From Source to Execution

Assembling, Compiling, Debugging, and Programming

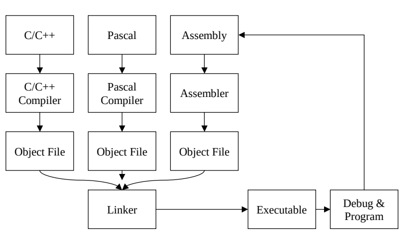
Babak Kia
Adjunct Professor
Boston University
College of Engineering
Email: bkia -at- bu.edu

Topics Discussed

- **Assembler**
- **Compiler**
- **Object Files**
- **Extended Linker Format (ELF)**
- **Debuggers & Debugging Techniques**
- **Simulators**
- **Emulators**
- **Programmers**
- **Logic Analyzers**
- **Digital Storage Scope**

From Source to Execution

- **The purpose of Assemblers & Compilers are to translate code into machine executable binary.**



Assembler

- Assembler takes instructions written in assembly language, and translates them into the binary code which a target processor can then execute
- Assemblers are relatively simple, there is limited macro support, and no optimization
- Advantages
  - Just about as optimized as it gets
- Disadvantages
  - You can't simply reassemble code for a new target
  - It can get very complex very quickly

Compiler

- The role of a Compiler, much like an Assembler, is to translate source code to run on a target processor
- Advantages
  - Code is usable across different targets. It is a simple matter of recompiling
  - The complexity of writing assembly code is greatly reduced
  - Good software techniques can be employed with a high level language such as C or C++
- Disadvantages
  - Generated object code is generally larger and executes slower than an assembly version

Assembler versus Compiler

- Ultimately, assembly language is as close you can get to running efficient code on hardware
- However, time-to-market pressure, good software design techniques, code re-use, and optimizing compilers restrict usage of assembly code
- Although for an 8-bit processor you primarily use assembly code, for a 32-bit processor, assembly is used mainly to tune code
- This is achieved by rewriting some of the key loops in assembly

Compiler objectives

- Compilers usually optimize for one of the following objectives
  - Speed
    - Execution speed of a program can be a critical factor in some applications
    - Produces larger code, and takes longer to compile
  - Size
    - Size too can be a critical factor, specially for systems with limited resources
  - Debug information
    - Debug info is of great use during debug, but of little value for a final production image

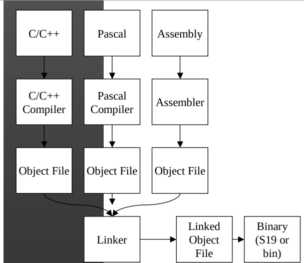
Cross Compiling

- It makes little sense to run the compiler on the target processor
  - A Host PC is usually more versatile and powerful than a target Processor
  - Disadvantage is that you now need a mechanism for loading the object code onto the target processor
  - You also need a means of debugging the code
- Cross-Compiling is the process of compiling and linking code on a Host PC, in order to create an object file which will run on a target Processor.

Compiling on PC versus an Embedded processor

- Preparing code on a PC is relatively simple
  - The Compiler already knows a lot of information about your target and hides that information from you.
- Preparing code for an Embedded System is more involved
  - The compiler needs to be configured with specific information about the target system, such as available memory, Stack address, etc.

Object Files



- An object code file contains executable code, but isn't executable by itself
- Object code can be thought of as a very large data structure
- Usually falls into one of two standard formats:
  - COFF – Common Object File Format
  - ELF – Extended Linker Format

What goes into an Object File

- An object file contains the following information:
  - Header Information – Information about the file such as size of code, date, etc.
  - Object Code – Binary instructions created by the compiler or the assembler
  - Relocation Information – Used by the linker to change addresses in the object code
  - Symbols – Global symbols defined for this module
  - Debug Information – Links to the source code, line number information, C data structures, etc.

Linker

- It is the linker's job to take one or more object files and combine them together to form a single file
- The linker merges the code and the data sections of the object files together with resolving the symbol information in order to create a final and relocatable object file
- On a PC this would be the final step prior to running the created executable. The Operating System is aware of where to load your program.
- On an embedded system however, a final step is required which assigns absolute memory addresses to the code. For instance, the linker needs to know where stack starts, where heap starts, etc.
- This step may be an inherent part of the Linker, or a separate script file called the Linker Script file (ld).

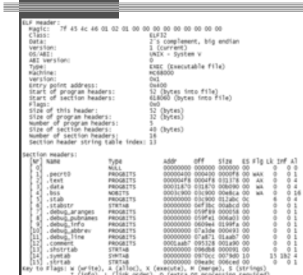
C Runtime (crt.0)

- A PC is adept at preparing memory, setting up stack and other setup procedures prior to loading code
- On an Embedded System, we need to provide a start-up code
- This is usually (and automatically) provided in a file called the C runtime or crt0 file
- The crt0 file is linked and therefore automatically performs the following tasks
  - Copies initialized data from ROM to RAM
  - Zeros out the un-initialized data in RAM
  - Initializes the Stack Pointer
  - Initializes the Heap
  - Calls main()

Extended Linker Format (ELF)

- An ELF file can be one of three types
  - Executable – Can be loaded into memory and executed
  - Relocatable – Prior to loading, the location addresses need to be processed
  - Shared Object – Or shared library, contains the runnable code, and symbol information for the linker

Extended Linker Format (ELF)

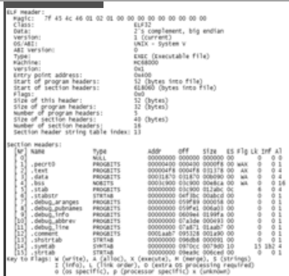


```
ELF Header:
  Magic:   7f 45 4c 46 01 02 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, big endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           MC68000
  Version:                           0x1
  Entry point address:               0x400
  Start of program headers:          52 (bytes into file)
  Start of section headers:          618060 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         5
  Size of section headers:           40 (bytes)
  Number of section headers:         16
  Section header string table index: 13

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .pecrt0           PROGBITS        00000400 000400 0000f8 00 WAX  0   0  1
  [ 2] .text             PROGBITS        000004f8 0004f8 031378 00  AX  0   0  4
  [ 3] .data             PROGBITS        00031870 031870 00b090 00  WA  0   0  4
  [ 4] .bss              NOBITS          0003c900 03c900 00e8ca 00  WA  0   0 16
  [ 5] .stab             PROGBITS        00000000 03c900 012abc 0c      6   0  4
  [ 6] .stabstr          STRTAB          00000000 04f3bc 00abcd 00      0   0  1
  [ 7] .debug_aranges    PROGBITS        00000000 059f89 000058 00      0   0  1
  [ 8] .debug_pubnames   PROGBITS        00000000 059fe1 006a03 00      0   0  1
  [ 9] .debug_info       PROGBITS        00000000 0609e4 0199fa 00      0   0  1
  [10] .debug_abbrev     PROGBITS        00000000 07a3de 000493 00      0   0  1
  [11] .debug_line       PROGBITS        00000000 07a871 01aab7 00      0   0  1
  [12] .comment          PROGBITS        0001aab7 095328 001a90 00      0   0  1
  [13] .shstrtab         STRTAB          00000000 096db8 000091 00      0   0  1
  [14] .symtab           SYMTAB          00000000 0970cc 0079d0 10     15 182  4
  [15] .strtab           STRTAB          00000000 09ea9c 006ced 00      0   0  1
Key to Flags: W (write), A (alloc), X (execute), M (merge), S (strings)
              I (info), L (link order), O (extra OS processing required)
              o (OS specific), p (processor specific) x (unknown)
```

- ELF files can have many sections, but they have three major sections
- .text section – contains all of the code segments
- .data section – contains initialized variables and their values
- .bss section – contains uninitialized global variables
- Other sections are a symbol table, and links to the actual source code

Extended Linker Format (ELF)



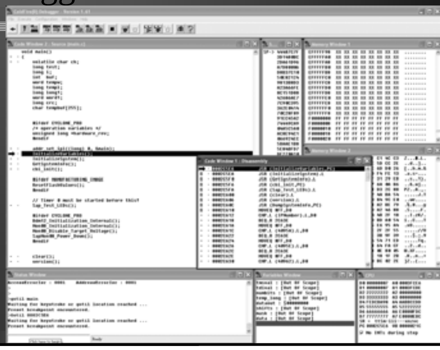
- PROGBITS are program content that contain debug info, code, and data
- NOBITS are like PROGBITS, but no space is allocated since it is used for BSS section during load time
- STRTAB is the string table
- SYMTAB is the symbol table

Debugging

- Compiling the code and getting it into a binary file is only the first step of creating a final product
- Once you have a binary code, you will need a means of loading it onto the target hardware and debugging it
- Traditionally an Embedded System designer's debug options have been limited because chip manufacturers paid little attention to the value of on-chip debug resources
- Today however, most modern processors have a debug port, either in the form of a Background Debug Mode (BDM) port, or a JTAG port, which allow external access to the chip's internal registers and memory

Debugger – P&E's ICDHCS08

Debugger - P&E's ICDCFZ



Debugger Characteristics

- Good debuggers share common characteristics:
  - Show internal CPU register and resources
  - Show and allow modification of memory content
  - Allow the developer to set source breakpoints
  - Enable task and thread aware breakpoints
  - Allow the developer to set memory access breakpoints
  - Provide a stack trace
  - Provide an execution trace
  - Evaluate and modify variables and complex data structures easily
  - Show high-level source code as well as low level assembly
  - Seamlessly take care of cache and interrupts

Debugging Techniques

- In spite of very complex and expensive debugger environments, some of the MOST effective debugging techniques come for free!
- Very useful, though unconventional debug techniques include
  - Using (toggling) an LED light to indicate execution milestones
  - Using printf statements to provide code insight

Debugger Hardware

- Debuggers are simply software that run on a host PC
- To actually debug target hardware, you will need a debug interface
- Debug interfaces come in many shapes and sizes
  - As simple as a serial port connection to a host PC, which interacts with a "Monitor ROM" resident on target hardware
  - A Parallel port or USB cable
  - A feature rich development and debug interface capable of stand-alone operation (for programming)
  - Ultimately they all "speak" the target processor's debug language, be it BDM or otherwise, and provide that information back to the host PC

Debugging Pitfalls

- Interrupts
  - Interrupts are always a source of bemusement to the person who is debugging a system
  - Their inherently asynchronous nature makes debugging a system which has interrupts turned on very difficult
  - Turning interrupts off certainly helps, but what if you are trying to debug an interrupt itself?
  - This is where more complex debug equipment such as emulators come to the rescue

Debugging Pitfalls

- Cache
  - Cache is of great value in microprocessor systems because it stores local copies of data on a high speed, low latency memory
  - However this very mechanism hampers debugging because modified data is not always updated on main memory
  - One solution is to simply turn cache off
  - Another solution is to configure the system cache for a write-through operation (as opposed to other configurations such as write-back)
  - A third option, not usually used, is to invalidate cache and force it to write back modified data, however, this takes time

Emulators

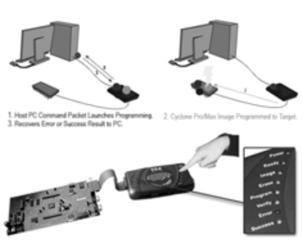
- Emulators are glorified (albeit very useful) debuggers
- An emulator actually takes the place of a target processor, and is itself an embedded system, complete with the target processor, ROM, RAM, etc.
- They provide the designer with tools that simple debugger environments cannot, such as
  - Bus-state analyzers
  - Timing information
  - Real-time execution trace
  - Complex breakpoint capabilities
  - Multiprocessor support
- Ultimately, they are the most powerful tool in the designer's debug arsenal

Simulators

- Simulators are also a great tool for debugging target hardware, even though it is purely software which runs on a host PC
- A simulator is useful particularly in the beginning phase of a development cycle, when target hardware, or even the silicon are not available
- It is an indispensable tool for developing algorithms and verifying the logical functionality of programs

Programmers

- Ultimately, once the development and debug of a system is done, the code needs to be programmed into the target board
- Programmers come in many shapes and sizes, ranging from $100 - $100,000
- Perhaps the simplest way of programming the target is via an in-circuit programmer, using the same debug port to program a non-volatile memory such as flash

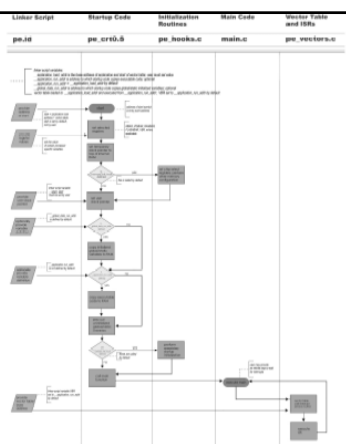


Logic Analyzers

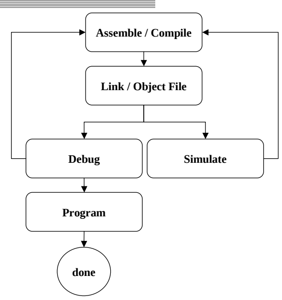
- Logic Analyzers are also an indispensable tool for debugging hardware.
- They are particularly useful in identifying and debugging the interaction between the target processor and other chipsets onboard
- Since logic analyzers do not have access to a microprocessor's internal resources, their primary utility lies in debugging external events
- However, the drawback is that it is intended almost exclusively for debugging digital hardware

Digital Storage Scopes

- Digital Storage Scopes, and oscilloscopes are the final defense when it comes to debugging hardware issues relating to
  - Timing
  - Signal integrity & noise issues
- The drawback to Storage Scopes and oscilloscopes are that they have limited channels and can be very expensive



Summary



Assemble / Compile
Link / Object File
Debug
Simulate
Program
done