3: Embedded System Software Development

E.S. Dev. Scenario 1

- use existing h/w (COTS)
  - dev. board, EVB, generic PC
    - ex: Arduino, Raspberry Pi, NI RIO
  - removes h/w problems
  - more s/w support available
    - monitor, libraries, examples
  - may be overkill for a given app.
3: Embedded System Software Development

E.S. Dev. Scenario 2

- h/w co-design
  - tailored to app. requirements
  - adds delay of bringing up h/w
  - additional complexity of debugging h/w
  - typically requires use of emulation techniques & support equipment
  - more cost-effective

Software Dev. Process

- modern PC-based approach "hides" many details
  - ok for PC environment, not for E.S.!
- issues:
  - cross-compilation / assembly
  - runtime libraries
  - linking loader (memory map)
- diagram...
3: Embedded System Software Development

Compilation Process

```
<table>
<thead>
<tr>
<th>Source Code</th>
<th>Pre-processor</th>
<th>Compiler</th>
<th>Assembler</th>
<th>Linker</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Listing</strong></td>
<td><strong>Listing</strong></td>
<td><strong>Listing</strong></td>
<td><strong>Map</strong></td>
</tr>
<tr>
<td>includes,</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>defines</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>options</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>libraries*,</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>other modules</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

*including RTE

R.T.E. & Libraries

- HLLs provide only a subset of desired functions
  - i.e.: convert an IF-THEN-ELSE construct to processor-specific machine code
- what about run-time environment & non-intrinsic functions?
  - ex: who calls `main()`?
  - ans: support libraries
    - or `startcf.c` in CodeWarrior Coldfire projects
3: Embedded System Software Development

Run-time Library

- provide additional functions & services beyond HLL
  - CPU dependent - simply manipulate data structures or perform math
  - I/O dependent - must match E.S. actual h/w
  - system services - allocate memory, task control, etc.
  - entry/exit - performs setup/cleanup functions before & after running main()
    - ex: stack, global initializers, files

C Development Steps

1. pre-processor
2. compiler
3. assembler
4. linker
5. loader
1: Pre-processor

- prepares source code for compilation
  - include files, constant defs, type defs, library functions, conditional compilation

- commands:
  a) \#define id [string]
     - "replacement", used to define constants
     - typically all caps
     - ex: \#define TRUE 1
  
  b) \#define MACRO()
     - adds expression with parameters
     - ex: \#define SQR(x) (x * x)
     - caution: use parentheses as expression guards!
     - ex: \#define PI 3+1/7
         : twopi = 2 * PI;
     - above SQR macro needs fixed!
     - a neat trick:
        \#define FOREVER while(1)
     - also: \#undef id

Preprocessor commands...
Preprocessor commands...

c) \texttt{#include <filename> | "filename"}
- header file inclusion (.h files)
- the \textit{interface} to library routines
- \texttt{<...>} means use \textit{system path}
- \texttt{"..."} means \textit{user-defined path}
- typically use for project-specific header files

- CodeWarrior path settings
  - project Properties $\rightarrow$ C/C++ Build $\rightarrow$ Settings $\rightarrow$ Compiler $\rightarrow$ Input

d) \texttt{#ifdef / #ifndef id}
- conditionally includes code based on existence of previous \texttt{#define}'s
- ex:

\begin{verbatim}
#ifdef ARDUINO
    #define MAXINT 32767
#else
    #define MAXINT 2147483648
#endif
\end{verbatim}
3: Embedded System Software Development

Preprocessor commands...

e) \#if expr

- conditionally includes code based on constant result of expression (no program variables!)
- ex:

  \#ifdef RAMSIZE > 1024
  char SCIbuf[800];
  \#else
  char SCIbuf[128];
  \#endif

2: Compiler

- translates HLL source code to either assembly source or direct object code
- native- vs. cross-compiler
- supports multiple source modules
  - object code is "intermediate" & becomes input to the linker
- typically has many features
  - controlled by ES programmer
- may also provide optimization
  - time and/or space, ex...
3. Assembler

- def: a program which takes the mnemonic form of a processor's instructions and converts into a binary object code equivalent
  - a "machine language compiler"
- frees programmer of knowing opcodes, hex/binary but still requires intimate knowledge of a machine's architectural details
  - ex: D=A:B on S12
Assembler Operations

- convert mnemonics to opcodes
- assigns memory addresses to mnemonics & labels
- evaluates & assigns operands using addressing mode rules
- generates object code file and/or listing, symbol table

Assembler "Commands"

a) mnemonics: symbolic names of CPU opcodes
b) pseudo-ops: directives to the assembler that affect assembly process
   - process - if, base, end
   - memory allocation - org, ds/rmb, dw/fdb
   - listing control - page
3: Embedded System Software Development

Assembler "Commands"

c) macro instructions: a programmer-defined "instruction" which represents a predefined sequence of other instructions (body), possibly with arguments
- macro definition vs. macro expansion
- ex: BSET MACRO

4. Linker
- converts "generic" (relative) object code modules into executable (ready to execute) form
- combines multiple programmer-defined modules (compilation units) plus library routines
- resolves external references
  - ex: library function calls
- assigns entities to final memory locations (absolute)
  - ES programmer must control this step too!
  - Controlled via “linker control file”
3: Embedded System Software Development

5: Loader

- takes care of loading executable into memory and resolving all physical memory locations
- for embedded systems, this normally entails programming of non-volatile memory
  - EPROM, EEPROM, Flash

HLL Issues for Embedded Systems

i.e.: "C Extensions for Embedded Systems Development"
3: Embedded System Software Development

The "Issues"

- physical location assignment?
- in-line assembly?
- interrupt service routines (ISRs)?
- vectors / vector table?
- memory map assignment?

Physical Location Assignment

- programmer-defined assignment of identifiers to physical locations
  - ex: \texttt{DDRA=0xFF;}
  
  a) as an \texttt{absolute variable}
     \begin{verbatim}
     char ddrA @ 0x04; \quad // compiler-specific
     \end{verbatim}
  
  b) as a \texttt{pointer define} - more generic
     \begin{verbatim}
     \#define DDRA (*(char *)0x04)
     \end{verbatim}
Volatile Variables

- what about "variables" that are actually control or I/O registers which can change "on their own"?
  - ex: waiting for portA input to change
- the C qualifier *volatile* tells the compiler to make no assumptions about the var
  - and thus, no optimizations!
  - ex:
    ```c
    #define PORTA (*(volatile char *)0x00)
    ```

In-line Assembly

- used for full programmer control of CPU
- timing-sensitive situations
- take full advantage of target CPU features
  - ex: CLI, SEI
- a) a single in-line instruction statement
  ```c
  asm <assembly instr.>; <comment>
  ```
- b) "instruction" may also be a block
  ```c
  asm {
  :
  }
  ```
3: Embedded System Software Development

In-line Assembly Example

swapping bytes of a 16-bit word:

all in C:

```c
#define HIBYTE(w)    ((w)>>8 & 0xff)
#define LOBYTE(w)    ((w) & 0xff)
#define SWAPBYTES(w) (HIBYTE(w)+(LOBYTE(w)<<8))

data=SWAPBYTES(data);
```

using in-line assembly:

```assembly
asm {
  move.w data,d0
  byterev d0
  swap.w d0
  move.w d0,data
}
```

ISRs/Exception Handlers

- normal C functions end in RTS but ISRs must use RTI/RTE! how?
- typically via use of a compiler feature
  a) using the non-standard `interrupt` keyword
     ```c
     interrupt void Timer_ISR()
     ```
  b) using a `#pragma`; CodeWarrior ex:
     ```c
     #pragma TRAP_PROC
     void Timer_ISR()
     { ... }
     ```
3: Embedded System Software Development

Vectors

- linker normally takes care of reset vector, pointing it to “_Startup()”
  - which calls main()
  - _Startup() part of C runtime environment
- Q: what about other interrupt vectors?
  - A: handled by interrupt option or VECTOR option in the linker parameter file
    - also written by ES programmer
    - also compiler-specific

Memory Map Assignment

- placement of code/data/etc. is crucially important in ES engineering
- Q: how does ES programmer specify where data (RAM) & code (ROM) gets assigned?
  - A: handled by PLACEMENT or MEMORY option in the linker parameter file
  - ex: Project_Settings/Linker_Files/Project.lcf
3: Embedded System Software Development

ES Example

- **ONE.C** – C implementation of ONE.ASM
  - continuously increment output port as an 8-bit binary up counter
- exhibits:
  - one.c source listing
  - outputs from preprocessor & compiler
  - linker parameter file
  - map file output from linker

ES Application Development Tools

i.e.: "Tools of the Trade"
ES Development Scenarios

- previously: we used an editor and cross-assembler on a host PC then downloaded/executed on an EVB
  - ok if EVB has sufficient support including communication & monitor
- many ES development scenarios do not have these luxuries!
  - so, we use other tools...

ES Development Tools

1. Simulator
   - a software tool that mimics the operation of the target microcontroller
   - pros: don't need actual target h/w
   - cons: don't have actual target h/w
     - execution is non-realtime
     - actual I/Os not present
3: Embedded System Software Development

ES Development Tools

2. In-Circuit Simulator (ICS)
   - a simulator, as above, with h/w interfaces that allow connection to actual application circuitry
     - actual I/O now present
     - still not realtime execution

ES Development Tools

3. Emulator / In-Circuit Debugger (ICD)
   - a realtime development tool based on actual MCU target
     - executes instructions as will be executed in final application
     - may implement target ROM as RAM
     - includes circuitry / MCU support for h/w breakpoints
     - may include bus-state analysis features
3: Embedded System Software Development

ES Development Tools

4. Programmer / Burner
- converts s/w into firmware via programming into non-volatile memory
- EPROM, EEPROM, FLASH, OTP
- may be on-board MCU or in external ICs

ES Development Tools

5. Logic Analyzer
- an expensive, multi-channel digital storage scope with sophisticated triggering and display capabilities
- can capture/display all MCU activity before/after a particular event of interest
- typically used with conventional bus-type processors
3: Embedded System Software Development

Agilent 68-channel Logic Analyzer