

# The Firmware Development Cycle

# Firmware Development Cycle

1. Introduction
2. The “Burn and Learn” Method
3. Development using a Monitor Program
4. Development Using an In-Circuit Debugger/Simulator/Emulator
5. Development Using a Simulator
6. The Capstone Course Development Cycle
7. Dealing with Limited Vision
8. Assembly/Compilation Process
9. Conclusion

# 1. Introduction

- We have already seen the stages of development of a project:

Conceptualization → Research → Design → Prototyping →  
Testing → Board Layout → Board Manufacture → Board  
Testing → Integration → Final Testing → Production

# 1. Introduction

- Where microcontroller code is required (in testing and integration), the firmware development cycle comes into play.
- Firmware → machine-code placed in nonvolatile memory. It is termed “firm” as it is somewhat permanent.

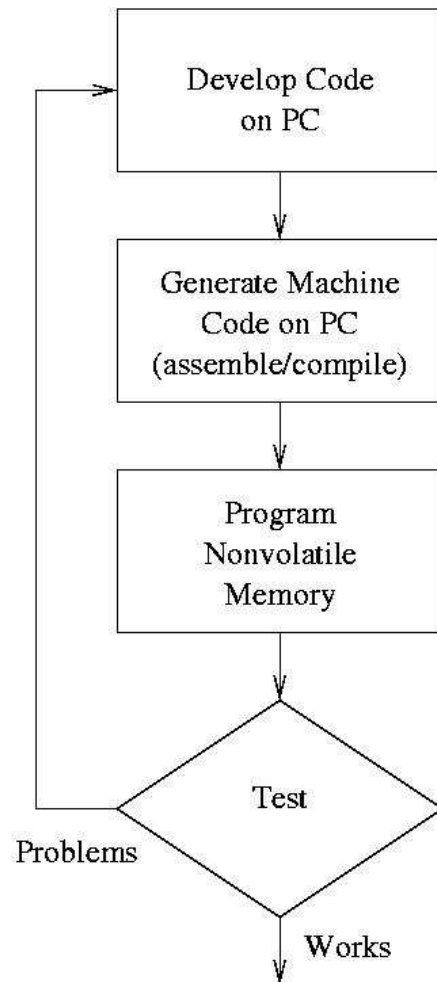
# 1. Introduction

- Firmware is used to control the microcontroller's CPU state-machine to perform some action:
  1. Design-related; or
  2. testing.

# 1. Introduction

- Firmware differs from software in several ways:
  - The goal is to have the final code reside in nonvolatile memory (mask ROM in production runs)
  - Code development is typically not performed on the target CPU: a cross-compiler/assembler is used.
  - The developer receives very limited feedback from the system.

## 2. The “Burn and Learn” Method



- This process is flawed:
  - Identifying the origin of a problem is difficult;
  - it is time-consuming to erase/program non-volatile memory; and
  - some devices have PROM memory and cannot be reprogrammed.

# 3. Development using a Monitor Program

Monitor → a piece of firmware which often includes facilities to:

- download code (into RAM, usually)
- execute code
- trace code
- set breakpoints
- view memory contents

# 3. Development using a Monitor Program

Examples of monitors:

CPU32BUG - MC68020, MC68332

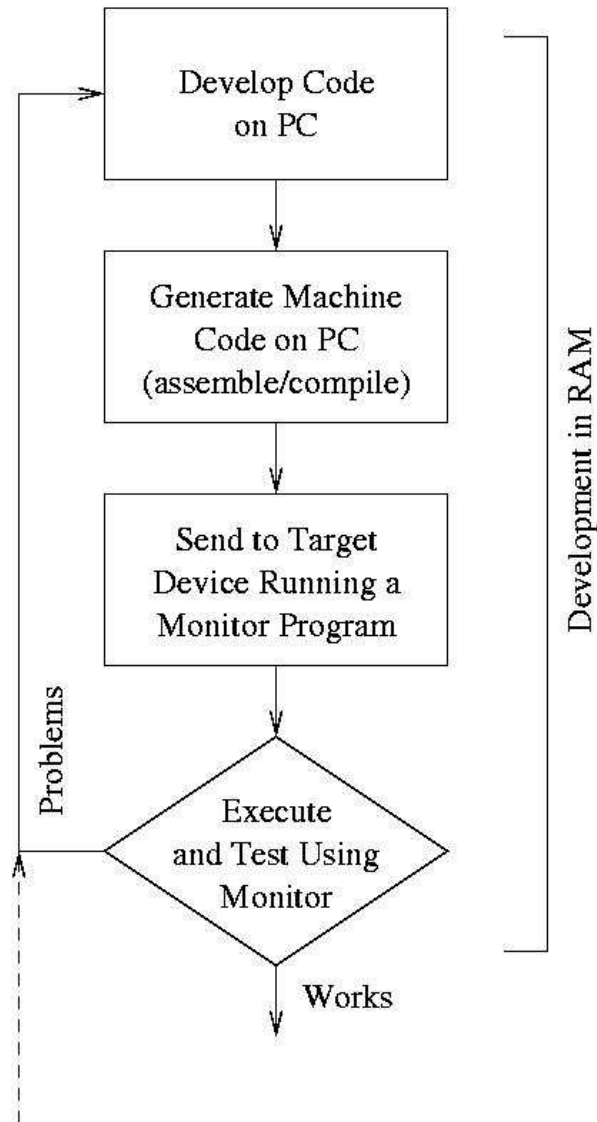
TUTOR - MC68000

BUFFALO - MC68HC11

Micro11 - MC68HC11

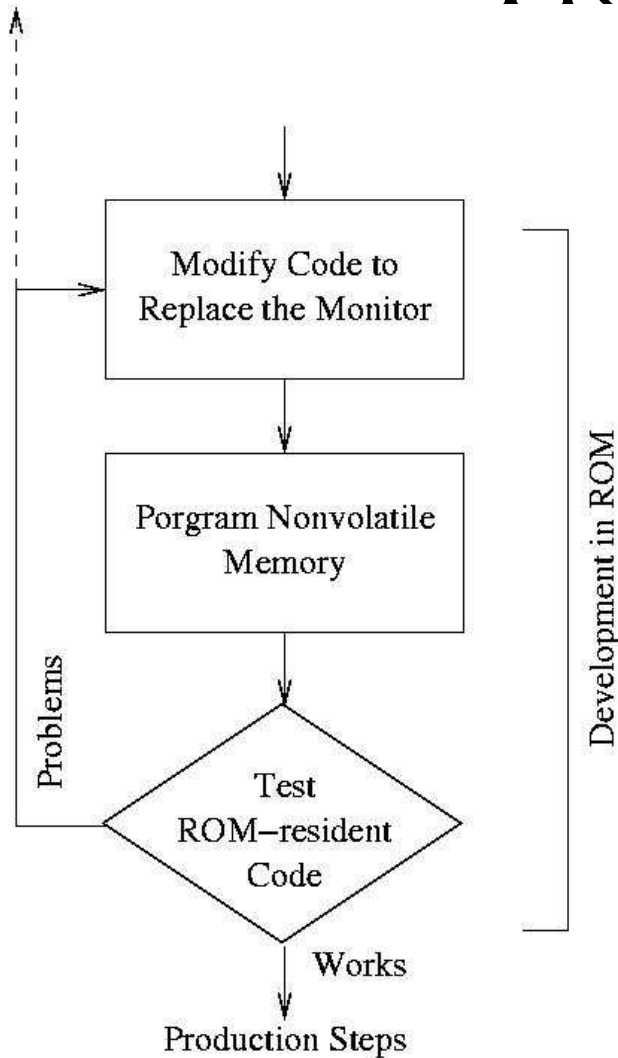
These programs are typically provided by microcontroller manufacturers free of charge, although some cost big \$\$\$.

# 3. Development using a Monitor Program



This is the first half of the development cycle (the RAM portion).

# 3. Development using a Monitor Program



This is the second half of the development cycle (the ROM portion).

### 3. Development using a Monitor Program

Porting of code from RAM to ROM for execution is not a trivial task and requires thorough knowledge of the microcontroller and the assembler/compiler.

Some devices cannot have a comprehensive monitor due to the fact that they are of the “Harvard” structure: program memory and data memory are separate.

## 4. Development Using an In-Circuit Debugger/Simulator/Emulator

- Some devices have special connections used for debugging control and data transfer (see, for instance, JTAG).
- The development sequence looks similar to that used with a monitor program.
- An in-circuit simulator or emulator typically uses a PC or a (more complex) microcontroller to act like the target device.

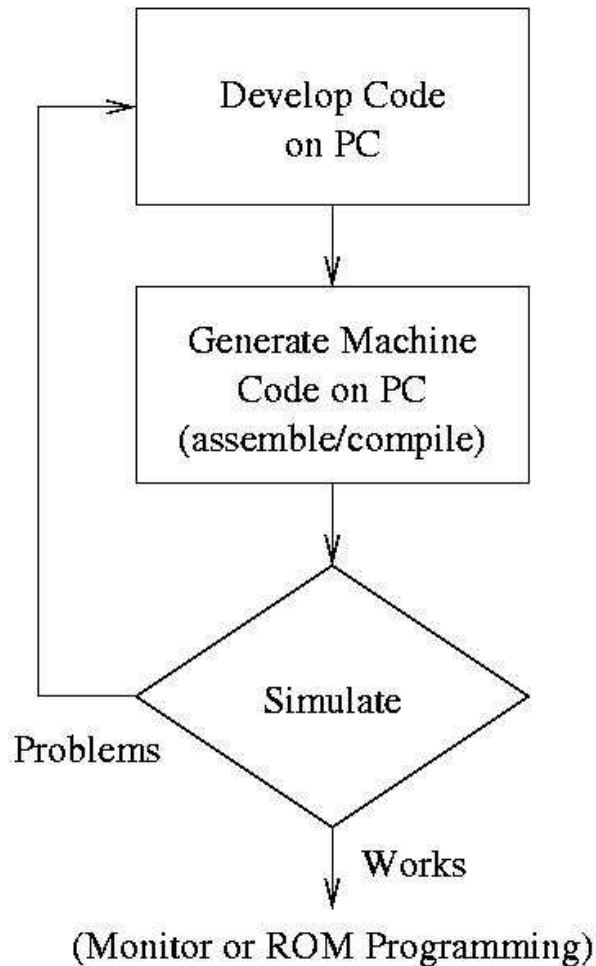
## 4. Development Using an In-Circuit Debugger/Simulator/Emulator

- Most in-circuit systems are usually sold by the microcontroller manufacturer or 3rd party vendors and are quite expensive.

## 5. Development Using a Simulator

- For simpler microcontrollers, there exist PC-based simulators through which code can be tested.
- These simulator are software-only, unlike the “In-Circuit Simulators” as outlined in the last section.

# 5. Development Using a Simulator



- Simulators do not usually run in real-time.
- Some simulators do not simulate all microcontroller peripherals.

## 5. Development Using a Simulator

- It is a lot of work to follow the actions of a simulator, but the effort is well expended.
- Some simulators allow you to add off-chip hardware peripheral simulation, but this typically takes knowledge in PC programming.

## 6. Capstone Course Development Cycle

- Since there are a variety of microcontrollers being used, you will likely see a variety of tools in action.
- Atmel AVR: JTAG In-circuit emulator, simulator.
- PIC16F87X: simulator, burn and learn.
- XInC2: burn and learn.

## 7. Dealing with Limited Vision

- Debugging of firmware can be difficult, particularly when there is limited hardware to allow the user to determine the state of execution.
- Planning ahead and using a few tricks can help this out.

# 7. Dealing with Limited Vision

1. Use a simulator, if possible.
2. Implement code/hardware that can display debugging information first:
  - UART
  - LCD
  - a digital output with an LED or multimeter connected.

## 7. Dealing with Limited Vision

3. Use a colleague's code to determine if the problem is hardware or software-oriented.
4. Simplify the program.
5. Terminate program execution at specific points:

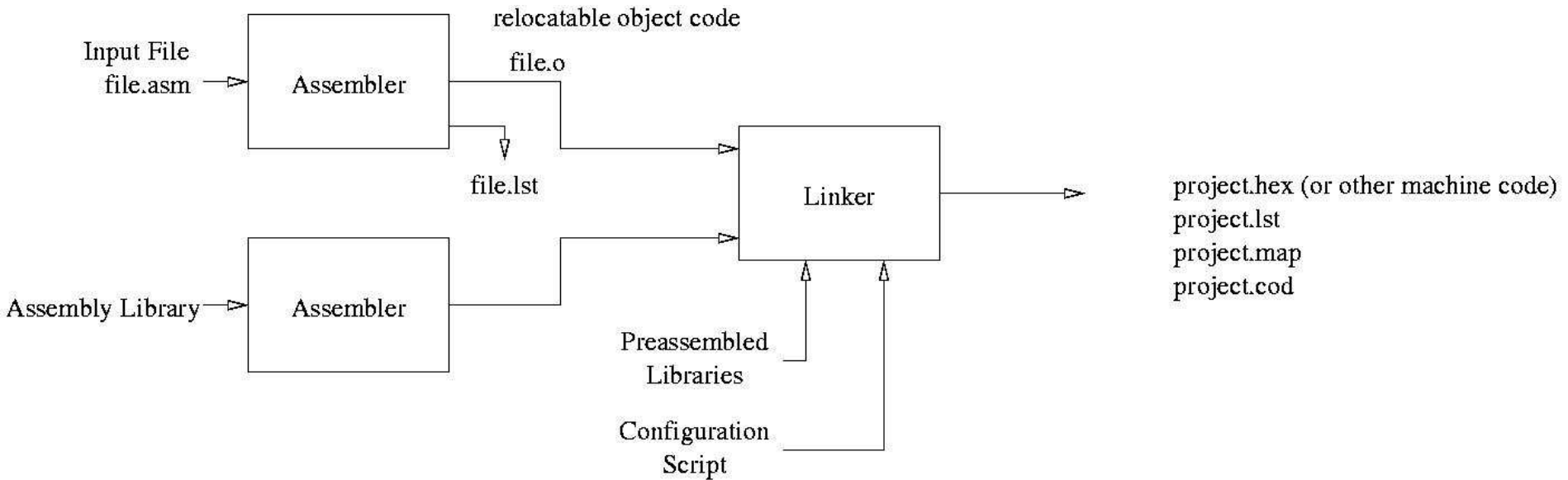
```
Loop:   goto Loop  
while (1);
```

## 7. Dealing with Limited Vision

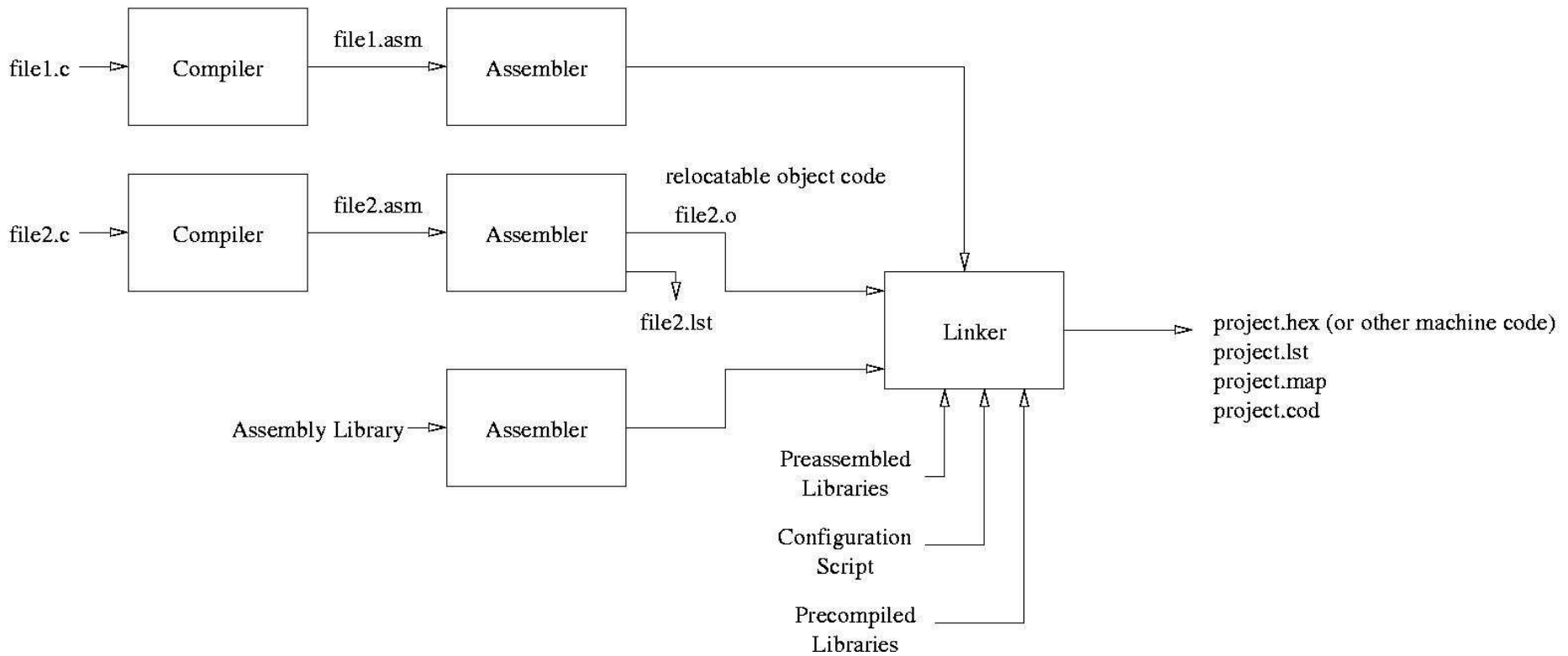
6. Assemble or compile and test your code often.
7. Use a “binary search” to isolate a problem.

Do not necessarily follow these steps in the stated order: they are just a list of techniques that can help.

# 8. Assembly/Compilation Process: Assembly



# 8. Assembly/Compilation Process: Compilation



## 8. Assembly/Compilation Process

- Calling of the assembler/compiler and the linker can be automated through the use of a Makefile or through a “Project” definition (available on many IDEs)

## 8. Assembly/Compilation Process

- Files output by the assembler:
  - .lst - a file listing containing opcodes, line numbers, etc. This is a useful file for debugging as it also contains error information.
  - .o - object (machine) code. This is not yet executable because it doesn't have addresses assigned.

## 8. Assembly/Compilation Process

- Files used by the linker:

.o - the object code from the modules.

A linker script which informs the linker about the availability and location of memory resources on the device. This file occasionally needs to be modified. The standard one is usually sufficient.

## 8. Assembly/Compilation Process

- Files output by the linker:
  - .hex - the “executable” object code that can be run on the device.
  - .lst - the completed listing for the entire project.

## 8. Assembly/Compilation Process

- Files output by the linker:
  - .map - a file that describes how memory has been allocated by the linker.
  - .cod - a special file that contains symbol information useful for simulation of the program.
  - .elf – executable code

## 9. Conclusion

- There's still a lot to learn, but at least you have enough information to begin experimenting!
- You are expected to familiarize yourself with the tools that you will be using. Some tools that may help:
  - Resources link on the course web page
  - help within a particular program
  - search the Internet.
  - “man” or “man -k” on the Linux command-line (if the tool you are using is installed)