This chapter will focus on the operation of basic logic elements, which should be familiar to most readers, such as flip/flops and registers that form the basic building blocks for interface elements. It will also cover the issue of address decoding to enable these elements through a programme statement and provide an introduction to the ARM architecture and its built-in peripherals. Finally the linkage between C and low-level assembler code will be shown through a simple example.

1.1 Embedded Systems

Embedded systems take many different forms but all rely on some computing processor whether it has a physical interface with the outside world like the keyboard and screen of a typical PC, the physical connections forming a communication network or specialised sensor and actuator hardware to control an automated machine. The other essential element of any system making use of a computer is its controlling program and in fact most embedded system component vendors provide integrated development systems or environments (IDE) that run on a PC to facilitate the software development. Interface design requires careful assessment of both the hardware and software requirements.
so that when the objectives are considered the most effective solution can be achieved. In summary, we can say that an embedded system is computerised and tailor-made for a particular application.

1.1.1 Processor Architecture (Revision)

It is important to have a clear understanding of the processor architecture so that the integration of interface peripherals can be appreciated. The fundamental hardware architecture of the computing element will be examined briefly to show where interface components of any variety link up. Figure 1.1 shows a simple architecture of processor and memory that is widely adopted; this is interconnected by address, data and control buses. It is assumed that the reader is familiar with the general operation of the processor in terms of the machine instructions and data held in the memory as well as the cyclic operations of instruction fetch and execution to perform the desired task. The interface circuits connect onto the same buses, so as far as the processor is concerned the interfaces look like an extension to the memory. An advantage of this arrangement is that the same processor instructions can be used to manipulate data in memory or interface values.

As a reminder, the function of the address bus is to differentiate all the different elements of the system, as no two parts can have the same address or a conflict and confusion will arise. It is usual to use a memory map in order to show how the range offered by the address bus is allocated to different parts of the system. Table 1.1 shows how memory and interface elements might be allocated in a simple system with a 16-bit address bus.

The data bus has an obvious function but it should be noted that its operation is bi-directional; except in special circumstances data is sent to the processor or delivered by it. The most important task of the control bus is to indicate the data bus flow direction and thus avoid a bus conflict. Memory and interface

![Figure 1.1  Fundamental computer architecture](image)
elements can then be arranged to perform the appropriate read or write function as demanded. In a complex processor like the ARM the control bus may have many other functions to accommodate such as direct memory access (DMA), where the processor operation is temporarily suspended, and interrupt, which enable hardware signals and specific software instructions to be linked. These special functions will be examined in more detail in a later chapter.

The binary machine instructions held in the memory for the processor to access are usually determined from a high-level programming language, like C, for example by a Compiler but this approach will always involve some redundancy or overhead that may form an unacceptable burden in some demanding situations where memory is limited or processing time is at a premium. So for greater efficiency a low-level assembler language, which has a close linkage with each machine instruction, is sometimes employed. Unfortunately the assembler language approach introduces considerable complexity and is associated with low programming productivity, so it should only be considered for the most demanding applications.

<table>
<thead>
<tr>
<th>Address range</th>
<th>System element</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xc000 to 0xffff</td>
<td>I/O interface</td>
</tr>
<tr>
<td>0xb000 to 0xbfff</td>
<td>Redundant space</td>
</tr>
<tr>
<td>0x9000 to 0xafff</td>
<td>12k Data memory</td>
</tr>
<tr>
<td>0x0000 to 0x8fff</td>
<td>36k Programme memory</td>
</tr>
</tbody>
</table>

**Table 1.1 Memory map**

1.1.2 Interface Subsystem

When an interface circuit is designed it is essential to link up with the bus signals using appropriate hardware circuits. In essence, the address bus must connect with an address decode function to create latch enable signals and the data bus must connect with output from registers or special input functions. Finally, the control line, representing data bus direction, is usually combined with the address decode logic so that an output function or input function can be enabled correctly when bus access is demanded. This is quite a complex task for an interface designer but some typical circuits will be discussed in more detail in the following sections to show the principles involved. In fact, the STM32F4 has many such interface subsystems integrated within the microprocessor chip itself making their application more straightforward for the user, removing a complex task for the designer. The resulting address map
for the STM32F4 device is shown in (ARM Cortex-4 Data Sheet Doc ID 022152 Rev 3 [1]) Section 1.4 and the principal eight 512 Mbyte blocks are shown on the left hand side. Each of these blocks is further sub-divided for the many different roles within the processor.

1.2 Software Architecture

Any interface design will involve elements of both hardware and software development. Fortunately for many applications the hardware design is almost complete because it can make use of the inbuilt components. So the essential tasks for the interface designer involve the use good software practice and implement reliable operation. The C language will be used for the examples throughout as this promotes a good software structure. If the design has special performance requirements where assembly language should be used much greater care in implementing the code is needed.

For the software design, a slightly more abstract model is required and it is most effective to start with a simple system block diagram arranged to emphasise the interface functions, as illustrated in Figure 1.2. Any potential application will require a diagram of this type so it is beneficial to start one at an early stage in the design process and keep it updated with changes as the design proceeds. An early development might identify the different aspects of initialisation and use of the particular interface elements, for example.

This diagram simply shows a conceptual application, which makes a call on specific interface elements to control an actuator of some kind or deliver on-screen status messages to the user. The links are shown as bi-directional to

![Figure 1.2 Interface software structure](image)
account for the feedback that should be delivered in a good design to confirm correct interface operation. In a practical situation this may also require additional status signals from the actual hardware to provide the possibility of a hand-shake communication with the application program. Most of the built-in peripheral subsystems provide a host of status signals that can be used to implement various forms of hand-shake.

For the software aspects of interface design and development this book will make use of the Keil uVision4 Integrated Development Environment (IDE) provided by ST Microelectronics, which includes a C compiler. A brief introduction to the uVision4 platform will be provided towards the end of this chapter and Appendix A and B are included to provide a brief tutorial reference to help when readers lack experience with this particular package. The debug facilities included in the package provide valuable tools to track program flow, observe changes in variable data and examine information from the integrated peripherals.

1.3 Essential Basic Logic Elements

A brief resume of significant logic elements is provided in this section for reference to show in particularly how interface circuits are constructed. The interface designer will rarely need to develop new interface circuits but it is useful to have a clear understanding of the main principles so that system block diagrams of complex subsystems can be appreciated more fully.

It will be assumed that the reader has working familiarity with logic gates and the creation of combinational functions using several gates, as well as optimisation of the circuit using Boolean equations and Karnaugh mapping. If revision of these topics is required the reader should refer to a suitable reference text of which there are many, Wakerly in reference [2] is particularly recommended. For interface circuits the most important elements apart from simple gates are the flop/flops so this type of component will be reviewed in more detail in the following sections.

1.3.1 The Basic Flip/Flop

The simplest two-state flip/flop can be created from a pair of cross-coupled NAND gates as shown in the circuit Figure 1.3. This circuit is rather difficult to analyse because of this coupling so it is most convenient to temporarily break one of the links creating an extra input q as shown in Figure 1.4. If the
inputs on $\overline{R}q\overline{S}$ are 111, $\overline{Q}$ will be 0 and $Q$ will be 1 by virtue of the NAND logic so reconnecting will make no change and the circuit will be stable in the set state. If the inputs are 001 $Q$ will be 1 and $\overline{Q}$ will be 0 by virtue of the logic so reconnecting will again make no change and the circuit will be stable in the reset state. In cases when $q$ and $Q$ are different when the reconnection is made the subsequent change brings the circuit to one of its stable conditions. For example, when $\overline{R}q\overline{S}$ is 011 $\overline{Q}$ is 1 and $Q$ is 0 by virtue of the logic so the subsequent change due to reconnection makes $q$ become 0 but this will not alter the circuit from its reset state.

However, this particular circuit is not functionally convenient in most applications because when the set and reset inputs are active (i.e. at logic 0) simultaneously the resulting state is not defined because both outputs take up the same logic level and the subsequent stable state when the inputs are deactivated is arbitrary. This conflicting situation is shown in the first row of the truth table in Table 1.2.

\begin{table}[h]
\centering
\begin{tabular}{c c c c}
\hline
$\overline{S}$ & $\overline{R}$ & $Q$ & $\overline{Q}$ \\
\hline
0 & 0 & 1 & 1 \\
0 & 1 & 1 & 0 \\
1 & 0 & 0 & 1 \\
1 & 1 & Last $Q$ & Last $\overline{Q}$ \\
\hline
\end{tabular}
\caption{Truth table for the simple flip/flop}
\end{table}
1.3.2 The Edge-Triggered D-Type Flip/Flop (Latch)

The most practical form of flip/flop, usually referred to as a latch, uses a more complex circuit containing a minimum of six gates, which will not be discussed in detail here, and introduces the idea of a ‘data’ input and a ‘clock’ as shown in the block diagram in Figure 1.5. A good description of this function can be found in Wakerly [2]. The functionality of this form of latch is most useful to know and can be described in several ways, that is truth table, Table 1.3 and timing diagram, Figure 1.6, both of which are useful in some circumstances. In either case the clock edge, the rising edge for the sake of this particular description, determines when the latch state changes and the D input determines the next state value, that is when \( D = 1 \) the resulting state is set \( (Q = 1) \) or when \( D = 0 \) the resulting state is reset \( (Q = 0) \), as shown in the timing diagram in Figure 1.6. Importantly, the input D has no effect and the state will not change until a clock edge arrives.

![Figure 1.5 D-type edge-triggered latch](image)

**Figure 1.5** D-type edge-triggered latch

<table>
<thead>
<tr>
<th>Clock</th>
<th>D-input</th>
<th>Q</th>
<th>Q next</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td></td>
<td>No change</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td></td>
<td>No change</td>
</tr>
<tr>
<td>Edge</td>
<td>0</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>Edge</td>
<td>1</td>
<td>X</td>
<td>1</td>
</tr>
</tbody>
</table>

![Figure 1.6 Timing diagram](image)

**Figure 1.6** Timing diagram
It should be observed that in the example timing diagram shown the latch state only changes when there is a rising clock edge. If the clock is at a steady high or low the latch state does not change even when the D input changes. Note also that for reliable operation in practice the D input must be stable a few nano-seconds before the clock edge and remain stable for a few nano-seconds after. In logic documentation these times are referred to as the setup and hold time. In most practical applications these requirements are not difficult to achieve because the bus timing specifications are set up conveniently to avoid any critical timing situations so problems should not arise unless long chains of gate are employed at any point in the system.

Note that in a special alternative form of this function the latch state follows the D input when clock is high so the resulting state is only stable indefinitely when clock is taken low. In logic terminology this operation is referred to as a ‘transparent’ latch because when Q follows D while clock is high and the latch appears to do nothing. This type will not be used in the interface circuits discussed subsequently but may find application in some special circumstances.

For applications in interface circuits the edge-triggered D-type latch has significant advantages of reliability and will be employed throughout. It is, however, important to carefully consider the timing relationship between clock and the D input because when the criteria of setup and hold time is compromised the resulting state will not always follow the designer’s expectations.

1.3.3 Edge-Triggered Latch with Enable

In practical situations an additional functionality of enable/disable is frequently required to ensure reliable operation under all possible conditions. This can be achieved quite simply with an additional gate circuit, as shown in Figure 1.7, when the latch is not enabled this makes the latch D input equal to its Q output so that the latch state doesn’t change when active clock edges arrive. When

![Figure 1.7 Flip/flop with enable functionality](image-url)
enabled, the data input reaches the latch and the next clock edge sets it to the required state.

Importantly, this avoids the need to disable the latch by interrupting its clock signal, which could be achieved with an additional gate. If used this arrangement unfortunately has the side effect of delaying the clock edge by a few nano-seconds possibly compromising setup and hold times when least expected. The solution shown is much more reliable and predictable so is automatically adopted in most cases.

Note that in this circuit an output enable is also included by using a tristate buffer to improve functionality as this switches the output to the third high-impedance state when disabled. This will be needed, for example if the latch output is connected to a bus line where other circuits take the driving role some of the time. This structure, with or without the output buffer, will form the D-type functional block in many of the following interface circuit structures.

1.3.4 Multi-Bit Registers

For many interface applications multi-bit latches, often referred to as registers, are required to link up with the multi-bit data bus employed in a typical microcomputer design. This is quite simply achieved as illustrated by the 4-bit example, shown in Figure 1.8. Here, the clock and enable lines are taken in common across the structure of D-type latches. This scheme can be extended easily to accommodate as many bits as the application requires and a register block in a more complex diagram will be assumed to take this form internally.

![Figure 1.8](image.png)  

**Figure 1.8**  Four-bit register with group input enable IE and group output enable OE
In an interface circuit signals to drive the group input enable (IE) and group output enable (OE) inputs need to be determined and this is the main function of a block of logic called the address decode because it effectively places the interface at a unique position within the memory map of the processor. Note that the group output enable may be fixed if the Q outputs are driving external port connections but not if they are connected to a bus where conflicts could arise. In the case of external output ports the tristate buffer will not strictly be required.

### 1.4 Output Configuration Options

When registers are used in interface circuits the outputs may be required to drive other components and this presents various situations that need to be considered. The normal flip/flop output will drive a one or zero level with equal strength; this will be quite satisfactory if it is only driving a simple input on the connected circuit. The output circuit can be seen to have an active circuit to pull the pin up to a one and an active circuit to pull the pin down to a zero as shown in Figure 1.9. This bidirectional configuration is often referred to as a *push-pull output*.

#### 1.4.1 Open Drain Configuration

An alternative option, which can be created using the output control block in Figure 1.9, is to disable the pull-up driver and replace it with a simple resistor either on-chip or externally; this is known as the open drain configuration.

*Figure 1.9  Push/pull output driver*
Note that this has a detrimental effect on the maximum speed because the resistor acts as a rather weak pull-up driver but importantly allows wired logic and multisource bus connections to be made and in fact will be used in connection with the I2C serial interface later in Chapter 5. The optional resistors are particularly useful in this situation. The diagram shown in Figure 1.10, with one pull-up resistor, exhibits a wired-AND configuration because the input to the system on the right will only get to a logic one when all the drivers are off, as soon as any one of the drivers is on the input will become zero.

So the designer’s choice of push/pull or open-drain and the terminating resistor arrangement will depend on the application envisaged. The natural choice in many situations will be push/pull with no resistors; only in special circumstances will other configurations be needed. Careful analysis of the requirements will be the best way to ensure that a satisfactory configuration is created.

1.5 The Address Decode

As stated previously interface elements are usually required to be located at a particular point, or use a limited range of addresses, within the memory map of the microcomputer’s architecture so that they can be accessed reliably by the software instructions. This is achieved through a combinational logic function taking the address bus as its inputs and delivering an enable for the interface register when the defined address is presented by the processor. In the arbitrarily chosen example shown in Figure 1.11, the interface is allocated the address 0x8FA3 on a 16-bit address bus so requires all 16 address bits to present appropriate states to the interconnected group of AND gates for the output enable to become active. This scheme is referred to as *full address decoding* because it takes all individual address lines into account. The circuit presented here is not sufficient for the IE and OE signals required because the state of the control bus is not included so further gate combinations will often be required in a practical situation.
Note that in modern practical processor devices the address bus typically has more bits, actually 32-bits in the ARM processor, and the decode circuit can become rather complex and unwieldy, requiring many gates to achieve the required function. A more economical scheme is widely employed and this is referred to as partial address decode. This uses fewer gates but has a serious side effect in that it creates a partitioned memory map with unusable gaps in some sections. With a vast 32-bit address range this, however, will not present a serious issue for most system designs.

1.5.1 Partial Address Decode

In the partial decode scheme a few of the most significant address bits and a few of the least significant address bits are employed as inputs to the decode circuit making the combinational function much simpler. The simplified circuit is shown in Figure 1.12. However, the resulting enable will inevitably be activated by a whole block of addresses within the memory map because the middle bits now don’t care. Taking the same example previously but removing the eight middle connections and their gating, that is A4 through A11, means that any addresses between 0x8003 and 0x8FF3 will also activate the interface. So a whole block of nearly 4K address locations are locked out from other uses or otherwise redundant.

In practice the top group of address lines is fully decoded so as to divide the whole memory map into useful blocks; some of these can be allocated to memory and others to interface circuits as required by the system configuration. A simple example of this uses the top three address bits connected to a three to eight line decoder as shown in Figure 1.13. If address lines A15, A14 and A13 are used, eight blocks of 8 Kbytes are created in the memory map Table 1.4.
An examination of the memory map for the ARM processor (ARM Cortex 4 STM32F4 Data Sheet Chapter 4) reveals that the top four address bits of its 32-bit address bus are decoded to provide eight 512 Mbyte blocks. Some of these blocks are further subdivided by using lower address bits to activate the built-in memories and I/O functions; in particular most of the I/O functions make use of 1 Kbyte blocks.
1.6 ARM Architecture

The architecture of the STM32F4 ARM based microprocessor system is quite complex in comparison with the simple architecture shown earlier in Figure 1.1 and has introduced many advanced features during its development. It has several types of embedded memory and two high speed internal buses that link with the embedded interface subsystems. The STM32F4 Data sheet has a detailed block diagram in Section 2.2 Figure 2.5. This shows that the 32-bit floating point ARM cortex processor has three independent bus connections, the instruction bus I, the data bus D and the system bus S. These are optimised for particular types of interaction between the processor, the memories and the peripherals. The advanced high-performance bus (AHB) and the advanced peripheral bus (APB) form the main resources that link the memories and peripherals through the 32-bit multi-AHB bus matrix, see Figure 6 in the (ARM Cortex-4 Data Sheet Doc ID 022152 Rev 3).

The 32-bit multi-AHB bus matrix interconnects all the system masters (processor (CPU), DMA subsystems, Ethernet peripherals and USB High Speed peripherals) and the slaves (Flash memory, RAM, Flexible static memory controller (FSMC) as well as the AHB and APB that link up the peripherals) and ensures a seamless and efficient operation even when several high-speed peripherals are required to work simultaneously.

The peripherals provide a wide variety of functions but the most obvious in the block diagram are the set of up to 14 timer modules, six universal synchronous or asynchronous serial interfaces (universal synchronous/asynchronous receiver/transmitter, USART), various proprietary interfaces as well as analogue to digital (AD) and digital to analogue (DA) converters for analogue interface requirements. Most of these will be discussed in detail in later chapters.

The examples given in the text make use of the STM32F4-Discovery board, which is available at low cost from various commercial distributors. The family of boards included provides some useful expansion capabilities that will be used for the case studies in Chapter 7.

1.7 Interface Software Development

The uVision4 IDE, which provides an extensive range of development and debug tools, can be downloaded freely from the ST Microelectronics web site and this package comes with a wide selection of examples set up to illustrate the operation and control of various specific interface subsystems. These will be used extensively throughout the text to show how practical design of interface modules is achieved.
It is important to gain working familiarity with the tools presented through the IDE screen and in overview it will be seen that this is split into several sub-windows. The top left window provides for project management so that the various files needed for a particular implementation can be determined. Note that this area has some alternative functions but more detail will be given when particular examples are discussed. The central window area presents resources for programming and debug that can be employed in the software development. Some useful word processing functions are provided and debug facilities include single-stepping through the code statements, setting breakpoints to halt execution at a particular point in the code so that the current state can be examined and trace facilities to observe programme sequence and flow. Various optional windows on the right allow the user to examine code action on important aspects of the system memory and the peripheral subsystems. The bottom central window shows the IDE command sequence and will reflect the compiling and liking process steps, for example. Finally, an optional window at the top of the central area shows the assembler language derived from the C code statements, which is very useful in some situations. A screen shot from a simple I/O programme is shown in Figure 1.14.

In summary, the user should be able to make use of the IDE to set up a project, edit, compile and download the designed code and finally use the debug facilities to confirm correct operation of the function. The various steps involved
in these processes are described in more detail in Appendix A to help the reader gain familiarity with the IDE and the STM32F4-Discovery board if required.

The examples provided by ST Microelectronics are listed in Appendix B together with a brief summary of their intended target in terms of the peripherals that they address. These examples are written in C and all include the correct project settings so as to make it a simple matter for the user to get a practical implementation to work correctly. In many situations it will be most convenient to start with one of these examples so as to ensure that all the differing requirements are satisfied. Once the various aspects are well understood optimisation of the setup for a particular application will be possible.

The examples are set up to focus on a particular peripheral and the C language is employed to enable the potential user to gain a rapid understanding of the software requirements. It is also essential to use the product data sheet to provide the hardware description of the target peripheral so that the hardware and software aspects can be linked effectively. For example, the block diagram for an I/O port interface can be found in Section 6.3 of the STM32F4 Reference manual together with a tabulation of the port configuration bits, this information is essential to enable the user to control it more efficiently when required.

The C language examples activate and configure the port using a user friendly and well documented approach. However, it should be appreciated that this involves considerable overhead in terms of the individual machine instructions and processing cycles that will be required. Much greater efficiency can be achieved in terms of both processing cycles and memory requirements if Assembler language programming is adopted. This is not as challenging as might be expected because the debug facilities, provided within the Keil IDE, allow the Assembler translation of the C code statements to be inspected, thus allowing the user to rapidly learn how the assembler language statements are constructed.

To illustrate this linkage a simple C program is shown next where an array of consecutive numbers is declared and the program calculates the total summation.

```c
main()
{
    int value[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int i, sum = 0;
    for(i = 0; i <= 9; i++)
    {
        sum = sum + value[i];
    }
}
```
The assembler language code delivered by the Keil uVision 4 compiler in debug mode is shown next and it can be seen that the program contains many more statements than might be imagined from the brevity of the C statements. The first field shows the memory location, the second field the instruction code, which is 16-bits in most cases, and the third field the assembler mnemonic with its operands. The function of each instruction is fairly easy to understand remembering that the destination is the first operand but more detail can be obtained from the ARM documentation. In summary MOV and MOVS mean move, LDR means load register, ADD, ADDS and SUM perform the obvious calculations, CMP provides compare and B, BL and BLE provide program branches. The original C statements are included as comments to roughly indicate how the two representations are related.

```
0x0800038E   B08B            SUB         sp, sp, #0x2C
3:        int value[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
0x08000390    2228        MOVS        r2, #0x28
0x08000392    4908        LDR        r1, [pc, #32];
@0x080003B4
0x08000394    A801        ADD        r0, sp, #0x04
0x08000396    F000F821    BL.W       __aeabi_memcpy
(0x080003DC)
4:        int i, sum = 0;
5:        for(i = 0; i <= 9; i++)
6:          {
7:            sum = sum + value[i];
8:        }
9:        {                                 }
10:    }
0x080003A0     A801       ADD          r0, sp, #0x04
0x080003A2     F8500024   LDR      r0, [r0, r4, LSL #2]
0x080003A6     4405       ADD          r5, r5, r0
0x080003A8     1C64       ADDS         r4, r4, #1
0x080003AA     C09        CMP          r4, #0x09
0x080003AC     DDF8       BLE          0x080003A0
10:    }
0x080003AE     2000       MOVS        r0, #0x00
0x080003B0     B00B       ADD        sp, sp, #0x2C
```
The data values are placed on the stack and 40 bytes are reserved for this. Register r4 is used as the loop counter in the for-loop. Register r0 is used as the stack address of the current integer value in some instructions and the actual representation of integer i in others. Register r5 represents the sum variable accumulation. The tabulation Table 1.5 gives some interpretation of the code to assist with comprehension in terms of the required operations.

In fact this code is quite efficient in most respects; in particular the instruction at 080003A2 does a lot of work in calculating the address on the stack of the data value required at each iteration of the program. Note that each value uses 4 bytes. The only small inefficiency comes from the call to a function, which initialises the data values on the stack.

### 1.7.1 Software Development for Embedded Systems

It is quite unusual to be able to address all the requirements of a system design within a single program module so the IDE presents an optimal way to combine all the resources that will be required. The development resources provided by ST include a library of routines for each of the integrated subsystems. For example, the General Purpose Input and Output (GPIO) functions are in the library stm32f4xx_gpio.c and, for example, this provides routines

<table>
<thead>
<tr>
<th>Instruction address</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0800038E</td>
<td>40 bytes reserved on push down stack</td>
</tr>
<tr>
<td>08000394</td>
<td>R0 points to first stack location</td>
</tr>
<tr>
<td>08000394</td>
<td>Call function to initialise array data values</td>
</tr>
<tr>
<td>08000396</td>
<td>Initialise sum to zero</td>
</tr>
<tr>
<td>0800039E</td>
<td>Branch to end test</td>
</tr>
<tr>
<td>0800039C</td>
<td>Initialise i to zero</td>
</tr>
<tr>
<td>0800039E</td>
<td>Branch to test against limit</td>
</tr>
<tr>
<td>080003A0</td>
<td>R0 points to first stack location</td>
</tr>
<tr>
<td>080003A2</td>
<td>Next value retrieved from stack (at r0 indexed by four times loop count)</td>
</tr>
<tr>
<td>080003A6</td>
<td>Add next value to sum</td>
</tr>
<tr>
<td>080003A8</td>
<td>Loop counter increment</td>
</tr>
<tr>
<td>080003AA</td>
<td>End test, loop counter compared with limit value</td>
</tr>
<tr>
<td>080003AC</td>
<td>Branch back when incomplete</td>
</tr>
<tr>
<td>080003AE</td>
<td>End of program, r0 returned to 0</td>
</tr>
<tr>
<td>080003B0</td>
<td>Stack space given back</td>
</tr>
</tbody>
</table>
for GPIO_SetBits() and GPIO_ResetBits(), which will be examined more extensively in the next chapter. Each of these files includes useful notes on the particular subsystem and the functionality provided. Rather lengthy names are used throughout these library modules to help the user understand their application.

The only complex C programming concept that is used extensively is the use of data structures throughout the peripheral driver package. In each driver a data structure is defined, using the C typedef, which contains all the aspects required in the initialisation of the particular module in question. All the required structures are defined in the drivers so all the user has to do is to assign appropriate values to the structure elements, that is structure_name.element_name=0xf0Ae. In many cases the structure elements have a limited range of possible values so in these situations a series of names are defined to handle the values in a more intelligible way.

Examining the top left hand corner of the IDE screen a small panel showing all of the file names involved in a particular development will be observed, see Figures 1.14 and 1.15 for greater detail. These are grouped together in four sub categories for convenience, user, STM_Discovery, StdPeriphDrivers and MDK_ARM, but can be modified as required. Notice that startup_stm32f4xx.s is always included in the MDK_ARM category to establish various initial system resources like the clock generation and will call the user’s main code module. The file stm32f4_discovery.c in the STM_Discovery category provides board specific elements. The file system_stm32f4xx.c resides in the user category with any user designed code modules providing system specific elements. Finally the StdPeriphDrivers category should contain any peripheral drivers relevant to the particular application.

1.8 C Programming Revision

It will be found that the C programming examples used throughout the text and the support package rely heavily on structures and pointers as well as functions so a few notes on these have been included here for reference. If the reader is familiar with C programming conventions, this particular section can be safely ignored.

1.8.1 Arrays

Array names are themselves pointers so can be passed to functions where read and write access will be possible. In this example p[i] reads the message characters from main. The string can be altered if required p[0] = ‘X’ will
change the first character from T to X. Note that the declaration `char *p` can also use the alternative form `char p[]`.

```c
int main(void)
{
    int length;
    char message[] = "TEST Message/r/n";
    length = my_length(message);
}

int my_length(char *p)
{
    int i = 0;
    while(p[i] != 0)
```
 Structures and typedef

These definitions allow a very convenient way to set up small data bases that are particularly useful in grouping together all the parameters that need to be initialised in a particular interface module. In this short example the structure is declared in main and the function call passes a pointer to it. The function can access the structure elements using the special $\rightarrow$ shorthand instead of *(p).first. If the structure is declared within a function, instead of in main, its pointer can be passed to another function quite easily.

```c
typedef struct MY_DATA {
    int first;
    int second;
} MY_STRUCT;

int main(void) {
    MY_STRUCT record;
    init_structure(&record);
}

void init_structure(MY_STRUCT *p) {
    p->first = 100;
    p->second = 200;
}
```

1.8.3 Header Files

Header files (such as stm32f4xx_gpio.h) are used extensively within the support package as there is a specific header file for each of the peripherals that define both reference values and data structures. In order to incorporate a
header file the #include statement is used and the header file location is specified. For example:

```c
#include "my_header.h" if the file is in a local or specified directory or
#include <my_header.h> if the file is in a standard location.
```

It is usual to have a #include statement for each required file but as the application may involve many different peripherals this would become rather tedious to handle. In order to pick the header files up automatically it is only necessary to declare #include “stm32f4_discovery.h” to cover all possibilities. This file actually contains #include “stm32f4xx.h”, which contains all the references.

Remember that any functions that you create must be declared before the start of main(), for example:

```c
void GPIO_setup_pins(void);
```

These declarations can advantageously be part of your own header file, as this is useful particularly when your source code spans several files, otherwise they would have to be declared in each of them.

In summary, remember to declare the structures you want to use either in main() or in your own functions. When access to the data structure is required, such as with a supporting function, you must use a pointer to the structure that has been defined correctly. Header files for the peripherals make their programming more straightforward, allowing realistic names to be associated with raw data values.

### 1.9 Conclusion

Some essential revision of processor architecture and a number of basic interface design hardware and software issues have been discussed in this chapter. It should be emphasised that familiarity with these aspects is key to establishing techniques that can be used in effective interface design and will also enable the reader to gain the maximum benefit from the following chapters.

The C language will be used extensively to promote rapid understanding of the issues so the most straightforward constructs will be used and the reader will not require knowledge of more than fundamental programming techniques. The linkage between C and assembler has been illustrated by a simple example but although this particular example is comparatively efficient assembler
language short cuts will be included where appropriate because it may help the interface designer to obtain the required efficiency in a demanding application. The uVision compiler and debug facilities always make it possible to examine the assembler code so shortcuts can be seen quite easily from careful analysis.

The next chapter will deal with simple parallel interface designs that can be used to link up with a variety of basic input and output devices. The ARM processor has several interfaces of this type built in so its user has to provide only straightforward hardware circuitry for many practical applications.

References


Further Reading