GETTING STARTED

Embedded Software Design

熊博安

國立中正大學資訊工程研究所

pahsiung@cs.ccu.edu.tw

Textbook: Programming Embedded Systems with C and GNU Development Tools, 2nd Edition, Michael Barr and Anthony Massa, O'Reilly

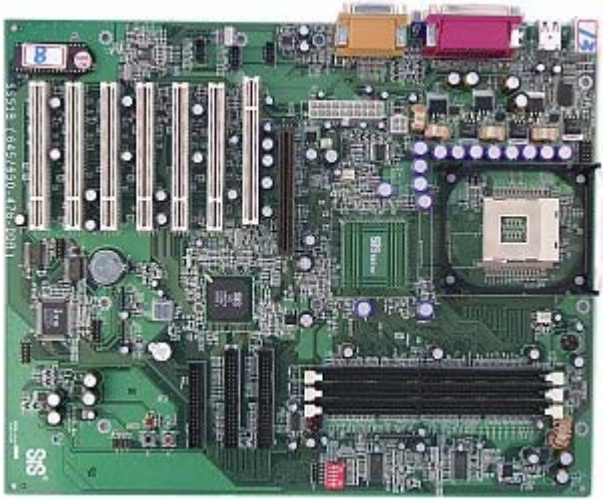
Contents

- Getting to Know the Hardware
- Your First Embedded Program
- Compiling, Linking, and Locating
- Downloading and Debugging

Getting to Know the Hardware

Getting to Know the Hardware

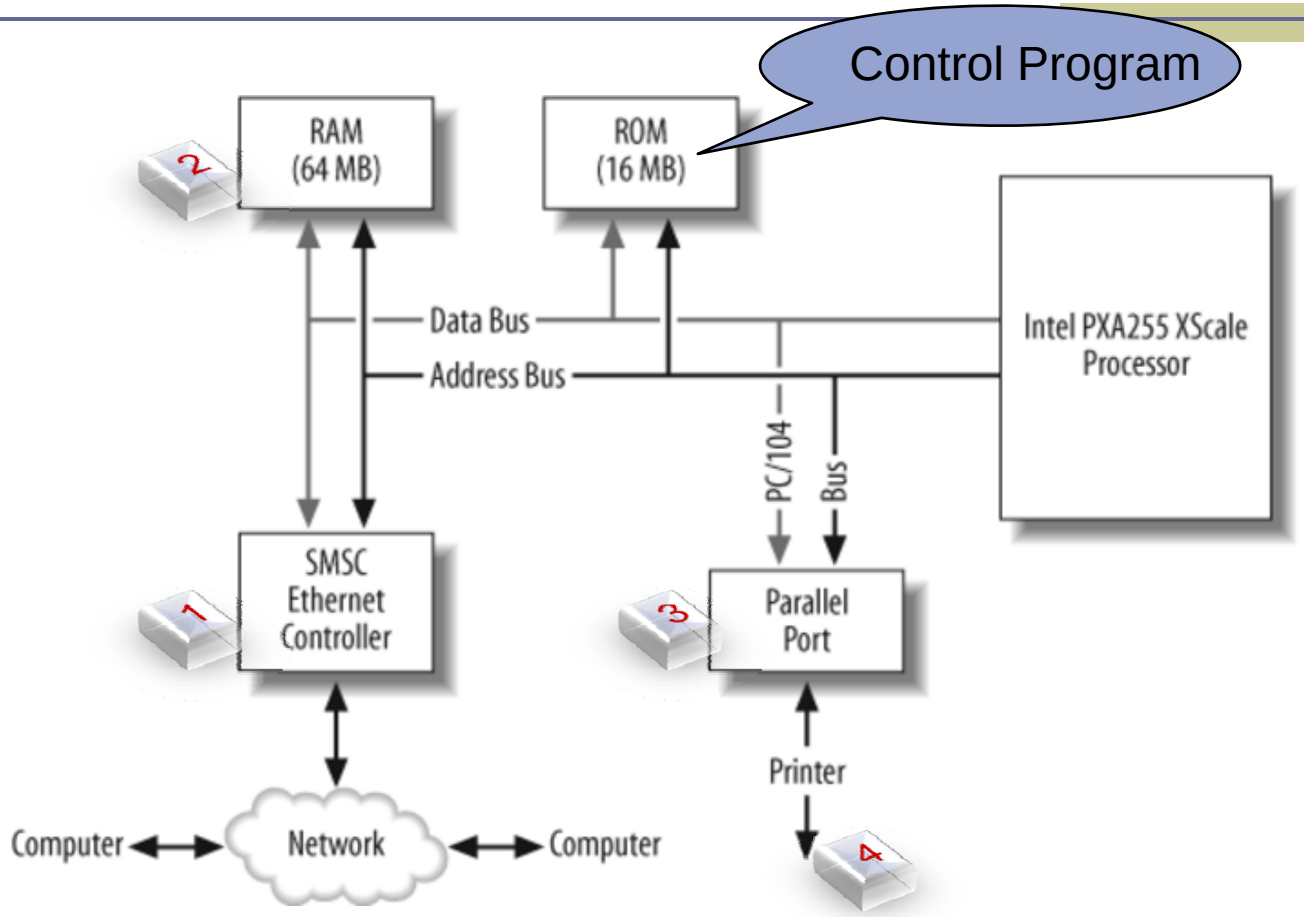
- How to familiarize with a new board?
- How to create a header file with the board's important features?
- How to write software code to initialize a new board?



Understanding the Big Picture

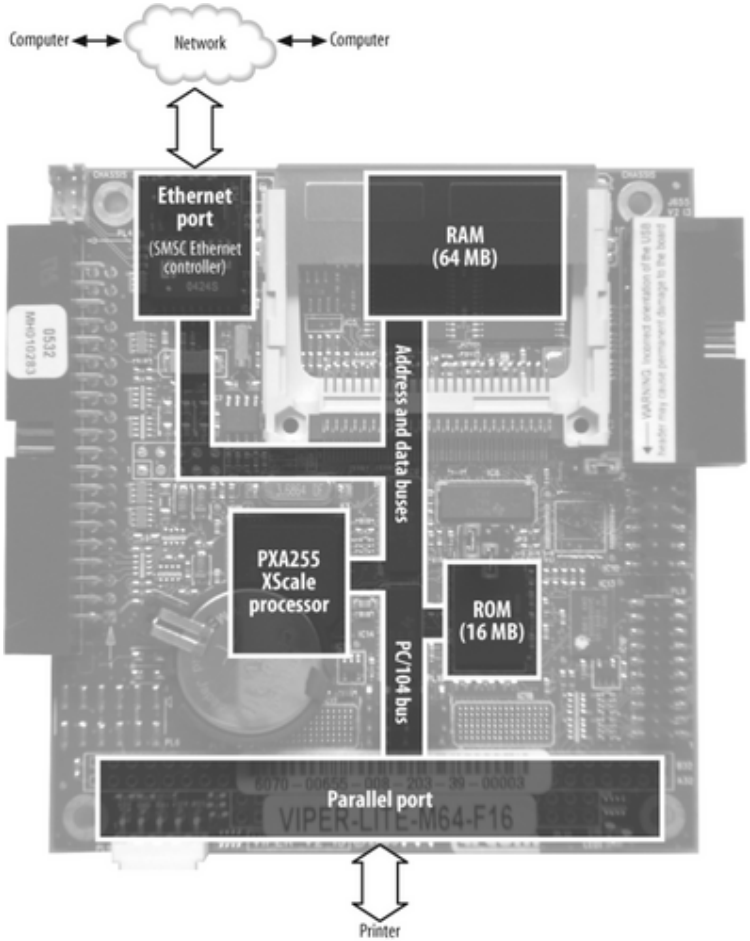
- Understand the general operation of the system first
  - Main function? Inputs? Outputs?
- Read all documentations
  - User's Guide, Programmer's Manual, …
- **Before** picking up the board should answer:
  - What is the overall purpose of the board?
  - How does data flow through it?

Draw your own data-flow diagram



Block Diagram for Print Server using Arcom board

Data-flow on the Board …



Hardware Basics

- Schematic
  - A drawing showing hardware components, interconnections, and oscilloscope probe points
  - Standard symbols
- Datasheets
  - Complete specification of hardware components
    - Electrical, timing, and interface parameters

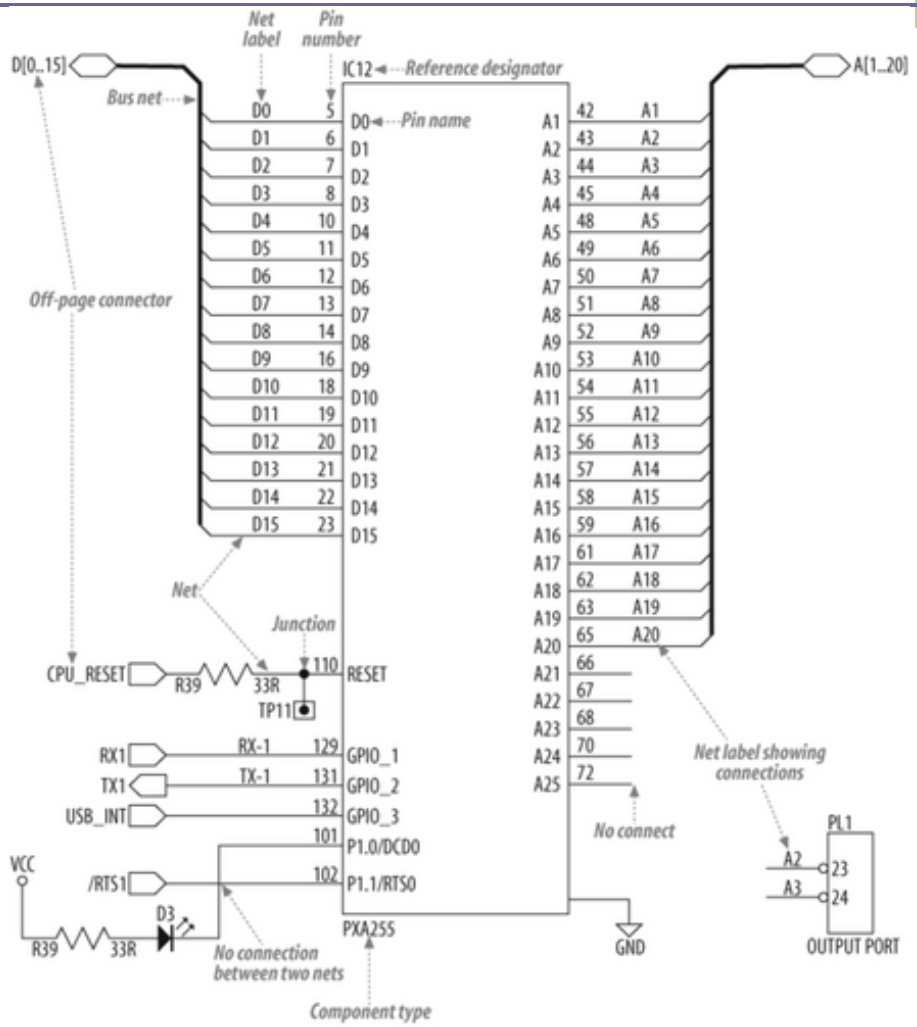
Schematic Fundamentals

Component	Reference Designator Prefix	Symbol
Resistor	R	
Capacitor	C	
Diode	D	
Crystal	X	
Inductor	L	
Power	VCC	
Ground	GND	

Light Emitting Diode (LED)

Basic Schematic Symbols

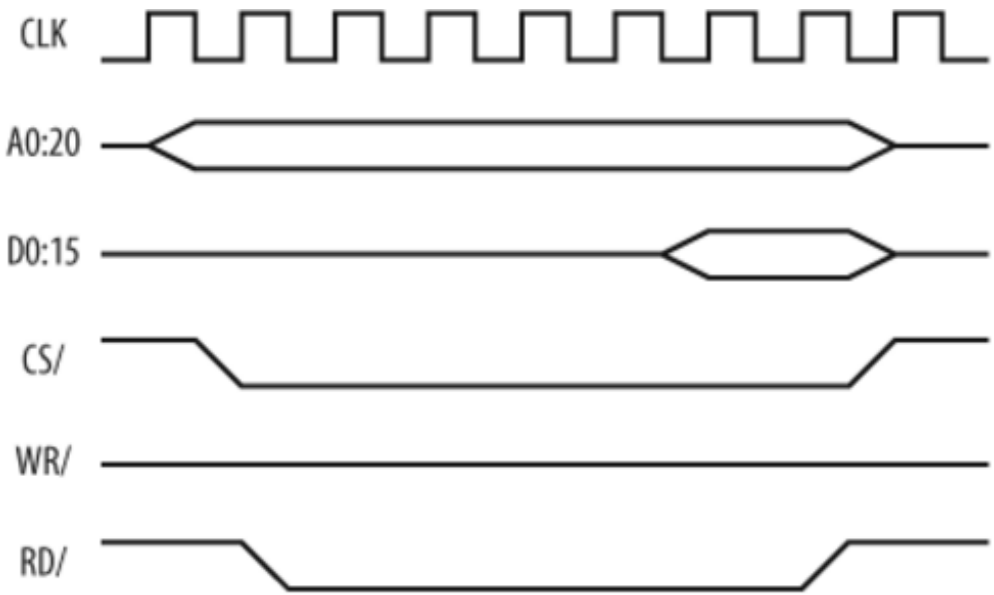
Example Schematic



Examine the landscape

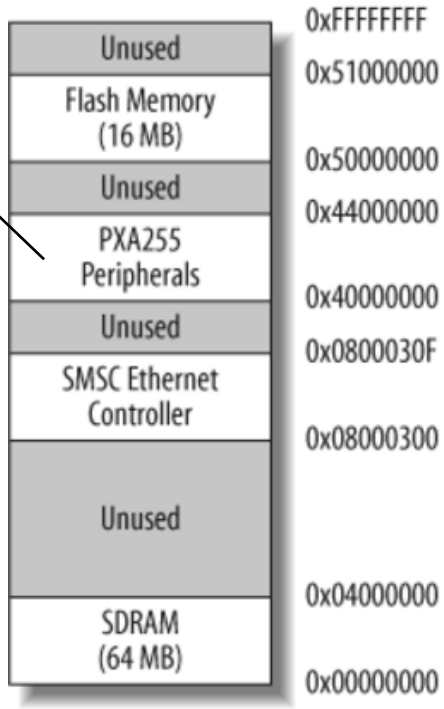
- Put yourself in the processor's SHOES!
- What does the processor's world look like?
- The processor has a lot of compatriots!
  - Memories: Storage and retrieval of data / code
  - Peripherals: Coordinate interaction with outside world (I/O), or specific hardware func
    - Examples: serial ports, timers
- Address Spaces (Address Book of processor)
  - Memory Space
  - I/O Space

Timing Diagram



Memory Map for Arcom Board

Internal Peripheral Registers



Most important information for Embedded Software Engineers in a project

Header File

- Describes most important features of a board
- Abstract interface to hardware
- Refer to devices by name, instead of addresses
  - Makes software more portable
- If the 64 MB RAM is moved,
  - just change header file only, and
  - recompile program
    (no need to change program code)

Header File: Memory Map

```
/*****************************************************
*
* Memory Map
*
*   Base Address  Size   Description
*   ------------  -----  ---------------------------
*   0x00000000    64M    SDRAM
*   0x08000300    N/A    Ethernet controller
*   0x50000000    16M    Flash
* ****************************************************/
#define SDRAM_BASE (0x00000000)
#define ETHERNET_BASE (0x08000300)
#define FLASH_BASE (0x50000000)
```

How to Communicate?

- Two basic communication techniques:
  - polling
  - interrupts
- Processor issues some commands to device
- Processor waits for device to complete action
  - Timer: 1000 → 0 (countdown)

Polling: “are we there yet?”

```
do
{
   /* Play games, read, listen to music, etc. */
   …
   /* Poll to see if we're there yet. */
   status = areWeThereYet();
} while (status == NO);
```

Interrupts

Asynchronous signal from external/internal peripheral or from software to CPU

- Processor issues commands
- Processor does other things
- Device interrupts processor
- Processor suspends its work
- Processor executes interrupt service routine (ISR) or interrupt handler
- Processor returns to the interrupted work

Interrupts

- Initially
  - Not all automatic! Programmer must:
    - write and install ISR
    - enable its execution when interrupt occurs
  - A significant challenge!!!
- Advantages
  - Code is better structured!
  - More efficient than polling!
- Overhead
  - save registers in memory, disable lower-priority interrupts, transfer control, etc.

Interrupts vs. Polling

- Both are used frequently in practice
- When to used interrupts?
  - Efficiency if paramount
  - Multiple devices must be monitored simultaneously
- When to use polling?
  - When it is required to respond more quickly than is possible using interrupts
  - Large amounts of data are expected to arrive (real-time data acquisition)

Getting to Know the Processor

- Read databooks of processors!
  - What address does the processor jump to after a reset?
  - What is the state of the processor's registers and peripherals at reset?
  - What is the proper sequence to program a peripheral's registers?
  - Where should the interrupt vector table be located?
    - Does it have to be located at a specific address in memory? If not, how does the processor know where to find it?

Getting to Know the Processor

- What is the format of interrupt vector table?
  - Is it just a table of pointers to ISR functions?
- Are there any special interrupts – sometimes called traps – that are generated within the processor itself?
  - Must an ISR be written to handle each of these?
- How are interrupts enabled and disabled?
  - Globally and individually?
- How are interrupts acknowledged or cleared?

Getting to Know the Processor

- Three types of processors:
  - Microprocessors: powerful, general-purpose,
    Eg: Freescale's 68K, Intel's 80x86
  - Microcontrollers: less powerful, embedded system-specific,
    Eg: 8051, Motorola's 68HCxx, Intel's 386EX
  - Digital Signal Processors (DSP): fast calculations of discrete-time signals,
    Eg Vendors: Analog Devices, Freescale, TI

The PXA255 XScale Processor

- PXA255 uses the XScale core
  - ARM Version 5TE architecture
- **On-chip peripherals:**
  - Interrupt control unit
  - Memory controller
  - Several general-purpose I/O pins
  - 4 timer/counters
  - I^2C bus interface unit
  - 4 serial ports
  - 16 DMA channels
  - Memory controller for DRAM
  - USB client
  - LCD controller
  - 2 PWM (pulse width modulators)
  - A real-time clock
  - A watchdog timer unit
  - A power management unit
- CPU controls them via internal buses

Header file for On-Chip Peripherals

- What is going on here?

```
If (bLedEnable == TRUE)
{
  *((uint32_t *)0x40E00018 = 0x00400000;
}
```

- Need more intuitive ways of writing code!
  - Create and use header files!

PXA255 On-Chip Peripherals (1/4)

```
/*****************************************************************
* PXA255 XScale ARM Processor On-Chip Peripherals
*****************************************************************/
/* Timer Registers */
#define TIMER_0_MATCH_REG        (*((uint32_t volatile *)0x40A00000))
#define TIMER_1_MATCH_REG        (*((uint32_t volatile *)0x40A00004))
#define TIMER_2_MATCH_REG        (*((uint32_t volatile *)0x40A00008))
#define TIMER_3_MATCH_REG        (*((uint32_t volatile *)0x40A0000C))
#define TIMER_COUNT_REG          (*((uint32_t volatile *)0x40A00010))
#define TIMER_STATUS_REG         (*((uint32_t volatile *)0x40A00014))
#define TIMER_INT_ENABLE_REG     (*((uint32_t volatile *)0x40A0001C))
```

PXA255 On-Chip Peripherals (2/4)

```
/* Timer Interrupt Enable Register Bit Descriptions */
#define TIMER_0_INTEN        (0x01)
#define TIMER_1_INTEN        (0x02)
#define TIMER_2_INTEN        (0x04)
#define TIMER_3_INTEN        (0x08)

/* Timer Status Register Bit Descriptions */
#define TIMER_0_MATCH        (0x01)
#define TIMER_1_MATCH        (0x02)
#define TIMER_2_MATCH        (0x04)
#define TIMER_3_MATCH        (0x08)

/* Interrupt Controller Registers */
#define INTERRUPT_PENDING_REG    (*((uint32_t volatile *)0x40D00000))
#define INTERRUPT_ENABLE_REG     (*((uint32_t volatile *)0x40D00004))
#define INTERRUPT_TYPE_REG       (*((uint32_t volatile *)0x40D00008))
```

PXA255 On-Chip Peripherals (3/4)

- /* Interrupt Enable Register Bit Descriptions */
- #define GPIO_0_ENABLE (0x00000100)
- #define UART_ENABLE (0x00400000)
- #define TIMER_0_ENABLE (0x04000000)
- #define TIMER_1_ENABLE (0x08000000)
- #define TIMER_2_ENABLE (0x10000000)
- #define TIMER_3_ENABLE (0x20000000)

PXA255 On-Chip Peripherals (4/4)

```
/* General Purpose I/O (GPIO) Registers */
#define GPIO_0_LEVEL_REG        (*((uint32_t volatile *)0x40E00000))
#define GPIO_1_LEVEL_REG        (*((uint32_t volatile *)0x40E00004))
#define GPIO_2_LEVEL_REG        (*((uint32_t volatile *)0x40E00008))
#define GPIO_0_DIRECTION_REG (*((uint32_t volatile *)0x40E0000C))
#define GPIO_1_DIRECTION_REG (*((uint32_t volatile *)0x40E00010))
#define GPIO_2_DIRECTION_REG (*((uint32_t volatile *)0x40E00014))
#define GPIO_0_SET_REG          (*((uint32_t volatile *)0x40E00018))
#define GPIO_1_SET_REG          (*((uint32_t volatile *)0x40E0001C))
#define GPIO_2_SET_REG          (*((uint32_t volatile *)0x40E00020))
#define GPIO_0_CLEAR_REG        (*((uint32_t volatile *)0x40E00024))
#define GPIO_1_CLEAR_REG        (*((uint32_t volatile *)0x40E00028))
#define GPIO_2_CLEAR_REG        (*((uint32_t volatile *)0x40E0002C))
#define GPIO_0_FUNC_LO_REG      (*((uint32_t volatile *)0x40E00054))
#define GPIO_0_FUNC_HI_REG      (*((uint32_t volatile *)0x40E00058))
```

Code snippet easier to read

- The earlier code snippet becomes much easier to read …

```
If (bLedEnable == TRUE)
{
  *((uint32_t *)0x40E00018 = 0x00400000;
}
```

```
if (bLedEnable == TRUE)
{
    GPIO_0_SET_REG = 0x00400000;
}
```

- which means: some bit of GPIO 0 is set!

Study the External Peripherals

- LCD, keyboard controllers, A/D converters, network interface chips, or ASICs
- Arcom VIPER-Lite Board:
  - SMSC Ethernet controller and
  - Parallel port

Study the External Peripherals

- Need datasheet for each device to answer:
  - What does the device do?
  - What registers are used to issue commands and receive results?
  - What do the bits and larger fields in registers mean?
  - Does the device generate interrupts?
  - How are interrupts acknowledged or cleared at device?
- Good idea to write a device driver for each device
  - A collection of software routines:
    - To control the operation of a peripheral
    - To isolate application software from hardware device

Initialize the Hardware

at reset address,
2~3 assembly instr.

1

```
reset:
; Reset Code
; (in assembly)
jmp hw_init
```

2

```
hw_init:
; Hardware
; Initialization
; (in assembly)
...
jmp startup
```

- Tell processor about its env,
- Initialize interrupt ctrl, …(peri)
- Initialize chip-select regs

3

```
startup:
; Startup Code
; (in assembly)
...
call main
```

- Setup stack, heap, …
- Initialize global vars

4

```
main()
{
 /*The C/C++ program starts here. */
...

}
```

Reset Address for PXA255 is 0x00000000.

Your First Embedded Program

Your First Embedded Program

Hello World! … ???

- A difficult and perhaps even impossible program to implement in embedded systems
- Printing of text:
  - More an end-point than a beginning
  - Need an output device, which may be lacking
  - Need a display driver (a challenging program!)
- Need a no-brainer!

Embedded Programmers

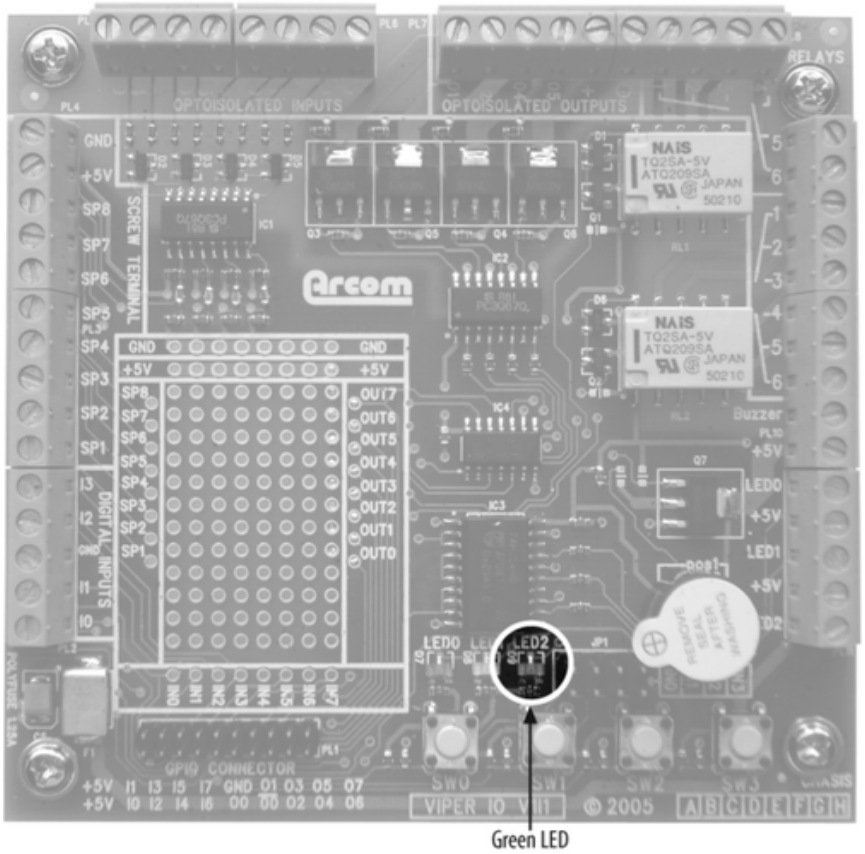
- Must be self-reliant
- Assume nothing works first
- All you can rely on is programming language syntax
- Standard library routines might not be available (`printf`(), `memcpy`(), etc.)
- Every embedded system has at least one LED controlled by software

First Program

- Blink an LED at a rate of 1 Hz
- 1 Hz = 1 complete on-off cycle per second
- A few lines of C or assembly
- Little room for programming errors
- Underlying concept extremely portable
- Hardware-independent program
  - Some functions are hardware-dependent

How to control Green LED (LED2)?

- LEDs on add-on module
- Look up Technical and I/O Manuals
- Check the schematics to trace connection from LED back to processor
  - OUT2 signal
  - Inverted!
  - GPIO pin 22



Blinking LED Program Code

```
int main(void) {
    ledInit(); /* Configure green LED control pin */
    while (1) {
        /* Change green LED state */
        ledToggle();
        /* Pause for 500 ms */
        delay_ms(500);
    }
    return 0;
}
```

Hardware-Independent

Hardware-Dependent

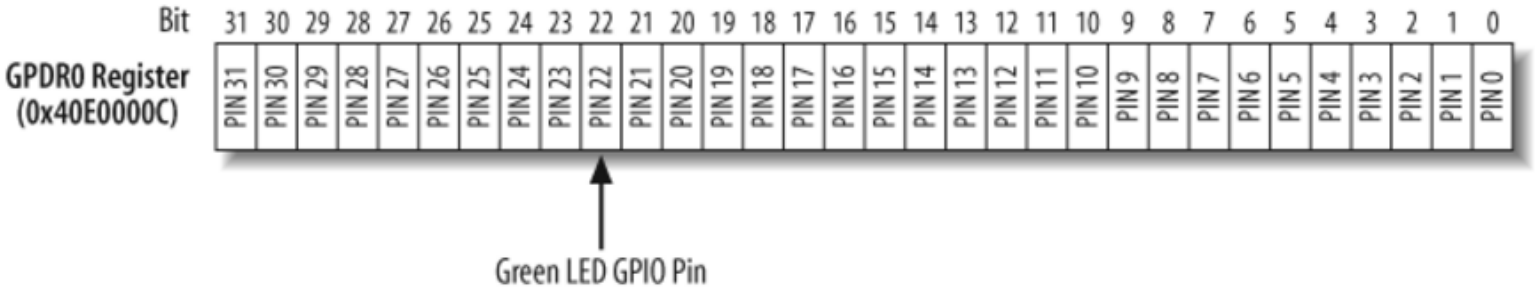
The ledInit Function

- LED2 connected to one of PXA255 processor's 85 bidirectional GPIO pins
  - Multiple functions
    - User-controllable I/O, or
    - To support particular peripheral functionality
  - Configuration registers are used to select how each pin is used by an application
    - Alternate-function 1, 2, 3, (system defined) or
    - General-purpose pin (used by user)

PXA255 GPIO registers

Register name	Type	Address	Name	Purpose
GPLR0	Read-only	0x40E00000	GPIO Pin-Level Register	Reflects the state of each GPIO pin. 0 = Pin state is low. 1 = Pin state is high.
GPDR0	Read/write	0x40E0000C	GPIO Pin Direction Register	Controls whether a pin is an input or output. 0 = Pin is configured as an input. 1 = Pin is configured as an output.
GPSR0	Write-only	0x40E00018	GPIO Pin Output Set Register	For pins configured as output, the pin is set high by writing a 1 to the appropriate bit in this register. 0 = Pin is unaffected. 1 = If configured as output, pin level is set high.
GPCR0	Write-only	0x40E00024	GPIO Pin Output Clear Register	For pins configured as output, the pin is set low by writing a 1 to the appropriate bit in this register. 0 = Pin is unaffected. 1 = If configured as output, pin level is set low.
GAFR0_U	Read/write	0x40E00058	GPIO Alternate Function Register (High)	Configures GPIO pins for general I/O or alternate functionality. 00 = GPIO pin is used as general-purpose I/O. 01 = GPIO pin is used for alternate function 1. 10 = GPIO pin is used for alternate function 2. 11 = GPIO pin is used for alternate function 3.

PXA255 processor GPDR0 register (configuration)



- Reserved bits should not be used!
  - In PXA255, reserved bits should be written as zeros, and ignored during read.
- Good practice: always initialized HW before use, even if default behavior is fine
  - Eg.: By default, all GPIO pins are configured as INPUTS after RESET. How do you know if it is still input before using a pin?

I/O Space Register Access

- Use assembly language
  - **`80x86: in`** and **`out`**
- No built-in support in the C language
  - **`80x86 std lib: inport(), outport()`**
- Use assembly instead of C for accessing 80x86 I/O space registers

LED Initialization

- Configure GPIO pin 22 as
  - Output: Set bit 22 of GPDR0 register
  - General-purpose I/O: Clear bit 22 of GAFR0_U register
- Blinking LED without library routines
  - Read contents of register
  - Modify the bit that controls the LED
  - Write value back to register

ledInit Function

- Bitmask for GPIO pin 22 (controls green LED)

```
#define LED_GREEN     (0x00400000)
```

- Two read-modify-write operations (in order)
  - On GAFR0_U
  - On GPDR0 (clear LED_GREEN)

ledInit Function

```
#define PIN22_FUNC_GENERAL (0xFFFFCFFF)
/* Function: ledInit
•Description: Initialize GPIO pin controlling LED. */

void ledInit(void) {
   /* Turn the GPIO pin voltage off, which will light
      the LED. This should be done before the pins are
      configured. */
        GPIO_0_CLEAR_REG = LED_GREEN;
   /* Make sure the LED control pin is set to perform
      general purpose functions. RedBoot may have
      changed the pin's operation. */
        GPIO_0_FUNC_HI_REG &= PIN22_FUNC_GENERAL;
   /* Set the LED control pin to operate as output. */
        GPIO_0_DIRECTION_REG |= LED_GREEN;  }
```

ledToggle Function

- Separate set and clear registers in PXA255
  - Cannot use read-modify-write method
- Algorithm
  - Use LED_GREEN bitmask (bit 22)
  - Check current state
    - Read: GPIO_0_LEVEL_REG
  - Depending on current state, toggle state:
    - To turn on LED: GPIO_0_CLEAR_REG
    - To turn off LED: GPIO_0_SET_REG

ledToggle Function

```
/*****************************************
* Function: ledToggle
* Description: Toggle the state of one LED.
* Returns: None.
*****************************************/
void ledToggle(void) {
  /* Check the current state of the LED control
     pin. Then change the state accordingly. */
     if (GPIO_0_LEVEL_REG & LED_GREEN)
          GPIO_0_CLEAR_REG = LED_GREEN;
     else
          GPIO_0_SET_REG = LED_GREEN;
}
```

delay_ms Function

- delay_ms(500) ➔ delay 500 ms = 0.5 sec
- Delay can be implemented as busy-waiting
- #while-loop iterations
  = delay in ms × CYCLES_PER_MS
- while-loop iteration = decrement-and-test cycle
- CYCLES_PER_MS:
  - determined by trial and error
  - depends on processor type and speed
  - Can use a hardware timer for better accuracy

delay_ms Function

```
#define CYCLES_PER_MS (9000)
/*********************************************
* Function: delay_ms
* Description: Busy-wait for requested num of ms.
* Notes: The number of decrement-and-test cycles per ms
        was determined through trial and error. This value
        is dependent upon the processor type, speed,
        compiler, optimization.
* Returns: None.
*********************************************/
void delay_ms(int milliseconds) {
        long volatile cycles=(milliseconds*CYCLES_PER_MS);
        while (cycles != 0)
                cycles--;
}
```

Porting to Other Platforms

- 4 functions
  - `main()`
  - `ledInit()`
  - `ledToggle()`
  - `delay_ms()`
- Read documentation
- Rewrite ledInit(), ledToggle()
- Change CYCLES_PER_MS

Role of Infinite Loop

- Embedded programs almost always end with an infinite loop
- Infinite loop
  - Is a significant part of system functionality
  - Intended to run until
    - world ends or
    - board is reset
  - A very common behavior in embedded programs

“Hello World” in ARM Assembly

```
        AREA   HelloW,CODE,READONLY ; declare code area
SWI_WriteC  EQU          &0     ; output r0[7:0]
SWI_Exit    EQU          &11    ; finish program
            ENTRY               ; code entry point
START       ADR    r1, TEXT     ; r1 → “Hello World”
LOOP        LDRB   r0,[r1],#1   ; get next byte
            CMP    r0, #0       ; check end
            SWINE  SWI_WriteC   ; if not end print
            BNE    LOOP         ; .. & loop back
            SWI    SWI_Exit     ; end of execution
TEXT        =      “Hello World”,&0a,&0d,0
            END                 ; end of source
```

“Hello World” in C for ARM

```
#include <stdio.h>

int main() {
   printf(“Hello World\n”);
   return (0);
}
```

- Save in `HelloW.c` file
- Create a new project using Project Manager
- Add this file into the project
- Click “**Build**” button
- Click “**Go**” button to run on the ARMulator
- Output can be seen in terminal window

“Hello World” in ARM & Thumb

```
        AREA    HelloW_T,CODE,READONLY    ; declare code area
SWI_WriteC      EQU     &0                ; output r0[7:0]
SWI_Exit EQU            &11               ; finish program
                ENTRY                     ; code entry point
                CODE32                    ; enter in ARM state
                ADR     r0, START+1       ; get Thumb entry address
                BX      r0                ; enter Thumb area
                CODE16
START           ADR     r1, TEXT          ; r1 → “Hello World”
LOOP            LDRB    r0,[r1]           ; get next byte
                ADD     r1, r1, #1        ; increment point … **T
                CMP     r0, #0            ; check end
                BEQ     DONE              ; finished? … **T
                SWI     SWI_WriteC        ; if not end print
                B       LOOP              ; .. & loop back
DONE            SWI     SWI_Exit          ; end of execution
                ALIGN                     ; to ensure ADR works
TEXT            DATA
                =       “Hello World”,&0a,&0d,0
                END                       ; end of source
```

ARM vs. Thumb Code Size

- ARM
  - # instructions = 6; size = 24 bytes
  - data = 14 bytes
  - Total = 38 bytes
- Thumb (ignoring preamble to enter Thumb)
  - # instructions = 8; size = 16 bytes
  - data = 14 bytes
  - Total = 30 bytes

Compiling, Linking, and Locating

Embedded Programming

- Not substantially different from other programming
- Main difference:
  - Each target hardware platform is **UNIQUE**
    - Adds a lot of additional software complexity

Software	**Hardware1**
Software	**Hardware2**
Software	**Hardware3**

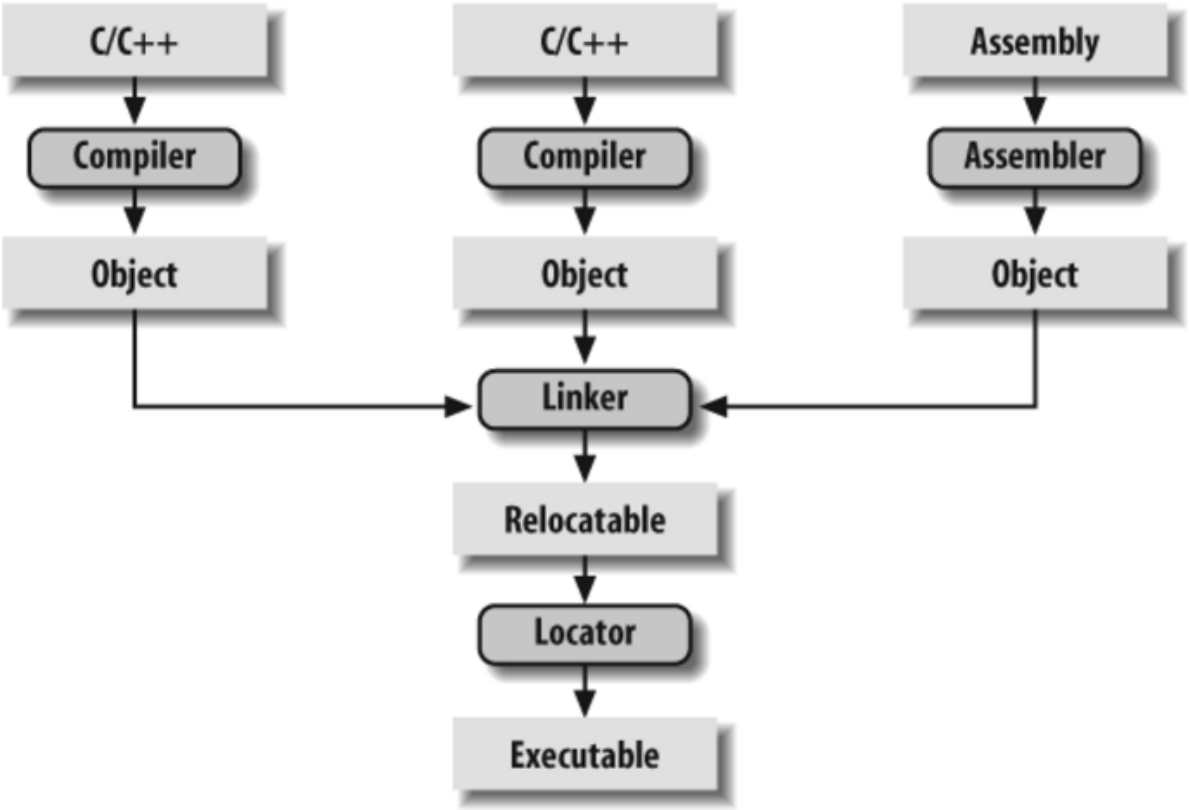
The Build Process

- Not as automatic as conventional programming
  - Cannot make assumptions about target platform because it is different from the host computer where development is done.
- Need to define or provide knowledge about the system to design tools

The Build Process

- Source Code → Embedded SW Executable Program
  - Compilation: Compile or assemble each Source File into Object File
  - Linking: Link all object files into a single object file (relocatable program)
  - Relocation: relative offsets ← physical memory addresses

The Build Process

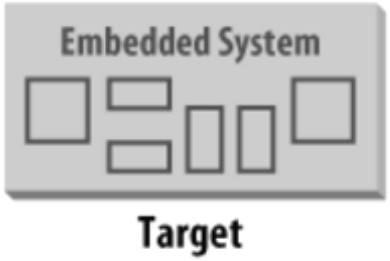


The Build Process (Host vs Target)

- Host Computer:
  - A general-purpose computer:
    - PC or workstation
  - Compiler, Assembler
  - Linker,
  - Locator
- Target Embedded System
  - Run the executable binary image

Split between Host and Target

The development tools that build the embedded software run on a general-purpose computer



The embedded software that is built by those tools runs on the embedded system

GNU Tools

- Freely available
- Open source
- Includes
  - Compiler, assembler, linker, debugger
- Supports
  - Many popular embedded processors
- Manuals
  - http://www.gnu.org/manual

Compiling

- Human-readable language → processor opcodes
- Assembler = assembly language compiler
- Each processor has its own unique machine language
- Compiler must produce programs for your specific target processor (e.g. ARM7TDMI)

Cross-Compilers

- Compiler runs on host computer (NOT target embedded system), called: Cross-Compiler
- Can be configured as native compilers or cross-compilers
  - GNU C/C++ Compiler (gcc)
  - GNU Assembler (as)
- An impressive set of host-target combinations

Hosts/Targets supported by gcc

Host Platforms	Target Processors
DEC Alpha Digital Unix	AMD/Intel x86 (32-bit only)
HP 9000/700 HP-UX	Fujitsu SPARClite
IBM Power PC AIX	Hitachi H8/300, H8/300H, H8/S
IBM RS6000 AIX	Hitachi SH
SGI Iris IRIX	IBM/Motorola PowerPC
Sun SPARC Solaris	Intel i960
Sun SPARC SunOS	MIPS R3xxx, R4xx0
X86 Windows 95/NT	Mitsubishi D10V, M32R/D
X86 Red Hat Linux	Motorola 68k
	Sun SPARC, MicroSPARC
	Toshiba TX39

- ARM family of processors
  - ELF, PE (COFF), AOUT formats
- Additional information
  - http://gcc.gnu.org

Object Files

- Not executable
- A very large, flexible data structure
- Standard structure Formats:
  - Common Object File Format (COFF)
  - Extended Linker Format (ELF)
- If you use different compilers for different source files
  - Ensure all object files are in SAME FORMAT!
- Some vendors have proprietary formats:
  - Buy all development tools from same vendor!

Object Files

- Begins with a header
  - Describes sections in the file
- Code blocks are regrouped by compiler into related sections
  - Text: all code blocks
  - Data: all initialized global variables with initial values
  - Bss: all uninitialized global variables
  - Symbol Table: Names & locations of all variables and functions

Linking

- Object files are incomplete
  - internal variables not resolved
  - functions not resolved
- Job of linker:
  - combine all object files
  - resolve unresolved symbols in symbol table
  - merge text, data, and bss sections
  - output a new object file in the same format

Linking

- GNU linker: ld (a command-line tool)
- For embedded development, a special **compiled startup code** must also be linked!!!
  - Examples
    - startup.asm (for assembly language)
    - crt0.s (short for C runtime)
  - Location of startup code is compiler-specific

Linking

- Same symbol declared in more than one object file
  - display error message
  - exit
- Unresolved symbol after all object files linked
  - check in standard libraries
  - associate code and data sections within output object file (static linking)
  - use stub codes (dynamic linking)

Linking standard libraries

- Some changes required before linking standard library object files
  - Not possible with object files
- Cygnus (part of Red Hat) provides freeware version of standard C library for use in embedded systems (NEWLIB)
  - Download newlib (http://sourceware.org/newlib)
  - Implement target-specific functions
  - Compile everything
  - Link it with your embedded software

Linking

- After merging all code and data sections
  - linker produces a “relocatable” program
  - (no memory addresses assigned to code and data)
- Even OS is statically linked together with embedded application
- Executed as a single binary image

Startup Code

- Disable all interrupts
- Copy any initialized data from ROM to RAM
- Zero the uninitialized data area
- Allocate space for and initialize the stack
- Initialize the processor's stack pointer
- Create and initialize the heap
- Execute the constructors and initializers for all global variables (C++ only)
- Enable interrupts
- Call main

Startup Code

- Some instructions may follow main()
  - executed after main() returns
  - to halt the processor
  - reset entire system
  - transfer control to a debugging tool
- Not inserted automatically
- Programmer must:
  - assemble it and
  - link it with other object files
- GNU startup codes in GNU package: libgloss

Locating

- Job of Locator:
  - Relocatable program → Executable binary image
- Programmer provides information about memory on target board as input to locator
- Locator assigns physical memory addresses to each code and data sections
- Locator produces an output file that can be loaded into the target ROM

Locator

- A separate development tool (sometimes)
- Locator built into linker (GNU ld)
- General-purpose computer: OS does relocation at load time
- Embedded systems: programmer performs the relocation using a special tool
- Memory information:
  - A linker script
  - Controls exact order of code/data sections
  - Establish locations of each section in memory

Linker Script: 64 MB RAM, 16 MB ROM

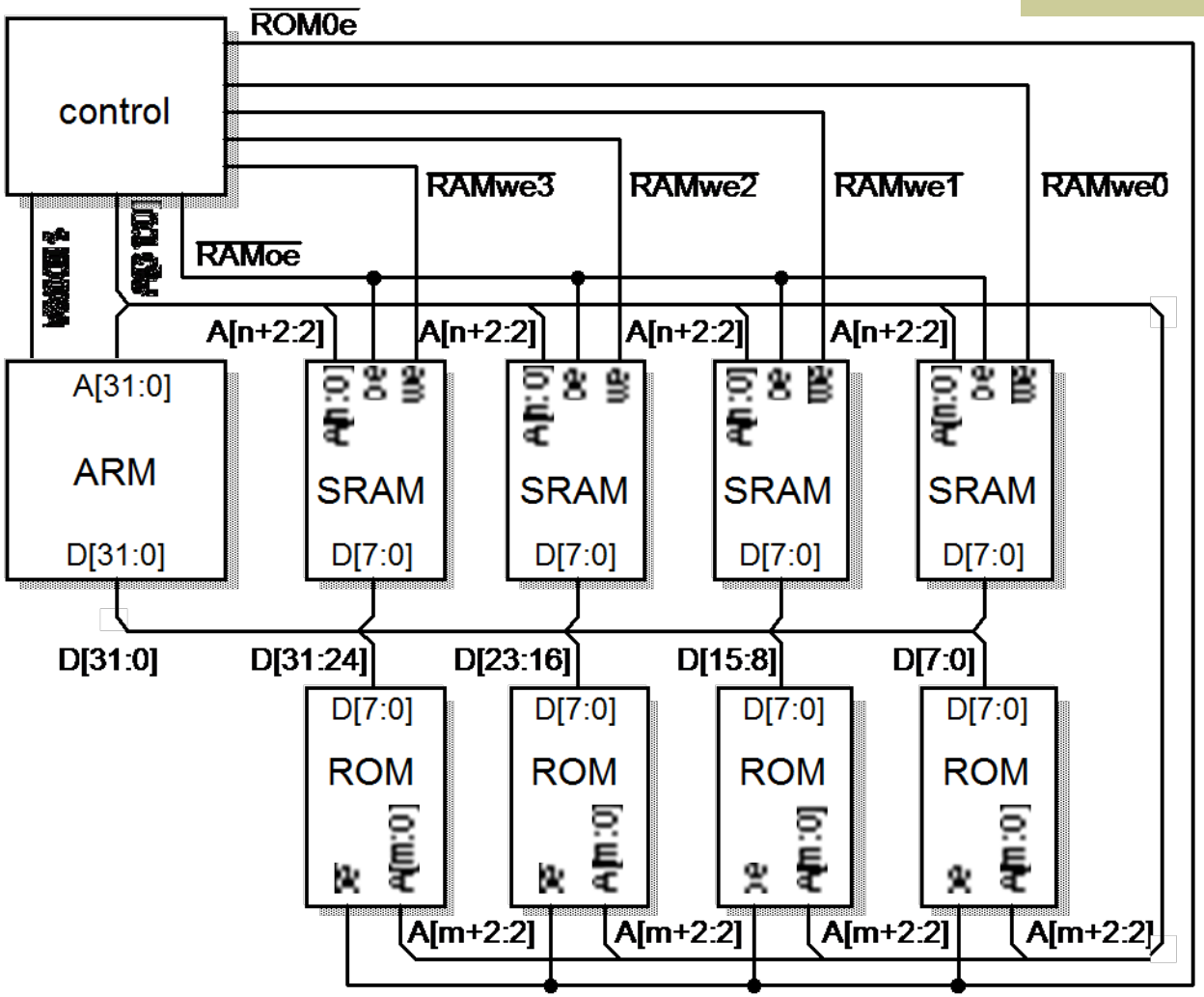
```
ENTRY (main)
MEMORY {
   ram: ORIGIN=0x00400000, LENGTH=64M
   rom: ORIGIN=0x60000000, LENGTH=16M }
SECTIONS {
   data : {                    /* Initialized data */
         _DataStart = .;
         *(.data)
         _DataEnd = .;
   } >ram
```

Linker Script (contd)

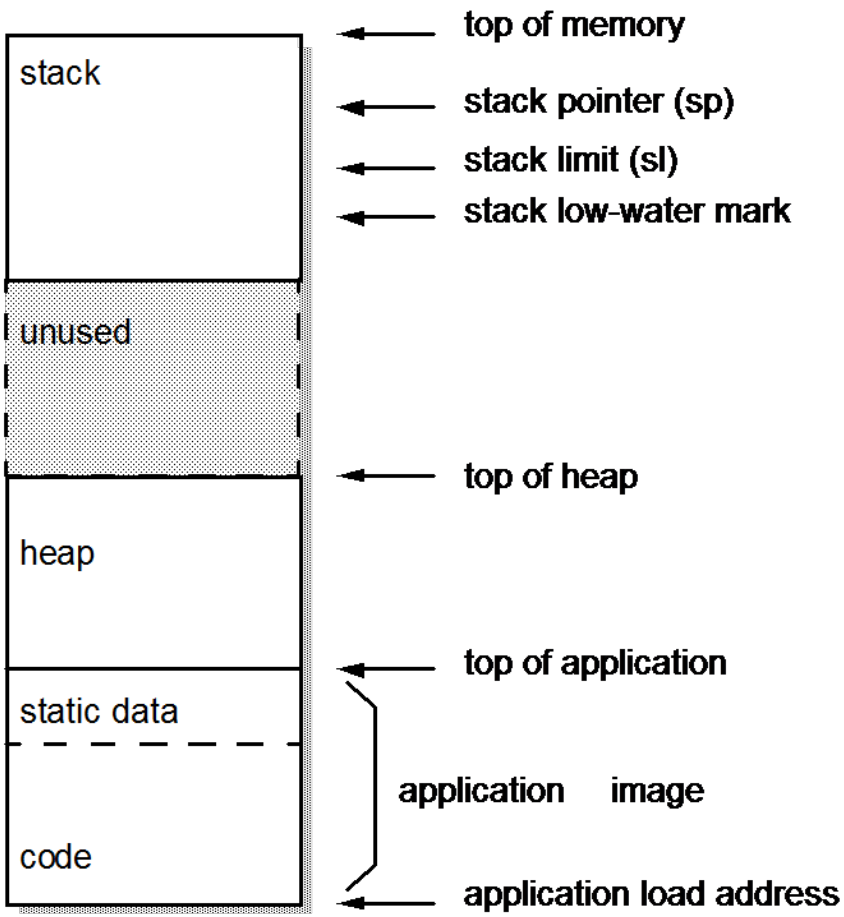
```
    bss: {                  /* Uninitialized data */
        _BssStart = .;
        *(.bss)
        _BssEnd = .;
    }  >ram
    text : {
        *(.text)
    }  >ram
}
```

Further information at http://www.gnu.org

A basic ARM memory system



The standard ARM C program address space model



Debug Monitors

- In some cases, debug monitors are the first code executed when board powers up
- Example: RedBoot in the Arcom board
  - (RedHat's Embedded Debug and Bootstrap)
- Used to:
  - Download software
  - Perform basic memory operations
  - Manage nonvolatile memory

RedBoot

- Does all the above things, but …
- RedBoot also contains startup code
- Programs downloaded to run in RAM via RedBoot
  - do not need to be linked with startup code and
  - should be linked but not located
- After hardware initialization
  - RedBoot prompts for user input

RedBoot

- Supports
  - Commands to load software
  - Dump memory
  - Various other tasks
    - Check at http://ecos.sourceware.org/redboot
- Reference book
  - *Embedded Software Development with eCos*, Anthony Massa, Prentice Hall PTR

Building Your First Program

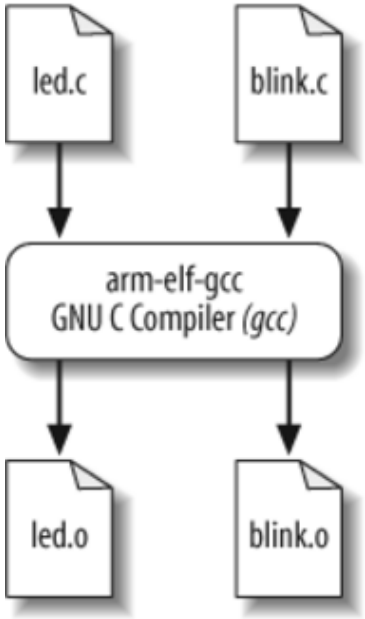
- Arcom Board:
  - GNU tools installation in Appendix B
  - Enter commands into a command shell
- Two source modules: led.c and blink.c
- Compiling
  - **`arm-elf-gcc [`*`options`*`] `*`file`*** …
    - **`-g`**: to generate debugging info in default format
    - **`-c`**: to compile and assemble but not link
    - **`-Wall`**: to enable most warning messages
    - **`-I../include`**: to look in the dir include for header files

Compiling

- Commands for compiling C source files

```
#arm-elf-gcc -g -c -Wall -I../include led.c
#arm-elf-gcc -g -c -Wall -I../include blink.c
```

- Additional information
  - http://gcc.gnu.org

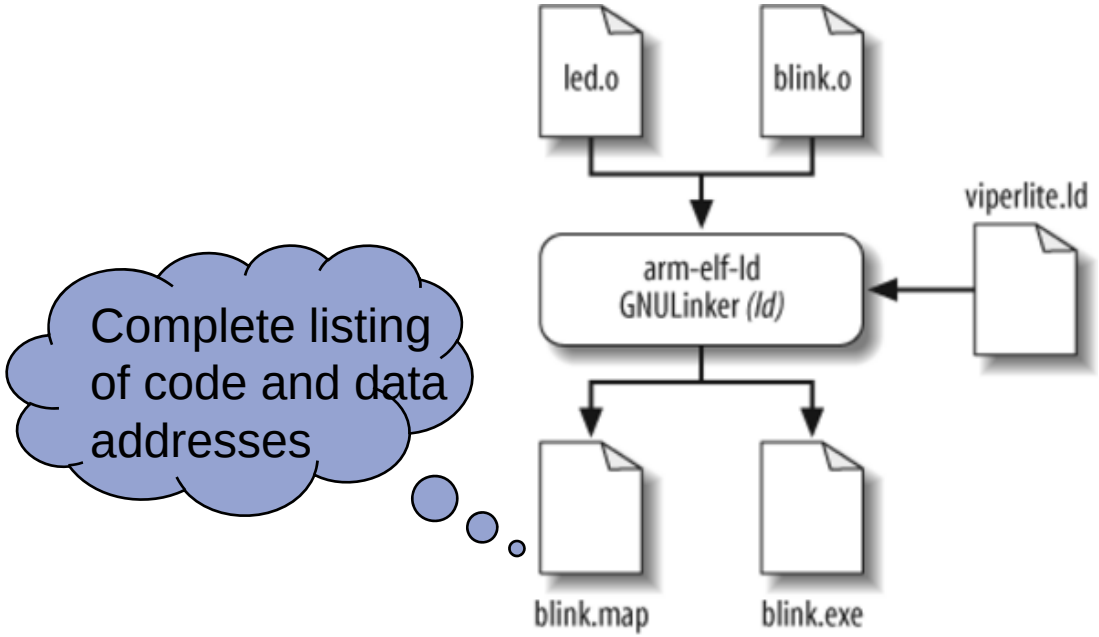


Linking and Locating

- GNU linker (ld) performs locating also
- Linker script: viperlite.ld
- **`arm-elf-ld [options] file …`**
  - **`-Map blink.map`**: to generate a map file
  - **`-T viperlite.ld`**: to read the linker script
  - **`-N`**: to set the text and data sections to be readable and writable
  - **`-o blink.exe`**: to set output filename

Linking and Locating

- **`arm-elf-ld –Map blink.map –T viperlite.ld –N –o blink.exe led.o blink.o`**



Format the Output File

Tools from **binutils**

- Stripping the binary image
  - **arm-elf-strip [*options*] *input-file* … [-o *output-file*]**
  - **# arm-elf-strip –remove-section=.comment blinkdbg.exe –o blink.exe**
    - Removes the **.comment** section from blinkdbg.exe (with debug information)
    - Outputs the stripped binary image blink.exe
- Transforming it into downloadable format
  - **arm-elf-objcopy [options] input-file [output-file]**
  - **# arm-elf-objcopy –O ihex blink.exe blink.hex**
    - Intel Hex Format: an ASCII format devised by Intel for storing and downloading binary images

No output filename → overwrite original files

Other Tools from binutils

- **size**
  - Lists section sizes and total size for object file

```
# arm-elf-size blink.exe
text    data bss  dec  hex  filename
328        0   0  328  148  blink.exe
```

- File size of blink.exe is 3 KB
  - Much larger! Why?
    - Includes debug information!
- Additional information: http://www.gnu.org

Downloading and Debugging

Downloading and Debugging

- Executable Binary Image (stored as a file on host computer)
- Must be downloaded into some memory on target board
- Executed from the memory
- Tools needed to
  - Set breakpoints in program
  - Observe program execution

Debug Monitors

- Debug Monitor, also called ROM Monitor
  - A small program in nonvolatile memory
  - Facilitates development tasks
    - Hardware initialization / configuration
    - Download and run software in RAM
    - Debug program
    - Command-Line Interface (CLI)
    - Peeking (reading) / Poking (writing) / Comparing / Displaying memory and processor registers
  - Also, in production units
    - Upgrade firmware for new features
    - Fix bugs after deployment

RedBoot

- In Arcom board, Redboot
  - Is in bootloader flash
  - Uses COM1 for command-line interface
    - Need terminal program (minicom in Linux or HyperTerminal in Windows) on the host computer
    - Baud rate: 115200
    - Data bits: 8
    - Parity: None
    - Stop bits: 1
    - Flow control: None

RedBoot

Further information:
http://ecos.sourceware.org

```
Ethernet eth0: MAC address 00:80:12:1c:89:b6
No IP info for device!

RedBoot(tm) bootstrap and debug environment [ROM]
Non-certified release, version W468 V3I7 - built 10:11:20, Mar 15
2006

Platform: VIPER (XScale PXA255)
Copyright (C) 2000, 2001, 2002, 2003, 2004 Red Hat, Inc.

RAM: 0x00000000-0x04000000, [0x00400000-0x03fd1000] available

FLASH: base 0x60000000, size 0x02000000, 256 blocks of 0x00020000
bytes each.
== Executing boot script in 1.000 seconds - enter ^C to abort
^C
RedBoot>
```

CTRL-C: Stop boot script from loading Linux

Downloading with RedBoot

- RedBoot can load and run ELF files
  - So, we can run **`blink.exe`**
- To transfer files using **`xmodem`** protocol
  - **`RedBoot> load –m xmodem`**
  - RedBoot prompts **`C`** and waits for file transfer
  - From Windows HyperTerminal Menu
    - Select Transfer → Send File
    - Select blink.exe for transfer
  - File will be transferred!

Downloading with RedBoot

- After transfer completes:

```
Entry point: 0x00400110, address range:
0x0x00000024-0x0040014c
xyzModem - CRC mode,
24(SOH)/0(STX)/0(CAN) packets, 2 retries
```

- Compare with blink.map:

```
Name        Origin          Length
.text       0x004000b0      0x9c    blink.o
            0x00400110              main
```

File size of blink.o

-

When in ROM …

Device programmer

- A computer system
- Several IC sockets on top of different shapes and sizes
- Capable of programming memory devices of all sorts
- Connected to the same network as host computer

When in ROM …

- Transfer binary image to device programmer
- Place memory chip on an appropriately sized and shaped socket on device programmer
- Select device-type from on-screen menu
- Start programming memory
- Takes a few seconds ~ several minutes, depending on
  - binary image size and
  - type of memory device

When in ROM …

- After programming ROM
  - Insert into socket on board
  - (Power must be off!)
- As soon as power is applied, processor fetches and executes the code in ROM
- Where is the code (first instruction)?
  - Each processor has its own rules
  - E.g., for ARM: 0x00000000
  - Called RESET ADDRESS, RESET CODE
    - In Arcom board, reset code is part of RedBoot

Program Not Working?

- Check RESET CODE
- Check target processor's RESET RULES
- All satisfied?
- Hint:
  - Turn on LED just after reset code has completed

Managing ROM in RedBoot

- RedBoot can be used to manage flash filesystems called Flash Image System (FIS)
  - Create, write, erase locations of flash based on “filenames”
  - To check what is in FIS:

```
RedBoot> fis list

Name             FLASH addr Mem addr   Length Entry point
FIS directory    0x00000000 0x00000000 0x0001F000 0x00000000
RedBoot config   0x0001F000 0x00000000 0x00001000 0x00000000
filesystem       0x00020000 0x00000000 0x01FE0000 0x00000000
```

Running from flash

- Since flash is ROM, a debugger cannot be used if the program runs from flash
  - A debugger needs to insert software interrupts when single-stepping or executing to a breakpoint
- Flash can be used if we are sure the software works and debugger is not needed

Running from flash

- Workarounds in some processors:
  - TRACE instruction: executes a single instruction and then automatically vectors to an interrupt
  - Breakpoint register: gets you back to the debug monitor

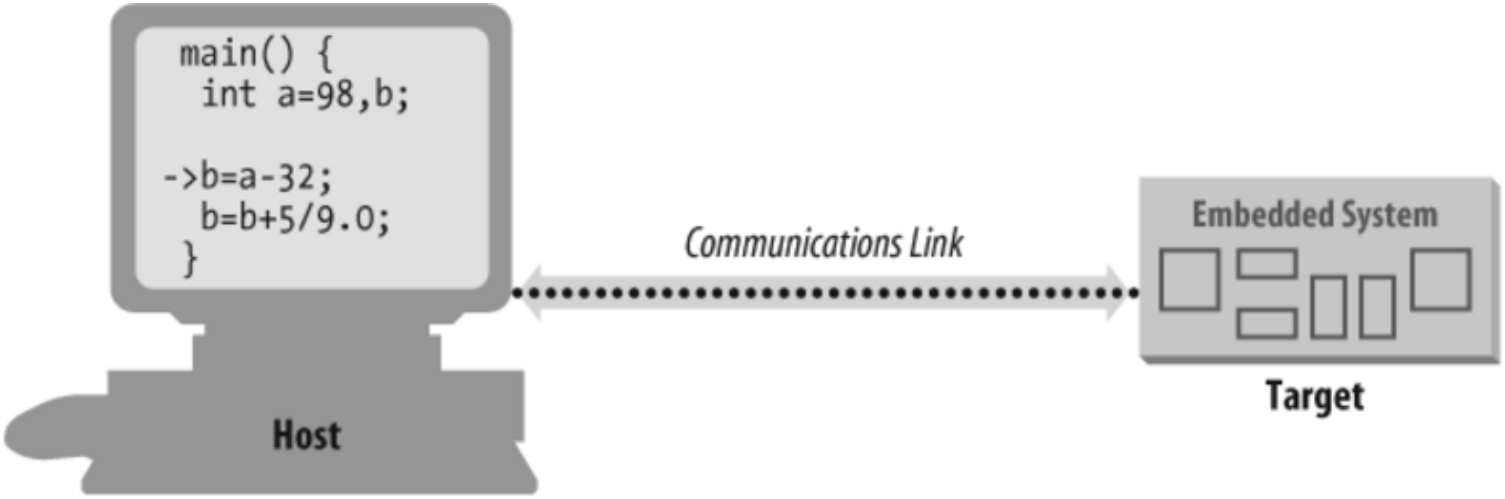
Debugging Tip

- Use LED as an indicator of SUCCESS or FAILURE!
- Slowly walk the LED enable code through the larger program
- Begin with LED enable code at RESET ADDRESS
- If LED turns on, edit program, move LED enable code to another execution breakpoint, rebuild, and test again

Remote Debugging

- To download, execute, and debug embedded software (cross-platform debugging)
- Two pieces of software:
  - Frontend or Remote Debugger
    - run by host computer
    - GUI: source code, register contents, info, …
  - Backend or Debug Monitor
    - run by target processor from ROM
    - Low-level control of processor

A remote debugging session



Remote Debugger

Debug Monitor

Debug Monitor

- Automatically started when processor is reset
- Monitors communication link to host computer
- Responds to requests from remote debugger
- Examples:
  - read register *x*
  - modify register *y*
  - read *n* bytes of memory starting at *address z*
  - modify data at *address a*

Debug Monitor

- Combines sequences of low-level commands
- Accomplish high-level debugging tasks
  - Downloading a program
  - Single-stepping through a program
  - Setting breakpoints

GNU Debugger (gdb)

- Originally, a native debugger
- Later, cross-platform debugging added
- **Frontend**: Build a version of gdb frontend to run on a host PC for a target processor. Also called remote debugger.
- **Backend**: Source code for a compatible debug monitor is included in gdb package, must be ported to target platform (not an easy task!) Also called gdb stub or debug monitor.
- About gdb: http://sources.redhat.com/gdb

GNU Debugger (gdb)

- Advantages
  - Low cost
  - Easy to use
- Disadvantages
  - Cannot debug startup code
  - Code must execute from RAM
  - Need a communication channel between host and target

Debugging on the Arcom Board

- gdb communication is byte-oriented and over
  - Serial port, or
  - TCP/IP port
- RedBoot supports gdb debug sessions over either of the two ports
- Power on Arcom board
- Halt RedBoot script by pressing Ctrl-C
- Invoke gdb

```
# arm-elf-gdb blink.exe
```

Debugging on the Arcom Board

Gdb outputs a message as follows:

```
GNU gdb 6.3
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public
   License, and you are welcome to change it and/or
   distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show
   warranty" for details.
This GDB was configured as "--host=i686-pc-cygwin --
   target=arm-elf"...
(gdb)
```

GDB Prompt

Debugging on the Arcom Board

- Connect to the Arcom board

```
(gdb) target remote /dev/ttyS0
```

-- Assuming target board is connected to COM1

- Message:

```
Remote debugging using /dev/ttyS0
```

- Now gdb CLI is connected to gdb stub

Debugging on the Arcom Board

- Download blink.exe program onto target

```
(gdb) load blink.exe
```

- Message:

```
Loading section data, size 0x4 lma 0x400000
Loading section text, size 0x148 lma 0x400004
Start address 0x400110, load size 332
Transfer rate: 2656 bits in <1 sec, 166 bytes/write.
```

- Ready to start debugging!

Debugging on the Arcom Board

- Setting a breakpoint

```
(gdb) b ledToggle
Breakpoint 1 at 0x400070: file led.c, line 66
```

(gdb) breakpoint ledToggle
(gdb) break ledToggle
(gdb) br ledToggle
(gdb) b ledToggle

- Getting breakpoint information

```
(gdb) info b
Num Type        Disp Enb  Address     What
1   breakpoint  keep y    0x400070    in ledToggle at led.c:66
```

- Continue

```
(gdb) c
```

Debugging on the Arcom Board

- At breakpoint

```
Breakpoint 1, ledToggle() at led.c:66
66    if(GPIO_0_LEVEL_REG & LED_GREEN)
```

- List command

```
(gdb) l
```

- Repeat command

```
(gbd)
```

Debugging on the Arcom Board

- Check symbol values

```
(gdb) print /x gChapter
$1 = 0x5
```

- Change symbol values

```
(gdb) p/x gChapter=12
$2 = 0xc
```

Debugging Tip

- A binary image might not have debug symbols (var/func ←→ addresses)
  - For example: vendor given library object files
- Problem: How to trace the value of symbols?
- Solution: Use **`blink.map`**
- Example: to lookup the value of **`gChapter`**
  - Search in **`blink.map`** for **`gChapter`** address

```
.data     0x00400000  0x4   blink.o
          0x00400000        gChapter
```

```
(gdb) x/d 0x400000
0x400000 <gChapter>: 12
```

(x = examine, /d = decimal)

Debugging on the Arcom Board

- Single-stepping

```
(gdb) n
69    GPIO_SET_REG = LED_GREEN;
```

- Backtracing

```
(gdb) bt
#0    ledToggle() at led.c:66
#1    0x00400140 in main() at blink.c:75
```

Debugging on the Arcom Board

- View processor registers

```
(gdb) info registers
```

- Print value of a specific register (PC here)

```
(gdb) p/x $pc
```

- Delete breakpoint

```
(gdb) d
```

Emulators

ICE (In-Circuit Emulator)

- Examines processor state
- Emulates the processor
- Has its own copy of processor, RAM, ROM, and embedded software
- More expensive than target hardware
- Powerful debugging tool
  - Hardware breakpoints
  - Real-time tracing

ICE debugging

Hardware Breakpoints

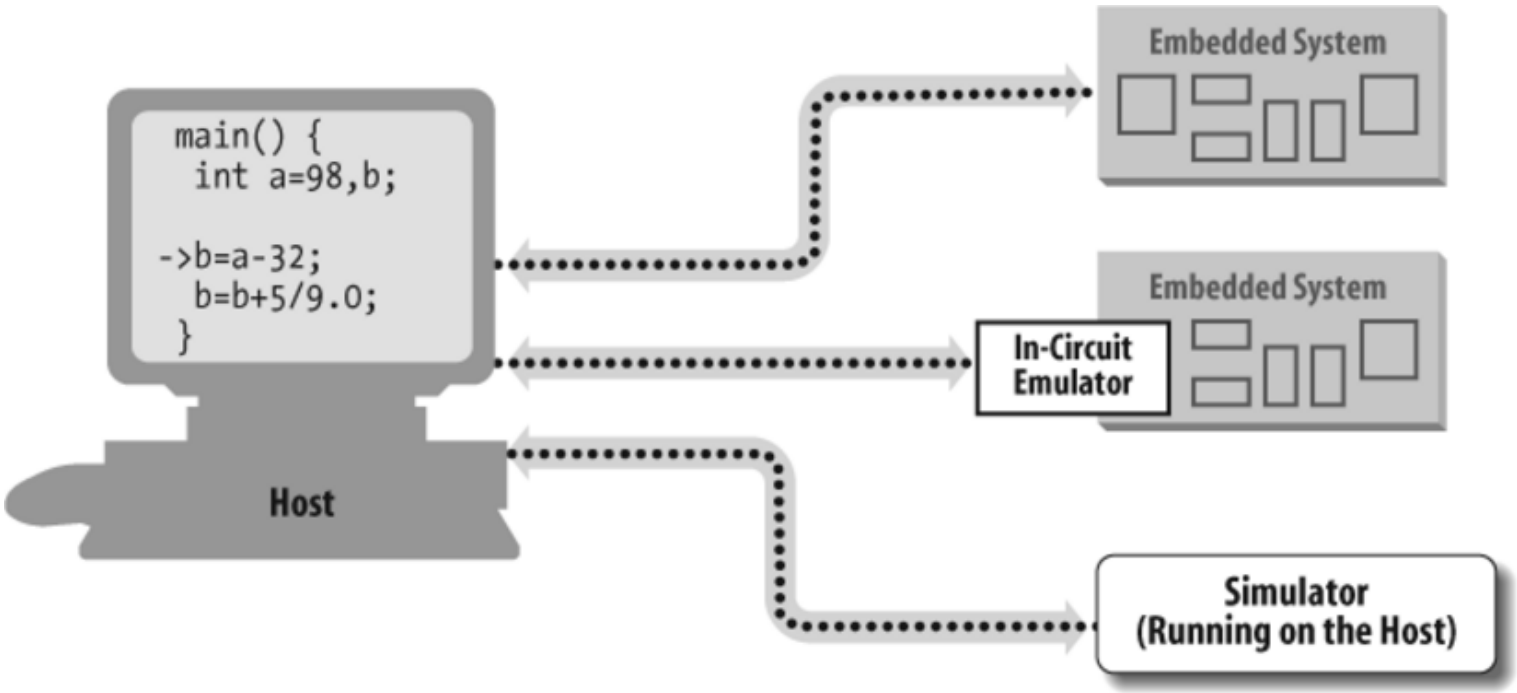
- Example
  - "**`address bus = 0x2034FF00 and data bus = 0x20310000`**"
- Not only instruction fetches (as in remote debugging)
  - Memory and I/O reads and writes
  - Interrupts

ICE debugging

Real-Time Tracing

- Large block of special-purpose RAM
  - Stores information about each processor cycle executed
- To see what happened in what order
  - “Did the timer interrupt occur before or after the variable bar became 12?”

Simulators


main() {
int a=98,b;
->b=a-32;
b=b+5/9.0;
}
Host
Embedded System
Embedded System
In-Circuit Emulator
Simulator (Running on the Host)

Debugging Tip

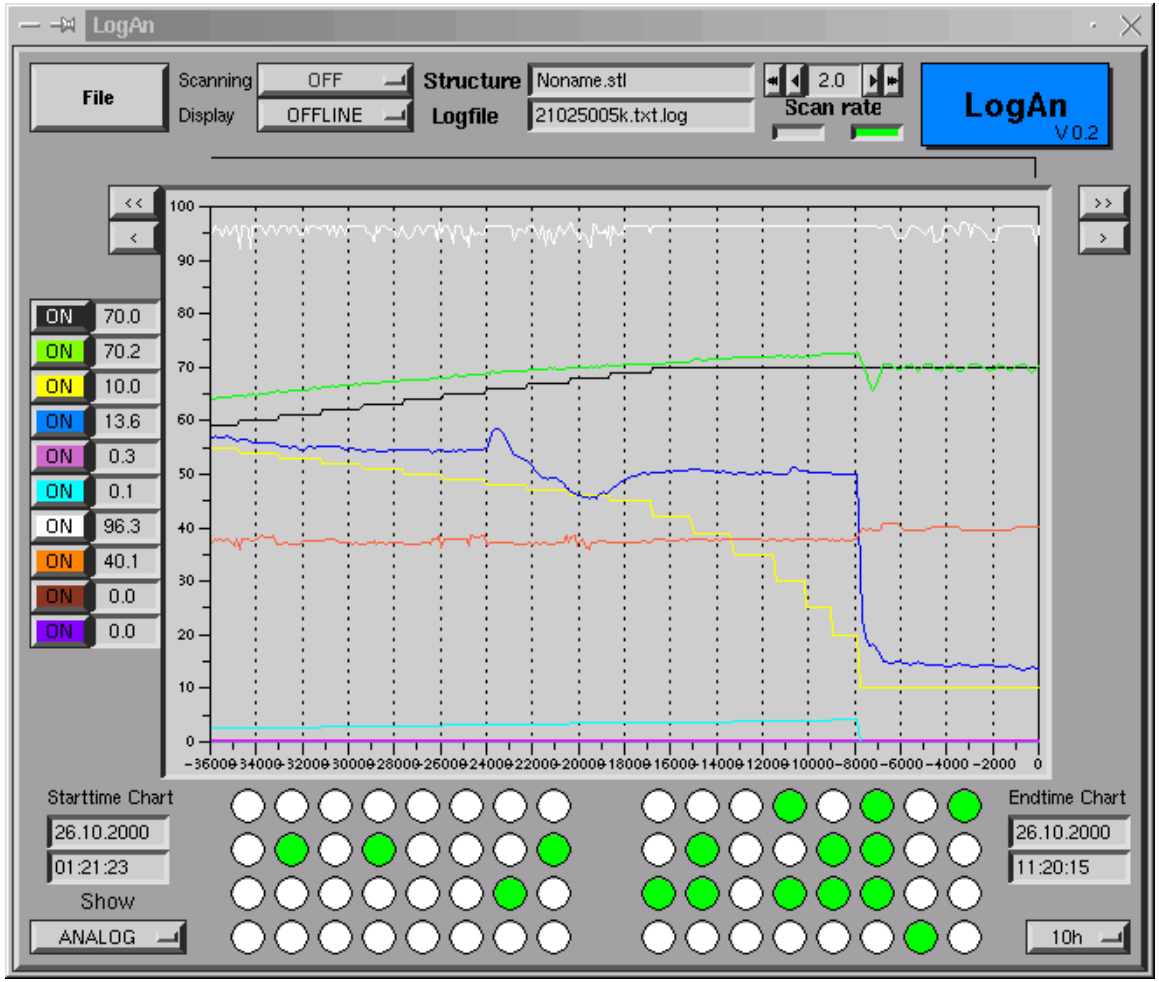
- Processor behaving differently?
- Run the same software in simulator!
  - OK → HW problem
  - Behaves differently → You are wrong!

Logical Analyzer

Logical Analyzer

- Views signals external to processor
- Cannot control software execution flow
- Useful only with debuggers
- Troubleshoots digital hardware
- Dozens or hundreds of inputs
  - Detect each signal is high or low
- "Display the values of input signals 1 through 10, but don't start recording what happens until inputs 2 and 5 are both zero at the same time."

Logical Analyzer Display

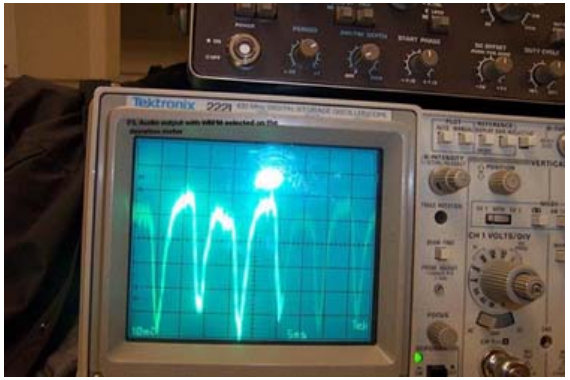


Debugging Tip

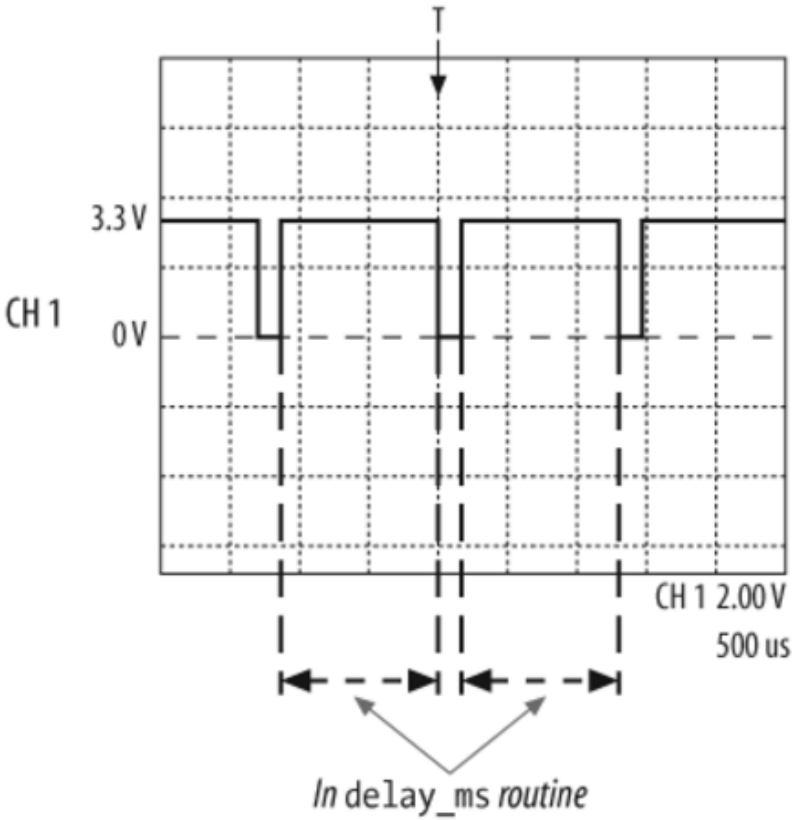
- Coordinate signal observation with software execution
- Example: processor / peripheral interaction
- Before interaction
  - Add an output statement in software: cause a unique logic pattern on processor pins (e.g. spare I/O pin: zero → one)
- Set logic analyzer to trigger on that pattern
- LA records everything after that!

Oscilloscope

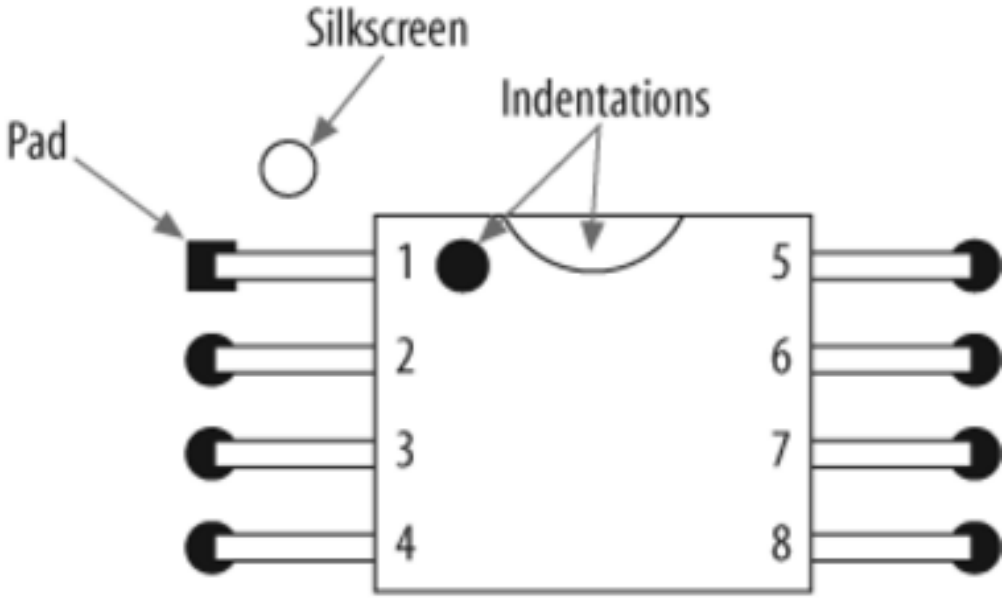
- Examine any hardware signal (analog or digital)
- Quick observation of voltages on pins
- # inputs ≈ 4
- Not useful as a software debugging tool



Using Oscilloscope



Finding Pin 1



Lint

- Static checking of source code for
  - Portability problems
  - Common coding syntax errors
    - Ignored return values
    - Type inconsistencies
- More careful checking than compilers
- Can trim the output of lint tool using options
- Reference
  - “Introduction to Lint” from Embedded Systems Programming (http://www.embedded.com)

Lint

- Open source lint
  - Splint
    - http://www.splint.org

Version Control

- Required when software is large or there are multiple developers
- Storage of source code in a repository
- Updated as the project progresses
- Logging, file comparisons, tagging releases, tracking bug fixes, codes updates for new features
- All files associated with a project
  - Programs, tools, documentation

Version Control

- Concurrent Versions System (CVS) (http://ximbiot.com/cvs/cvshome)
  - Combine changes by different people to a single file
  - Ref. Book: *Essential CVS* by Jennifer Vesperman (O'Reilly)
- Subversion (http://subversion.tigris.org)
  - Ref. Book: *Version Control with Subversion* by B. Collins-Sussman, B.W. Fitzpatrick, C. M. Pilato (O'Reilly)
- Revision Control System (http://www.gnu.org/software/rcs)
  - Free GNU project

Which tools to use?

- Oscilloscopes, Logic Analyzers
  - To debug hardware problems
- Simulators
  - To test software before hardware is available
- Lint and version control software
  - Throughout entire project