uOttawa

L'Université canadienne
Canada's university

CEG3136
Computer Architecture II

Software Development using C

Notes for

Dr. Voicu Groza

Université d'Ottawa | University of Ottawa

Topics of discussion

- Programming in C
  - Basics: Variables, Operators, Statements
  - Functions
  - Pointers and Structures
  - Standard Functions
- Creating Program Modules
- Extensions for Embedded Systems Software
- Reading: Chapter 10

A Simple C Program

- A C program contains
  - Comments enclosed in /* */ characters
  - Preprocessor directives executed before compilation
  - A set of functions - processing starts with the *main()* function
  - C statements are executed within the functions and consist of:
    - Assignment statements
    - Arithmetic and logical expressions
    - Control statements
    - Calls to to other functions

```
/* Comments are enclosed as shown */
/*-------------------------------------------
FILE: simple.c
Description:  simple program.  Note
              how comments can be
              visual.
-------------------------------------------*/
#include <stdio.h>  /* Preproc.directive */

void main(void)
{
   /* Stmt is call to standard function */
     printf("Hello world\n");
}
```

Things you should know already

- Basic variable types: char, int, float, double
  - Variables must be declared
  - Variable name is associated to an address in memory
- Arrays – collection of variables
  - Indexing
  - In C array names represent the address
- C Operators: arithmetic, comparison, logic, assignment
- C Control Statements: decision, loop

C Functions

- A C program is a collection of functions
  - The function **main** is the first function executed
  - *type name*(*paramlist*) { *stmts* }
    - *type* defines type of value returned by function
    - *name* is the function name
    - *paramlist* provides a list of arguments to the function
    - *stmts* is a set of statements executed when the function is called
- On variables:
  - What is a local variable?
  - What is a global variable?

```
#include <stdio.h>
int sum(int, int);  /* Prototype */
int gz;

void main(void)
{
    printf("sum of 2+3 = %d\n",
            sum(2,3));
}

int sum(int x, int y)
{
    int z;
    z=x+y;
     gz = z;
    return(z);
}
```

C Function Prototype

- The prototype provides specifications of functions
  - Used by compiler to check functions calls, including use of return value and argument types
  - Defined at start of program file or within headers
  - Example

    int sum(int, int);
  - Note that parameter names are not required.
  - The following is also valid:

    int sum(int a, int b);

```
#include <stdio.h>
int sum(int, int);  /* Prototype */
int gz;

void main(void)
{
    printf("sum of 2+3 = %d\n",
           sum(2,3));
}

int sum(int x, int y)
{
    int z;
    z=x+y;
    gz = z;
    return(z);
}
```

C Pointer Variables

- A C pointer variable (a.k.a. pointer) contains an address

  int c;
  - &c gives the address of variable c
  - “string” actually returns an address to the character array
- A pointer points to a type of value
  - Use * after *type* to declare pointer

    ```
    char *strPtr;
    ```
    - **strPtr** is a variable that can contain an address to a *char* or *char* array
  - Array name gives address of first element

    ```
    float realnums[10], *realPtr;
    ```
    - **realnums** can be used to reference address of first element

    ```
    realPtr = realnums; //points to array
    ```

    is equivalent to

    ```
    realPtr = &realnums[0];
    ```

```
#include <stdio.h>
char str[20];

void main(void)
{
    char *pt;
    strcpy(str,”Hello world\n”);
    pt = str;
    pt = &str[0];
    printf(“%s”,pt);
}
```

C Function Arguments – pass by reference

- For arrays, an address to the array is passed as function argument

```
int arr[10];
…
sumArray(arr);
```

- Consider a function to exchange the values of two integer variables

```
/* what is wrong with this function ? */
exchange(int x, int y)
{ … }
```

C Variables – Structures (records)

- The C structure is a data type used to define complex data structures (e.g. linked list).
  - Definition of a structure has the following format:

```
struct name
{
  Type memberName1;
  Type memberName2;
  Type memberName3;
  .
  .
} <list of variables>;
```

  - The type definition of the structure (without list of varaibles) can be used as a type in variable declarations:

```
struct name variableName;
```

- Arrays of structure: `struct name variableName[dimension];`

C Structure - Example

- Defining structure type

```
struct pers
    {
     char name[30];
     char street[40];
     char city[20];
     char prov[4];
     long employee_num;
    };
```

- Declaring variable

```
struct pers person1;/* memory allocated */
```

- Finding size of memory allocated to structure

```
sizeof(person1); /* gives amount of memory allocated
                    to person1 */
```

- Array of 10 structures: `struct pers persons[10];`

Using Structure Members

- The dot (.) operator is used to access structure members
  - Referencing a structure member has the format

    structureName.memberName
  - Examples

```
person1.employee_num = 654321;

printf("%ld\n",person1.employee_num);
gets(person1.name);
for(i=0 ; person1.street[i];  i++)
   putchar(person1.street[i]);

persons[4].employee_num = 123456;
```

Pointers to structures

- When passing a structure to a function, the complete structure is placed on the stack

```
printRecord(person1);
```

  - This can be time-consuming if the structure is large
  - Does not allow the function to access (i.e. update) the original structure
- Using a pointer to a structure is normally used to have a function manipulate its contents

```
fillRecord(&person1);  // call
```

  - One single address is passed to the function, much more efficient
  - Allows the function to update the addressed structure
  - Replace the dot "." operator with "->" operator when using a pointer to access structure members

```
fillRecord(struct pers *persPtr)
{ …
   persPtr->employee_num = 123456;
```

Defining data types - `typedef`

- Can define new data types using “typedef”
  - **typedef “*type expression*” *newTypeName***
- Examples:

```
typedef int size_t;
typedef struct adr
    {
     char name[30];
     char street[40];
     char city[20];
     char prov[4];
     long employee_num;
    } EmployeeRecord;
EmployeeRecord employees[10];
```

C Standard Functions

- A C development system comes with standard libraries
  - System library provides functions for making UNIX system calls
  - Standard C library provides functions for manipulating strings, I/O (such as printf), math, time, etc.
  - Other libraries come with some applications, such as X Window.
- CodeWarrior does not support all standard C functions.
- For this course, standard functions are not required.

```
#include <stdio.h>
/*------------------------------------------------
   time.h contains prototypes for
   time() and ctime(), as well as the type
   time_t
------------------------------------------------*/
#include <time.h>

void main(void)
{
  time_t tm;

  time(&tm);   /* get system time */
  printf("Current time is %s\n",
            ctime(tm));
}
```

uOttawa

Compiling a C Program - Single Module

- For small programs, all functions are stored in the same file
- A C file represents a C module

```
#include <stdio.h>
void reverse(char *, char *);  /* prototype*/
void main(void)
{
   char str[100];
   reverse(“cat”,str);
   printf(“Reverse of cat is %s\n”,str);
}
void reverse(char *bef, char *aft)
{
    int x,y,len;
    len = strlen(bef);
    for(x=0; y=len-1; y>x ; x++, y- -)
           aft[x] = bef[y];
    aft[len] = ‘\0’;
}
```

C Preprocessor

- Before compilation, the C preprocessor translates its directives
  - #include <file> -
    - inserts file contents
  - #define NAME value
    - defines constant NAME
    - all occurrences of NAME are replaced with value.
  - Other directives are also available

```
#include <stdio.h>
#define NUM1 10
#define STRING1 “Hello world”

void main(void)
{
  int x;
  float y;
  for(x=1 ; x<NUM1 ; x++)
   {
     y = y*x;
     printf(“x: %d x!: %f\n”,x,y);
   }
   printf(“%s\n”,STRING1);
}
```

Compiling a C Program - Multiple Modules

- For larger or modular programs, separate code among many files
  - Certain functions can be made available to many programs
  - Files are compiled separately (e.g. “cc -c prog.c”) - creates object file “prog.o”
  - Object files can then be linked to create final executable (e.g. “cc -o *execFile objectFiles*”
- Use of *static*
  - Local variable becomes permanent
  - Global variable is local to the file module

- For example, create the following files:
  - reverse.h - with prototype of reverse function
  - main.c - main function (contains #include “reverse.h”)
  - reverse.c - reverse function
- To create application, use

```
cc -c main.c
cc -c reverse.c
cc -o rev main.o reverse.o
```

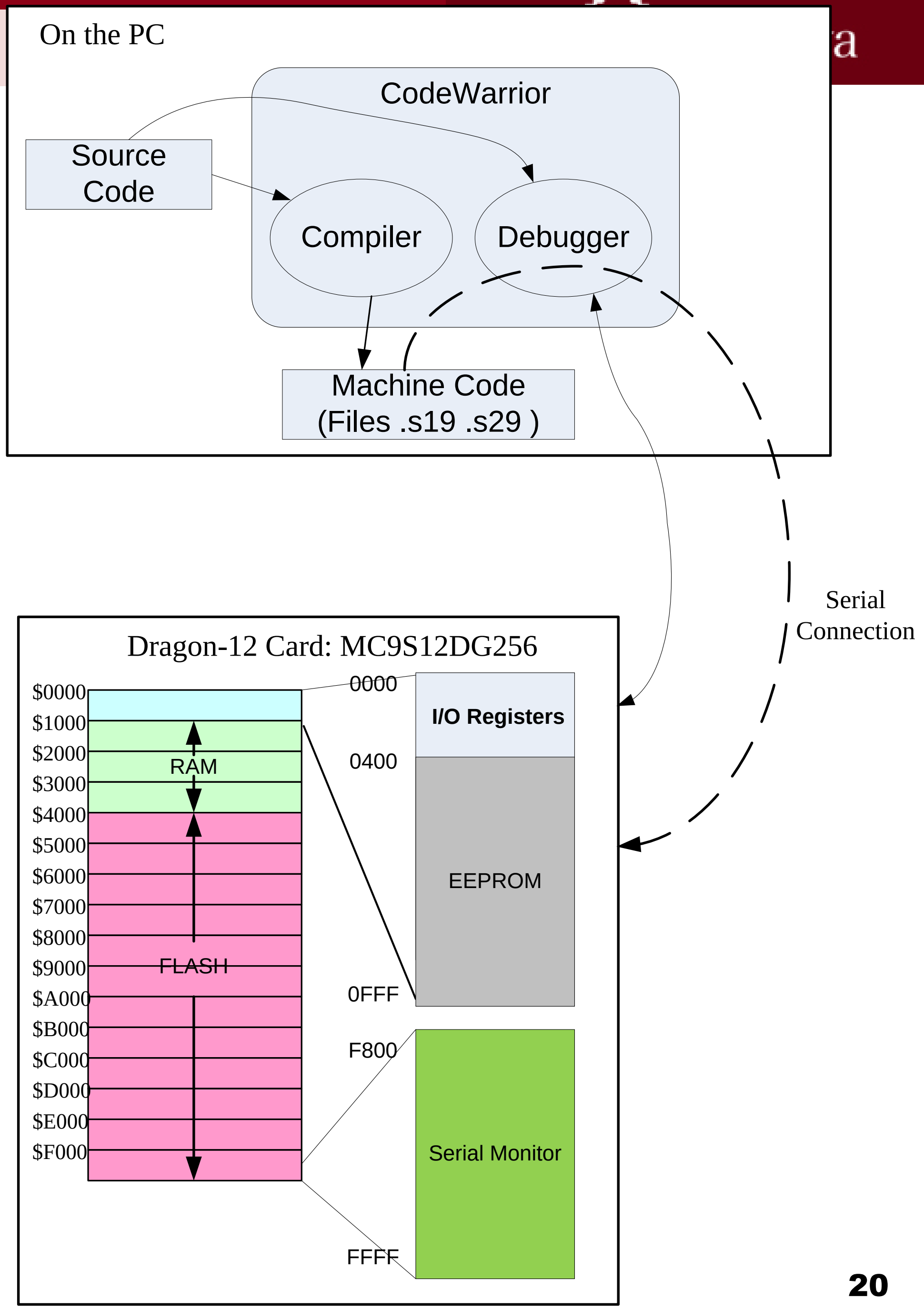
Using C for Developing Embedded Software

- Offers a high level language to develop software for embedded systems
- Must keep in mind a number of additional issues
  - Program is to be loaded into regions of memory: code in ROM, variables in RAM
  - Need to use assembly language subroutines to effectively manipulate hardware.
    - Also use inline assembly instructions
  - Require extensions to C for implementing Interrupt Service Routines (to service hardware interrupts)
- Shall use FreeScale CodeWarrior C Compiler

Architecture of a C Program

- Startup Code
  - Hardware initialization
  - Data initialization
  - Stack pointer
  - Call main
- void main(void)
  - The start of developed software
- Automatic variables
  - Local variables on the **stack**
- Static variables
  - Global variables
  - Static local variables
- Volatile variables
  - Variables that are updated from hardware registers
  - Prevent compiler optimization from removing duplicate instructions

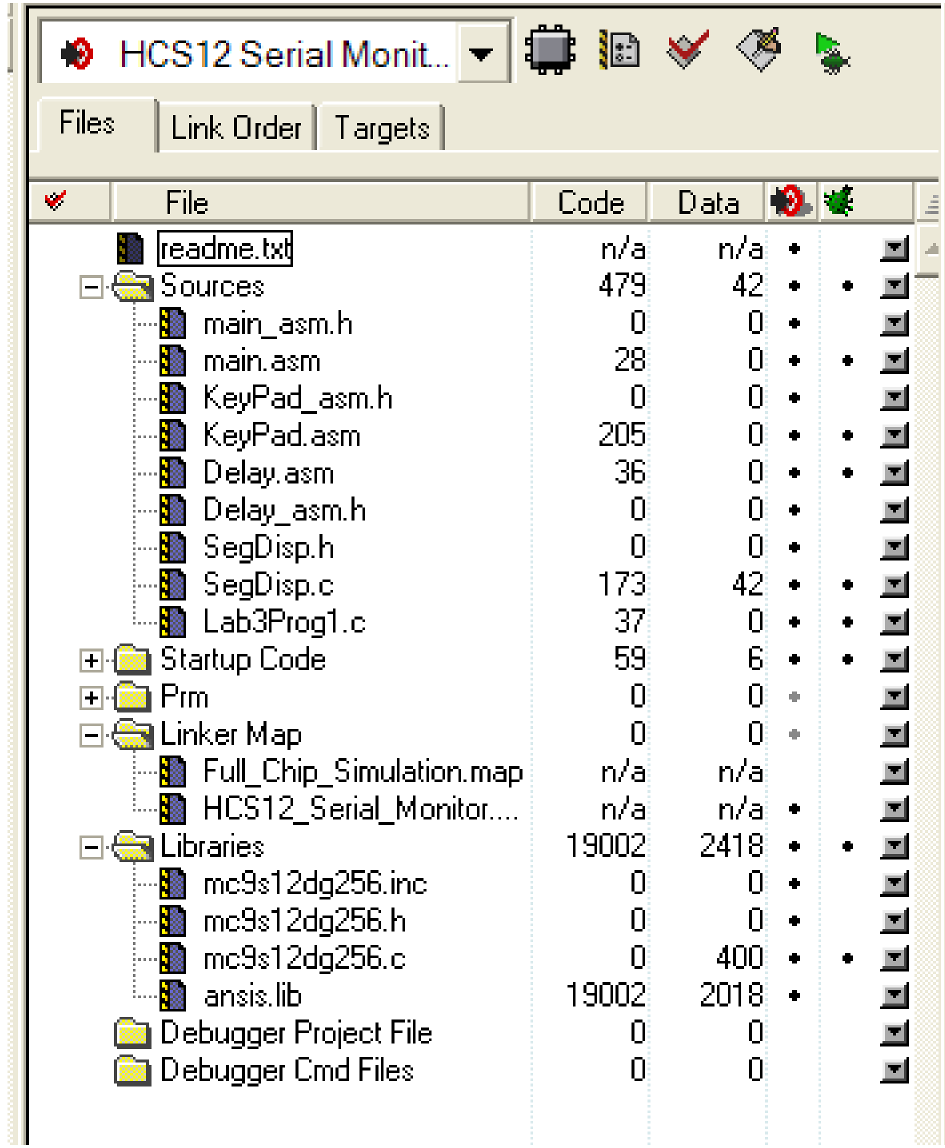
Development Environment



Development Environment

- Use the course sample project
  - Available on Virtual Campus (Labs/lab3 module)
  - Install in

    C:\Program Files\Freescale\(CodeWarrior for S12(X) V5.0\(CodeWarrior_Examples)



Volatile Variables

Source code

```
typedef unsigned char BYTE;
BYTE PORTA;
void main(void)
{
  volatile static BYTE a_val;
           static BYTE b_val;
  /* Read from port A */
  a_Val = PORTA;



  a_Val = PORTA;



  /*  */
  b_val = PORTA;
  b_val = PORTA;


}
```

listing showing compiled code

```
.
.
.
.
11: a_val = PORTA;
0000 f60000    LDAB PORTA
0003 7b0000    STAB a_Val


12: a_val = PORTA;
0006 f60000    LDAB PORTA
0009 7b0000    STAB a_Val

14: b_val = PORTA;
15: b_val = PORTA;
000c f60000    LDAB PORTA
000f 7b0000    STAB b_Val

rts
```

Assembly Language Interface

- Should understand how the compiler works
  - E.g. see previous slides
  - E.g. see next slide on initialization of automatic and static variables
- Often parts of a C program are replaced with assembler instructions/subroutines
  - To reduce size of program
  - Use profiling to identify bottlenecks and replace with faster assembly code
  - Need to understand calling conventions

Overhead for Initializing Variables

```
/**********************/
void main(void) {
  static char a_Val[] =
            {2,90,53, 8 };
  char b_Val[] =
            {0,7,255, 34};



  char c_Val[4], i;
   i=2;
   c_Val[i] = a_Val[i]+
              b_Val[i];


}
```

```
.
.
.
9: char b_Val[] = {0,7,255,34}
 0000 69a8 CLR 8,-SP
 0002 c607 LDAB #7
 0004 6b81 STAB 1,SP
 0006 86ff LDAA #255
 0008 6a82 STAA 2,SP
 000a c622 LDAB #34
 00Cc 6b83. STAB 3,SP

.

12: i = 2;
13: c_Val[i] = a_Val[i]+b_Val[i]
 000e f60000 LDAB a_Val:2
 0011 eb82 ADDB 2,SP
 0013 6b86 STAB 6,SP
.
 0015 1b88 LEAS 8,SP
 0017 3d RTS
```

Calling Convention

- Fixed number of arguments (Pascal convention)
  - Arguments (values) are pushed onto the stack in order (from left to right). The last argument is passed in a register if possible (see below).
- Variable number of arguments (C convention)
  - Arguments (values) are pushed onto the stack in reverse order (from right to left). The last argument is passed in a register if possible (see below).
- When the argument value to be passed via a register is ≤ 4 bytes, it is passed as follows:
  - 1 byte, in B 2 bytes, in D
  - 3 bytes, in X (LW) B(HB) 4 bytes, in D(LW), X(HW)
- When the argument value to be passed in a register is larger than 4 bytes, it is pushed onto the stack
- All values pushed onto the stack are pulled from the stack after the function/subprogram returns

Calling Convention (cont'd)

- The return value are passed back using registers when they are ≤ 4 bytes
  - 1 byte, in B — 2 bytes, in D
  - 3 bytes, in X (LW) B(HB) — 4 bytes, in D(LW), X(HW)
- When return values are larger than 4 bytes, an address is added to the argument list, and the return value copied to that address

Calling Convention - Example

```
/************************
* Test showing the Pascal
* calling convention used by
* CodeWarrior.
*****************/
int functionl(char argl,
              int arg2,
              float arg3);
int function2(char arg1);
void main (void) {
  static char stacy;
  static int sam, mike;
  static float susan;
  mike = functionl(stacy, sam,
                   susan);




  sam = function2(stacy);


}
```

```
13: mike = functionl(stacy…
 0000 f60000 LDAB stacy
 0003 37     PSHB
 0004 fc0000 LDD sam
 0007 3d     PSHD
 0008 fc0000 LDD susan:2
 000b fe0000 LDX susan
 000€ 160000 JSR function1
 0011 1b83 LEAS 3,SP
 0013 7c0000 STD mike
14:  sam = function2(stacy);
 00:6 160000 LDAB stacy
 0019 160000 JSR function2
 0010 7c0000 STD sam
.
 0011 3d RTS
```

C Prototypes for Assembler Routines

```
/*---------------------
File: KeyPad_asm.h
Description:  Header file to use the KeyPad Module
------------------------*/
#ifndef _KEYPAD_ASM_H
#define _KEYPAD_ASM_H

//C Prototypes to assembler subroutines
void initKeyPad(void);
byte pollReadKey(void);
byte readKey(void);

// Some Definitions
#define NOKEY 0  // See KeyPad.asm

#endif /* _KEYPAD_ASM_H */
```

External Symbols in Assembler Source

```
; File: Keypad.asm

; External Symbols Referenced
 XREF delayms

; Define External Symbols
 XDEF initKey, pollReadKey, readKey
```

Inline Assembler Instructions

- It is possible to add assembler instructions within C code.
- Careful (see example),
  - C instructions in between assembler instructions can change register contents
  - Assembler instructions can inconvenience C instructions
- With CodeWarrior, assembler instructions can be inserted in place of C instructions
  - Many forms of inline assembly instructions are available – see example

Careful – Register Changes with Inline Assembler Instructions

```
int a_val;
void main(void){
  int fred;
  fred = 1;




  asm {
    ldd #5555
    std a_Val
  }
  fred = 2;

  for(;;) {

  }
```

```
.
.
.
10: fred = 1;
 0000 c601   LDAB #1
 0002 87     CLRA
 0003 3b     PSHD
.
12:  asm {
 0004 cc15b3 LDD #5555
 0007 7c0000 STD a_Val
15: }
16:  fred = 2;
 000a 58     ASLB
 000b 6c80   STD 0,SP

 000d
```

Examples of Inline Assembler Instructions

```
void main(void){
  static char static_var;
  char auto_var;
  /*In-line Asm. Inst.*/
  asm nop;
  asm ldaa #55;
  /*Block of Asm. Inst.*/
  asm {
    nop  /*Comment*/
    staa static_var;
    staa auto_var;
  }
```

```
/* Mult. Inst. On a line*/
  asm nop; asm nop;
  asm (nop; /*comment*/);

/*Block of Asm Inst*/
  #asm
    nop
    bset static_var,0x01
    bclr auto_var, 0x02
  #endasm
```

Accessing I/O Registers

- As we shall see, manipulating hardware registers requires accessing them at specific locations and requires bit manipulation
- Addressing Registers
  - CodeWarrior provides the global address modifier @, to tie variable names to specific addresses
  - Otherwise pointer manipulation is required
- Manipulating Bits
  - C Structures can be defined to assign member names to specific bits
  - CodeWarrior documentation does NOT recommend this approach, but use bitwise operators & and |
  - CodeWarrior recognizes binary values, e.g. 0b01001000

Options for Associating Variables to Hardware Registers

```
/**************************************************/
/*  Define a PORT type = unsigned char  */
typedef unsigned char PORT;
/**************************************************/
/*Declare PORTA to be volatile unsigned char at addr 0x0000 */
volatile PORT PORTA @0x0000;
/*************************************************/
/*Declare PORTB to be the contents of a memory location
  pointed to by a volatile unsigned char pointer 0x0001 */
#define PORTB (*(volatile PORT *) 0x0001)
/**************************************************/
void main(void) {
   unsigned char p_a_val;
   PORTA = 6;         /* Write to PORT A */
   p_a_val = PORTA;   /* Read from PORT A */
   PORTB = 26;        /* Write to PORT B */
}
```

Structures with Bitfields

```
/* Define a bitfield type as unsigned int */
typedef unsigned int BITFIELD;
/*********************/
/* Declare an eight-bit field for the PORTB which is
   volatile at address 0x0001 */
struct {
  BITFIELD BIT0 : 1;
  BITFIELD BIT1 : 1;
   .
   .
  BITFIELD BIT7 : 1;
} volatile PORTB @0x0001;
/*****************************************************/
void main(void) {
  /* translates to bit-set and bit-clr asm instr */
  PORTB.BIT0 = 1;
  PORTB.BIT0 = 0;
  PORTB.BIT1 = 1;
  PORTB.BIT1 = 0;
}
```

Not Recommended

Combining Register and Bit Addressing

```
/***** Types ******/
typedef unsigned char PORT;
typedef unsigned int BITFIELD;
/**** Union that gives two perspectives ****/
typedef union {
   PORT PortByte;
   struct {
      BITFIELD BIT0 : 1;
      BITFIELD BIT1 : 1;
      .
      .
      BITFIELD BIT7 : 1;
  } PortBits;
}IOPort;
/** Data and Data direction registers for Port T **/
volatile IOPort PORTT @0x0240
volatile IOPort DDRT @0x0242
/*****/
void main(void) {
  DDRT.PortByte = 0x0F;  /* Bits 3 - 0 , output */
  PORTT.PortBits.Bit3 = 1;  /* make Bit 3 = 1 */
}
```

Interrupts – an Introduction

- An interrupt is a hardware generated signal that request some service
  - Efficient way of dealing with hardware
  - Basis for operating systems
- The interrupt service routine (ISR)
  - A routine (i.e. function/sub-routine) executed to service an interrupt
  - Terminated with RTI, and not RTS
  - The CPU maintains a vector of addresses to these routines to associate them to the specific interrupts

Servicing an Interrupt

- CodeWarrior provides the « interrupt » qualifier to declare functions that service interrupts
  - Include the « interrupt » keyword in the function definition.
  - Can also add the interrupt vector number to associate the function to a specific interrupt
  - More on this in module 8

Example of an ISR C Function in CodeWarrior

```
/* LED connected to bit 0 of Port A */
/* Interrupts received on IRQ pin */
#define BIT0 0b00000001
void main(void) {
  DDRA = DDRA | BIT0;  /*Output*/
  PORTA = PORTA & ~BIT0; /*Turn off LED*/
  INCR = INCR | BIT0; /*Activate IRQ */
  asm cli; /* allow interrupts*/
  for(;;) ;  /* infinite loop */
}
/**********ISR**********/
/* Vector # 6 can be replaced with VectorNumber_Virq
   see mc9s12dg256.h */
void interrupt 6 irq_handler(void) {
  if(PORTA & 1)  /* is lit? */
     PORTA |= BIT0;  /* turn off */
  else
     PORTA &= ~BIT0;  /* turn on */
}
```