Chris Crary

More On C Programming

EEL4744C – Microprocessor Applications

Special thanks to:
Daniel Gonzalez
Raz Aloni

1

Reserved keywords

The following list shows the reserved words in C. These reserved words may not be used as constants or variables or any other identifier.

auto	else	long	switch
break	enum	register	typedef
case	extern	return	union
char	float	short	unsigned
const	for	signed	void
continue	goto	sizeof	volatile
default	if	static	while
do	int	struct	_Packed
double			

2

The volatile keyword

There's really only one reason to use the volatile keyword: when you interface with hardware.

The volatile keyword "alerts" the compiler **not** to optimize anything that is considered volatile (could change after being defined).

In general, when using the following, the volatile keyword is required:

1. Memory-mapped peripheral registers
2. Global variables modified by an interrupt service routine
3. Global variables within a multithreaded application (not necessary in 4744, as of now, but used in μP2!)
4. Inline assembly (not covered in this lecture)

3

3

The volatile keyword, cont.

Peripheral register example without `volatile` keyword:

```
/* Create a pointer, p_reg, to memory location 0x4744. */
/* For this example, assume that a peripheral register
 * is memory-mapped to this address. */
uint8_t *p_reg = (uint8_t *) 0x4744;

/* Wait for data within 0x4744 to be equal to 0x37. */
/* (REMEMBER that a memory-mapped component is, in general,
 * subject to change at any point in time!) */
while(*p_reg != 0x37);

/* With compiler optimization off (and likely even with),
 * this while loop will almost assuredly either run only
 * once or infinitely (i.e., not our desired functionality),
 * because the compiler mistakenly interprets that the
 * data stored at 0x4744 should never change, even though it could! */
```

4

4

The volatile keyword, cont.

Peripheral register example without `volatile` keyword:

Likely generated assembly code (WRONG AND UNINTENDED):

```
LDS R24,0x4744 ; Load direct from 0x4744
CPI R24,0x37   ; Compare value at 0x4744 to 0x37

BRNE PC-0x01   ; If not equal, erroneously continue
               ; to compare to the same value forever.
               ; NOTE: PC-0x01 represents the address
               ; of the above CPI instruction.
```

5

5

The volatile keyword, cont.

Peripheral register example with `volatile` keyword:

```
/* Create a pointer, p_reg, to memory location 0x4744. */
/* For this example, assume that a peripheral register
 * is memory-mapped to this address. */
uint8_t volatile *p_reg = (uint8_t volatile *) 0x4744;
/* Below might work too (with a warning), but above is better, since it will
   prevent you from accidentally assigning an integer as a pointer. */
uint8_t volatile *p_reg = 0x4744;

/* Wait for data within 0x4744 to be equal to 0x37. */
/* (REMEMBER that a memory-mapped component is, in general,
 * subject to change at any point in time!) */
while (*p_reg != 0x37);

/* Adding the `volatile` keyword as done above will hint to the compiler that
 * the data located within the address pointed to by `p_reg`, i.e., address
 * 0x4744, could change at any time, meaning that whenever it is necessary to
 * read from the value stored within `p_reg`, e.g., 'while (*p_reg != 0x37)',
 * data should be reloaded from the relevant data memory location (0x4744),
 * and no previous version of the data at that location should be utilized
 * (unlike as shown within the previous assembly code). */
```

6

6

The volatile keyword, cont.

Peripheral register example with `volatile` keyword:

Likely generated assembly code (CORRECT AND INTENDED):

```
LDS R24,0x4744  ; Load direct from 0x4744
CPI R24,0x37    ; Compare value at 0x4744 to 0x37

BRNE PC-0x02    ; If not equal, reload potentially
                ; new data from address 0x4744 before
                ; re-comparing to 0x37.
                ; NOTE: PC-0x02 represents the address
                ; of the relevant LDS instruction.
```

7

7

The volatile keyword, cont.

Interrupt service routine example without `volatile` keyword:

```
/* assume all other necessary items are included */
#define FALSE  0
#define TRUE   1

uint8_t course_grade; /* global variable for 4744 grade */
uint8_t impressedSchwartz = FALSE;

void main(void)
{
  /* assume that miscellaneous things are here */
  while(!impressedSchwartz)
  {
    /* not good enough yet */
  }
  /* assume all other miscellaneous things are here */
}
```

8

8

The volatile keyword, cont.

Interrupt service routine example without `volatile` keyword (cont.):

```
ISR(schwartz_vect)
{
  /* ... */
  if (course_grade >= 90)
  {
    impressedSchwartz = TRUE;
  }
  /* ... */
}
```

(**NOTE:** Fix any infinite loop issues by making *course_grade* and *impressedSchwartz* volatile.)

9

9

The volatile keyword, cont.

Some compilers allow you to implicitly declare all variables as volatile. Resist this temptation, since it is essentially a substitute for thought. It also leads to potentially less efficient code.

10

10

Other useful keywords

typedef:
typedef allows you to give a type, a new name

```
11  typedef unsigned char BYTE    // give unsigned char the alias name "BYTE"
12
13  BYTE b1, b2   // declare BYTEs (unsigned chars)
```

11

11

Data types

There are many data types defined in C. Some of the types can be classified as follows:

Basic Types:
These are arithmetic types and are further classified into: (a) integer types and (b) floating point types

Enumerated Types:
These are also arithmetic types and they are used to define variables that can only assign certain discrete integer values throughout a program.

Derived types
These include (a) Pointer types, (b) Array types, (c) Structure types, (d) Function types, etc.

12

12

Integer data types

The C99 Standard for integers allows programmers to write more portable code by providing integer data types that specify the length and range of each data type

To use the C99 standard, you must include the header file *stdint.h.*
Including <avr/io.h> is also sufficient when using the XMEGA.

C99 Integer Data Type	MSP432 C89 Equivalent	Range
`int8_t`	`signed char`	-128 to 127
`uint8_t`	`unsigned char`	0 to 255
`int16_t`	`short`	-32,768 to 32,767
`uint16_t`	`unsigned short`	0 to 65,535
`int32_t`	`int, long`	-2,147,483,648 to 2,147,483,647
`uint32_t`	`unsigned long, unsigned int`	0 to 4,294,967,295
`int64_t`	`long long`	-2^{63} to $2^{63}-1$
`uint64_t`	`unsigned long long`	0 to $2^{64}-1$

13

13

Floating point data types

Floating Point Numbers allow a processor to represent real numbers (with a certain precision).

Warning: If your processor does not have a floating point unit (FPU), it is best practice to avoid using floats. This is the case with our XMEGA processor!

Floating Point Type	Range
`float`	$\pm(1.17549 \times 10^{-38}$ to $3.40282 \times 10^{38})$
`double`	$\pm(2.22507 \times 10^{-308}$ to $1.79769 \times 10^{308})$

14

14

Enumerated types

An enumeration (enum) consists of a set of named integer constants. A variable with enumeration type stores one of the values of the enumeration set defined by that type. Enumerations provide an alternative to the #define preprocessor directive with the advantages that the values can be generated for you and obey normal scoping rules.

There are multiple ways to declare/declare enumerations in C. The following is the recommended syntax:

typedef enum { constant1, constant2, ...} enum_name;

After this statement, enum_name can be used as a data type.

Example: Defining an enumerated type

```
82  // define bool to be a new enumerated data type
83  typedef enum { FALSE, TRUE } bool;
84
85  bool foo = TRUE;    // create/initialize a boolean variable to TRUE
```

15

15

Structs

A **struct**(ure) is a compound datatype that holds a grouped list of variables in one block of memory under one name. Structs are useful for representing items defined by multiple properties.

Structs can be made up of any datatype in C, primitive, compound, or user-created.

To create a struct, use the struct keyword.

```
struct date       /* data type for date */
{
   uint16_t year;  /* integer to represent year */
   uint8_t month;  /* integer to represent month */
   uint8_t day;    /* integer to represent day */
};
```

16

16

Structs

To access properties of a struct directly, use the dot '.' operator. To access the properties via a pointer to a struct, you can use the '->' operator.

```
/* instantiate a date struct */
struct date date;

/* set the day of the Date to 3 */
date.day = 3;

/* read the day into x */
uint8_t x = date.day;
```

17

17

Preprocessors

In simple terms, a C Preprocessor is just a text substitution tool and it instructs the compiler to do required pre-processing before the actual compilation.

You use preprocessor directives when you need to do something outside of the scope of the actual application.

All preprocessor commands begin with a hash symbol (#).

18

18

Macros

A **macro** (short for "macroinstruction") in computer science is a rule or pattern that specifies how a certain input sequence (often a sequence of characters) should be mapped to a replacement output sequence (also often a sequence of characters) according to a defined procedure.

The preprocessor directive #define gives symbolic names for anything by performing a text replacement in the pre-processor (before compile time).

There are two common types of macros: object-like and function-like macros.

19

19

Macros, cont.

Object-like macros serve to typically define some constant, where function-like macros can act like small functions with parameters.

Object–like macro examples:

```
#define PI 3.14159
#define LED_PORT PORTC
```

Function-like macro example:

```
#define BAD_DELAY(X) for(int i = 0; i < X; i++)
#define BIT(x) 1<<x
```

20

20

Libraries

Libraries are especially useful in writing modular code (the process of subdividing a computer program into separate sub-programs, with the intention of reusing the sub-programs).

- To create a library, you will need to create both a **source file and a header file**.
- **Header Files (.h)** will consist primarily of function prototypes/descriptions and defines.
- **Source Files (.c)** will consist of the actual implementation of every function defined in its corresponding header file.

In your main program, all that is necessary to include your header/source files is to type `#include "FILENAME.h"` , and make sure both your source and header files are in your dependencies (explained later).

21

21

Libraries, cont.

An example of helpful header files to create for this course:

- **Initialization Headers (INITs)**
  - Any initialization of a particular system or module can/should be put into a header file.
    - Examples: TC, EBI, EBI_DRIVER, DAC, ADC, DMA, etc.
  - Remember to confirm that all your configurations are correct! This is where group configurations and bit masks are very useful.
    - See the include file for group configuration and bitmask definitions.

By creating header files for each module, our main source file will minimize substantially.

- See ebi.c and ebi_driver.h on our website

22

22

What belongs in a header file

1. **DO** include source code documentation (the purpose of the various functions, parameters, and return values)!
2. **DO** include header guards (to prevent multiple includes across multiple source files).
3. **DO** include all of the function prototypes for the public interface of the module it describes.
4. **Do NOT** include any executable lines of code in a header file, including variable declarations.
   - Exceptions made for inline functions (not necessary for this course).

23

23

Example header file

```
/*
 * USART.h
 */

// header guard
#ifndef USART_H
#define USART_H

/*
 * Function to output a string of characters
 * Param pointer to an array of characters
 */
void out_string(char*);
/*
 * Function to output a single character
 */
void out_char(char);
/*
 * Function to receive a single char
 */
char in_char(void);

#endif    // end of header guard
```

You may also choose to include any relevant defines you want to use in your source file or main source file.

24

24

Example source file

```
/*
 * USART.c
 */

// include header file
#include "USART.h"

/*
 * Function to output a string of characters
 * Param pointer to an array of characters
 */
void out_string(char* str)
{
  // implementation goes here
}
/*
 * Function to outputs a single character
 */
void out_char(char c)
{
  // implementation goes here
}
/*
 * Function to receive a single char
 */
char in_char(void)
{
  // implementation goes here
}
```

25

25

Creating/including a library

- Adding a header/source file:
  - File > new > file > include file (.h) or source file (.c)
  - Save this file in a separate folder (maybe a folder called 4744_Libraries)
- Including a header/source file:
  - Right click on your project's name in the solution explorer
  - Add > Existing Item > browse to your .h and .c file you saved > add as link*
  - Remember to type `#include "FILENAME.h"` at the top of your main source file as well as the corresponding source file

26

26

A few good coding practices

1. Write meaningful comments, in any appropriately available location.
2. Write modular code.
3. Write concise, not overly-complicated code.

27