Locke Labs
for the common defense

New Tools for STEM, Cyber, and Makers

www.lockelabs.net

Locke Labs
for the common defense

Overview

- Motivation and Rationale
  - Execute Java source code directly on hardware
  - Concept inspired in part by modular features of Java 9
- Demo Configuration and Slides
  - Embedded Hardware
  - Host configuration
- Caveats
  - Proof of concept, necessary but not sufficient capabilities
    - Memory access, memory allocation and constraints, process interrupts
  - "Execution environment" is not a JVM
- Workflow and Package Structure
- Embedded Code Examples
  - Java main, memory allocation, a SoC module, memory access, interrupts
  - Memory corruption due to stack or heap overflow is currently prevented. Current heap protection approach is presented.
- Discussion

Locke Labs
for the common defense

Motivation and Rationale

- Currently we have large numbers of cyber vulnerabilities and attacks
  - At the same time, we have an increase in the number of people interested in STEM, cyber technology, and makers interested in electronics
- Suggesting that these 3 communities have a similar requirement
  - *How to more easily experiment with the interaction between hardware and software*
- This proof of concept demonstrates that the Java language and pending changes (modularity) have several advantages over traditional embedded programming (C and flavors of Unix)
  - Reduce the attack surface by reducing the number of lines of code executing
  - Stronger typing
  - More consistent memory constraints
  - Per TIOBE, the most popular programming language
  - Reduce the effort of setting up a cross-platform toolchain

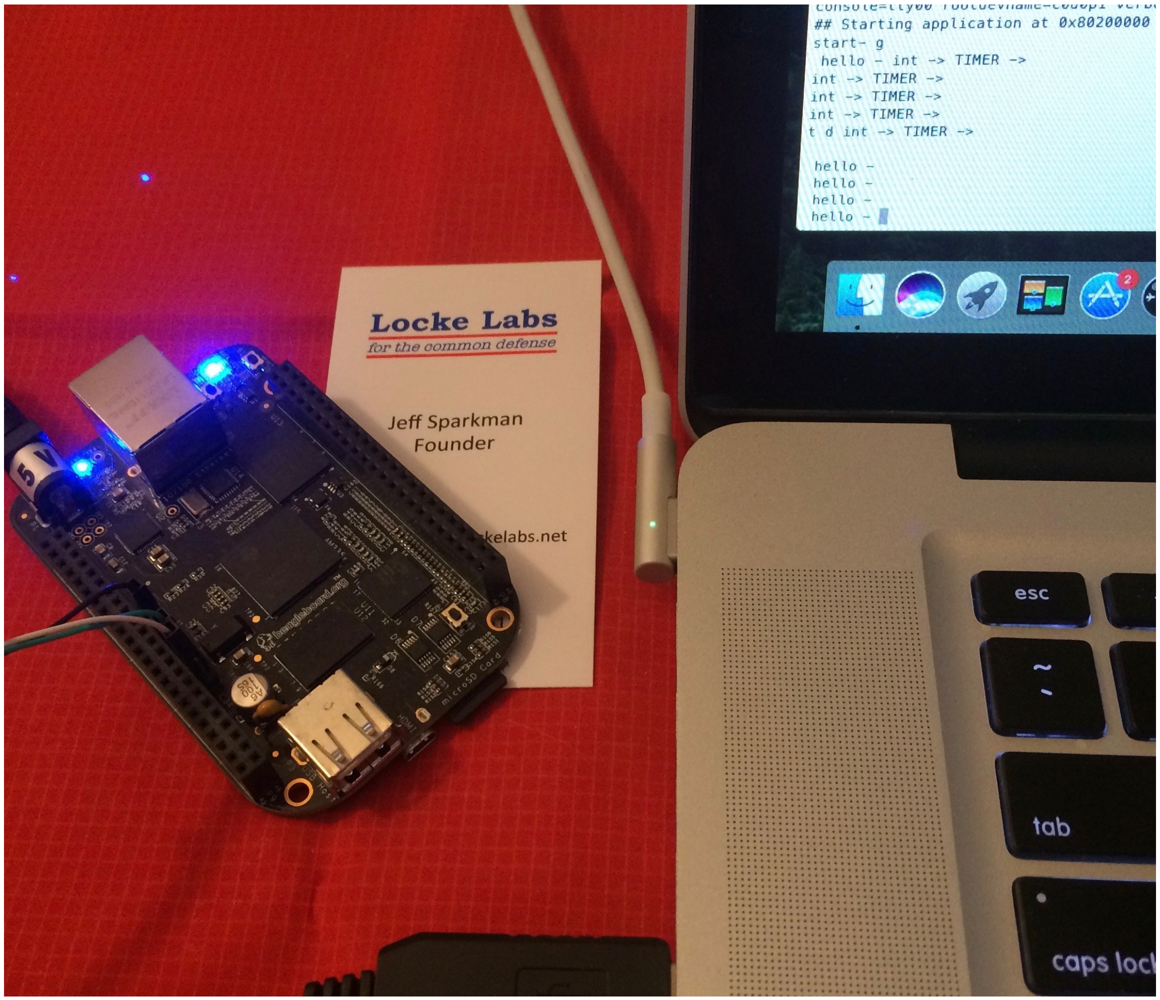
Locke Labs
for the common defense

Demo Slides

- Demo Java application utilizes timers, interrupts, and General Purpose Input Output to blink LEDs on a fixed frequency
- Embedded Hardware
  - Initially targeting a Beagle Bone Black board with a TI SoC (AM3358B) with an ARM Cortex-A8 MPU
  - SoC includes:
    - UART
    - Dual Mode Timer
    - Interrupt Controller
    - General Purpose Input Output
    - Power, Reset, and Clock Management
    - Programmable Real-Time Unit and Industrial Communication Subsystem (dual 32-bit RISC cores)
    - Enhanced Direct Memory Access
    - 3-port gigabit ethernet switch
    - Pulse Width Modulation Subsystem
    - USB
    - I2C
    - Controller Area Network (CAN)
    - Multichannel Serial Port Interface (McSPI)
- Host configuration
  - 2010 MacBook Pro
  - Version 4.0.2 of 'screen' terminal emulator
  - USB to TTL Serial Cable and Drivers to BBB Console Serial UART

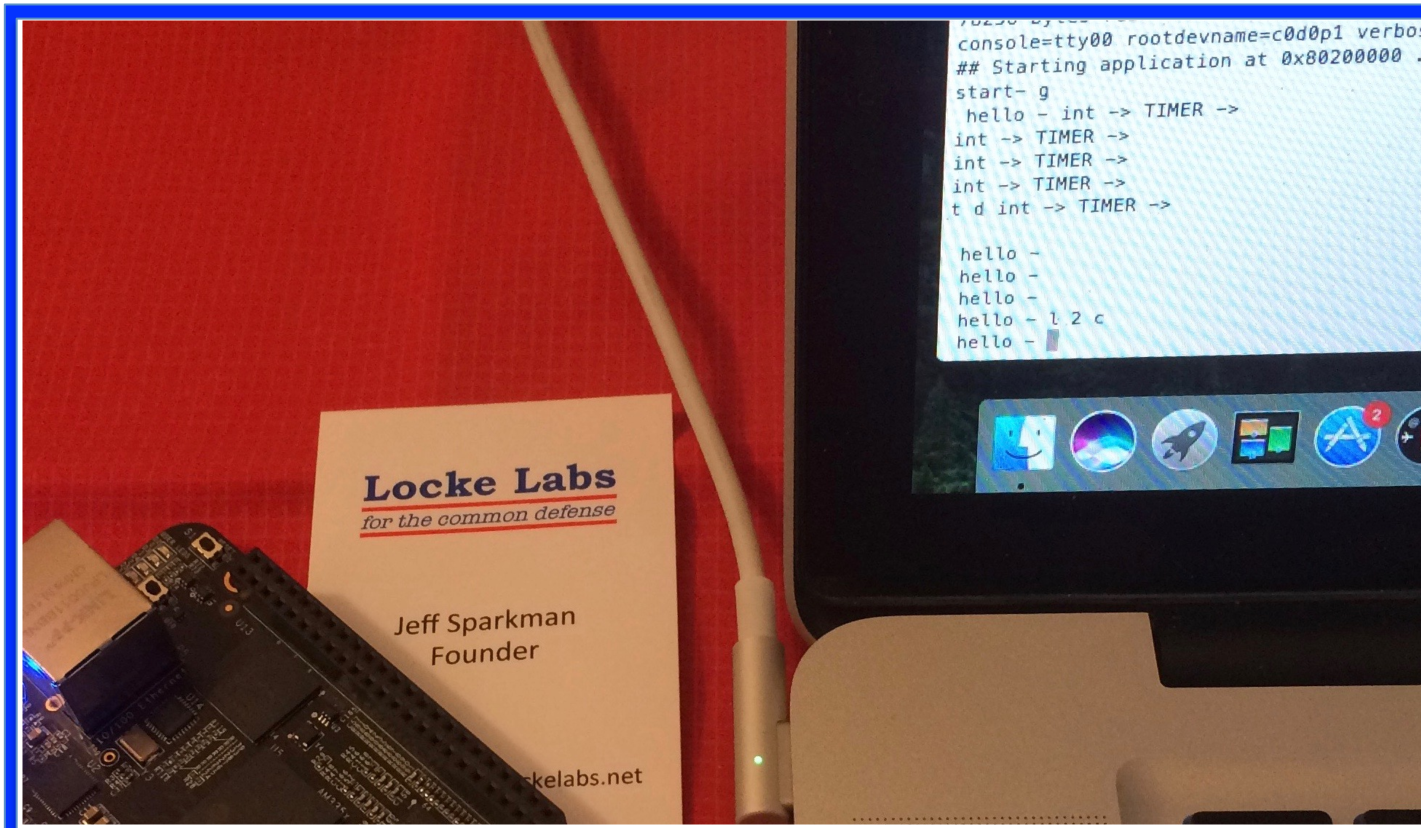
Locke Labs
for the common defense

Start the runtime, then disable Timers



- Runtime waits for ‘g’ command to run
- Hardcoded to start 2 sec timer. Timer interrupt service routine advances to the next LED
    - ‘int’ in output denotes that INTC generated the ARM exception
    - ‘TIMER’ denotes that the interrupt request was generated by the timer
- Timer Disable (‘t d’) command stops the timer

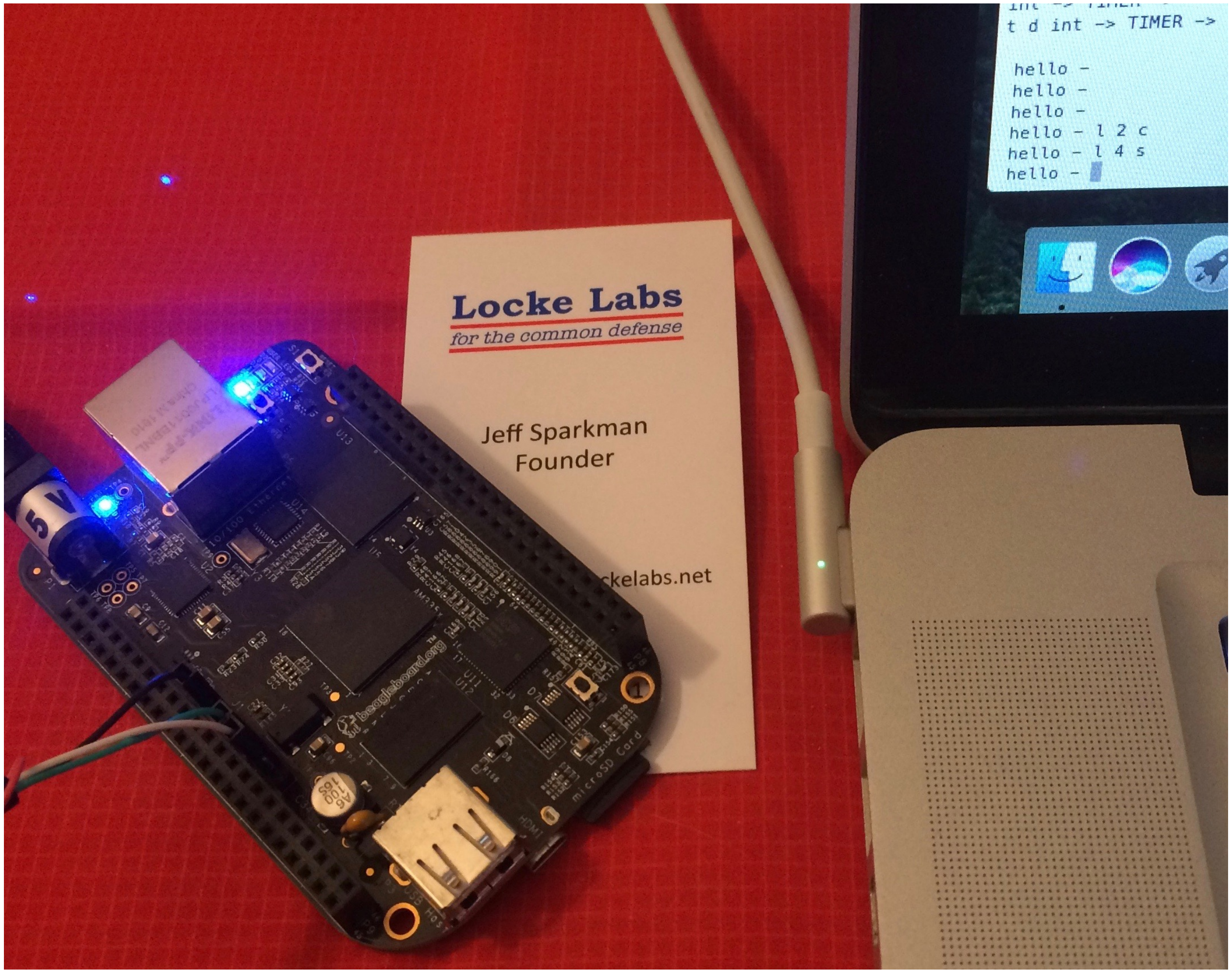
Interactively Turn a LED Off



- On previous slide, a single LED was illuminated (LED 2)
- LED clear command ('l 2 c') writes 0 to the control bit, turning the LED off

Locke Labs
for the common defense

Interactively Turn a LED On



- LED set command ('l 4 s') writes a 1 to the control bit, turning the LED on.
- Now LED 4 is illuminated.

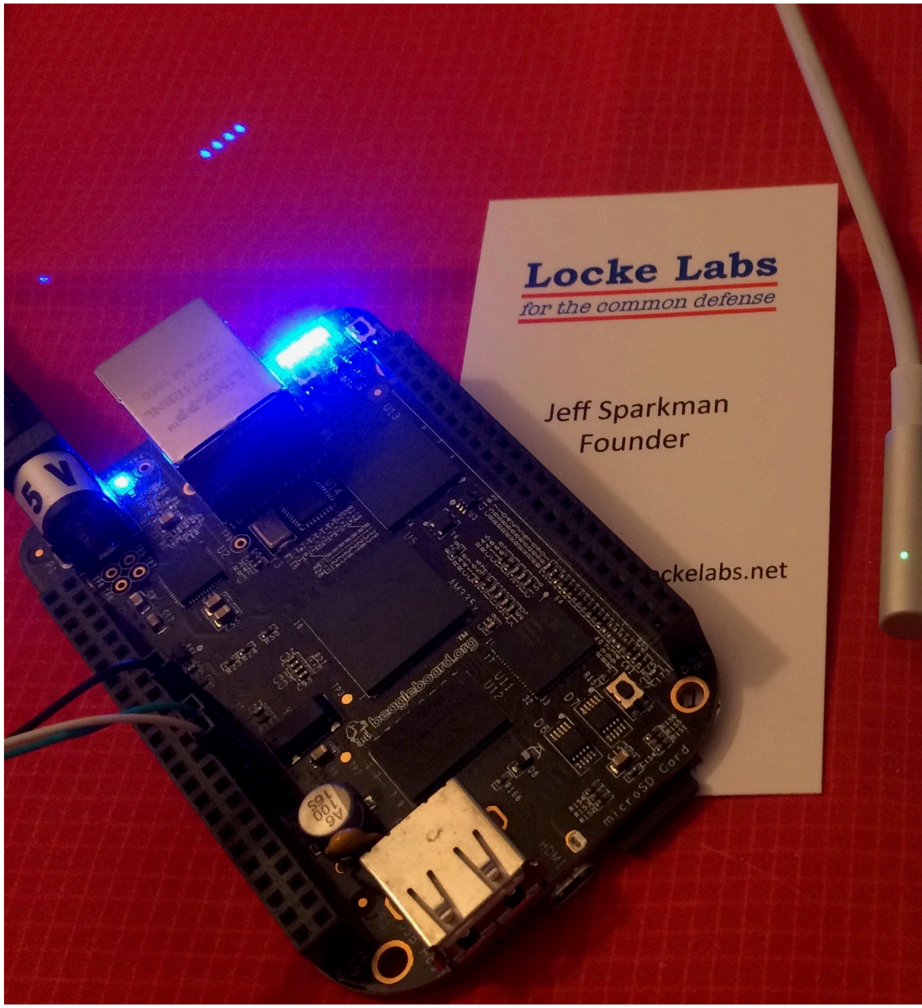
Locke Labs
for the common defense

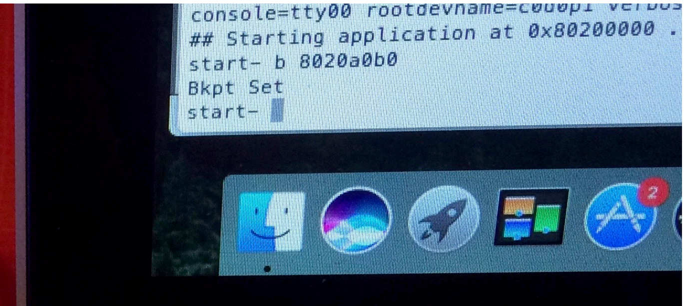
Simple Console Port Debugger

- Java code also implements a simple command line debugger that interactively provides limited (only on NOP) breakpoints, with the ability to display:
  - registers
  - memory contents at address
  - call stack
  - Memory contents of method variables

Locke Labs
for the common defense

Currently No Pause, set an initial Breakpoint

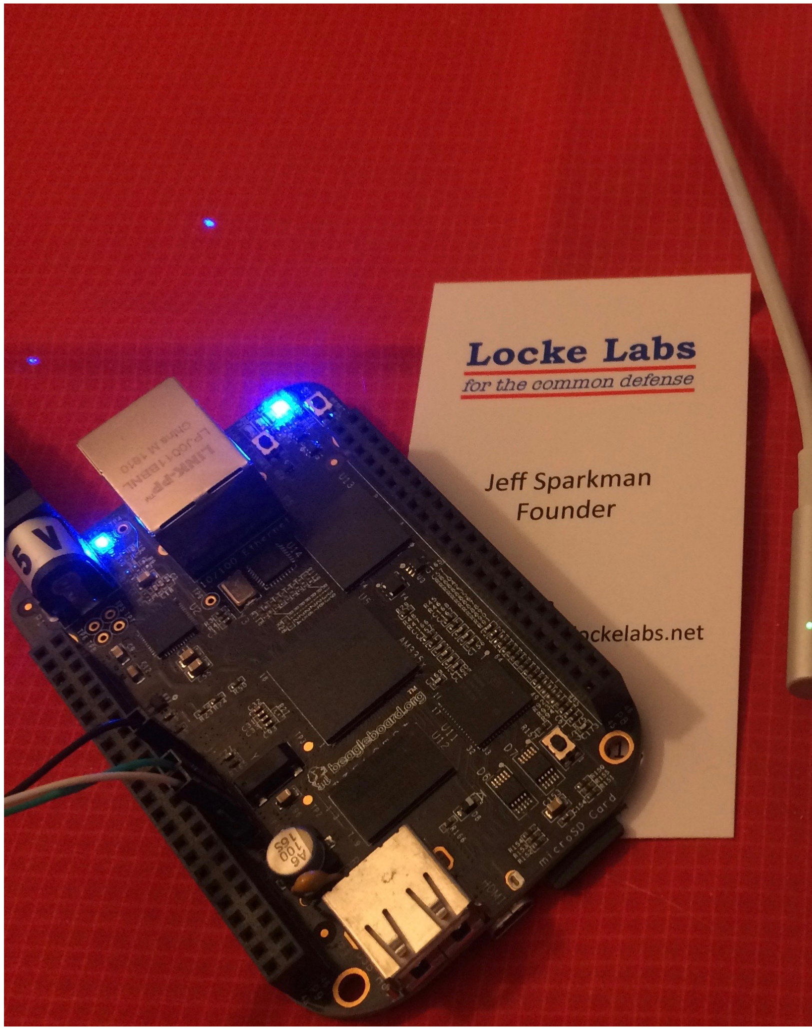


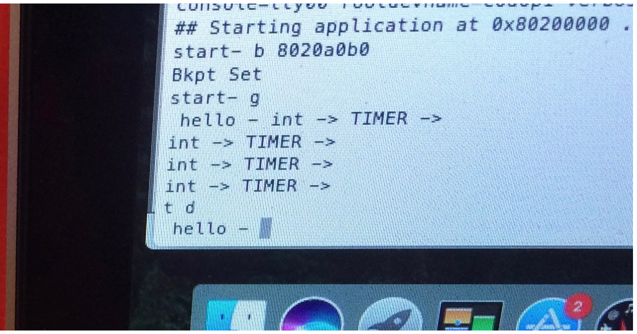


- In between this and the previous slide, a warm reset of the board was performed by the ‘p r’ command.
- Instead of entering the ‘g’ command, a breakpoint was set at the NOP instruction at 0x8020a0b0.
- This instruction is in the return path for the sum of a series from N to 1.
- The debugger verified the instruction was a NOP, set the breakpoint and echoed corresponding status ‘Bkpt Set’.

Locke Labs
for the common defense

Breakpoint is not set in Timer processing

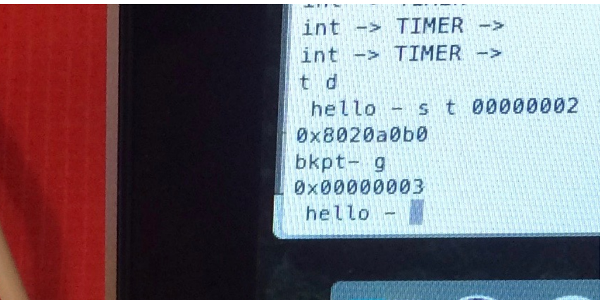




- Enter ‘g’ to continue execution of the runtime.
- The timer fires as before, no difference in execution.
- ‘t d’ disables the timer.
- The breakpoint is in a recursive routine used to test stack memory management and overflows.
- The routine is demonstrated in following slides.

Locke Labs
for the common defense

Hit a Breakpoint, But then Continue



- The series summation is invoked by the ‘s t’, or stack test command.
- Here, N = 2.
- The breakpoint at 0x8020a0b0 is hit.
- Execution is continued by entering the ‘g’ command.
- The stack test routine prints the sum, 3.
- Here, the first series summation call received a value of 2, followed by a recursive call with argument of 1 [2 – 1 = 1], where the breakpoint was hit.

Locke Labs

for the common defense

Display Stack Frames and Local Variables

```
console=tty00 rootdevname=c0d0p1 verbos
## Starting application at 0x80200000 .
start- b 8020a0b0
Bkpt Set
start- g
 hello - int -> TIMER ->
int -> TIMER ->
int -> TIMER ->
int -> TIMER ->
t d
 hello - s t 00000002
0x8020a0b0
bkpt- g
0x00000003
 hello - s t 00000003
0x8020a0b0
bkpt- f l
0x80211798
0x80211784
ls frames
index - 0x00000000
#vars - 0x00000002
rtnAd - 0x8020a0fc

index - 0x00000001
#vars - 0x00000002
rtnAd - 0x8020a0fc

index - 0x00000002
#vars - 0x00000002
rtnAd - 0x8020a394

index - 0x00000003
#vars - 0x00000004
rtnAd - 0x8020acd8

index - 0x00000004
#vars - 0x00000006
rtnAd - 0x80202098

bkpt-
```

- Several lines down on the left, the stack test is repeated with the command ‘s t 00000003’.
- The breakpoint is hit, and all the current stack frames are listed with the ‘f l’ command.
- For each frame, the number of local variables and return address are listed
- A few lines from the top on the right, the register values at the breakpoint are displayed with the ‘dr’ command.
- On the lower half of the right, the local variables of several of the stack frames are displayed via the ‘f v l 0000000i’ command, where “i” is 0, 1, and 2.

```
index - 0x00000004
#vars - 0x00000006
rtnAd - 0x80202098

bkpt- dr
cpsr- 0x60000193
R0: 0x00000001
R1: 0x00000001
R2: 0x80211000
R3: 0x80200000
R4: 0x80211798
R5: 0x80211784
R6: 0x80211798
R7: 0x9ff63c99
R8: 0x80211800
R9: 0x80211000
RA: 0x00000000
RB: 0x00000001
RC: 0x00000000
RD: 0x8021173c
RE: 0x8020a0fc
bkpt- f v l 00000000
0x80211798
0x80211784
ls vars
index - 0x00000000
val - 0x00000001

index - 0x00000001
val - 0x00000001

bkpt- f v l 00000001
0x80211798
0x80211784
ls vars
index - 0x00000000
val - 0x00000002

index - 0x00000001
val - 0x00000001

bkpt- f v l 00000002
0x80211798
0x80211784
ls vars
index - 0x00000000
val - 0x00000003

index - 0x00000001
val - 0x802123a8

bkpt-
```

Locke Labs
for the common defense

Caveats (1 of 2)

- Proof of concept, necessary but not sufficient capabilities to be a compliant JVM
  - Memory access, memory allocation and constraints, process interrupts
- “Execution environment” is not a JVM
  - Only executes native code statically linked with the execution environment
  - Currently a very small subset of the Java language features are implemented
    - Static classes, methods, primitive fields (int and boolean), dynamic allocation of character arrays (no memory reclamation)
    - Utilizing custom annotations to integrate link time information, Java source, and a limited amount of hard coded native assembly source
    - No objects, exceptions, or threads yet
  - Have not created a target Java platform, using project specific packages for defining and testing the current capability of heap and stack errors.

Locke Labs
for the common defense

Caveats (2 of 2)

- Java Specs
  - Java Language Specification, Java SE 8 Edition, 2015-02-13
  - Java Virtual Machine Specification, Java SE 8 Edition, 2015-02-13
- Status of JVM Instruction implementation
  - Implemented to varying degrees - aload, arraylength, astore, bipush, caload, castore, dup, getstatic, goto, iadd, iand, iconst, if_icmpge, ifeq, iflt, iinc, iload, imul, invokestatic, ireturn, istore, isub, ldc, newarray, putstatic, return
  - Currently no implementation includes – any array other than char array, double instructions, float instructions, related to objects such as invokespecial or invokevirtual, switch statements such as lookupswitch or tableswitch, synchronization such as monitorenter or monitorexit

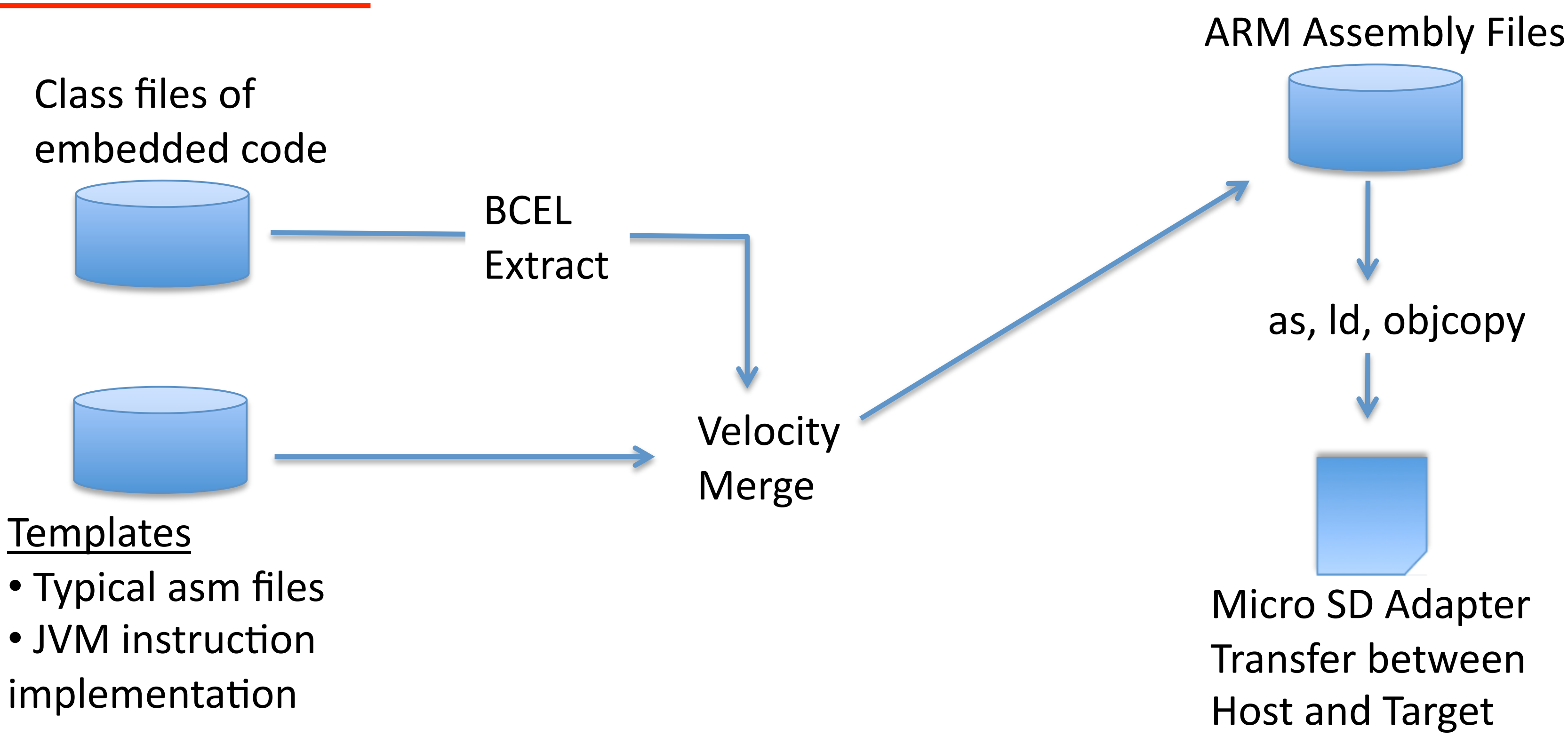
Locke Labs
for the common defense

Workflow (1 of 2)

- Cross Platform Toolchain
  - Cross compiled Minix ARM port to BeagleBone Black
  - http://wiki.minix3.org/doku.php?id=developersguide:minixonarm
  - From this, I am using u-boot and cross-platform GNU binutils – as, ld, objcopy, and objdump
- Target Build Process
  - Netbeans compile of embedded Java and native generation tool (also Java)
  - Run the native generation tool
    - Depends on BCEL and Velocity and generates ARM assembly
    - Running with JDK 8
  - Run as, ld, objcopy, and objdump
    - Ld is generating ELF, objcopy is transforming to binary
    - Objdump generates asm of linked executable to manually lookup addresses for breakpoints

Locke Labs
for the common defense

Workflow (2 of 2)

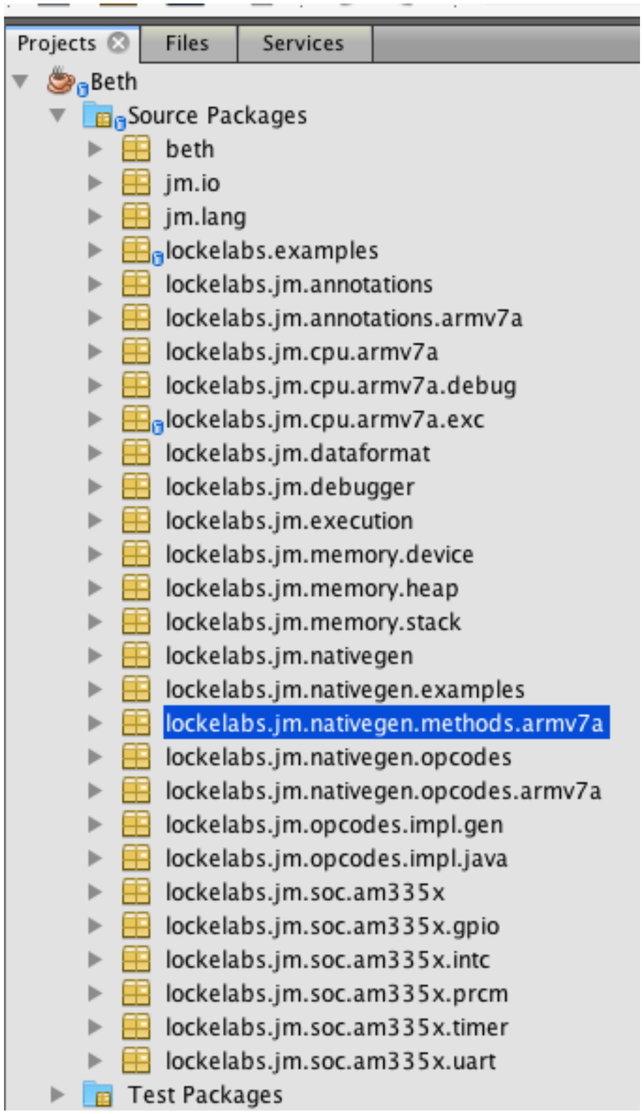


- This is a notional diagram of the native code generation process
- The flow 'Templates, Merge, Assembly File' is repeated for a variety of files and types.
- Some of these file types are the entry point to the executable, assembly routines that process ARM exceptions, templates for implementation of JVM instructions, etc.
- This native generation approach is very straightforward, no optimization is performed to reduce code size, number of operand stack accesses, etc

Locke Labs

for the common defense

Package Structure



- Non-compliant placeholder namespace for Stack and Heap errors (jm.lang)
- User application code (lockelabs.examples)
- Support code, either offline or on target (eg, lockelabs.jm.annotations, lockelabs.jm.cpu, lockelabs.jm.memory.heap)
- Ahead of time generation of native code (lockelabs.jm.nativegen)
- ‘Drivers’ for modules on the SoC (lockelabs.jm.soc)

Locke Labs
for the common defense

Embedded Code Examples

- Java main, memory allocation, a SoC module, memory access, interrupts
- Memory limit implementations
- *The following slides illustrate code that runs on the target.*
- *Information from the target build process is included with these slides to illustrate what was required to execute the Java source code.*

Java main - Initialization

```
62
63          char[] helloPrompt = new char[]{' ', 'h', 'e', 'l', 'l', 'o', ' ', '-', ' '};
64
65  //        char[] lsrString = new char[HexString.requiredBufferLength];
66
67          char[] counterString = new char[HexString.requiredBufferLength];
68
69          char[] eol = new char[]{'\n', '\r'};
70
71          int uartLineStatusRegister;
72
73          WatchDogTimer.disable();
74
75          GenlPurposeInputOutput.initialize();
76          DualModeTimer.initialize();
77          InterruptController.initialize();
78
79          DualModeTimer.startTimer();
80
81  //        Adder adder = new Adder(3, 4);
82
83          int counter = 0;
84          while (true) {
85
```

- Line 63 illustrates allocation and initialization of char array. Will look at implementation in following slides.
- Lines 73 – 79 illustrate initializing modules on the TI SoC (eg, GPIO, Timers, Interrupt Controller). Examine Interrupt Controller initialization in following slides.
- Line 84 starts the main loop of the main method. Illustrated on next slide.

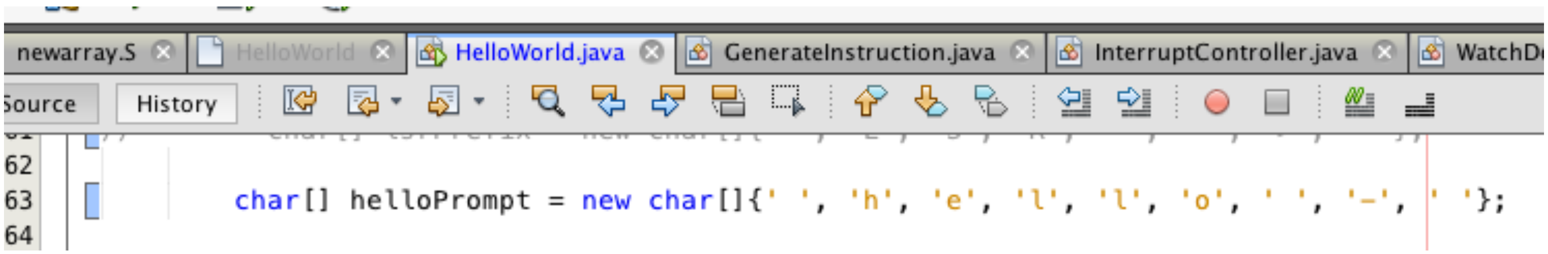
Java main - Loop

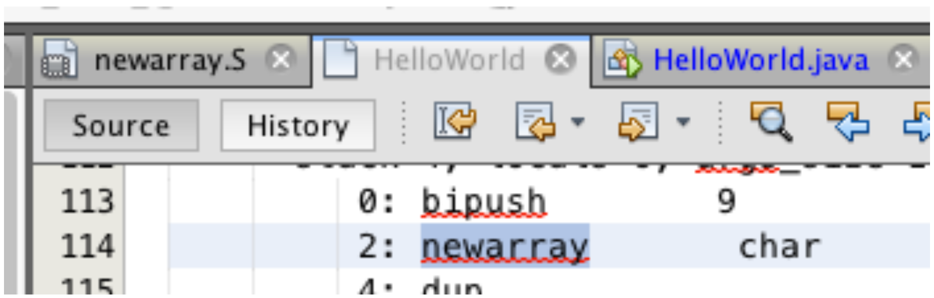
```
82
83          int counter = 0;
84          while (true) {
85
86
87              Instance.printf(uart0BaseAddress, helloPrompt);
88              Instance.readLine(uart0BaseAddress, fromUart);
89  //
90  //              InterruptController.printInfo();
91
92              GenlPurposeInputOutput.processCommand(fromUart);
93
94              DualModeTimer.processCommand(fromUart);
95
96              PrmDevice.processCommand(fromUart);
97
98              HeapManager.processCommand(fromUart);
99
100             StackManager.processCommand(fromUart);
101
102 //              HexString.convert(adder.add(), counterString);
103 //
104 //              Instance.printf(uart0BaseAddress, counterString);
105 //              Instance.printf(uart0BaseAddress, eol);
106
107             counter++;
108         }
109     }
```

- Line 88 reads characters and carriage return entered on Mac keyboard and transmitted over USB-to-TTL to UART (aka console port) on Beagle Bone Black.
- Lines 92 – 100 pass the current ‘command’ to each of the processCommand static methods.

Locke Labs
for the common defense

Memory Allocation (1 of 3)

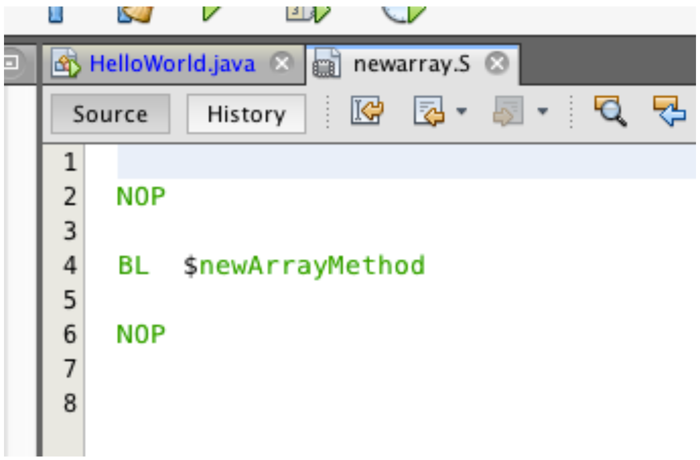




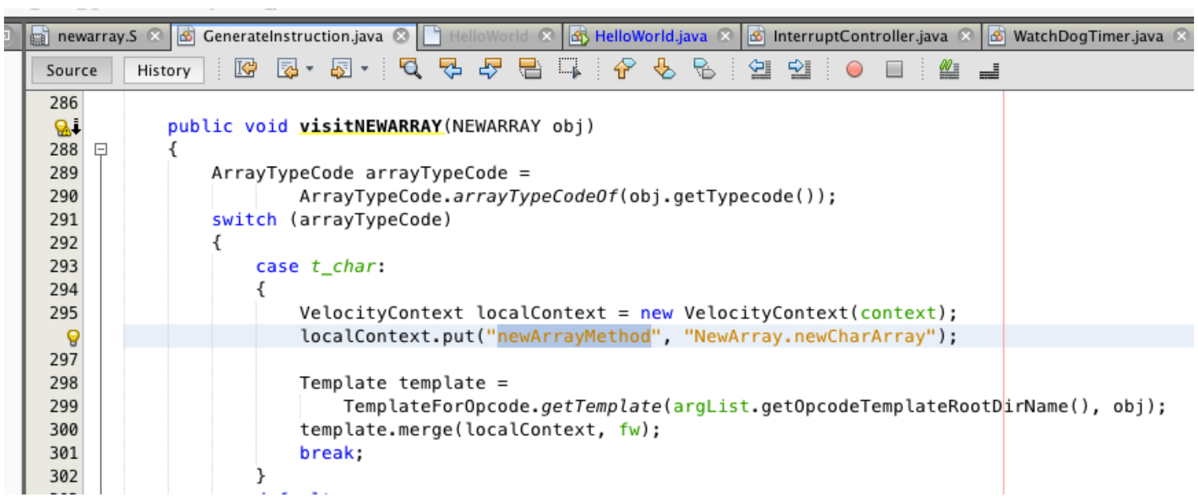
- Code above is from user main.
- At left are the disassembled JVM instructions for the allocation of the char array above.
- On the next slide, the current JVM instruction (newarray) is transformed to native code.

Locke Labs
for the common defense

Memory Allocation (2 of 3)



- Code below only runs during native code generation.
- Each JVM instruction has a corresponding template. At left is the template for newarray.
- During code generation, template is merged with relevant data from BCEL and written to current asm file. In this case, the HelloWorld.S file.
- Below, the variable newArrayMethod in the template at left is replaced with the class and static method (NewArray.newCharArray on line 296). This method is shown on the next slide.



Locke Labs
for the common defense

Memory Allocation (3 of 3)

GenerateInstruction.java | NewArray.java

Source | History

```java

    public static int newCharArray(int numberOfChars)
    {
        int address = allocateCharArray(numberOfChars);

        HeapMemory.writeHeapInt(address, numberOfChars);

        return address;
    }

```

- This method runs only on the target and allocates char arrays from the heap and initializes each array with the length of the array. The presence of the length enables index range checking to verify that the bounds of the array are not being exceeded by application code.

Locke Labs
for the common defense

SoC Module – Interrupt Controller

HelloWorld.java | InterruptController.java

Source | History

```
75            GenlPurposeInputOutput.initialize();
76            DualModeTimer.initialize();
77            InterruptController.initialize();
78
```

- Lines 75 – 77 at left are from the Java main method. One of the methods, InterruptController.initialize, is shown below.

HelloWorld.java | InterruptController.java

Source | History

```java

    public static void setTimer2InterruptPriority()
    {
        // 0 is highest priority
        // following line is using a priority of 1
        DeviceMemory.writeInt(INTC_ILR_address_irq68, 0x04);
    }


    private static void enablePriorityThresholdMechanism()
    {
        // set priority of the current 'thread' to 7
        // this is 'lower' than priority of timer shown above
        DeviceMemory.writeInt(INTC_THRESHOLD_address, 0x07);
    }


    public static void initialize()
    {
        enablePriorityThresholdMechanism();
        setTimer2InterruptPriority();
        unmaskTimer2Interrupts();
        acknowledgeIRQ();
        enableInterrupts();
    }

```

- The code at left calls the DeviceMemory class to write the memory mapped control registers of the InterruptController.

Memory Access (1 of 3)

```java
 */
package lockelabs.jm.annotations;

/**
 *
 * @author jeff
 */
public @interface WholeMethodGeneration {

}
```

HelloWorld.java | DeviceMemory.java

Source | History

```java
package lockelabs.jm.memory.device;

import lockelabs.jm.annotations.WholeMethodGeneration;

/**
 *
 * @author jeff
 */
public class DeviceMemory {

    @WholeMethodGeneration
    public static int readInt(int address)
    {
        return 0;
    }

    @WholeMethodGeneration
    public static void writeInt(int address, int value)
    {
    }

}
```

- In the code at left, each method is decorated with the WholeMethodGeneration annotation.
- During the native code generation process, any method with this annotation is generated by reading one template for the entire method instead of iterating through the JVM instructions generated by the Java compiler for the method.
- An example of this process is shown on the next slide.

Locke Labs

for the common defense

Memory Access (2 of 3)

HelloWorld.java | GenerateInstructions.java | StackManager.java | GenerateInstruction.java

Source | History

```
    private static void wholeMethodGeneration(ArgList argList, FileWriter fw,
36          JavaClass javaClass, Method method, VelocityContext context)
37  {
38      try
39      {
40          String templateFullPath;
41          Template template;
42          VelocityContext localContext = new VelocityContext(context);
43
44          templateFullPath = argList.getMethodTemplateRootDirName()
45                  + File.separator + method.getName() + ".S";
46          template = Velocity.getTemplate(templateFullPath);
47
48          template.merge(localContext, fw);
49      }
        catch (Throwable t)
51      {
            t.printStackTrace();
53      }
54  }
55
56
57  public static void forMethod(ArgList argList, FileWriter fw,
58          JavaClass javaClass, Method method, StackFrame stackFrame,
59          VelocityContext context)
60  {
61
62      AnnotationEntry[] annotations = method.getAnnotationEntries();
63      String annotationName;
64      for (AnnotationEntry annotation : annotations)
65      {
66          annotationName = annotation.getAnnotationType();
67          if (annotationName.equals(
68                  "Llockelabs/jm/annotations/WholeMethodGeneration;"))
69          {
70              System.out.println("annotation type -> "
71                      + annotation.getAnnotationType());
72              wholeMethodGeneration(argList, fw, javaClass, method, context);
73              return;
```

Find: whole | Previous | Next

- Shown at line 57, forMethod generates the native assembly code for an embedded Java source code method.
- If the Java method has the WholeMethodGeneration annotation, the template is loaded and merged with the current context (shown in lines 36 – 54).
- The result is written to the ARM assembly file being generated for the current Java class.

Memory Access (3 of 3)

HelloWorld.java | writeInt.S | newarray.S | NewArray.java | HeapManager.java

Source | History

```

# manual code of write Int method

# read address from slot 0, per DeviceMemory javap output
LDR     $ocReg1, [$regVP, #-$slot0offset]

# read value from slot 1, per DeviceMemory javap output
LDR     $ocReg2, [$regVP, #-$slot1offset]

# write the value from src register, using base address and 0 offset
STR     $ocReg2, [$ocReg1]

# manual code of write Int method


# SAVE THE VP REGISTER IN TEMP LOCATION
MOV     $regTempVP, $regVP

# PREP THE SP TO RESTORE THE PREVIOUS FRAME
MOV     SP, $regFP

# RESTORE THE CALLER'S REGISTERS
LDMFD   SP!, {$frameRegs}

# RETURN SP TO END OF CALLER STACK, DROPPING CALLEE LOCAL VARIABLES
MOV     SP, $regTempVP

# BRANCH TO THE RETURN ADDRESS
BX      LR

```

- The template at left is not the whole method for writeInt.
- There is a preMethod template that implements stack overflow checks and initializes the stack frame for the current method.
- The native generation process needs some refactoring. You can see at left that each method is responsible for popping the current frame.
- Template variables are used here as well so that I can easily change register convention when needed.

Locke Labs

for the common defense

Interrupts (1 of 4)

```
LDR R0, =exc_vector_table
MCR p15, 0, R0, c12, c0, 0
```

During startup of the execution environment, initialize the ARM Vector Base Address Register

GenerateInstruction.java | exc.S

Source | History

```
.text
.balign 4096
#  LABEL(exc_vector_table)
.globl exc_vector_table
exc_vector_table:
    ldr pc, =exc1  /* Reset */
    ldr pc, =exc2  /* Undefined Instruction */
    ldr pc, =exc3  /* Supervisor Call */
    ldr pc, =exc4  /* Prefetch Abort */
    ldr pc, =exc5  /* Data Abort */
    ldr pc, =exc6  /* Hypervisor Call */
    ldr pc, =exc7  /* Interrupt - irq_entry */
    ldr pc, =exc8  /* Fast Interrupt */
```

- On the ARM MPU, when one of the exceptions (Reset, Data Abort, Interrupt from Interrupt Controller, etc) occurs, instruction execution jumps to the corresponding handler (Reset : exc1, Interrupt : exc7) by loading the PC with the address of that handler.

Interrupts (2 of 4)

HelloWorld.java | exc.S | readInt.S | GenerateInstructions.java | StackManager.java

Source | History

```
208
209  .text
210  .align 2
211  .globl exc7
212  .type exc7, %function
213  .code 32
214  exc7:
215          sub     lr, lr, #4          /* do the adjustment */
216          srsdb   sp!, #0x013         /* store LR and SPSR on SVC stack */
217          cps     #0x013              /* switch to SVC. */
218          push    {r0-r12, lr}
219
220          bl      $className.$interruptJavaHandler
221
222          dsb
223          pop     {r0-r12, lr}
224          rfeia   sp!
225
```

- The merged version of this file, which is part of the execution environment, is how Java source code is called when an interrupt occurs.

- The file shown above, exc.S, is the ‘template’ version.
- Line 220 is an example of two Velocity ‘variables’ in this template.
- At native code generation time, $className and $interruptJavaHandler are replaced with their actual names as supplied by Java source code. Example of this on the next slide, line 157 and method setMethodNames.

Locke Labs

for the common defense

Interrupts (3 of 4)

```java
    private static void installAnyInterruptHandlers(String outDirNode,
            ArgList argList, JavaClass javaClass, VelocityContext context,
            GenerationResult result) throws Exception
    {

        final String fileName = "exc";

        AnnotationEntry[] annotations = javaClass.getAnnotationEntries();
        String annotationName;
        for (AnnotationEntry annotation : annotations)
        {
            annotationName = annotation.getAnnotationType();
            if (annotationName.equals(
                    "Llockelabs/jm/annotations/armv7a/ArmExceptionHandlers;"))
            {
                System.out.println("annotation type -> "
                        + annotation.getAnnotationType());

                FileWriter fw = new FileWriter(outDirNode + File.separator +
                        fileName + ".S");

                String templateFullPath;
                Template template;
                VelocityContext localContext = new VelocityContext(context);

                setMethodNames(javaClass, annotation, localContext);

                templateFullPath = argList.getMethodTemplateRootDirName()
                        + File.separator + fileName + ".S";
                template = Velocity.getTemplate(templateFullPath);

                template.merge(localContext, fw);
                fw.close();
```

```java
 */
package lockelabs.jm.annotations.armv7a;

/**
 * ARM defines the 8 exception handlers listed b
 * same as Java exceptions.  These ARM handlers
 * interrupt handlers.
 * @author jeff
 */
public @interface ArmExceptionHandlers {

    String resetHandler();

    String undefinedHandler();

    String supervisorHandler();

    String prefetchHandler();

    String dataHandler();

    String hypervisorHandler();

    String interruptHandler();

    String fastIntHandler();

}
```

- Both of the files above are relevant only during native code generation.
- The file on the left illustrates reading the template exc.S and generating corresponding merged ARM assembly language file exc.S.

Locke Labs

for the common defense

Interrupts (4 of 4)

```java

// package lockelabs.jm.cpu.armv7a.exc;

@ArmExceptionHandlers(dataHandler = "dataHandler",
        fastIntHandler = "fastIntHandler",
        hypervisorHandler = "hypervisorHandler",
        interruptHandler = "interruptHandler",
        prefetchHandler = "prefetchHandler",
        resetHandler = "resetHandler",
        supervisorHandler = "supervisorHandler",
        undefinedHandler = "undefinedHandler")
public class InterruptHandlers {

    public static void interruptHandler ()
    {
        int irq = InterruptController.getActiveIRQ();
        if (InterruptController.timer2IRQ == irq)
        {
//              InterruptController.printInfo();

            Instance.printf(uart0BaseAddress, InterruptHandlerMessages.preMsg);
            Instance.printf(uart0BaseAddress, InterruptHandlerMessages.timerMsg);
            Instance.printf(uart0BaseAddress, eol);

            GenlPurposeInputOutput.advanceLED();

            DualModeTimer.clearPendingEvent();
            InterruptController.acknowledgeIRQ();
        }
        else
        {
            Instance.printf(uart0BaseAddress, InterruptHandlerMessages.preMsg);
            Instance.printf(uart0BaseAddress,
                    InterruptHandlerMessages.interruptHandlerMsg);
            Instance.printf(uart0BaseAddress, eol);
        }
    }

```

- This file was rearranged to place relevant details on one screen.
- This illustrates the Java source code that processes an interrupt.
- As shown in previous slides, the ArmExceptionHandlers annotation is used during native code generation to insert the name of the Java source methods to call from the execution environment.
- The method interruptHandler is the method invoked when the ARM MPU responds to the Interrupt Controller exception, the exc7 label on a previous slide.

Memory Limits – Heap (1 of 3)

HelloWorld.java | HeapManager.java | NewArray.java

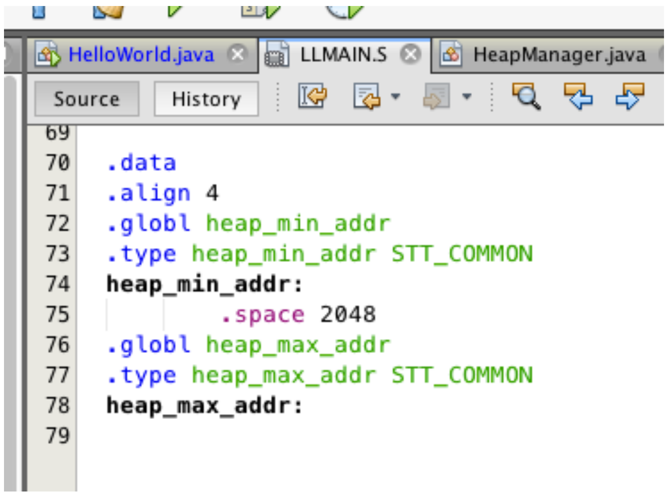
Source | History

```
116         public static int allocateBytes(int numberOfBytes)
117         {
118             // TODO - throw exc if out of heap memory
119             if ((nextFreeAddress + numberOfBytes) > heapEndAddress)
120             {
121                 throw new OutOfMemoryError();
122             }
123 
124             int currentFreeAddress = nextFreeAddress;
125             nextFreeAddress = nextFreeAddress + numberOfBytes;
126 
127 //              return 0;
128             return currentFreeAddress;
129         }
```

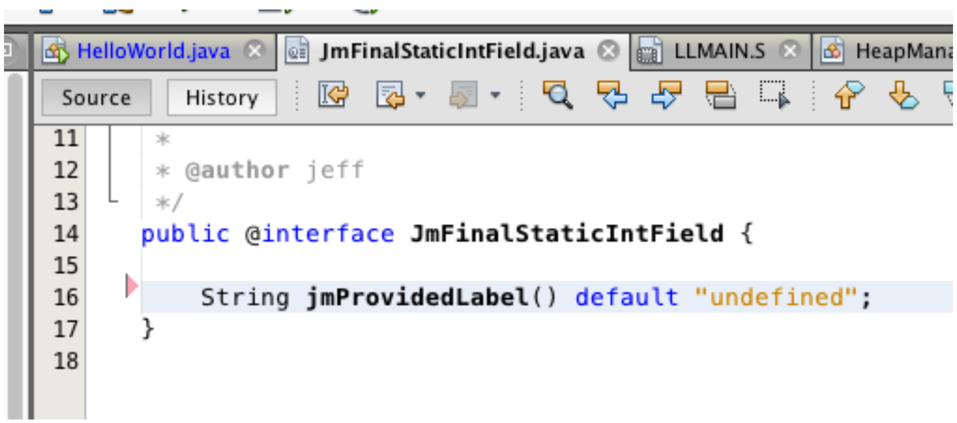
- HeapManager.allocateBytes, shown above, is called in the context of allocating a char array.
- On line 121 above, an OutOfMemoryError exception is thrown when a heap allocation fails.
- The current implementation of new and throwing exceptions is only a proof of concept that illustrates Java detection and halt when stack or heap allocation fails.
- Current implementations of new and throw are not presented.
- Implementation of the heapEndAddress is illustrated on following slides.

Locke Labs
for the common defense

Memory Limits – Heap (2 of 3)



- This slide and the following illustrate how the heap end address is currently integrated with Java source.
- LLMAIN shown at left is the entry point for the executable started by u-boot.
- On line 78, the end address of the heap is statically defined.



- The annotation at left is utilized to integrate the assembly language heap end address and the Java source references to the same address.

Memory Limits – Heap (3 of 3)

```java
@JmFinalStaticIntField(jmProvidedLabel = "heap_min_addr")
private static int heapBaseAddress = 0;
private static int nextFreeAddress = heapBaseAddress;

@JmFinalStaticIntField(jmProvidedLabel = "heap_max_addr")
private static int heapEndAddress = 100;
```

- During class init, the address of the provided label is stored in the annotated int.
- This is implemented with a 'special case' template for the putstatic JVM instruction.
- This special case drops the value provided by the class file and instead uses the address of the label provided by the annotation.

Locke Labs
for the common defense

Discussion

- Working towards a KickStarter campaign for June 2017
  - Prep and release all code as open source, considering a BSD 4-clause license
  - Write a book with 2 general topics
    - Step by step instructions for novice to reproduce the capabilities described here
    - Detailed description of design and implementation
  - Identify a stretch goal of a Java implementation of required GNU binutils. Primarily, as, ld, objcopy, objdump. Believe these would have to be released as GPL.
- Questions

Locke Labs
for the common defense

Backup Slides

Locke Labs
for the common defense

Classes Java main doesn't depend on?

- InterruptHandlers
- HeapManager
- HeapMemory
- NewArray

Native code generation visits all dependencies of Java main and generates corresponding native code. The classes above don't appear as dependencies and therefore have to be included manually in the list of classes to generate.

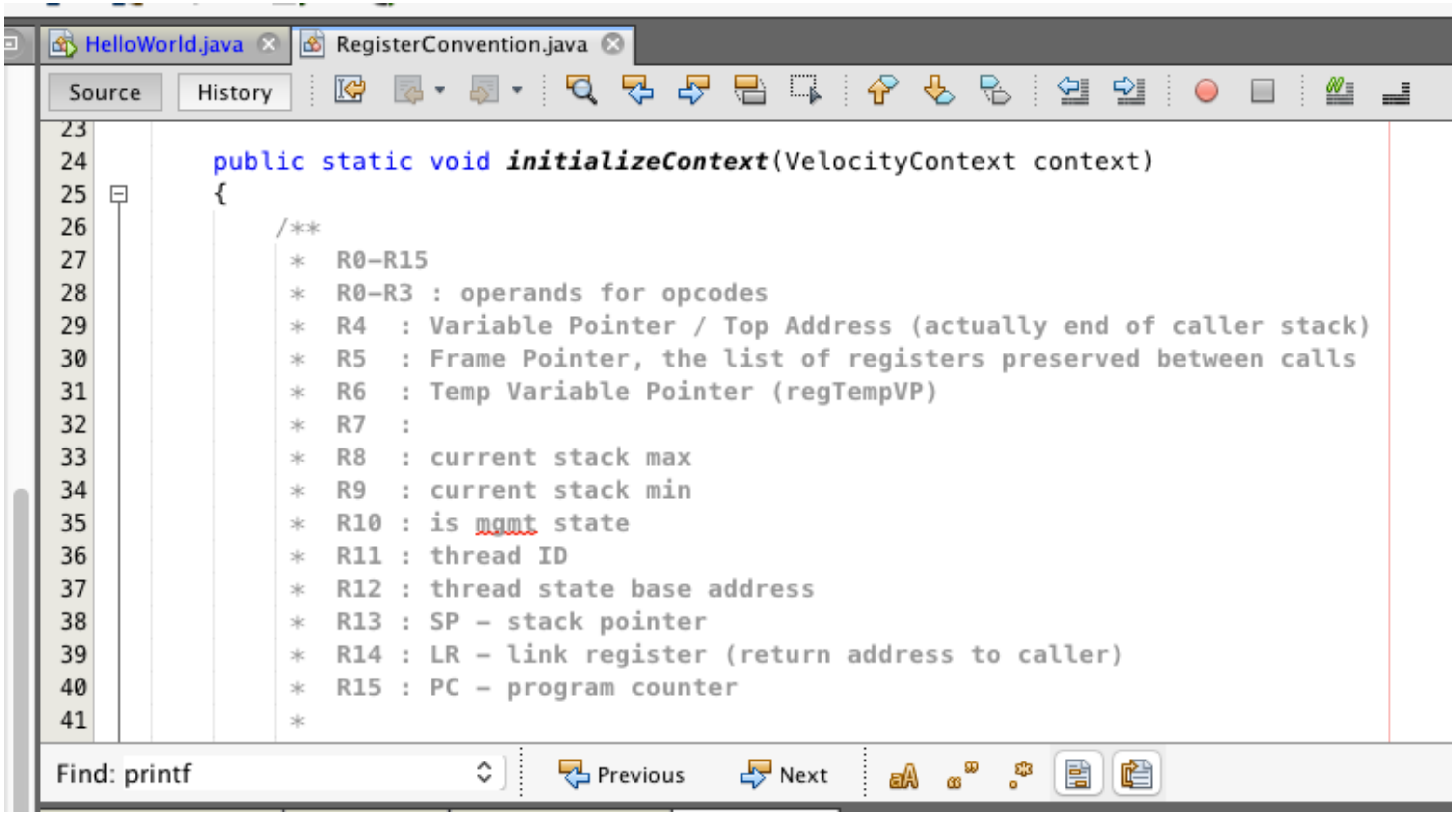
Locke Labs
for the common defense

How Much ‘Plain’ Assembly?

- LLMAIN.S – defines the method started by u-boot.
- exc.S – defines the exception vector table and initial service routines.
- jvmMain.S – invokes methods to initialize static data of classes and invokes Java main method.
- Files to read and write system control / status registers : flush caches to implement setting breakpoint, address of instruction fault, read and write int values to memory, etc.

Locke Labs
for the common defense

Register Convention




```java
    public static void initializeContext(VelocityContext context)
    {
        /**
         *  R0-R15
         *  R0-R3 : operands for opcodes
         *  R4  : Variable Pointer / Top Address (actually end of caller stack)
         *  R5  : Frame Pointer, the list of registers preserved between calls
         *  R6  : Temp Variable Pointer (regTempVP)
         *  R7  :
         *  R8  : current stack max
         *  R9  : current stack min
         *  R10 : is mgmt state
         *  R11 : thread ID
         *  R12 : thread state base address
         *  R13 : SP - stack pointer
         *  R14 : LR - link register (return address to caller)
         *  R15 : PC - program counter
         *
```

- Some of the above are notional at this point.
- R0-R3, R4, R5 are the ones listed most frequently in the examples.
- R0-R3 referenced as ocReg1 – ocReg3 in templates (oc = op code)
- R4, R5, and LR are currently pushed to stack between calls.

Locke Labs
for the common defense

Notional Stack Contents

zzz	R4 - VP - end of caller stack (Var Ptr)
aaa	
bbb	
ccc	
ddd	
eee	
fff	Local Variables
LR	
R5	R5 - FP - caller's local
R4	variables (Frame Ptr)
ggg	
hhh	
iii	
jjj	
kkk	
lll	
mmm	
nnn	SP - Callee local stack