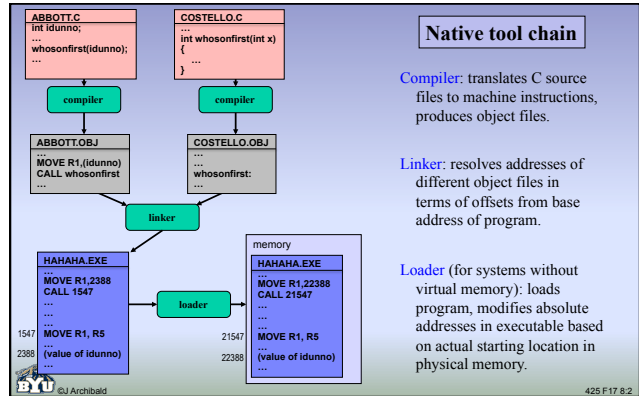


Chapter 9: Development tools

- 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.



425 F17 8.1



425 F17 8.2

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 requiring "**fix-ups**"
 - Including all references to labels at 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



425 F17 8.3

Loader

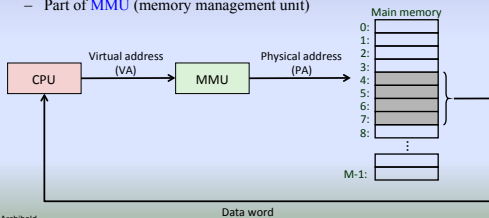
- In system **without virtual memory**, a program might be loaded into a different region of memory each time it runs.
 - Loader must **adjust memory addresses** in code and data based on where program is loaded.
 - Notable exception: no change required with PC-relative addresses.
- In system **with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



425 F17 8.4

Virtual memory

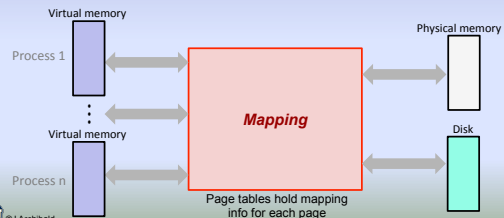
- Addresses in program are **virtual**, not **physical**.
- Hardware **address translation unit** dynamically translates virtual addresses to physical addresses.
 - Part of **MMU** (memory management unit)



425 F17 8.5

Virtual memory: a useful abstraction

- Operating system maintains page tables that map virtual address space of each process to 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.



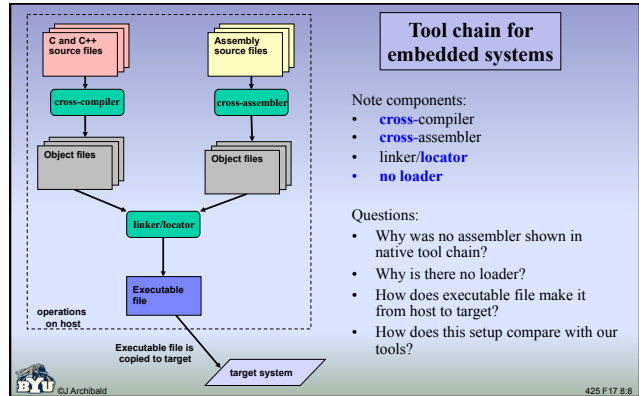
425 F17 8.6

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.
- Disadvantages:
 - Increases **complexity** of hardware and OS
 - High runtime **overhead** (page faults, translation buffer misses)



425 F17.8.7



425 F17.8.8

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
- Word sizes (and hence `int`, `long`, `ptr` variables) may be different
- Alignment conventions may change data layout (especially within structs)

```
int fun(x) float x;
{ ... }
```



425 F17.8.9

Locator

- Quite different functionality than native linker
 - Determines **final memory image** of program
 - No loader required to adjust memory addresses later
 - 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?



425 F17.8.10

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:
- Simply the binary image to be copied into ROM

Intel hex file format

```

:10106000F908193E03400FAA9077B021227BE901B
:101070008193E0FEA3E0FF128157124FD1125BDB98
:10108000EF700F1222BEF70097B057A1279BB121C
:0510900027BE31212C48E
:011095002238
:011096002237
:10109700E49082AEF012278B7E197E007D807C0062
:0410A7001281C622CA
:0110AB002222
:1010AC007F807E0012808EF60149082AE0700321
:-----

```

Annotations:

- First character always a colon
- Count of data bytes on this line
- Address where these data bytes are to be written in ROM
- Indication that this line contains data (and not other kinds of information allowed in hex files)
- The data
- Checksum for the line



425 F17.8.11

Memory placement

- Obvious **constraints**:
 - Code, initial values of data need to be in ROM.
 - Writable data needs to 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.



425 F17.8.12

Segments

- Each segment can be placed as a unit at desired address in physical memory.
- Examples:
 - Special “Start Code” segment can automatically be placed where processor begins execution after reset. (**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.



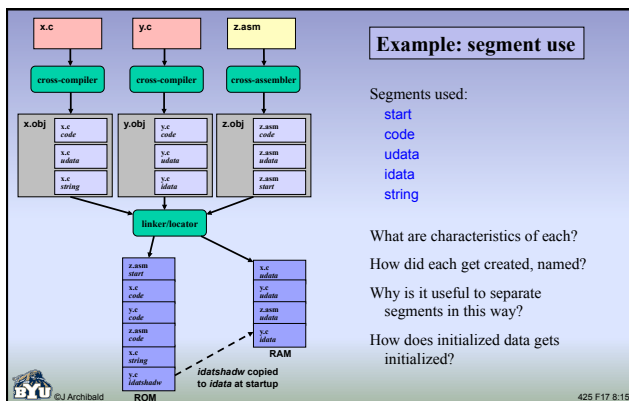
425.F17.8:13

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 transparently.
 - Naming should be consistent with compiler-generated segments.
- All systems have similar **categories** of segments.
 - Actual names depend on tools and developer.



425.F17.8:14



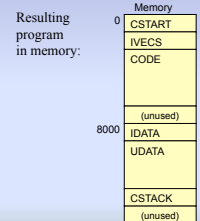
425.F17.8:15

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
```



425.F17.8:16

Handling initialized data

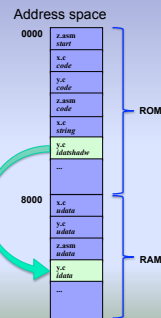
- How does normal tool chain (e.g., Linux) handle initialization of iFreq in code at right?
 - Is there an instruction that **writes the initial value** to that variable?
- Does that same approach work on an embedded system?
 - Initial value must be in ROM.
 - Runtime value must be in RAM.
 - Therefore,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

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



425.F17.8:17

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.



425.F17.8:18

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[1], "OK");`
 - Cross-compilers deal with this problem in different ways.
 - Probably a good idea to see how it is handled in your system.



425.F17.8.19

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 *****				
XDATA	000H	810H	UNIT	"" GAP ""
XDATA	810H	00FH	UNIT	?XD?PROGFLSH
XDATA	810H	00CH	UNIT	?XD?PROG?PROGFLSH
XDATA	810H	006H	UNIT	?XD?CHKSM?PROGFLSH
XDATA	8113H	008H	UNIT	?C LIB XDATA
XDATA	8113H	002H	UNIT	?D?MAIN?PAD
XDATA	8155H	002H	UNIT	?D?RXCALLBACK?PAD
***** CODE MEMORY *****				
CODE	000H	007H	UNIT	"" GAP ""
CODE	008H	00FH	UNIT	PROGFLSTSTA
CODE	008FH	005H	UNIT	PROGFLSA
CODE	0024H	0140H	UNIT	?PR?PROG?PROGFLSH
CODE	0291H	0073H	UNIT	?PR?SEND?PROGFLSH
CODE	0304H	0010H	UNIT	?PR?RX?PROGFLSH
CODE	0321H	0072H	UNIT	?PR?CHKSM?PROGFLSH
CODE	0355H	007EH	INLOCK	SCC INT
CODE	0411H	002EH	UNIT	?C LIB CODE
***** SYMBOL TABLE *****				
MODULE: XYZ				
VALUE	TYPE	NAME		
-----	PROC	_FDECIMALASCITOWORD		
X.830H	SYMBOL	p_b		
X.830H	SYMBOL	p_byAscii		
X.837H	SYMBOL	hwByAscii		
D.8007H	SYMBOL	Return		
D.8008H	SYMBOL	hTemp		
-----	PROC	_FDECIMALASCITOWORD		
X.830H	SYMBOL	p_b		
X.830H	SYMBOL	p_byAscii		
X.830H	SYMBOL	hwByAscii		



425.F17.8.20

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 *construct* programs this way and for start-up code that *copies the code*.



425.F17.8.21

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



425.F17.8.22

Flash

- 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.



425.F17.8.23

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 to ensure this can't happen?**



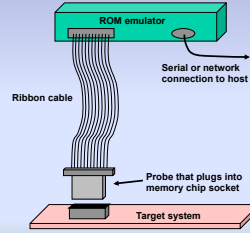
425.F17.8.24

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.



425 F17 8:31

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.

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.



425 F17 8:32

Top 10

4. Stack overflow

Problem: Stack size can't handle *rare* 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.



425 F17 8:33

Top 10

2. Non-reentrant function

Problem: Shared function contains unrecognized critical sections. Not limited to your code: may be 3rd 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.



425 F17 8:34

Chapter 10: Debugging

- 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?
- Are `expectations` higher for embedded systems?



425 F17 8:35

Avoiding software bugs

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



425 F17 8:36

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 will cause a problem.
- Bugs can be virtually **impossible to reproduce**.
 - Often 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.



425.F17.8.37

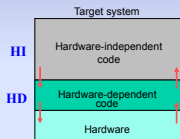
Testing

- 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 not the same, so less useful for race-condition/shared-data bugs.
 - Still, more useful than you might think.
- **Let's consider two ways to test embedded software on the host system.**



425.F17.8.38

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.



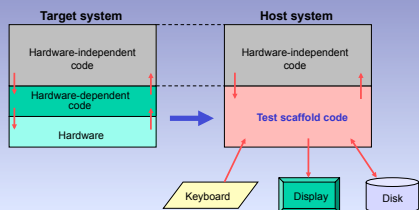
425.F17.8.39

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?



425.F17.8.40



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, mimicking hardware actions, "faking" the behavior of the hardware subsystems.

Note that a detailed hardware simulator is not required.



425.F17.8.41

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 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.



425.F17.8.42

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.



425.F17.8.43

Script file example

```

# Frame arrives (beacon with no element)
# Dst Src Ctrl Typ Stn Timestamp
mr/56 ab 0123456789ab 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 0123456789ab 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



425.F17.8.44

Sample output from script

Output from scaffold code shown in red

How useful would this mechanism be?

```

# Frame arrives (beacon with no element)
# Dst Src Ctrl Typ Stn Timestamp
mr/56 ab 0123456789ab 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 0123456789ab 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
    
```



425.F17.8.45

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.

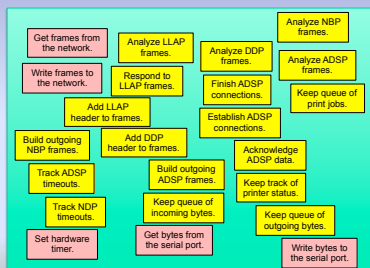


425.F17.8.46

Fraction of “Telegraph” code that is hardware-dependent

HI Hardware-independent block
18 blocks

HD Hardware-dependent block
5 blocks



425.F17.8.47

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.



425.F17.8.48

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 are requirements for simulator?



425.F17.8.49

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?

425.F17.8.50

Simulator limitations

- Can you simply buy what you want?
 - Commercial simulator will know nothing about your **custom hardware**.
 - May be possible to add desired functionality yourself, but vendors don't like to make 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.



425.F17.8.51

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 your assumptions about machine state (global variable or parameter values) explicit.
- 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 the final product.



425.F17.8.52

Example definition (in assert.h)

```
#ifndef NDEBUG
#define assert(boo!_expression) /* define as nothing */
#else
#define assert(boo!_expression) \
if (boo!_expression) \
: \
else \
bad_assertion(__FILE__, __LINE__, #boo!_expression);
#endif
```



425.F17.8.53

Benefits of assertions

- With **assert()**, 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.
- Commonly used to inspect parameters to functions.



425.F17.8.54

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 you can catch with a logic analyzer.
 - Cause emulator or target debugger to stop execution somehow.
 - Could execute an illegal instruction, for example.



425.F17.8.55

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 systems.
- 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?



425.F17.8.56

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 anything connected that shouldn't be?



425.F17.8.57

Oscilloscopes

- Graphs voltage vs. time, potentially multiple signals
- Can select trigger to start its 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?
- Storage oscilloscope:
 - Signals stored in memory, screen displays memory contents
 - Can capture one-time events, but much more expensive



425.F17.8.58

Logic analyzers

- Capture signals, store in memory, graph on screen.
- Can track 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 once triggered.
 - Captures data without reference to signals it records: **self-clocked**.
- State mode: capture based on events observed in system, including clock.
 - Typical use: see what instructions ran, what memory accessed.



425.F17.8.59

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.



425.F17.8.60

Software-only monitor / debugging kernel

- Small debugging program in ROM on target system that can
 - receive software over serial line, copy to RAM, and run it.
- From interface on host, you can usually
 - set breakpoints,
 - examine memory and registers, etc.
- Typical methodology:
 - Compile code and download to target via monitor.
 - Set breakpoints, run, and debug on target.
 - Monitor is removed in final product.
- 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.



425 F17 8:61

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.



425 F17 8:62