18ES611 Embedded System Programming

Sarath tv

Embedded C

- C -general purpose programming language, for system programming.
- Embedded c is an extension to c programming language
  - Support for developing efficient programs for embedded devices.
- Set of language extensions for the c programming language.
- The c standards committee.
- To address the commonality issues that exist between c extensions for different embedded systems.
- Embedded C use most of the syntax and semantics of standard C

- differences do exist,
- C is generally used for desktop computers, while embedded C is for microcontroller based applications.
- C can use the resources of a desktop PC. While, embedded C has to use with the limited resources
- Embedded C includes extra features over C, such as multiple memory areas, and I/O register mapping.
- Compilers for C (ANSI C) typically generate OS dependent executables.
- Embedded C requires compilers to create files to be downloaded to the microcontrollers/microprocessors where it needs to run.

C +Assembly

- more comfortable writing in C,
- C is a mid-level language (in comparison to Assembly, which is a low-level language), and
- spares the programmers some of the details of the actual implementation.
- some low-level tasks that either can be better implemented in assembly, or can *only* be implemented in assembly language.
- the programmer to look at the assembly output of the C compiler, and hand-edit, or *hand optimize* the assembly code in ways that the compiler cannot.
- Assembly is also useful for time-critical or real-time processes.

Inline Assembly

```
#include<stdio.h>

void main() {

  int a = 3, b = 3, c;

  asm {
    mov ax,a
    mov bx,b
    add ax,bx
    mov c,ax
  }

  printf("%d", c);
}
```

Inline assembly is invoked in different compilers in different ways

Linked Assembly

- When an assembly source file is assembled by an assembler, and a C source file is compiled by a C compiler, those two **object files** can be linked together by a **linker** to form the final executable.
- The beauty of this approach is that the assembly files can be written using any syntax and assembler that the programmer is comfortable with.
- Also, if a change needs to be made in the assembly code, all of that code exists in a separate file, that the programmer can easily access.
- The only disadvantages of mixing assembly and C in this way are that a)both the assembler and the compiler need to be run, and b) those files need to be manually linked together by the programmer.

- **Advantages of inline assembly:**
- Short assembly routines can be embedded directly in C function in a C code file.
- The mixed-language file then can be completely compiled with a single command to the C compiler
- This method is fast and easy.
- **Advantages of linked assembly:**
- If a new microprocessor is selected, all the assembly commands are isolated in a ".asm" file. The programmer can update just that one file—there is no need to change any of the ".c" files.

Reentrancy

- Every embedded system uses interrupts; many support multitasking or multithreaded operations.
- The program's control flow to change contexts at just about any time.
- When that interrupt comes, the current operation gets put on hold and another function or task starts running.
- *What happens if functions and tasks share variables*?
- Functions that can be called by more than one task and that will always work correctly even if an rtos switches from one task to another in the middle of executing the function.

Example

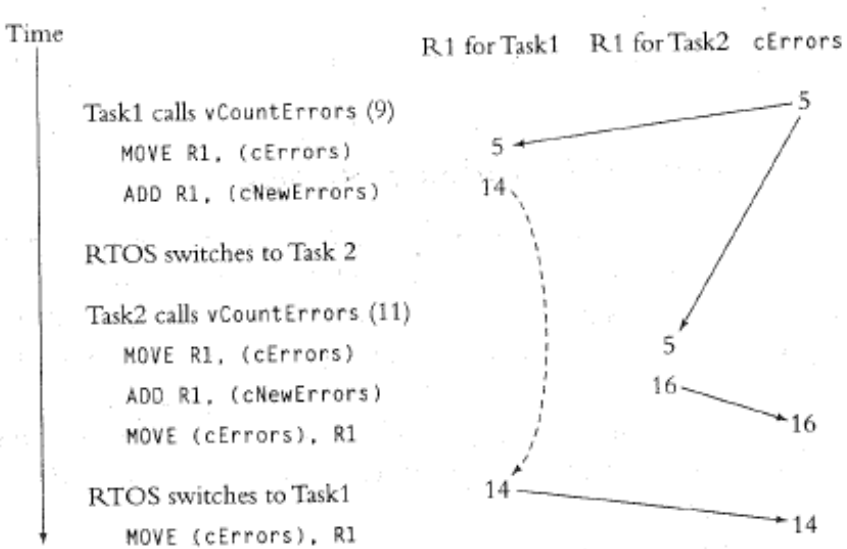
```
void Task1 (void)
{
   .
   .
   .
   vCountErrors (9);
   .
   .
   .
}

void Task2 (void)
{
   .
   .
   .
   vCountErrors (11);
   .
   .
   .
}

static int cErrors;

void vCountErrors (int cNewErrors)
{
   cErrors += cNewErrors;
}
```

```
; Assembly code for vCountErrors
:
; void vCountErrors (int cNewErrors)
:{
;      cErrors += cNewErrors;
       MOVE R1, (cErrors)
       ADD R1, (cNewErrors)
       Move (cErrors), R1
       RETURN
:}
```


Time
R1 for Task1
R1 for Task2
cErrors
Task1 calls vCountErrors (9)
MOVE R1, (cErrors)
ADD R1, (cNewErrors)
RTOS switches to Task 2
Task2 calls vCountErrors (11)
MOVE R1, (cErrors)
ADD R1, (cNewErrors)
MOVE (cErrors), R1
RTOS switches to Task1
MOVE (cErrors), R1
5
5
14
5
16
16
14
14

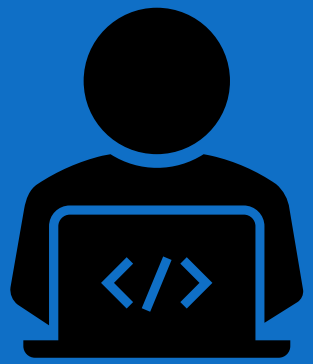
Another Example

```
BOOL fError;   /* Someone else sets this */

void display (int j)
{
   if (!fError)
   {
      printf ("\nValue: %d", j);
      j = 0;
      fError = TRUE;
   }
   else
   {
      printf ("\nCould not display value");
      fError = FALSE;
   }
}
```

Rules to decide if function is reentrant

- May not use variables in a non atomic way unless they are stored on the stack of the task.
- May not call any other functions that are not themselves reentrant.
- May not use the hardware in a non atomic way.



THANK YOU!!!!!