03 – SIMULATION SESSION

INTRODUCTION TO C PROGRAMMING: POINTERS, STRUCTURES AND UNIONS

INTRODUCTION

By the end of this session, you should be able to:

1. Use data pointer types in C to read and write to data memory
2. Perform binary and unary manipulation of variables and individual bits in data memory
3. Use unions and bitfields as an alternative method of bit-manipulation in data memory
4. Write a C-function and observe its the stack
5. Read and write to from digital outputs
6. Configure multi function pins as either inputs or outputs
7. Use functions to structure code

These are the new concepts and techniques you will encounter in this session:

C-programming Concepts	Hardware awareness / design	Development Tools
• Structures • Unions • Bit-wise logical operators • Bitfields • Type definitions • Constants in different bases • Pointer types • functions • the function stack • stdlib and the random() function	• Setting up Tri-state digital pins • Configuring a port to be digital • Reading digital inputs	• Stepping in and out of functions • Using the help system to find useful functions • Using the Visual Device Initialiser

INTRODUCTION

In previous sessions we have looked at the basic C data types. We also saw how the compiler creates space in data memory when we declare variables in our C-program. Each variable has an **address** (the location in memory), **content** (the data itself) and **size** (the number of bytes it occupies in memory). We also looked at **arrays** of data, whereby a contiguous block of memory is allocated. We also looked at initialisation of variables.

In this session, we shall examine the following new ways to work with variables:

- **structures** – a way to combine a collection of variables together in a convenient and readable way
- **unions** – a way to provide alternative and more convenient ways to **access** a variables that share the same memory
- **bitfields** – a convenient and readable mechanism for accessing individual groups of bits in a variable
- **pointers** – a mechanism for working with the address of variables / data memory rather than the data itself.

We will also begin to use our own functions as opposed to writing everything in the main() function.

IMPORTANT NOTES

1. For the rest of this document, I shall refer to the PIC24FJ128GA010 as the PIC24
2. On your module site, you will find a datasheet for the PIC24. This is a critical document that you should keep on a memory stick or your U drive as I will be referring to it.

01 - LOGICAL BITWISE OPERATORS

Often we wish to set, reset or read an individual bit of a variable or memory mapped register, such as PORTA. The C-language makes this very possible in a number of ways. In this section, we shall consider the bit-wise operators &, |, ^, ~ (AND, OR, XOR and NOT). We will also look at ways to specify integer data types in different **bases**, including hex, octal and binary.

To make this overall section (and our code) clearer, we first look at how to assign values to variables in different types.

CONSTANTS IN DIFFERENT BASES

Consider the binary number 0001 0110.

- The decimal equivalent of this is 16+4+2 = 22;
- The hexadecimal equivalent of this number is 16h
- The octal (base 8) equivalent of this is 026 (tip – write the bit in groups of 3, 000 010 110)

All these representations can be written in C as follows:

```
int x = 0b00010110; //binary(16 + 4 + 2)
int x = 0x16;       //hexadecimal (1*16 + 6)
int x = 026;        //octal (2*8 + 6)
int x = 22;         //decimal (2*10 + 2)
```

The octal representation can really catch you out**. If you write an integer beginning with a zero, the compiler will take this to be an octal constant and not a decimal**. You have been warned!

BIT-WISE OPERATORS

Logical operators included in the C-Language are used to test and set individual bits in a variable. Consider the following example:

```
int a = 0b00101010;
int b = 0b00001111;
```

Table 1 - example of logical bitwise operators

Operator	Meaning	Task
&	AND	a & b =
|	OR	a | b =
^	XOR	a ^ b =
~	NOT	~a =

TASK 1-1 – BINARY BITWISE OPERATORS

Complete Table 1 using MPLAB to calculate your answers.

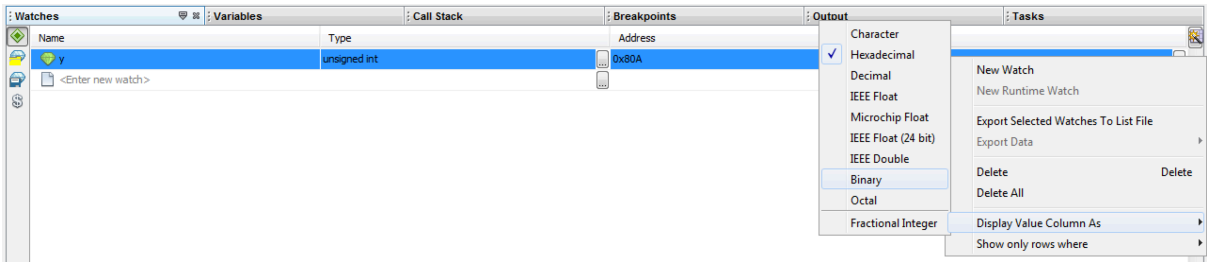
1. In the task 1 folder, open and build example1.1
    a. In the project properties, ensure the `circuit_simple_leds` circuit is selected
2. Use MPLAB to check you answers
    a. Examine `main1.c`
    b. Put a break point on the line that reads `asm("nop");`
    c. Run the code and add a watch to the variable `y` (view with Windows->Debugging->Watches)
    d. Right click on the value column in the watch windows and change the format (base) to binary (see figure below)
    e. Note the result.
3. Change the line that reads `y = (a & b)` to use another operator and repeat 2 to complete Table 1.

Using the logical bit-wise operators, you can set or reset individual bits: For example:

```
int a = 0x0000;     //reset all bots
a = a | 0b00001000; //set fourth bit
a = a & 0b11110111; //reset the fourth bit
a = a ^ 0b00001000; //toggle the fourth bit
a = a & 0x000F;     // Mask with 000Fh
```

4. Using logical operators, add lines to your code to do each of the following (in this sequence):
    a. **set** (switch on) all the LED's using a hexadecimal constant (all lights should go on)
    b. **reset** (switch off) D7 using the bit-wise logical operator **&**
    c. **toggle** D6 using bit-wise logical operator **^** (it should go out)
    d. set (switch on) D6 using the bit-wise logical operator **|**
    e. toggle D7 using bit-wise logical operator **^** (it should switch back on)
    f. invert all bits using the logical operator ~ (all lights should go out)
5. Use Proteus VSM and the watch window to check your answer. Show your code to the tutor when you are done.

Task (4) above is important as setting, resetting and toggling individual bits is a common operation in embedded C.



TASK 1-2 – FUNCTIONS

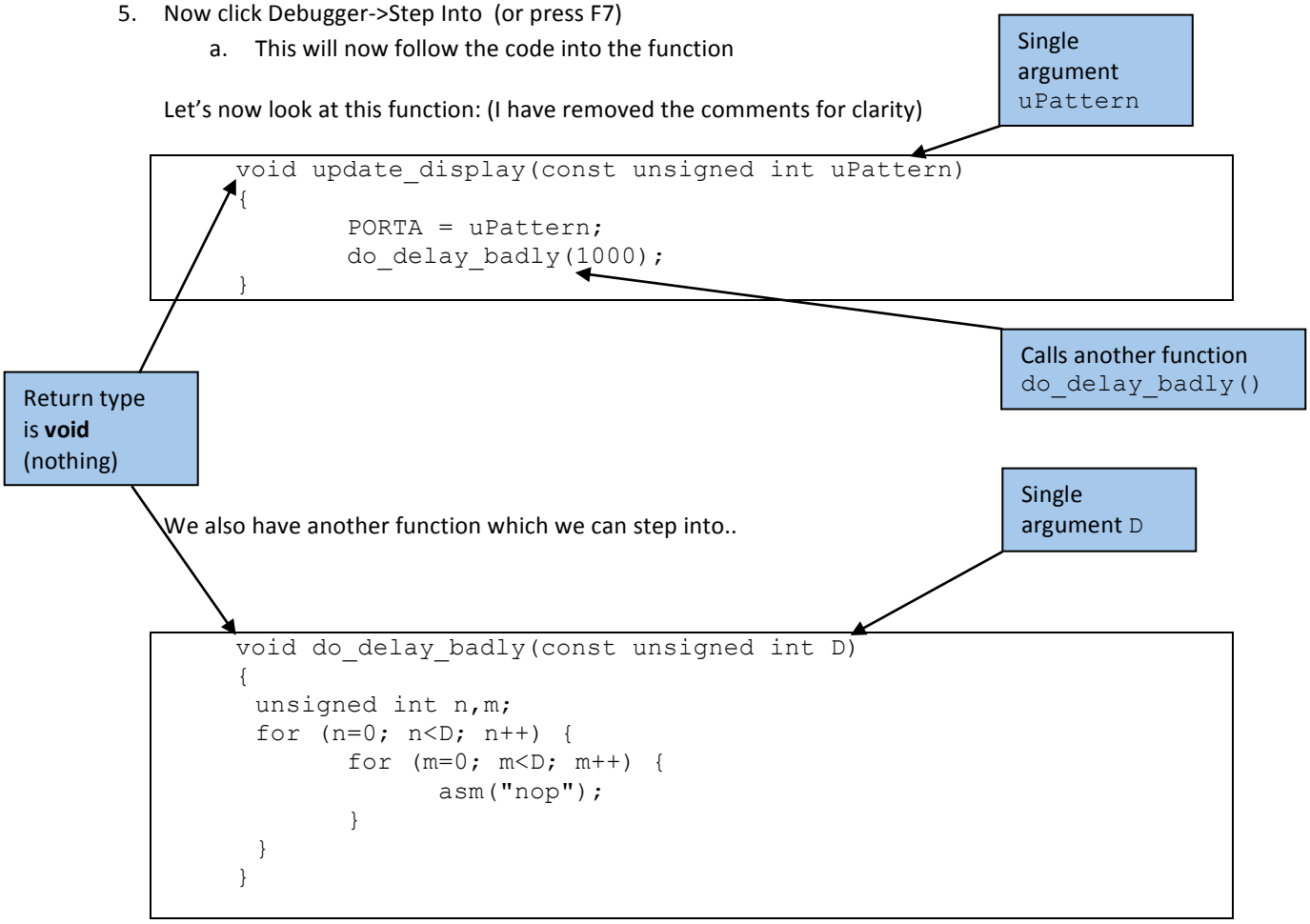
A function is declared as follows:

```
<return type> function_name ([argument 1], [argument 2],……)
```

Note that items in square braces are optional. It is easiest to explain this with some examples. Continuing with the theme of bit-wise operators, we again consider displaying a sequence of patterns, but this time using functions to make the code easier to read and more maintainable.

1. Open exercise1.2
2. Build and run to ensure the Proteus VSM debugger is being used and using the circuit `simple_circuit_leds`
3. Set a break point on the first lines that reads:
   `update_display(uOutputPattern);`
4. Run the simulation
   a. this will break on the line that reads `update_display(uOutputPattern);`
5. Now click Debugger->Step Into (or press F7)
   a. This will now follow the code into the function

Let's now look at this function: (I have removed the comments for clarity)

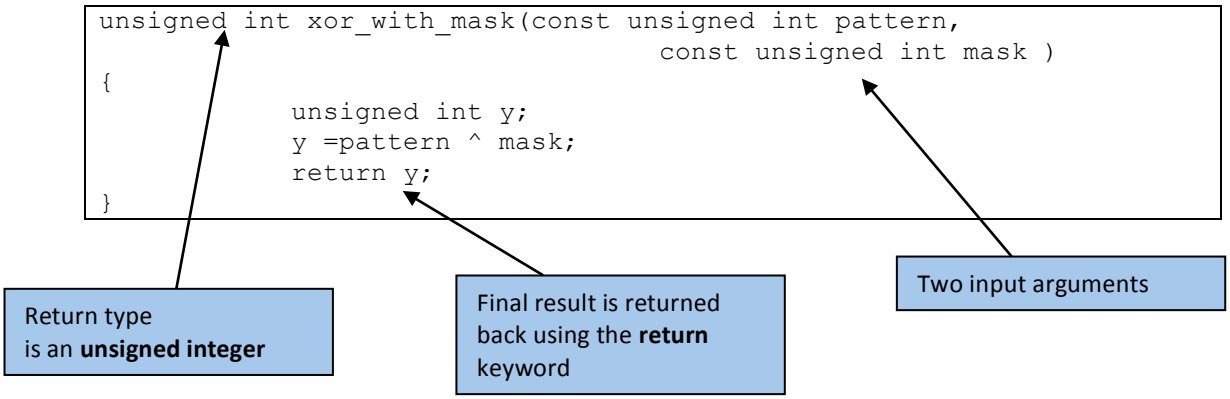


Some comments:

- If the function does not return a value, it must be declared as returning a variable of type **void**.
- The const is used to tell the compiler that the argument **D** cannot be overwritten by the function itself
- Note the variable name `n` is used in both the `main()` function and the `do_delay_badly()` functions. However, they are independent of each other (local variables).

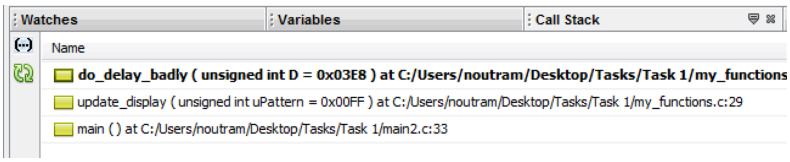
b. ~~Click the step-out button~~  ~~to allow the function to complete (~~missing from MPLABX)
Step over the code until you find yourself back in the calling function (main)

6. Step **into** the line that reads
   a. `uOutputPattern = xor_with_mask(uOutputPattern, 0b11111111);`
7. You will now find yourself inside the `xor_with_mask function`.
   Let's now inspect this function



This function calculates a value and returns it to the calling function.

8. Now step until you leave the `xor_with_mask` function
9. Step **into** the `update_display` function
10. Step **into** the `do_delay_badly` function.
11. Click Window->Debugging->call stack (In case you get lost!)



The call stack helps you navigate nested function calls. This is not unique to C of course. We will discuss function stacks at a later stage. However there is one further detail you need to observe. In main.c, there will find the following lines at the top of the file:

```
extern void do_delay_badly(const unsigned int D);
extern void update_display(const unsigned int uPattern);
extern unsigned int xor_with_mask(const unsigned int pattern,
                                  const unsigned int mask );
```

These lines tell the compiler that any reference to these functions should be to global functions (that are declared in another file). Although the code will compile without these lines, it will issue warnings if they are absent. **It is considered good practise to explicitly declare all global functions and variables in this way** to avoid ambiguity.

02 – TYPEDEF AND STRUCTURES

Before we discuss structures, there is a very simple keyword in C called **typedef**. It simply enables you to create new data types based on existing types. This is done as follows:

```
typedef oldname newname
```

For example:

```
typedef long int32;
```

You can now use the int32 datatype in your own code. For example:

```
int32 w1;
int32 w2;
```

This is useful for portability, and it can be also used to clarify some details about a data type in your code, but when you combine `typedef` with **structures**, it is also very useful.

STRUCTURES

Structures are a way to group variables together to make code more maintainable and easy to read. They are particularly useful when combined with unions (more on that later). They may also be seen as a way to give variables individual 'properties' that can be read or written.

- You can visualise a structure as being 'similar' to a class in C# or Java, only without methods.
- Note however that structures are **value types**, whereas an instance of a class is a **reference type**.

An example might be a motor controller, whereby you want to keep track of the 4 properties of 3 individual motors being controlled by your application : `motor1`, `motor2` and `motor3`. Each motor has 4 individual properties

- max_torque (integer)
- max_power (integer)
- digtital_port (A – G)
- serial_number (long)

What would **not** be convenient is to have 4x3=12 variables to store each property of each motor. You could use arrays, but there is a clearer way: use a **structure**. This is a way to create a new type of compound variable. For the motor example, you would define a `motor` variable as follows:

```
struct motor {
       unsigned int max_torque;
       unsigned int max_power;
       char digital_port;
       long serial_number;
} motor1, motor2, motor3;
```

Now we can access the individual elements of each motor in a neat and easy to read way. For example:

```
 //Local variables here
 motor1.max_torque = 10;
 motor1.max_power = 50;
 motor1.digital_port = 'A';
 motor1.serial_number = 104354934L;

 motor2.max_torque = 11;
 motor2.max_power = 60;
 motor2.digital_port = 'B';
 motor2.serial_number = 900012343L;
```

The 'dot' notation makes is explicitly clear what property of the variable we are reading or writing. However, it has another important purpose:

- **You can pass structures are arguments to a function** – this can help avoid great long lists of arguments and can be more descriptive
- **A function can return structures** – this means you can return more complex information than one single type

Now, as suggested earlier, we can create new data types with `typedef` based on structures as follows:

```
//Create new data type 'motor' - defined as a structure
typedef struct {
       unsigned int max_torque;
       unsigned int max_power;
       char digital_port;
       long serial_number;
} motor;

//Declare variables of type 'motor'
motor motor1, motor2, motor3;
```

We can now treat `motor` as any other data type. For example, an alternative could be to create an array of motors

```
 motor[3] m;
```

You could then access each motor via an index. For example:

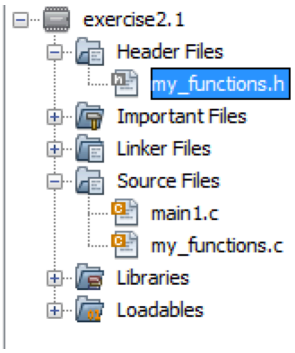
```
m[0].digital_port = 'A'; //set the port for motor 0
m[2].max_power    = 100; //set the max power for motor 2
x = m[1].max_torque;     //read back the max torque of motor 1
```

Question
Calculate how much space (in bytes) the structure motor would occupy in data memory on a PIC24? Show your calculations Use MPLABX and the **sizeof** function to confirm your answer. Do they match, and if not, why?

In the next task, we shall not be controlling motors just yet, but we will look at how structures are used.

TASK 2-1 – PASSING STRUCTURES AS ARGUMENTS

1. Open exercise2.1
   a. The code has now been changed to use a new data type '`pattern`'
   b. This new datatype is defined in '`my_functions.h`'



2. Step through the code and check you understand it
   a. Note the `xor_with_mask` has only one argument, whereas before it had two
   b. Note also that for illustrative purposes, **I have removed the const term**, so one could overwrite these parameters in the function

Not dreadfully inspiring I appreciate, but if you extend the concept to data type with tens of properties, you can see how this can clarify the meaning of code significantly.

3. Add a watch to the variable `outputPattern`. Step through the code in the main() function
   a. While still in the main function, note down the **address and contents** of `outputPattern` and it's individual elements

Property	Address	Content
outputPattern.bit_pattern		
outputPattern.bit_mask		

Question
How are the elements of the structure placed in memory relative to each other? **(in what order to they appear, are they adjacent to each other, how much space do they consume?)**

4. Now step into the `xor_with_mask` function
   a. This time, watch and note down the address of **m** and it's individual elements

Property	Address	Content
m.bit_pattern		
m.bit_mask		

You should have observed that **an extra copy** of the data has been made available as the (local) variable **m**. The parameter **m** becomes a local variable of the function, and is stored in a region of data known as the functions 'local stack'. The compiler will also let you overwrite this, as it is no longer declared as const. This data is only temporary however (only in **scope** while inside the function). Once we return to the main function,

we observe that the original `outputPattern` **remains unchanged** and the stack-space is released for other uses. In summary, when you call the following function

```
xor_with_mask(outputPattern);
```

a **copy** of `outputPattern` is placed on the function stack. Passing function arguments in this way is known as **passing by value**. This is because **structures are value types.**

Although this has a degree of built-in safety, for large structures this can become quite expensive in terms of data memory and CPU cycles. In the next task, we learn about a way to overcome this.

TASK 2-2 PASSING STRUCTURES BY REFERENCE

What if we wanted to modify the function parameters from with in a function such that the change is permanent? This is what is done in the next example.

1. Open exercise2.2
2. Set a breakpoint on the first line that read `xor_with_mask(&outputPattern);`
    a. Note this time, that `xor_with_mask` does not return a value
3. Run to the breakpoint and add a watch on `outputPattern`
4. Note the address and contents of `outputPattern`

Property	Address	Content
outputPattern.bit_pattern		
outputPattern.bit_mask		

5. Step into `xor_with_mask` and by adding a watch, note the address and contents of the symbol *m

Property	Address	Content
m->bit_pattern		
m->bit_mask		

6. Step out of the function, and again, Note the address and contents of `outputPattern`

Property	Address	Content
outputPattern.bit_pattern		
outputPattern.bit_mask		

From this is should be clear that by **passing by reference**, a separate copy is NOT placed on the function stack. In fact, it is the **address** of the parameter data that is placed on the function stack. This is evident in the way the function in called in main

```
xor_with_mask(&outputPattern);
```

The & prefix obtains the address of the object. By passing the address of the data, the function is able to manipulate the data directly, and thus does not (in this case) either need a copy or to return a modified copy.

Note also, the function declaration itself:

```
void xor_with_mask(pattern* m )
```

A data type is followed by a * indicates that the variable is actually an address of some data (of type `pattern` in this case). Such an address is known as a **pointer**. Knowing the type is important for the compiler, but more on that later.

When working with **pointers to structures**, you must also use the **->** operator and not the dot. Failure to do this will result in a compiler error.

Some further points to note:

- For anything other than a simple data type (char, int, long etc.), **it is usually faster to pass by reference** as only the address (and not the contents) of the data are placed on the function stack
- Passing data by reference allows the function to overwrite the data directly.
- Even if you do not need to overwrite the data, it is still faster for complex data types
    - In such cases, ensure you include the const term in the function arguments. This way, the compiler will refuse to compile statements that try to accidentally overwrite your data inside the function.
- For structures, you distinguish between a reference and a copy by using the **->** notation (as opposed to the dot notation for values)
- In the calling function, you must pass the address of the data using the **&** prefix
- It is often felt that code that passes-by-reference is less clear to read, so you should only do so if appropriate
    - You do not see (from the calling function) what is actually overwritten by the function. If you were using a closed source third-party library (where you cannot see the function code), then you would be forced to refer to the documentation[1] to understand the code.
    - The -> operator is often considered ugly and harder to read!

If you have not written C before, this is your first direct exposure to C-pointer types.

[1] Which no-one likes to read!

UNIONS AND BITFIELDS

When working closely with hardware, we sometimes want to manipulate data at bit level, and sometimes it is more convenient to work at byte or word level. Two very useful features in C that let you do this are unions and bitfields.

- Unions are a mechanism that allow you to refer to the same block of memory using different variable types
- Bitfields are a mechanism that allow you to specify a data type of a fixed number of bits – they can ONLY be used within a structure or union.

An example of a bitfield is as follows:

```
unsigned int x  :2; //2-bit integer, range of values = 0..3
unsigned int b0 :1; //single bit, range 0..1
```

These are very useful in embedded applications and this is best illustrated through some examples.

TASK 3-1 UNIONS AND BITFIELDS

1. In the task 3 folder, open, build and test exercise 3.1. (nb. It will run too fast to observe any values)
2. Ensure the circuit "circuit_seven_seg" opens
    a. Note a change in this circuit (see Figure 1) – you can now control whether the display is **enabled** via the $\overline{\text{EN\_LED}}$ pin connected to RA4 of the microcontroller (bit 4 of PORTA).
    b. The $\overline{\text{overbar}}$ indicates the pin is ACTIVE LOW, meaning the LED display will only register the input when the enable pin is a zero. This is a common pattern seen in digital electronics.
3. Step through the code to see what it does
4. Using the watch window, complete the following table

symbol	address	content
outputPattern		
word16		
led_code		

Question	Comment on the address of `outputPattern.word16` and `outputPattern.led_code`

Note the following code is used to declare the variable outputPattern

```
union {
    unsigned int word16;            //full 16 bits
    unsigned int led_code :4;       //least significant 4 bits
} outputPattern;
```

- This creates a variable `outputPattern` which is 16 bits in size
- It also creates two additional symbols "`word16`" and "`led_code`" **that have the same address in memory**
  - `word16` is of type unsigned int (16 bits)
  - `led_code` is of type **unsigned int : 4** (4 bits) – this is a **bitfield**

You access bit fields like structures using the dot notation. The key difference is this:

- The elements of a structure are sequentially arranged in contiguous memory (one after the other)
- The elements of a union **overlap** in memory (they all have the same start address)

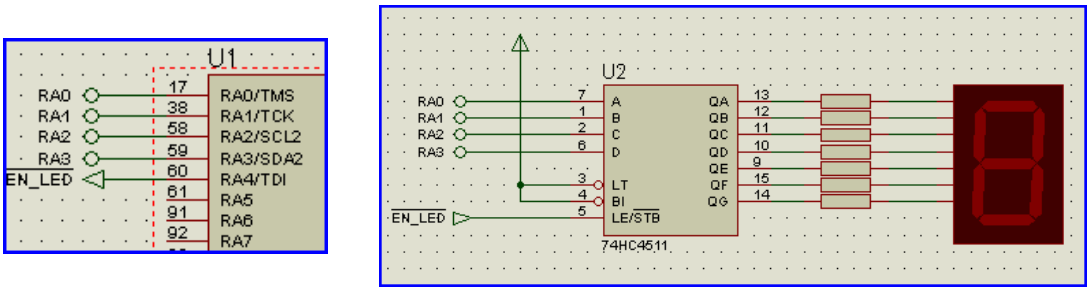


Figure 1 – showing the (active low) enable pin for the 7-segment LED display. The decoder will only read new data when the enable pin in LOW. When the enable pin goes high it remembers the last value it was given.

END OF WEEK 1

WEEK 2

UNIONS, STRUCTURES AND BITFIELDS

The combination of unions, structure and bit fields is a powerful and widely used method to access data. Often, the programmer wants to have a choice in how they modify / read a variable. For example, some requirements on a 16-bit CPU are as follows:

- Reading/writing to a complete 16-bit word
- Setting/reading individual bits in a 16-bit word
- Setting/reading smaller word lengths within the word (e.g. 4-bits at a time)

This can all be achieved using bitwise-logical operators of course, but this results in code that is hard to read and prone to error.

1. Open, build and run exercise 3.2
    a. Ensure circuit double_seven_seg opens (it should show two displays updating)
2. Restart the code and break point at the line that reads `for (n=0; n<100; n++) {`
3. Click and expand all variables (Windows->Debugging->Variables)
4. Step through the code. To understand how it works, keep a close watch on
    a. outputPattern
    b. the pins RA0..RA5 on the microcontroller

The union data type used in this example was first designed on paper. It is good practise to first draw a diagram of how you would like to access your data.

bit number	Word16	bits	led_code
0	16 bit integer	Bit0	4 bit unsigned int
1		Bit1	
2		Bit2	
3		Bit3	
4		ENABLE_DISP1	
5		ENABLE_DISP2	
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			

From this, the following code was produced:

```
union {
    unsigned int word16;            //full 16 bits

    struct {
     unsigned int bit0 :1;   // structure of 6 consecutive bits
     unsigned int bit1 :1;
     unsigned int bit2 :1;
     unsigned int bit3 :1;
     unsigned int DISP1_EN   :1; //Enable display 1
     unsigned int DISP2_EN   :1; //Enable display 2
    } bits;

    unsigned int led_code :4;    //least significant 4 bits as
} outputPattern;                 //a single 4-bit word
```

There are three over-lapping elements in this union:

- An unsigned int "`word16`" (16 bits) used at the start of the code to set all bits to zero
- 6 single bits – two of them ("`DISP1_EN`" and "`DISP2_EN`") are used to enable each of the displays
- A 4-bit word "`led_code`" – used to write the data you want displayed

In the main code, you can observe how easy it is to set the individual bits. For example:

```
outputPattern.bits.DISP1_EN = 1;      //disable display 1
outputPattern.bits.DISP2_EN = 0;      //enable display 2
```

The meaning of this code is clearer to another developer than using bit-wise operators or whole integers.

Question	How would you set bit 4 using the logical bit-wise operator |
	How would you reset bit 5 using the logical operator &

MORE (ADVANCED) POINTERS

You might have already been wondering why we have to use a separate variable for the output pattern for PORTA. It leads to a reasonable question:

COULD YOU MANIPULATE THE INDIVIDUAL BITS OF PORTA USING UNIONS, STRUCTURES AND BITFIELDS?

The answer is (of course) yes. One could argue that the advantage of using a separate variable is that all the changes are synchronous, but you can choose. The solution is to create a variable as we did in the previous example (with unions, structures and bitfields), and somehow force its address to overlap with PORTA.

PORTA is a 16-bit register with an address in data memory.

1. Open exercise3.3
    a. Put a break point on the line `TRISA = 0xFF00;`
    b. `run`
2. Click Window->Pick Memory Views->SFRs
3. Find the address of PORTA

Answer
Address of PORTA =

1. Click Windows->Debugging->Variables
2. Find the address and value of outputPattern

Answer
Address of outputPattern = **Value of outputPattern =**

3. Step through the code to see how it works

How did this work? First I created a new data type which I call `display_control` (based on the previous exercise)

```
typedef union {
    unsigned int word16;            //full 16 bits
    struct {
     unsigned int bit0 :1;
     unsigned int bit1 :1;
     unsigned int bit2 :1;
     unsigned int bit3 :1;
     unsigned int DISP1_EN   :1;
     unsigned int DISP2_EN   :1;
    } bits;
    unsigned int led_code :4;       //least significant 4 bits
} display_control;
```

I now create a pointer variable – remember that a pointer is a special integer variable with a value that holds an address of some other data (of a given type)

```
display_control *outputPattern;
```

So far, outputPattern does not point to anything (if you try to use it, the compiler will complain or you will get a run-time error). It is the next line that assigns the actual address of outputPattern.

```
outputPattern = (display_control*)&PORTA;
```

Notes:

- &PORTA is the address of PORTA.
- Putting (display_control*) explicitly tells the compiler the type of data the address points to. This is known as a **type-cast**.

I could have also written this without the type-cast

```
outputPattern = &PORTA;
```

and it would have worked, but the compiler will (rightly) show warnings. Other languages actually enforce type-casting.

Don't Panic![2]

If you are feeling rather confused at this stage do not be surprised – these are concepts which are not-simple to understand the first time you encounter them. However, once you master them, it will enable you to write more readable code with less errors.

This is one topic where I suggest you need time to ingest

(i) take time out
(ii) review the material later in the week
(iii) try and read up on the subject in a book
(iv) never give up until you have mastered it – it is important

There will be plenty of further opportunities to practise programming with unions, structures, bit-fields and pointers. The next section is rather less tricky.

[2] I know, I keep saying this

USING DIGITAL INPUTS

You may have noticed the line `TRISA = 0xFF00;` in the source code. Some of you will have already worked out what it means.

- TRIS stands for Tri-State (this means it can be an input or an output)
- **"A"** refers to PORTA (there are 7 ports, A-G on this device)

To keep it simple, the pins on the PIC24 usually have multiple functions. **Where** pins are used as "digital ports"[3], they can be configured as either a digital input or a digital output. Each port has a corresponding TRIS register to control this. For example:

```
TRISA = 0b00000001 //Sets bit0 (pin RA0) on PORTA to be an input,
                   //all other pins on PORTA are outputs
TRISB = 0b11110000 //Sets bits 4..7 (pins RB4..7)on PORTB to be inputs
                   //all other pins on PORTB are outputs
```

An easy way to remember this is this:

- a 1 looks like an `I` (`I` for **I**nput)
- a 0 looks like an O (O for **O**utput)

For more details, read section 9 of the data sheet. Let's now build on the previous circuit to include a digital input.

[3] Never **assume** they are – more on this later

TASK 4-1 READING A DIGITAL INPUT

In this task we use a switch as an input device. When the switch is pressed, the counter should reset to zero.

1. Open the file exercise4.1 from the task4 folder
2. Ensure the circuit double_seven_seg.DSN is open (the version in the task4/circuits folder)
3. Run the simulation
   a. Click on the press button with the mouse
4. Now take a look at the code changes made here

In main, some additional configuration has been performed

`AD1PCFG = 0xFFFF;` configures PORTB to be a digital port (rather than an analogue pin).

`unsigned int uButtonHasBeenPressed;` this records if the push button has been pressed

`TRISB = 0x0001;` sets RB0 to be a digital input (all others are outputs)

5. Step into the line that reads
   `uButtonHasBeenPressed = do_delay_badly(500);`
6. Look at the code in the function `do_delay_badly` and read the comments

This code is scanning or "polling" bit0 (RB0) of PORTB during the delay routine (as this is where we currently spend most of our time). If it detects a '1', then the function immediately exits, returning a 1. If no button is detected, then a 0 is returned.

7. Step out of the function

Here the code is examining the return value. If a button was pressed, a "break" statement is run and the code jumps out of the for loop.

Question

What flaws can you see in this code? You might like to comment on the following aspects:

- **Reliability**
- **Scalability (how easy it is to now add more features/complexity/inputs/outputs)**
- **Power consumption**
- **Accuracy**

CHALLENGE – WRITE A DIGITAL DICE

By modifying the code so far, can you write a program that does the following:

- User presses the button and the display cycles rapidly through random digits
- User presses the button and the display freezes
- Repeat the above

You've covered enough information to make this work, although you have yet to encounter some of the better programming techniques.

If you succeed, show the tutor next week. You might also like to reflect on any difficulties you encountered and to suggest how improvements could be made. To get you thinking about this, read the following section.

FINAL THOUGHTS

Most real systems have to read and write data in real-time. They also have to do some computation which also takes time. When you scale up to more complex systems, you may encounter network devices, audio devices, video output, serial communication (inc. USB), user input.

HANDLING ALL OF THIS AT ONCE IMPLIES THE NEED FOR 'CONCURRENCY', SOMETIMES REFERRED TO AS 'MULTI-TASKING'.

To provide a good user experience, the following usually apply

- a user expects a system to react in good time
- a user does not expect to be ignored (inputs lost)
- a user input should be acknowledged (if good HCI is being adhered to)
- the system should be able to preserve its state between uses (switch on, carry on where you left off)
- Errors should be handled gracefully

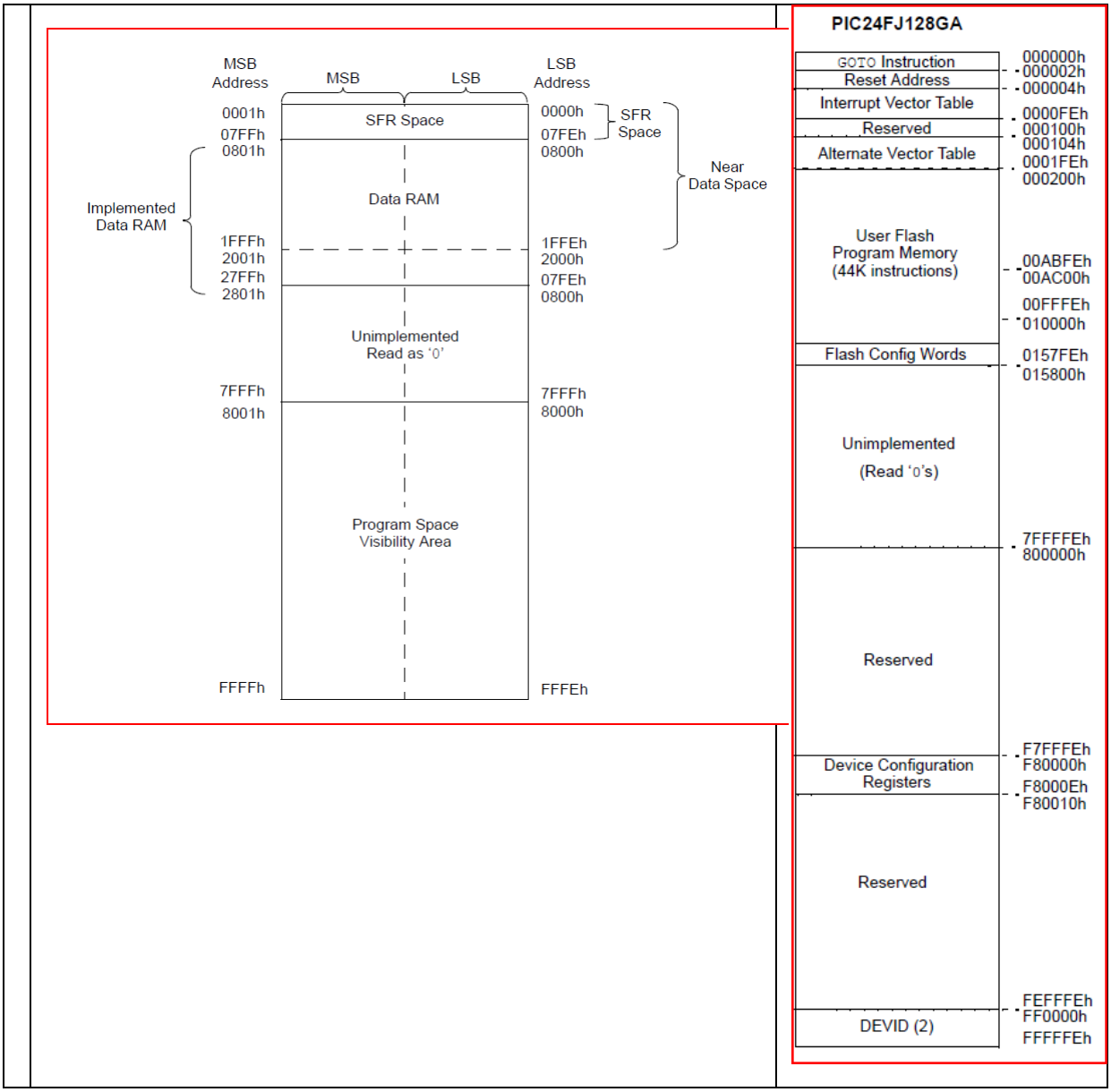
It might also be a **real-time** system – so in addition to the above

- Reading inputs will often have enforced deadlines – the system must response within a given time or data will be lost
- Writing outputs might have enforced deadlines – the output device might drop data if not serviced fast enough

It might also be a **safety critical system** – so in addition to the above,

- an input must NEVER be missed
- an output must NEVER be forgotten
- a deadline must NEVER be missed
- Errors should not exist or be handled in a pre-determined way (thus managing any risk)
    - Watch-dog timers
    - Brown-out detection
    - Exception handling
- Loss of input or output data, or failure to meet deadlines could injure or kill someone – you must build in contingency for such an event even though if should not ever happen
- Risk should be managed upfront in the design – you must assume a failure could happen
    - Build in redundancy

APPENDIX A - PIC24 MEMORY MAP



APPENDIX B – BLOCK DIAGRAM OF SHARED PORTS

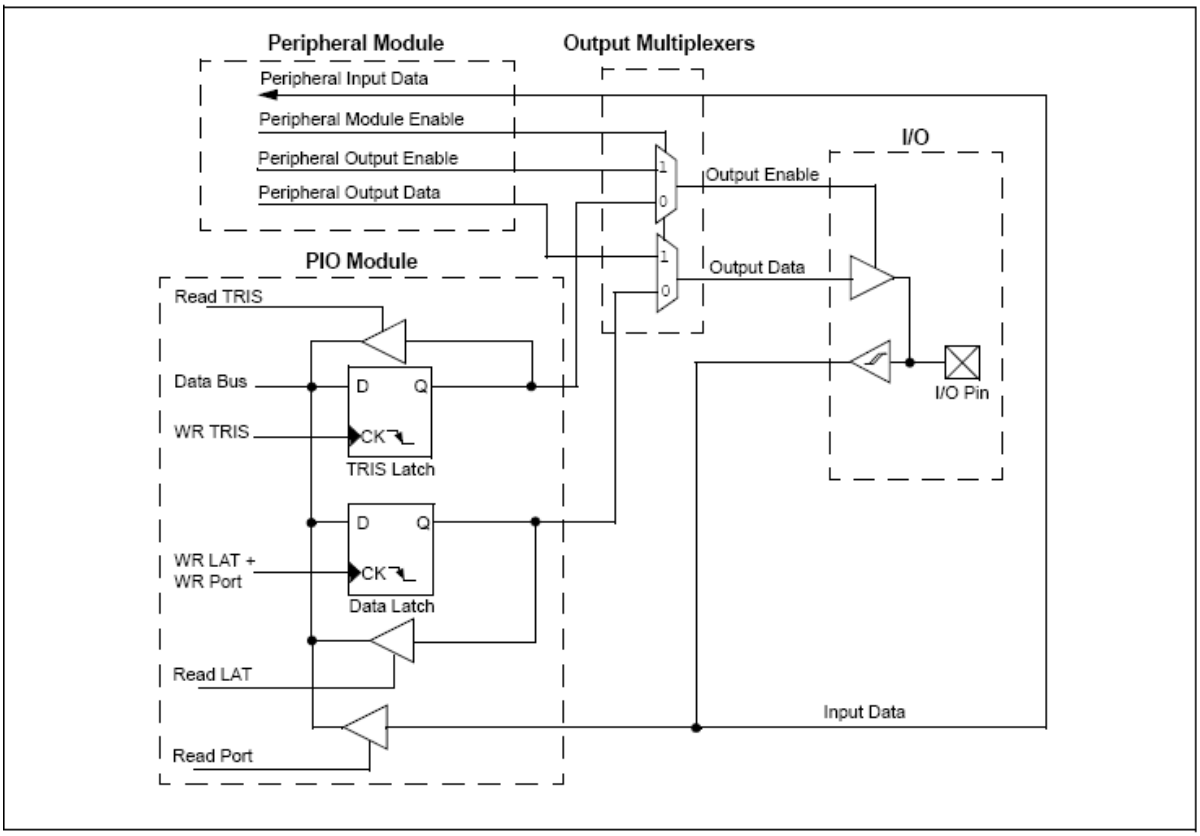


Figure 2 From figure 9-1 in the PIC24 datasheet, showing the block diagram of a shared port