02 – SIMULATION SESSION

INTRODUCTION TO C PROGRAMMING: MEMORY AND VARIABLES

INTRODUCTION

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

1. Describe the memory map of the PIC24FJ128GA010
2. Explain the difference between program memory and data memory
3. Inspect and edit the data memory via MPLAB
4. Describe the built-in variable data types in C
5. Perform binary and unary operations on variables
6. Describe any platform dependence issues with variables in C
7. Describe the function of the stack and heap
8. Write a C program from scratch that contains a main() loop and can write to a digital output

PIC24FJ128

The Microchip PIC24 family is very large. Each device varies in terms of memory, io pins, power consumption and interfacing options. This is to allow the designer to choose an optimal device that exactly meets a set of product requirements. Dedicated microcontroller devices are often used in very cost sensitive products (where each US cent makes a difference). The specific PIC24 microcontroller we shall be using this term is a PIC24FJ128GA010. This is one of the devices that can be fully simulated in Proteus. Furthermore, it is one of the more capable devices in the range, yet is very cheap ($3.63 at the time of writing).

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 useful document that you should keep handy.

HARDWARE ARCHITECTURE

The traditional model of a computer has a "Von Neumann architecture" as depicted in Figure 1. This has a single central processing unit (CPU) that sequentially **fetches** instructions and data from memory, **decodes** the instructions before finally **exectuting** them. Each of these processes requires **at least** one clock cycle. This complete cycle is illustrated in Figure 2. Note that there is only one "data bus" and one "address bus".

When writing assembly code, the programmer enters their software as a sequence of text-based instructions, for example: (pseudo assemble code)

```
LD R1,#1      //register R1=1 (constant data)
LD R2,(0123)  //register R2=contents of address 0123h in memory
              //                                  (variable data)
ADD A,R1,R2   //A=R1+R2 single instruction, no data
```

Instructions such as these end up as numbers stored in memory. The numbers are generated by an "**assembler**"[1]. Instructions (operators) are either stored on their own or packed together with some **constant** data (operand). Note that both the instruction and operand data has to fit into an 8 or 16 bit wide word (depending if is an 8-bit or 16-bit microprocessor), so this limits the size of any number[2]. However, when dealing with variable data (which is the more common scenario), the data is not pre-known to the assembler, and has to be fetched from / written to a separate memory location. Therefore, both the instruction and data have to be fetched separately before an instruction can be executed. If they both come from the same memory, this creates a 'bottleneck', and clearly has an impact on performance.

Note: Figure 2 would only be correct if the memory is fast enough to be read every clock cycle[3]. In practise, each fetch stage often requires multiple clock cycles (known as **wait states**) to reliably read and write data to memory. Memory speed has a major impact of overall performance.

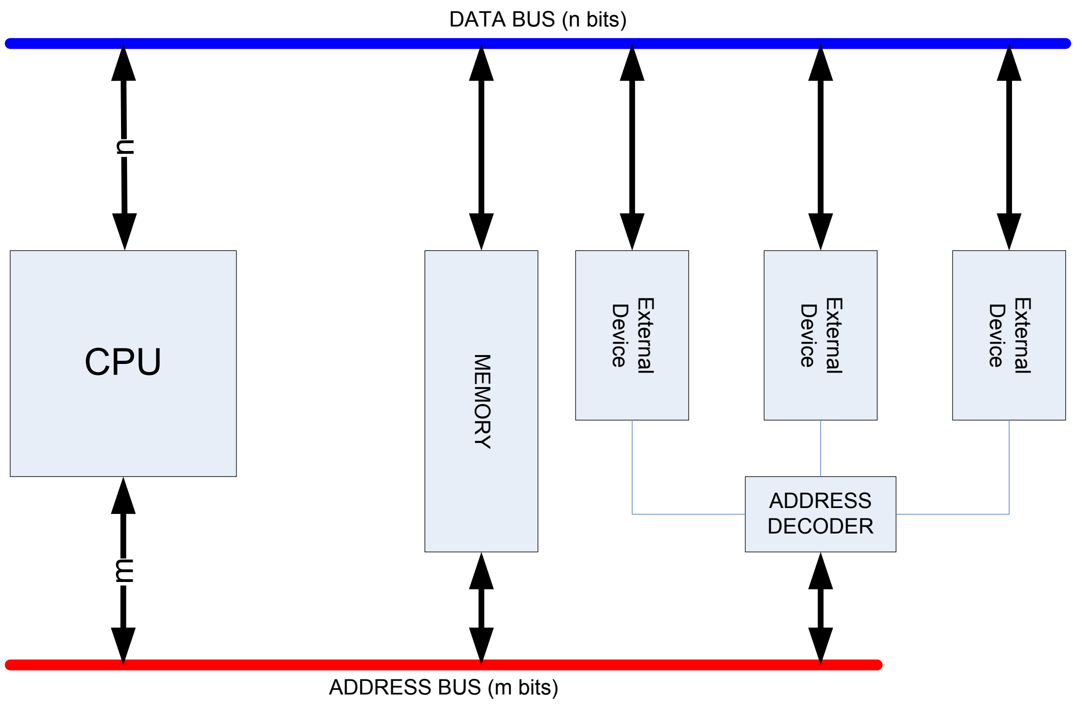


Figure 1. Simplified Von Neuman Architecture

[1] The very first computers used punch-card readers. The cards contained numbers directly written by the programmer

[2] Some processors have a variable length instruction word. Modern processors have multiple instructions packed into 64 bit words.

[3] In modern processors, this is known as cache ram and it is used for fast data access. Cache ram takes up a lot of silicon space and consumes much more power than the cheaper external ram modules, so is limited.

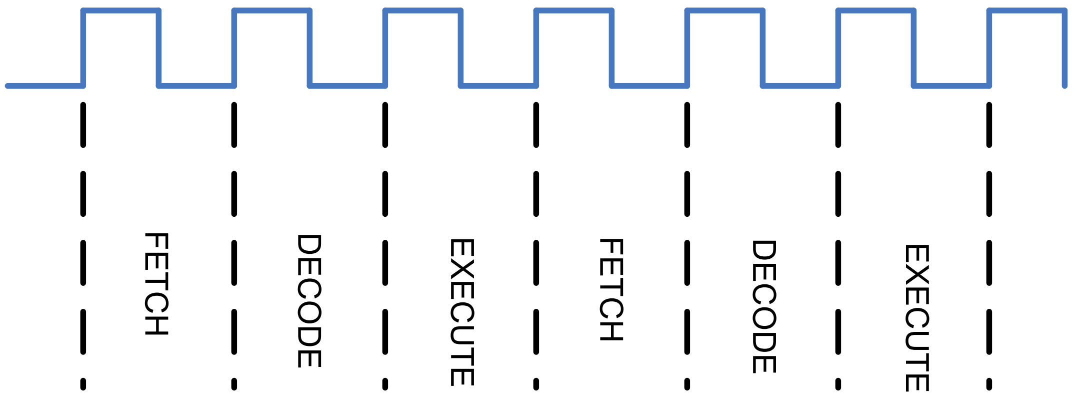


Figure 2 - Traditional Fetch-Decode-Execute cycle of a microprocessor (no wait states)

More modern devices typically have more than one address bus and data bus. The PIC24 family use what is called a (modified) **Harvard Architecture**, as depicted in Figure 3. Using this approach, **it is now possible to fetch a program instruction and move a data word simultaneously**. Furthermore, the device is **pipelined** (see Figure 4). This is a technique that performs 3 concurrent out-of-phase fetch-decode-execute cycles. Once the pipeline is full, t**he net throughput is one instruction per instruction cycle**. Note the following:

- Program memory on the PIC24 requires 2 clock cycles.
- Program memory uses FLASH memory technology – it retains its contents when the power is removed. This makes it very convenient for programming.
- Data memory is "static ram" (SRAM) – the contents of this are lost when the power is removed.
- Data memory is **dual-ported** – you can simultaneously read and write to data memory, and is often referred to as a **register-file**.

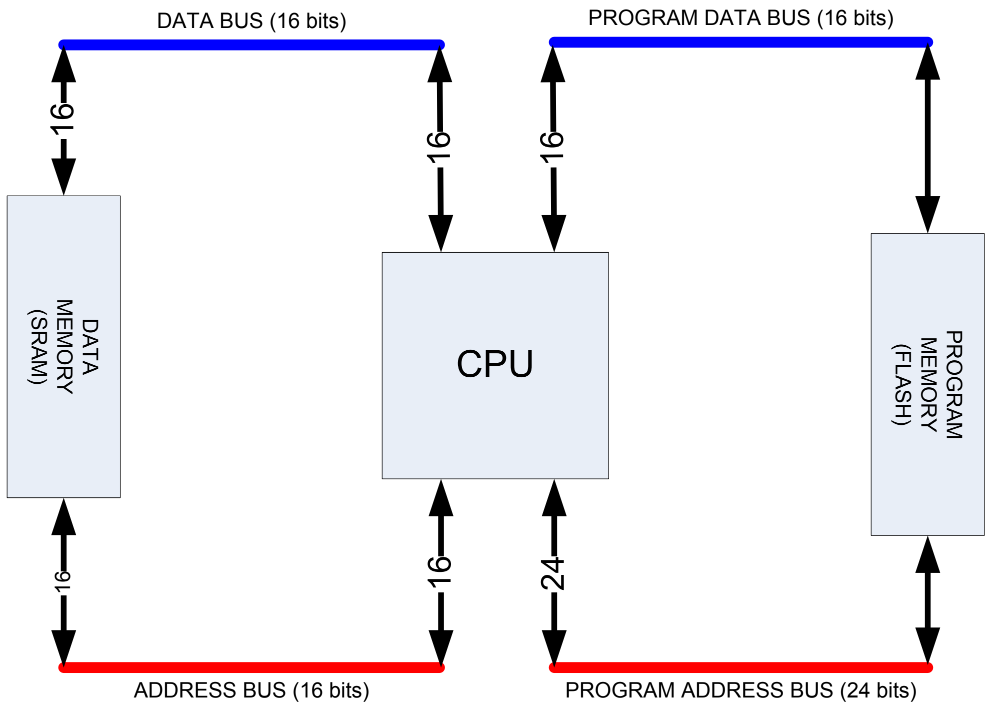


Figure 3. memory organisation for Harvard Architecture.

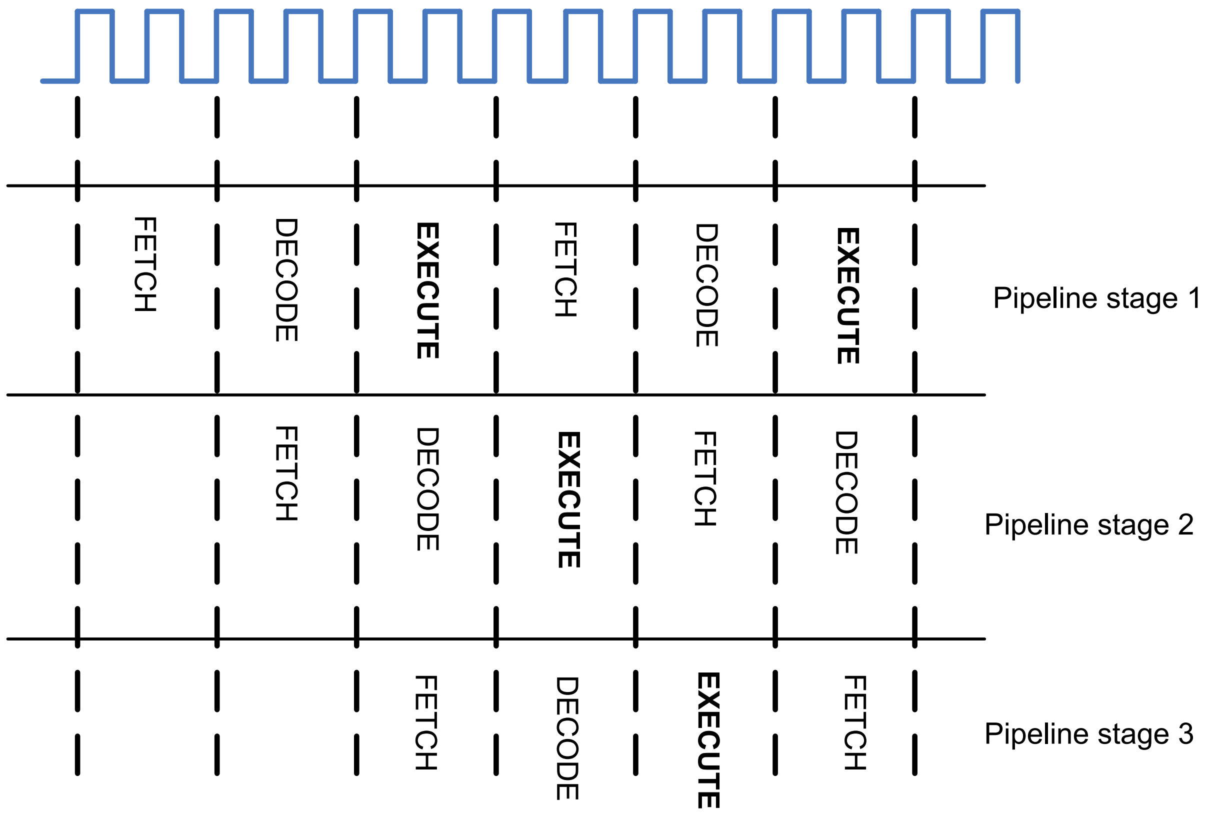


Figure 4 - Showing the throughput of a 3-stage pipeline (with 2-waitstates)

Refer to page 1 of the PIC24 datasheet (title “General Purpose, 16-bit Flash Microcontrollers”). You will observe a table at bottom of the page. Two columns of particular interest are “program memory” and “sram”.

- 128K Bytes of program memory (64K words) of which 44,032 (words) are available for instructions
- 8K Bytes data memory (4K Words)

This illustrates one of the major differences between dedicated embedded systems and a desktop computer!

VARIABLES IN C

A lot of effort in programming involves manipulating variables[4] (data that can change) – alongside the internal CPU registers, they define the 'state' of any computer program at given time. Variables, by definition change their value. **Variables are stored in data memory** (constants are usually stored in program memory – see later). Variables in C must be declared so the compiler can reserve or 'allocate' space in data memory. Some examples are shown below:

```
//Global variables - available to ALL functions in the software
double p;                  //a double-precision floating point number
long y = 0;                //a long-integer initialised to 0
const float PI=3.1415926; //a constant floating point number

main()
{
 //local variables - only visible to this function
 int x=1;                  //a single integer, initialised to 1
 char c;                   //a single character, not yet initalised
 double two_pi=2.0*PI;     //double precision floating point number

}
```

In the code above, you might notice that there are two groups of variables – **global** and **local**. Global variables are always visible from any function that may wish to use them. Local variables however are only visible within the function in which they are declared. This has the advantage that variable names can be reused in different functions without name clashes.

Note: For this tutorial, we will postpone discussion about pointer variables until a later date.

[4] Sometimes you hear the expression "state"

TASK 1.1 – INSPECTING VARIABLES IN DATA MEMORY

In this task, we will create an integer in C and look at how it is stored. Hopefully, this will help de-mystify what the C compiler is actually doing. This is fundamentally the same as any other imperative programming language.

1. Watch the video "Task 1-1 1080p.mp4" (on Moodle and iTunesU).
2. Download and unzip Task.zip on your Desktop. Use 7-zip to do this.
3. Run MPLABX, and open example1.1 – This includes a simple circuit to simply switch on LED's connected to a digital output port.
4. Build the project and single step the line that reads "PORTA = uOutputPattern;"
5. Put a **watch** on the variable **uOutputPattern** (as shown in the video) and note its address and its value
6. In the File Registers view (data memory)
    a. Scroll to the address of **uOutputPattern** and verify you can see the correct value
7. You can also modify memory directly in this view (useful for debugging)
    a. Try changing the value of **uOutputPattern** to 4 in the file registers view
    b. Confirm it has also changed in the watch window
8. Step through the code to understand what it does. View the Proteus simulator to see what it does.

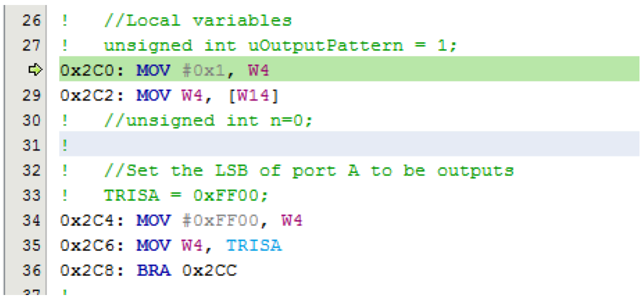
Question	**Answer**
From the file-register window, what are the first and last addresses in the data memory?	
What does the line `uOutputPattern = uOutputPattern << 1;` actually do? (Hint – view `uOutputPattern` in binary format)	

TASK 1.2 – VIEWING PROGRAM CODE AND DATA MEMORY FOR A SIMPLE VARIABLE

In this task, we take a brief look 'under the hood' to see how the compiler organises **local variables** in memory. You will need to have seen the lecture slides on function stacks before attempting this.

Using the same project as in the previous section, follow these instructions: (and brace yourself)

1. Reset the code
2. Break point the line that says `unsigned int uOutputPattern = 1;`
3. Run the code and click on Window->Debugging->Disassembly Listing. You should see something *similar* to the following (depending on the version of MPLAB):



```
!   //Local variables
!   unsigned int uOutputPattern = 1;
0x2C0: MOV #0x1, W4
0x2C2: MOV W4, [W14]
!   //unsigned int n=0;
!
!   //Set the LSB of port A to be outputs
!   TRISA = 0xFF00;
0x2C4: MOV #0xFF00, W4
0x2C6: MOV W4, TRISA
0x2C8: BRA 0x2CC
!
```

Inside this window, right-click and select "Disassembly File". Here you can see what the C-compiler has produced, including the machine code itself. Put simply, the compiler has converted C-code into lines of **assembly code**, and from this, the assembler has generated **machine code**.

To try and demystify this, I will talk you through it....

   a. The line mov.w #0x1, W4 stores the constant 1 into **Special Function Register** (SFR) W4
   b. The line mov.w W4, [W14] stores the contents of W4 into another data memory address ... only this time, the destination **address** is stored in SFR W14. Indeed, if you look at W14, you will see the value 0806h, which is the **frame pointer** address.
   This is because the **local** variable `uOutputPattern` is on the main **function stack**.

4. Step over this line in the normal editor and inspect address 0806h in data memory
5. You can also view W14 by clicking Window->PIC Memory Views->SFR's

To (try) and summarise, **register W14 is the frame pointer** – this is the start address of the function stack. Stored in stack memory are the local variables, function arguments etc. In this case, the first item on the stack is the content of the variable `uOutputPattern`.

TIP: You can always reference the address of a (value-type) variable by prefixing it with an ampersand &.

Confused? **DON'T PANIC!** Does all this seem rather complex? The good news is... you don't normally have to examine assembly language. You can safely let the C-compiler sort it out for you.

TASK: Contrast this with Java or C#. What assembly language platform do these language compilers generate?

INTRODUCTION TO ARRAYS

So far all we have managed to store is a single integer. There are many occasions where you want to store an ordered list of data in memory.

For example, we might want to store the last 8 samples room temperatures in memory. We know that (i) we need to store 8 values and (ii) the order is important. This is known as an **array**. The C-language was designed to manage arrays very efficiently.

To declare an integer array, you simple add square braces as follows:

```
int <variable name>[Number of elements];
```

So, for example,

```
int x[8];
```

allocates space for 8 integers. To write to an element of an array, you must provide an index, so for example:

```
x[0]=10;
```

writes the value 10 into the first element of the array. Similarly,

```
x[7]=99;
```

writes the value 99 into the last element of the array. You can read the array back in the same way:

```
p = x[3];
```

reads the 4th element in the array and stores it in the variable p.

TASK 2.1 – INTEGER ARRAYS IN MEMORY

In this task we look at how arrays are stored in physical memory.

1. In folder Task2, open Example2.1 and inspect the code in main.c
2. Run the code a few times to try and understand what it does
3. Breakpoint the line `for (n=0; n<8; n++) {`
4. Re-run the code – it should stop at the breakpoint.

Question: At what address is the array variable `mydata` stored in data memory?	**Answer**:

5. Now set a break point on the line that reads `n = 0;` and run the code.
6. Inspect the data memory (WIndow -> PIC Memory Views -> File Registers) to find out what data is now stored in the array

Question: Use the watch window to read the contents of the array **mydata**. Write each of these values down **and** its corresponding address in data memory Does it match what you expected?	**Answer**:
Question: View the data memory (File Registers) and find the array mydata in that view Why are the addresses for each element in the array incrementing in 2's? How much memory space (in bytes) does the array consume?	**Answers**

Question: The initial values were created with the following code:
 	
`for (n=0; n<8; n++) {`
`   mydata[n] = (1 << n);`
`}`
 	
Explain what the mathematical expression (1 << n) actually does.
 	
Hint 1: ' << ' is one of the standard C operators (alongside +,-,*,/,%,==)
	Answer:

Question: In C, you refer to the nth element of an array mydata as mydata[n]. If X = **address** of the first element of the array mydata, and I give you n (the index), write a formula for the address of the nth element X[n] (it is a function of X and n) (Use MPLAB to check your answer)	**Answer:** Address of mydata[n] =

In this task, we saw how an array of integers is simply a reserved linear block of memory, with each integer taking up 2-bytes (16 bits) of data. Next we will look at other data types.

TASK 2.2 OTHER VARIABLE TYPES

In this example, we look at an array of alternative data types and compare how this is stored in memory with the previous example.

1. Open Example2.2
2. Breakpoint and run to the line that reads `asm("nop");`
3. Inspect mydata in the watch window

Question	**Answer**
How much memory space does the character array `mydata` consume?	
Look at the address of each element – by what quantity (in bytes) does it increment?	
From the above, how many bytes does a single char data type consume?	

4. Now, by changing the variable type of your array `mydata`, complete the table below. You might find the results somewhat unexpected.

Type name	**Description**	**Type Size (bytes)**
char	Signed character	
short	Short integer	
int	Integer	
long	Long integer	
float	Floating point value	
double	Double-precision floating point	
long double	Extended-precision floating point	
long long	Long long integer	

You should note that the C data-type sizes **depend on the compiler**. See the Microchip X16 compiler documentation for further discussion.

As hinted at earlier, sometimes we wish to initialise an array with values at the start of a program or a function. In the next task we look at how this is done in the C language.

TASK 2.3 – INITIALISATION OF ARRAYS

In this task we see how the compiler sometimes initialises arrays of data for us. We also see how the compiler performs this for us, how it hides this from us, and how this can be an inefficient use of program memory.

We return to character data types. An array of characters is known as a **string**. We declare a string of 10 characters as follows:

```
char mystring[10];        //declare a string without initialisation
```

Alternatively, we can **declare and initialise** a string (and let the compiler work out the length) as follows:

```
char[] mystring = “hello world”;
```

We shall now look at how the compiler manages both these scenarios.

1. Open example2.3
2. Add a break point to the line that reads `for (n=0; n<10; n++) {`
3. Run the code and put a watch on the string `strMystring` (Make a note of the address)

Address of strMystring =	

4. Now step through the code and watch the string (array of characters) fill with characters.

What has happened here is that you, the programmer, have decided to populate the string (character array) yourself programmatically.

5. Now stop the code, and replace the line that reads `char strMystring[10];` with the following
   ```
   char strMystring[] = "Hello world";
   ```
6. Run your code again (keeping the break point) and inspect the variable `strMystring` in the watch window.

In this latter case, the compiler has somehow initialised the character array (string) for you. There is a memory implication doing this however. Data ram is not persistent (it is wiped when power is lost) – only program memory is persistent (flash memory), so a persistent copy of the constant string needs to be stored in the program image (in flash)

WHEN YOU BUILD YOUR APPLICATION, THE INITIALISER STRING “HELLO WORLD” IS STORED IN FLASH PROGRAM MEMORY AS PART OF THE APPLICATION IMAGE.

WHEN YOU RUN THE APPLICATION, THE CODE WILL AUTOMATICALLY COPY THE STRING FROM PROGRAM MEMORY TO DATA MEMORY (THIS PROCESS IS HIDDEN FROM YOU). ALTHOUGH THIS CREATES MUCH MORE EFFICIENT CODE[5], IT ALSO CREATES DUPLICATION.

[5] A write to flash memory is VERY slow, and there are a finite number of writes that can be performed before it begins to fail.

(optional) Advanced task	
Using the dissassembly view, can you find the code that actually copies the data from program memory to data memory? Hint – it is run before your code in main	

When space is at a premium, you may look to ways to save valuable data memory. If your string is not actually going to change (if it is constant), then you can prevent the compiler from copying data across by using the keyword **const**.

7. Now stop the code, and move the line that reads `char strMystring[] = "Hello world";` to a line **before** the main function. Now prefix with const as follows:

```
const char strMystring[] = "Hello world";
```

8. Build the code and you will find there are errors
   a. The line `strMystring[n] = 'x';` This has to be commented out (you cannot write to a constant string)
   b. The declaration of constant strings must also come before main()

9. Run the code and note the address of the string `strMystring` in the watch window.

Address of (constant) strMystring =	
Note: The (**PSV)** in the watch window refers to "Program Space Visibility". This indicates the data is in program memory	

Initialising arrays of other data types is very similar. Here are some examples using numerical data:

```
double x[] = {0.0, 1.0, 3.0};
int xx[] = {10, 20, 100};
```

The compiler selects the correct array length for you in such cases. If you try and write data over the end of an array, you will most likely corrupt another variable. So for example, the following are legal:

```
x[0] = 1.0;
x[2] = 10.0;
```

The following is not legal and will corrupt your memory:

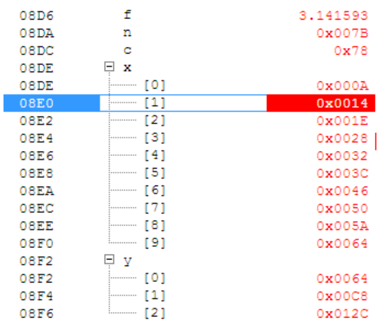
```
x[3] = 1.0
```

You will get no warning about this. **C is designed to be fast, so it does not perform array bounds checking**.

TASK 2.4 – MEMORY CORRUPTION

In this exercise you will observe one of the biggest dangers of the C-language – **memory corruption**.

1. Open example2.4
2. Break point the line that reads `x[0]=123;`
3. Run the code.
4. Click the Variables tab and expand arrays x and y. You should see *something* similar to the figure opposite
5. Step over the next three instructions, watching their contents as you do it



```
08D6       f              3.141593
08DA       n                0x007B
08DC       c                  0x78
08DE     ⊟ x
08DE        [0]             0x000A
08E0        [1]             0x0014
08E2        [2]             0x001E
08E4        [3]             0x0028
08E6        [4]             0x0032
08E8        [5]             0x003C
08EA        [6]             0x0046
08EC        [7]             0x0050
08EE        [8]             0x005A
08F0        [9]             0x0064
08F2     ⊟ y
08F2        [0]             0x0064
08F4        [1]             0x00C8
08F6        [2]             0x012C
```

Question	Answer
What corruption did you observe?	
Explain how memory corruption has occurred	
Is it possible to corrupt your program code in this way? ... explain	

LOCAL AND GLOBAL VARIABLES

You might have noticed the watch window sometimes displays an error "out of scope". This is caused by watching a variable not either (i) does not exist or (ii) did once exist, but has been dropped as it is no longer needed. This is evidence that the compiler is trying to save memory without changing the function of your code.

- Variables that go in and out of scope can only be 'local' variables. These are variables declared inside a function (such as main, but it could be any function you write).
- Variables declared outside any function are global and are always in scope.

There are in fact two regions of memory – the heap and the stack, but more on that in the next session.

CHALLENGE

Design a digital dice.

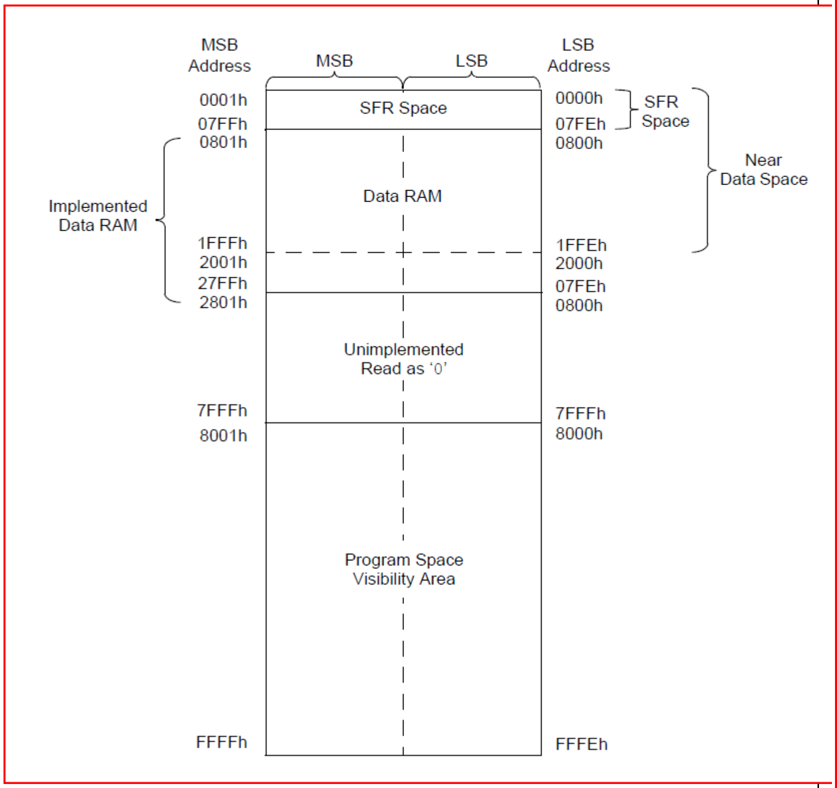
1. In the task3 folder, open example3.1.
2. Single step the code
3. Using the help, find the documentation for the function ‘rand’
4. Use the rand function to calculate a random number between 1 and 6 and display it on the 7-segment display

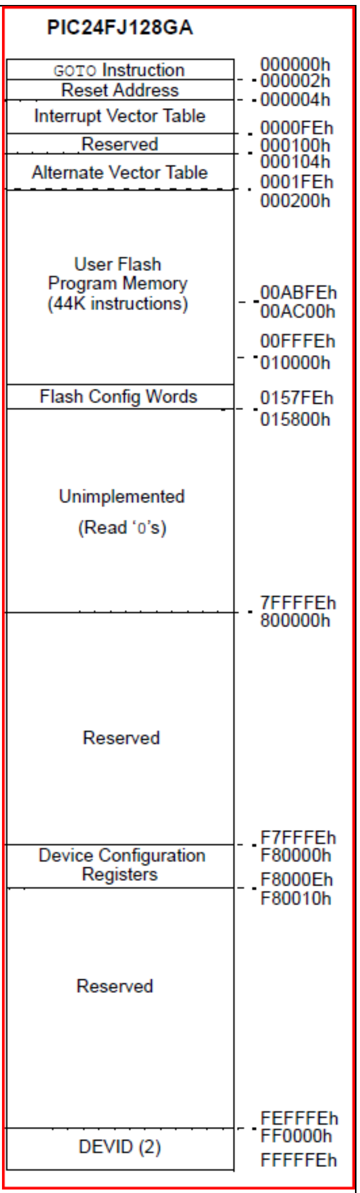
Tip:

- Use the ‘%’ operator to control the range of the random number

A MORE DETAILED LOOK AT THE PIC24 MEMORY MAP

In this section, we examine the layout of memory in the PIC24 microcontroller. Although this is specific to one device from one manufacturer, it is not untypical for a device in this class.


MSB Address
MSB
LSB
LSB Address
0001h
07FFh
0801h
SFR Space
0000h
07FEh
0800h
SFR Space
Near Data Space
Implemented Data RAM
Data RAM
1FFFh
2001h
27FFh
2801h
1FFEh
2000h
07FEh
0800h
Unimplemented Read as '0'
7FFFh
8001h
7FFFh
8000h
Program Space Visibility Area
FFFFh
FFFEh


PIC24FJ128GA
GOTO Instruction
Reset Address
Interrupt Vector Table
Reserved
Alternate Vector Table
User Flash Program Memory (44K instructions)
Flash Config Words
Unimplemented (Read '0's)
Reserved
Device Configuration Registers
Reserved
DEVID (2)
000000h
000002h
000004h
0000FEh
000100h
000104h
0001FEh
000200h
00ABFEh
00AC00h
00FFFEh
010000h
0157FEh
015800h
7FFFFEh
800000h
F7FFFEh
F80000h
F8000Eh
F80010h
FEFFFEh
FF0000h
FFFFFEh