Fundamentals of Embedded and Real Time Systems

MODULE 05

TAMER AWAD

Module 05

Bit-banding
- Read-Modify-Write
- What is Bit-Banding
- Mapping bit-banding addresses
- *Demo:* Bit-banding

C Programming (continued)
- Arrays and pointers
- *Demo:* Arrays & Pointers

C Programming (continued)
- Functions and the C-Stack
- *Demo* Functions and Stack

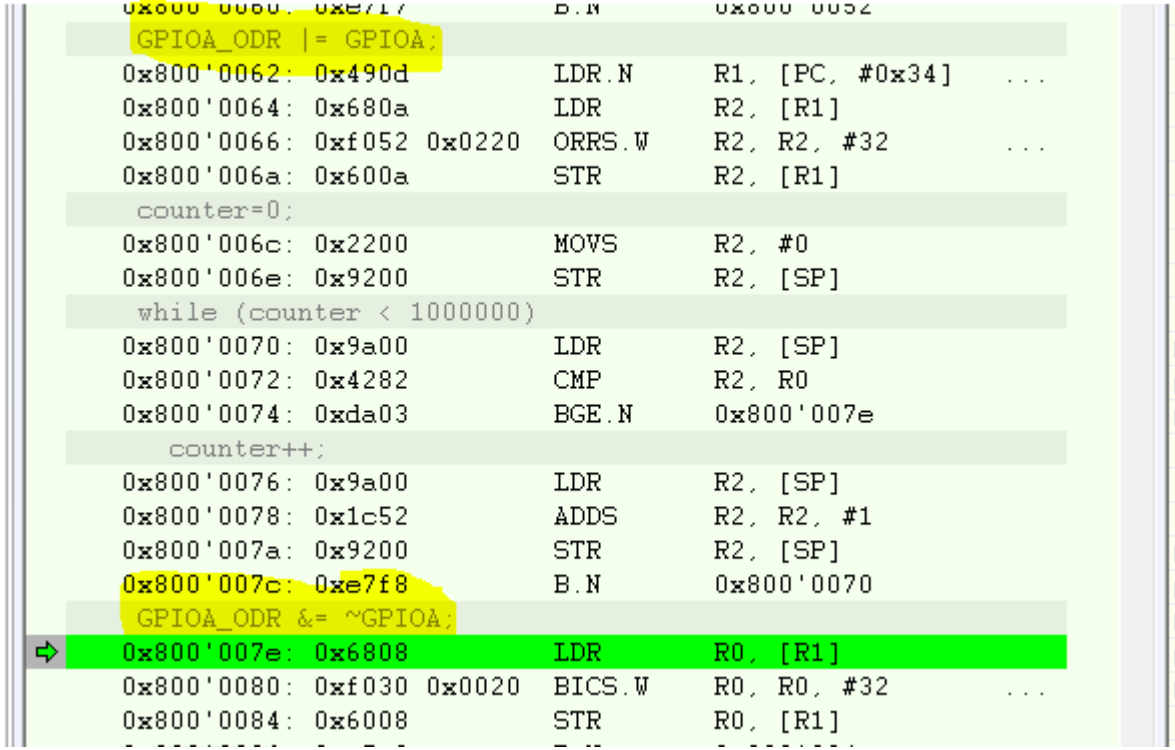
Putting concepts together (building a FIFO):
- FIFO requirements
- FIFO APIs
- FIFO design
- FIFO implementation
- FIFO testing
- Code organization

Assignment 04

Bit-Banding

- Read-Modify-Write
- Interrupts
- What is Bit-Banding
- Mapping bit-banding addresses
- Bit-Banding Demo

Read-Modify-Write



Why “Read-Modify-Write”

8.4.6 GPIO port output data register (GPIOx_ODR) (x = A..E and H)

Address offset: 0x14

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ODR15	ODR14	ODR13	ODR12	ODR11	ODR10	ODR9	ODR8	ODR7	ODR6	ODR5	ODR4	ODR3	ODR2	ODR1	ODR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:16 Reserved, must be kept at reset value.

Bits 15:0 **ODRy**: Port output data (y = 0..15)

These bits can be read and written by software.

Note: For atomic bit set/reset, the ODR bits can be individually set and reset by writing to the GPIOx_BSRR register (x = A..E and H).

- Register shared by different Ports/GPIOs
- Cannot “directly” address specific bits inside that register

Source: RM0368

Interrupts

- Interrupts are a ***hardware*** mechanism for a processor to abruptly change the flow of control of the program.
- When an interrupt occurs, a special hardware in the processor changes the value of the program counter register so that the processor suddenly starts executing a different piece of code, called the interrupt service routine or ISR
- ISR is typically short.
- When the ISR ends, the processor resumes the execution of the original code as if nothing happened.

So what's the problem?

- FunctionFoo()
  - {
    - Reads GPIOx_MODER // bit[5] == 1 (for Example)
  - …
- **An interrupt occurs and invokes ISRx()**
- ISRx()
  - {
    - Reads GPIOx_MODER
    - Modifies **bit[5]** // bit[5] = 0
    - Writes back new value to GPIOx_MODER
  - }
- FunctionFoo()
  - //Resumes
    - Modifies **bit[10]**
    - Writes back value to GPIOx_MODER // At this point, bit[5] = 1 (which is wrong)
  - }

What is Bit-Banding

- Special feature introduced in Cortex-M3 and available in M4 processors
- For the following bits stored between:
  - 0x2000.0000 and 0x2010.0000 in SRAM, or
  - 0x4000.0000 and 0x4010.0000 for peripherals
- Each bit has a corresponding alias address which can be used to reference that one bit specifically.

2.2.5 Optional bit-banding

A bit-band region maps each word in a *bit-band alias* region to a single bit in the *bit-band region*. The bit-band regions occupy the lowest 1MB of the SRAM and peripheral memory regions.

The memory map has two 32MB alias regions that map to two 1MB bit-band regions:

- accesses to the 32MB SRAM alias region map to the 1MB SRAM bit-band region, as shown in Table 2-13
- accesses to the 32MB peripheral alias region map to the 1MB peripheral bit-band region, as shown in Table 2-14.

Table 2-13 SRAM memory bit-banding regions

Address range	Memory region	Instruction and data accesses
0x20000000-0x200FFFFF	SRAM bit-band region	Direct accesses to this memory range behave as SRAM memory accesses, but this region is also bit addressable through bit-band alias.
0x22000000-0x23FFFFFF	SRAM bit-band alias	Data accesses to this region are remapped to bit band region. A write operation is performed as read-modify-write. Instruction accesses are not remapped.

Table 2-14 Peripheral memory bit-banding regions

Address range	Memory region	Instruction and data accesses
0x40000000-0x400FFFFF	Peripheral bit-band alias	Direct accesses to this memory range behave as peripheral memory accesses, but this region is also bit addressable through bit-band alias.
0x42000000-0x43FFFFFF	Peripheral bit-band region	Data accesses to this region are remapped to bit band region. A write operation is performed as read-modify-write. Instruction accesses are not permitted.

Bit-banding is optional

Source: Cortex M4 Generic User Guide **DUI0553A**

Mapping bit-banding addresses

A mapping formula shows how to reference each word in the alias region to a corresponding bit in the bit-band region. The mapping formula is:

bit_word_addr = *bit_band_base* + (*byte_offset* x 32) + (*bit_number* × 4)

where:

- *bit_word_addr* is the address of the word in the alias memory region that maps to the targeted bit
- *bit_band_base* is the starting address of the alias region
- *byte_offset* is the number of the byte in the bit-band region that contains the targeted bit
- *bit_number* is the bit position (0-7) of the targeted bit

Source: Reference Manual RM0368

Example

The following example shows how to map bit 2 of the byte located at SRAM address 0x20000300 to the alias region:

0x22006008 = 0x22000000 + (0x300*32) + (2*4)

Writing to address 0x22006008 has the same effect as a read-modify-write operation on bit 2 of the byte at SRAM address 0x20000300.

Reading address 0x22006008 returns the value (0x01 or 0x00) of bit 2 of the byte at SRAM address 0x20000300 (0x01: bit set; 0x00: bit reset).

Summary of steps for blinking LED

// 1. Enable clock

// RCC_BASE = 0x40023800 // Base Address for RCC registers

// RCC_AHB1_ENR offset: 0x30 // Peripheral Clock Enable Register

// Set bit[0] to 1

// 2. Set GPIOA to Output mode

// GPIOA_BASE = 0x40020000 // Base Address for GPIO registers

// GPIOx_MODER offset is 0x00 // To enable port mode (IN, OUT, AF..)

// Set bit[11:10] to 0x01 so --> 0x400 // To enable Port5 as output

// 3. Write to GPIO Data Register to toggle LED

// GPIOA_BASE = 0x40020000 // Base Address for GPIO registers

// GPIOx_ODR offset: 0x14 // Port output data register

// Set bit[5] to 1 --> 0x20; // Turn LED ON

// Set bit[5] to 0 --> 0x00; // Turn LED OFF

Demo Bit-Banding: RCC_AHB1_ENR

Writing to **RCC_AHB1_ENR** :

0x400**23830**

Set bit[**0**] to 1

bit_word_addr = *bit_band_base* + (*byte_offset* * 32) + (*bit_number* * 4)

bit_band_base = 0x4200.0000

byte_offset = 0x23830

bit_number = 0

bit_word_addr = 0x42000000 + (0x23800 * 32) + (0 * 4);

Before & After

NOT USING BIT-BANDING

```
int main()
{
main:
   0x800'0040: 0xb081          SUB       SP, SP, #0x4
    RCC_AHB1ENR |= 0x1;
   0x800'0042: 0x4813          LDR.N     R0, [PC, #0x4c]   ...
   0x800'0044: 0x6801          LDR       R1, [R0]
   0x800'0046: 0xf051 0x0101   ORRS.W    R1, R1, #1
   0x800'004a: 0x6001          STR       R1, [R0]
```

USING BIT-BANDING

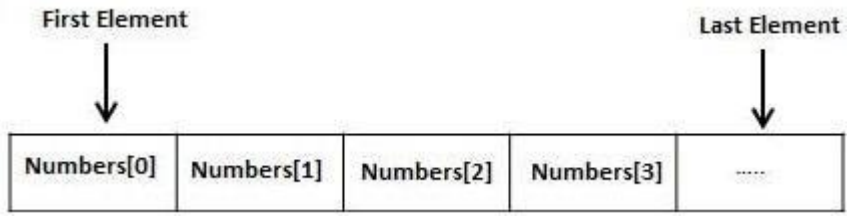
```
main:
   0x800'0040: 0xb081          SUB       SP, SP, #0x4
    *((unsigned int*)(0x42000000+(0x23830 *32) + (0*4))) =...
   0x800'0042: 0x2001          MOVS      R0, #1
   0x800'0044: 0x4911          LDR.N     R1, [PC, #0x44]   ...
   0x800'0046: 0x6008          STR       R0, [R1]
```

Arrays & Pointers

- Arrays in C
- Arrays & Pointers
- Passing arrays in functions
- The sizeof operator
- Demo: Arrays & Pointers

Arrays in C

- What is an array?
  - An array is a group of variables of the same type occupying consecutive memory locations
- How to declare an array?
  - ***int numbers[10];***
  - The First element in the array starts at index 0
- How to initialize an array?
  - ***int numbers[5] = {1000, 2, 3, 7, 50};***
  - ***int numbers[] = {1000, 2, 3, 7, 50};***
- How to access array elements?
  - ***numbers[4] = 50;***
  - ***Int myNumber = numbers[9];***



Arrays & Pointers

- Arrays in C are closely related to pointers.
- The C compiler treats an array as a pointer, which points to the beginning of the array.
  - **int myIntArray[5] = {40, 41, 42, 43, 44};**
  - **int *myIntPtr = myIntArray;**
- To get a pointer to an element with index i, you simply need to add i to the array pointer.
- So instead of myArray[1] you can write ***(myArray + 1)** // Example of pointer arithmetic
  - **int myInt = *(myIntPtr+1);**
- Every pointer can also be viewed as an array.

Passing arrays as function arguments

- Passing arrays as functions arguments

```
/* passing as a pointer */
void printAarry(int *param, int size) {
    ....
}
```

```
Int SIZE 5;
int numbers[SIZE] = {1, 2, 3, 4, 5};
int *p = numbers;
printArray(p, SIZE);
```

```
Int SIZE 5;
int numbers[SIZE] = {1, 2, 3, 4, 5};
printArray(&numbers[0], SIZE);
```

The **sizeof** operator

- The **sizeof** operator is a compile-time operator that yields the size (in bytes) of its operand
- The size is determined from the type of the operand.
- The result is an integer.
- When applied to an operand that has type char, unsigned char, or signed char, the result is 1.
- When applied to an operand that has array type, the result is the total number of bytes in the array.
  - One use is to compute the number of elements in an array: **sizeof array / sizeof array[0]**

WATCH OUT!

- When applied to a pointer to an array, or to a parameter passed to function as a pointer to an array, the **sizeof** operator yields the size of the pointer type.

Demo Arrays & Pointers

- Pointer arithmetic (Ex: int, char)
- Arrays
- sizeof

```c
int myIntArray[5] = {40, 41, 42, 43, 44};
char myCharArray[] = "Hello my array";
char *myString = "Hello my string";


int *myIntPtr = myIntArray;

int getIncorrectArraySize(int* p);

int main()
{
    // Poitner arithmetic
    int myInt = 0x77778888;
    int* pInt = &myInt;
    *pInt = 0x41424344;
    pInt++;     // What is the value of pInt?
    char* pChar = (char*)&myInt;
    *pChar = 0x00;  // Hint for Assignment 04
    pChar++;    // What is the value of pChar?

    // Pointer arithmetic with arrays
    *myIntPtr = 70;
    myIntPtr[1] = 2000;
    *(myIntPtr+1) = 72;
    *(myIntPtr+5) = 77;
    myIntPtr++;
    *myIntPtr = 100;


    // sizeof operator
    int numberOfElements = sizeof(myIntArray)/sizeof(myIntArray[0]);
    numberOfElements = getIncorrectArraySize(myIntArray);
    numberOfElements = sizeof(myIntPtr)/sizeof(myIntPtr[0]);

    (void)numberOfElements;
    return 0;
}

int getIncorrectArraySize(int* p)
{
    return sizeof(p)/sizeof(p[0]);
}
```

Demo Arrays & Pointers

BREAK 1

Functions

- C Functions
- CSTACK
- BL instruction
- Demo: Functions & CSTACK

C Functions

> Function signature: A name, a list of arguments, and a return type.

> Calling the function means changing the flow of control

1. **Jump** to the beginning of the function code.
2. **Execute** the function code
3. **Return** to the next instruction after the call

> **Arguments** allow the caller to specify initial values of the local variables at the point of the function is called.

C Functions

> Function prototypes
  - Declare the function name and argument list
  - Allow the compiler to type-check arguments to function calls
  - Are generally declared
    > In a header (*.h) file for public functions
    > In an implementation file for private functions

> Function definitions
  - Implement the function
  - Contain executable instructions
  - Are always declared in an implementation (*.c) file
  - Parameter names need not agree. Indeed, parameter names are optional in a function prototype, so for the prototype we could have written

C Functions: Declaring a Function

For public functions, the pattern is:
- Prototype in header
- Definition in implementation file

```
// file header.h
/*
 * myFunc - performs a function
 *
 * params:  none
 * returns: none
 */
void myFunc( void );
```

```
#include "header.h"

void myFunc( void ){
  // C statements in here to do something
}
```

C Functions: void Arguments

Functions are declared to accept a list of arguments

- They're called **parameters** (or formal parameters) in the function declaration
- They're called **arguments** when calling code passes in values

C allows an empty list

- This can cause problems for Java and C# developers moving to C An empty argument list defeats type checking!
- Use the **void** keyword to declare a function that takes no arguments
  - In a header file for public functions
  - In an implementation file for private functions

When calling a function

- Pass values as arguments, if any, into the function
- Read or discard the return value

```
// simple math operation to demonstrate function calling

// create some arguments
int a = 10;
int b = 20;

// and a place to hold the result
int c;

// now call the function, passing in the arguments and storing the result
c = multiply( a, b );  // c = a * b
```

CSTACK

- Used for:
  - 1) Holding the local variables &
  - 2) Storing the return addresses
- The function starts with adjusting the SP register.
- The SP is the hardware implementation of the C call stack mechanism
- A C stack is an area of RAM that can grow or shrink from one end only.
- The end is called the top of the stack and the SP register contains this top address.
- In the ARM processor
  - The stack grows towards the lower addresses (which is up in the memory view)
  - And shrinks towards to high addresses (which is down in the memory view).
- In other processors, the stack might grow in the opposite direction.

Assembly: BL instruction

- The BL instruction is a branch to “label” or the address indicated in “Rm”.
- Changes the value of the PC register
- It will also write the address of the next instruction to LR (the Link Register, R14)
- BL is a 32-bit instruction
- Cortex-M4 supports “Thumb2” ISA which is a mix of mostly 16-bit instructions and some 32-bit instructions
- The least significant bit in LR is interpreted as the instruction set “exchange” bit.

Reference: Cortex-M4 Devices Generic User Guide

The LSb in LR

- The least-significant bit in the PC is always set to zero (b/c return address must be even).
- So instead of being used for addressing, the least-significant bit in LR is interpreted as the instruction set exchange bit.
- If this bit is 1, the processor switches to the THUMB instruction set
- If it is zero it switches to the ARM instruction set.
- ARM Cortex-M supports only the THUMB2 instruction set, and cannot really switch to ARM.
- So in Cortex-M this behavior of the BX instruction is “unused”.

Demo “Delay()” Function & CSTACK

1. Optimization enabled
2. Optimization disabled
3. Prototype Required //Compiler option
4. “void” parameter
5. BL instruction and its effects
6. SP & the function stack
7. Stack views
8. View the “counter” variable on the stack
9. LR register
10. Add a new leaf function (setFlag()) & show assembly code changes to preserve LR of the non-leaf function via PUSH & POP
11. Add function argument and show how parameters are passed.

Assignment question: Arguments?

- What does the compiler do when a large number of parameters need to be passed to a function?
- Try something like the code on this slide
- Trace thru the assembler
- Note down your observations

```
int sum(int var0, int var1, int var2, int var3, int var4)
{
    int lvar0;
    int lvar1;
    int lvar2;
    int lvar3;
    int lvar4;

    int sum;


    lvar0 = var0;
    lvar1 = var1;
    lvar2 = var2;
    lvar3 = var3;
    lvar4 = var4;

    sum = lvar0 + lvar1 + lvar2 + lvar3 + lvar4;
    return sum;
}
```

BREAK 2

Putting concepts together – *Building a FIFO Queue*

- FIFO requirements
- FIFO APIs
- FIFO design
- FIFO implementation
- FIFO testing
- Code organization

FIFO Queue - Requirements

1. Implement a FIFO Queue data structure in C
2. FIFO has a predefined size
3. FIFO supports char data types
4. Provide function to put a new char data into FIFO
5. Provide function to get the oldest element from FIFO
6. Provide function to check if FIFO is empty
7. Provide function to check if FIFO is full
8. Return error (-1) if attempting to add a new element when FIFO is full
9. Return error (-1) if attempting to retrieve an element from an empty FIFO
10. Otherwise, return success (0).

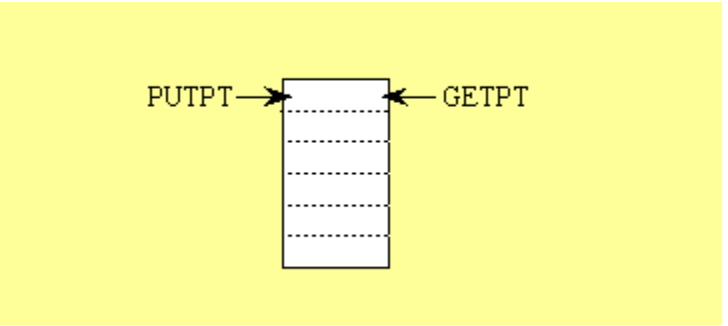
FIFO Queue - APIs

- **int queue_put(char data);**
  - Return value used to indicate error or success

- **int queue_get(char* data);**
  - // Return value used to indicate error or success
  - // data is also used as a return value

FIFO Queue - Design

- Backing store?
  - Array
- How to keep track of elements for “adding” to the queue or “retrieving” from the queue?
  - Two pointers:
    - One tracks which elements to retrieve from the queue
    - One tracks where to add the next element in the queue
- Now we need to think who will we initialize those pointers?
  - Option: add new “init” type of API
  - ***void queue_init();***



Source: Jonathan W. Valvano and Ramesh Yerraballi (1/2015)

FIFO Queue – Implementation & Testing

- Implement one method at a time
- Test the method
- Iterate to fix issues
- Implement next method

Demo: Implementation of FIFO Queue

```
#define QUEUE_SIZE 10

// Backing store
char QueueStore[QUEUE_SIZE];

// Pointers to queue elements for adding and retrieving
char* putPtr;
char* getPtr;
```

Demo: Implementation of FIFO Queue

```
// Initialize internals of the queue
void queue_init()


// Add data to queue
// If queue is full return error
// FIFO Queue is Full if putPtr+1 > Address of last element in the queue;
// Otherwise return success
int queue_get(char* data)
```

```
// Get data from queue
// If queue is empty return error
// FIFO Queue is Empty if putPtr==getPtr;
// Otherwise return success
int queue_put(char data)
```

Unit Testing: AAA

“Arrange, Act, Assert" is a testing pattern to describe the natural phases of most software tests.

Arrange describes the setup needed for the test

Act invokes the actions of the test

Assert describes the verification of the test

```
// Test1: Successfully retrieve an element from the queue
    // Arrange (setup)
    queue_init();

    // Act (execute)
    queue_put('A');

    // Assert (verify)
    assert(0 == queue_get(&testChar));
    assert('A' == testChar);
```

Queue Test cases:

1. get() from an empty queue
   - Expect queue_get() to return failure
2. put() one item, and get() item back
   - Expect to get that item and both put() & get() return success
3. Fill queue, then get() all items
   - Expect to get all items in order
4. Fill queue, then put() one more item
   - Expect: queue_put() to return -1
5. Fill queue, get() one item, then put() one item, then get() all items
   - Expect to get all items in order

The “assert” marco

7.2.1.1 The `assert` macro

Synopsis

```
#include <assert.h>
void assert(scalar expression);
```

Description

The **assert** macro puts diagnostic tests into programs; it expands to a void expression. When it is executed, if **expression** (which shall have a scalar type) is false (that is, compares equal to 0), the **assert** macro writes information about the particular call that failed (including the text of the argument, the name of the source file, the source line number, and the name of the enclosing function — the latter are respectively the values of the preprocessing macros **__FILE__** and **__LINE__** and of the identifier **__func__**) on the standard error stream in an implementation-defined format.[165] It then calls the **abort** function.

Returns

The **assert** macro returns no value.

Demo: Testing reveals a problem

What happens when we add elements beyond the queue size (queue putPtr & getPtr are both reach the end of the queue), can we add more elements?

> Need a way to reset the pointers back.

Demo: Testing reveals next problem

What happens if we add elements to fill the queue, we get one element, then we attempt to add one more element?

> Need a way to wrap pointers around.

> Need a way to recognize queue is not empty when getPtr==putPtr after wrapping around.

Code organization

SPLIT INTO .C AND .H FILES

Each module in its own .h & .c files

Prototypes of a module should be in their own .h file (queue.h)

Implementation in it's own .c file (queue.c)

Protect against multiple inclusions:

```
#ifndef __QUEUE_H__
#define __QUEUE_H__
<prototypes>
#endif
```

Assignment 04

Suggested Reading

- ***"The C Programming Language" By Brian Kernighan & Dennis Ritchie (Second Edition)***
  - Chapters: 4, 5
- ***"The Cortex-M4 Device Generic User Guide"***
  - Chapter 2.2.5
- ***"The Definitive Guide to ARM Cortex M3 & M4" by Joseph Yiu (Third Edition)***
  - Chapter 3.1

Reference: Pointers and Address