

# C51 Primer

by Mike Beach, [Hitex \(UK\) Ltd.](#)

HTML version by  
Steffen Duffner, Irena & Olaf Pfeiffer

Last modified on 03/03/96.

[Click here for the Table of Contents](#)

---

## About The C51 Primer

If you've flicked through this publication, you may be left thinking that it is necessary to be an expert to produce workable programs with C51. Nothing could be further from the truth. It is perfectly possible to write real commercial programs with nothing more than a reasonable knowledge of the ANSI C language.

However, to get the maximum performance from the 8051, knowing a few tricks is very useful. This is particularly true if you are working on a very cost-sensitive project where needing a bigger RAM or EPROM can result in an unacceptable on-cost. After all, if cost was not a consideration, we would all be using 80C166s and 68000s!

Whilst the C51 Primer is really aimed at users of the [Keil C51 Compiler](#), it is applicable in part to other compilers. However, some compilers do not allow such low-level access and have fewer 8051-specific extensions. They are less likely to be used on projects where getting maximum performance is essential.

## 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 to assembler code.
- Use special versions like the 80C517 and 87C751.
- Use assembler tricks in C.
- Help the optimiser to produce the smallest, fastest code.

## The C51 Primer Will Not Help You

- Program in ANSI C - get a good reference like Kernighan & Ritchie.
- Write portable code - simply use the compiler without using any extensions.
- Set-up each and every on-chip peripheral on all of the 90 different 8051 variants! Some are, however, covered in the appendices.

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

---

# Contents

- [About The C51 Primer](#)
- [1 Introduction](#)
- [2 Keil C51 Compiler Basics - The 8051 Architecture](#)
  - [2.1 8051 Memory Configurations](#)
    - [2.1.1 Physical Location Of The Memory Spaces](#)
    - [2.1.2 Possible Memory Models](#)
    - [2.1.3 Choosing The Best Memory Configuration/Model](#)
    - [2.1.4 Setting The Memory Model - #Pragma Usage](#)
  - [2.2 Local Memory Model Specification](#)
    - [2.2.1 Overview](#)
    - [2.2.2 A Point To Watch In Multi-Model Programs](#)
- [3 Declaring Variables And Constants](#)
  - [3.1 Constants](#)
  - [3.2 Variables](#)
    - [3.2.1 Uninitialised Variables](#)

- [3.2.2 Initialised Variables](#)
  - [3.3 Watchdogs With Large Amounts Of Initialised Data](#)
  - [3.4 C51 Variables](#)
    - [3.4.1 Variable Types](#)
    - [3.4.2 Special Function Bits](#)
    - [3.4.3 Converting Between Types](#)
    - [3.4.4 A Non-ANSI Approach To Checking Data Type](#)
- [4 Program Structure And Layout](#)
  - [4.1 Modular Programming In C51](#)
  - [4.2 Accessibility Of Variables In Modular Programs](#)
  - [4.3 Building A Real Modular Program -](#)
    - [4.3.1 The Problem](#)
    - [4.3.2 Maintainable Inter-Module Links](#)
  - [4.4 Task Scheduling](#)
    - [4.4.1 8051 Applications Overview](#)
    - [4.4.2 Simple 8051 Systems](#)
    - [4.4.3 Simple Scheduling - A Partial Solution](#)
    - [4.4.4 A Pragmatic Approach](#)
- [5 C Language Extensions For 8051](#)
  - [5.1 Accessing 8051 On-Chip Peripherals](#)
  - [5.2 Interrupts](#)
    - [5.2.1 The Interrupt Function Type](#)
    - [5.2.2 Using C51 With Target Monitor Debuggers](#)
    - [5.2.3 Coping Interrupt Spacings Other Than 8](#)
    - [5.2.4 The Using Control](#)
  - [5.3 Interrupts, USING, Registerbanks, NOAREGS In C51](#)
    - [5.3.1 The Basic Interrupt Service Function Attribute](#)
    - [5.3.2 The absolute register addressing trick in detail](#)
    - [5.3.3 The USING Control](#)
    - [5.3.4 Notes on C51's "Stack Frame"](#)
    - [5.3.5 When To Use USING](#)
    - [5.3.6 The NOAREGS pragma](#)
    - [5.3.7 The REGISTERBANK Control Alternative To NOAREGS](#)
    - [5.3.8 Summary Of USING And REGISTERBANK](#)
    - [5.3.9 Reentrancy In C51 - The Final Solution](#)
    - [5.3.10 Summary Of Controls For Interrupt Functions](#)
    - [5.3.11 Reentrancy And Library Functions](#)
- [6 Pointers In C51](#)
  - [6.1 Using Pointers And Arrays In C51](#)
    - [6.1.1 Pointers In Assembler](#)
    - [6.1.2 Pointers In C51](#)
  - [6.2 Pointers To Absolute Addresses](#)
  - [6.3 Arrays And Pointers - Two Sides Of The Same Coin?](#)

- [6.3.1 Uninitialised Arrays](#)
  - [6.3.2 Initialised Arrays](#)
  - [6.3.3 Using Arrays](#)
  - [6.3.4 Summary Of Arrays And Pointers](#)
- [6.4 Structures](#)
  - [6.4.1 Why Use Structures?](#)
  - [6.4.2 Arrays Of Structures](#)
  - [6.4.3 Initialised Structures](#)
  - [6.4.4 Placing Structures At Absolute Addresses](#)
  - [6.4.5 Pointers To Structures](#)
  - [6.4.6 Passing Structure Pointers To Functions](#)
  - [6.4.7 Structure Pointers To Absolute Addresses](#)
- [6.5 Unions](#)
- [6.6 Generic Pointers](#)
- [6.7 Spaced Pointers In C51](#)
- [7 Accessing External Memory Mapped](#)
  - [7.1 The XBYTE And XWORD Macros](#)
  - [7.2 Initialised XDATA Pointers](#)
  - [7.3 Run Time xdata Pointers](#)
  - [7.4 The volatile Storage Class](#)
  - [7.5 Placing Variables At Specific Locations -](#)
  - [7.6 Excluding External Data Ranges From Specific](#)
- [8 Linking Issues And Stack Placement](#)
  - [8.1 Basic Use Of L51 Linker](#)
  - [8.2 Stack Placement](#)
  - [8.3 Using The Top 128 Bytes of the 8052 RAM](#)
  - [8.4 L51 Linker Data RAM Overlaying](#)
    - [8.4.1 Overlaying Principles](#)
    - [8.4.2 Impact Of Overlaying On Program Construction](#)
      - [8.4.2.1 Indirect Function Calls With Function Pointers](#)
      - [8.4.2.2 Indirectly called functions solution](#)
      - [8.4.2.3 Function Jump Table Warning \(Non-hazardous\)](#)
      - [8.4.2.4 Function Jump Table Warning Solution](#)
      - [8.4.2.5 Multiple Call To Segment Warning \(Hazardous\)](#)
      - [8.4.2.6 Multiple Call To Segment Solution](#)
    - [8.4.3 Overlaying Public Variables](#)
- [9 Other C51 Extensions](#)
  - [9.1 Special Function Bits](#)
  - [9.2 Support For 80C517/537 32-bit Maths Unit](#)
    - [9.2.1 The MDU - How To Use It](#)
    - [9.2.2 The 8 Datapointers](#)
    - [9.2.3 80C517 - Things To Be Aware Of](#)
  - [9.3 87C751 Support](#)
    - [9.3.1 87C751 - Steps To Take](#)

- [9.3.2 Integer Promotion](#)
- [10 Miscellaneous Points](#)
  - [10.1 Tying The C Program To The Restart Vector](#)
  - [10.2 Intrinsic Functions](#)
  - [10.3 EA Bit Control #pragma](#)
  - [10.4 16 Bit sfr Support](#)
  - [10.5 Function Level Optimisation](#)
  - [10.6 In-Line Functions In C51](#)
- [11 Some C51 Programming Tricks](#)
  - [11.1 Accessing R0 etc. directly from C51](#)
  - [11.2 Making Use Of Unused Interrupt Sources](#)
  - [11.3 Code Memory Device Switching](#)
  - [11.4 Simulating A Software Reset](#)
  - [11.5 The Compiler Preprocessor - #define](#)
- [12 C51 Library Functions](#)
  - [12.1 Library Function Calling](#)
  - [12.2 Memory-Model Specific Libraries](#)
- [13 Outputs From C51](#)
  - [13.1 Object Files](#)
  - [13.2 HEX Files For EPROM Blowing](#)
  - [13.3 Assembler Output](#)
- [14 Assembler Interfacing To C Programs](#)
  - [14.1 Assembler Function Example](#)
  - [14.2 Parameter Passing To Assembler Functions](#)
  - [14.3 Parameter Passing In Registers](#)
- [15 General Things To Be Aware Of](#)
  - [15.1](#)
  - [15.2](#)
  - [15.3](#)
  - [15.4](#)
  - [15.5](#)
  - [15.6](#)
  - [15.7 Floating Point Numbers](#)
- [16 Conclusion](#)

# 1 Introduction

C can be a rather terse and mystifying language. Widely quoted as being a high level language, C does indeed contain many such features like structured programming, defined procedure calling, parameter passing, powerful control structures etc.

However much of the power of C lies in its ability to combine simple, low-level commands into complicated high-level language-like functions and allow access to the actual bytes and words of the host processor. To a great extent then, C is a sort of universal assembly language. Most programmers who are familiar with C will have been used to writing programs within large machines running Unix or latterly MS-DOS. Even in the now cramped 640KB of MSDOS, considerable space is available so that the smallest variable in a program will be an int (16 bits). Most interfacing to the real world will be done via DOS Ints and function calls. Thus the actual C written is concerned only with the manipulation and processing of variables, strings, arrays etc.

Within the modern 8 bit microcontroller, however, the situation is somewhat different. Taking the 8051 as an example, the total program size can only occupy 4 or 8K and use only 128bytes of RAM. Ideally, real devices such as ports and special function registers must be addressed from C. Interrupts have to be serviced, which require vectors at absolute addresses. Special care must be taken with a routine's data memory allocation if over-writing of background loop data is to be avoided. One of the fundamentals of C is that parameters (input variables) are passed to a function (subroutine) and results returned to the caller via the stack. Thus a function can be called from both interrupts and the background without fear of its local data being overwritten (re-cutrancy).

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..

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 is endowed with a stack system which is really only capable of handling return addresses. With only 256 bytes of stack potentially available, it would not take too much function-calling and parameter-passing to use this up.

From this you might think that implementing a stack-intensive language like C on the 8051 would be impossible. Well, it very nearly has been!

While there have been compilers around for some years now that have given C to 8051 users, they have not been overly effective. Most have actually been adapted from generic compilers originally written for more powerful micros such as the 68000. 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. 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 all this overhead, the provision of banked switch expanded memory, controlled by I/O ports, becomes almost a necessity!

Therefore, with the 8051 in particular, the assembler approach to programming has been the only real alternative for small, time-critical systems.

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.

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. However the latest 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.

Other noteworthy considerations for C on a microcontroller are:

1. control of on and off-chip peripheral devices
2. servicing of interrupts
3. making the best use of limited instruction sets
4. supporting different ROM/RAM configurations
5. a very high level of optimisation to conserve code space
6. control of registerbank switching
7. support of enhanced or special family variants (87C751, 80C517 etc..).

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 C language ANSI standard specifically for 8051 processors.



## 2 Keil C51 Compiler Basics – The 8051

### Architecture

The Keil C51 compiler has been written to allow C programmers to get code running quickly on 8051 systems with little or no learning curve. However, to get the best from it, 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 such as K & R

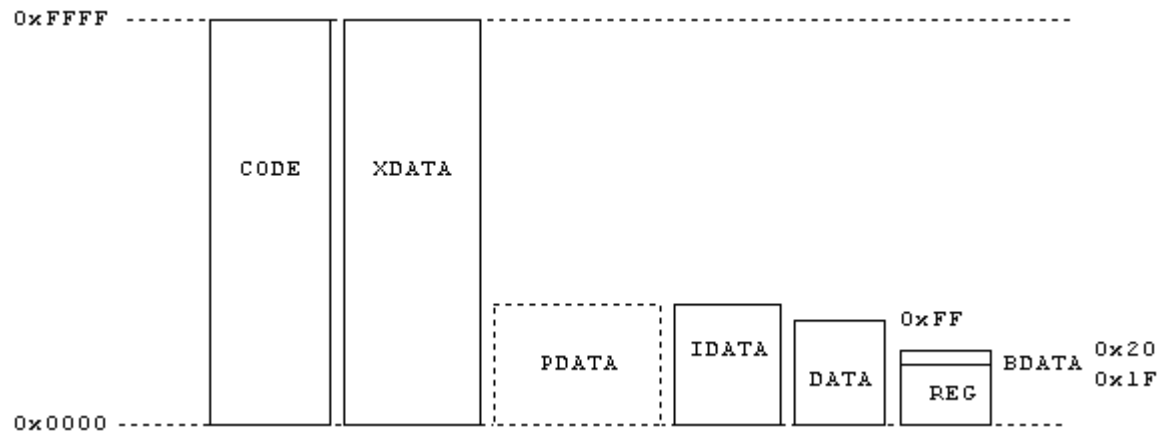
### 2.1 8051 Memory Configurations

#### 2.1.1 Physical Location Of The Memory Spaces

Perhaps the most initially confusing thing about the 8051 is that there are three different memory spaces, all of which start at the same address.

Other microcontrollers, such as the 68HC11, have a single Von Neuman memory configuration, where memory areas are located at sequential addresses, regardless of in what device they physically exist.

Within the CPU there is one such, the DATA on-chip RAM. This starts at D:00 (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. Above 80H the special function registers are located, which are again directly addressable. However, a second memory area exists between 80H and 0FFH which is only indirectly addressable and is prefixed by I: and known as IDATA. It is only accessible via indirect addressing (MOV A,@Ri) and effectively overlays the directly addressable sfr area. This constitutes an extended on-chip RAM area and was added to the ordinary 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 stack pointer SP. Just to confuse things, the normal directly addressable RAM from 0-80H can also be indirectly addressed by the MOV A,@Ri instruction!



*Fig. 1. - The 8051's Memory Spaces.*

A third memory space, the CODE segment, also starts at zero, but this is reserved for the program. It typically runs from C:0000 to C:0FFFFH (65536 bytes) but as it is held within an external EPROM, it can be any size up to 64KB (65536 bytes). The CODE segment is accessed via the program counter (PC) for opcode fetches and by DPTR for data. Obviously, being ROM, only constants can be stored here.

A fourth memory area is also off-chip, starting at X:0000. 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. Finally, 256 bytes of XDATA can also be addressed in a paged mode. Here an 8 bit register (R0) is used to access this area, termed PDATA.

The obvious question is: "How does the 8051 prevent an access to C:0000 resulting in data being fetched from D:00?"

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.

```
MOV    A, 40      ; Put value held in location 40 into the accumulator.
                    This addressing mode (direct) is 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 by the normal MOV instructions and addressed by special bit-orientated instructions, as shown below:

```
SETB  20.0  ;  
CLR   20.0  ;
```

The external EPROM device (C:0000) is not enabled during RAM access. In fact, the external EPROM is only enabled when a pin on the 8051 named the PSEN (program store enable) is pulled low. The name indicates that the main function of the EPROM is to hold the program.

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, whereas the CODE device only responds when the PSEN pin is low.

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

```
MOV    DPTR, #08000H  
MOVX   A, @DPTR      ; "Put a value in A located at address in the  
                      external RAM, contained in the DPTR register (8000H)".
```

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 instructions are being fetched; READ and WRITE are active when MOVX... ("move external") instructions are being carried-out.

Note that the 'X' means that the address is not within the 8051 but is contained in an external device, enabled by the READ and WRITE pins.

### 2.1.2 Possible Memory Models

With a microcontroller like the 8051, the first decision is which memory model to use. Whereas the PC programmer chooses between TINY, SMALL, MEDIUM, COMPACT, LARGE and HUGE to control how the processor segmentation

of the RAM is to be used (overcome!), the 8051 user has to decide where the program and data are to reside.

C51 currently supports the following memory configurations:

1. **ROM:** currently the largest single object file that can be produced is 64K, although up to 1MB can be supported with the BANKED model described below. All compiler output to be directed to Eprom/ROM, constants, look-up tables etc., should be declared as "code".
2. **RAM:** There are three memory models, SMALL, COMPACT and LARGE
3. **SMALL:** all variables and parameter-passing segments will be placed in the 8051's internal memory.
4. **COMPACT:** variables are stored in paged memory addressed by ports 0 and 2. Indirect addressing opcodes are used. On-chip registers are still used for locals and parameters.
5. **LARGE:** variables etc. are placed in external memory addressed by @DPTR. On-chip registers are still used for locals and parameters.
6. **BANKED:** Code can occupy up to 1MB by using either CPU port pins or memory-mapped latches to page memory above 0xFFFF. Within each 64KB memory block a COMMON area must be set aside for C library code. Inter-bank function calls are possible.

See the [section on BL51](#) for more information on the BANKED model.

A variation on these models is to use one model globally and then to force certain variables and data objects into other memory spaces.

This technique is covered later.

### 2.1.3 Choosing The Best Memory Configuration/Model

With the four memory models, a decision has to be made as to which one to use. 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!

**SMALL: Total RAM 128 bytes (8051/31)**

Rather restricting in the case of 8051/31. 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 emulators like the Hitex T51 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.

**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 stack usage is very high, 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.

**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 with SMALL. As with COMPACT, register variables are still used and so efficiency remains reasonable.

In summary, there are five memory spaces available for data storage, each of which has particular pros and cons.

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; Not model-dependant

*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 bytes

*Best For:*

Constants and large lookup tables, plus opcodes, of course!

*Worst For:*

Variables!

PDATA: 256bytes; 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; 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.

## **2.1.4 Setting The Memory Model - #Pragma Usage**

The overall memory type is selected by including the line `#pragma SMALL` as the first line in the C source file.

[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!

## 2.2 Local Memory Model Specification

### 2.2.1 Overview

C51 version 3.20 allows memory models to be assigned to individual functions. Within a single module, functions can be declared as SMALL, COMPACT or LARGE thus

```
#pragma COMPACT
/* A SMALL Model Function */
fsmall() small {
    printf("HELLO") ;
}
/* A LARGE Model Function */
flarge() large {
    printf("HELLO") ;
}
/* Caller */
main() {
    fsmall() ; // Call small func.
    flarge() ; // Call large func.
}
```

See pages 5-20 in the C51 reference manual for further details.

### 2.2.2 A 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 putchar() etc. from two different libraries.

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{
/*code*/
}
```

## 3 Declaring Variables And Constants

### 3.1 Constants

The most basic requirement when writing any program is to know how to allocate storage for program data. Constants are the simplest; these can reside in the code (Eprom) 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 Eprommed constant data are:

```
code unsigned char coolant_temp = 0x02 ;
code unsigned char look_up_table[5]='1','2','3','4' ;
code unsigned int pressure = 4 ;
```

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

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

```
/* Base FuelMap */

/* x = Load : y = engine speed : output = Injector PW, 0 – 8.16ms */

/* (x_size,y_size,
   x_breakpoints,
   y_breakpoints,
   map_data)
*/

code unsigned char 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, 0x09, 0x41, 0x80, 0x66, 0x66,
0x00, 0x00, 0x00, 0x00, 0x4D, 0x63, 0x66, 0x66,
0x00, 0x00, 0x00, 0x02, 0x4D, 0x63, 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 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](#)

## 3.2 Variables

### 3.2.1 Uninitialised Variables

Naturally, all variables exist in RAM, the configuration of which is given in [section 2.1.1](#).

The #pragma SMALL line will determine the overall memory model. In this case, all variables are placed within the on-chip RAM. However, specific variables can be forced elsewhere as follows:

```

#pragma SMALL
.
.
xdata unsigned char engine_speed ;
xdata char 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. This is a very important point – never forget that the 8051 has very limited on-chip RAM.

Another example is:

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

### 3.2.2 Initialised Variables

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.

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.

### 3.3 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.

```
_____  
This file is part of the C-51 Compiler package Copyright KEIL ELEKTRONIK GmbH 1990  
_____  
INIT.A51: This code is executed if the application program contains initialised  
variables at file level.  
_____  
; User-defined Watch-Dog Refresh.  
;
```

```
; If the C application containing many initialised variables uses a watchdog it
; might be possible that the user has to include a watchdog refresh in the
; initialisation process. The watchdog refresh routine can be included in the
; following MACRO and can alter all CPU registers except DPTR.
;
```

```
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
```

```
.
```

```
.
```

```
.
```

```
                XCH      A, R0
```

```
                XCH      A, R2
```

```
                XCH      A, DPH
```

```
                XCH      A, R2
```

```
                DJNZ     R7, XLoop
```

```
                DJNZ     R6, XLoop
```

```

S JMP    Loop
L JMP MAIN          ; C51 Program start

R SEG    ?C_INITSEG
DB      0
END

```

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.

## 3.4 C51 Variables

### 3.4.1 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
char	=8 bits	0 - +/- 127
unsigned char	=8 bits	0 - 255
int	=16 bits	0 - +/-32768
unsigned int	=16 bits	q0 - 65535
long	=32 bits	0 - +/- 2.147483648x10 <sup>9</sup>
unsigned long	=32 bits	0 - 4.29496795x10 <sup>9</sup>
float	=32 bits	+/-1.176E-38 to +/-3.4E+38
pointer	=24/16/8 bits	Variable address

Typical declarations would be:

```

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

```

(Note: bit variables are always placed in the bit-addressable memory area of the 8051 - [see section 2.1.1](#))

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, which does have an extended arithmetic instruction set. This has, 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. A suite of runtime libraries is available from Keil to allow the compiler to take advantage of the 80C537 enhancements.

### 3.4.2 Special Function Bits

A major frustration for assembler programmers coming to C is the inability of ANSI 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. In 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:

```
bdata char test ;  
sign_bit is defined as:  
sbit sign ^ 7 ;
```

To use this:

```
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
;
; bdata char test ;
; sbit sign = test ^ 7 ;
;
; void main(void) {
main:
;   test = -1 ;
    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 an awful 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.

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

### 3.4.3 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 ;
unsigned char y ;
unsigned char z ;
```

```
x = 10 ;
y = 5  ;
```

```
z = x * y ;
```

Results in z = 50

However:

```
x = 10 ;
y = 50 ;
```

```
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 ;
```

While C51 will automatically promote chars to int, it is best not to rely on it! It could be argued that on any small microcontroller you should always be aware of exactly what size data is.

#### 3.4.4 A Non-ANSI Approach To Checking Data Type

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 ANSI 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:
    ; SOURCE LINE # 8
;
;   if(((unsigned int)x + (unsigned int)y) > 0xff) {
    ; SOURCE LINE # 10
    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 ;    // ANSI C version
;                  ; SOURCE LINE # 12

    MOV     z, #0FFH
;      }
;                  ; SOURCE LINE # 13

```

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 ANSI C as a reference to the 8051 carry flag has been made.

```

?C0001:
;
;      z = x + y ;
;                  ; SOURCE LINE # 15
    MOV     A, x
    ADD     A, y
    MOV     z, A
;
;      if(CY) {
;                  ; SOURCE LINE # 17
    JNB     CY, ?C0003
;
;      z = 0xff ;    // C51 Version using the carry flag
;                  ; SOURCE LINE # 19
    MOV     z, #0FFH
;      }
;                  ; SOURCE LINE # 20
;
;
;
;
;      }
;                  ; SOURCE LINE # 25
?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. 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. Unfortunately, complete portability always compromises performance!

## 4 Program Structure And Layout

### 4.1 Modular Programming In C51

This is possibly not the place to make the case for modular programming, but a brief justification might be appropriate.

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, the tasks are the building blocks which are brought together to achieve the final objective.

The overall program thus has a loosely-predefined modular structure which could sensibly form the basis of the final software layout. The largest identifiable blocks within the program are the tasks. These are in turn built from modules, which themselves are constructed from functions in the case of C.

The modules are in reality individual source files, created with a text editor. Grouping the software sections together according to the function with which they are associated is the basis of modular programming.

Using the CEMS engine control system again as a real example, the task of running the engine is divided into the following tasks:

#### Task 1

Provide Timed Sparks For Ignition

#### Task 2

Provide controlled pulsewidths for fuel injection

#### Task 3

Allow alteration of tune parameters via terminal

Considering Task 1, this is in turn composed of modules thus:

Task 1, Module 1

Determine crank shaft position and speed

Task 1, Module 2

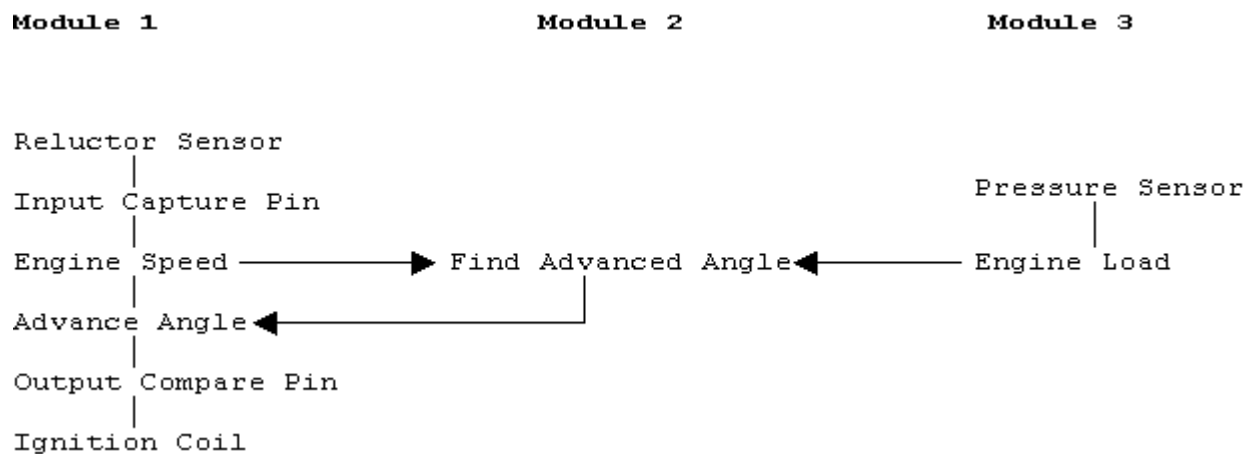
Measure engine load

Task 1, Module 3

Obtain required firing angle from look-up table

Taking module 2, a C function exists which uses an A/D converter to read a voltage from a sensor. It is part of the overall background loop and hence runs in a fixed sequence. In module 1 an interrupt function attached to an input capture pin calculates engine speed and generates the ignition coil firing pulse. Module 3 is another function in the background loop and takes speed and load information from the other modules constituting the ignition function, to calculate the firing angle. Obviously, data must be communicated from the data collecting functions to the processing functions and thence to the signal generation parts across module boundaries.

In this case, the data flows are thus:



Commonly, the variables used are declared in the module that first supplies them with data. Hence the engine\_load would be defined in Module 2 as that is where its input data comes from.

In this system the data would be declared thus:

**Module\_1.c**

**Module\_3.c**

**Module\_2.c**

```
/* Global Data Declaration */  
Declaration */
```

```
/* Global Data Declaration */
```

```
/* Global Data
```

```

unsigned char engine_speed      unsigned char advance      unsigned char
engine_load

/* External Data References */ /* External Data References */ /* External Data
References */

extern unsigned char advance    extern unsigned char engine_speed  extern unsigned
char engine_load

```

The most important thing to note is how the data defined in another module is referenced by redeclaring the required data item but prefixed with "extern".

Now, with a complete program spread across many different source files, the problem arises of how data is communicated between modules (files) and how separate C functions which lie outside of the home module may be accessed.

*The next section illustrates how the linkage between modules is undertaken.*

## 4.2 Accessibility Of Variables In Modular Programs

A typical C51 application will consist of possibly five functional blocks (modules) contained in five source files. Each block will contain a number of functions (subroutines) 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. As used to be done years ago in assembler, all variables (even the temporary ones) will be defined in one place and will remain accessible to every routine.

This approach is very inefficient and would seriously limit the power of C programs, as the internal RAM would soon be used up. The high-level language feature of a clearly defined input and output to each function would also be lost.

Similarly, an entire C program might be written within one single source file. As has been said, this practice was common many years ago with simple assemblers. Ultimately the source program can get so big that the 640K of a PC will get full and the compiler will stop. Worse than this, the ideal of breaking programs into small, understandable chunks is lost.



Programs then become a monolithic block and consume huge amounts of listing paper...

There should therefore be a hierarchical arrangement of variables and functions within a program; complete functional blocks should be identified and given their own individual source files or modules. Use should be made of the ability to access external variables and functions to achieve small program files!

The following should help explain:

```
MODULE1.c: *****
    unsigned char global1 ;    (1)
    unsigned char global2 ;
    extern unsigned char ext_function(unsigned char) ;    (2)

/* Utility Routine */
    int_function(x)    (3)
    unsigned char x ;    (4)
    {
        unsigned int temp1 ;    (5)
        unsigned char temp2 ;
        temp 1 = x * x ;
        temp2 = x + x ;

        x = temp1/temp2 ;

        return(x)    (6)
    }

/* Program Proper */
    main()    (7)
    {
        unsigned char local1 ;    (5)
        unsigned char local2 ;
        local2 = int_function(local1) ;    (8)
        local1 = ext_function(local2) ;    (9)

    }
end of MODULE1.c *****

MODULE2.c: *****
    extern unsigned char global1 ;    (10)
```

```

ext_function(y)
unsigned char y ;
{
unsigned char temp ;
static unsigned char special ;    (11)

special++ ;
y = temp * global1 ;    (12)

return(y) ;
)

```

**Line (1)** declares variables which will be accessible from all parts of the program. Ideally, such global usage should be avoided but where an interrupt has to update a value used by the background program, for example, they are essential.

**Line (2)** makes an external reference to a function not defined in the current module (block). This line allows all the functions in this MODULE to call the external function.

**Line (3)** declares a function which is to be used by another function in this module. These utility functions are placed above the calling function (here "main()").

**Line (4)** declares the variable which has been passed-over by the calling function. When the variable left "main()", it was called "local1". Within this function it is known simply as "x". The byte of ram is allocated to "x" only while the 8051's program counter is within this function. At the closing }, x will vanish.

**Line (5)** like "x" above, these variables are simply used as intermediate values within the function. They have no significance outside. Again, the byte of RAM will be re-assigned within another function. However the locals defined in "main()" will always exist as the C program is entirely contained within "main()".

**Line (6)** allows the result of the calculation to be passed back to the calling function. Once back in "main()" the value is placed in "local2".

**Line (7)** defines the start of the C program. Immediately prior to the point at which the program counter reaches main(), the assembler routine "STARTUP.A51" will have been executed. This in turn starts at location C:0000, the reset vector. Note that no parameters are passed to "main()".

**Line (8)** effectively calls the function defined above, passing the value "local1" to it.

**Line (9)** is like 8, but this time a function is being called which resides outside of the current module.

**Line(10)** links up with line(1) in that it makes "global1" visible to function within MODULE 2.

**Line(11)** declares a variable which is local to this function but which must not be destroyed having exited. Thus it behaves like a global except that no other function can use it. If it were placed above the function, accessibility would be extended to all functions in MODULE 2.

The physical linking of the data names and function names between modules is performed by the L51 linker. This is covered in detail [in section 8](#).

## 4.3 Building A Real Modular Program -

### The Practicalities Of Laying Out A C51 Program

The need for a modular approach to program construction has been outlined earlier. Here the practicalities of building easily maintainable and documentable software is given, along with a trick for easing the development of embedded C programs using popular compilers such as the Keil C51.

#### 4.3.1 The Problem

The simplest embedded C program might consist of just

```
/* Module Containing Serial Port Initialisation */ /* V24IN537.C */
void v24ini_537(void)
{

    /* Serial Port Initialisation Code */
}

/* Module Containing Main Program */ /* MAIN.C */
/* External Definitions */

extern void v24ini_537(void) ;

void main(void) {
```

```

v24ini_537() ;
while(1) {
    printf("Time = ") ;
}

```

This minimal program has only one purpose – to print an as yet incomplete message on the terminal attached to the serial port. Obviously, a single source file or “module” is sufficient to hold the entire C program.

Any real program will of course contain more functionality than just this. The natural reaction is to simply add further code to the existing main function, followed by additional functions to the MAIN.C source file. Unless action is taken the program will consist of one enormous source file, containing dozens of functions and interrupts and maybe hundreds of public variables.

Whilst compilers will still compile the file, the compilation time can become greatly extended, meaning that even the smallest modification requires the entire program to be re-compiled. A monolithic program is usually symptomatic of a lack of proper program planning and is likely to contain suspect and difficult to maintain code.

The next stage in the sample program development is to add some means of generating the time thus:

```

/* Module Containing Timer0 Initialisation */
/* T0INI537.C */

void timer0_init_537(void) {
    /* Enable Timer 0 Ext0 interrupts */
} /*init_timer_0*/

/* Module Containing Timer0 Service Routine */
/* RLT_INT.C */
/* Local Data Declarations */
/* Clock Structure Template */

struct time { unsigned char msec ;
              unsigned char sec  ; } ;

/* Create XDATA Structure */

struct time xdata clock ;
bit clock_run_fl = 0 ; // Flag to tell timer0 interrupt
                      // to stop clock

```

```

/* External References */

extern bit clock_reset_fl // Flag to tell timer0 interrupt
// to reset clock to zero

/**/ INTERRUPT SERVICE FOR TIMER 0 /**/
void timer0_int(void) interrupt 1 using 1 {
    if(clock.msec++ == 1000) {
        clock.sec++ ;
        if(clock.sec == 60) {
            clock_sec = 0 ;
        }
    }
}

```

To make this 4 module program useful, the main loop needs to be altered to:

```

/* Module Containing Main Program */
/* MAIN.C */

#include <reg517.h>

/* External Definitions */

extern void v24ini_537(void) ;
extern void timer0_init_537(void) ;

/* General Clock Structure Template */

struct time { unsigned char secs ;
              unsigned char msec ; } ;

/* Reference XDATA Structure In Another Module */

extern struct time xdata clock ; extern bit clock_reset_fl // Flag to tell timer0
interrupt to reset clock to zero
/* Local Data Declaration */
bit clock_run_fl ; // Flag to tell timer0 interrupt
// to stop clock

void main(void) {
    v24ini_537() ;
    timer0_init_537() ;
    while(1) {

```

```

        printf("Time = %d:%d:%d:%d", clock.hours,
               clock.mins,
               clock.secs,
               clock.msecs) ;
    }
    if(P1 | 0x01) {
        clock_run_fl = 1 ; // If button pressed start clock
    }
    else {
        clock_run_fl = 0 ; // If button released stop clock
    }
    if(P1 | 0x02) {
        clock_reset_fl = 1 ; // If button pressed clear clock
    }
}

```

#### 4.3.2 Maintainable Inter-Module Links

The foregoing program has been constructed in a modular fashion with each major functional block in a separate module (file). 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 – the module containing the data declaration plus any other module which makes a reference to the additional items. With long and meaningful names common in C and complex memory space qualification widespread in C51, much time can be wasted in getting external references to match at the linking stage. Simple typographic errors can waste huge amounts of time!

In large programs with many functions and global variables, the global area preceding the executable code can get very untidy and cumbersome. Of course, there is an argument that says that having to add external references to the top of a module when first using a new piece of global data is good practice, as it means that you are always aware of exactly which items are used. It is preferable to the common approach of having a single include 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.

This latter method inevitably leads to the undesirable situation where an original data declaration in the source module is sitting alongside its external reference in the general include file.

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

auxilliary file contains both original declarations and function prototypes plus the external references. It is therefore similar in concept to the standard library .h files used in every C compiler. The trick is, however, to use conditional compilation to prevent the original declarations and the external versions being seen simultaneously.

When included in their home modules, i.e. the ".c" file having the same root, only the original declarations are seen by C51 whereas, when included in a foreign module, only the external form is seen. To achieve this apparent intelligence, each source module must somehow identify itself to the include file.

The means to achieve this is to place a #define at the top of each module giving the name of the module. When included in its "home" module, the #ifdef-#else-#endif will cause the preprocessor to see the original declarations. When placed in foreign modules not sharing the same root, the preprocessor will see the external equivalents. Keil supports \_\_FILE\_\_ but it is not of practice use in this context, as its "value" cannot be used for a #define name.

By only including module-specific header files in those modules that actually need to access an item in another module, the operation of powerful make utilities such as Polymake or Keil's own AMAKE, is improved; provided the dependency list is kept up to date, any changes to a .h file will cause all modules that reference it to be recompiled automatically. Thus a modified program cannot be built for testing unless all modules referencing the altered item successfully re-compile. This usefully relieves the linker from being alone responsible for symbol attribute cross-checking - something which some linkers cannot be relied upon to do.

In most embedded C dialects this can be a major help in program development as, for example, a change in a widely-used function's memory model attribute can easily be propagated through an entire program; the change in the intelligent header file belonging to the function's home module causing the AMAKE to recompile all other modules referencing it. Likewise, a change in a variable's memory space from say XDATA to PDATA needs only one header file to be edited - AMAKE will do the rest!

Here's how it's done in practice:

```
/* Module Containing Main Program - MAIN.C */
#define _MAIN_
/* Define module name for include file control */
```

```

#include <reg517.h>          // Definitions for CPU
#include <v24ini537.h> // External references from V24INI.C #include <t0ini537.h>
// External references from
                        //T0INI537.C
#include <rlt_int.h>
// External references for RLT_INT.C

void main(void) {

    v24ini_537() ;

    timer0_init_537() ;

    while(1) {

        printf("Time = %d.%d", clock.secs, clock.msecs) ;
    }
    if(P1 | 0x01) {
        clock_run_fl = 1 ; // If button pressed start clock
    }
    else {
        clock_run_fl = 0 ; // If button released stop clock
    }
    if(P1 | 0x02) {
        clock_reset_fl = 1 ; // If button pressed clear clock
    }
}

/* Module Containing Timer0 Service Routine - RLT_INT.C */
#define _RLT_INT_ /* Identify module name */

/* External References */
extern bit clock_reset_fl // Flag to tell timer0 interrupt to
                        // reset clock to zero

/**/ INTERRUPT SERVICE FOR TIMER 0 /**/
void timer0_int(void) interrupt 1 using 1 {
    if(clock.msec++ == 1000) {
        clock.sec++ ;
        if(clock.sec == 60) {
            clock_sec = 0 ;
        }
    }
}

```



Taking the include files:

```
/* Include File For RLT_INT.C */

/* General, non-module specific definitions */
/* such as structure and union templates */
/* Clock Structure Template - Available To All Modules */
struct time { unsigned char secs ;
              unsigned char msec ; } ;

#ifdef _RLT_INT_
/* Original declarations - active only in home module */
/* Create XDATA Structure */
struct time xdata clock ;
bit clock_run_fl = 0 ; // Flag to tell timer0 interrupt to stop clock
#else
/* External References - for use by other modules */
extern struct time xdata clock ;
extern bit clock_run_fl = 0 ; // Flag to tell timer0 interrupt to stop clock
#endif

/* Include File For MAIN.C */
#ifdef _MAIN_
/* Local Data Declaration */
bit clock_run_fl = 0 ; // Flag to tell timer0 interrupt to stop clock
#else
/* External References - for other modules */
extern bit clock_run_fl ; // Flag to tell timer0 interrupt to stop clock
#endif

/* Include File For V24INI537.C */
#ifdef _V24INI537_
/* Original Function Prototype - for use in V24INI537.C */
void v24ini_537(void) ;
#else
/* External Reference - for use in other modules */
extern void v24ini_537(void) ;
#endif
```

Now, should any new global data be added to, for example, RLT\_INT.C, adding the original declaration above the "#endif" and the external version below, this makes the new item instantly available to any other module that wants it.

To summarise, the basic source module format is:

```
#define _MODULE_
#include <mod1.h>#include <mod2.h>
.
.
.
functions()
```

The include file format is:

```
/* General, non-module specific definitions such as structure and union templates */
#ifdef _MODULE_
/* Put original function prototypes and global data declarations here */
#else
/* Put external references to items in above section here */
#endif
```

## Standard Module Layouts For C51

To help integrate this program construction method, the following standard source and header modules shown overleaf may be used.

### Standard Source Module Template

```
#define __STD__
/* Define home module name */
*****/
*****/
/* Project:      X                               */
/* Author:       X          Creation Date:  XX\XX\XX  */
/* Filename:     X          Language:      X          */
/* Rights:       X          Rights:        X          */
/*
/*
/* Compiler:    X          Assembler:   X          */
/* Version:     X.XX       Version:     X.XX       */
*****/
/* Module Details:
*****/
/* Purpose:
/*
/*
*****/
/* Modification History
```

```

/*****
/* Name:          X                      Date:  XX\XX\XX */
/* Modification:  X                      */
/*                      */
/* Name:          X                      Date:  XX\XX\XX */
/* Modification:  X                      */
/*                      */
/* Name:          X                      Date:  XX\XX\XX */
/* Modification:  X                      */
/*                      */
/*****
/*****
/* External Function Prototypes                      */
/*****
#include ".h"
/* Standard ANSI C header files                      */
/*****
/* Global Data Declarations                      */
/*****
#include ".h"
/* Home header file                      */
/*****
/* External Declarations                      */
/*****
#include ".h"
/* Header files for other modules                      */
/*****
/* Functions Details:                      */
/*****
/* Function Name:                      */
/* Entered From:                      */
/* Calls:                      */
/*****
/*****
/* Purpose: main loop for training program                      */
/*                      */
/*****
/* Resource Usage:                      */
/*                      */
/* CODE      CONST      DATA      IDATA      PDATA      */
/* n/a       n/a       n/a       n/a       n/a       */
/*                      */
/* Performance:                      */
/* Max Runtime:                      Min Runtime:      */

```

```

/* */
/* */
/*****/
/* Executable functions */
/*****/
/*****/
/* End Of STD. c */
/*****/

```

## Standard Include Header File Template

```

/*****/
/* Project:      X */
/* Author:      X      Creation Date:  XX\XX\XX */
/* Filename:    X      Language:      X */
/* Rights:      X      Rights:      X */
/* */
/* Compiler:    X      Assembler:    X */
/* Version:     X.XX    Version:     X.XX */
/*****/
/* Modification History */
/*****/
/* Name:      X      Date:  XX\XX\XX */
/* Modification:  X */
/* */
/* Name:      X      Date:  XX\XX\XX */
/* Modification:  X */
/* */
/* Name:      X      Date:  XX\XX\XX */
/* Modification:  X */
/* */
/*****/
/*****/
/* Global Definitions */
/*****/
/* Structure and union templates plus other definitions */

#ifdef _STD_
/* Check for inclusion in home module */
/*****/
/*****/
/* Within Module Function Prototypes */
/*****/
/* Function prototypes from home module */

```

```

/*****
/* Within Module Data Declarations */
/*****
/* Data declarations from home module */
/*****
#else

/*****
/*****
/* External Function Prototypes */
/*****
/* External function prototypes for use by other modules */
/*****
/* External Data Declarations */
/*****
/* External data definitions for use by other modules */
/*****

#endif

```

## Summary

Provided the necessary module name defines are added to the first line of any new module and the new globals placed into the associated ".h" file, the overall amount of editing required over a major project is usefully reduced. 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.

## 4.4 Task Scheduling

### 4.4.1 8051 Applications Overview

When most people first start to learn to program, BASIC is used on a PC or similar machine. The programs are not usually too complicated; they start when you type \_"RUN" and finish at END or STOP. In between, the PC is totally devoted to executing your "HELLO WORLD" program. When it is finished you are simply thrown back to the BASIC editor/"operating environment".

All this is very good and you think you now know how to program. However, when writing for an embedded microcontroller like the 8051, the problem of where does the program start and finish suddenly presents itself. The average 8051 software system consists of many individual programs which, when executed together, contribute towards the fulfilment of the overall system objective. A fundamental problem is then how to ensure that each part is actually run.

#### **4.4.2 Simple 8051 Systems**

The simplest approach is to 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 constitutes a background loop. In the foreground might be interrupt functions, initiated by real time events such as incoming signals or timer overflows.

Data is usually passed from background to foreground 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 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 what to do this time, the code is executed and finally the program is exited, probably 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.

In this 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.

#### **4.4.3 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 each function to specify the next one to be executed. 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.

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.

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.

#### 4.4.4 A Pragmatic Approach

*Without resorting to a full real time executive like RTX51, what can be done?*

A simple mechanism to control the running of the background loop can be a simple switch statement, with the switch variable controlled by some external real time event. Ideally this should be the highest priority interrupt routine. The high priority background tasks are placed at the top case, with lower priority tasks located further down the case statement. Thus, on every occurrence of the interrupt, the switch is set back to the top. As the background tasks execute, they increment the switch. If the interrupt is absent for long enough, the switch will reach the lowest level and then return to the highest level automatically.

Should the interrupt occur at level 2, the switch variable is forced back to zero and so tasks at the lowest levels are simply missed. This is by no means an ideal system, since only the top level is ever executed, given a high enough interrupt 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 utilise an unused interrupt to guarantee that high priority tasks will be run on time. By setting the unused interrupt pending flag within the exiting high priority interrupt routine and placing the background task into the corresponding service routine, the punctual execution of the second task will occur. Of course, the unused interrupt priority must be set to a lower priority in the appropriate interrupt priority register(s).



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.

## 5 C Language Extensions For 8051

### Programming

8051 programming is mainly concerned with accessing real devices at specific locations, plus coping with interrupt servicing. C51 has made many extensions to the C language to allow near-assembler code efficiency. The main points are now covered.

### 5.1 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  ADCON; 0xDE
sbit EA  0x9F
```

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.*

## 5.2 Interrupts

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

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

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.

### 5.2.1 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.

### 5.2.2 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 familiar "8 \* n + 3" formula outlined in section 5.2.1 effectively becomes:

$$8 * n + 3 + \text{INTVECTOR}$$

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.

### 5.2.3 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 */
```

will allow a 6 byte spacing.

*Please note that for convenience INTERVAL defaults to 8 and INTVECTOR to 0x80000!*

#### 5.2.4 The Using Control

The "using" control tells the compiler to switch register banks. This is an area where the 8051 architecture works for the compiler rather than against it; the registers R0 to R7 are used extensively for the temporary storage of library routines and for locals. Ordinarily Bank 1 is used. However, to be able to use this standard code in an interrupt the register bank must be switched to 2 in the above example. Thus the variables of the interrupted routines are preserved.

As a rule interrupts of the same priority can share a register bank, since there is no risk that they will interrupt each other.

If interrupt runtime is not important the USING can be omitted, in which case C51 examines the registers which are actually used within the routine and pushes only these onto the stack. This obviously increases the effective interrupt latency.

### 5.3 Interrupts, USING, Registerbanks, NOAREGS In C51

#### Everything You Need To Know

Interrupts play an important part in most 8051 applications and fortunately, C51 allows interrupt service routines to be written entirely in C. Whilst you can write perfectly workable (and safe) programs by using just straight ANSI C, you can significantly improve the efficiency of your code by gaining an understanding of the following special C51 controls:

- INTERRUPT
- USING

- NOAREGS
- RE-ENTRANT
- REGISTERBANK

### 5.3.1 The Basic Interrupt Service Function Attribute

The correct vector must be generated so that the routine may be called. C51 does this based on the argument to the interrupt keyword. The linker thereafter does not allow local data from interrupt routines to be overlaid with that from the background by creating special sections in RAM. C51 special "interrupt" function attribute example:

```
/*Timer 0 Overflow Interrupt Service Routine */
```

```
timer0_int() interrupt1
```

```
{
```

```
unsigned char temp1 ;
```

```
unsigned char temp2 ;
```

```
/* executable C statements ; */
```

```
}
```

- The "interrupt 1" causes a vector to be generated at  $(8*n+3)$ , where n is the argument of the "interrupt" declaration. An "LJMP timer0\_int" will be placed at location 0BH in the code memory.
- Local variables declared in the routine are not overlaid by the linker to prevent the overwriting of background variables.
- When the interrupt occurs, compiler-inserted code is run which pushes the accumulator, B,DPTR and the PSW (program status word) onto the stack if used in function, along with any registers R0-R7 used in the function.
- A RETI is inserted at the end of the function rather than RET. Taking an empty interrupt service function for the timer 0 overflow interrupt, this is how C51 starts off an interrupt routine that uses no registers at all:

#### timer0\_int Entry Code

```
void timer0_int(void) interrupt1
```

```
{
```

```
RSEG ?PR?timer0_int?TIMER0
```

```
USING 0
```

```
timer0_int:
```

```
; SOURCE LINE # 2
```

If a function, here called "sys\_interp" is now called from the timer0 service function, this is how the entry code to the interrupt changes.

### timer0\_int Entry Code Now With Called Function

```
; void timer0_int(void) interrupt 1
{
RSEG ?PR?timer0_int?TIMER0
USING 0
timer0_int:
PUSH ACC
PUSH B
PUSH DPH
PUSH DPL
PUSH PSW
PUSH AR0
PUSH AR1
PUSH AR2
PUSH AR3
PUSH AR4
PUSH AR5
PUSH AR6
PUSH AR7
```

Note that the entire current registerbank is pushed onto the stack when entering timer0\_int() as C51 assumes that all will be used by sys\_interp. Sys\_interp receives parameters in registers; if the entry to sys\_interp is examined, an important compiler trick is revealed:

### sys\_interp() Entry Code

```
; unsigned char sys_interp(unsigned char x_value,
RSEG ?PR?_sys_interp?INTERP
USING 0
_sys_interp:
MOV y_value?10, R5
MOV map_base?10, R2
MOV map_base?10+01H, R3
;--Variable 'x_value?10' assigned to Register 'R1' --
MOV R1, AR7
```

The efficient MOV of R7 to R1 by using AR7 allows a MOV direct, direct on entry to sys\_interp(). This is absolute register addressing and is a useful dodge for speeding up code.

### 5.3.2 The absolute register addressing trick in detail

The situation often arises that the contents of one Ri register needs to be moved directly into another general purpose register. This usually occurs during a function's entry code when a pointer is passed. Unfortunately, Intel did not provide a MOV Reg, Reg instruction and so Keil use the trick of treating a register as an absolute D: segment address:

*Simulating A MOV Reg, Reg Instruction:*

In registerbank 0 – MOV R0, AR7, is identical to – MOV R0, 07H.

*Implementing a "MOV Reg, Reg" instruction the long way:*

```
XCH A, R1
MOV A, R1
```

The use of this trick means however, that you must make sure that the compiler knows which is the current registerbank in use so that it can get the absolute addresses right. If you use the USING control, problems can arise! See the next few sections...

### 5.3.3 The USING Control

"using" tells the compiler to switch register banks on entry to an interrupt routine. This "context" switch is the fastest way of providing a fresh registerbank for an interrupt routine's local data and is to be preferred to stacking registers for very time-critical routines. Note that interrupts of the same priority can share a register bank, since there is no risk that they will interrupt each other.

#### 8051 Register Bank Base Addresses

R0 AR0 Absolute Addr. 0x00 REGISTERBANK 0

R1 AR1

R2 AR2

R3 AR3

R4 AR4

R5 AR5

R6 AR6

R7 AR7

R0 Absolute Addr. 0x08 REGISTERBANK 1, "USING 1"

R1

R2

R3

R4



R5  
R6  
R7

R0 Absolute Addr. 0x10 REGISTERBANK 2, "USING 2"

R1  
R2  
R3  
R4  
R5  
R6  
R7

R0 Absolute Addr. 0x18 REGISTERBANK 3, "USING 3"

R1  
R2  
R3  
R4  
R5  
R6  
R7

If a USING 1 is added to the timer1 interrupt function prototype, the pushing of registers is replaced by a simple MOV to PSW to switch registerbanks. Unfortunately, while the interrupt entry is speeded up, the direct register addressing used on entry to sys\_interp fails. This is because C51 has not yet been told that the registerbank has been changed. If no working registers are used and no other function is called, the optimizer eliminates the code to switch register banks.

### timer0\_int Entry Code With USING

*With USING 1*

```
; void timer0_int(void) interrupt 1 using 1 {  
RSEG ?PR?timer0_int?TIMER0  
USING 1 <--- New register bank now  
timer0_int:  
PUSH ACC  
PUSH B  
PUSH DPH  
PUSH DPL  
PUSH PSW  
MOV PSW, #08H
```

## sys\_interp() Entry Code

*Still using registerbank 0*

```
; unsigned char sys_interp(unsigned char x_value,  
RSEG ?PR?_sys_interp?INTERP  
USING 0  
_sys_interp:  
MOV y_value?10, R5  
MOV map_base?10, R2  
MOV map_base?10+01H, R3;  
--Variable 'x_value?10' assigned to Register 'R1' --  
MOV R1, AR7      <----- FAILS!!!!
```

Absolute register addressing used assuming registerbank 0 is still current and so program fails! (Solutions in 5.3.6-8).

### 5.3.4 Notes on C51's "Stack Frame"

C51 uses a degree of intelligence when entering interrupt functions. Besides the obvious step of substituting RETI for RET at the end of the function, it automatically stacks only those registers that are actually used in the function.

*There are however, some points to be aware of:*

- If an interrupt function calls a function, C51 will stack all the Ri registers, regardless of whether they are used or not. The total time to PUSH and POP these is 16us at 12MHz, which may be viewed as unacceptable for a time critical interrupt.  
Therefore you should either avoid calling functions or use the USING control. This will do a simple registerbank switch at the entry and exit from the routine. As the PUSHING of registers onto the stack uses the same overall number of DATA locations, there is no difference in overall RAM usage.
- Any variable declared within an interrupt function will not be overlaid onto background data or that originating from other interrupts.
- Never call an interrupt function from the background. There is sometimes a temptation to do this during program initialisation, for example. The linker will get very confused and will quite likely make dangerous mistakes like overwriting background variables!
- Using the USING control will generally consume more RAM than simply PUSHing registers onto the stack: in the case where the interrupt function employs less than 8 registers, 8 - <number of registers actually used> will be wasted. Thus there is no virtue in avoiding the USING control!
- Interrupts of equal priority can share the same register bank as there is no chance of them interrupting each other.

### 5.3.5 When To Use USING

- Interrupts which must run as fast as possible, regardless of overall RAM usage.
- Interrupts which call other functions.

### 5.3.6 The NOAREGS pragma

*Dealing With C51's Absolute Register Addressing.*

As has been pointed out, the 8051 has no MOV Register, Register instruction so the compiler uses MOV R1, AR7 where AR7 is the absolute address of the current R7. To do this though, the current registerbank number must be known. If a function is called from an interrupt where a using is in force, when compiling a called function the compiler must be told:

(i) not to use absolute register addressing with #pragma NOAREGS control before the function, and #pragma RESTORE or #pragmas AREGS control enter the function.

*Or:*

(ii) the current registerbank number with #pragma REGISTERBANK(n).

For (i), applying NOAREGS to the sys\_interp function removes the MOV R7, AR7, replacing it with an awkward move of R7 to R1 using XCH A, Ri!

#### timer0\_int Entry Code

```
; void timer0_int(void) interrupt 1 using 1 {
RSEG ?PR?timer0_int?TIMER0
USING 1
timer0_int:
PUSH ACC
PUSH B
PUSH DPH
PUSH DPL
PUSH PSW
MOV PSW, #08H
```

#### sys\_interp() Entry Code With NOAREGS

```
; unsigned char sys_interp(unsigned char x_value,
RSEG ?PR?_sys_interp?INTERP
USING 0
_sys_interp:
MOV y_value?10, R5
```

```

MOV map_base?10, R2
MOV map_base?10+01H, R3;
--Variable 'x_value?10' assigned to Register 'R1' --
XCH A, R1 ;
MOV A, R7 ; Slow Reg to Reg move

```

### 5.3.7 The REGISTERBANK Control Alternative To NOAREGS

#pragma REGISTERBANK(n) tells C51 the absolute address of the current "using" registerbank base so that direct register addressing will work.

#### EXAMPLE:

```

/* Timer 0 Overflow Interrupt Service Routine */
timer0_int() interrupt 1 USING 1 {
    unsigned char temp1 ;
    unsigned char temp2 ;
    /* executable C statements */
}

```

Called function:

```

#pragma SAVE // Rember current registerbank
#pragma REGISTERBANK(1) // Tel C51 base address of current registerbank.
void func(char x) { // Called from interrupt routine
    // with "using1"
    /* Code */
}
#pragma RESTORE // Put back to original registerbank

```

Applying #pragma REGISTERBANK(1) to sys\_interp() restores absolute register addressing as C51 now knows the base address of the current register bank.

*Note:* Always try to use the REGISTERBANK(n) control for any functions called from an interrupt with a USING!

#### sys\_interp() Entry Code With REGISTERBANK(n)

```

; unsigned char sys_interp(unsigned char x_value,
RSEG ?PR?_sys_interp?INTERP
USING 1
_sys_interp:
MOV y_value?10, R5
MOV map_base?10, R2

```

```
MOV map_base?10+01H, R3;--  
Variable 'x_value?10' assigned to Register 'R1' --  
MOV R1, AR7
```

### 5.3.8 Summary Of USING And REGISTERBANK

*Expressed in psuedo-code!*

```
if(interrupt routine = USING 1){  
subsequently called function uses #pragma REGISTERBANK(1)  
}
```

*Note:* subsequently called function must now only be called from functions using register bank 1.

### 5.3.9 Reentrancy In C51 – The Final Solution

In addition to calling a function from interrupt, it is also sometimes necessary to call the same function from the background as well. This leaves the possibility open that the function may be called from two places simultaneously with disastrous results!

The attribute required to permit a function to be safely called both from background and interrupt routines simultaneously is "reentrant". This can also help in the previous situation of a function being called from an interrupt. The linker's "MULTIPLE CALL TO SEGMENT" warning is the first sign that you may be trying to use a function reentrantly.

Due to the way that C51 allocates storage for local variables and parameters, it is not possible to call a function from both an interrupt and the background loop. To allow only those functions to be used reentrantly that really need to be, it is possible to specify the reentrant attribute when declaring a function.

The ?C\_IBP value set up in startup.a51 tells C51 where to locate the artificial stacks used for reentrant functions. Each time a reentrant function is called, its incoming parameters are moved from the registers in which they were passed into an area of RAM, starting at the address indicated by ?C\_IBP. Likewise, any local variables used by the reentrant function are also allocated a place on this special stack.

When startup.a51 is executed before main(), the line:

```
IF IBPSTACK <> 0  
EXTRN DATA (?C_IBP)
```

```
MOV ?C_IBP, #LOW IBPSTACKTOP
ENDIF
```

initialises ?C\_IBP to the value of IBPSTACKTOP that you set up earlier. As each local is "pushed" on to the reentrant stack, ?C\_IBP is decremented. Thus if an interrupt occurs which calls the function again, the new call will start its reentrant stack from the current ?C\_IBP value. Thereafter, any local data or parameter is accessed by the code sequence:

*Get a local variable at offset 2 from the current base of the re-entrant stack:*

```
MOV R0, ?C_IBP ;    Get stack base
MOV A, @R0 ;        Add offset of local
ADD A, #002 ;
MOV A, @R0 ;        Get local via indirect addressing.
MOV R7, A ;         Store value whilst other local is ;
accessed.
```

On leaving the function, ?C\_IBP is restored to entry value by adding the total number of locals and parameters that were used. This represents a very large overhead and shows why reentrancy should only be used where absolutely necessary.

#### EXAMPLE:

The Reentrant Stack When Located In The IDATA Area

```
0xff    sys_interp parameter 0
0xfe    sys_interp parameter 1
0xfd    sys_interp parameter 2L
0xfc    sys_interp parameter 2H - call from background:
        ?C_IBP= 0xfc
0xfb    sys_interp parameter 0
0xfa    sys_interp parameter 0
0xf9    sys_interp parameter 1
0xf8    sys_interp parameter 2L
0xf7    sys_interp parameter 2H - call from timer0
interrupt: ?C_IBP = 0xf7
0xf6    sys_interp parameter 0
0xf5    sys_interp parameter 0
0xf4    sys_interp parameter 1
0xf3    sys_interp parameter 2L
0xf2    sys_interp parameter 2H - call from background
?C_IBP = 0xf2
```

0xf1  
0xf0  
0xef  
0xee

?C\_IBP acts as a base pointer to the reentrant stack and is used to access all locals in a reentrant function.

Adding the reentrant attribute to sys\_interp() still requires the NOAREGS control as the registerbank has been changed by USING 1. As a matter of policy, any reentrant function should also have the NOAREGS control so that it becomes totally registerbank-independent.

### sys\_interp() Entry Code

```
; unsigned char interp_sub(unsigned char x,  
RSEG ?PR?_?interp_sub?INTERP  
USING 0  
_?interp_sub:  
DEC ?C_IBP  
DEC ?C_IBP  
MOV R0, ?C_IBP  
XCH A, @R0  
MOV A, R2  
XCH A, @R0  
INC R0  
XCH A, @R0  
MOV A, R3  
XCH A, @R0  
DEC ?C_IBP  
MOV R0, ?C_IBP  
XCH A, @R0  
MOV A, R5  
XCH A, @R0  
DEC ?C_IBP  
MOV R0, ?C_IBP  
XCH A, @R0  
MOV A, R7  
XCH A, @R0  
DEC ?C_IBP ;  
SOURCE LINE # 22
```

### sys\_interp() Exit Code

?C0009:

```

MOV A, ?C_IBP
  ADD A, #010H    <-- Restore ?C_IBP to original
position
MOV ?C_IBP, A
RET ;
END OF _?sys_interp
END

```

### 5.3.10 Summary Of Controls For Interrupt Functions

Provided the following combinations of controls are used, you will avoid linker warnings and potentially dangerous code.

Interrupt Function Attribute	Called Function Attribute:
No USING USING n	"non-reentrant"
	no special attribute required
	USING n
	or
Interrupt Function Attribute	#pragma REGISTERBANK(n)
	or
	#pragma NOAREGS
	Called Function Attribute
no USING USING n	"reentrant"
	no register attribute
	#pragma NOAREGS

### 5.3.11 Reentrancy And Library Functions

The majority of C51 library functions are reentrant and can be freely used from interrupts and background. However, some larger library functions such as printf(), scanf() etc. are not reentrant. If you have used a non-reentrant library function reentrantly, you will get a "MULTIPLE CALL TO SEGMENT" warning, as would be expected.

"Hidden" library functions used to perform integer divides and multiplies etc. are all reentrant so you can perform a 16/16 divide in an interrupt without fear of upsetting the background.

#### *To Summarise:*

You can generally use library functions reentrantly but always check the C51 manual section 9 to check whether a function is reentrant or not.



## 6 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

### 6.1 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!

#### 6.1.1 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, @R0           addressed in R0
```

```
MOV  R0, #40          ; Put off-chip address to be indirectly
MOVX A, @R0           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
MOVX 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 '@'.

#### 6.1.2 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:

1. Place address to be indirectly addressed in a register.
2. 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
                                indirectly

}

```

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 ;
```

## 6.2 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:

```
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.

## 6.3 Arrays And Pointers – Two Sides Of The Same Coin?

### 6.3.1 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.

### 6.3.2 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.

### 6.3.3 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.

#### 6.3.4 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++ ;
```

## 6.4 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.

### 6.4.1 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 = 0x04 ;  
sensor_database.amp_gain = 0x21 ;
```

#### 6.4.2 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...

```

### 6.4.3 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 ; }
} ;

```

### 6.4.4 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
```

A trick using the linker is required here so the structure creation must be placed in a dedicated module. This module's XDATA segment, containing the RTC structure, is then fixed at the required address at link time.

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:

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

See section 7.6 for further examples of this placement method.

### 6.4.5 Pointers To Structures

```
/* Define pointer to structure */
```

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

```
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.

#### 6.4.6 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.

#### 6.4.7 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 I/O 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:

1. Define template
2. Declare structure pointer as normal
3. At run time, force pointer to required absolute address in the normal way.

## 6.5 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! */
    }

}

```

## 6.6 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 ?C0?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
```

## 6.7 Spaced Pointers In C51

Considerable run time savings are possible by using 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.

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.

## 7 Accessing External Memory Mapped

### Peripherals

Commonly, extra I/O 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,@DPTR` 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:

1. Use normal variables, char, ints etc, located by the linker
2. Use pointers and offsets, either via the `XBYTE` macros or directly with user-defined pointers.
3. Use the `_At_` and `_ORDER` directives.

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

### 7.1 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)*

## 7.2 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 initial 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

```

### 7.3 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.

## 7.4 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.

## 7.5 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.

## 7.6 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:



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

## 7.7 -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.

## 7.8 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!

## 8 Linking Issues And Stack Placement

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

### 8.1 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:

```
!51 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.

## 8.2 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

```

### 8.3 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.

## 8.4 L51 Linker Data RAM Overlaying

### 8.4.1 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 bytes 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.

#### **8.4.2 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:

1. Indirectly-called functions using function pointers
2. Functions called from jump tables of functions
3. 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

#### 8.4.2.1 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)  
//overlaid with

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
?PR?_EXECUTE?MAIN       000FH    000EH    //execute's, its
+_> ?C_LIB_CODE          000FH    000EH    //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.

#### 8.4.2.2 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	
<hr/>			
?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	



```

                                ?C0?MAIN1 segment
    }
}

void func2(void) {

    unsigned char i2 ;

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

        printf("THIS IS FUNCTION 2\n") ; // String stored in
                                           ?C0?MAIN1 segment
    }
}

code void(*jump_table[]) ()={func1,func2}; //Jump table to
                                           functions,
                                           // table stored in
                                           ?C0?MAIN1
                                           // segment.

/**/ Calling Function /**/

main() {

    (*jump_table[P1 & 0x01]) () ; // Call function via jump
                                   table in ?C0?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)

MCS-51 LINKER / LOCATER L51 V2.8

DATE 04/08/92 PAGE 2

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

---

?C_C51STARTUP	-	-	-	-
+> ?PR?MAIN?MAIN1				
?PR?MAIN?MAIN1	-	-	-	-
+> ?C0?MAIN1				
+> ?C_LIB_CODE				
?C0?MAIN1	-	-	-	-
+> ?PR?FUNC1?MAIN1				
+> ?PR?FUNC2?MAIN1				
?PR?FUNC1?MAIN1	-	-	0008H	0001H
+> ?PR?PRINTF?PRINTF				
?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: ?C0?MAIN1
    CALLER:  ?PR?FUNC1?MAIN1
```

```
*** WARNING 13: RECURSIVE CALL TO SEGMENT
    SEGMENT: ?C0?MAIN1
    CALLER:  ?PR?FUNC2?MAIN1
```

#### 8.4.2.4 Function Jump Table Warning Solution

The solution is to use the OVERLAY command when linking thus

```
main1.obj &
to main1.abs &
OVERLAY(?C0?MAIN1 ~ (func1,func2), main ! (func1,func2))
```

This deletes the reference to func1 & 2 from the ?C0?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 +> CALLED SEGMENT	BIT-GROUP		DATA-GROUP	
	START	LENGTH	START	LENGTH
?C_C51STARTUP +> ?PR?MAIN?MAIN1	-	-	-	-
?PR?MAIN?MAIN1 +> ?C0?MAIN1 +> ?C_LIB_CODE +> ?PR?FUNC1?MAIN1 +> ?PR?FUNC2?MAIN1	-	-	-	-
?PR?FUNC1?MAIN1 +> ?C0?MAIN1 +> ?PR?PRINTF?PRINTF	-	-	0008H	0001H
?PR?PRINTF?PRINTF +> ?C_LIB_CODE +> ?PR?PUTCHAR?PUTCHAR	0020H.0	0001H.1	0009H	0014H
?PR?FUNC2?MAIN1 +> ?C0?MAIN1 +> ?PR?PRINTF?PRINTF	-	-	0008H	0001H

#### 8.4.2.5 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?TIMERO_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?TIMERO_INT?MAIN2
      CALLER2: ?C_C51STARTUP
```

#### 8.4.2.6 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
<hr/>		
?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.

### 8.4.3 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.

## 9 Other C51 Extensions

### 9.1 Special Function Bits

.v.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 */
```

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

## 9.2 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.

### 9.2.1 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.

### 9.2.2 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 `memcpy()` and `strcpy()` library functions.

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.

### 9.2.3 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 */
}

```

## 9.3 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

### 9.3.1 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.

### 9.3.2 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...

## 10 Miscellaneous Points

### 10.1 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 `C0000` (Lowest EPROM location)

The startup routine just clears the internal RAM and sets up the stack pointer. Finally it executes a `LJMP` to "main", (hopefully) the first function in the C program.

```
LJMP main
.
.
.
.
main()
{
}
```

In fact this need be the only assembler present in a C51 program.

### 10.2 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

### 10.3 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.

### 10.4 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.

### 10.5 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.

## 10.6 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 '\n' 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) != 0)\
                               {\ *s1++ ; *s2++; }\
                               }

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.

## 11 Some C51 Programming Tricks

### 11.1 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.

### 11.2 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 CCO Input Capture */
```

```
tdc_int() interrupt 8 {
```

```
/* High priority section - may not be interrupted */
```

```
/* Enable lower priority section attached to */  
/* ADC interrupt */
```

```
IADC = 1 ; // Force ADC interrupt  
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 */
```

```
}
```

### 11.3 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.

```
"reg.h"
#include "cemb537.h"
#include <stdio.h>

main()
{

    unsigned char tx_char, rx_char, i ;

    P4 = map2 ;
#include
    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)
```



```

        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

```

## 11.5 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 that the preprocessor only allows integer calculations.

Other more sophisticated examples are given in the C51 manual, pages 4–2.

## 12 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.

### 12.1 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.

## 12.2 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.



## 13 Outputs From C51

### 13.1 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.

### 13.2 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.

### 13.3 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.

## 14 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.

### 14.1 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  OFAH ;
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 ;
```

[illegible]

```

        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
;

```

## 14.2 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).

## 14.3 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

## 15 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!

### 15.1

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.

### 15.2

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.

### 15.3

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!

### 15.4

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.

### 15.5

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.

## 15.6

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.

## 15.7 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.

## 16 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. It will allow well structured programs devoid of the dreaded goto or LJMP. In fact most of the extra code generated by C over an assembler is employed in ensuring good program structure rather than just inefficient use of the 8051 instruction set. It offers true portability from the 8051 to other processors and, unusually, the reverse is also true. Thus existing functions can be re-used, so reducing development time.