1.30 Embedded software development environment

EE302 – Real time and embedded systems

1

Overview

- Aims
  - Show how the embedded software development environment differs from that of the desktop and consider the main issues/implications.
- Learning outcomes – you should be able to…
  - Describe the embedded software development environment and list its main elements
  - Differentiate between the embedded and desktop environments
  - Give some strategies for runtime monitoring and debugging of embedded software

目标。说明嵌入式软件开发环境与桌面软件开发环境的不同之处，并考虑主要问题/影响。
学习成果——
描述嵌入式软件开发环境并列出其主要元素
区分嵌入式环境和桌面环境
给出了嵌入式软件运行时监控与调试的策略

2

Embedded software development

Requirements + design

- **Model based design**
- **Hardware/software co-design**
- Most details beyond scope of this course

Construction

Write code

- **Cross-compile** & link

…and test it

- **Simulate** [maybe]
- **Program device**
- Test with **hardware debugger** or **in-circuit emulator**
- **Evaluation board** or final target system

Repeat!

Final deployment

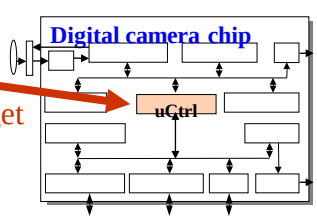
- Beyond scope of this course

Embedded Software Development Environment

- Develop on the **development system**…
  - The development system's processor on which we write (and possibly debug) our programs (usually a PC) usually differs from the target processor
- …for eventual execution on the **target processor** in the **target system**
  - The target system is the embedded system in which our program will be executed
  - The target processor is the actual processor in our embedded system on which our program will run (e.g. PIC16xxx, or ARM, or Intel IA, etc)
  - Usually target processor is different from development processor

Develop here

For target here

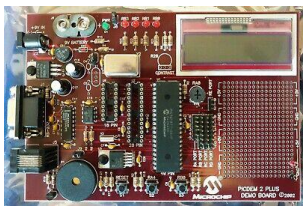


Development vs Target system

Develop here

For target here

ARM based processor



PIC 16xxx
Microcontroller

5

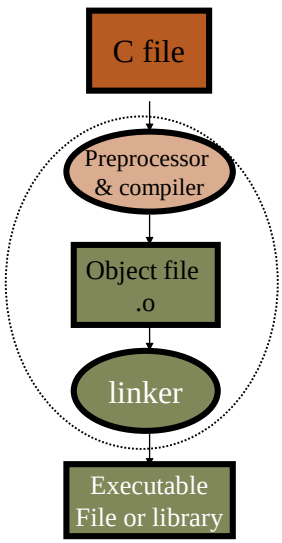
(Cross) compiling and linking

- First lets consider the standard compilation process (for C/C++/assembly programmes)
  - Preprocessor: expands #include, #define, before compiler runs
  - Compiler: Compiles C, C++, etc, to [binary] object
  - Assembler: Assembles assembly language to [binary] object
  - Linker
    - Combines [binary] objects into libraries (if desired)
    - Or combines [binary] objects and libraries into an executable image
- Depending on the tools you use and whether or not your programme is just one file, you might not notice these separate stages, but they do exist
- For more complex systems with multiple files, it becomes more important to understand these stages
- How does cross compiling/linking differ from compiling/linking?
  - Cross compiler/linker runs on development system but outputs instructions for target processor

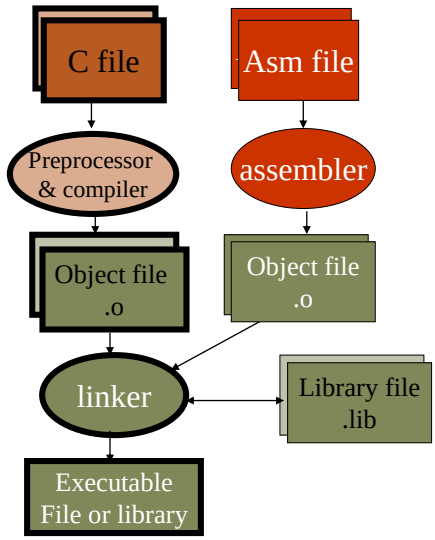
6

(Cross) Compiling and linking

Simple single file app.



Multi-file app. including libraries



Cross compiling vs native compiling

- native compiler
  - The compiler runs on the development machine
  - Generates an executable that runs on the development machine (or similar)
- Cross compiler
  - The compiler runs on the development machine
  - Generates an executable that runs on the target machine (it usually cannot run on the development machine)
- Note:
  - You cannot run or test a cross compiled programme unless you can somehow transfer the executable to the target machine or a simulator

Native

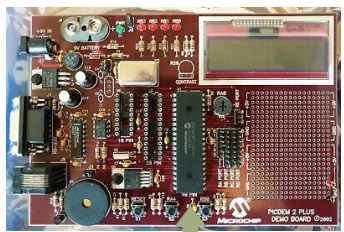
Cross

How to run and test your programme

Software Simulator – runs on development machine

(In circuit) Device Programmer

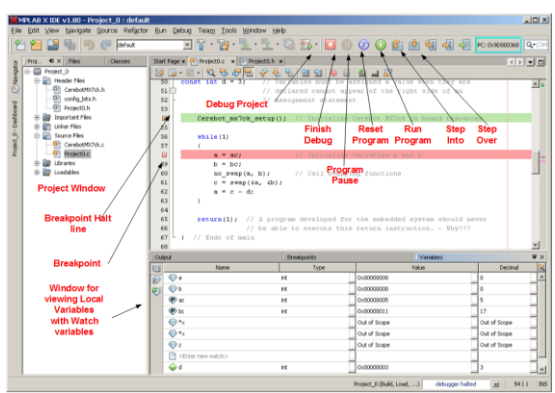
Hardware debugger



In circuit emulator
(replaces chip)

ASIDE: what is a debugger

- A debugger allows you stop your programme at **breakpoints** so that you can inspect values (variables, memory, etc.)
- You can set breakpoints on lines of code. Many debuggers also support conditional breakpoints.
- It may also allow you to see what a programme was doing when it crashed.
- All of this is very useful and important when debugging embedded systems.



Instruction set or system simulator

- Simulate the target processor/target system on the development machine
  - Enables some program verification and testing before programming on target device
  - Allows standard debugger functionality (e.g. breakpoints)
  - This can be convenient and might provide a fast test-debug-fix turnaround for some parts of the development
- However, it usually can't simulate everything
  - Usually slower than real time so not much use for testing issues that arise with asynchronous real time events (e.g. interrupts)
  - Difficult to simulate real input/output (so difficult for testing communications)

Testing tools

- Basic device programmer (without debugger)
  - Facilitates executable upload to the target system
  - Does not provide any additional features for debugging the programme while running so you must use other means to determine what is happening
    - E.g. the Arduino system has an on-board device programmer but doesn't offer debugger functionality by default (at least up to early 2020)
  - Slowest test-debug-fix turnaround in general

- hardware debugger
  - Communicates with real target processor running in-circuit
    - May work with JTAG or BDM industry standard interfaces for testing processors and support chips
  - Some debugger functionality: breakpoints, view registers, etc
    - But usually subject to some restrictions due to limited interface to the processor being debugged
  - Medium test-debug-fix turnaround

Contd.

- In Circuit Emulator (ICE)
  - pin compatible replacement for target processor
  - Real time emulation of target processor, in situ
  - Powerful real time debugging
  - Medium turn around
- Note: none of the solutions for embedded systems offer a fast turnaround.
  - Test-debug-fix is always harder for embedded systems than desktop systems if complexity etc. is similar

How to use the tools in practice

- First cross-compile on the development machine
- Where possible, test parts of the system with a simulator first
  - Eliminates the programmer/upload step and allows good debug functionality
- To test on the target system
  - Upload the executable to the target system using a device programmer
  - If possible, use a hardware debugger or ICE to debug the programme on the target system if there are issues
  - But, note that debuggers are just one tool and are not a magic bullet for all solving all issues
    - E.g. some problems (typically related to timing) cannot always be tested easily with a debugger
- Because of the slower test-debug-fix turnaround for embedded systems, it is well worth putting more effort into code reading and problem hypothesis generation

When a debugger/ICE is not sufficient

- Embedded systems usually have no default concept of sending debug messages to screen or file (since neither might exist on the target system), so alternatives are needed for feedback
- Some possibilities for giving progress/status feedback about target system
  - for both the developer & end user…
    - LEDs: on, off, flashing, etc
    - Beeper sounds
  - for the developers only…
    - Print to output port/serial port (if there is one)
    - Create status/debug packets and send via output port, wired, or wireless communication (e.g. Bluetooth) if available
    - Create text or binary debug records and save them to a serial memory (e.g. a flash memory)
    - Update various debug flag and counter variables etc. that can be inspected using the debugger when it will not affect application timing or after a crash
- Summary: Debugging/testing embedded software is not easy!!

Example questions

1. *Describe the embedded software development environment and list its main elements.*
2. *Differentiate between the embedded and desktop environments.*
3. *Give some strategies for runtime monitoring and debugging of embedded software.*