MultiZone^{TM}Security

SDK

Document Version Rev 1.1

November 14, 2018

Table 1: Version History

Version	Date	Changes
1.0	Oct 26, 2018	Initial Release
1.01	Oct 26, 2018	Minor formatting and typography errors
1.1	Nov 14, 2018	Convert to Latex and substantial cleanup

Contents

Chapter 1

Introduction

This technical note describes how to build and run a secure application using MultiZone Security – the first Trusted Execution Environment (TEE) for RISC-V. It demonstrates the best practices of security through separation using a standard SiFive 32 or 64 bit RISC-V (E31 or E51 respectively) core supporting the privileged extensions V1.10. This implementation demonstrates the core features of MultiZone Security including: MultiZone nanoKernel, InterZone messenger, MultiZone Configurator and MultiZone Signed Boot.

1.1 Security Through Separation

Security through separation of duties is a classic, time-tested approach to protecting computer systems and the data contained therein. Security is the policy principle for protecting an asset. Separation was historically associated with "air-gapped" systems not interconnected by a network. In the context of this document, separation is a technical mechanism used to implement and maintain security. Separation may entail the use of different physical devices or other means, such as memory mapping. By separating and restricting the availability and use of assets, security is enforced according to prescribed policy.

It is often said that the only secure system is one that is not connected to any other system – and even then an "air gapped" system might be compromised by non-traditional means (e.g. Stuxnet virus compromise on Iranian uranium enrichment centrifuges used in nuclear reactors). However, in a world where much value is ascribed to the interconnection of systems to create networks – so called Internet of Things (IoT), a physically and logically isolated system is not very interesting to most people. This application note focuses on systems that can retain their security attributes even when connected to open networks.

For a detailed overview of these concepts – review the prpl Foundation Security Guidance Report at https://prplfoundation.org/documents/

1.2 RISC-V ISA Components Supporting Security Through Separation

The RISC-V ISA contains several features or "hooks" which enable security through separation to be implemented without the use of additional hardware components: Multiple Levels of Privilege – The RISC-V ISA defines four levels of privilege – the highest being machine mode (M), the next is reserved, followed by Supervisor Mode (S) with the lowest being User mode (U). Physical Memory Attributes (PMA) and Physical Memory Protection (PMP) – the RISC-V ISA includes a set of memory protection features in PMA and PMP which allow software operating in Machine Mode (M) to set limitations on the ranges of memory and memory mapped peripherals which can be accessed at lower levels of privilege. Trapping Functions Executions of invalid commands at S or U mode generate traps which can be intercepted at M mode and held or emulated back (Trap and Emulate) to the S or U mode code base by software running at M mode.

1.3 RISC-V – the Most Secure Computing Environment Ever.

The goal of separation used for security purposes is to create and preserve a trusted execution environment for an embedded system. Separation is intended to prevent exploitable defects in one zone from propagating to adjacent zones, or to the physical platform as a whole. Failures that occur in one zone are limited to that zone. Of course, when an adversary has a greater level of access, the challenge grows to fend off attacks. The greatest level of access is full physical access to the host system. Secure separation allows an embedded system to process sensitive data securely on behalf of client applications, and to continue doing so if one of the zones is compromised. Separation also enables protection across and between all subsystems of a system-on-a-chip within a unified memory architecture. This means protection covers not only the CPU, but also graphics processors, audio and video processors, communications subsystems, and other subsystems of the chip. The strategic goal of MultiZone Security is to achieve widespread adoption of trusted execution environments in RISC-V that are not limited to a single trusted computing domain, a single application environment, or to the CPU. With RISC-V we have the opportunity to make it the most Secure Computing Platform ever by proliferating these best practices to all applications rather than confining them to niche application where a regulatory framework demands a TEE.

1.4 MultiZone Security – The First Trusted Execution Environment for RISC-V

MultiZone Security implements a Trusted Execution Environment (TEE) using the hooks built into the standard RISC-V ISA. The objective is to enable system designers to implement a robust security environment without them having to be experts in security best practices or disrupt their development process or toolchain.

The design point for MultiZone Security is to separate sensitive functional blocks into independent zones and provide these zones with captive assets (ram, rom, i/o, interrupts) and communication with other zones via a secure InterZone messenger which uses no shared memory.

MultiZone Security differs from traditional TEEs in several key ways:

- Enables an unlimited number of equally secure zones – no concept of secure vs. non-secure
- Imposes a negligible cost on performance and memory - ¡1% of CPU cycles and ¡1kB of RAM
- Creates a signed boot structure by default
- Requires no modifications to existing code base
- Provides a high-performance API to securely delegate most privileged instructions
- Works with your existing toolchain and IDE – i.e. Eclipse and GNU command line tools
- Is formally verifiable – as the it is self-contained with no compiler or library dependencies

Typically an operating system or bare metal code would run one zone and key security functions are separated out into additional zones to prevent them from being compromised by the monolithic operating system code base which is subject to frequent vulnerability discovery and patching.

1.5 MultiZone Security – The First Trusted Execution Environment for RISC-V

MultiZone Security allows developers to properly implement robust Trusted Execution Environment (TEE) through a simple and intuitive process:

1. Compile and link individual functional blocks for each zone using your existing IDE and toolchain (examples are provided with the Eclipse IDE and GNU command line tools
   (a) Optionally include the MultiZone header to access APIs from the nanoKernel such as sending and receiving messages between zones, registering interrupt handers and yielding the zone when there is no work to be done.
2. Assign resources to each Zone in the MultiZone Configuration file
   (a) Up to (6) ranges of physical Memory mapped resources per Zone – ram, rom, i/o
      i. Range 1 is for ROM – the program counter points to this base address when the Zone starts
      ii. Range 2 is typically used for RAM
      iii. Range 3-6 are typically used peripherals
   (b) Define any combination of Read / Write / Execute policy for each individual range
      i. ROM would normally have [R]ead and e[X]ecute privileges as fixed variables are loaded from rom along with code be executed
      ii. RAM would normally have [R]ead and [W]rite privileges only
      iii. Peripherals would normally have [R]ead only or [R]Read and [W]rite privileges

(c) Range 1 and 2 may have any base address and any size that is a multiple of 4 Bytes; this is done as RAM is often the most scarce resource in a system and cannot be allocated efficiently pages at a time.

(d) Range 3-6 need to be naturally aligned power of two (NAPOT) – meaning the base address must be a multiple of the sizeInterrupts are assigned to each Zone – PLIC and CLINT

(e) The tick time for the preemptive scheduler is set – this is the maximum time (in ms) a Zone operates before it is preempted by the scheduler, the Zone can release control earlier with use of a Yield() command. A setting of 0 ms disables the preemptive scheduler which means context switches between Zones only occur on Yield() commands (ie cooperative scheduling).

```
# Copyright(C) 2018 Hex Five Security, Inc. - All Rights Reserved

# Kernel
tick  = 10 # ms

# Zone 1
mz1_fence = FENCE
mz11_base = 0x40410000; mz11_size = 64K; mz11_rwx = RX # FLASH
mz12_base = 0x80001000; mz12_size =  4K; mz12_rwx = RW # RAM
mz13_base = 0x20000000; mz13_size =  32; mz13_rwx = RW # UART

# Zone 2
mz2_irq = 11, 21, 22 # BTN0 BTN1 BTN2
mz21_base = 0x40420000; mz21_size = 64K; mz21_rwx = RX # FLASH
mz22_base = 0x80002000; mz22_size =  4K; mz22_rwx = RW # RAM
mz23_base = 0x0200BFF8; mz23_size =   8; mz23_rwx = RO # RTC
mz24_base = 0x20005000; mz24_size =  64; mz24_rwx = RW # PWM
mz25_base = 0x20002000; mz25_size =  64; mz25_rwx = RW # GPIO
mz26_base = 0x0C000000; mz26_size =  4M; mz26_rwx = RW # PLIC

# Zone 3
mz3_irq = 23 # BTN3
mz31_base = 0x40430000; mz31_size = 64K; mz31_rwx = RX # FLASH
mz32_base = 0x80003000; mz32_size =  4K; mz32_rwx = RW # RAM
mz33_base = 0x0200BFF8; mz33_size =   8; mz33_rwx = RO # RTC
mz34_base = 0x20002000; mz34_size =  64; mz34_rwx = RW # GPIO
```

Fig. 1 This is the multizone.cfg used in the Evaluation SDK, you can find by opening the hexfive-multizone Project or Directory. It is hard coded in this evaluation version – meaning that changes to this file will not be reflected in the Configuration. In the licensed commercial version this configuration is updatable.

It is possible to overlap memory regions in order to share resources (as you can see from the real time clock shared between Zone 2 and 3 in Fig. 1. In this case it is a read-only resource, thus it does not increase the attack surface of these zones. Sharing writeable resources is allowed, but can undermine the separation model of the TEE so this practice is not recommended.

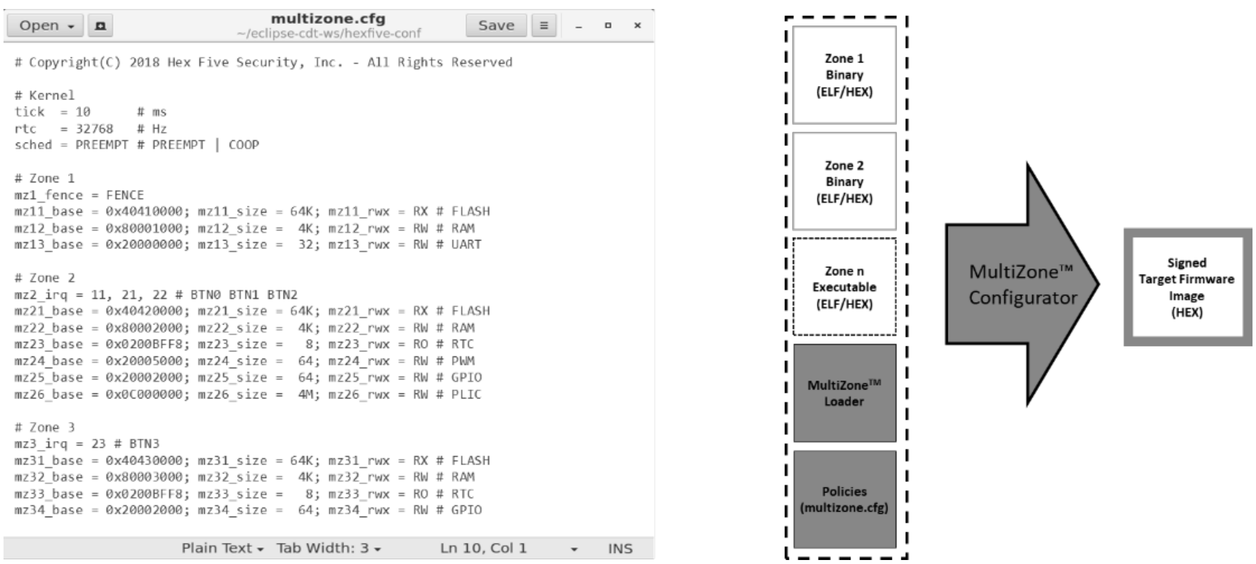


Fig. 2 MultiZone Configurator flow showing (3) pre-compiled and linked Zone binaries coming together with the multizone.cfg file into a signed target firmware image (HEX).

Chapter 2

MultiZone Security SDK - Demo Application

This describes the demonstration application included with the MultiZone Security Evaluation SDK. The system architecture is comprised of three separate bare-metal applications, each running in its own zone. The MultiZone nanoKernel enforces hardware-level separation of CPU and memory-mapped resources, policy-based access to I/O peripherals, and user mode execution of interrupt handlers. The InterZone Messenger provides secure communications across the three zones.

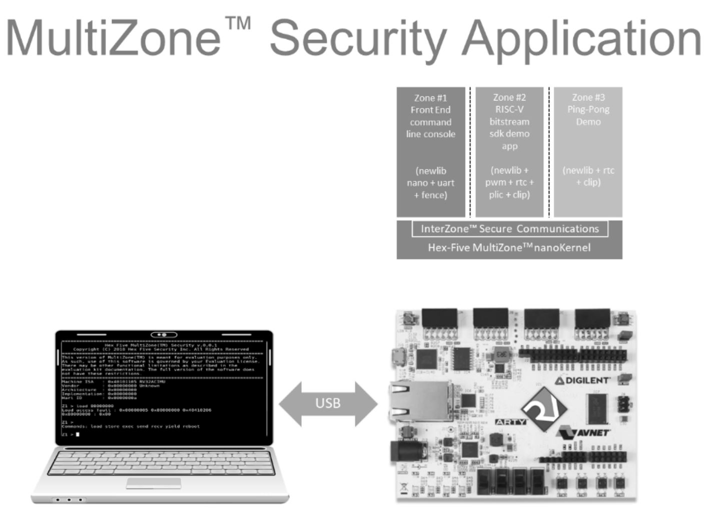


Fig. 3. Demonstration Application shipped with the MultiZone Security Evaluation SDK consisting of (3) bare metal applications each running in independent Zones with separate resources and code running in user mode.

Zone 1 connects to the host PC via UART over USB at 115,200 baud 8N1. Operating in Zone 1 is an ANSI terminal application which presents the user with a command line interface to send and receive messages from other zones, exercise the assets assigned to Zone 1, challenge the enforcement capabilities of the nanoKernel with discrete load, store and exec commands, displays kernel performance statistics, and demonstrate cooperative behavior by yielding the execution context to the other zones.

Zone 2 is a slightly modified version of the SiFive coreplexip welcome demo which uses the realtime clock and PWM peripheral to drive LED LD1 through the rgb color pallet. It also has three CLINT interrupts mapped to buttons (BTN0, 1, 2) which cause LED LD1 to change color for 5 seconds and send a message back to Zone 1 using the InterZone messenger.

Zone 3 causes the GPIO connected LED LD0 to flash once per second, the color of flash can be rotated by pressing BTN3, mapped to a PLIC interrupt, or by sending messages from zone 1. It will also respond to a "ping" message sent by zone 1 with a "pong" response.

2.1 Operating the MultZone Security SDK Demo Application

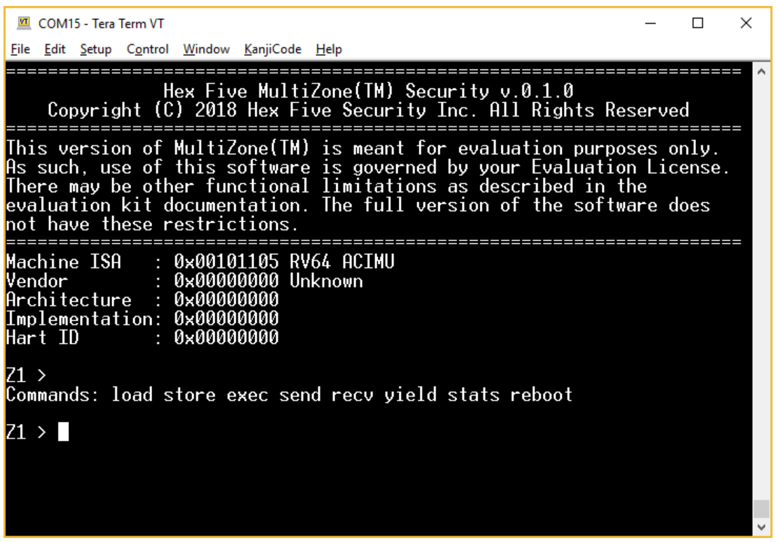


Fig. 4. Serial terminal window connected to Zone 1. Press Enter on the window to get a list of commands that you can issue to Zone 1.

You may issue discrete load, store and exec commands inside Zone 1 to test the memory protection provided by MultiZone Security as noted in table below. The memory map is available by expanding the hexfive-multizone Project and double clicking multizone.cfg. As noted earlier, the zone configuration is locked in the Evaluation version of MultiZone Security, thus edits to this file will not have any effect.

N.B. The memory map for each board / core is different, consult the appendix to verify the actual multizone.cfg file for your version.

```
# Copyright(C) 2018 Hex Five Security, Inc. - All Rights Reserved

# Kernel
tick  = 10 # ms

# Zone 1
mz1_fence = FENCE
mz11_base = 0x40410000; mz11_size = 64K; mz11_rwx = RX # FLASH
mz12_base = 0x80001000; mz12_size =  4K; mz12_rwx = RW # RAM
mz13_base = 0x20000000; mz13_size =  32; mz13_rwx = RW # UART

# Zone 2
mz2_irq = 11, 21, 22 # BTN0 BTN1 BTN2
mz21_base = 0x40420000; mz21_size = 64K; mz21_rwx = RX # FLASH
mz22_base = 0x80002000; mz22_size =  4K; mz22_rwx = RW # RAM
mz23_base = 0x0200BFF8; mz23_size =   8; mz23_rwx = RO # RTC
mz24_base = 0x20005000; mz24_size =  64; mz24_rwx = RW # PWM
mz25_base = 0x20002000; mz25_size =  64; mz25_rwx = RW # GPIO
mz26_base = 0x0C000000; mz26_size =  4M; mz26_rwx = RW # PLIC

# Zone 3
mz3_irq = 23 # BTN3
mz31_base = 0x40430000; mz31_size = 64K; mz31_rwx = RX # FLASH
mz32_base = 0x80003000; mz32_size =  4K; mz32_rwx = RW # RAM
mz33_base = 0x0200BFF8; mz33_size =   8; mz33_rwx = RO # RTC
mz34_base = 0x20002000; mz34_size =  64; mz34_rwx = RW # GPIO
```

The available command format and syntax for the Zone 1 terminal is:

Command	Syntax and function	Example

load	load [address] – where address is a physical memory address without the 0x header.
	Completes a byte load from the address listed, if the address is not within the Zone 1 memory map it will return an exception
		`Z1> load 80001000 [valid]`
		`0x80001000 : 0x0c`
		
		`Z1> load 80000FFF [invalid]`
		`Load access fault : 0x00000005..`
		
		`The format of a load address fault is:`
		`Fault type | Attempted address | Program Pointer Location`

store	Store [address] [value] – where address is a physical memory address without the 0x; value is a byte (eg aa), a half-word (eg aabb) or a word (eg aabbccdd)
	When storing a byte, the byte store instruction is used and no alignment is required; when storing a half word the half-word store instruction is used and alignment must be to a half word, when storing a word, the word store instruction is used and alignment must be to the word.
		`Z1> store 80001000 aabbccdd`
		`0x80001000 : 0xaabbccdd`
		
		`Z1> store 80001001 aabb`
		`Store/AMO address misaligned :`
		
		`Z1> store 8000FFE aabb`
		`Store access fault : 0x00000007`
		`[Outside Address Range]`

exec	Exec [address] – where the address is a physical memory address without the 0x header.
	This is equivalent to a jump command.
		`Z1> exec 40410000 [rom start]`
		`- Causes Zone 1 to reboot`
		
		`Z1> exec 40410004`
		`[Valid address, but not valid instructions here as is not the starting address so Zone 1 will hang.  Reset to Recover`
		
		`Z1> exec 80001000`
		`Instruction access fault : ...`
		`[No e[X]ecute privilege in RAM]`

Command	Syntax and function	Example

send	send [Zone #] message
	Sends a message to another zone – in this application, zone 1 and zone 3 are actively listening for messages and will respond to the examples shown.
		`Z1> send 3 r | b | g`
		`[changes the color of LED LD0]`
		
		`Z1> send 3 ping`
		`Z3> pong`

yield	yield – releases context from zone 1 and measures the amount of elapsed time (in us) until context returns to zone 1.	`Z1> yield`
		`yield : elapsed time 17us`
		`[Zones 2 & 3 are configured`
		`to yield by default]`

stats	Stats – completes a set of 11 yield commands, measures can calculates context switch time in cycles and microseconds	`Z1> stats`
		`1354 cycles in 20 us`
		`1091 cycles in 16 us`
		`1234 cycles in 18 us`
		`1085 cycles in 16 us`
		`1223 cycles in 18 us`
		`1088 cycles in 16 us`
		`1222 cycles in 18 us`
		`1318 cycles in 20 us`
		`1223 cycles in 18 us`
		`1085 cycles in 16 us`
		`1227 cycles in 18 us`
		`---------------------`
		`cycles  min/med/max = ...`
		`time    min/med/max = ...`
		`ctx sw instr  min/med/max = ...`
		`ctx sw cycles min/med/max = ...`
		`ctx sw time   min/med/max = ...`

Command	Syntax and function	Example

reboot	reboot – jumps to the starting flash address of zone 1 to reboot zone 1, equivalent to issuing exec 40410000	`Z1> reboot`
		`==================...`
		`Hex Five MultiZone(TM)...`
		`Copyright (C) 2018 Hex Five ...`
		`==================...`
		`...`
		`Z1>`

2.2 Common Elements of all Zones

In this demonstration application, all zones take advantage of the Multizone library by including libhexfive.h in main.c – this provides access to the MultiZone APIs (see MultiZone API section for more detail)

```
/* Copyright(C) 2018 Hex Five Security, Inc. - All Rights Reserved */

#include <fcntl.h>
. . .
#include <libhexfive.h>
```

2.3 Items of Note in Zone #1

One of the features of Zone 1 is to enable command line testing of the PMA and PMP functions, when invalid accesses are issued these generate exceptions that are trapped in the nanoKernal. The Zone may register an exception handler to provide feedback to the user:

```
void trap_0x5_handler(void)__attribute__((interrupt("user")));
void trap_0x5_handler(void){

int msg[4]={0,0,0,0};
ECALL_RECV(1, msg);
printf("Load access fault : 0x%08x 0x%08x 0x%08x \n", msg[0], msg[1], msg[2]);
}
```

The definition of these exceptions is shown in the RISC-V Privileged Architectures V1.1, Table 3.6. You can register an exception handler against multiple exceptions; however in this case as the output text is different using different exception handlers for each is a more performant solution. Calls to privileged functions can be done in two ways as shown in the example that reads the ISA ID register. They can either be made directly as a privileged call as they would in an application running in machine mode or then can be made using on of the MultiZone APIs (commented out in this example). In the privileged call case, the call is trapped by the nanoKernel, validated, executed and emulated back to the Zone. This works, but is less performant than simply using the MultiZone API call.

```
// ----------------------------------------------------------------------
void print_cpu_info(void) {
// ----------------------------------------------------------------------

// misa
uint64_t misa = 0x0; asm ( "csrr %0, misa" : "=r"(misa) );
//const uint64_t misa = ECALL_CSRR_MISA();
```

To interact with the user, Zone 1 runs a simple loop which performs the following functions: a. Checks the UART and manages cursor, backspace and other commands b. Checks for incoming messages from Zone 3 and prints them

```
// poll & print incoming messages
int msg[4]={0,0,0,0};

ECALL_RECV(3, msg);

if (msg[0]){

write(1, "\e7", 2); // save curs pos
write(1, "\e[2K", 4); // 2K clear entire line - cur pos dosn't change

switch (msg[0]) {
case 1   : write(1, "\rZ3 > USB DEVICE ATTACH VID=0x1267 PID=0x0000\r\n", 47); break;
case 2   : write(1, "\rZ3 > USB DEVICE DETACH\r\n", 25); break;
case 331 : write(1, "\rZ3 > CLINT IRQ 23 [BTN3]\r\n", 27); break;
case 'p' : write(1, "\rZ3 > pong\r\n", 12); break;
default  : write(1, "\rZ3 > ???\r\n", 11); break;
}
```

Checks for incoming messages from Zone 2 and prints them

```
ECALL_RECV(2, msg);
if (msg[0]){
    write(1, "\e7", 2); // save curs pos
    write(1, "\e[2K", 4); // 2K clear entire line - cur pos dosn't change
    switch (msg[0]) {
      case 201 : write(1, "\rZ2 > PLIC  IRQ 11 [BTN0]\r\n", 27); break;
      case 211 : write(1, "\rZ2 > CLINT IRQ 21 [BTN1]\r\n", 27); break;
      case 221 : write(1, "\rZ2 > CLINT IRQ 22 [BTN2]\r\n", 27); break;
      default  : write(1, "\rZ2 > ???\r\n", 11); break;
    }
}
```

Yields context to the next Zone (in this case Zone 2)

```
ECALL_YIELD();
```

In main() two test options are shown and commented out The first one simulates a locked up Zone and forces the nanoKernal to preempt Zone 1 and force a context switch based on the defined tick time (10ms). The second one immediately yields Zone 1 and allows for measurement of context switching performance.

```
int main (void) {
// ------------------------------------------------------------------------

//volatile int w=0; while(1){w++;}
//while(1) ECALL_YIELD();
```

Next the exception handlers are registered using MultiZone APIs

```
ECALL_TRP_VECT(0x0, trap_0x0_handler); // 0x0 Instruction address misaligned
ECALL_TRP_VECT(0x1, trap_0x1_handler); // 0x1 Instruction access fault
ECALL_TRP_VECT(0x2, trap_0x2_handler); // 0x2 Illegal Instruction
ECALL_TRP_VECT(0x4, trap_0x4_handler); // 0x4 Load address misaligned
ECALL_TRP_VECT(0x5, trap_0x5_handler); // 0x5 Load access fault
ECALL_TRP_VECT(0x6, trap_0x6_handler); // 0x6 Store/AMO address misaligned
ECALL_TRP_VECT(0x7, trap_0x7_handler); // 0x7 Store access fault
```

The elapsed cycles time for a yield relies on reading MCYCLE which is a privileged register – it is shown in two different methods – via the MultiZone API and via a direct ASM call which is commented out

```
} else if (tk1 != NULL && strcmp(tk1, "yield")==0){

    const int MHZ = 64995; //64952;  // 64951956

    //const uint64_t C0 = ECALL_CSRR_MCYCLE();
    const uint64_t C1 = ECALL_CSRR_MCYCLE();
    ECALL_YIELD();
    const uint64_t C2 = ECALL_CSRR_MCYCLE();
    const int C = (C2-C1)*1000/MHZ;
/*  uint32_t r1, r2, r3, r4;
    asm ("li a0, 6; ecall; mv %0, a1; mv %1, a0;" // ECALL_CSRR_MCYCLE()
       "li a0, 0; ecall;                        " // ECALL_YIELD();
       "li a0, 6; ecall; mv %2, a1; mv %3, a0;" // ECALL_CSRR_....
       : "=r"(r1), "=r"(r2), "=r"(r3), "=r"(r4)::"a0","a1");

    const uint64_t C1 = (uint64_t)r1<<32 | r2;
    const uint64_t C2 = (uint64_t)r3<<32 | r4;

    const int C = (C2-C1-(C1-C0))*1000/MHZ;
*/
```

2.4 Items of Note in Zone #2

Zone 2 is designed to show how an existing application – in this case the SiFive corplexip welcome code can be dropped into a zone without modification and simply run in user mode and by trapping an emulating necessary privileged instructions. The first item of note is that the UART functionality from the coreplexipwelcome code has a UART defined and interacts with it. In the MultiZone Configuration, the UART peripheral is not assigned to Zone 2, thus these command generate exceptions; since there is no handler registered in Zone 2 for these exceptions, they end up doing nothing. The first three buttons are tied to local interrupts which cause LED LD1 to change color for 5 seconds. This code shows how to create a user mode interrupt handler, then in main() how to register that handler against a specific interrupt. You can also see how Zone2 sends a message to Zone 1 in the interrupt handler.

```
void button_0_handler(void)__attribute__((interrupt("user")));
void button_0_handler(void){ // global interrupt

    ECALL_SEND(1, (int[4]){201,0,0,0});

    plic_source int_num  = PLIC_claim_interrupt(&g_plic); // claim
```

```
    LED1_GRN_ON; LED1_RED_OFF; LED1_BLU_OFF;

    volatile uint64_t *  now = (volatile uint64_t*)(CLINT_CTRL_ADDR + CLINT_MTIME);
    volatile uint64_t then = *now + 3*32768;
    while (*now < then) ECALL_YIELD();

    LED1_RED_OFF; LED1_GRN_OFF; LED1_BLU_OFF;

    GPIO_REG(GPIO_RISE_IP) |= (1<<BUTTON_0_OFFSET); //clear gpio irq

    PLIC_complete_interrupt(&g_plic, int_num); // complete
}
```

```
/*configures Button1 as a local interrupt*/
void b1_irq_init()  {

    //dissable hw io function
    GPIO_REG(GPIO_IOF_EN )    &=  ~(1 << BUTTON_1_OFFSET);

    //set to input
    GPIO_REG(GPIO_INPUT_EN)   |= (1<<BUTTON_1_OFFSET);
    GPIO_REG(GPIO_PULLUP_EN)  |= (1<<BUTTON_1_OFFSET);

    //set to interrupt on rising edge
    GPIO_REG(GPIO_RISE_IE)    |= (1<<BUTTON_1_OFFSET);

    //enable the interrupt
    ECALL\_IRQ\_VECT(16+LOCAL\_INT\_BTN\_1, button\_1\_handler); // set\_csr....
}
```

The interrupt service routine in this case stalls for 5 seconds (which is obviously not a typical design point), but illustrates how an ISR can be pre-empted by another interrupt. If button_0_handler is operating and b1 is pressed, button_0_handler is pushed onto the stack inside the zone and button_1_handler executes; once button_1_handler is complete, button_0_handler finishes then Zone 2 returns to its normal operation – wherever the program counter was pointed to prior to the button_0 being pressed.

2.5 Items of Note in Zone #3

Zone 3 implements an interrupt handler for button 3 that rotates the color of the flashing LED LD0 from Green to Blue to Red. This is very similar to the interrupt handler in Zone 2.

```
void button_3_handler(void)__attribute__((interrupt("user")));
void button_3_handler(void){ // local interrupt

    ECALL_SEND(1, ((int[]){331,0,0,0}));

    GPIO_REG(GPIO_OUTPUT_VAL)=0;

    led = (led==GREEN_LED_OFFSET ? BLUE_LED_OFFSET :
        led==BLUE_LED_OFFSET ? RED_LED_OFFSET :
        GREEN_LED_OFFSET);

    volatile uint64_t * now, then;
    now = (volatile uint64_t*)(CLINT_CTRL_ADDR + CLINT_MTIME);
    then = *now + 500*32768/1000;
    while (*now < then) ECALL_YIELD();

    GPIO_REG(GPIO_RISE_IP) |= (1<<BUTTON_3_OFFSET);
}
```

However, Zone 3 is also able to receive messages from Zone 1 to change the LED color and shows an implementation of this functionality. Messages are of fixed size and each zone has a separate inbox for every other Zone. Thus Zone 3 is ONLY listening to Zone 1, there is no way for Zone 1 to overflow the message buffer or send messages that could cause harm to Zone 3 because Zone 3 only responds to specific messages:

- R – change LED to RED
- G – change LED to Green
- B – change LED to blue
- Anything else – send a message to Zone 1 with “Pong”

```
while(1){
GPIO_REG(GPIO_OUTPUT_VAL) ^= (0x1 << led);
    const uint64_t timeout = ECALL_CSRR_MTIME() + (GPIO_REG(GPIO_OUTPUT_VAL) & (0x1 << led)
    while (ECALL_CSRR_MTIME() < timeout){
        int msg[4]={0,0,0,0}; ECALL_RECV(1, msg);
        if (msg[0]){
  switch (msg[0]) {
            case 'r': led = RED_LED_OFFSET;   break;
```

```
            case 'g': led = GREEN_LED_OFFSET; break;
            case 'b': led = BLUE_LED_OFFSET;  break;
            default: ECALL_SEND(1, msg);    break; // echo
          }
        }
    ECALL_YIELD();
}
} // While (1)
```

Chapter 3

SiFive E31 & E51 Cores on Xilinx A7 Arty Instructions

3.1 Quickstart

Pre-requisites for using this Quickstart

- Follow SiFive Freedom E310 Arty FPGA Dev Kit Getting Started Guide
- Upload one of the following bitstreams to the FPGA
  - SiFive E31 Core v3p0 (RISC-V RV32ACIMU)
  - SiFive E51 Core v3p0 (RISC-V RV64ACIMU)
- Get the hardware up and running and able to import, build, upload and debug the SiFive coreplexip welcome project

MultiZone Quickstart - Freedom Studio (Eclipse) for Windows and Linux

1. Download and import the multizone-freedomstudio project - https://github.com/hex-five/multizone-freedomstudio
2. File ¿ Import ¿ General ¿ Existing Projects into Workspace
   (a) select archive file MultiZone.zip
   (b) select and import all four projects:
      i. hexfive-multizone
      ii. hexfive-zone1
      iii. hexfive-zone2
      iv. hexfive-zone3
3. Windows user: add '.exe' to external tool configuration for multizone RV32 and multizone RV64
   (a) Run - External Tools ¿ External Tool Configurations

   (b) Edit Main - Location eclipse-home../jre/bin/java.exe

4. To build & upload sample as is:
   (a) Select projects: hexfive-zone1, hexfive-zone2, hexfive-zone3 ¿ Right click ¿ Clean
   (b) Click Build dropdown ¿ Select either RV32 or RV64 for E31 or E51)
   (c) Builds the zone binaries
   (d) Click Run External Drop Down, select RV32 or RV64
   (e) Runs the Configurator to merge Zone Binaries into a signed HEX file
   (f) Click Run or Debug, select RV32 or RV64
   (g) Uploads the HEX file via JTAG and starts it Running or Debugging

3.2 Detailed Startup Instructions - Freedom-e-sdk (Linux)

Installation

Prerequisites:

- https://github.com/sifive/freedom-e-sdk (see relative install notes)
- java jre 1.8

```
git clone https://github.com/hex-five/multizone-freedom-e-sdk.git
```

- rename folder multizone-freedom-e-sdk to multizone_security
- move folder multizone_security to /freedom-e-sdk/software/multizone_security

Usage

```
cd ~/freedom-e-sdk
to clean: make clean PROGRAM=multizone_security BOARD=coreplexip-e31-arty
to build: make software PROGRAM=multizone_security BOARD=coreplexip-e31-arty
to upload & run: make upload PROGRAM=multizone_security BOARD=coreplexip-e31-arty
to debug: open two terminal sessions:
session 1: make run\_openocd BOARD=coreplexip-e31-arty
session 2: make run\_gdb PROGRAM=multizone_security BOARD=coreplexip-e31-arty

(gdb) add-symbol-file ./software/multizone_security/zone1/zone1.elf 0x40410000
(gdb) break main
(gdb) info local
(gdb) ctrl-c
(gdb) continue

Notes
```

```
to debug zone2: (gdb) add-symbol-file ./software/multizone_security/zone2/zone2.elf 0x404200
to debug zone3: (gdb) add-symbol-file ./software/multizone_security/zone3/zone3.elf 0x404300
```

Ubuntu 18.04: The prebuilt toolchain provided by SiFive doesn't work with Ubuntu 18.04. You can either build the one included in the repo (approx 20 minutes build time) or point to the one packaged with FreedomStudio. Then make sure the environment points to these folders:

```
export RISCV_PATH=/home/hexfive/riscv64-unknown-elf-gcc-20180928-x86_64-linux-centos6
export RISCV_OPENOCD_PATH=/home/hexfive/riscv-openocd-20180928-x86_64-linux-centos6
```

If java is not installed in your system:

```
sudo apt install openjdk-8-jre-headless
```

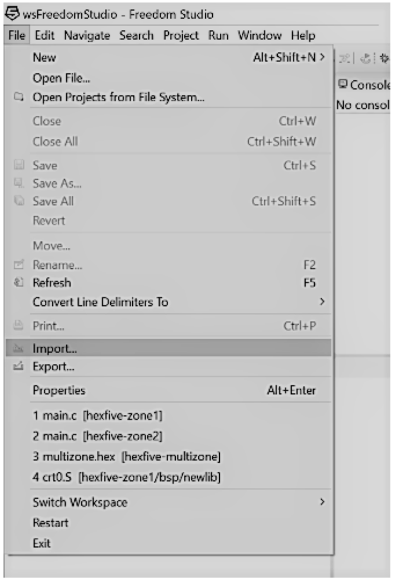
3.3 Detailed Startup Instructions - Freedom Studio

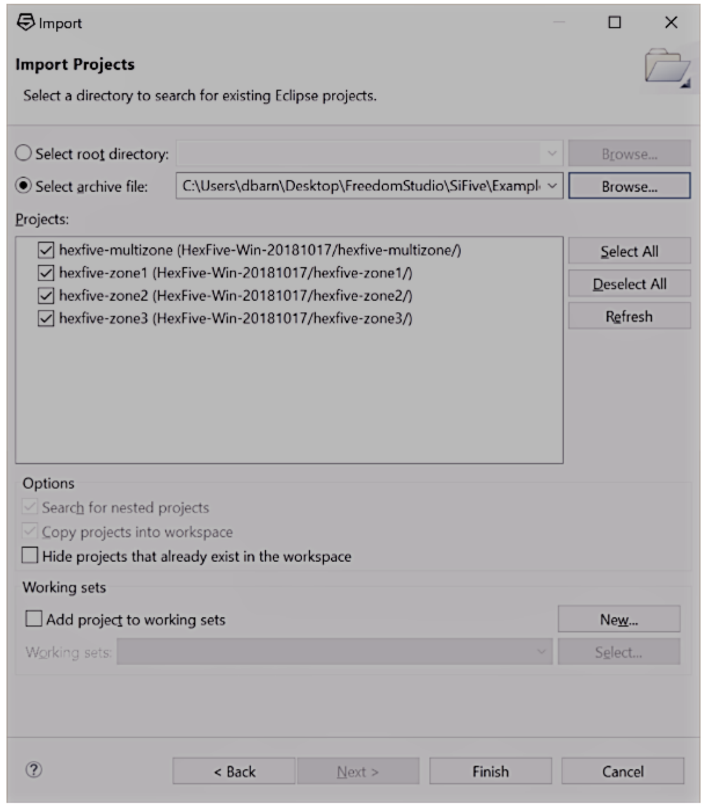
Pre-requisites for using this startup workflow

- Follow SiFive Freedom E310 Arty FPGA Dev Kit Getting Started Guide
- Upload one of the following bitstreams to the FPGA
  - SiFive E31 Core v3p0 (RISC-V RV32ACIMU)
  - SiFive E51 Core v3p0 (RISC-V RV64ACIMU)
- Get the hardware up and running and able to import, build, upload and debug the SiFive coreplexip_welcome project

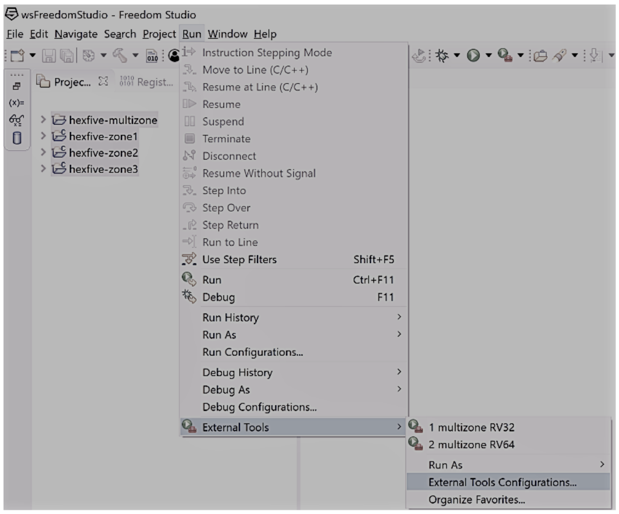
MultiZone Startup Instructions - Freedom Studio (Eclipse) for Windows and Linux

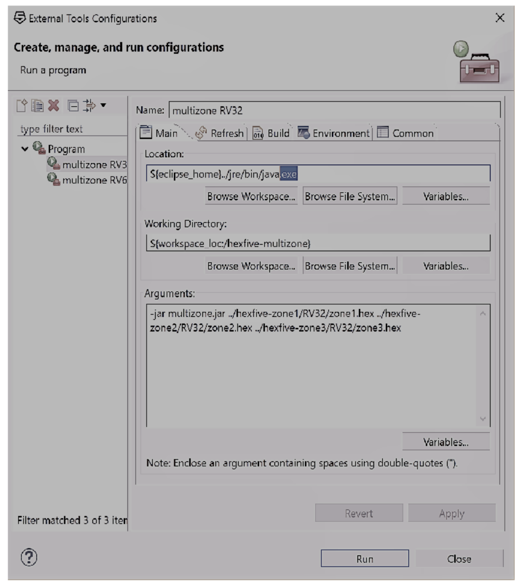
1. Download and import the multizone-freedomstudio project - https://github.com/hex-five/multizone-freedomstudio
2. File - Import - General - Existing Projects into Workspace
   (a) select archive file MultiZone.zip
   (b) select and import all four projects:
      i. hexfive-multizone
      ii. hexfive-zone1
      iii. hexfive-zone2
      iv. hexfive-zone3


wsFreedomStudio - Freedom Studio
File Edit Navigate Search Project Run Window Help
New Alt+Shift+N
Open File...
Open Projects from File System...
Close Ctrl+W
Close All Ctrl+Shift+W
Save Ctrl+S
Save As...
Save All Ctrl+Shift+S
Revert
Move...
Rename... F2
Refresh F5
Convert Line Delimiters To
Print... Ctrl+P
Import...
Export...
Properties Alt+Enter
1 main.c [hexfive-zone1]
2 main.c [hexfive-zone2]
3 multizone.hex [hexfive-multizone]
4 crt0.S [hexfive-zone1/bsp/newlib]
Switch Workspace
Restart
Exit


Import
Import Projects
Select a directory to search for existing Eclipse projects.
Select root directory:
Browse...
Select archive file:
C:\Users\dbarn\Desktop\FreedomStudio\SiFive\Exampl
Browse...
Projects:
hexfive-multizone (HexFive-Win-20181017/hexfive-multizone/)
hexfive-zone1 (HexFive-Win-20181017/hexfive-zone1/)
hexfive-zone2 (HexFive-Win-20181017/hexfive-zone2/)
hexfive-zone3 (HexFive-Win-20181017/hexfive-zone3/)
Select All
Deselect All
Refresh
Options
Search for nested projects
Copy projects into workspace
Hide projects that already exist in the workspace
Working sets
Add project to working sets
New...
Working sets:
Select...
< Back
Next >
Finish
Cancel

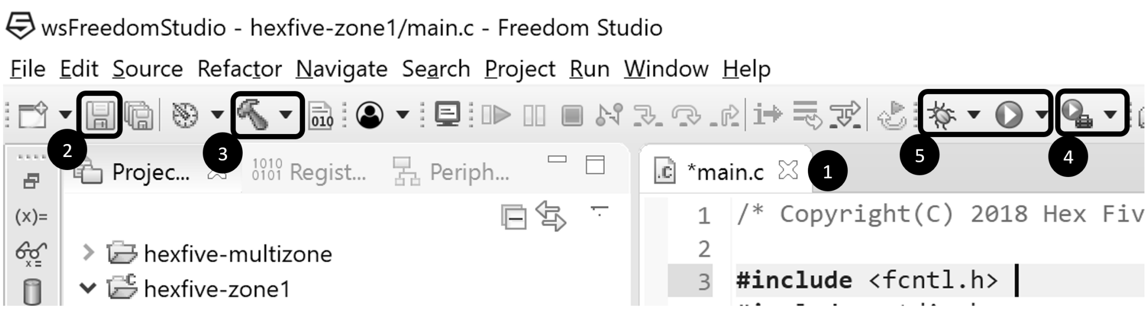
3. Windows user: add '.exe' to external tool configuration for multizone RV32 and multizone RV64
   (a) Run - External Tools - External Tool Configurations
   (b) Edit - Main - Location eclipse_home../jre/bin/java.exe
      i. Make this change for both the multizone.RV32 and multizone.RV64 entries
      ii. This is requires as the same repository supports both Linux and Windows; this is the only difference that could not be abstracted



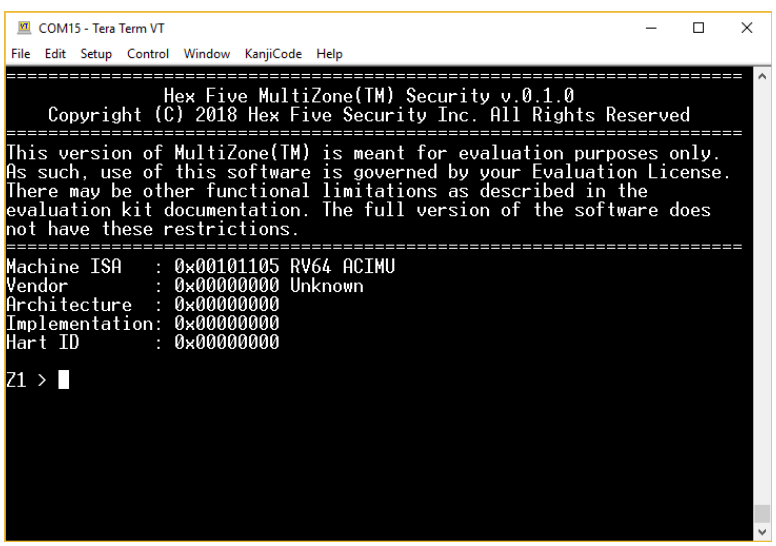


4. Build Process involves (5) steps – all shown as Icons on the Icon Bar
    1) Make a change to a project file (in this case main.c)
    2) Save Changes - Disk Icon
    3) Build Zone Files - Hammer Icon
        i. This compiles and links the zone files to produce HEX files for each Zone
    4) Run Configurator - Run Icon with Red Toolbox
        i. This creates a signed HEX file by merging the zone HEX files with the configured nanoKernel
    5) Upload (Run or Debug) - Run Icon or Debug Icon
        i. This uploads the signed HEX file to the target and starts it running

    The first time you do each action with a dropdown arrow, you need to select RV32 or RV64 for the E31 or E51 core accordingly. Subsequently you can just click the icon.



5. Open a terminal program to access the Zone 1 Terminal : 115200 baud 8N1, VT100



Chapter 4

Multizone Security API

If you expand the hexfive-multizone project and double click on libhexfive.h you will see the API that is available to each Zone.

```
/* Copyright(C) 2018 Hex Five Security, Inc. - All Rights Reserved */

#include <unistd.h>

#ifndef LIBHEXFIVE_H_
#define LIBHEXFIVE_H_

void ECALL_YIELD();
void ECALL_SEND(int, void *);
void ECALL_RECV(int, void *);

void ECALL_TRP_VECT(int, void *);
void ECALL_IRQ_VECT(int, void *);

uint64_t ECALL_CSRR_MTIME();
uint64_t ECALL_CSRR_MCYCLE();
uint64_t ECALL_CSRR_MINSTR();
uint64_t ECALL_CSRR_MHPMC3();
uint64_t ECALL_CSRR_MHPMC4();

uint64_t ECALL_CSRR_MISA();
uint64_t ECALL_CSRR_MVENDID();
uint64_t ECALL_CSRR_MARCHID();
uint64_t ECALL_CSRR_MIMPID();
uint64_t ECALL_CSRR_MHARTID();

#endif /* LIBHEXFIVE_H_ */
```

The design point of the API is to be minimalist, additional services can be built into Zones as needed. MultiZone Security is capable of operating code designed for M-mode natively using a trap and emulate structure, when a zone attempts to execute a privledged instruction the nanoKernel will intercept it and, if it is allowed, will emulate and return the value to the zone. However, this results in a performance penalty and thus is not the recommended approach for system design. In the hexfive-zone1/main.c – examples of trap and emulate and ecalls are both shown on rows 71 and 72 for the same function:

```
uint64_t misa = 0x0; asm ( "csrr %0, misa" : "=r"(misa) );
//const uint64_t misa = ECALL_CSRR_MISA();
```

In the first example, an misa read is directly executed which will cause an exception that is trapped an emulated by the nanoKernel. In the second example (commented out), the ECALL_CSRR_MISA(); API is used to read MISA with a materially lower performance impact.

Function	Syntax and function	Example
ECALL_YIELD	ECALL_YIELD(); Indicates to the nanoKernel scheduler that the Zone has nothing pressing to do and causes the nanoKernel to immediately move to the next Zone in context.	ECALL_YIELD(); In the case of a three zone implementation with a tick time of 10ms, the maximum time to come back to context is 20ms, faster if the other zones Yield as well.
ECALL_SEND	ECALL_SEND([Zone #], [0-3][Int]); Send transmits a message from the current zone to the [Zone #]; the message size is an array of [4] integers and the nanoKernel manages transmission with no shared memory.	ECALL_SEND(1, {201, 0, 0 ,0}); Sends an array to Zone 1 of 201, 0, 0, 0
ECALL_RECV	ECALL_RECV[Zone #], [0-3][int]); Checks the mailbox of the current Zone for a message from the listed Zone #, if a message exists it copies it to the array structure provided.	int msg[4]={0,0,0,0}; ECALL_RECV(1, msg); If a message exists in the mailbox from zone 1, it copies it to msg, otherwise msg value is unchanged.
ECALL_TRP_VECT	ECALL_TRP_VECT([Exception Code], [Trap Handler]) Registers a handler against a trap generated for an unauthorized instructions; the TRAP #s are defined in the RISC-V Privileged Architectures definition V1.1, Table 3.6 Interrupt 0 types.	ECALL_TRP_VECT(0x0, trap_0x0_handler); Where trap_0x0_handler is registered at the User level of privilege as shown in the Zone 2 sample code.
ECALL_IRQ_VECT	ECALL_IRQ_VECT([Interrupt #], [Trap Handler]) Registers a handler for an interrupt that has been assigned to a Zone in the multizone.cfg file. When an interrupt occurs, the nanoKernel will immediately pull the zone assigned to that interrupt into context and execute the registered interrupt handler.	ECALL_IRQ_VECT(11, button_0_handler); Where button_0_handler is a registered at the user level of privilege as shown in the Zone 2 example code.

Function	Syntax and function	Example
ECALL_CSRR_MTIME	Returns MTIME to a variable in a zone, MTIME is a privileged registered normally only available in M mode.	Int64 mtime = ECALL_CSRR_MTIME();
ECALL_CSRR_MCYCLE	Returns MCYCLE to a variable in a zone, MCYCLE is a privileged registered normally only available in M mode.	Int64 mcycle = ECALL_CSRR_MCYCLE();
ECALL_CSRR_MINSTR	Returns MINSTR to a variable in a zone, MINSTR is a privileged registered normally only available in M mode.	Int64 minstr = ECALL_CSRR_MINSTR();
ECALL_CSRR_MHPMC3	Returns MHPMC3 to a variable in a zone, MHPMC3 is a privileged registered normally only available in M mode.	Int64 mhpmc3 = ECALL_CSRR_MHPMC3();
ECALL_CSRR_MHPMC4	Returns MHPMC4 to a variable in a zone, MHPMC4 is a privileged registered normally only available in M mode.	Int64 mhpmc3 = ECALL_CSRR_MHPMC4();

Function	Syntax and function	Example
ECALL_CSRR_MISA	Returns MHPMC4 to a variable in a zone, MHPMC4 is a privileged registered normally only available in M mode.	Int64 mhpmc3 = ECALL_CSRR_MHPMC4();
ECALL_CSRR_MVENDID	Returns MVENDID to a variable in a zone, MVENDID is a privileged registered normally only available in M mode.	Int64 misa = ECALL_CSRR_MVENDID();
ECALL_CSRR_MARCHID	Returns MARCHID to a variable in a zone, MARCHID is a privileged registered normally only available in M mode.	Int64 marchid = ECALL_CSRR_MARCHID();
ECALL_CSRR_MIMPID();	Returns MIMPID to a variable in a zone, MIMPID is a privileged registered normally only available in M mode.	Int64 mimpid = ECALL_CSRR_MIMPID();
ECALL_CSRR_MHARTID	Returns MHARTID to a variable in a zone, MHARTID is a privileged registered normally only available in M mode.	Int64 mhardid = ECALL_CSRR_MHARTID();

Chapter 5

MultiZone Security Configuration File Definition

The configuration file for the evaluation version of MultiZone Security is shown below, it is presented for your reference but the configuration of the evaluation version of MultiZone Security is locked, thus changes to this file have no effect. Program code operating in each zone is fully modifiable and debuggable inside the zone constraints definitions shown below.

```
# Copyright(C) 2018 Hex Five Security, Inc. - All Rights Reserved

# Kernel
tick  = 10 # ms

# Zone 1
mz1_fence = FENCE
mz11_base = 0x40410000; mz11_size = 64K; mz11_rwx = RX # FLASH
mz12_base = 0x80001000; mz12_size =  4K; mz12_rwx = RW # RAM
mz13_base = 0x20000000; mz13_size =  32; mz13_rwx = RW # UART

# Zone 2
mz2_irq = 11, 21, 22 # BTN0 BTN1 BTN2
mz21_base = 0x40420000; mz21_size = 64K; mz21_rwx = RX # FLASH
mz22_base = 0x80002000; mz22_size =  4K; mz22_rwx = RW # RAM
mz23_base = 0x0200BFF8; mz23_size =   8; mz23_rwx = RO # RTC
mz24_base = 0x20005000; mz24_size =  64; mz24_rwx = RW # PWM
mz25_base = 0x20002000; mz25_size =  64; mz25_rwx = RW # GPIO
mz26_base = 0x0C000000; mz26_size =  4M; mz26_rwx = RW # PLIC

# Zone 3
mz3_irq = 23 # BTN3
mz31_base = 0x40430000; mz31_size = 64K; mz31_rwx = RX # FLASH
mz32_base = 0x80003000; mz32_size =  4K; mz32_rwx = RW # RAM
mz33_base = 0x0200BFF8; mz33_size =   8; mz33_rwx = RO # RTC
mz34_base = 0x20002000; mz34_size =  64; mz34_rwx = RW # GPIO
```

Parameter	Definition
tick	tick is the maximum time in ms a Zone may stay in context before the nanoKernel preemptively switches to the next Zone in a round robin manner. The value zero switches to cooperative behavior whereas context switch happens only in response to ECALL_YIELD().
mzx_fence	FENCE – this determines whether fencing is enabled when this zone comes into and leaves context. If no mzx_fence parameter is present, then fencing is disabled by default. The purpose of FENCE commands is to allow the processor to synchronize the thread and the cache to prior to changing context. If FENCE is turned on, a FENCE and FENCE.I command is issued prior to bring that Zone into context and prior to having that zone leave context. There is a core dependent performance penalty for this instruction that can be material, however in the SiFive E31 and E51 implementation this penalty is negligible.
mzx_irq	Interrupt mapping – all interrupts are received by the nanoKernel and, if mapped to a Zone cause that zone to immediately come into context, if it is not already in context, and the assigned interrupt handler to execute upon policy verification. Arguments are: mzx_irq = [interrupt numbers assigned to zone, separated with a comma]
mzx1_base	Each zone has (6) available ranges of memory, including mapped peripherals, that can be uniquely assigned to that zone. • mzx1 is for ROM; size that is a multiple of 4 Bytes, base address that is aligned to 4B boundary; when the Zone begins the program counter points to this base address; generally permissions should be RX so that fixed variable loads can also be done from ROM. • mzx2 is for RAM; size that is a multiple of 4 Bytes, base address that is aligned to 4B boundary; generally permissions would be RW as code is typically not executed from RAM. • mzx3-6 are for other memory mapped peripherals – base address must be a multiple of the size (NAPOT) Arguments are: [where x is a number from 1-6]: • mzx1_base = [physical base address] • mzx1_size = [It parses byte, K or M] • mzx1_rwx = any combination of [RWX] – defines memory range permissions Read, Write and Execute

Chapter 6

MultiZone Configurator Command line Options

The MultiZone Configurator is invoked automatically when you click the Run External Icon in Eclipse, however it can be operated from a command line as well.

It ships as a java runtime for platform independence:

```
$ java -jar multizone.jar -?
Usage: hexfive-conf [OPTION...] file.hex... [-o file.hex]
Hex Five MultiZone(TM) Configurator

 -c, --output=file.cfg      Config file. Default: multizone.cfg
 -o, --output=file.hex      Output file. Default: multizone.hex
 -a, --arch={rv32|rv64}     Architecture. Default: autodetect
 -q, --quiet                Don't produce any output
 -v, --verbose              Produce verbose output
 -?, --help                 Give this help list
     --usage                Give a short usage message
 -V, --version              Print program version

Example: java -jar multizone.jar zone1.hex zone2.hex zone3.hex -o multizone.hex
Report bugs to <bug@hex-five.com>.
```

Chapter 7

Errata

Issue
Compatibility with SiFive ‘C’ compiler optimizations – the bundled version of the SiFive ‘C’ compiler generates an internal error if optimizations are turned on when compiling core tagged as “user” privilege level.

Work Around
Optimizations have been disabled in the script files shipped with the MultiZone Security Evaluation SDK. There is no indication that this results in any reduction in performance or increase in code size.