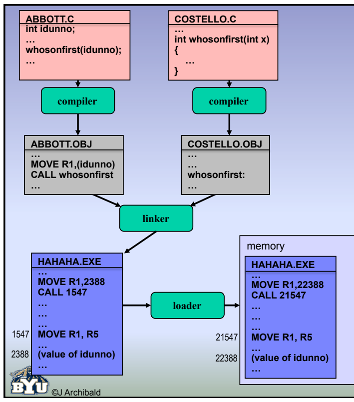
Chapter 9: Development tools

- Development tools for embedded systems must address some interesting and unique challenges
- Let's explore by considering a "native tool chain" – tools that run on the host and prepare a program to run on the host
- We'll then explore the differences of embedded tools that run on the host and prepare a program for a different target



Native tool chain

Compiler: translates C source files to machine instructions, produces object files

Linker: resolves addresses of different object files in terms of offsets from base address of program

Loader (for systems without virtual memory): loads program, modifies absolute addresses in executable based on actual starting location in physical memory

Linker

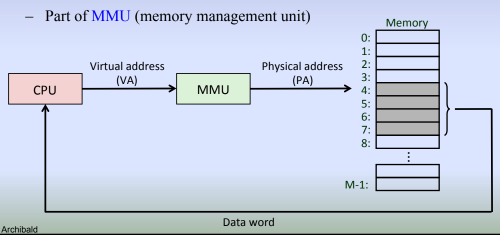
- Determines addresses of labels that assembler could not resolve
  - Obvious: `extern` functions and variables
  - Less obvious: addresses for symbols in same file
- Assembler will have marked instructions that require "fix-ups"
  - Includes all references to labels with unknown addresses
- Linker combines files, determines memory layout
  - Also "fixes" address references that assembler could not take care of
  - Linker throws error if any label or symbol not defined in combined files

Loader

- In system without virtual memory, a program might be loaded into a different region of memory each time it runs
  - Loader would then have to adjust memory addresses in code and data based on where program is loaded
  - Notable exception: no change required with PC-relative addresses
- In system with virtual memory, no address modification is required
  - Why? Let's review how virtual memory works

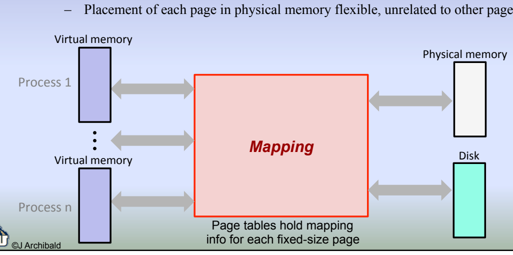
Virtual memory

- All addresses in program are <u>virtual</u>, not <u>physical</u>
- Hardware address translation unit dynamically translates virtual addresses to physical addresses
  - Part of MMU (memory management unit)



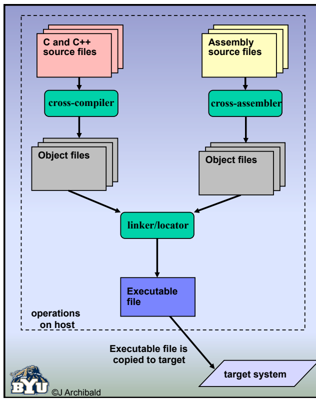
Virtual memory: a useful abstraction

- Operating system maintains page tables that map virtual address space of each process to shared physical address space
  - Each allocated page is either in physical memory or on disk (page fault)
  - Placement of each page in physical memory flexible, unrelated to other pages



Virtual memory

- Advantages
  - Allows for multiple processes, provides protection from other processes
  - User code thinks it has entire memory space to itself
  - Programs are written, compiled and run without any concern about where they will be placed in physical memory
  - No need to adjust memory addresses when program is loaded
  - Virtual memory can be larger than physical memory (DRAM)
- Disadvantages:
  - Increases complexity of hardware and OS
  - High runtime overhead (page faults, translation buffer misses)



Tool chain for embedded systems

Note components:
- **cross**-compiler
- **cross**-assembler
- linker/**locator**
- **no loader**

Questions:
- Why was no assembler shown in native tool chain?
- Why is there no loader?
- How does executable file make it from host to target?
- How does this setup compare with our tools?

Cross-Compilers

- C code is not perfectly portable: what are consequences?
  - Assume that C code compiles with native compiler and runs correctly on host
  - What can cause problems in moving code to target?

> - Using functions before declaring
> - Using old style function declarations → `int fun(x) float x; { ... }`
> - Word sizes (and hence `int`, `long`, `ptr` variables) may be different
> - Alignment conventions may change data layout (especially within structs)

Locator

- Quite different functionality than native linker
  - Determines final memory image of program
    - No loader required to adjust memory addresses when program is run
  - Locator can do this because:
    - No other program will be in memory at runtime; no resource conflicts
    - Final address of everything can be determined at this point, including all kernel and library functions called in application code
- Includes mechanism for programmer to specify memory placement
  - Can you think of examples of placement constraints?

Example locator output file format

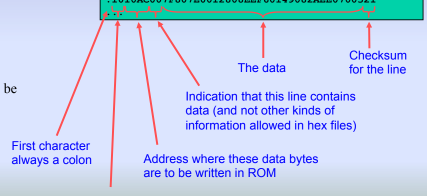
Other formats exist, with similar information

Essential information:
- Data to place in memory
- Address to write data to

Another common format:
- The actual binary image to be copied into ROM

Intel hex file format

```
...
:10106000FF908193E03400FAA9077B021227BE901B
:101070008193E0FEA3E0FF128157124FD1125BDB98
:10108000EF700F1222EBEF70097B057A1279BB121C
:0510900027BE1212C48E
:011095002238
:011096002237
:10109700E49082AEF012278B7f197E007D807C0062
:0410A7001281C622CA
:0110AB002222
:1010AC007F807E0012808EEF60149082AEE0700321
...
```



Memory placement

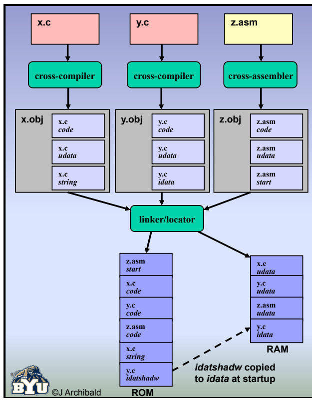
- Obvious constraints:
  - Code, initial values of data must be stored in ROM
  - Writable data must be in RAM at runtime
- How does the linker/locator know to put certain things in address space corresponding to RAM or ROM?
  - It doesn't – it can only do what programmer tells it to
  - Placement simplified by dividing program and data into segments
    - Contiguous portions of the runtime program that are similar in content
  - Each segment can be handled separately

Segments

- Each segment can be placed as a unit at desired memory address
- Examples:
  - Special “Start Code” segment can automatically be placed where processor begins execution after reset. (Typically **main()** in C code)
  - Interrupt vector table can be placed where CPU requires it
  - Code segments can be placed in ROM
  - Constant data segments can be placed in ROM
  - Variable data segments can be placed in RAM

Segment creation

- Created automatically by compiler
  - Also happens in desktop systems, but usually transparent to user
- In assembly files, programmer must specify segments
  - Assemblers not sophisticated enough to manage automatically
  - Naming should be consistent with compiler-generated segments
- All systems have similar categories of segments
  - Actual names depend on tools and developer



Example: segment use

Segments used:
- start
- code
- udata
- idata
- string

What are characteristics of each?

How did each get created, named?

Why is it useful to separate segments in this way?

How does initialized data gets initialized?

How linker/locator knows where to place segments

Instructions can be
- given on command line (placed in Makefile), or
- included in user-created assembly file

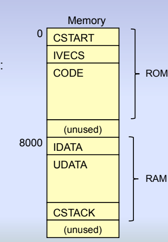
Starting address or ending address may be specified

Segments may be treated individually or as a group

Instructions to the locator:

```
-CSTART,IVECS,CODE=0
-IDATA,UDATA,CSTACK=8000
```

Resulting program in memory:

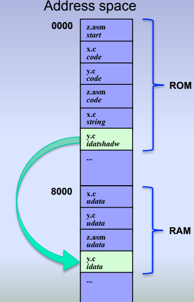


Handling initialized data

- How does normal tool chain (e.g., Linux) handle initialization of iFreq in code at right?
  - Is there an instruction in the program that writes the initial value to that variable?
- Does that same approach work on an embedded system?
  - Initial value must be in ROM
  - Runtime value must be in RAM
  - Therefore, initial value must be copied at startup

```
#define FREQ_DEFAULT 2410
...
static int iFreq = FREQ_DEFAULT;
...
void vSetFreq(int iFreqNew)
{
    iFreq = iFreqNew;
}
```


Handling initialized data



- Initialized data requires allocation of two distinct segments in memory space
  - Initial version (in ROM)
  - Run-time version (in RAM)
- For correct operation, segment must be copied from ROM to RAM *before* application begins execution
  - Some locators automatically insert code to do copy, but may require tweaking
  - Ultimately application code is responsible for this copying
  - Development tools differ in support they provide for this operation

Other initialization issues

- Can we assume that global variables are always initialized to 0?
  - C standard specifies this, but not true of all embedded tools
  - Startup code *may* be inserted to clear memory, but dangerous to assume that this is always done
- Where are strings stored?
  - Example: **char *sMSg = "Reactor is melting!";**
  - Initial value is in ROM; can stay there if all accesses are reads
  - But what if string is modified?
    - Perfectly legal C: **strcpy (&sMsg[11], "OK");**
  - Cross-compilers deal with this problem in different ways
    - Probably a good idea to see how it is handled in your system

Locator Maps

Maps provide a quick way of checking where the locator actually placed segments

Good idea to double check placement of critical segments

Also useful to know variable and function addresses when debugging

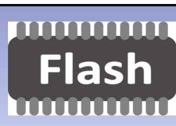
```
LINK MAP OF MODULE:  XYZ
  TYPE      BASE    LENGTH    RELOCATION       SEGMENT NAME
------------------------------------------------------------

     ******* XDATA MEMORY *******

            0000H   8100H                          *** GAP ***
  XDATA     8101H   0001H     UNIT             ?XD?PROGFLSH
  XDATA     8101H   000CH     UNIT             ?XD?VPROG?PROGFLSH
  XDATA     810DH   0006H     UNIT             ?XD?CHKSM?PROGFLSH
  XDATA     8113H   0080H     UNIT             ?C_LIB_XDATA
  XDATA     8193H   0002H     UNIT             ?XD?MAIN?PAD
  XDATA     8195H   0002H     UNIT             ?XD?RXCALLBACK?PAD
  ...

    ******* CODE MEMORY *******

            0000H   0017H                          *** GAP ***
  CODE      0080H   000FH     UNIT             PROGFLSTSTA
  CODE      008FH   0055H     UNIT             PROGFLSA
  CODE      00E4H   01ADH     UNIT             ?PR?VPROG?PROGFLSH
  CODE      0291H   0073H     UNIT             ?PR?SEND?PROGFLSH
  CODE      0304H   001DH     UNIT             ?PR?RX?PROGFLSH
  CODE      0321H   0072H     UNIT             ?PR?CHKSM?PROGFLSH
  CODE      0393H   007EH     INBLOCK          SCC_INIT
  CODE      0411H   082EH     UNIT             ?C_LIB_CODE
  ...

  SYMBOL TABLE OF MODULE:  XYZ

  VALUE             TYPE      NAME
------------------------------------------------------------
  ------            PROC      _FDECIMALASCIITOBYTE
  X:8301H           SYMBOL    p_b
  X:8304H           SYMBOL    p_byAscii
  X:8307H           SYMBOL    sizeofAByAscii
  D:0007H           SYMBOL    fReturn
  D:0006H           SYMBOL    bTemp
  ------            PROC      _FDECIMALASCIITOWORD
  X:8308H           SYMBOL    p_w
  X:830BH           SYMBOL    p_byAscii
  X:830EH           SYMBOL    sizeofAByAscii
  ...
```


One more complication

- RAM is often faster than flash and ROM, so better performance may be obtained by executing program in RAM
- Requires startup code that copies code segments from ROM to RAM, then transfers control to code in RAM
- Consider new challenge for locator:
  - Build a program that is stored at one address (in ROM), but will run correctly at a different address (in RAM)
  - Tricky: requires support from development system to (1) construct programs this way and (2) to insert code to perform the copy at start up

9.3: How does program get to target?

- Several alternatives:
  - Write it to flash memory on target
  - Program a PROM chip, then insert into socket on target system
  - Use a ROM emulator
  - Use an in-circuit emulator (ICE)

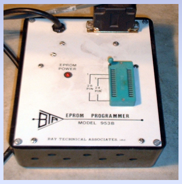


- Flash memory is field programmable
  - Host can connect to target, reprogram flash without pulling and reinserting chips, bending pins, etc.
- Requires special program on target that receives new program from host via communication link and writes it to flash
  - Tricky: this program cannot run in flash while flash is being updated, so program must copy itself from flash to RAM before executing
  - Locator will have built program to run at original location in flash, but it has to run correctly in new location in RAM

Flash and field upgrades

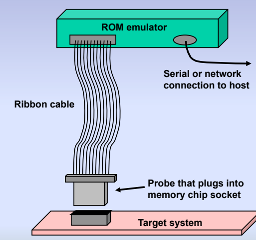
- Product code fixes can be expensive
  - Product must be physically present to upgrade memory, or customer must be able to cause firmware upgrade in the field
- Some products are obvious candidates for upgrading in field
  - Cell phones
  - Satellite/cable TV receivers
  - Digital radios
- Tricky: what if communication link fails during update?
  - Disaster if neither old nor new code works
  - How can designers ensure this never happens?

PROM Programmer



- Writes executable code into a PROM
- PROM is then inserted into memory socket on the target system
- Okay for production mode, but inconvenient during development
  - Tiresome to pull, program, and reinsert chip to test each new change in software

ROM emulator



- Device plugs into memory socket, looks like ROM to target
- Host can easily make changes to memory that target sees
- Particularly convenient during development and debugging
- In shipped system, ROM will be inserted into memory socket

In-circuit emulator

- Replaces microprocessor in target system: plugs into CPU socket
- Memory accesses will access memory in emulator instead of memory on target system
- Useful for debugging and development; shipped products will have microprocessor in its place

10 TOP CAUSES OF NASTY EMBEDDED SOFTWARE BUGS

Michael Barr, EE Times, April 2010, November 2010

Barr: Quote

Finding and killing latent bugs in embedded software is a difficult business. Heroic efforts and expensive tools are often required to trace backward from an observed crash, hang, or other unplanned run-time behavior to the root cause. In the worst case, the root cause damages the code or data in a way that the system still appears to work fine or mostly fine – at least for a while.

Too often engineers give up trying to discover the cause of infrequent anomalies that cannot be easily reproduced in the lab – dismissing them as user errors or "glitches." Yet these ghosts in the machine live on. Here's a guide to the most frequent root causes of difficult to reproduce bugs. Look for these top bugs whenever you are reading firmware source code. And follow the recommended best practices to prevent them from happening to you again.

Top 10

10. Jitter

    Problem: Having too much variation in timing between runs of same task or job. Affects accuracy in sampling a physical signal (e.g. A/D converter, optical encoder).

    Solution: Increase task priority, or put code in an ISR rather than a task.

9. Incorrect priority assignment

    Problem: Using ad hoc priorities for tasks that seem to work in testing, but that may fail in field workloads.

    Solution: Assign task and ISR priorities using a rate monotonic approach that proves that worst-case "transient overloads" can be handled.

Top 10

8. Priority inversion
   Problem: Arises using RTOS with fixed task priorities; high-priority task misses deadline because lower-priority task holds resource exclusively, and medium priority task has CPU.
   Solution: Use RTOS with priority-inversion work-around (e.g. priority inheritance), call only appropriate routines.
7. Deadlock
   Problem: Circular dependency blocks multiple tasks.
   Solution: Never attempt or require simultaneous acquisition of multiple exclusive resources; alternately, acquire exclusive resources in same order system-wide.

Top 10

6. Memory leak
   Problem: Systems with dynamic memory allocation that fail to return all blocks of memory to available pool; eventually system runs out of free space.
   Solution: Ensure that every allocated object has a designated destroyer to free memory it uses; follow a clear ownership pattern for all objects.
5. Heap fragmentation
   Problem: Heap (pool used by dynamic memory allocator) consists only of smaller, non-adjacent fragments after many allocations and deletions. Next allocation request fails, even though enough memory is available.
   Solution: Avoid use of the heap; if dynamic memory allocation is required, make all requests the same size, or use memory pools of fixed size blocks.

Top 10

4. Stack overflow
   Problem: Stack size can't handle <u>rare</u> worst-case needs. Testing cannot guarantee that stacks are big enough. Overflow clobbers arbitrary data or instructions.
   Solution: Perform detailed static analysis of control flow; repeat every time code changes. Also, fill stacks with specific pattern, have supervisor task test to ensure no changes above high-water mark.
3. Missing `volatile` keyword
   Problem: Failure to tag certain variables as `volatile` changes system behavior with compiler optimization.
   Solution: Use for all shared globals, pointers to memory-mapped peripherals, and delay loop counters.

Top 10

2. Non-reentrant function
   Problem: Shared function contains unrecognized critical sections. Not limited to your code: may be third-party middleware, legacy code, device drivers, or even standard library routines.
   Solution: If each module is not intrinsically reentrant, add and use mutex (semaphore) that protects shared resource.
1. Race condition
   Problem: Outcome of 2+ execution threads depends on precise instruction execution order.
   Solution: Recognize critical sections (accesses to shared objects); ensure atomic execution with appropriate preemption-limiting mechanism; use naming conventions (e.g. "`g_`" prefix) for all potentially shared objects so that risk is obvious to everyone who reads the code.

Chapter 10: Debugging

- Quote from author:

  > If you write code with lots of bugs in it, you will ship code with lots of bugs in it.

- What does this say about testing and debugging?
  – Why are these hard in general?
  – Why are they even harder in embedded systems?
- How tolerant is the world of software bugs, in general?
  – Do consumers expect embedded systems to be more reliable?

Avoiding software bugs

- The best approach would be to produce bug-free code, but virtually all software has errors
- Developers are foolish if they don't do a lot of testing, but it cannot be their primary technique for ensuring software quality
- Unfortunately, embedded systems pose special challenges for testing

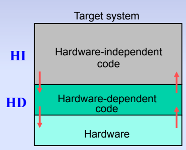
Problems testing embedded software

- Target hardware may not be available or stable early on while code is being written and debugged
- Difficult to generate all pathological timing scenarios
  - Impossible to test all combinations, and difficult to know which combinations could cause a problem
- Some bugs can be virtually impossible to reproduce
  - Caused by specific event sequence and timing
  - Very hard to generate using standard software test suites
- Embedded systems generally lack detailed logging capabilities to identify cause of failures

Testing

- Conclusion: can't rely on extensive testing on target system
  - You'll inevitably do some, but it can't be your main plan of attack
  - What else can you do?
- How about testing on the host?
  - Debugging and testing is more convenient, but full code won't run
  - Timing is altered, so not much help with race-condition/shared-data bugs
  - Still, more useful than one might think
- Let's consider two ways to test embedded software on the host system

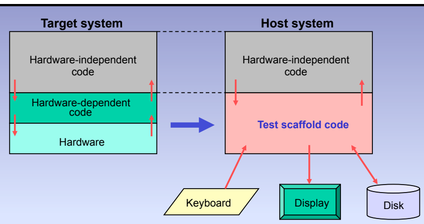
Testing on the host: method 1



- In design phase, separate the application code into hardware dependent code (HD) and hardware independent code (HI)
- Essential: creation of a clean interface between HD and HI
- HI is just C code – easy to compile and run on other computers
- HD portion is more problematic

Hardware dependent code

- It is clearly not portable, yet it is crucial to system operation:
  - It interfaces to sensors: all system events and interrupts that cause HI code to run come through HD code
  - It interfaces to actuators: all externally observable actions taken by HI code go through device drivers in the HD code
- Conclusion: system won't actually do anything meaningful without the HD code
  - So how could we test the HI code on the host?



Solution: construct test scaffold code with same entry points (API) as HD portion of code

Scaffold code triggers actions in HI code; it also responds to actions by writing log files, and mimicking hardware actions at a high level, "faking" the behavior of the hardware subsystems

This approach does *not* require a detailed, low-level hardware simulator

Testing the system

- ISRs and interrupt handlers must also be divided into HD, HI parts
  - Required organization: HD code must call the HI part
  - Scaffold code will mimic HD actions, calling HI code to test its response
  - This can be more effective and thorough than testing HI code on target
- Consider challenge of generating tick interrupts
  - Clearly responsibility of test scaffold code, but how to implement?
    - Generate automatically at fixed time intervals?
    - Generate directly and explicitly?
  - The latter is preferred: more likely to turn up errors with unusual combinations of events occurring within same tick interval

Testing the system

- Important to use a scripting mechanism for convenience
  - Scaffold code reads script file that tells it what events to generate
  - General form: take this action (call this HI function) with these parameters at this time
  - Challenge: events are platform specific, so custom script "language" is needed
- Good news: it is not difficult to create simple parser that reads files and generates specified function calls
  - Even simple tools can be very effective
  - Well worth the effort to consider testing in design phase, and to build tools that make testing easier
- For ease of use, scaffold code can output results interleaved with script input
  - Makes it easy to follow actions and confirm correct operation

Script file example

```
# Frame arrives (beacon with no element)
# Dst   Src   Ctrl            Typ  Stn  Timestamp
mr/56   ab    01234566789ab   30   00   6a6a

# Backoff timeout expires
# (Software should send association frame)
kt0

# Timeout expires again
# (Association process should fail)
kt0

# Some time passes ---
# (Software should retry sending the association frame)
kn2
kn2

# Another beacon frame arrives
# Dst   Src   Ctrl            Typ  Stn  Timestamp
mr/56   ab    01234566789ab   30   00   6a6a

# More time passes
# (Should NOT send another association until
#  backoff time expires)
kn1

# Backoff timeout expires
#(System should send association frame)
kt0
.
.
.
```

- For cordless bar-code scanner
- Each command causes scaffold code to call an interrupt routine
- kt0: call timer interrupt routine
- kn: calls another timer routine a specified number of times
- mr: writes data into memory (as if received via radio) and calls radio interrupt routine

Sample output from script

```
# Frame arrives (beacon with no element)
# Dst   Src   Ctrl            Typ  Stn  Timestamp
mr/56   ab    01234566789ab   30   00   6a6a

# Backoff timeout expires
# (Software should send association frame)
kt0
-->SENDING FRAME: ab ff 01 23 45 67 89 ab 50 09 30 09 01 02 05 03

# Timeout expires again
# (Association process should fail)
kt0

# Some time passes ---
# (Software should retry sending the association frame)
kn2
kn2
-->SENDING FRAME: ab ff 01 23 45 67 89 ab 50 09 30 09 01 02 05 03


# Another beacon frame arrives
# Dst   Src   Ctrl            Typ  Stn  Timestamp
mr/56   ab    01234566789ab   30   00   6a6a

# More time passes
# (Should NOT send another association until
#  backoff time expires)
kn1

# Backoff timeout expires
#(System should send association frame)
kt0
-->SENDING FRAME: ab ff 01 23 45 67 89 ab 50 09 30 09 01 02 05 03

.
.
.
```

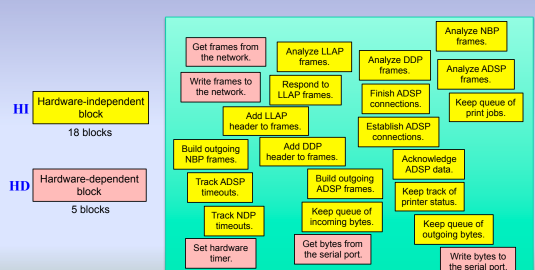
Output from scaffold code shown in red

How useful would this mechanism be?

Possible objections

- "Too much of software is hardware dependent"
  - Actually most software is hardware independent and can be tested in this way (See next slide)
- "Scaffold software is too much work to create"
  - It is actually quite simple – it just reads a script and calls specified HI functions; it adds output when its functions are called from HI code
- "You'd need a version of RTOS running on host"
  - Most vendors happily supply this

Fraction of "Telegraph" code that is hardware-dependent



Limitations of testing on host

- Some things cannot be tested, including
  - Software/hardware interaction
  - Response time and throughput
  - Shared data problems and timing pathologies
  - Portability problems (endianness, packed data-structures, etc.)
- But it makes sense to test as much as you can on the host before testing on the target, since that is more difficult

Testing on the host: method 2

- Use an instruction set simulator that runs on host
- Uses actual binary image that will run on the target, as constructed by cross-compiler and linker/locator
  - Avoids portability problems with word-size, endianness, etc.
  - Tests both assembly and C code
- What capabilities must the simulator have to be useful?

Simulator requirements

- Must simulate all assembly instructions of target CPU
- Must simulate all built in peripherals at some level
  - Timers, DMA, I/O devices, etc.
- Must model RAM and ROM at proper addresses
- Should provide a debugger interface:
  - Set breakpoints
  - Examine/change memory
  - Single step execution
- Should track timing in terms of *instructions* or *bus cycles*
  - Can give accurate measurements of the run time of various routines
  - Can give some insight into throughput and response time

How does our 425 simulator compare?

Limitations of this approach

- Unlikely that you can simply buy the simulator you want
  - Commercial simulator will know nothing about your custom hardware
  - May be possible to add desired functionality yourself, but vendors don’t generally make their source code available
- Simulator is unlikely to reveal obscure shared-data bugs
  - You are unlikely to do exhaustive testing to turn up all problems
- Tough to use scripting because simulators don’t usually give access to host’s keyboard, screen, and file system
  - 425 simulator is unusual in this regard – by design
- Recommendation: use simulator to test what cannot be tested using scaffold approach:
  - Startup code, ISRs, etc.

Section 10.3: assert macros

- Low cost technique that catches lots of bugs
- Sprinkle “assert” macro calls throughout code:

```
assert (pFrame != NULL);
assert (byMacAddrFrom <= ADDR_MAX);
assert (pframe->byMode & MAX_MODE_USE_STATION);
...
```

- Makes explicit assumptions about machine state (global variable or parameter values)
- If condition is true, nothing happens. If false, output is generated, generally halting execution. Example:

```
Assertion failed: ptr != 0, file foo.c, line 27     Abort (coredump)
```

- Implemented as macros so they can be turned off (#undef debug) and thus generate no code in shipped product

Example definition (in assert.h)

```
#ifdef NDEBUG
  #define assert(bool_expression)  /* define as nothing */
#else
  #define assert(bool_expression)       \
     if (bool_expression)               \
        ;                               \
     else                               \
        bad_assertion(__FILE__, __LINE__, #bool_expression);
#endif
```


Benefits of assertions

- Errors are caught much sooner, bringing the failure point closer to the error itself
- Quote from article in Linux magazine:

“Enthusiastic use of assert( ) can turn a three-day debug fest into a three minute bug fix. Practice the lazy developer mantra: An assertion failed is an hour saved.”

- Usually requires just **#include <assert.h>** and calls to **assert( )**
  - Make a practice of using in all your code, not just embedded applications
  - Mere presence of assertions helps document operational details and assumptions
- One common use: inspect function parameters

Assertions in embedded systems

- Particularly useful in development and when testing on host
- Harder to use on target system: typically has no screen to write "bad_assertion" message to
- Things you could do when assertion fails:
  - Make machine enter some easily detected state. Examples:
    - Turn off interrupts, spin in loop
    - Turn on special pattern of LEDs
  - Write one or more special error codes to a specific memory location
    - Values could then be determined with a logic analyzer
  - Cause emulator or target debugger to stop execution somehow
    - Could execute an illegal instruction, for example

Section 10.4: Using other tools

- Quote from text:

> "No book can do true justice to the experience of tracking down some subtle, inconsistent bug that only happens once every several hours and then only when your back is turned."

- You can probably relate, but worse on "real" projects due to limited visibility into target system
- What do you do? Bring in the heavy-duty tools
  - Volt meters, ohm meters, oscilloscopes, logic analyzers
  - Not part of typical programmers tool set!
  - What can each of these do for you and what are their limitations?

Volt meters, ohm meters, multi-meters

- Is the hardware working?
  - Do all chips in the circuit have power?
  - Is there a broken lead?
  - Is the wiring possibly incorrect?
  - Is a fuse blown?
  - Is everything connected that should be?
  - Is something connected that shouldn't be?

Oscilloscopes

- Graphs voltage vs. time, potentially multiple signals
  - Can select trigger to start operation
- Typical questions that can be answered:
  - Is anything running?
  - Is processor getting a decent clock input?
  - Is memory getting chip-enable signals?
  - Are output signals reasonable?
  - Is there a loading problem or a bus fight?

Logic analyzers

- Captures logical signals, stores in memory, graphs on screen
  - Can record many signals simultaneously – up to several hundred if you have $ and patience!
- Typical operation: trigger on symptom of problem, then look backward through captured data to find cause
  - Triggering mechanism can be very complex
- Timing mode: samples at fixed frequency
  - Captures data without reference to signals it records
- State mode: captures based on events observed in system
  - Typical use: record what instructions executed, what addresses accessed

In-circuit emulators

- Hardware emulator that plugs into CPU socket, appears to target system as regular microprocessor
  - Programmable, controlled by host
- Functionality similar to desktop debugger:
  - Set breakpoints
  - Single-step
  - Dump register and memory contents
- Often includes overlay memory that can be used instead of actual memory in the target system
  - Overlays specify subset of memory, RAM or ROM
  - On memory accesses in specified ranges, emulator uses overlay

Software-only monitor / debugging kernel

- Small debugging program in ROM on target system
  - Receives software over serial line, copies to RAM, and causes it to run
  - Provides debugging interface on host
  - Removed in final product
- Typical functionality from interface on host:
  - Set breakpoints
  - Examine memory and registers
- Typical methodology:
  - Compile code and download to target via monitor
  - Set breakpoints, run, and debug on target
- Requires no hardware modifications other than connection to host
  - Limitations: timing changed, breakpoints problematic in real-time systems, hardware breakpoint support required for code in flash

Trends making testing more difficult

- Pins on chips are getting closer
  - Harder to attach logic analyzers, oscilloscopes
- ASICs, FPGAs are replacing many simpler parts
  - Much more internal state that can't be observed externally
- Microprocessors with on-chip caches
  - You can't monitor accesses to internal cache
  - You can typically turn caches off, but this changes execution timing
  - Caches, pipelines complicate execution timing