Microcomputers I – CE 320

Mohammad Ghamari, Ph.D.
Electrical and Computer Engineering
Kettering University

Announcement

Lecture 20: Subroutines in C

Today’s Topics

By the end of this class you should be able to:

- Use multiple files to write a C program
- Share variables and labels between files
- Explain how to pass parameters to C functions
- Explain how to return values from C functions
- Call a subroutine written in assembly from a C program

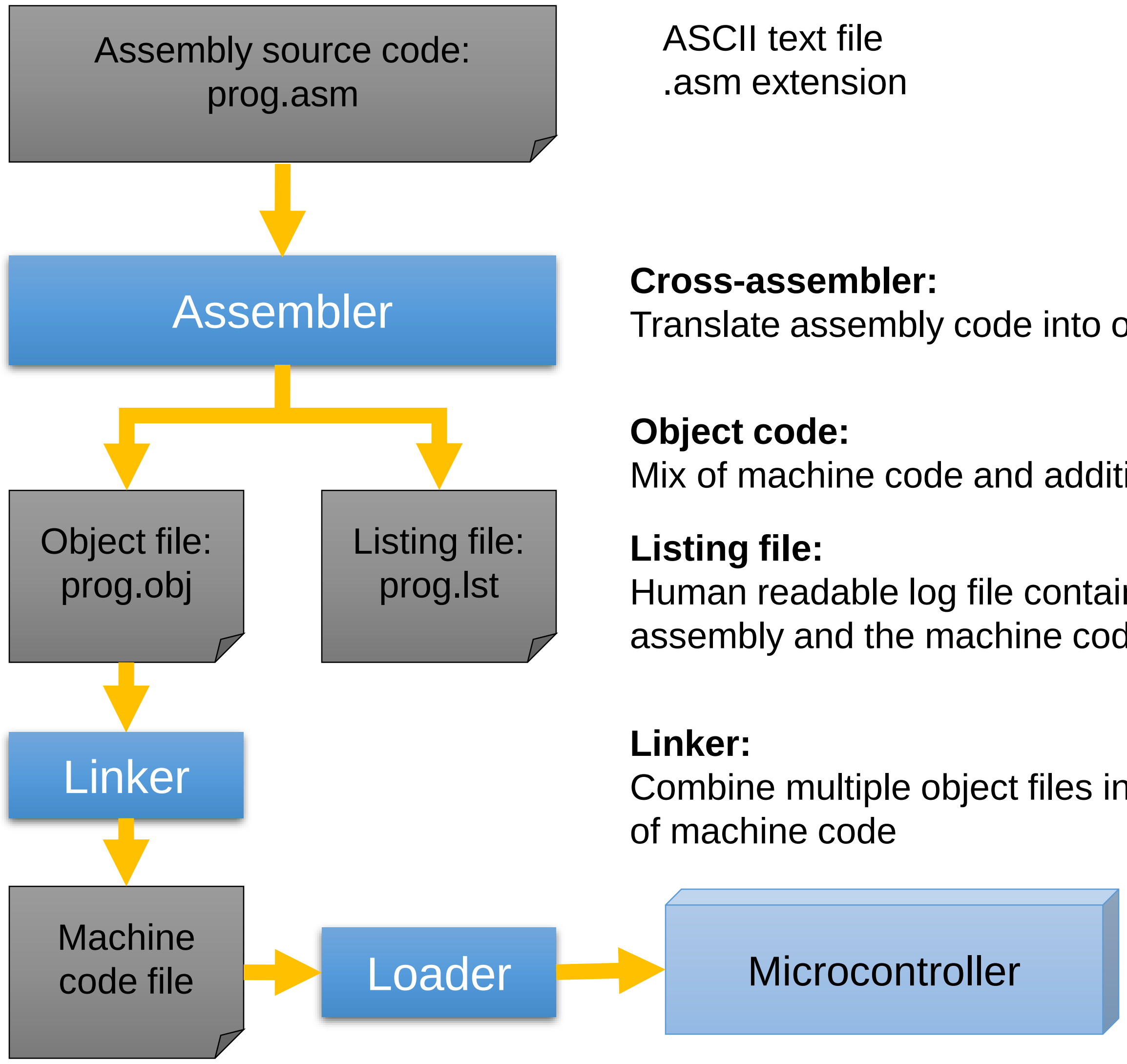
Introduction

- Up to this point in the course, we mostly have been working with "*Absolute Assembly*" programs where all code is put into one file, and it is assembled at absolute addresses indicated with "ORG" statements.

- We shall now learn how to put assembly code into two or more "*Relocatable Assembly*" files which are separately assembled into object (.obj) files and then linked together into an executable file under the control of a "Linker Parameter" (.PRM) file, which tells the linker where to locate the program, data, and stack segments of each relocatable file in the memory space of the microcontroller.

Assembly Process

General case



ASCII text file
.asm extension

Cross-assembler:
Translate assembly code into object code.

Object code:
Mix of machine code and additional information

Listing file:
Human readable log file containing the original assembly and the machine code.

Linker:
Combine multiple object files into a single piece of machine code

Introduction…

- The need for separately assembly-language modules, which are separately assembled into object files, and then finally linked together into a single executable file, may not be very apparent in this course, where our applications are relatively short and simple.

- However, in large software development efforts, where several programmers are involved,
    - It is often more convenient for each programmer to develop and maintain their own separate program file which is eventually linked with other programmer's program files into an executable program.

- This approach also shortens compile time, since only the faulty module needs to be recompiled before the files are re-linked.

Using Multiple Files

- To better divide a programming project between programmers, a program is written using several files.
    - One file contains the **main program**, and it calls **subroutines** which are contained in the other files.
- During compilation, each separate file is compiled into a separate object file.
    - Recall that an object file contains assembly code and additional information. Among this information are blanks for specific addresses.
    - Consider that assembly (machine) code calls a subroutine by jumping to a specific memory address, not an alphanumeric label.
    - Thus, the specific address must be available in the file with the calling program, not just the assigned label.
    - So, a given project file needs to share the addresses assigned to its labels with the other files for them to complete their call.
    - This *sharing and replacing is the* ***link process***.

Using Multiple Files …

- A file shouldn’t necessarily share all of its labels with all other files. This would require every label in every file to be globally unique.
- For example, the variable “loop” could only be used by one programmer in one file. Everyone else would be prohibited from using it.
- To avoid this, ***files should be selective in which variable are shared.***

Sharing Labels Between Files in Assembly

- To share a label, two things must happen.
  - The file that declares the label must state to the assembler that it will be ***global***, and only one file can do so for a unique label.
  - Any file that wants to use the predefined global label must ***explicitly ask for it***. This process requires three steps.

1. The file that creates the label declares it normally, i.e. by labeling a line in a subroutine, a DS.B statement, etc.
2. The file that creates the label **makes it global** with the **XDEF** assembler directive.
3. All other files that wish to use the label's global value, link to it with an **XREF** directive.

File1.asm

```
            XDEF        SUB1,TEMP
            ORG         $1000
TEMP        DS.B        4
            ORG         $C800
SUB1        CLRA
            RTS
```

File 1 creates the labels and declare them normally by labeling lines in the subroutine.

File 1 creates the labels. So makes them **global** with the **XDEF** assembler directive.

File2.asm

```
            XREF        SUB1
            ORG         $2000
Main        LDS         #$3600
            JSR         SUB1        ; jumps to $C800
            SWI
```

File 2 wishes to use the label's global value, so it links to it with an **XREF** directive.

File3.asm

```
            ORG         $2900
SUB1        CLRB
            RTS
            ORG         $2100
            LDS         #$3600
            JSR         SUB1        ; Jumps to $2900
            SWI
```

Functions in C

- A C function may return one value, or “void” if there is no return value.

- The function must be declared in the file that uses it BEFORE any code that calls it. (unlike assembly)

- The *function declaration* shows the **return type** and **the types of the parameters** and **order of the parameters**. (assembly just had a label)

- The function code looks like the declaration, but parameter labels are included.

- The function itself does not need to be in the same file as the caller.
  - If in the same file, it is typically written after the main program.

- The “**return**” instruction passes the number specified as the function’s output and returns to the calling program.

- If both the <u>calling program</u> and <u>function</u> are written in C and *compiled by the same compiler*… <u>we don’t really care how or where the inputs and output are passed.</u>

- *Generally, C compilers do not preserve register values.*

Functions in C …

Example:

```
signed int answer;
signed int myequation(int, int, int);
```

- C function is declared before any code calls it.
- Declaration shows:
    - Return type
    - Types of parameters
    - Order of parameters

```
void main(void)
{
      signed int my_num = -10;
      answer = myequation(5,  my_num, 0x0007);
}
```

- Main program that calls the function.

```
signed int myequation(int num1, int num2, int num3);
{
       return num1*num2+num3;
}
```

- **C function:** It returns a value.
- The function code looks like the declaration, but parameter labels are included.
- Since the function is written in the same file as the caller, it is located after the main program.

Using Assembly Subroutines in C

- Assembly subroutines are often written in separate files so that inline assembly is not used.
- It is also considered good programming practice not to mix a *high-level language* and *assembly* in the same file.
- The basic steps are below, and note that many of the steps deal with the caller and callee being in separate files.

1. Write the subroutine in an assembly file, such as *subfile.asm*.

   In the assembly, use an **XDEF** directive for the **name of the subroutine**.

2. Write the calling C program in a C file, such as *main.c.*

   In the C file, use a one line **function declaration** with the same name as the assembly subroutine.

Example

- Some useful assembly subroutines would be conversions.
- Convert a signed char to a signed long.

```
; assembly file
            XDEF        charToLong
            ORG         $2800
char_s      SEX         B,X
LDD         #0
CPX         #0
BPL         endsub
LDD         #$FFFF
endsub      RTS                         ; D now has signed int
```

- Uses an **XDEF** directive for the **name of the subroutine.**

```
// c file

long charToLong(signed char);
signed char tinynum;
long longnum;

....
shortnum = charToLong(tinynum);
```

- Uses a one line **function declaration** with the **same name as the assembly subroutine**.

Parameter Passing Order

- Consider the following C subroutine:

```
signed int myequation(int num1, int num2, int num3);
{
        return num1*num2+num3;
}
```

- In many programming languages, subroutine parameters are *pushed on the stack* by the calling program, and the calling program is also responsible for pulling them off afterwards.

- C Convention: Right to left as they appear in the function declaration.

- Pascal Convention: Left to right as they appear in the function declaration.

Parameter Passing Order …

- The Freescale’s CodeWarrior uses the **Pascal Convention** for subroutines with a fixed number of inputs (which is the only case we’ll discuss).

- Also, the **last parameter** (i.e. the rightmost) is **passed by register** if the parameter is four bytes or less.

- The **result**, if there is one, is **passed back in register**.

- The list below shows which registers are used:

  One Byte: B
  Two Bytes: D
  Three Bytes: B:X
  Four Bytes: D:X

Homework Example

- Some C subroutines use the returned value to indicate errors.

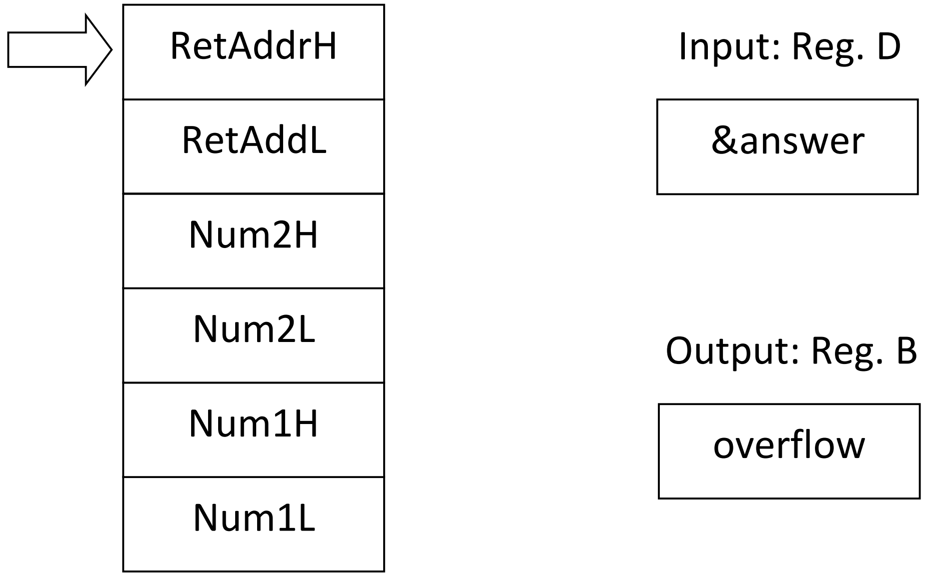
- Write a subroutine that adds two unsigned integers, generates the answer, and returns 0 for no overflow and 1 for overflow.

- Note, since the subroutine’s one return value is used to indicate success or failure, we’ll need to return the answer as we did in assembly: pass it by reference as an output.

Homework Example …

// C declarations

char add_uint(unsigned int, unsigned int, unsigned int*); // passing Num1,
// Num2, pointer to Answer



```
; assembly file
            XDEF      add_uint

add_uint:   TFR       D,X        ; pointer to sum passed in D
            LDY       #0         ; assume no overflow
            LDD       4,SP       ; load first number
            ADDD      2,SP
            BCC       skip
            LDY       #1
skip        STD       0,X
            TFR       Y,B
            RTS
```

```
// c file that calls subroutine
unsigned int n1 = 5;
unsigned int n2 = 7;
unsigned int answer;
…
// exits program if overflow is detected since in C
// 0 means false, anything else means true
if( add_uint(n1,  n2, &answer)) return 1;
```