

**EECE 690/890**  
**Digital Radio Hardware Design**

**Team 3**  
**Assignment 4**

**Due: Monday 11/23/98**

## Introduction

This is the fourth in a series of assignments designed to guide you through the tasks needed to complete the software development. In this assignment, you will work on documenting the code you have written so far, and preparing it for integration with that of your teammates.

In the next assignment following this one, you will integrate your code into a single program and exercise that program on the test bench and on the actual hardware built.

## Tasks

All team members should perform the following tasks.

### Preparation for Integrating Code

- ◆ Get copies of the code written by each of your teammates and study their code and yours to guarantee that:
  - Any routines you assumed were written by others are indeed done.
  - The function behavior, arguments, and return values are as you expected.
  - The code will integrate properly with yours (e.g. Is bank switching compatible with your implementation? Are port and variable names compatible?)
- ◆ Write down any problems you anticipate and then hold a meeting with your team mates.
- ◆ Correct any incompatibilities.
  - If there are multiple versions of a routine, decide which will be used in the final code and be sure it will work for all team members.

- If a routine is not done, determine who should develop it and work with them to see that it is coded.
- Resolve any “name-space” conflicts. I.e. guarantee that variables intended to be accessed by multiple team members use the same name in all code and that variables and functions not intended to be shared are not used in other member’s code.
- ◆ Compile your modified programs to be sure that they are ready to be integrated in the next assignment.

## Documenting Code

The goal of the documentation task is to improve the readability and maintainability of your code. Although it is sometimes more efficient to write code without much documentation initially, it is important to add documentation soon after the code is done if there is any chance that you or someone else will ever look at it in the future.

In this assignment, you should put yourself in the place of someone who was not involved in writing the code, but has been given the task of reading through it, understanding it, and modifying it to add new features. Examine your code from this perspective (and imagine that this person is grading you based on the ease with which they can assimilate your code!). Then, work on refining your code structure, adding comments, and providing top-level documentation including a “call-tree” and pseudo-code (or flowchart) descriptions of your routines. Some directions for performing these tasks are provided below. Others have been provided in previous graded project assignments that have been returned to you. If you have written your code well and documented it well in previous assignments, this task should go fairly smoothly and quickly.

- ◆ Refining your code.
  - Look for places where the same set of instructions is repeated multiple places, and if appropriate, write a function (subroutine), to do that action.
  - Make sure your function and variable names are descriptive of what they are actually used for.
  - Be sure you used modern programming constructions. For example, use “for loops”, “while loops”, and “if-then-else” constructs to make your code compact and readable. These are easily emulated in assembly code using the PIC17C44’s ‘btfss’, ‘btfsc’, and ‘decfsz’ instructions.
  - Be sure “side-effects” and “interactions” of your routines are minimized, and/or well documented. For example, does your routine assume the bank register is set to a certain value? If so, how do you guarantee that it is? You may want to explicitly set the bank register immediately before each use of an instruction that accesses a Special Function Register to eliminate the chance that the bank register is not in the correct state.

- ◆ Add useful comments to your code.
  - Every function should begin with a “header” comment that should clearly separate the function from previous code (so reader immediately recognizes that this is where a new function starts). The header should contain, as a minimum, the name of the function, a sentence that states what the function does, and documentation of what arguments it takes (if any), and what value it returns (if any).
  - Add “white-space” (blank lines) to divide long columns of instructions in a routine into readable segments. In general, it takes 2 to 5 assembly instructions to do the job of each basic action you wish to perform, such as setting or toggling a pin on a port. You should precede the instructions with a comment that describes what these 2 to 5 assembly instructions are trying to accomplish so that the reader does not have to read all the instructions and “reverse engineer” your code.
  - Finally, make sure any comments on individual instructions describe what you are trying to accomplish and do not simply echo the meaning of the op-code. For example,

```

BAD:      btfss PORTD, Hook      ; See if bit Hook of port D is set
GOOD:     btfss PORTD, Hook      ; Check to see if phone is off-hook

```

- ◆ Generate pseudo-code descriptions of all of your routines. If you have done the commenting specified above, this should be relatively easy. Just make a copy of your code and go through it, keeping the function headers and the comments before each set of instructions, and deleting the instructions themselves. Then delete the ‘;’ characters, and do any additional fixup needed to make your pseudo-code read well. Depending on your wording in your comments and use of capitalization, etc., you may need to do a fair amount of modifications, but it should go reasonably quickly.
- ◆ Print out your pseudo-code for each function, one per page. You can group several on a single page provided the names begin with a common prefix.
- ◆ Place your pseudo-code descriptions in alphabetical order so that it is easy for a reader to find the code for a particular routine.
- ◆ Finally, generate a “call-tree” that gives an overview of your program. A call-tree is an indented listing of the nesting of your function calls like this:

```

start_main
  init_pic
    init_port_dir
    set_port_in
  init_lcd
    write_lcd_inst
    clock_lcd
  init_synth
  ...

```

This shows that the top-level routine is 'start\_main', and that it calls, 'init\_pic', 'init\_lcd', 'init\_synth', etc. in that order. In turn, 'init\_pic' calls 'init\_port\_dir' and 'set\_port\_in', while 'init\_lcd' calls 'write\_lcd\_inst' which calls 'clock\_lcd'.

To avoid making the call tree excessively long, you should expand a function fully only the first time it is called. For example, if 'write\_lcd\_inst' is called later in the program, do not list 'clock\_lcd' below it.

Such a call-tree gives a good overview of the code, much as a Table of Contents in a book gives an overview of the book. It will also help us track down bugs and will show us the total depth so that we can be sure we will not have "stack-overflow" problems.

## **Deliverables**

Your deliverables are listed below. Each team member should provide this information for their code.

- ◆ A complete copy of your modified code (plus include files), with revisions made to prepare for integration, and with all comments added.
- ◆ Your annotated call-tree.
- ◆ Your pseudo-code (or flowcharts) for each function, one per page, in alphabetical order.

## **Team 3 Future Assignments**

The following gives an overview of the tasks remaining after task 4.

- ◆ Integrate code from all team members together into a single program.
- ◆ Add code to write messages to LCD that show what mode the phone is in.
- ◆ Test the code first on the simulator, and then on the Hardware Test Bench.
- ◆ Get code working on phone hardware!
- ◆ Merge documentation from all team members.