fixed addresses, especially arrays, is by using the linker rather than the compiler. For example, to produce a 10 character array in external memory, starting at 8000H, the following steps are necessary:

```
/*** Module 1 ***/

/* This module contains only data declarations! */

xdata unsigned char array[30] ;

/* End Module 1 */

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

/*** Module 2 ***/

/* This module contains the executable statements */

extern xdata unsigned char array[10] ;

   main()

   {
   unsigned char i ;

   i = array[i] ;

   }
```

Now by linking with the invocation:

```
L51 module1.obj, module2.obj XDATA (?XD?module1 (8000H))
```

the linker will make the XDATA segment in Module 1 (indicated by ?XD?module1) start at 8000H, regardless of other xdata declarations elsewhere. Thus the array starts at 8000H and is 10 bytes (+ null terminator) long.

This approach lacks the flexibility of the above methods but has the advantage of making the linker reserve space in the XDATA space.

Similar control can be exercised over the address of segments in other memory spaces. C51 uses the following convention for segment names:

```
CODE   ?PR?functionname?module_name  (executable code)
CODE   ?CO?functionname?module_name  (lookup tables etc.)
BIT    ?BI?functionname?module_name
DATA   ?DT?functionname?module_name
XDATA  ?XD?functionname?module_name
PDATA  ?PD?functionname?module_name
```

Thus the parameter receiving area of a LARGE model function 'test()' in module MOD1.C would be:

```
?XD?TEST?MOD1,
```

The code is:

```
?PR?TEST?MOD1
```

And so on.

A knowledge of this is useful for assembler interfacing to C51 programs. See section 14.

8.6 Excluding External Data Ranges From Specific Areas

This very much follows on from the previous section. Occasionally a memory-mapped device, such as real time clock chip, is used as both a source of time values and RAM. Typically the first 8 bytes in the RTC's address range are the time counts, seconds, minutes etc. whilst the remaining 248 bytes are RAM.

Left to its own devices, the L51 linker will automatically place any xdata variables starting at zero. If the RTC has been mapped at this address a problem occurs, as the RTC time registers are overwritten. In addition, it would be convenient to allow the registers to be individually named.

One approach is to define a special module containing just a structure which describes the RTC registers. In the main program the RTC registers are accessed as elements in the structure. The trick is that, when linking, the XDATA segment belonging to the special module is forced to a specific address, here zero. This results in the RTC structure being at zero, with any other XDATA variables following on. The basic method could also be used to stop L51 locating any variables in a specific area.

Example Of Excluding Specific Areas From L51

```
/* Structure located at base of RTC Chip */
```

MAIN.C Module

```
extern xdata struct {   unsigned char seconds ;
                              unsigned char minutes ;
                              unsigned char hours   ;
                              unsigned char days    ; } RTC_chip ;

/* Other XDATA Objects */

xdata unsigned char time_secs, time_mins ;

void main(void) {

time_secs = RTC_chip.seconds ;
time_mins = RTC_chip.minutes ;

}
```

RTCBYTES.C Module

```
xdata struct { unsigned char seconds ;
               unsigned char minutes ;
               unsigned char hours   ;
               unsigned char days    ; } RTC_chip ;
```

Linker Input File To Locate RTC_chip structure over real RTC
Registers is:

```
l51 main.obj,rtcbytes.obj XDATA(?XD?RTCBYTES(0h))
```

8.7 -missing ORDER and AT now in C51

Perhaps the most curious omission from C51 was the inability to fix the address of a data object at an absolute address from the source file. Whilst there have always been methods of achieving the same effect, users have long requested an extension to allow the address of an object to be included in the original declaration. In C51 v4.xx, the new_AT_control now exists.

8.8 Using The _at_and _ORDER_ Controls

Here, the order of the variables must not change as it must match the physical location of the real time clock's registers. The #pragma ORDER tells C51 to place the data objects at ascending addresses, with the first item at the lowest address. The linker must then be used to fix the address of the whole block in memory.

Source File MAIN.C

```
#pragma ORDER
unsigned char xdata RTC_secs ;
unsigned char xdata RTC_mins ;
unsigned char xdata RTC_hours ;

main() {   RTC_mins = 1 ; }
```

Linker Input File MAIN.LIN

```
main.obj & to main & XDATA(?XD?MAIN(0fa00h))
```

The alternative _at_ control forces C51 to put data objects at an address given in the source file:

```
/** Fix Real Time Clock Registers Over Memory-Mapped Device **/
/** Fix each item individually **/
unsigned char xdata RTC_secs _at_ 0xfa00 ;
unsigned char xdata RTC_mins _at_ 0xfa01 ;
unsigned char xdata RTC_hours _at_ 0xfa02 ;

main() {   RTC_mins = 1 ;
  }
```

... which hopefully is self-explanatory!

9 Linking Issues And Stack Placement

This causes some confusion, especially to those used to other compiler systems.

9.1 Basic Use Of L51 Linker

The various modules of a C program are combined by a linker. After compilation no actual addresses are assigned to each line of code produced, only an offset is generated from the start of the module. Obviously before the code can be executed each module must be tied to a unique address in the code memory. This is done by the linker.

L51, in the case of Keil (RL51 for Intel), is a utility which assigns absolute addresses to the compiled code. It also searches library files for the actual code for any standard functions used in the C program.

A typical invocation of the linker might be:

l51 startup.obj, module1.obj, module2.obj, module3.obj, C51L.lib to exec.abs

Here the three unlocated modules and the startup code (in assembler) are combined. Any calls to library functions in any of these files results in the library, C51L.lib, being searched for the appropriate code.

The target addresses (or offsets) for any JMPs or CALLs are calculated and inserted after the relevant opcodes.

When all five .obj files have been combined, they are placed into another file called EXEC.ABS, the ABS implying that this is absolute code that could actually be executed by an 8051 cpu. In addition, a "map" file called EXEC.M51 is produced which summarises the linking operation. This gives the address of every symbol used in the program plus the size of each module.

In anything other than a very small program, the number of modules to be linked can be quite large, hence the command line can become huge and unwieldy. To overcome this the input list can be a simple ASCII text file thus:

```
   l51 @<input_file>

where input_file = ;

   startup.obj,&
   module1.obj,&
   module2.obj,&
   module3.obj,&
   &
   C51L.lib &
   &
   to exec.abs
```

There are controls provided in the linker which determine where the various memory types should be placed.

For instance, if an external RAM chip starts at 4000H and the code memory (Eprom) is at 8000H, the linker must be given:

```
l51 startup.obj, module1.obj, module2.obj, module3.obj, C51L.lib to exec.abs CODE(8000H)
XDATA(4000H)
```

This will move all the variables in external RAM to 4000H and above and all the executable code to 8000H. Even more control can be exercised over where the linker places code and data segments. By further specifying the module and segment names, specific variables can be directed to particular addresses - see 2.1.8 for an example.

9.2 Stack Placement

Unless you specify otherwise, the linker will place the stack pointer to give maximum stack space. Thus after locating all the sfr, compiled stack and data items, the real stack pointer is set to the next available IDATA address. If you use the 8032 or other variant with 128 bytes of indirectly-addressable memory (IDATA) above 80H, this can be used very effectively for stack.

```
?C_C51STARTUP  SEGMENT   CODE ;Declare segment in indirect
                                            area
?STACK         SEGMENT   IDATA ;

               RSEG    ?STACK          ; Reserve one byte
               DS      1
               EXTRN CODE (?C_START)
               PUBLIC  ?C_STARTUP
               CSEG    AT      0
?C_STARTUP:    LJMP    STARTUP1

               RSEG    ?C_C51STARTUP
STARTUP1: ENDIF
               MOV     SP,#?STACK-1 ; Put address of STACK
                                            location into SP
               LJMP    ?C_START     ; Goto initialised data
                                            section
```

9.3 Using The Top 128 Bytes of the 8052 RAM

The original 8051 design has just 128 bytes of directly/indirectly addressable RAM. C51, when in the SMALL model, can use this for variables, arrays, structures and stack. Above 128 (80H) direct addressing will result in access of the sfrs. Indirect addressing (MOV A,@R0) does not work.

However with the 8052 and above, the area above 80H can, when indirectly addressed, be used as additional storage. The main use of this area is really as stack. Data in this area is addressed by the MOV A,@Ri instruction. As only indirect addressing can be used, there can be some loss of efficiency as the Ri register must be loaded with the required 8 bit address before any access can be made.

Left to its own devices C51 will not use this area other than for stack. Unusually, the 8051 stack grows up through RAM, so the linker will place the STACK area at the top of the area taken up with variables, parameter passing segments etc.. If your application does not need all the stack area allocated, it is possible to use it for variables. This is simply achieved by declaring some variables as “idata” and using “RAMSIZE(256)” when linking.

Such is human nature that most people will not think of using idata until the lower 128 bytes actually overflows and a panic-driven search begins for more memory!

As has been pointed out, idata variables are rather harder to get at because of the loading of an Ri register first. However there is one type of variable which is ideally suited to this - the array or pointer-addressed variable.

The MOV A,@Ri is ideal for array access as the Ri simply contains the array index. Similarly a variable accessed by a pointer is catered for, as the @Ri is effectively a pointer. This is especially significant now that version 3.xx supports memory space specific pointers. The STACK is now simply moved above these new idata objects.

To summarise, with the 8052 if you are hitting the 128 byte ceiling of the directly addressable space, the moving of arrays and pointer addressable objects can free-up large amounts of valuable directly addressable RAM very easily.

9.4 L51 Linker Data RAM Overlaying

9.4.1 Overlaying Principles

One of the main tricks used to allow large programs to be generated within an 8051 is the OVERLAY function. This is a mechanism whereby different program variables make use of the same RAM location(s). This possibility arises when automatic local variables are declared. These by definition only have significance during the execution of the function within which they were defined. Once the function is exited the area of RAM used by them is no longer required. Of course static locals must remain intact until the function is next called. A similar situation exists for C51's reserved memory areas used for parameter passing.

Taken over a complete program, each function will have a certain area of memory reserved for its locals and parameters. Within the confines of an 8051 the on-chip RAM would soon be exhausted.

The possibility then arises for these individual areas to be combined into a single block, capable of supplying enough RAM for the needs of the single biggest function.

In C51 this process is performed by the linker's OVERLAY function. In simple terms, this examines all functions and generates a special data segment called "DATA_GROUP", able to contain all the local variables and parameters of all C51 functions. As an example, if most functions require only 4 byes of local data but one particular one needs 10, the DATA_GROUP will be 10 bytes long.

Using the registers as a location for temporary data means that a large number of locals and parameters can be accommodated without recourse to the DATA_GROUP - this is why it may appear smaller than you expect.

The overlayer works on the basis that if function 1 calls function 2, then their respective local data areas may not be overlaid, as both must be active at the same time. A third function 3, which is also called by 1, may have its locals overlaid with 2, as the two cannot be running at the same time.

```
main
 |
funcA – func2 - func3 - func4
 |
funcB – func5 - func6 - func7
 |
funcC – func8 - func9 - func10
 |
```

As funcA refers to func2 and func2 refers to func3 etc., A,2,3 and 4 cannot have their locals overlaid, as they all form part of the same path. Likewise, as funcB refers to func5 and func6 refers to func7 etc., B,6,7 and 4 cannot have their locals overlaid. However the groups 2,3,4; 5,6,7 and 8,9,10 may have their locals overlaid as they are never active together, being attached to sequential branches of the main program flow. This is the basis of the overlay strategy.

However a complication arises with interrupt functions. Since these can occur at any time, they would overwrite the local data currently generated by whichever background (or lower priority interrupt) function was running, were they also to use the DATA_GROUP. To cope with this, C51 identifies the interrupt functions and called functions and allocates them individual local data areas.

9.4.2 Impact Of Overlaying On Program Construction

The general rule used by L51 is that any two functions which cannot be executing simultaneously may have their local data overlaid. Re-entrant functions are an extension of this in that a single function may be called simultaneously from two different places.

In 99% of cases the overlay function works perfectly but there are some cases where it can give unexpected results.

These are basically:

(i) Indirectly-called functions using function pointers
(ii) Functions called from jump tables of functions
(iii) Re-entrant functions (-incorrect or non-declaration thereof)

Under these conditions the linker issues the following warnings:

```
MULTIPLE CALL TO SEGMENT

UNCALLED SEGMENT

RECURSIVE CALL TO SEGMENT
```

9.4.3 Indirect Function Calls With Function Pointers (hazardous)

Taking (i) first:

Here func4 and func5 are called from main by an intermediate function called EXECUTE. A pointer to the required function is passed. When L51 analyses the program, it cannot establish a direct link between execute and func4/5 because the function pointer received as a parameter breaks the chain of references - this function pointer is undefined at link time. Thus L51 overlays the local segments of func4, func5 and execute as if they were all references from main. Refer to the overlay diagram above if in doubt.

The result is that the locals of func4/5 will corrupt the locals used in execute. This is clearly VERY dangerous, especially as the overwriting may not be immediately obvious – it may only appear under abnormal operating conditions once the code has been delivered.

```
#include <reg517.h>
/**********************************************************
  ***   OVERLAY HAZARD 1 - Indirectly called functions ***
***********************************************************/
char func1(void) {    // Function to be called directly

char x, y, arr[10] ;

for(x = 0 ; x < 10 ; x++) {
   arr[x] = x ;
   }

return(x) ;
}

char func2(void) {   // Function to be called directly
(.... C Code ...)
}

char func3(void) {   // Function to be called directly
(.... C Code ...)
return(x) ;
}

char func4(void) {   // Function to be called indirectly


char x4, y4, arr4[10] ;      // Local variables

for(x4 = 0 ; x4 < 10 ; x4++) {

   arr4[x4] = x4 ;
   }

return(x4) ;
}

char func5(void) {   // Function to be called indirectly

char x5, y5, arr5[10] ;      // Local variables

for(x5 = 0 ; x5 < 10 ; x5++) {

   arr5[x5] = x5 ;
   }

return(x5) ;
}

/*** Function which does the calling ***/

char execute(fptr)  //Receive pointer to function to be used
   char (*fptr)() ;
   {

   char tex ;       // Local variables for execute function
   char arrex[10] ;

   for(tex = 0 ; tex < 10 ; tex++) {
      arrex[tex] = (*fptr)() ;
      }

   return(tex) ;
   }

/*** Declaration of general function pointer ***/

char (code *fp[3])(void) ;

/*** Main Calling Function ***/

void main(void) {

   char am ;

   fp[0] = func1 ;     // Point array elements at functions
   fp[1] = func2 ;
   fp[2] = func3 ;

   am = fp[0] ;        // Execute functions
   am = fp[1] ;
   am = fp[2] ;

   if(P1) {            // Control which function is called



      am = execute(func4) ; // Tell execute function which
                                      to run
      }
   else {

      am = execute(func5) ; // Tell execute function which
                                      to run
      }
   }
```

Resulting Linker Output .M51 File for the dangerous condition.

```
MS-DOS MCS-51 LINKER / LOCATER  L51 V2.8, INVOKED BY: L51 MAIN.OBJ TO EXEC.ABS

OVERLAY MAP OF MODULE:   EXEC.ABS (MAIN)

SEGMENT                          DATA-GROUP
  +—> CALLED SEGMENT           START    LENGTH

?C_C51STARTUP
  +—> ?PR?MAIN?MAIN

?PR?MAIN?MAIN                   000EH    0001H
  +—> ?PR?FUNC1?MAIN
  +—> ?PR?FUNC2?MAIN
  +—> ?PR?FUNC3?MAIN
  +—> ?PR?FUNC4?MAIN
  +—> ?PR?_EXECUTE?MAIN
  +—> ?PR?FUNC5?MAIN

?PR?FUNC1?MAIN                  000FH    000BH

?PR?FUNC2?MAIN                  000FH    000BH

?PR?FUNC3?MAIN                  000FH    000BH        //Danger func4's
                                                                //local
?PR?FUNC4?MAIN                  000FH    000BH    //func4's data
                                                                              //overlaid
with
?PR?_EXECUTE?MAIN               000FH    000EH    //execute's, its
  +—> ?C_LIB_CODE                                               //caller!!

?PR?FUNC5?MAIN                  000FH    000BH    //func5's local
                                                                //data overlaid
                                                                //with execute's,
                                                                  //its caller!!

RAM Locations Used:

D:0012H         SYMBOL        tex      // execute's locals overlap
D:0013H         SYMBOL        arrex    // func4 and func5's - OK

D:000FH         SYMBOL        y
D:0010H         SYMBOL        arr4

D:000FH         SYMBOL        y5
D:0010H         SYMBOL        arr5
```

Incidentally, the overlay map shows which functions referred to which other functions. By checking what L51 has found against what you expect, overlay hazards may be spotted.

9.4.4 Indirectly called functions solution

Use the overlay command when linking thus:

```
main.obj & to exec.abs & OVERLAY(main ; (func4,func5), _execute ! (func4,func5))
```

Note: The tilde sign ';' means: "Ignore the reference to func4/5 from main" The '!' means: "Manually generate a reference between intermediate function 'execute' and func4/5 to prevent overlaying of local variables within these functions."

Please make sure you understand exactly how this works!!!

The new linker output is:

```
MS-DOS MCS-51 LINKER / LOCATER  L51 V2.8, INVOKED BY:

L51 MAIN.OBJ TO EXEC.ABS OVERLAY(MAIN ;(FUNC4, FUNC5), _EXECUTE ! (FUNC4, FUNC5))
OVERLAY MAP OF MODULE:   EXEC.ABS (MAIN)

SEGMENT                          DATA-GROUP
 +—> CALLED SEGMENT    START       LENGTH

?C_C51STARTUP
  +—> ?PR?MAIN?MAIN

?PR?MAIN?MAIN                     0024H    0001H
  +—> ?PR?FUNC1?MAIN
  +—> ?PR?FUNC2?MAIN
  +—> ?PR?FUNC3?MAIN
  +—> ?PR?_EXECUTE?MAIN

?PR?FUNC1?MAIN                   0025H    000BH

?PR?FUNC2?MAIN                   0025H    000BH

?PR?FUNC3?MAIN                   0025H    000BH

?PR?_EXECUTE?MAIN                0025H    000EH
  +—> ?C_LIB_CODE

D:0028H         SYMBOL        tex    // Execute's variables
                                                no longer
D:0029H         SYMBOL        arrex  // overlaid with func4/
                                                5's
```

```
D:0008H         SYMBOL        y
D:0009H         SYMBOL        arr4

D:0013H         SYMBOL        y5
D:0014H         SYMBOL        arr5

*** WARNING 16: UNCALLED SEGMENT,IGNORED FOR OVERLAY        PROCESS
    SEGMENT: ?PR?FUNC4?MAIN

*** WARNING 16: UNCALLED SEGMENT,IGNORED FOR OVERLAY PROCESS
    SEGMENT: ?PR?FUNC5?MAIN
```

Note: The WARNING 16's show that func4/5 have been removed from the overlay process to remove the hazard. See section 8.4.2.6 on the "UNCALLED SEGMENT, IGNORED FOR OVERLAY PROCESS" warning.

9.4.5 Function Jump Table Warning (Non-hazardous)

Here two functions are called an array of function pointers. The array "jump_table" exists in a segment called "?CO?MAIN1, i.e. the constant area assigned to module main. The problem arises that the two message string arguments to the printf 's are also sited here. This leads to a recursive definition of the function start addresses in the jump table.

While this is not in itself dangerous, it prevents the real function references from being established and hence the overlaying process is inhibited.

```
*************************************************************************;
*<<<<<<<<<<<<<Recursive Call To Segment Error>>>>>>>>>>>>>>*
*************************************************************************;
#include <stdio.h>
#include <reg517.h>

void func1(void) {

   unsigned char i1 ;

   for(i1 = 0 ; i1 < 10 ; i1++) {

      printf("THIS IS FUNCTION 1\n") ;  // String stored in
                                                 ?CO?MAIN1 segment
      }
   }

void func2(void) {

   unsigned char i2 ;

   for(i2 = 0 ; i2 < 10 ; i2++) {

      printf("THIS IS FUNCTION 2\n") ;  // String stored in
                                                 ?CO?MAIN1 segment
      }
   }

code void(*jump_table[])()={func1,func2}; //Jump table to
                                                        functions,
                                          // table stored in
                                                   ?CO?MAIN1
                                          // segment.
/*** Calling Function ***/


main() {
   (*jump_table[P1 & 0x01])() ;   // Call function via jump
                                          table in ?CO?MAIN1
    }
    ^^^^^^^^^^^^^^^^^^^^^^^ End of Module
```

The resulting link output is:

Note: No reference exists between main and func1/2 so the overlay process cannot occur, resulting in wasted RAM.

```
OVERLAY MAP OF MODULE:   MAIN1 (MAIN1)

SEGMENT                    BIT-GROUP          DATA-GROUP
  +—> CALLED SEGMENT   START    LENGTH     START    LENGTH

?C_C51STARTUP
  +—> ?PR?MAIN?MAIN1

?PR?MAIN?MAIN1
  +—> ?CO?MAIN1
  +—> ?C_LIB_CODE

?CO?MAIN1
  +—> ?PR?FUNC1?MAIN1
  +—> ?PR?FUNC2?MAIN1

?PR?FUNC1?MAIN1                                                     0008H    0001H
  +—> ?PR?PRINTF?PRINTF

MCS-51 LINKER / LOCATER  L51 V2.8
DATE  04/08/92   PAGE    2

?PR?PRINTF?PRINTF      0020H.0  0001H.1    0009H    0014H
  +—> ?C_LIB_CODE
  +—> ?PR?PUTCHAR?PUTCHAR

?PR?FUNC2?MAIN1                                                     0008H    0001H
  +—> ?PR?PRINTF?PRINTF

*** WARNING 13: RECURSIVE CALL TO SEGMENT
    SEGMENT: ?CO?MAIN1
    CALLER:  ?PR?FUNC1?MAIN1

*** WARNING 13: RECURSIVE CALL TO SEGMENT
    SEGMENT: ?CO?MAIN1
    CALLER:  ?PR?FUNC2?MAIN1
```

9.4.6 Function Jump Table Warning Solution

The solution is to use the OVERLAY command when linking thus:

```
main1.obj &
to main1.abs &
OVERLAY(?CO?MAIN1 ~ (func1,func2), main ! (func1,func2))
```

This deletes the reference to func1 & 2 from the ?CO?MAIN1 segment and inserts the true reference from main to func1 & func2.

The linker output is now thus:

```
OVERLAY MAP OF MODULE:   MAIN1.ABS (MAIN1)

SEGMENT                   BIT-GROUP          DATA-GROUP
  +—> CALLED SEGMENT    START    LENGTH     START    LENGTH
```

```
?C_C51STARTUP
  +—> ?PR?MAIN?MAIN1

?PR?MAIN?MAIN1
  +—> ?CO?MAIN1
  +—> ?C_LIB_CODE
  +—> ?PR?FUNC1?MAIN1
  +—> ?PR?FUNC2?MAIN1

?PR?FUNC1?MAIN1                                             0008H    0001H
  +—> ?CO?MAIN1
  +—> ?PR?PRINTF?PRINTF

?PR?PRINTF?PRINTF     0020H.0  0001H.1    0009H    0014H
  +—> ?C_LIB_CODE
  +—> ?PR?PUTCHAR?PUTCHAR

?PR?FUNC2?MAIN1                                0008H    0001H
  +—> ?CO?MAIN1
  +—> ?PR?PRINTF?PRINTF
```

9.4.7 Multiple Call To Segment Warning (Hazardous)

This warning generally occurs when a function is called from both the background and an interrupt. This means that potentially the interrupt may call the function whilst it is still running, as a result of a background level call. The result could be the over-writing of the local data in the background. The fact that the offending function is also overlaid with other background functions makes the chances of failure very high. The simplest solution is to declare the function as REENTRANT so that the compiler will generate a local stack for parameters and variables. Thus on each call to the function, a new set of parameters and local variables are created without destroying any existing ones from the current call.

Unfortunately this significantly increases the run time and code produced. Another possibility is to make a second and renamed version of the function, one for background use and one for interrupt. This is somewhat wasteful and presents a maintenance problem, as you now have effectively two versions of the same piece of code.

In many cases the situation is not a problem, as the user may have ensured that the reentrant use could never occur, but is left with the linker warning. However this must be viewed as dangerous, particularly if more than one programmer is involved.

```
#include <stdio.h>
#include <reg517.h>

void func1(void) {

   unsigned char i1,a1[15] ;

   for(i1 = 0 ; i1 < 10 ; i1++) {

      a1[i1] = i1 ;
      }
   }

void func2(void) {

   unsigned char i2,a2[15] ;

   for(i2 = 0 ; i2 < 10 ; i2++) {
      a2[15] = i2 ;
      }
   }

main() {
   func1() ;
   func2() ;
   }

void timer0_int(void) interrupt 1 {
   func1() ;
   } ^^^^^^^^^^^^^^^^^^^^^^^ End of Module

This produces the linker map:

OVERLAY MAP OF MODULE:   MAIN2 (MAIN2)
SEGMENT                          DATA-GROUP
  +—> CALLED SEGMENT           START    LENGTH

?PR?TIMER0_INT?MAIN2
  +—> ?PR?FUNC1?MAIN2

?PR?FUNC1?MAIN2                0017H    000FH

?C_C51STARTUP
  +—> ?PR?MAIN?MAIN2

?PR?MAIN?MAIN2
  +—> ?PR?FUNC1?MAIN2
  +—> ?PR?FUNC2?MAIN2

?PR?FUNC2?MAIN2                0017H    000FH

D:0007H         SYMBOL         i1  // Danger!
D:0017H         SYMBOL         a1

D:0007H         SYMBOL         i2
D:0017H         SYMBOL         a2

*** WARNING 15: MULTIPLE CALL TO SEGMENT
    SEGMENT: ?PR?FUNC1?MAIN2
    CALLER1: ?PR?TIMER0_INT?MAIN2
    CALLER2: ?C_C51STARTUP
```

9.4.8 Multiple Call To Segment Solution

The solution is to:

(i)   Declare func1 as REENTRANT thus:

```
void func1(void) reentrant {  }
```

(ii) Use OVERLAY linker option thus:

```
main2.obj &
to main2.abs &
OVERLAY(main ~ func1,timer0_int ~ func1)
```

to break connection between main and func1 and timer0_int and func1.

OVERLAY MAP OF MODULE: MAIN2.ABS (MAIN2)

```
SEGMENT                          DATA-GROUP
  +—> CALLED SEGMENT           START    LENGTH
```

```
?C_C51STARTUP
  +—> ?PR?MAIN?MAIN2

?PR?MAIN?MAIN2
  +—> ?PR?FUNC2?MAIN2

?PR?FUNC2?MAIN2              0017H    000FH

*** WARNING 16: UNCALLED SEGMENT, IGNORED FOR OVERLAY PROCESS
    SEGMENT: ?PR?FUNC1?MAIN2
```

This means that the safe overlaying of func1 with other background functions will not occur. Removing the link only with the interrupt would solve this:

```
main2.obj &
to main2.abs &
OVERLAY(timer0_int ~ func1)
```

Another route would be to disable all overlaying but this is likely to eat up large amounts of RAM very quickly and is thus a poor solution.

```
main2.obj & to main2.abs & NOOVERLAY
```

With the MULTIPLE CALL TO SEGMENT WARNING the only really “safe” solution is to declare func1 as REENTRANT, with the duplicate function a good second. The danger of using the OVERLAY command is that a less experienced programmer new to the system might not realise that the interrupt is restricted as to when it can call the function and hence system quality is degraded.

9.4.9 Overlaying Public Variables

All the preceding examples deal with the overlaying of locals and parameters at a function level. A case occurred recently in which the program was split into two distinct halves; the divide taking place very early on. To all intents and purposes the 8051 was able to run one of two completely different application programs, based on some user input during initialisation.

Each program half had a large number of public variables, some of which were known to both sides but the majority of which were local to one side only. This is almost multitasking.

This type of program structure really needs a new storage class like “GLOBAL”, with public meaning available to a certain number of modules only. GLOBAL would then be available to all modules. The new C166 supports this type of task-based variable scope. Unfortunately C51 does not, so a fix is required.

The linker’s OVERLAY command does not help, as it only controls the overlaying of local and parameter data. One possible solution uses special modules to declare the publics. Module1 declares the publics for program (task1); Module2 declares the publics for program2 (task2). Finally, Module3 declares the publics which are available to both sides.

The trick then is to use the linker to fix the data segments on Module1 and Module2 at the same physical address, whilst allowing Module3’s variables to be placed automatically by the linker.

This solution uses three special modules for declaring the publics:

```
/* Example of creating two sets of public data */
/*in same memory space */

extern void main1(void) ;
extern void main0(void) ;

/* Main module where system splits into two parts */

void main(void) {
   bit flag ;

   if(flag) {
      main0() ;   // Branch 0
      }
   else {
      main1() ;   // Branch 1
      }
   } ^^^^^^^^^^^^^^^^^^^^^^^ End of Module

/* Module that declares publics for branch 2 */

/* Publics for branch 2 */

unsigned char x2,y2 ;
unsigned int z2 ;
char a2[0x30] ;

/* A variable which is accessible from both branches */

extern int common ;

^^^^^^^^^^^^^^^^^^^^^^^ End of Module


void main0(void) {

   unsigned char c0 ; /* Local - gets overlaid with c1 in*/
                              /*other branch */
   x2 = 0x80 ;
   y2 = x2 ;

   c0 = y2 ;

   z2 = x2*y2 ;

   a2[2] = x2 ;

   common = z2 ;

   }

^^^^^^^^^^^^^^^^^^^^^^^ End of Module



/* Module that declares publics for branch 1 */

/* Publics for branch 1 */

unsigned char x1,y1 ;
unsigned int z1 ;
char a1[0x30] ;

/* A variable which is accessible from both branches */

extern int common ;

void main1(void) {

   char c1 ;
   x1 = 0x80 ;
   y1 = x1 ;

   c1 = y1 ;

   z1 = x1*y1 ;
   a1[2] = x1 ;

   common = z1 ;

   }
^^^^^^^^^^^^^^^^^^^^^^^ End of Module

/* Module that declares variables that both */
/*branches can access */

int common ; /* A variable common to both branches */

^^^^^^^^^^^^^^^^^^^^^^^ End of Module

/* Linker Input */

l51 t.obj,t1.obj,t2.obj,com.obj to t.abs
DATA(?DT?T1(20H),?DT?T2(20H))
```

The choice of “20H” for the location places the combined segments just above the register banks.

The main problem with this approach is that a DATA overlay warning is produced. This is not dangerous but is obviously undesirable.

10 Other C51 Extensions

10.1 Special Function Bits

A frustration for assembler programmers with the old C51 version was the need to use bit masks when testing for specific bits with chars and ints, despite there being a good set of bit-orientated assembler instructions within the 8051. In version 3, however, it is possible to force data into the bit-addressable area (starting at 0x20) where the 8051's bit instructions can be used.

An example is testing the sign of a char by checking for bit = 1.

Here the char is declared as “bdata” thus:

```
   bdata char test_char ;

sign_bit is defined as:

   sbit sign_bit = test_char ^ 7 ;
```

to use this:

```
     test_char = counter ;
     if(sign_bit) { /* test_char is negative */ }
```

the opcodes executed are:

```
     MOV   A,counter    ;
     MOV   test_char,A  ;
     JNB   0,DONE       ;
     /* Negative */
DONE:
```

All of which is a lot faster than using bit masks and &'s!

The important points are that the “bdata” tells C51 and L51 that this variable is to be placed in the bit-addressable RAM area and the “sbit sign_bit = test_char ^ 7” tells C51 to assume that a bit called sign_bit will be located at position 7 in the test_char byte.

```
Byte Number: test_char           20H    Start Of BDATA area
Bit Number:  0,1,2,3,4,5,6,7<— sign_bit
Byte Number:                     21H
Bit Number:  8,9,10,11,12,13,14,15
Byte Number:                     22H
Bit Number:  16,17,18,19,20,21,22,23,24.....
```

The situation with ints is somewhat more complicated. The problem is that the 8051 does not store things as you first expect. The same sign test for an int would require bit 7 to be tested. This is because the 8051 stores int's high byte at the lower address. Thus bit 7 is the highest bit of the higher byte and 15 is the highest bit of the lower.

```
Byte Number: test_int(high)          20H
Bit Number:  0,1,2,3,4,5,6,7

Byte Number: test_int+1(low)         21H
Bit Number:  8,9,10,11,12,13,14,15

Bit locations in an integer
```

10.2 Support For 80C517/537 32-bit Maths Unit

The Siemens 80C537 and 80C517A group have a special hardware maths unit, the MDU, aimed at speeding-up number-crunching applications.

10.2.1 The MDU - How To Use It

To allow the 8051 to cope with 16 and 32-bit ("int" and "long") multiplication and division, the Siemens 80C517 variant has a special maths co-processor (MDU) integrated on the cpu silicon. A 32-bit normalise and shift is also included for floating point number support. It also has 8 data pointers to make accessing external RAM more efficient.

The compiler can take advantage of these enhancements if the "**MOD517**" switch is used, either as a #pragma or as a command line extension. This will result in the use of the MDU to perform > 8 bit multiplies and divides. However a special set of runtime libraries is required from Keil for linking.

Using the MDU will typically yield a runtime improvement of 6 to 9 times the basic 8051 cpu for 32 bit unsigned integer arithmetic.

Optionally the blanket use of the 80C517 enhancements after **MOD517** can be selectively disabled by the **NOMDU** and **NODP** pragmas. Predictably **NOMDU** will inhibit the use of the maths unit, while **NODP** will stop the eight data pointers being used.

10.2.2 The 8 Datapointers

To speed up block data moves between external addresses, the 517A has 8 datapointers. These are only used by C51 in the memory and stering manipulation library functions like memcpy() and strcpy() for example.

The general "MOD517" switch will enable their use. Note that the strcat() routine does not use the additional data pointers.

If the extra pointers are to be used both in background and interrupt functions, the DPSEL register is automatically stacked on entry to the interrupt and a new DPSEL value allocated for the duration of the function.

10.2.3 80C517 - Things To Be Aware Of

The 80C517 MDU is used effectively like a hardware subroutine, as it is not actually part of the 8051 cpu. As such it is subject to normal sub-routine rules regarding re-entrancy. If, as an example, both a background program and an interrupt routine try to use the MDU simultaneously, the background calculation will be corrupted. This is because the MDU input and output registers are fixed locations and the interrupt will simply overwrite the background values.

To allow the background user to detect corruption of the MDU registers, the MDEF bit is provided within the ARCON register. After any background use of the MDU, a check should be made for this flag being set. If so, the calculation must be repeated. Appropriate use of the NOMDU pragma could be used instead.

Note: the compiler does not do this - the user must add the following code to overcome the problem:

```
#pragma MOD517
#include "reg517.h"

    long x,y,z ;
    func()
      {
      while(1)
         {
           x = y / z ;      /* 32-bit calculation */
           if(MDEF == 0)    /* If corruption has */
              { break ; }   /* occurred then repeat */
         }                  /* else exit loop */
      }
```

10.3 87C751 Support

The Philips 87C751 differs from the normal 8051 CPU by having a 2k code space with no option for external ROM. This renders the long LJMP and LCALL instructions redundant. To cope with this the compiler must be forced to not generate long branch instructions but to use AJMPs and ACALLs instead.

10.3.1 87C751 - Steps To Take

1. Invoke C51 with:
   C51 myfile.c ROM(SMALL) NOINTVECTOR or use "#pragma ROM(SMALL)"

2 Use the INIT751.A51 startup file in the LIB directory.

3. Do not use floating point arithmetic, integer or long divides, printf, scanf etc., as they all use LCALLs.

4. A special 87C751 library package is available which will contain short call versions of the standard library
   routines.

10.3.2 Integer Promotion

Automatic integer promotion within "IF" statements is incorporated in version >= 3.40 to meet recent ANSI stipulations in this area. This makes porting code from Microsoft or Borland PC C compilers much easier. Thus any char(s) within a conditional statement are pre-cast to int before the compare is performed. This makes some sense on 16 bit machines where int is as efficient as char but, in the 8051, char is the "natural" size for data and so some loss of efficiency results.

Fortunately Keil have provided "#pragma NOINTPROMOTE" to disable this feature! In this case explicit casts should be used if another data type might result from an operation.

To show why this #pragma is important, this C fragment's code sizes are influenced thus:

```
char c ; unsigned char c1, c2 ; int i ;
main() {
   if((char)c == 0xff) c = 0 ;
   if((char)c == -1) c = 1 ;
   i = (char)c + 5 ;

   if((char)c1 < (char)c2 + 4) c1 = 0 ;

   }
```

Code Sizes

```
47 bytes - C51 v3.20
49 bytes - C51 v3.40 (INTPROMOTE)
63 bytes - C51 v3.40 (NOINTPROMOTE)
```

Again this goes to show that C portability compromises efficiency in 8051 programs...

11 Miscellaneous Points

11.1 Tying The C Program To The Restart Vector

This is achieved by the assembler file “STARTUP.A51”. This program simply places a LJMP STARTUP at location C:0000 (Lowest EPROM location).

The startup routine just clears the internal RAM and sets up the stack pointer. Initialisation routines might also take place between LJMP MAIN and Main() if global variables that are created and initialised in your application, for example… char x= 10; in this case startup .a51 is going to call init.a51 and then it is finally going to call main();.

```
        LJMP main
               .
               .
               .
               .
               main()
               {
               }
```

In fact this need be the only assembler present in a C51 program.

11.2 Intrinsic Functions

There are a number of special 8051 assembler instructions which are not normally used by C51. For the sake of speed it is sometimes useful to get direct access to these.

Unlike the normal C51 ‘>>’ functions, _cror_ allows direct usage of an 8051 instruction set feature, in this case the “RR   A” (rotate accumulator). This yields a much faster result than would be obtained by writing one using bits and the normal >> operator. There are also _iror_ and _lror_ intrinsic functions for integer and long data as well.

The _nop_ function simply adds an in-line NOP instruction to generate a short and predictable time delay. Another function, _testbit_, makes use of the JBC instruction to allow a bit to be tested, a branch taken and the bit cleared if set. The only extra step necessary is to include “intrins.h” in the C51 source file.

Here is an example of how the _testbit_() intrinsic function is used to save a CLR instruction:

```
; #include <intrins.h>
;
;
; unsigned int shift_reg = 0 ;
;
; bit test_flag ;
;
; void main(void) {
        RSEG  ?PR?main?T
        USING   0
main:
```

```
                       ; SOURCE LINE # 12
;
; /* Use Normal Approach */
;
;    test_flag = 1 ;
                       ; SOURCE LINE # 14
       SETB   test_flag
;
;    if(test_flag == 1) {
                       ; SOURCE LINE # 16
       JNB    test_flag,?C0001
;       test_flag = 0 ;
                       ; SOURCE LINE # 17
       CLR    test_flag
;       P1 = 0xff     ;
                       ; SOURCE LINE # 18
       MOV    P1,#0FFH
;       }
                       ; SOURCE LINE # 19
?C0001:
;
; /* Use Intrinsic Function */
;
;    test_flag = 1 ;
                       ; SOURCE LINE # 21
       SETB   test_flag
;
;   if(!_testbit_(test_flag)) {
                       ; SOURCE LINE # 23
       JBC    test_flag,?C0003
;       P1 = 0xff     ;
                       ; SOURCE LINE # 24
       MOV    P1,#0FFH
;       }
                       ; SOURCE LINE # 25
;
;    }
                       ; SOURCE LINE # 27
?C0003:
       RET
; END OF main
       END
```

See pages 9-17 in the C51 Manual

11.3 EA Bit Control #pragma

Whilst the interrupt modifier for function declarations remains unchanged a new directive, DISABLE, allows interrupts to be disabled for the duration of a function. Note that this can be individually applied to separate functions within a module but is given as a #pragma rather than as part of the function declaration. Although not verified yet, DISABLE gives the user some control over the EA or EAL bit.

11.4 16-Bit sfr Support

Another new feature is the 16bit sfr type. Within expanded 8051 variants in particular, many 16 bit timer and capture registers exist. Rather than having to load the upper and lower bytes individually with separate C statements, the sfr16 type is provided. The actual address declared for a 16 bit sfr in the header file is always the low byte of the sfr. Now to load a 16 bit sfr from C, only a single int load is required. Be warned - 8-bit instructions are still used, so the 16 bit load/read is not indivisible - odd things can happen if you load a timer and it overflows during the process! Note that usually only timer 2 or above has the high/low bytes arranged sequentially.

11.5 Function Level Optimisation

Optimisation levels of 4 and above are essentially function optimisations and, as such, the whole function must be held in PC memory for processing. If there is insufficient memory for this, a warning is issued and the additional optimisation abandoned. Code execution will still be correct however. See p1-8 in the C51 manual.

11.6 In-Line Functions In C51

One of the fundamentals of C is that code with a well-defined input, output and job is placed into a function i.e. a subroutine. This involves placing parameters into a passing area, whether a stack or a register, and then executing a CALL. It is unavoidable that the call instruction will use two bytes of stack.

In most 8051 applications this not a problem, as there is generally 256 on-chip RAM potentially available as stack. Even after allowing for a few registerbanks, there is normally sufficient stack space for deeply nested functions.

However in the case of the 8031 and reduced devices such as the 87C751, every byte of RAM is critical. In the latter case there are only 64 bytes!

A trick which can both save stack and reduce run time is to use macros with parameters to act like “in-line” functions. The ability to create macros with replaceable parameters is not commonly used but on limited RAM variants it can be very useful.

Here a strcpy() function created as a macro named “Inline_Strcpy”, whilst it looks like a normal function, it does not actually have any fixed addresses or local data of its own. The ‘\’ characters serve to allow the macro definition to continue to a new line, in this case to preserve the function-like appearance.

It is “called” like a normal function with the parameters to be passed enclosed in ( ). However no CALL is used and the necessary code is created in-line. The end result is that a strcpy is performed but no new RAM or stack is required.

Please note however, the drawback with this very simple example is that the source and destination pointers are modified by the copying process and so is rather suspect!

A further benefit in this example is that the notional pointers s1 and s2 are automatically memory-specific and thus very efficient. Thus in situations where the same function must operate on pointer data in a variety of memory spaces, slow generic pointers are not required.

```
#define Inline_Strcpy(s1,s2)  {\ while((*s1++ = *s2++)  != NULL }\
                               }
char xdata *out_buffx = { “                           ” } ;
char xdata *in_buffx = { “Hello” } ;
char idata *in_buffi = { “Hello” } ;
char idata *out_buffi = { “                           ” }  ;  char code *in_buffc = {
“Hello” } ;

void main(void) {
   Inline_Strcpy(out_buffx,in_buffx)  // In line functions
   Inline_Strcpy(out_buffi,in_buffi)
   Inline_Strcpy(out_buffx,in_buffc)
   }
```

Another good example of how a macro with parameters can be used to aid source readability is in the optimisation feature in Appendix D. The interpolation calculation that originally formed a subroutine could easily be redefined as a macro with 5 parameters, realising a ram and run time saving at the expense of code size.

Note that ‘r’, the fifth parameter, represents the return value which has to be “passed” to the macro so that it has somewhere to put the result!

```
#define interp_sub(x,y,n,d,r)  y -= x ; \
if(!CY) { r = (unsigned char) (x +(unsigned char)(((unsigned
               int)(n * y))/d)) ;\

} else { r = (unsigned char) (x - (unsigned char)(((unsigned int)(n * -y))/d)) ; }
```

This is then called by:

```
/*Interpolate 2D Map Values */
/*Macro With Parameters Used*/

interp_sub(map_x1y1,map_x2y1,x_temp1,x_temp2,result_y1)

and later it is reused with different parameters thus:

interp_sub(map_x1y2,map_x2y2,x_temp1,x_temp2,result_y2)
```

To summarise, parameter macros are a good way of telling C51 about a generalised series of operations whose memory spaces or input values change in programs where speed or RAM usage is critical.

12 Some C51 Programming Tricks

12.1 Accessing R0 etc. directly from C51

A C51 user was using existing assembler routines to perform a specific task. For historical reasons the 8 bit return value from the assembler was left in R0 of register bank 3. Ordinarily C51 would return chars in R7 and therefore simply equating a variable to the assembler function call would not work.

The solution was to declare an uncommitted memory specific pointer to the DATA area. At run time the absolute address of the register (here 0x18) was assigned to the pointer. The return value was then picked up via the pointer after exiting the assembler section.

```
/*** Example Of Accessing Specific Registers In C ***/
char data *dptr ;  // Create pointer to DATA location

/* Define Address Of Register */

#define R0_bank3 0x40018L   /* Address of R0 in */
                                  /* bank 3, 4 => DATA space */
char x,y ;

/* Execute */

main() {
dptr = (char*) R0_bank3 ;  // Point at R0, bank3

x = 10 ;
dptr[0] = x ;   // Write x into R0, bank3
y = *dptr ;     // Get value of R0, bank3

}
```

An alternative might have been to declare a variable to hold the return value in a separate module and to use the linker to fix that module's DATA segment address at 0x18. This method is more robust and code efficient but is considerably less flexible.

12.2 Making Use Of Unused Interrupt Sources

One problem with the 8051 is the lack of a TRAP or software interrupt instruction. While C166 users have the luxury of real hardware support for such things, 8051 programmers have to be more cunning.

A situation arose recently where the highest priority interrupt function in a system had to run until a certain point, from which lesser interrupts could then come in. Unfortunately, changing the interrupt priority registers part way through the interrupt function did not work, the lesser interrupts simply waiting until the RETI. The solution was to hijack the unused A/D converter interrupt, IADC, and attach the second section of the interrupt function to it. Then by deliberately setting the IADC pending flag just before the closing "}", the second section could be made to run immediately afterwards. As the priority of the ADC interrupt had been set to a low level, it was interruptable.

```
/* Primary Interrupt Attached In CC0 Input Capture */
```

```
tdc_int() interrupt 8 {

/* High priority section - may not be interrupted */


/* Enable lower priority section attached to */
                                  /* ADC interrupt */

IADC = 1 ; // Force ADCinterrupt
EADC = 1 ; // Enable ADC interrupt
}

/* Lower priority section attached to ADC interrupt */

tdc_int_low_priority() interrupt 10

IADC = 0 ; // Prevent further calls
EADC = 0 ;

/* Low priority section which must be interruptable and */
    /* guaranteed to follow high priority section above */

}
```

In the decade or so since the first edition and some 500 new variants later there is now the CC1010 from Chipcon that has the TRAP instruction Op code #A5h for break points and debugging.

12.3 Code Memory Device Switching

This dodge was used during the development of a HEX file loader for a simple 8051 monitor. After receiving a hexfile into a RAM via the serial port, the new file was to be executed in RAM starting from 0000H. A complication was that the memory map had to be switched immediately prior to hitting 0000H.

The solution was to place the map switching section at 0xfffd so that the next instruction would be fetched from 0x0000, thus simulating a reset. Ideally all registers and flags should be cleared before this.

```
#include “reg.h”
#include “cemb537.h”
#include  <stdio.h>

   main()
      {

      unsigned char tx_char,rx_char,i ;

      P4 = map2 ;

      v24ini_537() ;

      timer0_init_537() ;

      hexload_ini() ;

      EAL = 1 ;

      while(download_completed == 0)
         {

         while(char_received_fl == 0)
            { receive_byte() ; }
         tx_byte = rx_byte ; /* Echo */
         hexload() ;
         send_byte(tx_byte) ;

         char_received_fl = 0 ;
         }

      real_time_count = 0 ;
      while(real_time_count < 200)
         { ; }

      i = ((unsigned char (code*)(void)) 0xFFFD) () ;
                                   // Jump to absolute address.

      }

^^^^^^^^^^^^^^^^^^^^^^^ End of Module



;
   NAME SWITCH
;
; Cause PC to roll-over at FFFFH to simulate reset
;
   P4      DATA 0E8H
;
   CSEG AT 0FFFDH
;
   MOV  P4,#02Fh  ;
;
   END

^^^^^^^^^^^^^^^^^^^^^^^ End of Module “MAPCON”
```

There are other ways of doing this. For instance the code for the MAPCON module could be located at link time thus: CODE(SWITCH(0FFFDH)), so dispensing with the “CSEG AT”.

12.4 Simulating A Software Reset

In a similar vein to the above, the 8051 does not possess a software reset instruction, unlike the 80C166 etc.. This method uses abstract pointers to create a call to address zero, thus simulating a software reset.

However it should be remembered that all internal locations must be cleared before the CPU can be considered properly reset! The return address must be reset as the stack still contains the return address from the call.

```
;
;
; void main(void) {

       RSEG  ?PR?main?T1
       USING   0
main:
                       ; SOURCE LINE # 9
;
; ((void (code*) (void)) 0x0000) () ;
                       ; SOURCE LINE # 11
       LCALL   00H       ; Jump to address ZERO!
;
; }
                       ; SOURCE LINE # 13
       RET
; END OF main
```

12.5 The Compiler Preprocessor - #define

This is really just a text replacement device.

It can be used to improve program readability by giving constants meaningful names, for example:

```
        #define FUEL_CONSTANT (100 * 2)
```

so that the statement temp = FUEL_CONSTANT will assign the value 200 to temp.

NOTE: the #define is purely a textual replacement and care must be taken to ensure no side effects. Unless the define is a single item eg

#define MAX_TEMP 10

The text should be   in perenthasis ( ) as in the case

```
        #define FUEL_CONSTANT (100 * 2)
```

Many safety critical and high integrity coding guides mandate that ALL defines shall use perenthasis. In the case of the example above without the parenthasis it could have many stange side effect….

temp = `FUEL_CONSTANT    ==   temp = 100* 2`

Is quite clear but how about :

```
temp = 10 +  FUEL_CONSTANT    temp = 10 + 100 *  2
```

In this case precedence gives   (200 *2) + 10 but then I am sure you knew that. However if it had been :

```
#define BASE  50
#define FUEL_CONSTANT 100 + BASE
```

then

```
temp = miles * FUEL_CONSTANT
```

The answer would have been   (miles * 100) + 50 and NOT miles * 150

13 C51 Library Functions

One of the main characteristics of C is its ability to allow complex functions to be constructed from the basic commands. To save programmer effort many common mathematical and string functions are supplied ready compiled in the form of library files.

13.1 Library Function Calling

Library functions are called as per user-defined functions, i.e.;

```
        #include ctype.h
        {
        char test_byte ;
        result = isdigit(test_byte) ;
        }
```

where "isdigit()" is a function that returns value 1 (true) if the test_byte is an ASCII character in the range 0 to 9.

The declarations of the library functions are held in files with a ".h" extension - see the above code fragment.
Examples are:

```
ctype.h,
stdio.h,
string.h etc..
```

These are included at the top of the module which uses a library function.

Many common mathematical functions are available such as ln, log, exp, 10x, sin, cos, tan (and the hyperbolic equivalents). These all operate on floating point numbers and should therefore be used sparingly! The include file containing the mathematical function prototypes is "math.h".

Library files contain many discrete functions, each of which can be used by a C program. They are actually retrieved by the linker utility covered in section 8. These files are treated as libraries by virtue of their structure rather than their extension. The insertion or removal of functions from such a file is performed by a library manager called LIB51.

13.2 Memory-Model Specific Libraries

Each of the possible memory models requires a different run-time library file. Obviously if the LARGE model is used the code required will be different for a SMALL model program.

Thus with C51, 6 different library files are provided:

```
C51S.LIB - SMALL model
C51C.LIB - COMPACT model
C51L.LIB - LARGE model
```

plus three additional files containing floating point routines as well as the integer variety.

C51 library functions are registerbank independent. This means that library functions can be used freely without regard to the current REGISTERBANK() or USING status. This is a major advantage as it means that library functions can be used freely within interrupt routines and background functions without regard to the current register bank.

14 Outputs From C51

14.1 Object Files

Being closely related to the original Intel tools, C51 defaults to the Intel object file format. This is a binary file containing the symbolic information necessary for debugging with in-circuit emulators etc.. It may be linked with object files from either Intel PLM51 or ASM51 using the Keil L51 linker. The final output is Intel OMF51.

Versions >2.3 of the compiler will produce an extended Intel OMF51 object file if the DEBUG OBJECTEXTEND command line switches are used. This passes type and scope information into the OMF51 file which any debugger/in-circuit emulator should be able to use. The extensions to the original Intel format are a proprietary Keil development but have been widely copied by IAR et al.

14.2 HEX Files For EPROM Blowing

To blow EPROMS an additional stage is usually necessary to get a HEX file. This is an ASCII representation of the final program without any symbol information. Almost every EPROM programmer will understand Intel HEX. The OH51/OHS51 utility performs the conversion from the linker's OMF51 file to the standard 8bit Intel HEX format.

14.3 Assembler Output

Optionally, a valid A51 assembler/C source listing file can be produced by C51 if the SRC command line switch is used. This has the original C source lines interleaved with the assembler and is very useful for getting to know how the compiler drives the 8051.

Do not be tempted to try hand-tweaking the compiler's efforts. Whilst you may be able to save the odd instruction here and there, you will create a totally unmaintainable program! It is much better to structure source code so that you write efficient code from the start. Simple, efficient C will produce the best 8051 code.

15 Assembler Interfacing To C Programs

The calling of assembler routines from C51 is not difficult, provided that you read both this and the user manual.

15.1 Assembler Function Example

The example below is taken from a real application where an EEPROM was being written in a page mode. Because of a 30us timeout of this mode, the 25us run time of the C51 code was viewed as being a bit marginal. It was therefore decided to code it in assembler.

If an assembler-coded function is to receive no parameters then an ordinary assembler label at the beginning of the function is simply called like any C function. Note that an extern function prototype must be given after the style of:

C51 File:

```
extern void asm_func(void).
```

A51 File:

```
ASM_FUNC:  MOV  A,#10   ; 8051 assembler instructions
```

Should there be parameters to be passed, C51 will place the first few parameters into registers. Exactly how it does this is outlined in section

The complication arises when there are more parameters to be passed than can be fitted into registers.

In this case the user must declare a memory area into which the extra parameters can be placed. Thus the assembler function must have a DATA segment defined that conforms to the naming conventions expected by C51.

In the example below, the segment

"?DT?_WRITE_EE_PAGE?WRITE_EE SEGMENT DATA OVERLAYABLE"

does just that.

The best advice is to write the C that calls the assembler and then compile with the SRC switch to produce an assemblable equivalent. Then look at what C51 does when it calls your as yet unwritten assembler function. If you stick to the parameter passing segment name generated by C51 you will have no problems.

Example Of Assembler Function With Many Parameters

C Calling Function

Within the C program that calls this function the following lines must be added to the calling module/source file:

```
   /* external reference to assembler routine */

extern unsigned char write_ee_page(char*,unsigned
                                              char,unsigned char) ;
  .
   dummy()
  .  {
     unsigned char number, eeprom_page_buffer,
                 ee_page_length ;
     char * current_ee_page ;
  .
     number = write_ee_page (current_ee_page,
                 eeprom_page_buffer, ee_page_length) ;
  .  } /* End dummy */
```

The assembler routine is:

```
        NAME EEPROM_WRITE ;

        PUBLIC  _WRITE_EE_PAGE              ; Essential!
        PUBLIC  ?_WRITE_EE_PAGE?END_ADDRESS ;
        PUBLIC  ?_WRITE_EE_PAGE?END_BUFFER  ;
;
P6     EQU  0FAH  ;
Port 6 has watchdog pin ;
;************************************************************************* ;
*<<<<<<<<< Declare CODE And DATA Segments For
              Assembler Routine >>>>>>>>>>>*
;*************************************************************************;
?PR?_WRITE_EE_PAGE?WRITE_EE SEGMENT CODE ?DT?_WRITE_EE_PAGE?WRITE_EE SEGMENT DATA
OVERLAYABLE ;
; ;************************************************************************* ;
*<<<<<< Declare Memory Area In Internal RAM For Local
          Variables Etc. >>>>>>*
;************************************************************************* ;
        RSEG ?DT?_WRITE_EE_PAGE?WRITE; ?_WRITE_EE_PAGE?END_ADDRESS:  DS   2   ;
?_WRITE_EE_PAGE?END_BUFFER:   DS   1   ;
;
; ;************************************************************************* ;
*<<<<<<<<<<<<<<< EEPROM Page Write Function >>>>>>>>>>>>>>*
;************************************************************************* ;
        RSEG   ?PR?_WRITE_EE_PAGE?WRITE ;
; _
WRITE_EE_PAGE:
        CLR    EA
        MOV    DPH,R6  ; Address of EEPROM in R7/R6
        MOV    DPL,R7  ;
;
        MOV    A,R3  ; Length of buffer in R3
        DEC    A     ;
        ADD    A,R7             ; Calculate address of last
        MOV    ?_WRITE_EE_PAGE?END_ADDRESS+01H,A ; byte
                                                    in page in XDATA.
        CLR    A                                 ;
        ADDC   A,R6                              ;
        MOV    ?_WRITE_EE_PAGE?END_ADDRESS,A     ;
;
        MOV    A,R5   ;  Address of buffer in IDATA in R5
        MOV    R0,A   ;
        ADD    A,R3     ;
        MOV    ?_WRITE_EE_PAGE?END_BUFFER,A ;
;
LOOP:   MOV    A,@R0      ;
        MOVX   @DPTR,A    ;
        INC    R0         ;
        INC    DPTR       ;
        MOV    A,R0       ;
        CJNE   A,?_WRITE_EE_PAGE?END_BUFFER,LOOP ;
;
        MOV    DPH,?_WRITE_EE_PAGE?END_ADDRESS      ;
        MOV    DPL,?_WRITE_EE_PAGE?END_ADDRESS+01H  ;
        DEC    R0          ;
;
CHECK:  XRL    P6,#08      ; Refresh watchdog on MAX691
        MOVX   A,@DPTR     ;
        CLR    C           ;
        SUBB   A,@R0       ;
        JNZ    CHECK       ;
;
        SETB   EA          ;
        RET                ; Return to C calling program
;
        END
;
```

15.2 Parameter Passing To Assembler Functions

In the assembler example the parameter "current_ee_page" was received in R6 and R7. Notice that the high byte is in the lower register, R6. The fact that the 8051 stores high bytes at the low address of any multiple byte object always causes head scratching!

The "_" prefix on the WRITE_EE_PAGE assembler function name is a convention to indicate that registers are used for parameter passing. If you are converting from C51 version <3.00, please bear this in mind.

Note that if you pass more parameters than the registers can cope with, additional space is taken in the default memory space (SMALL-data, COMPACT-pdata, LARGE-xdata).

15.3 Parameter Passing In Registers

Parameter passing is now possible via CPU registers (R0-R7). Coupled with register auto/local variables means that function calls can be made very quickly. Up to three parameters may be passed this way although when using long and/or float parameters only two may be passed, due to there being 4 bytes per variable and only 8 registers available! To maintain compatibility with 2.5x the NOREGPARMS #pragma is provided to force fixed memory locations to be used. Those calling assembler coded functions must take note of this.

Parameter Type	Char	Int+Spaced ptr	Long/Float	Generic Ptr
Parameter	R7	R6/R7	R4-R7	R1,R2,R3
Parameter	R5	R4/R5	R4-R7	R1,R2,R3
Parameter	R3	R2/R3		R1,R2,R3

16 General Things To Be Aware Of

The following rules will allow the compiler to make the best use of the processor's resources. Generally, approaching C from an assembler programmer's viewpoint does no harm whatsoever!

16.1

Always use 8 bit variables: the 8051 is strictly an 8 bit machine with no 16 bit instructions. char will always be more efficient than int's.

16.2

Always use unsigned variables where possible. The 8051 has no signed compares, multiplies etc., hence all sign management must be done by discrete 8051 instructions.

16.3

Try to avoid dividing anything but 8 bit numbers. There is only an 8 by 8 divide in the instruction set. 32 by 16 divides could be lengthy unless you are using an 80C537!

16.4

Try to avoid using bit structures. Until v2.30, C51 did not support these structures as defined by ANSI. Having queried this omission with Keil, the explanation was that the code produced would be very large and inefficient. Now that they have been added, this has proved to be right. An alternative solution is to declare bits individually, using the "bit" storage class, and pass them to a user-written function.

16.5

The ANSI standard says that the product of two 8- bit numbers is also an 8 bit number. This means that any unsigned chars which might have to be multiplied must actually be declared as unsigned int's if there is any possibility that they may produce even an intermediate result over 255.

However it is very wasteful to use integer quantities in an 8051 if a char can do the job! The solution is to temporarily convert (cast) a char to an int. Here the numerator potentially could be 16 bits but the result always 8 bits. The "(unsigned int)" casts ensure that a 16 bit multiply is used by C51.

```
{

unsigned char z ;
unsigned char x ;
unsigned char y ;

z = ((unsigned int) y * (unsigned int) x) >> 8 ;

}
```

Here the two eight bit numbers x and y are multiplied and then divided by 256. The intermediate 16 bit (unsigned int) result is permissible because y and x have been loaded by the multiplier library routine as int's.

16.6

Calculations which consist of integer operands but which always produce an 8 bit (char) due to careful scaling result thus:

```
unsigned int x, y ;
unsigned char z ;
z = x*y/256 ;
```

will always work, as C51 will equate z to the upper byte (least significant) of the integer result. This is not machine-dependant as ANSI dictates what should be done. Also note that C51 will access the upper byte directly, thus saving code.

16.7 Floating Point Numbers

One operand is always pushed onto an arithmetic stack in the internal RAM. In the SMALL model the 8051 stack is used, but in other models a fixed segment is created at the lowest available address above the register bank area. In applications where on-chip RAM is at a premium, full floating point maths really should not be used. Fixed point is a far more realistic alternative.

17 Conclusion

The foregoing should give a fair idea how the C51 compiler can be used in real embedded program development. Its great advantage is that it removes the necessity of being an expert in 8051 assembler to produce effective programs. Really, for the 8051, C51 should be viewed as a universal low to medium level language, which both assembler and C programmers can move to very simply. Access to on and off-chip peripherals is painless and the need for assembler device-drivers is removed.

Well-constructed C programs will lend themselves to being re used and eventually you should be able to create your own libraries of functions and core modules.

This is for me a strange conculsion as it effectively comes halfway through this paper. However that is the nature of embedded engineering. The end never is…. So always constrtuct your programs well because they will probably have a life far longer than you ancicipate.

18 Appendix A

Constructing A Simple 8051 C Program. Please note this is a program from the original C51 Primer. In time it will be converted to C51 V7 and MISRA-C compliance.

Often the most difficult stage in 8051 C programming is getting the first program to run! Even if you are not having to grapple with C as a new language, the business of dealing with special function registers, interrupts and memory-mapped peripherals can be a bit daunting.

This simple program contains all the basic steps required to get an 8051 program to run. Like all the classic first programs, it prints “hello world” down the serial port which is assumed to be connected to a dumb terminal.

```
A First C51 Program

/*****************************************************************************
*                  Main Program - Simplest Version                          *
*****************************************************************************/

/* This program is entered from the reset vector.  It simply initialises the serial port,
and
     prints “hello world” repeatedly */

/* Declare Memory Model */

.i.#pragma;#pragma SMALL  // Set SMALL model (on-chip RAM only)

#include “\C51P\INC\stdio.h”  // Include file contains function prototype for printf.
/* Function Prototype */

void serial0_init_T1(void) ;   // Serial port initialisation function

/* Main Loop */

void main(void)          // Enter from reset vector
{

serial0_init_T1() ;      // Initialise serial port 0 timer1 baudrate generator

/*** Loop Forever ***/

while(FOREVER) {

   printf(“hello world”) ;  // Send message down 8051 serial port forever
   }

}

/***************************************************************************
       This function initialises Serial Port 0 to run at
       4800 Baud using the Timer 1 auto-reload mode with a
       12MHz XTAL.
***************************************************************************/

/* To get 9600 baud with timer1 requires an 11.059MHz
   crystal! */

void serial0_init_T1(void)
   {

   TH1 = 0x0f3        ;  /* Timer 1 high byte (= reload value) SMOD = 0, F(Osc) = 12 MHz,
                      and Timer 1 in mode 2, baudrate of 4800 Baud Timer 1 Interrupt is
                      disabled after RESET */

   TMOD |= 0x20       ;      /* Load Timer Mode Control Register Timer 1 under software
                         control with TR1 as Timer in mode 2(= 8 bit, auto-reload) */
   S0CON = 0x52       ;      /* Serial connection in mode 1 (= 1 Start-,8 Data-, 1 stop
                         bit)start enabled Transmitter empty, Receiver empty */

   PCON |= 0x80       ;      /* SMOD = 1 to double baud rate */

   TR1 = 1                   ;  /* Timer 1 start */

    }
```

This should be placed in a module, preferably called “main.c” and compiled with:

```
>C51 main.c
```

This produces a file, ‘main.obj’

Next, link main.obj with the printf function, held in a C51S.LIB library, and fix the location of the program:

```
>L51 main.obj,\c51p\lib\c51s.lib to exec.abs
```

To yield an Intel OMF51 format file named “exec.abs”. If you are using an EPROM programmer, you will need an Intel HEX file.
Use OHS51.EXE for this:

```
>OHS51 exec.abs
```

to give exec.hex an Intel HEX file.

Basically this is all there is to producing a working C51 program! Some refinements might be to make sure that the C51LIB DOS environment variable has been set to indicate where the C51S.LIB is located.
To do this, make sure that you have

```
SET C51LIB=\C51P\LIB
```

in your autoexec.bat file.

Likewise, if you also add

```
SET C51INC=\C51P\INC,
```

the long and untidy pathname for ‘stdio.h’ can be eliminated.

If C51 has been installed properly, this should have already been done.

19 Appendix B

Driving The 8051 For Real Please note this is a program from the original C51 Primer. In time it will be converted to C51 V7 and MISRA-C compliance.

The following example program does the following typical
8051 tasks:

(i) Read a port pin value
(ii) Write a port pin value
(iii) Generate a periodic timer interrupt
(iv) Transmit data via the serial port
(v) Write to a memory-mapped port

```
It is suggested that to get started you steal sections from the following program!
Although the Siemens  80C537 has been used as the basis for this, the approaches used are
applicable to all 8051 variants.

#include <stdio.h>            /*     include standard io libs           */
#include <reg517.h>    /*     include 80C517 register set       */
#include <math.h>             /*     include mathematical prototypes   */
#include <string.h>           /*     include string handling functions */

#pragma MOD517                 /*    Use 80C537 extensions                      */
```

The 8051 areas covered are:

1. Serial Port0 - Polled Mode

   - Baudrate generation from timer1
   - Baudrate generation from baudrate generator

2. Analog To Digital Convertor

   - Reading values into an array

3. Frequency Measurement

   - Input Capture CC0

4. Time Pulse Generation

   - Output compare CC4

5. Symmetrical PWM Generation

   - CC3 and timer2 overflow

6. Zero CPU Overhead Asymmetric PWM Generation

   - CMx/Compare Timer

7. Accessing Memory-Mapped Ports

   - Via pointers
   - With XBYTE[]

```
**************************************************************************
*                        Global Definitions                                *
```

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

/*** CCMx PWMS ***/

    xdata float pwm_period = 42.5 ;                  // Initial period in
                                                 us,variable located in
                                                 XDATA.

    xdata float pwm_duty_ratio = 50 ; // Initial ratio in %
    xdata unsigned int pwm_prescale = 0 ;

/*** Analog Inputs ***/
       xdata float analog_data[4];   // Floating point array
       xdata unsigned char rx_byte;

    xdata unsigned char channel_0 = 0 ;
    xdata unsigned char channel_1 = 0 ;

/*** Timer0 Overflow Timebase ***/

    xdata unsigned int real_time_count = 0 ;

/*** Timed Pulse Generation ***/
    xdata unsigned char marker_angle = 128 ;
    data unsigned int marker_time = 0 ;
    unsigned int time_for_360 = 0  ;
    unsigned int time_last_360 = 0 ;
    xdata unsigned int frequency = 0 ;
    xdata unsigned int analog_data10 = 0 ;

/*** Port 1 Bit Definitions ***/
    sbit  P10   = 0x90;    // CC0
    sbit  P13   = 0x93;    // CC3
    sbit  P14   = 0x94;    // CC3

/*** Symmetrical PWM Generation ***/

   xdata unsigned int symm_PWM_DR = 256 ;    // Integer ratio from background
   xdata unsigned int symm_PWM_period = 2048 ; //PWM Period = 4096us
/

***************************************************************************
*                  General Definitions                                     *
**************************************************************************/

#define FOREVER 1

/*** CMx PWM Control ***/

#define Pulse_Width     25                          /* 50us marker pulse  */
#define PWM_Resolution  0.1666667    /* Smallest PWM time is at  12MHz */

/*** Cursor Positioning Escape Codes For VT52 ***/
code char Line0[] = { 0x1b,'Y',0x20,0x20,0 } ;
code char Line1[] = { 0x1b,'Y',0x21,0x20,0 } ;
code char Line2[] = { 0x1b,'Y',0x22,0x20,0 } ;
code char Line3[] = { 0x1b,'Y',0x23,0x20,0 } ;
code char Line4[] = { 0x1b,'Y',0x24,0x20,0 } ;
code char Line5[] = { 0x1b,'Y',0x25,0x20,0 } ;
code char Line6[] = { 0x1b,'Y',0x26,0x20,0 } ;
code char Line7[] = { 0x1b,'Y',0x27,0x20,0 } ;

code char Line8[] = { 0x1b,'Y',0x28,0x20,0 } ;
code char Line9[] = { 0x1b,'Y',0x29,0x20,0 } ;
code char Line10[] = { 0x1b,'Y',0x2a,0x20,0 } ;
code char Line11[] = { 0x1b,'Y',0x2b,0x20,0 } ;
code char Line12[] = { 0x1b,'Y',0x2c,0x20,0 } ;
code char Line13[] = { 0x1b,'Y',0x2d,0x20,0 } ;
code char Line14[] = { 0x1b,'Y',0x2e,0x20,0 } ;
code char Line15[] = { 0x1b,'Y',0x2f,0x20,0 } ;
code char Clear[] = { 0x0C,0 } ;
code char double_bell[] = { 0x07,0x07,0x07,0 }       ;

/
**************************************************************************
*                        Function Prototypes                             *
**************************************************************************/

       void ad_init(void);
       void serial_init(void);
       void serial0_init_BD(void);
       void serial0_init_T1(void);
       void send_byte(unsigned char);
       void ad_convert(void);
       void capture_init(void);
       extern void control_pwm(void) ;

/

*****************************************************************************
*      This function initialises the A/D convertor (P103 of 517 manual)           *
*****************************************************************************/

void ad_init(void)
{
    ADCON0 &= 0x80 ;   // Clear register but preserve BD bit
       ADCON0 |= 0x01 ;  /* Single conversion internal Start Channel 0 */
}
/

*****************************************************************************
*      This function will perform three conversions on the A/D convertor reading values
from channels 0 - 3                                                *
*****************************************************************************/
/* Channel 0 is read using the 10 bit programmable reference method */

void ad_convert(void)
    {
    unsigned char i;

       for(i = 1 ; i < 4 ; i++)
          {
          ADCON0 &= 0x80 ;  // Preserve BD bit (80C537 only)
       ADCON0 |= i ;
       DAPR    = 0 ;
          while (BSY)
         { ; }

          analog_data[i] = ((float) ADDAT * 5) / 255 ;
   }
  }
*****************************************************************************
* These routines will transmit and receive single characters   by Polled use of the
serial Port 0                                                            *
*****************************************************************************/

/* Note: In real applications, an interrupt-driven serial
   port is usually preferable to avoid loss of characters.*/

   char receive_byte(void)                         /* Polled use of serial port */
      {

      if(RI == 1)                          /* Test for char received     */
         {
         rx_byte = S0BUF ;                 /* Place char in rx_byte */
            RI = 0 ;                              /* clear flag                 */
         }
      else {
         rx_byte = 0 ;
         }
       return(rx_byte) ;
       }


  void send_byte(char tx_byte) /*Polled use of serial port*/
      {

      TI = 0;  //      Clear TI flag
      S0BUF = tx_byte ;              //      Begin transmission
      while (!TI) {;}                //      Wait until transmit flag is set

      }

/

**************************************************************************
*      This function initialises Serial Port 0 to run at 9600 Baud using the Siemens Baud
rate generator (see P76 of the 517 Manual)          *
**************************************************************************/
/* This method does not tie up timer1 as on ordinary 8051’s */
void serial0_init_BD(void)
{
     BD   = 1;                /* Enable Baud rate generator */
     PCON = PCON | 0x80; /* Set SMOD to double baud rate */
     S0CON = 0x50;            /* Mode 1, Receiver enabled */
     TI   = 1;                /* Set Transmit interupt flag for first run through PRINTF
*/
}
/
**************************************************************************
*      This function initialises Serial Port 0 to run at  4800 Baud using the Timer 1
auto-reload mode.                                           *
**************************************************************************/

/* To get 9600 baud with timer1 requires an 11.059MHz
   crystal */

void serial0_init_T1(void)
   {

   TH1 = 0x0f3  ;      /* Timer 1 high byte (= reload value)  SMOD = 0,
                  F(Osc) = 12 MHz,and Timer 1 in mode 2,baudrate of 4800
                  Baud Timer 1 Interrupt is disabled after RESET */

   TMOD |= 0x20 ;      /* Load Timer Mode Control Register Timer 1
                          under software control with TR1 as Timer
                          in mode 2(= 8 bit, auto-reload) */

   S0CON = 0x52 ;      /* Serial connection in mode 1 (= 1 Start-, 8 Data-, 1 stop bit)
                          start enabled Transmitter empty, Receiver empty */

   PCON |= 0x80 ;      /* SMOD = 1 to double baud rate */

   TR1 = 1      ;      /* Timer 1 start */
   }
/

*****************************************************************************
*      Generate 2ms Timer Tick On Timer 0                                   *
*****************************************************************************/

/* Entered every timer0 overflow */
void timer0_init(void)
   {
   TR0 = 0             ;
   TMOD |= 01  ;       /* 16 bit timer mode */
   TH0 = 0xf8  ;       /* Reload with with count for 2ms time base at 12MHz        */
   TL0 = 0x82  ;
   TR0 = 1             ;       /*      Start timer
        */
   IEN0 |= 0x02        ;       /*      Enable Timer 0 Ext0 interrupts */
   } /*init_timer_0*/
/
*****************************************************************************
*       Timer0 Interrupt Service Routine                                          *
*****************************************************************************/

/* An allowance really needs to be made for the fact that  the timer is stopped during
the re-
    initialisation process */

/* “interrupt” arguments are:

  ‘1’ => generate interrupt vector at address 8*1 + 3 = 0x0b
  ‘2’ => Switch to register bank two on entry, restore
          original bank on exit */

void timer0_int(void) interrupt 1 using 2
     {

       /* Setup Next Interrupt ***/

       IEN0 &= 0xfd ;    /* Clear interrupt flags  */
       TR0 = 0        ;       /* Stop timer */

       TH0 = 0xf8     ;       /* Reset timer for next interrupt */
       TL0 = 0x2f     ;       /* 2ms at 12 MHZ                      */
       TR0 = 1        ;       /* Start timer */
       IEN0 |= 0x02  ;

       real_time_count++      ;

       P6 ^= 0x08             ;
       }
/

*****************************************************************************
*       This function sets up the Capture Compare Unit and generates a PWM output on Port
4.0 (Pin 1). See p112 of the 517 Manual *

*       => CTREL = 65536 - 255 for 42.5us period/ overflow rate at 12MHz *

*       Compare timer counts from CTREL to 65535 when Port bit is cleared Port bit set
when Compare timer = CM0 to give asymmetric 8 bit PWM *
*****************************************************************************/

/* This PWM requires no CPU time and is thus very efficient */

/* On 535 an interrupt service would be required to reload the compare */
/* register                                                                  */
void pwm_init(void)
   {

   union { unsigned int temp ;
           unsigned char tmp[2] ; } t ;

       CTCON = 0  ;   // Basic count time = 166ns

  t.temp = -pwm_period/PWM_Resolution ; // 42.5us
                                                       initial period

       CTRELH = t.tmp[0] ;
       CTRELL = t.tmp[1] ;

       CM1 = t.temp + ((unsigned int)(65536 - t.temp) * pwm_duty_ratio)/100 ; // Initial
duty ratio = 255:1

       CM0 = CM1 ;

       CMSEL   =       0       ;
       CMSEL |=       1 ;   // Assign CM0 to compare timer
       CMSEL |=       2 ;   // Assign CM1 to compare timer
```

```
        CMEN    =       0       ;
        CMEN |  =       1 ;   // Enable port 4.0 as PWM
                                        (front)
        CMEN |  =       2 ;   // Enable port 4.1 as PWM
                                        (rear)
        }
/

*****************************************************************************
*      This function initializes the Output Compare/Input Capture System on
        Timer2/Port 1. Two captures are enabled: CC0 captures an event on Pin      1.0,
CC1 will be triggered by a write to the low  order Byte CCL1       *
*****************************************************************************/

/* The capcom unit when attached to timer2 is suitable for
   frequency   */
/* measurement and pulse generation                                        */

   void capture_CC0_init(void)
        {

        T2CON   =       0       ;
        T2I1    =       0 ;  /* Timer 2 = 12MHz/24 = 2us/count */
        T2I0    =       1 ;
        T2PS    =       1 ;  /* /2 prescale for 2us/count */

        CTCON   =       0 ;

        T2CM    =       1 ;  /* Timer 2 compare/capture in mode 1*/
        T2R1    =       0 ;  /* No autoreload off CC0 */

        CCEN    =       0 ;
        CCEN    |= 0x01 ;      /* Input capture on CC0 */

        CCEN    |= 0x0C ;      /* Timer 2 latched into CC1 on write
                                  into CCL1 */

        CCEN    |= 0x80 ;   /* CC3 is output compare */

        I3FR    = 0              ;     /* CC0 is initially -ve edge
                                  triggered */

        P1 |    = 0x01  ;      /* Put port 1.0 high for input
                                  capture */

        EX3     = 1      ;            /* Enable capture interrupt for road
                                  speed  */
        EX2     = 1      ;            /* Enable output compare interrupt
                                  for ign0 */

        CC4EN   = 0x05  ;      /* CC4 port 1.4 is output compare
                                  mode 1    */

        IP0     = 0      ;            /* Initialise interrupt priorities */
        IP1     = 0      ;

        IP1     |= 0x26 ;      /* Make CC4 interrupt 3 priority */
        IP0     |= 0x3A ;      /* Input capture is 2 priority */
        }
/


*****************************************************************************
*             Input Capture Interrupt On Port1.0/CC                          *
*****************************************************************************/

/* On every negative edge at P1.0, this routine is entered*/
/* Frequency calculation is possible using:

frequency      = 100000/(Timer2 Count Time * (this T2 - last
                                                       timer2))

                        = 50000/(CRC - last CRC)

A new pulse is generated at a fixed angle after the interrupt using CC4 output compare

- This is the basis for ignition and injection timing in engine management
   systems
- The maths unit is essential for keeping run times short.

*/

   void CC0_int(void) interrupt 10 using 3
      {

      unsigned int temp ;

      /* Calculate Input Frequency */

      frequency = 500000 /(unsigned long) (CRC -
                                 time_last_360) ;

      time_for_360 = CRC - time_last_360 ;

      temp = CRC + (unsigned int)
         ((unsigned long)((unsigned long)time_for_360 * marker_angle)/255) ;

      EAL = 0 ;
      marker_time = temp ;
      EAL = 1 ;

      time_last_360 = CRC ;
      }

/


****************************************************************************
*       Generate marker pulse after CC0 interrupt                          *
***************************************************************************/

/* Entered in response to request from CC0 interrupt to generate a pulse at a predefined
time
    afterwards.                              */

void marker_int(void) interrupt 9 using 2
    {

    unsigned int timer_temp ;

    EX2 = 0 ;

       if(P14 == 0)
          {

   /* Port Pin Low */

          if((int)(marker_time - CC4 - 500) > 0) {
             timer_temp = marker_time ;
             }
          else {
             timer_temp = marker_time + time_for_360 ;
             }

          CC4 = timer_temp ;
          IEX2 = 0 ;
          P14 = 1 ;     // Turn on at next compare
          EX2 = 1      ;
          }
       else
          {
```

```
   /* Port Pin High */

          timer_temp = CC4 + Pulse_Width ;
          CC4 = timer_temp ;
          IEX2 = 0  ;
          P14 = 0 ;      // Turn off at next compare
          EX2 = 1                  ;
          }
    }
```

```
*****************************************************************************
*      This function initialises the Output Compare/Input Capture System on Timer2/
*      Port 1 to generate a symmetrical PWM on CC4.                        *
*****************************************************************************/

/* This gives a PWM output where the on-time grows from  either side of the timer 2
   overflow point */

/* This is very useful for motor control as the symmetrical nature of the waveform
reduces
   the higher current harmonics circulating in the windings under changing duty ratio
    conditions.                              */

/* Downside is that two interrupt services are required per  period                */
```

Symmetrical PWM Waveform

Asymmetrical PWM Waveform

```
   void symm_PWM_init(void)
      {

      T2CON = 0 ;   /* Clear configuration register  */
      T2I1 = 0  ;    /* Timer 2 = 12MHz/24 = 2us/count */
      T2I0 = 1  ;
      T2PS = 1  ;  /* /2 prescale for 2us/count  */
      /* Additional prescale possible on BB step    */

      T2CM = 1  ;  /* Timer 2 compare/capture in mode 1 */
      T2R1 = 1  ;  /* Autoreload off CC0  */
      T2R0 = 0  ;  /* mode 1 (CRC into Timer2 at rollover)*/

     /* Set initial reload value (4096us/2048 steps) */

      CRC = -2*symm_PWM_period ;

      ET2 = 1    ; /* Enable timer2 overflow interrupt */

      EX3 = 1   ; /*Enable capture interrupt for PWM drive*/

      CCEN = 0 ;  /* CRC - CC2 unused */
      CCEN |= 0x80 ; /* CC3 is symmetrical PWM output */
      IP0 = 0 ;  /* Initialise interrupt priorities  */
      IP1 = 0 ;

      IP1 |= 0x20 ; /* Make CC3/T2 Overflow interrupts
                           priority 3 */

      P10 = 0 ;
      }

/***************************************************************************
*                 Timer 2 Overflow Interrupt              *
***************************************************************************/

/* Interrupt at centre point of waveform to create next off point */

/* A good example of where C  now givesoverhead when compared with  assembler! */

/* USING gives single cycle registerbank switch like ‘166 */

   void timer2_overflow(void) interrupt 5 using 2
      {

      /* Runtime here limits min/max PWM DR */


       P1 |= 0x01  ; /* Toggle P1.0 to show centre of PWM */

      TF2 = 0     ; /* Clear interrupt request flag */

      CC3 = CRC + symm_PWM_DR ;
      IEX6 = 0 ;
      P13 = 0 ;
      EX6 = 1 ;

      P1 &= 0xfe  ; /* Toggle P1.0 to show centre of PWM */
      }

/***************************************************************************
*              CC4 Interrupt For Symmetrical PWM           *
***************************************************************************/

/* Interrupt at end of first on period of waveform to create next on point */

 void symm_PWM_CC3_int(void) interrupt 13 using 2
    {

    /* Runtime here limits min/max PWM DR */

    CC3 = -symm_PWM_DR ;
    IEX6 = 0 ;
    P13 = 1 ;
    EX6 = 0 ;   // No further interrupts this period
    }

/***************************************************************************
*       Modulate Symmetrical PWM With Analog Input0        *
***************************************************************************/

/* Duty ratio is calculated in background to prevent having
   to do floating */
/* point calculations in interrupts  */

/* Note: As PWM is symmetrical, duty ratio cannot exceed 1/2
   period  */


void mod_symm_pwm(void) {

   union { unsigned int temp ;
           unsigned char tmp[2] ;
} t ;
```

```
   t.tmp[0] = CRCH ;
   t.tmp[1] = CRCL ;

   symm_PWM_DR = ((65536-t.temp)/2 * (5-analog_data[1]))/5 ;
   }

/

****************************************************************************
*                  Drive TOC PWM’s                      *
****************************************************************************/

void configure_pwm(void) {

   unsigned int temp ;
   union { unsigned int temp ;
           unsigned char tmp[2] ; } t ;

   t.temp = -pwm_period/((float)pwm_prescale * PWM_Resolution) ;

   CTRELH = t.tmp[0] ;
   CTRELL = t.tmp[1] ;

   CM1 = t.temp + ((unsigned int)(65536 - t.temp) * pwm_duty_ratio)/100 ;
   }

/

****************************************************************************
*    Write First Message To Terminal      *
****************************************************************************/

/* Whilst many printf’s are used here, in a real program
   they would not */
/* in the main program loop due to huge run time  */

void initialise_screen(void) {

   printf(“%s”,Clear) ;  // Clear Screen
   printf(“%s  *** 80C537 Demo Program ***  “,Line0) ;
                                          //Print Sign-On
   printf(“%s”,Line1) ;   // Print Sign-On

   }
****************************************************************************
*           Modulate PWM With Analog Input0              *
****************************************************************************/
void mod_pwm(void) {

   union { unsigned int temp ;
           unsigned char tmp[2] ; } t ;

   t.tmp[0] = CTRELH ;
   t.tmp[1] = CTRELL ;

   CM0 = t.temp + ((65536-t.temp) * (5-analog_data[1]))/5 ;
   }

/****************************************************************************
*           Send Information To Terminal                *
****************************************************************************/
void print_info(void) {

printf(“%sAnalog 0a(8bits)     = %-1.2f Volts    “,Line3,analog_data[1]) ;
printf(“%sAnalog 2 (8bits)      = %-1.2f Volts    “,Line4,analog_data[2]) ;
printf(“%sPWM Fbck (8bit)    = %-1.2f Volts    “,Line5,analog_data[3]) ;
printf(“%sFrequency           = %d Hz            “,Line6,(unsigned int)frequency) ;
printf(“%sTimer                      = %d x2 ms        “,Line7,(unsigned int)
real_time_count) ;
```

```
   }
/***************************************************************************
*                  Access Memory-Mapped Port               *
***************************************************************************/

/* This function receives a port address and a value to
   write to it. It returns a value at a fixed address */

#include <absacc.h>     // Contains definition of XBYTE[] macro
                                       // '<' and '>' mean that the include
                       // file will be obtained from the
                       // directory indicated by
                                       // the C51INC DOS environment variable

unsigned char get_memory_port(unsigned int port_address, unsigned char value) {

   unsigned char port_value ;                // Returned variable
   unsigned char xdata *port_pointer ;       // Declare uncommitted pointer into external
                                                memory space (xdata)

   port_pointer = (char*) port_address ;     // Make uncommitted pointer point at
                                         required address


   *port_pointer = value ;                         // Write value to port

   port_value = XBYTE[0x8000] ;              // Get value from external address 0x8000

   return(port_value) ;
   }

/***************************************************************************
*                  Main Program - Full Version            *
***************************************************************************/

/* This program initialises the peripheral functions and  then loops around, reading the
    A/D converter and  transmitting values down the serial port  */

void main(void)      // Enter from reset vector
{
serial0_init_T1() ;  // Initialise serial port 0 timer1 baudrate generator

ad_init() ;      // Initialise A/D converter

capture_CC0_init() ; // Initialise input capture/T2 for freq. measurement
                              // and timed pulse generation /*
symm_PWM_init() ;    // Generate symmetrical PWM on CC3 (P1.3) */
                         // (may only be present if capture_CC0_init() is
                                     // commented out)
pwm_init() ;         // Initialise TOC PWM on CMx

timer0_init() ;        // Initialise timer 0 overflow 2ms  interrupt

EAL = 1 ;              // Enable interrupts

initialise_screen() ;  // Write startup message to terminal

/*** Loop Forever ***/

while(FOREVER) {

   P6 ^= 0x08 ;                      // Refresh MAX691 watchdog every background loop
                                             // This is attached to port 6, bit 3.

   ad_convert() ;      // Read all analog channels

   print_info() ;      // Send analog values etc. to terminal

   mod_pwm() ;         // Modulate PWM0 with analog channel  0 input

   mod_symm_pwm() ;    // Modulate symm PWM with analog channel 0 input
```

```
    }
}
```

20 Appendix C

C51 Version 6 Code Comparison

The following competitive benchmarks for the Keil C51 compiler were run in June 2001 to compare the output generated by the Keil Version 5 and Version 6 compilers. The source code used for the Whetstone and Dhrystone benchmarks is included with the Keil evaluation compiler.

20.1 Dhrystone

Dhrystone is a general-performance benchmark test originally developed by Reinhold Weicker in 1984. This benchmark is used to measure and compare the performance of different computers or, in this case, the efficiency of the code generated for the same computer by different compilers. The test reports general performance in Dhrystone per second.

Like most benchmark programs, Dhrystone consists of standard code and concentrates on string handling. It uses no floating-point operations. It is heavily influenced by hardware and software design, compiler and linker options, code optimizing, cache memory, wait states, and integer data types.

Compiler	**Keil C51 Version 6.12**	**Keil C51 Version 6.12**	**Keil C51 Version 5.02**
Memory Model ROM Model Optimization Level	LARGE LARGE 9, SIZE	LARGE LARGE 8, SPEED	LARGE LARGE 6, SIZE
Exe Time 12MHz 8051	1.112 secs	1.029 secs	1.096 secs
Exe Time 25MHz DS320	0.258 secs	0.234 secs	0.254 secs
Module Code Size	1717 bytes	2163 bytes	1905 bytes
Dynamic XDATA	5523 bytes	5523 bytes	5538 bytes
Total Code Size	5197 bytes	5614 bytes	5269 bytes

20.2 Whetstone

Whetstone is a benchmark test which attempts to measure the speed and efficiency at which a computer performs floating-point operations. The result of the test is given in units called whetstones.

Compiler	**Keil C51 Version 6.12**	**Keil C51 Version 6.12**	**Keil C51 Version 5.02**
Memory Model ROM Model Optimization Level	LARGE LARGE 9, SIZE	LARGE LARGE 8, SPEED	LARGE LARGE 6, SIZE
Exe Time 12MHz 8051	4.647 secs	4.445 secs	4.493 secs
Exe Time 25MHz DS320	0.973 secs	0.915 secs	0.941 secs
Module Code Size	3596 bytes	5446 bytes	4306 bytes
Dynamic XDATA	186 bytes	186 bytes	189 bytes
Total Code Size	8618 bytes	10486 bytes	9236 bytes

20.3 The Sieve of Eratosthenes

The Sieve of Eratosthenes is a standard benchmark used to determine the relative speed of different computers or, in this case, the efficiency of the code generated for the same computer by different compilers. The sieve algorithm was developed in ancient Greece and is one of a number of methods used to find prime numbers. The sieve works by a process of elimination using an array that starts with 2 and keeps all the numbers in position. The process is:

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25

Starting after 2, eliminate all multiples of 2.

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25

Starting after 3, eliminate all multiples of 3.

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25

Starting after 5, eliminate all multiples of 5.

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25

Continue until the next remaining number is greater than the square root of the largest number in the original series. In this case, the next number, 7, is greater than the square root of 25, so the process stops. The remaining numbers are all prime.

2 3 5 7 11 13 17 19 23

For each C operation the number of cycles to execute typical examples is given for all supported data types. To give some idea of execution times, with a 12MHz 8031, one cycle is 1us. Please note that timings for long and float operations are considerably reduced on the Siemens 80C537 due to its 32 bit maths unit.

Cycle Table Key

Unsigned Char	-	8 bits
Char	-	8 sign
Unsigned Int	-	16 bits
Int	-	16 sign
Unsigned Long	-	32 bits
Long	-	32 sign
float	-	float (32 bits IEEE single precision)

Notes:

- Timings include parameter loading pre-amble where appropriate.
- Clock speed assumed to be 12MHz (1us/cycle), if not otherwise stated.
- The small memory model was used so that no off-chip ram was employed.

Basic C Mathematical Functions

+ Addition

8 bits	8 sign	16 bits	16 sign	32 bits	32 sign	float

Cycles:	3	3	6	6	63	63	140

- **Subtraction**

	8 bits	8 sign	16 bits	16 sign	32 bits	32 sign	float
Cycles:	4	4	7	7	64	64	146

***** **Multiplication**

	8 bits	8 sign	16 bits	16 sign	32 bits	32 sign	float
Cycles:	10	13	46	48	160	160	131

/ **Division**

	8 bits	8 sign	16 bits	16 sign	32 bits	32 sign	float
Cycles:	8	19	26	39	1611	1624	134

% **Modulo**

	8 bits	8 sign	16 bits	16 sign	32 bits	32 sign	float
Cycles:	3	3	6	6	63	63	140

Examples

a = b + c ;
a = b/c   ;

Complex Mathematical Functions

sin(x)

float

Cycles: 1553

cos(x)

float

Cycles: 1433

tan(x)

float

Cycles: 2407-9570

exp(x)

float

Cycles: 3002-7870

sqrt(x)

float

Cycles: 42-2860

log(x)

float

Cycles: 45-6050

Other Maths Functions are:

cosh	Hyperbolic cosine
sinh	Hyperbolic sine
abs	find absolute value
rand	generate a random number

Examples:

x = sin(3.1415926/2) ; find the sine of (PI/2)
x = sqrt(2) ; find square root of x

Bitwise Functions

These allow direct bit by bit operations to be performed.

& **AND**

	8 bits	8 sign	16 bits	16 sign	32 bits	32 sign
Cycles:	3	3	6	6	63	63

| **Inclusive OR**

	8 bits	8 sign	16 bits	16 sign	32 bits	32 sign
Cycles:	3	3	6	6	63	63

^ **Exclusive OR**

	8 bits	8 sign	16 bits	16 sign	32 bits	32 sign
Cycles:	3	3	6	6	63	63

! **NOT (Invert)**

	8 bits	8 sign	16 bits	16 sign	32 bits	32 sign
Cycles:	3	3	6	6	63	63

Examples:

a = b & 0xfe ; make a equal to a bit wise AND with 0xFE (11111110)
a = b | 0x01 ; make a equal to a bit wise OR with 0x01 (00000001)

Two Operand Functions

=	Make left side equal to right side
==	test for left being equal to right

+= Add two operands and store result in first one.

	8 bits	8 sign	16 bits	16 sign	32 bits	32 sign	float
Cycles:	1	1	5	5	59	59	140

-= **Subtract two operands and store result in first one.**

	8 bits	8 sign	16 bits	16 sign	32 bits	32 sign	float
Cycles:	1	1	5	5	59	59	140

*= **Multiply two operands and store result in first one.**

	8 bits	8 sign	16 bits	16 sign	32 bits	32 sign	float
Cycles:	1	1	5	5	59	59	140

/= Divide two operands and store result in first one.

	8 bits	8 sign	16 bits	16 sign	32 bits	32 sign	float
Cycles:	1	1	5	5	59	59	140

Example:

```
a = b                   ;       Make a equal to b
if(a == b) { }          check whether a is equal to b
a += 3                  ;       a is equal to itself + 3
a /= 10         ;       a is equal to itself divided by 10
```

Relational And Logical Functions

These are used to test data and are usually used with if() and other control statements.

&& AND

	8 bits	8 sign	16 bits	16 sign	32 bits	32 sign	float
Cycles:	6	6	8	8	28	28	28

|| OR

	8 bits	8 sign	16 bits	16 sign	32 bits	32 sign	float
Cycles:	6	6	8	8	28	28	28

> Greater than

	8 bits	8 sign	16 bits	16 sign	32 bits	32 sign	float
Cycles:	5	9	7	11	85	88	302

< Less than

	8 bits	8 sign	16 bits	16 sign	32 bits	32 sign	float
Cycles:	5	9	7	11	85	88	302

>= Greater than or equal to

	8 bits	8 sign	16 bits	16 sign	32 bits	32 sign	float
Cycles:	5	9	7	11	85	88	302

<= Less than or equal to

	8 bits	8 sign	16 bits	16 sign	32 bits	32 sign	float
Cycles:	5	9	7	11	85	88	302

Examples:

```
if(a > b) {
   /* executable code 1 */
   }

if( (a == 1) && (b == 2)) {
   /* executable code 1 */
   }
else {
   /* Alternative executable code */
   }
```

Execute code 1 if a is equal to 1 and b equal to 2 otherwise execute the alternative block.

```
if( (a == 1) || (b == 2)) {
   /* executable code */
   }
```
Execute if a is equal to 1 or b equal to 2

Increment And Decrement

These make direct use of the INC xx opcodes and consequently are very fast. Normally, they are used as part of larger C expressions where a value needs incrementing or decrementing.

++ Increment

	8 bits	8 sign	16 bits	16 sign	32 bits	32 sign	float
Cycles:	1	1	5	5	59	59	140

Decrement

	8 bits	8 sign	16 bits	16 sign	32 bits	32 sign	float
Cycles:	1	1	5	5	59	59	140

Examples:

```
i ++ ;  Post-increment i
++ i ;  Pre-increment i
i - - ; Post-decrement i
- - i ; Pre-increment i
```

```
for(i = 0 ; i < 10 ; i) {
   P1 = array[i++] ;     /* Sequentially write all the   */
                         /* values in array onto Port 1. */
   }                     /* i points to next value after */
                         /* after current access         */
```

Shifting

These allow values to be shifted left or right by a number of bit positions, determined either by a constant at compile time or a variable at run time.

>> Right shift

	8 bits	8 sign	16 bits	16 sign	32 bits	32 sign
Cycles:	7	7	56	56	129	129 (7 shifts)

<< Left shift

	8 bits	8 sign	16 bits	16 sign	32 bits	32 sign	float
Cycles:	7	7	56	56	129	129 (7 shifts)	

Examples:

```
a << 2  ;   shift a left two bit places
a << b  ;   shift a left by a number of bit positions
                determined by the value of b
```

Strings And Arrays

These are a number of sequential locations that together constitute some sort of larger single data object. Arrays may be single or multidimensional, as is BASIC etc.. Strings are as in BASIC but, because of C’s near-assembler nature, they must be handled with care - you must always be aware where they end! A true string is always finished with a zero, called the “null terminator”.

array[4]	;	an array of four elements, STARTING at element 0
array[4][2]	;	a two-dimensional array of four by 2 elements,STARTING at element 0,0
“ABCDEF”	;	a true string of ascii characters, with a zero after the last element. It is the use of doublequotation marks that defines this as a true string. Looking at the memory in which this was declared would show:65,66,67,68, 69,70,00
{ ‘A’,’B’,’C’,’D’,’E’,’F’ }	;	an array of ascii characters with no null terminator. Note the { and } defining the limits of the complete data object.

Examples:

char array[4]	;	Reserve a RAM area of 4 bytes into which 8 bit data will be put at run time.

char array[] = { “ABCD” } ; Fill a RAM area with ABCD0 prior to starting the main() function. The ‘0’ is the null terminator

Handling Strings And Characters

strcpy(*destination,*source) ;
8 element strings

Cycles: 102

- Copy string pointed at by *source to another string pointed at by *destination. The second string is completely overwritten in the process.

strcat(*destination,*source) ;
8 element strings

Cycles: 913

- Concatenate the string pointed at by *source onto another string pointed at by *destination.

result = strcmp(*destination,*source) ;
8 element strings

Cycles: 152

- Compare two strings pointed at by *source with another string pointed at by *destination. If equal, value of 1 is returned.

result = strlen(*source) ;
8 element string

Cycles: 505

- Find the length of the *source string

In addition to these functions, a range of other string and character functions are provided to perform tasks such as:

```
atoi()         ascii to integer
atof()         ascii to floating point
itof()         integer to floating point
isalpha()              test for alpha character
isdigit()              test for digit
isalnum()              test for alpha-numeric
```

+ many other pre-defined routines.

Examples:

```
char x[10] ;
char *y = “String of chars” ;
```

strcpy(x,y) ; - Copies string pointed at by y to the empty array x. Note, C does not check that x is actually big enough to hold the string!

Program Control

```
if (condition) {/* Code */;} else { /* Alternative Code */ ; }
```

- Perform one of either two blocks of code,
  depending on the result of a specified condition

	8 bits	8 sign	16 bits	16 sign	32 bits	32 sign	float
Cycles:	3	3	6	6	79	79	131

```
for( i = 0 ; i < end_value ; i = i + 1) {/*Executable Code*/;}
```

- Repeat executable code until i = end_value.

	8 bits	8 sign	16 bits	16 sign	32 bits	32 sign
Cycles:	15	17	23	25	227	233

```
do { /* Executable Code */ ; } while(condition is true) ;
```

- Perform executable code while condition is true

	8 bits	8 sign	16 bits	16 sign	32 bits	32 sign
Cycles:	5	6	7	8	79	82

do-case - execute blocks of code determined by the value of a control variable

No data measured

Examples:

```
if(a == b) { /* Executable code*/ }
```

- execute code within braces if a equal to b

```
for(i = 0 ; i > end_value ; i++) { /* Executable code*/ }
```

- execute code until i is equal to end_value (i.e. not greater than)

```
do { /* Executable code*/ } while i++ < end_value ;
```

- execute code while i less than end_value

```
switch(x) {

   case 1 :
      y++ ;
   break ;

   case 2 :
      y ;
   break ;

   case 3 ;
      y *= y ;
   break ;
   }
```

- Perform the operation determined by the value of x.

Examples:

```
for(i = 0 ; i > end_value ; i++) {
   /* Executable code*/
   if( x == i) {
      break ;
      }
   }
```

- execute code until i is equal to end_value (i.e. not greater than) but if x is ever equal to i then break out of the loop immediately.

Accessing Bits

Bit	A single bit variable, located in the Bit-addressable memory area
Sbit	A single bit variable, located in the bit-addressable memory, either in the user or sfr area. When located in the user area, sbit is a defined bit within a larger char or int variable.

Examples:

```
bdata char x ;/* x is an 8 bit signed number in the bit area */

sbit sign_bit = x ^ 8 ;  /* bit 8 is the sign bit */
```

Now to test whether x is negative, the state of sign_bit need only be tested:

```
if(sign_bit) {
   /* x is negative */ ;
   sign_bit = 0 ;
   }
```

Gives:

```
     JNB   sign_bit  POSITIVE
     CLRB  sign_bit

POSITIVE:
```

Or using a non-sbit method:

```
if(x < 0) {
   /* x is negative */ ;
   sign_bit = 0 ;
   }
```

Gives:

```
      MOV   A,x
      ANL   A,080H
      JZ    POSITIVE
      ANL   x,07FH
POSITIVE:
```

Handling 8051 Ports and SFRs

Examples:

```
   P1 = 0xff                   ;      writes value ff to port 1
   ADCON |=  0x80              ;      OR 80 hex into ADCON
   P1^0 = 1                    ;      set bit 0 of port 1
```

Getting Data In And Out Of C Programs In The 8051

```
printf(“string”,*x,*y,...)
```

- Print the characters, numbers and or strings contained within () to the serial and thence to a terminal (VT100 etc).

```
       16 * 8 bit characters
```

```
Cycles: 3553
scan(&x,...)
```

- Store incoming characters from terminal into memory buffers indicated within (). Note that the “&” implies “the address of buffer x”.

```
       16 * 8 bit characters
```

Cycles: Not measurable but similar to “printf”

Examples:

```
value_1 = 3.000 ;
value_2 = 4.256 ;

printf(“Results Are: %f & %f”,value_1,value_2) ;
```

“Results Are: 3.000 & 4.256” is printed on terminal screen. Here the numerical values of the two numbers are substituted into the two “%f” symbols.

```
char keyboard_buffer[20]
```

scan(&keyboard_buffer) read incoming characters from terminal keyboard into memory starting at the address of keyboard_buffer.

21 Appendix D

A Useful Look-up Table Application Please note this is a program from the original C51 Primer. In time it will be converted to C51 V7 and MISRA-C compliance.

In a real system, getting a true floating point sine would take around 1ms. In a very time-critical application this may well be unacceptable. If only an approximation is required, it is possible to use linear interpolation to get values between the known values in the table.

To do this, a look-up table interpolator is required. Below is a combine one and two dimensional table interpolator, taken from a real project. Here, the 2-D capability is not used!

Note: The term ".i.Map;map" is used instead of look-up table.

```
#include <reg517.h>
*************************************************************************/
/*      Main Interpolation Routine      */
*************************************************************************/
/*                                                  */
/* This routine has been optimised to run as fast as
       possible at the ***/
/* expense of code size.  Further savings could be made by
       re-using  temporary RAM.                         */

/* With a 5 x 5 map, run time is 490us - 735us at 12MHz */
/* or 290us - 400us with 12MHz Siemens 80C537        */
*************************************************************************/
/* Input Map Format:                                            */
/*                                                          */
/*  { x_size,y_size,                                    */
/*    x_breakpoints,                                      */
/*    y_breakpoints,                                      */
/*                                                        */
/*    map_data } ;                                               */
/*                                                       */
*************************************************************************/
unsigned char interp(unsigned char x_value,
/* x-axis input                                                         */
                     unsigned char y_value,
/* y-axis input                        */
                     unsigned char const *map_base
/* pointer to table base                                */
                     )
   {

   /* Declare Local RAM */

   unsigned char x_size      ;
   unsigned char y_size      ;

   unsigned char x_offset ;
   unsigned char y_offset ;
   unsigned char x_break_point1,x_break_point2 ;
   unsigned char y_break_point1,y_break_point2 ;

   unsigned char map_x1y1 ;
   unsigned char map_x2y1 ;
   unsigned char map_x1y2 ;
   unsigned char map_x2y2 ;

   unsigned char result    ;
   unsigned char result_y1 ;
   unsigned char result_y2 ;
   unsigned char const *mp  ;
   unsigned char x_temp1,x_temp2, y_temp2 ;

   /* Get Size Of Map */

   x_size = *map_base   ;
   y_size = map_base[1] ;

   /* Create Temporary Map Scanning Pointer */

   map_base += 2 ;
   x_offset = x_size - 1 ;
   mp = map_base + (unsigned char)x_offset ;

   /* Locate Upper and Lower X Breakpoints       */
   /* Find break point immediately below x-value */
   while((x_value < *mp) && (x_offset != 0))
      {
      mp ;
      x_offset ;
      }

   /* Extract Upper And Lower X-Breakpoints From Map */

   x_break_point1 = mp[0] ;
   x_break_point2 = mp[1] ;
   x_temp2 = (x_break_point2 - x_break_point1) ;  // bpt2 still in ACC

   /* Safety Check To Prevent Divide By Zero */

   if(x_temp2 == 0) {
      x_temp2++ ;        // Ensure denominator never zero
      }

   /* Check For x_value Less Than Bottom Breakpoint Value */

   if((x_offset == x_size - 1) || (x_value <= x_break_point1))
      {
      x_value = x_break_point1 ;
      }

   x_temp1 = (x_value - x_break_point1) ;

   /* Locate Upper And Lower Y Breakpoints */

   /* Check For 1D Map */

   if(y_size != 0)
      {
      y_offset = y_size - 1 ;

      mp = map_base + (unsigned char)(x_size + y_offset) ;

      while ((y_value < *mp) && (y_offset != 0))
         {
         y_offset ;
         mp ;
         }

      /* Extract Upper And Lower Y-Breakpoints */

      y_break_point1 = mp[0] ;
      y_break_point2 = mp[1] ;

      if((y_offset == y_size - 1) || (y_value <= y_break_point1))
         {
         y_value = y_break_point1 ;
         }

       /* Get Map Values */
```

```
       map_base += x_size + y_size + x_size * y_offset + x_offset ;

       map_x1y1 = *(map_base)              ;
       map_x2y1 = *(map_base + 1)          ;
     /* Interpolate 2D Map Values                        */
     /* Defines used to remove need for function calling */

#define x map_x1y1
#define y map_x2y1
#define n x_temp1
#define d x_temp2

      y -= x ;
      if(!CY)
         {
         result_y1 = (unsigned char) (x + (unsigned char)(((unsigned int)(n * y))/d)) ;
         }
      else
         {
         result_y1 = (unsigned char) (x - (unsigned char)(((unsigned int)(n * -y))/d)) ;
         }

        map_x1y2 = *(map_base + x_size)     ;
        map_x2y2 = *(map_base + x_size + 1) ;

#undef x
#undef y

#define x map_x1y2
#define y map_x2y2

      y -= x ;
      if(!CY)
         {
         result_y2 = (unsigned char) (x + (unsigned char)(((unsigned int)(n * y))/d)) ;
         }
      else
         {
         result_y2 = (unsigned char) (x - (unsigned char)(((unsigned int)(n * -y))/d)) ;
         }

#undef x
#undef y
#undef n
#undef d

      y_temp2 = (y_break_point2 - y_break_point1) ;

      /* Prevent Divide By Zero */

      if(y_temp2 == 0) {
         y_temp2++ ;
         }

#define x result_y1
#define y result_y2
#define n (y_value - y_break_point1)
#define d y_temp2
      y -= x ;
      if(!CY)
         {
         result = (unsigned char) (x + (unsigned char)(((unsigned int)(n * y))/d)) ;
         }
      else
         {
         result = (unsigned char) (x - (unsigned char)(((unsigned int)(n * -y))/d)) ;
         }

     } /* End of 2D Section */
  else
    {
```

```
    /* 1D Interpolation Only */

    map_base = map_base + x_size + x_offset ;

    map_x1y1 = map_base[0] ;
    map_x2y1 = map_base[1] ;

#undef x
#undef y
#undef n
#undef d

#define x map_x1y1
#define y map_x2y1
#define n x_temp1
#define d x_temp2
      y -= x ;
      if(!CY)
         {
         result = (unsigned char) (x + (unsigned char)(((unsigned int)(n * y))/d)) ;
         }
      else
         {
         result = (unsigned char) (x - (unsigned char)(((unsigned int)(n * -y))/d)) ;
         }
    } /* End 1D Section */
     return result ;
   }
```

Here is the test harness used to drive it:

```
/*** Sine Conversion Map                      ***/
/* Converts integer angle into sine value, 0-255 */

/* (x_size,y_size,
    x_breakpoints,
    y_breakpoints,
    map_data)
*/

const unsigned char sine_table[] = {
07,00,
00,15,30,45,60,75,90,
00,66,127,180,220,246,255
} ;

/** Test Variables **/

   unsigned char input_x_val ;
   unsigned char input_y_val ;
   unsigned char sine_value      ;

/** Routine To Be Tested **/

   extern interp(unsigned char,
                 unsigned char,
                 unsigned char const *) ;

/** Global Variables **/

   unsigned int angle ;

/** Dummy Harness Program **/
   void main(void)
   {

   while(1)
      {
      for(angle = 0 ; angle < 0x100 ; angle++) {

         sine_value = interp(angle,0,sine_table) ;
```

```
            }
        }
    }
```

22 Appendix E Tile Hill Embedded C Style Guide

When writing C there are two types of guide available and programmers often confuse them. Firstly the style guide which will be described here. Style is the layout and making it look pretty, uniform and readable. It is not about a safe use of C as such. The other type of guide covers the safe use of C or a safe subset. There are many of these available. They are often industry specific however there is one which has escaped into widespread use. The MISRA-C guide (or to give it it's full title: - The **Motor Industry Software Reliability Association Guidelines For The Use Of The C Language In Vehicle Based Software)** MISRA-C is available from MIRA (and also from PhaedruS SystemS) It is a very readable set of rules that make sense for most C programming embedded or not.

The Style Guide- History

The Tile Hill style guide has been produced not because it is ***"The Best or Only Way"*** but because every time we mentioned time style guides we were asked if we had one. When we told people to search the net, look in libraries etc they usually said "Can you send us the one you use?". Well this is it, the one we use.

To get the folk law out of the way first: It is called the Tile Hill guide becausewhilst writing it I had to pass close to Tile Hill in Coventry UK and it is a play on the legendary ***Indian Hill Recommended C Style and Coding Standards*** from AT&T Bell labs. My copy is Version 6 dated 1990. I do not claim that the Tile Hill Guide is better or replaced the Indian Hill document. I just produced my own guide to the style I use because people asked for one.

The Tile Hill Embedded C Style Guide is freely availavble seperately in electronic form from http://quest.phaedsys.org/

At the time of writing the Indian Hill guide was available from several places Also the NASA C sytle Guide from .
Jack Ganssel also has an embeded project guide available from http://www.ganssle.com/index.htm

The style guide rational

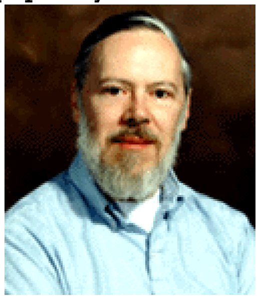
The idea behind the style guide and the reason they are misussed is free will. C is a free format language. One white space has the same wieght as 50 white spaces... This assumes that you are using black on white. For those of you useing DOS screesn it is black spaces (in the case of Jon lilac spaces) Effectivly this means that the programmer isd free to laout the source code as he or she sees fit. What is more there is no technical reason why the programmer has to keep their style constant.

Many programmers will complain that they can use their own style and anything else imposed is an infringement of civil liberties. Most Project manages have too many other things to worry about. As ling as it compiles they are happy.

The Tile Hill Embedded C Style guide explaines why a Style guide is a good idea… if not essential in embedded programming..

23 Apendix F A Standard History of C

The problem with C is its history. I do not propose to re-tell "The K&R Story" [K&R] here. However, there are some parts pertinent to this paper. I recommend that people read the paper by Dennis Ritchie [Ritchie] this is available from his web site: http://cm.bell-labs.com/cm/cs/who/dmr/index.html



C was developed initially (between 1969 and 1973) to fit into a space of 8K. Also C was designed in order to write an (portable) operating system. Unlike today, where disks and memory are inexpensive, at the time Multics was around operating systems had to take up as little space as possible, to leave room for applications on minimal memory systems. This makes it ideal for embedded systems.

C was developed from B and influenced by a group of several other languages. Interestingly BCPL, from which B was developed used // for comments just as C++ does and now finally C99!

One of the problems with C is that now the majority of people learn C in a Unix or PC environment with plenty of memory (real or virtual), disk space, native debugging tools and the luxury of a screen, keyboard and usually a multi-tasking environment.

Because C was originally designed for (compact) operating systems it can directly manipulate the hardware and memory addresses (not always in the way expected by the programmer). This can be very dangerous in normal systems let alone embedded ones!

C permits the user to do many "unorthodox" things. A prime example is to declare 2 arrays of 10 items A[10] and B[10]. Then "knowing" that (in the particular implementation in use) they are placed together in memory use the A reference "for speed" step from A[0] to A[19]. This is the sort of short cut that has got C a bad name. Yes, I have seen this done.

The syntax of C and its link with UNIX (famous for its terse commands) means that many programmers try to write C using the shortest and most compact methods possible. This has led to lines like:

```
while (l--) *d++ = *s++;
```

or

```
typedef boll (* func)(M *m);
```

This has given C the reputation for being a ***write only*** language and the domain of hackers.

As C was developed when computing was in its infancy and there were no guidelines for SW engineering. In the early days many techniques were tried that should by now have been buried. Unfortunately, some of them live on.

23.1 From K&R to ISO-C99 :- A Standard History of C

In the beginning in 197... Well it starts in the mists of legend... The best social/technical description I have seen is the paper **The Development of the C Language by** *Dennis M. Ritchie* it is (as of early 2001) available as a pdf http://cm.bell-labs.com/cm/cs/who/dmr/index.html. (If you have any problems finding it contact chris@phaedsys.org) This dates the beginnings of C as "about" 1969 to 1973 depending how you measure it. C evolved from B and BPCL when (modern) computing was only about 20 years old and microprocessors had yet to be invented. This paper is well worth reading as, in my view, it gives the best description of how it all came about (and why). Do not expect to learn C from this paper.

BTW Unix was so called because it was a ***Single User*** OS… a parody of Multics the multi user OS that they had. No, it was not run on a PDP11 first but a PDP7. Not a lot of people know that!

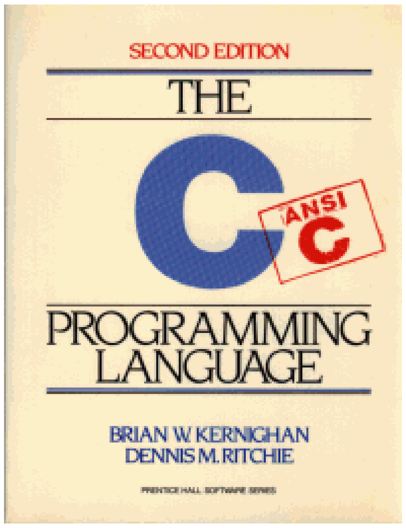
23.1.1 K&R (1st Edition) 1978

1978 saw the publication of The C Programming Language by Kernighan and Ritchie, thereafter known as "K&R". This was The Bible for all C programmers for over a decade. Unfortunately, many still cling to the faith despite the language changing a lot in the intervening 25 years. Even Dennis Richie said of K&R 1 "*Although it did not describe some additions that soon became common, this book served as the language reference until a formal standard was adopted more than ten years later."* See his paper cited above. This comment from one of the authors dents the mantra of many disciples that The Book is The Definitive Reference! K&R later published a new edition "K&R 2nd Ed" in line with ANSI C 1989.

Something else should be borne in mind when reading K&R 1st edition. It was written by experienced operating systems programmers for experienced UNIX programmers (by this time UNIX was a multi-user, multi-task OS). K&R is not, and never was; an introductory text on C for novices let alone 8 bit embedded systems programmers.

Having debunked K&R 1st Edition one should heed the commandment (found in some form in most faiths) "honour they parents." K&R 1st edition is the root of C and the source from which it all flowed. If you can find a copy (or K&R 2nd edition) buy one and dip into it but do not use it as a definitive reference or use it to teach people. Many (ten or twenty years ago) did learn from it, but then, it was the definitive (and only) work.

23.1.2 K&R (2nd edition 1988)



K&R 2nd edition gives the syntax changes and "improvements" in C over the decade since K&R1 and it brought K&R into line with the ANSI C 1989 standard. If you want a K&R for practical use this is the edition to have. You should remember it is not the definitive as from 1999. I expect there will not, despite public pressure, be a K&R3 as all the authors have moved on to new things in the last decade. (The authors previously have stated that there would *not* be a K&R3 but in early 2001 they left the door open…) Note the standard takes longer to ratify and publish that a book, which is why K&R (who were part of the US (ANSI) ISO committee anyway) got their book out ahead of the standard.

23.1.3 ANSI C (1989)

Eventually in 1989, due to the large number of people using C ANSI produced a USA standard that became the de-facto world wide standard until 1990. This stabilized the language and gave everyone (except Microsoft) a standard with which to conform.

23.1.4 ISO-C90 (1990)

ISO/IEC 9899 Programming Languages-C
At the end of 1989 ISO (and IEC) with all it's committees from many countries world wide adopted and ratified the US ANSI standard as an International Standard. From this point in theory, if not in practice ISO-C superceded ANSI C as the definitive standard. However, it should be noted that the only difference between ISO and ANSI C during the 1990's was the Chapter numbering. One of the standards had an additional chapter before the actual

standard throwing all the chapters out by one. Paragraph numbering was the same in both.

NOTE:- This version of ISO C is used for MISRA-C also for most embedded compilers as later "improvements" such as multi-byte characters and other changes for C99 were not needed (and in many cases not easy to implement) . At the time of writing , early 2003, there were still only two C99 compilers available.

ISO-C Amendment 1 1993
    Multi byte Characters

ISO-C Technical Corrigendum 1995/6 (T1)
    Work on the new standard starts.

Due to the fact that many things (eg MISRA-C) reference ISO C 90 the author has managed to persuade BSI (British Standards Institute) to make ISO C90 (with Amendment 1 and TC1) available again at a comparatively low price of £30 (about $45US).

23.1.5 ISO-C99 ISO/IEC 9899:1999

The ISO-C99 is now the definitive international work on the Language… It is not what I would call "readable" though. It was some months after the ISO-C99 was finished that ANSI (and all the other National Bodies around the world) adopted it.

A copy of ISO-C is a useful document to have (if only to win bets at lunchtime!) ISO-C99 ISO9899:1999 This can be obtained (correct as of early 2001) for $18 US as a PDF from :- www.techstreet.com/ncitsgate.html which is where I (and most of the UK standards panel) got my copy. It prints out to 537 pages. I printed it out, on a double-sided photocopier via the network on A4 double sided and it is quite usable.

The good news is, at the time of writing (November 2001) It is likely that a book publisher in partnership with the ACCU (see www.accu.org) will turn both the C and C++ standards in to books at around the £30 mark!

23.1.6 ISO/IEC 9899:1999 TC1 2001

The link leads to a seven page PDF document of 118062 bytes containing ISO/IEC 9899:1999 TECHNICAL CORRIGENDUM 1 Published 2001-09-01

http://ftp2.ansi.org/download/free_download.asp?document=ISO%2FIEC+9899%2FCor1%3A2001

The TC is freely available and the link abouve should download the PDF directly.

23.2 The Future: Back to C. (Why C is not C++)

The main problem at the moment is that as of November 2001 (nothing had changed by Jan 2003) no one has implemented a full C99 compiler for embedded use. There are dark mutterings in the embedded world that they may stay with C90.

Many people ask for C++ on small-embedded systems. What most people do not realize that whilst C++ was *developed from* C the two are now separate languages. In the early days C++ was a superset of C. This ceased to be true from the mid 1990's, both languages have moved on with slightly diverging paths.

C++ is being used on the desktop, 64, 32 and some 16 bit systems under UNIX, MS Windows and a variety of high end embedded RTOS. 2000 saw the start of some embedded C++ for 16 bit systems but that is as far as it will go. The use of C on the desktop has declined and the majority use is now in embedded systems often without an RTOS. After I wrote this some while ago I have been corrected that many compiler writers and systems writers also use C.

In late 2000 the ISO C committee was getting more work packages to do with embedded C and things to help the conversion of mathematical Fortran users to C. It was at this point the editor of this work took over as Convener of the UK ISO C committee.

The next round of work was meant to help the embedded user and will move C further from C++. Unfortunately the amendments were mainly in the form of DSP math and extensions only of use to 32-bit embedded systems with lots of space. There was a lot of discussion in 2003 with some violent disagreements of the direction C should take.

New features in C++ that might once have been put into C are less likely to happen. Partly because there are more embedded people involved and there are fewer desktop people involved. The other reason is that some C++ is not possible in 8 bit systems. There are some C++ compilers for small systems but the are not widely used and have some severe restrictions. Their use is dictated more by fashion than engineering reasons. In fact even C++ is being restricted as EC++ for embedded use. For embedded C++ see:- http://www.caravan.net/ec2plus/ Where you can get the Embedded C++ "standard" as supported by many compiler manufacturers. This was an initiative started in Japan that has speard world wide.

The other major thing the (UK) standards panel is trying to do is stabilize and iron out the ambiguities of the C99 standard. The ambiguities are one of the reasons that, two (now three) years later, no-one had managed to do a fully implemented C99 Embedded C compiler. Actually I don't think there is a full C99 compiler for any use. As of the summer of 2002 a couple of compilers had managed it but no mainstream industrial embedded compilers vendors even thought about it.

Where next for the C standard? Judging from history, you should expect preparation of the next revision of the C standard to begin around 2004. Most likely we will ask for feedback on an early draft around 2007. In between those times is the best time to provide constructive input, but be warned that unsolicited proposals without an active champion participating in the committee are unlikely to get very far. If you really want to work on substantial improvements, it would be wise to join the committee (via your National Body) well in advance, so you can gain a feel for how the group dynamics work. If you want to get involved please email me at chris@phaedsys.org

23.3 What to read for Embedded C?

There are thousands of C books out there… Few are *really* good. Most are for the desktop (MS & MAC) and Unix. For a good source of book recommendations try the ACCU at www.accu.org. They have independent reviews of over 2000 C, C++ and SW engineering books. And an embedded section at :-
http://www.accu.org/bookreviews/public/reviews/0sb/embedded_systems.htm

They do not sell books so the reviews are completely independent and written by working Engineers. There is one infamous review that starts "I did not pay money for this book and I would suggest that no one else should either…"

There is a list of books in the reference appendix. However, remember: Most C books are written for the desktop programmer not for embedded systems. I would still get K&R 1st edition as a *historical reference* but not as a first C book to learn from. I have an ISO C standard but that is not a book to learn from either! You do not buy a dictionary to learn how to write novels.

One thing to be wary of is that if the book is written by an academic it is likely to have been written for his course… It may well refer to development boards and other equipment made by him at the university and not generally available. Also it may assume you are doing or have done other courses and modules in the collage and therefore miss out useful information because you will get it on the other course. Not all books written by academics are like this but do take care when buying.

Incidentally if anyone wants the ISO C99 standard the best place to get it from is a US web site www.techstreet.com/ncitsgate.html

Note:- At the time of writing (summer 2003) BSI were looking at publishing the ISO C and C++ standards at £30 printed but loose leaf.

Stop press: PhaedruS SystemS are now able to offer the C90 standard, on which most embedded compilers, and MISRA-C are based . As of October 2005 PhaedruS SystemS was offereing a bundle of C90 and MISRA-C for 50 Uk Pounds

24 Appendix G Timers & Delays

I will get around to doing this shortly…

Appendix H Serial Ports and Baud rates

From day one all 8051's have had (at least) one serial UART. Note this is not RS232 but serial. It can, with the right hardware drivers be RS232, RS485 (current loop) or any one of several serial formats. In this case we are going to look at the basic set up for serial communications. Once you have this running you can communicate with your application.

The first step is to calculate the baud rate. This can be any rate you like but the commonly used standard settings are:- 110, 300, 600, 1200, 2400, 4800, 6900, 19200, 38400, 56700 and 115200.

To generate the pulses a timer must be used, except in a few cases where there is a baud rate generator peripheral provided. T1 is commonly used for baud rate generation. There are several modes for timers but the most common way of setting up serial communuitcations is to use Timer 1 in 8 bit auto-re-load mode.

That is an 8 bit number is used to generate the delay and it automatically rests and goes round again. The 8 bit number is stored in register TH1. The equation to find the value for TH1 is

$$TH1 = 256 - \frac{(xtal/Constant)}{baud\text{-}rate}$$

If SMOD = 0 then Constant = 384
If SMOD = 1 then Constant = 192

Now the problem is that TH1 can only hold 8 bit intergers. So, for example 5Mhz and 2400 with SMOD = 0 would give:-

$$TH1 = 256 - \frac{(5{,}000{,}000/384)}{2400}$$

This works out to

$$TH1 = 256 - 5.425347$$

Or

$$TH1 = 250.57466$$

You would, therefore, have to use 250 or 251 (0xFA or 0xFB) for TH1. Neither of theses is an exact match. The trick is to work the equation back using:

$$Baudrate = \frac{(xtal/constant)}{256-TH1}$$

In this case we get

$$Baudrate = \frac{(5{,}000{,}000/384)}{256-250} \quad \text{and} \quad Baudrate = \frac{(5{,}000{,}000/384)}{256-251}$$

This gives baudrate of 2170 or 2604

Neither is 2400 and is out a fair amount Keil recommend that the rate should be within 2%

A couple of other useful equations are:-

The maximum baud rate possible for a crystal

$$\text{Baud} = \frac{\text{Crystal/constant}}{256}$$

and the minimum crystal needed for a given baud rate

Min Crystal = Baudrate / constant

The constant is 384 for SMOD = 1
192 for SMOD = 0

There is more on baud rate calculation at : http://www.keil.com/support/docs/689.htm and Keil have an on-line calculator at: http://www.keil.com/c51/baudrate.asp

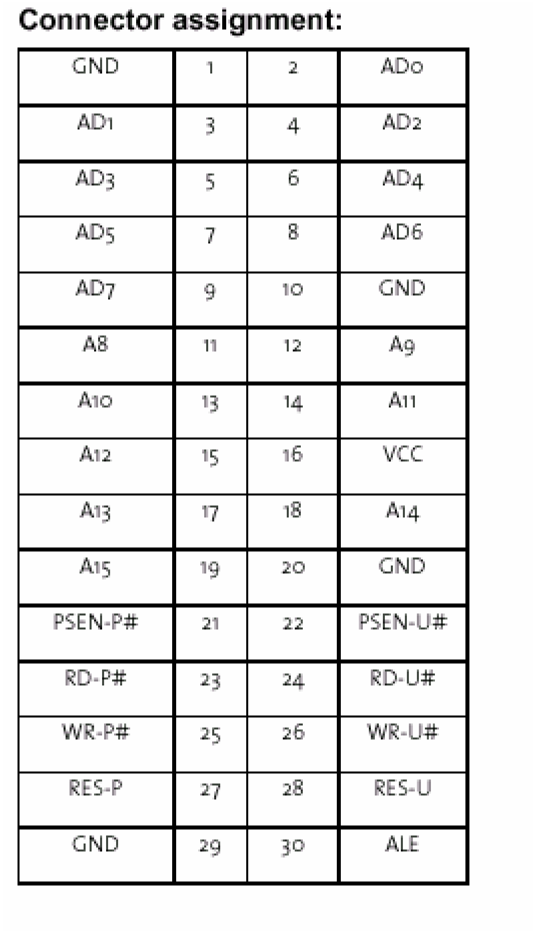
The table below shows part of the pattern for TH1 This shows why not all baud rates are possible at all frequencies.

baud Rate		110	300	600	1200	2400	4800	9600	19200	38400	56700	115200
Xtal	SMOD											
1.8432	0	D4	F0	F8	FC	FE	FF	-	-	-	-	-
1.8432	1	AH	E0	F0	F8	FC	FE	FF	-	-	-	-
2.0000	0	D1	EF	F7	-	-	-	-	-	-	-	-
2.0000	1	A1	DD	EF	F7	-	-	-	-	-	-	-
3.6864	0	A9	E0	F0	F8	FC	FE	FF	-	-	-	-
3.6864	1	51	C0	E0	F0	F8	FC	FE	FF			
4.0000	0	A1	DD	EF	F7	-	-	-	-	-	-	-
4.0000	1	43	BB	DD	EF	7F	-	-	-	-	-	-
4.096	0	9F	DC	EE	7F	-	-	-	-	-	-	-
4.096	1	3E	B9	DC	EE	7F	-	-	-	-	-	-
5.0000	0	8A	D5	EA	F5	-	-	-	-	-	-	-
5.0000	1	13	A9	D5	EA	F5	-	-	-	-	-	-
5.5296	0	7D	D0	E8	F4	FA	FD					
5.5296	1	-	A0	D0	E8	F4	FA	FD	-	-	-	-
6.0000	0	72	CC	E6	F3	-	-	-	-	-	-	-

25 Appendix J ICE Connect your design

If you had a simple, virtually overhead free, method of being able to put an ICE on to any of your prototype or production boards you would use it, wouldn't you? There is such a system. It will work with any 8051 design that has ports 0 and 2 available as address and data lines. This does not mean that there must be external memory just that porst 0 and 2 are not used for signals.

The method is ICEConnect a simple 2 way by 15 set of pads that are required on the PCB. These are the 8 multiplexed address/data lines the upper 16 bits of address line, PSEN, ReaD, WRite and ReSeT.



Connector assignment:

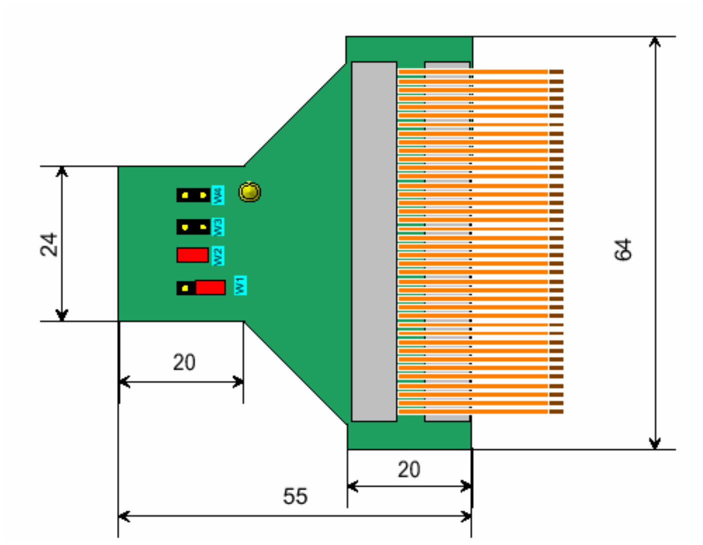
GND	1	2	AD0
AD1	3	4	AD2
AD3	5	6	AD4
AD5	7	8	AD6
AD7	9	10	GND
A8	11	12	A9
A10	13	14	A11
A12	15	16	VCC
A13	17	18	A14
A15	19	20	GND
PSEN-P#	21	22	PSEN-U#
RD-P#	23	24	RD-U#
WR-P#	25	26	WR-U#
RES-P	27	28	RES-U
GND	29	30	ALE

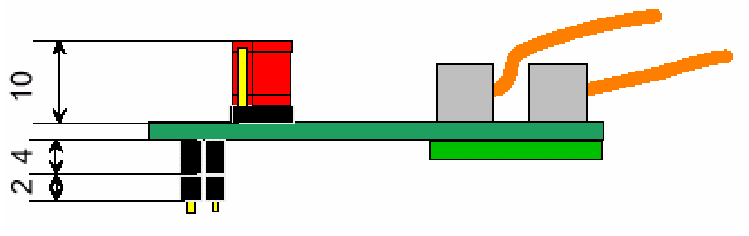
This method only requires tobe able to monitor the AD0-7 and D8-15. It does however require control of Read, Write, PSEN and Reset. These lines have to be redirected.

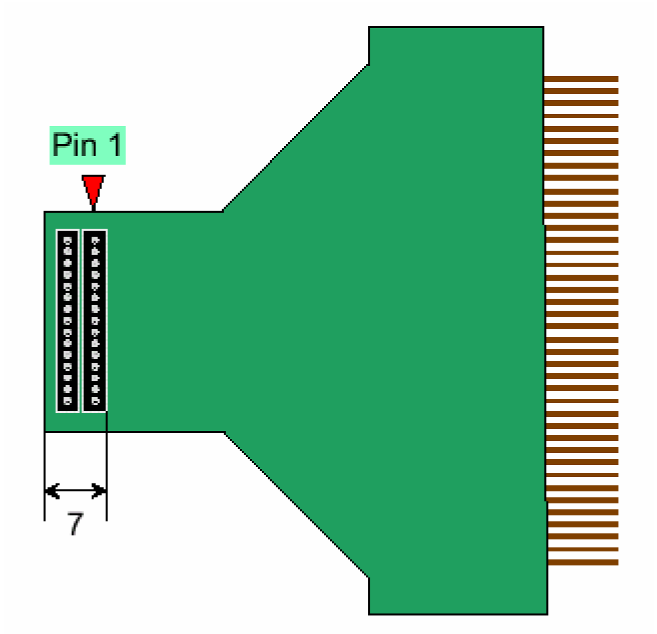
This is systems works well with most 8051's where there is not ICE or in the case of high integrity systems that have to test the code and production part in situ on a production board.

This is system is used in all maner of racing cars, medical equipment and aerospace systems.

As you can see there is no reason not to automatically design in the ICEConnect system into any 8051 system that can use it. There is also a demultiplexed version that is used with some ASICS.


24
20
55
20
64
W4
W3
W2
W1


10
2
4


Pin 1
7

26 Appendix K 8051 Instruction set (in Hex order)

Code	Bytes	Cycles	Mnemonic
00	1	1	NOP
01	2	2	AJMP codeaddr
02	3	2	LJMP codeaddr
03	1	1	RR A
04	1	1	INC A
05	2	1	INC dataAddr
06	1	1	INC @R0
07	1	1	INC @R1
08	1	1	INC R0
09	1	1	INC R1
0A	1	1	INC R2
0B	1	1	INC R3
0C	1	1	INC R4
0D	1	1	INC R5
0E	1	1	INC R6
0F	1	1	INC R7

Code	Bytes	Cycles	Mnemonic
10	3	2	JBC bitAdr, codeAddr
11	2	2	ACALL codeAddr
12	3	2	LCALL codeAddr
13	1	1	RRC A
14	1	1	DEC A
15	2	1	DEC dataAddr
16	1	1	DEC @R0
17	1	1	DEC @R1
18	1	1	DEC R0
19	1	1	DEC R1
1A	1	1	DEC R2
1B	1	1	DEC R3
1C	1	1	DEC R4
1D	1	1	DEC R5
1E	1	1	DEC R6
1F	1	1	DEC R7

Code	Bytes	Cycles	Mnemonic
20	3	2	JB bitAdr, codeAddr
21	2	2	AJMP codeAddr
22	1	2	RET
23	1	1	RL A
24	2	1	ADD A, #value
25	2	1	ADD A, dataAddr
26	1	1	ADD A, @R0
27	1	1	ADD A, @R1
28	1	1	ADD A, R0
29	1	1	ADD A, R1
2A	1	1	ADD A, R2
2B	1	1	ADD A, R3
2C	1	1	ADD A, R4
2D	1	1	ADD A, R5
2E	1	1	ADD A, R6
2F	1	1	ADD A, R7

Code	Bytes	Cycles	Mnemonic
30	3	2	JNB bitAdr, codeAdr
31	2	2	ACALL codeAdr
32	1	2	RETI
33	1	1	RLC A
34	2	1	ADDC A, #wert
35	2	1	ADDC A, dataAdr
36	1	1	ADDC A, @R0
37	1	1	ADDC A, @R1
38	1	1	ADDC A, R0
39	1	1	ADDC A, R1
3A	1	1	ADDC A, R2
3B	1	1	ADDC A, R3
3C	1	1	ADDC A, R4
3D	1	1	ADDC A, R5
3E	1	1	ADDC A, R6
3F	1	1	ADDC A, R7

Code	Bytes	Cycles	Mnemonic
40	2	2	JC codeAdr
41	2	2	AJMP codeAdr
42	2	1	ORL dataAdr, A
43	3	2	ORL dataAdr, #value
44	2	1	ORL A, #wert
45	2	1	ORL A, dataAdr
46	1	1	ORL A, @R0
47	1	1	ORL A, @R1
48	1	1	ORL A, R0
49	1	1	ORL A, R1
4A	1	1	ORL A, R2
4B	1	1	ORL A, R3
4C	1	1	ORL A, R4
4D	1	1	ORL A, R5
4E	1	1	ORL A, R6
4F	1	1	ORL A, R

Code	Bytes	Cycles	Mnemonic
50	2	2	JNC codeAdr
51	2	2	ACALL codeAdr
52	2	1	ANL dataAdr, A
53	3	2	ANL dataAdr, #value
54	2	1	ANL A, #value
55	2	1	ANL A, dataAdr
56	1	1	ANL A, @R0
57	1	1	ANL A, @R1
58	1	1	ANL A, R0
59	1	1	ANL A, R1
5A	1	1	ANL A, R2
5B	1	1	ANL A, R3
5C	1	1	ANL A, R4
5D	1	1	ANL A, R5
5E	1	1	ANL A, R6
5F	1	1	ANL A, R7

Code	Bytes	Cycles	Mnemonic
60	2	2	JZ codeAddr
61	2	2	AJMP codeAddr
62	2	1	XRL dataAdr, A
63	3	2	XRL dataAdr, #value
64	2	1	XRL A, #value
65	2	1	XRL A, dataAddr
66	1	1	XRL A, @R0
67	1	1	XRL A, @R1
68	1	1	XRL A, R0
69	1	1	XRL A, R1
6A	1	1	XRL A, R2
6B	1	1	XRL A, R3
6C	1	1	XRL A, R4
6D	1	1	XRL A, R5
6E	1	1	XRL A, R6
6F	1	1	XRL A, R7

Code	Bytes	Cycles	Mnemonic
T0	2	2	JNZ codeAddr
71	2	2	ACALL codeAddr
72	2	2	ORL C, bitAddr
73	1	2	JMP @A+DPTR
74	2	1	MOV A, #value
75	3	2	MOV dataAddr,#value
76	2	1	MOV @R0, #value
77	2	1	MOV @R1, #value
78	2	1	MOV R0, #value
79	2	1	MOV R1, #value
7A	2	1	MOV R2, #value
7B	2	1	MOV R3, #value
7C	2	1	MOV R4, #value
7D	2	1	MOV R5, #value
7E	2	1	MOV R6, #value
7F	2	1	MOV R7, #value

Code	Bytes	Cycles	Mnemonic
80	2	2	SJMP codeAddr
81	2	2	AJMP codeAddr
82	2	2	ANL C, bitAddr
83	1	2	MOVC A, @A+PC
84	1	4	DIV AB
85	3	2	**MOV dataAddr, dataAddr**
86	2	2	MOV dataAddr, @R0
87	2	2	MOV dataAddr, @R1
88	2	2	MOV dataAddr, R0
89	2	2	MOV dataAddr, R1
8A	2	2	MOV dataAddr, R2
8B	2	2	MOV dataAddr, R3
8C	2	2	MOV dataAddr, R4
8D	2	2	MOV dataAddr, R5
8E	2	2	MOV dataAddr, R6
8F	2	2	MOV dataAddr, R7

Code	Bytes	Cycles	Mnemonic
90	3	2	MOV DPTR, #value16
91	2	2	ACALL codeAdr
92	2	2	MOV bitAdr, C
93	1	2	MOVC A, @A+DPTR
94	2	1	SUBB A, #value
95	2	1	SUBB A, dataAddr
96	1	1	SUBB A, @R0
97	1	1	SUBB A, @R1
98	1	1	SUBB A, R0
99	1	1	SUBB A, R1
9A	1	1	SUBB A, R2
9B	1	1	SUBB A, R3
9C	1	1	SUBB A, R4
9D	1	1	SUBB A, R5
9E	1	1	SUBB A, R6
9F	1	1	SUBB A, R7

Code	Bytes	Cycles	Mnemonic
A0	2	2	ORL C, /bitAddr
A1	2	2	AJMP codeAddr
A2	2	1	MOV C, bitAddr
A3	1	2	INC DPTR
A4	1	4	MUL AB
A5	-	-	reserved (see 251) Also used by Chip con CC01010 for TRAP
A6	2	2	MOV @R0, dataAddr
A7	2	2	MOV @R1, dataAddr
A8	2	2	MOV R0, dataAddr
A9	2	2	MOV R1, dataAddr
AA	2	2	MOV R2, dataAddr
AB	2	2	MOV R3, dataAddr
AC	2	2	MOV R4, dataAddr
AD	2	2	MOV R5, dataAddr
AE	2	2	MOV R6, dataAddr
AF	2	2	MOV R7, dataAddr

Code	Bytes	Cycles	Mnemonic

B0	2	2	ANL C, /bitAddr
B1	2	2	ACALL codeAddr
B2	2	1	CPL bitAddr
B3	1	1	CPL C
B4	3	2	**CJNE A, #value, codeAddr**
B5	3	2	**CJNE A, dataAddr, codeAddr**
B6	3	2	**CJNE @R0, #value, codeAddr**
B7	3	2	**CJNE @R1,#value, codeAddr**
B8	3	2	CJNE R0, #value, codeAddr
B9	3	2	CJNE R1, #value, codeAddr
BA	3	2	CJNE R2, #value, codeAddr
BB	3	2	CJNE R3, #value, codeAddr
BC	3	2	CJNE R4, #value, codeAddr
BD	3	2	CJNE R5, #value, codeAddr
BE	3	2	CJNE R6, #value, codeAddr
BF	3	2	CJNE R7, #value, codeAddr

Code	Bytes	Cycles	Mnemonic
C0	2	2	PUSH dataAddr
C1	2	2	AJMP codeAddr
C2	2	1	CLR bit Addr
C3	1	1	CLR C
C4	1	1	SWAP A
C5	2	1	XCH A, data Addr
C6	1	1	XCH A, @R0
C7	1	1	XCH A, @R1
C8	1	1	XCH A, R0
C9	1	1	XCH A, R1
CA	1	1	XCH A, R2
CB	1	1	XCH A, R3
CC	1	1	XCH A, R4
CD	1	1	XCH A, R5
CE	1	1	XCH A, R6
CF	1	1	XCH A, R7

Code	Bytes	Cycles	Mnemonic
D0	2	2	POP dataAddr
D1	2	2	ACALL codeAddr
D2	2	1	SETB bitAddr
D3	1	1	SETB C
D4	1	1	DA A
D5	2	2	**DJNZ dataAddr, codeAddr**
D6	1	1	XCHD A, @R0
D7	1	1	XCHD A, @R1
D8	2	2	DJNZ R0, code Addr
D9	2	2	DJNZ R1, codeAddr
DA	2	2	DJNZ R2, codeAddr
DB	2	2	DJNZ R3, codeAddr
DC	2	2	DJNZ R4, codeAddr
DD	2	2	DJNZ R5, codeAddr
DE	2	2	DJNZ R6, codeAddr
DF	2	2	DJNZ R7, codeAddr

Code Bytes Cycles Mnemonic

E0	1	2	MOVX A, @DPTR
E1	2	2	AJMP codeAddr
E2	1	2	MOVX A, @R0
E3	1	2	MOVX A, @R1
E4	1	1	CLR A
E5	2	1	MOV A, data Addr
E6	1	1	MOV A, @R0
E7	1	1	MOV A, @R1
E8	1	1	MOV A, R0
E9	1	1	MOV A, R1
EA	1	1	MOV A, R2
EB	1	1	MOV A, R3
EC	1	1	MOV A, R4
ED	1	1	MOV A, R5
EE	1	1	MOV A, R6
EF	1	1	MOV A, R7

Code	Bytes	Cycles	Mnemonic
F0	1	2	MOVX @DPTR, A
F1	2	2	ACALL codeAddr
F2	1	2	MOVX @R0, A
F3	1	2	MOVX @R1, A
F4	1	1	CPL A
F5	2	1	MOV dataAddr, A
F6	1	1	MOV @R0, A
F7	1	1	MOV @R1, A
F8	1	1	MOV R0, A
F9	1	1	MOV R1, A
FA	1	1	MOV R2, A
FB	1	1	MOV R3, A
FC	1	1	MOV R4, A
FD	1	1	MOV R5, A
FE	1	1	MOV R6, A
FF	1	1	MOV R7, A

27 Appendix L Refferences

This is the full set of references used across the whole QuEST series. Not all the references are referred to in all of the QuEST papers. All of these books are in the authors own library and most have been reviewed for the ACCU. The reviews for these books and about 3000 others are on http://www.accu.org/bookreviews/public/

Andrews & Ince Practical Formal Methods with VDM, McGraw-Hill, 1991, ISBN 0--7-707214-6

Ball , Stuart. Debugging Embedded Microprocessor Systems, Newnes, 1998, ISBN 0-7506-9990-6

Ball , Stuart. Embedded Microprocessor Systems: Real world design 2nd Ed, Newnes, 2000, ISBN 0-7506-7234-X

Ball , Stuart. Analog Interfacing to Embedded Microprocessors Real world design, Newnes, 2001, ISBN 0-7506-7339-7

Baumgartner J Emulation Techniques, Hitex De internal paper, may 2001

Barr, Michael. Programming Embedded Systems in C and C++. O'Rilly, 1999, ISBN1-56592-354-5

Beach, M. Hitex *C51 Primer* 3rd Ed, Hitex UK, 1995, Beach, M. Hitex *C51 Primer* 3rd Ed, Hitex UK, 1995, http://www.hitex.co.uk (Draft 3.5 is on http://quest.phaedsys.org/)

Beach M, Embedding Software Quality Part 1, Hitex UK Available from www.Hitex.co.uk

Berger, Arnold. Embedded Systems Design: Anintroduction to Processes, Tools and Techniques. CMP Books, 2002, ISBN 1-57820-073-3

Black, Rex. Managing the Testing Process (2nd ed), Wiley, 2002, ISBN 0-471-22398-0

Bramer Brian & Susan, C for Engineers 2nd Ed, Arnold, 1997, ISBN 0-340-67769-4

Bramer Brian & Susan C++ for Engineers, Arnold, 1996 ISBN0-340-64584-9

Brooks, Fred. The Mythical Man Month: Essays On Software Engineering, Anniversary Edition. Addison Wesley, 1995 ISBN 0-201-83595-9

Brown John, Embedded Systems Programming In C and Assembley, VNR, 1994, ISBN 0-442-01817-7

Buchner F Embedding Software Quality Part 1, Hitex DE Available from www.Hitex.co.uk

Buchner F The Classification Tree Method, Internal paper: Hitex DE, 2002

Buchner F The Tessy article for the ESC II Brochure Hitex DE, 2002

Burden, Paul. Perilous Promotions and Crazy Conversions in C, PR Ltd, MISRA-C Conference 2002. http://www.programmingreasearch.com/

Burns & Wellings Real-Time Systems and Their Programming Languages, Addison Wesley, 1989, ISBN 0-201-17529-0

Chen Poon & Tse, Classification-tree restructuring methodologies: a new perspective IEE Procedings Software, Vol 149 no 2 April 2002 pp 65-74

Clements Alan, 68000 Family Assembly Language Pub PWS 1994

Coalman et al, Object-Orientated Development: The Fusion Method, Prentice-Hall, 1994, ISBN0-13-101040-9

Computer Weekly RAF JUSTICE :How the Royal Air Force blamed two dead pilots and covered up problems with the Chinook's computer system FADEC Computer Weekly 1997

Cooling J, Real-Time Software Systems ITC Press 1997 ISBN 1-85032-274-0

Cooling J. Software Design for Real time Systems ITC Press 1991 1-85032-279-1

COX B, Software ICs and Objective C, Interactive Programming Environments, McGraw Hill, 1984

Dasgupta, Subrata. Computer Architecture: A Modern Synthesis: Volume 1 Foundations, Wiley, 1989 , ISBN 0-471-61277-4

Dasgupta, Subrata. Computer Architecture: A Modern Synthesis: Volume 2 Advanced Topics, Wiley, 1989 , ISBN 0-471-61276-6

Defenbaugh & Smedley, C through Design, Franklin, Beedle & Associates, 1988, ISBN0-938661-10-8

Deitel, Harvey, Operating Systems, 2nd Ed Addison Wesley, 1990, ISBN 0-201-50939-3

Douglas BP Doing Hard Time, Developing Rea-Time Systems with UML, Addison Wesley, 1999, ISBN0-201-49837-5

Edwards, Keith. Real-Time Structured Methods: Systems Analysis, Wiley, 1993, ISBN 0-471-93415-1

Embley, Kurtz, Woodfield Object-Orientated Systems Analysis, Yourdon Press, 1992, ISBN 0-13-629973-3

Fenton et al, Software Quality Assurance and Measurement, A world wide Perspective, ITCP, 1995 ISBN1-85032-174-4

Fertuck, L, Systems Analysis and Design with CASe tools Pub WCB 1992

Gamma, Erich et al, Design Patterns: Elements of Reusable Object-Orientated Software, Addison Wesley, 1994, ISBN 0-201-63361-2

Gansel, Jack. The art of Programming Embedded Systems, Academic Press, 1992, ISBN 0-12,274880-8

Gansel Jack, The Embedded Muse Various editions. Pub Jack Gansel
http://www.ganssle.com/index.htm

Gerham, Moote & Cylaix, Real-Time Programming: A Guide to 32-bit Embedded Development, Addison Wesley, 1998, ISBN0-201-540-0

Goldberg & Rubin, Succeeding with Objects: Decision Frameworks for Project Management, Addison Wesley , 1995, ISBN 0-201-62878-3

Hatton Les, *Safer C:Developing Software for High-integrituy and Safety Critical Systems*, Mcgraw-Hill(1994) ISBN 0-07-707640-0

Heath , Steve, Microprocessor Architectures RISC, CISC & DSP 2nd ED, Butterworth-Heinemann 1995 ISBN 0-7506-2303-9

Heath , Steve, Embedded Systems Design, Newnes 1997 ISBN0-7506-3237-2

Hills C A, Embedded C: Traps and Pitfalls Chris Hills, Phaedrus Systems, September 1999, quest.phaedsys.org/

Hills C A, *Embedded Debuggers* –Chris Hills & Mike Beach, Hitex (UK) Ltd. April 1999 http://www.hitex.co.uk & quest.phaedsys.org

Hills C A, Tile Hill Style Guide Chris Hills, Phaedrus Systems, 2001, quest.phaedsys.org/

Hills CA & Beach M, Hitex, **SCIL-Level** A paper project managers, team leaders and Engineers on the classification of embedded projects and tools. Useful for getting accountants to spend money Download from www.scil-level.org

HMHO Home Office Reforming the Law on Involuntary Manslaughter : The governments Proposals www.homeoffice.gov.uk/consult/lcbill.pdf

Jacobson et al, Object-Orientated Software Engineering: A Use Case Driven Apporach, Addison-Wesely, 1992, ISBN 0-201-55435-0

Johnson S. C. Johnson, *'Lint, a Program Checker*,' in *Unix Programmer's Manual,* Seventh Edition, Vol. 2B, M. D. McIlroy and B. W. Kernighan, eds. AT&T Bell Laboratories: Murray Hill, NJ, 1979.

Jones A History of punched cards. Douglas W. Jones Associate Professor of Computer Science at the University of Iowa.
http://www.cs.uiowa.edu/~jones/cards/index.html
see also http://www.cwi.nl/~dik/english/codes/punched.html

Jones, Derek. The 7+/- 2 Urban Legend. MISRA-C Conference 2002.
http://www.knosof.co.uk/

Kaner, Bach & Pettichord, Lessons Learned in Software Testing, A Context Driven Approach. , Wiley, 2002 ISBN 0-471-08112-4

Kernighan Brian W & Pike , The Practice of Programming. Addison Wesley 1999 ISBN 0-201-61586-X

Kerzner, Harold. Project Management: A Systems Approach to Planning, Scheduling, and Controlling. (7th ed) Wiley, 2001. ISBN 0-471-39342-8

Koenig A *C Traps and Pitfalls*, Addison Wesley, 1989

K&R *The C programming Language* 2nd Ed., Prentice-Hall, 1988
Lyons. JL, Ariane 5: Flight 501 Failure. Report by the Enquiry Board , Ariane, 1996

Maric B, How to Misuse Code Coverage. Reliable Software Technologies, 1997. www.testing.com

Maguire, Steve. Writing Solid Code, Microsoft Press, 1993, ISBN1-55615-551-4

McConnell Steve, Code Complete, A handbook of Practical Software Construction. Microsoft Press, 1993, ISBN 1-55615-484-4

MISRA Guidelines For The Use of The C Language in Vehicle Based Software. 1998 From http://www.misra.org.uk/ and http://www.hitex.co.uk/

Morton, Stephen. Defining a "Safe Code" Development Process, Applied Dynamics International, 2001

Murphy, Nial. Front Panel: Designing Software for Embedded User Interfaces, R&D Books 1998 ISBN 0-87930-528-2

Oram & Talbot, Managing Projects with Make 2nd Ed , O'Reilly 1993 ISBN 0-937175-90-0

Parr, Andrew, Industrial Control Handbook 3rd Ed, Newnes, 1998, ISBN0-7506-3934-2

Pressman Software Engineering A Practitioners Approach. 3rd Ed McGrawHill 1992 ISBN 0-07-050814-3

PRQA Programming Research QA-C static analysis tool. www.programmingresearch.com

Randel, Brian. The Origins of Digital Computers, Springer Verlag 1973

Ritchie D. M. *The Development of the C Language* Bell Labs/Lucent Technologies Murray Hill, NJ 07974 USA 1993 available from his web site http://cm.bell-labs.com/cm/cs/who/dmr/index.html This is well worth reading.

Rumbaugh et al, Object Orientated Modelling and Design, Prentice Hall, 1991, ISBN 0-13-630054-5

Simon, David, An Embedded Software Primer, Addison Wesley,1999, ISBN 0-201-61569

Selis, Gullekson & Ward. Real-Time Object-Orientated Modeling, Wiley, 1994, ISBN 0-417-59917-4

Soligen & Berghout, The Goal/Question/Metric Method : A practical Guide for Quality Improvement of Software Development, McGraw-Hill, 1999, ISBN 0-07-709553-7

Sutter Ed. Embedded Systems: Firmware Demystified, CMP Books, 2002 ISBN 1-57820-09907

Vahid & Givargis Embedded System Design: A Unified Hardware/Software Introduction, Wiley, 2002, ISBN 0-471-38678-2

Van Vilet Software Engineering Principals and Practice Pub Wiley 1993 ISBN 0-471-93611-1BN 0-471-93611-1

Watkins, John. A Guide To Evaluating Software Testing Tools (V3) Rational Ltd 2001

Watson & McCabe, Structured Testing: A testing Methodology Using the Cyclomatic Complexity Model

Webster, Bruce. The Art of Ware, Sun Tzu's Classic Work Reinterpreted, M&T Books, 1995 ISBN 1-55851-396-5

Whitehead, Richard. Leading A Software Development Team: A Developers Guide to Successfully Leading People and Projects, Addison Wesley, 2001 ISBN 0-201-67526-9

Wilson, Graham.. Embedded Systems & Computer Architecture, Newnes, 2002, ISBN 0-7506-5064-8

Xie & Engler Using Redundancies to Find Errors, Computer Systems Laboratory
Stanford University: http://www.stanford.edu/~engler/p401-xie.pdf

28 Standards

This is the full set of standards used across the whole QuEST series. These are Standards as issued by recognised national or international Standards bodies. Note due to the authors position in the Standards Process some of the documents referred to are Committee Drafts or documents that are amendments to standards that may not have been made publicly available by the time this is read.

ISO

9899:1990 Programming Languages - C

9899:1999 Programming Languages - C

9899:-1999 TC1 Programming Languages-C Technical Corrigendum 1

9945 Portable Operating System Interface (POSIX)
9945-1 Base Definitions
9945-2 System Interfaces
9945-3 Shell and Utilities
9945-4 Rational

12207:1995 Information Technology- Software Life Cycle Processes

14764:1999 Information Technology - Software Maintenance

14882:1989 Programming Languages - C++

15288:2002 Systems Engineering - System Lifecycle Processes

JTC1/SC7 N2683 Systems Engineering Guide for ISO/IEC 15288

WDTR 18037.1 Programming languages, their environments and system software interfaces —Extensions for the programming language C to support embedded Processors

IEC

61508 :FCD Functional Safety or Electrical/Electronic/Programmable Electronic Safety - Relegated Systems

Part 1 General Requirements
Part 2 Requirements for Electrical/Electronic/Programmable
Electronic Safety -Relegated Systems
Part 3 Software Requirements
Part 4 Definitions and Abbreviations
Part 5 Examples of methods for the determination of SIL
Part 6 Guidelines for the application of parts 2 and 3
Part 7 Over View of Technical Measures

ISO/IEC JTC 1 N6981 Functional Safety and IEC61508: A basic Guide.

IEEE

You may be wondering where ANSI C is… ANSI C became ISO C 9899:1990 and ISO 9899 has been the International standard ever since. See "A Standard History of C" in Embedded C Traps and Pitfalls

1016-1998 Recommended Practice for Software Design Descriptions

5001:1999 The Nexus 5001 Forum™ Standard for a Global Embedded
Processor Debug Interface

NASA
http://sel.gsfc.nasa.gov/website/documents/online-doc.htm

SEL-81-305 Recommended Approach to Software Development Rev 3
SEL-84-101 Manager's Handbook for Software Development Rev 1
SEL-93-002 Cost And Schedule Estimation Study Report
SEL-94-003 C Style Guide August 1994 Goddard Space Flight Centre
SEL-94-005 An Overview Of The Software Engineering Laboratory
SEL-94-102 Software Measurement Guidebook Revision 1
SEL-95-102 Software Process Improvement Guidebook Revision 1
SEL-98-001 COTS Study Phase 1 Initial Characterization Study Report

OSEK

Network Management Concept and Application Programming Interface
Version 2.50 31st of May 1998

Operating System Version 2.1 revision 1 13. November 2000

OIL: OSEK Implementation Language Version 2.2 July 27th, 2000

Communication Version 2.2.2 18th December 2000

BCS

Standard For Software Component Testing Draft 3.3 1997

MOD Defence Standards

Def-Stan 00-13 REQUIREMENTS FOR THE ACHIEVEMENT OF TESTABILITY IN ELECTRONIC AND ALLIED EQUIPMENT

Def-Stan 00-17 MODULAR APPROACH TO SOFTWARE CONSTRUCTION, OPERATION AND TEST-MASCOT

Def-Stan 00-31 (obsolete) THE DEVELOPMENT OF SAFETY CRITICAL SOFTWARE FOR AIRBORNE SYSTEMS

Def-Stan 00-42 part 2 RELIABILITY AND MAINTAINABILITY ASSURANCE GUIDES PART 2: SOFTWARE

Def-Stan 00-54 Part 1 REQUIREMENTS FOR SAFETY RELATED ELECTRONIC HARDWARE IN DEFENCE EQUIPMENT PART 1: REQUIREMENTS

Def-Stan 00-54 part 2 REQUIREMENTS FOR SAFETY RELATED ELECTRONIC HARDWARE IN DEFENCE EQUIPMENT PART 2: GUIDANCE

Def-Stan 00-55 Part 1 REQUIREMENTS FOR SAFETY RELATED SOFTWARE IN DEFENCE EQUIPMENT PART 1: REQUIREMENTS

Def-Stan 00-55 Part 2 REQUIREMENTS FOR SAFETY RELATED SOFTWARE IN DEFENCE EQUIPMENT PART 2: GUIDANCE

Def-Stan 00-56 Part 1 SAFETY MANAGEMENT REQUIREMENTS FOR DEFENCE SYSTEMS PART 1: REQUIREMENTS

Def-Stan 00-56 part 2 SAFETY MANAGEMENT REQUIREMENTS FOR DEFENCE SYSTEMS PART 2: GUIDANCE

Def-Stan 00-58 part 1 HAZOP Studies on Systems Containing Programmable Electronics Part 1 Requirements

Def-Stan 00-58 part 2 HAZOP Studies on Systems Containing Programmable Electronics Part 2 General Application Guidance

QuEST Series (see http://QuEST.phaedsys.org)

QuEST 0	**Design and Documentation for Embedded Systems**
QuEST 1	Embedded C Traps and Pitfalls
QuEST 2	**Embedded Debuggers**
QuEST 3	Advanced Embedded Testing For Fun
QuEST 4	C51 Primer

QA1	SCIL-Level
QA2	Tile Hill Embedded C Style Guide
QA3	QuEST-C
QA4	PC-Lint & DAC MISRA-C Compliance Matrix

PhaedruS SystemS

Chris@phaedsys.org
http://www.phaedsys.org/