MultiZone^{TM}Security

Secure IoT Stack

Document Version Rev 1.0.1

February 25, 2019

Table 1: Version History

Version	Date	Changes
1.0	Feb 18, 2019	Initial Release
1.0.1	Feb 25, 2019	Update Zone 2 and 3 functionality from final code

Contents

Chapter 1

Introduction

This technical note describes the reference implementation of a Secure Iot Stack running on the MultiZone Security Trusted Execution Environment (TEE) on a newly modified version of the RISC-V Rocket Core with improved performance and added Ethernet peripheral.

Specifically, FreeRTOS has been implemented securely by dividing it into three components and implementing those in separate Zones with no shared memory:

1. FreeRTOS running three OS Tasks (CLI, LED and Robot) and the FreeRTOS Scheduler
2. picoTCP TCP/IP stack and wolfSSL TLS crypto library
3. Root of Trust for secret storage

A 4th Zone is implemented as a UART console to monitor performance of the overall system and help illustrate the messaging functionality.

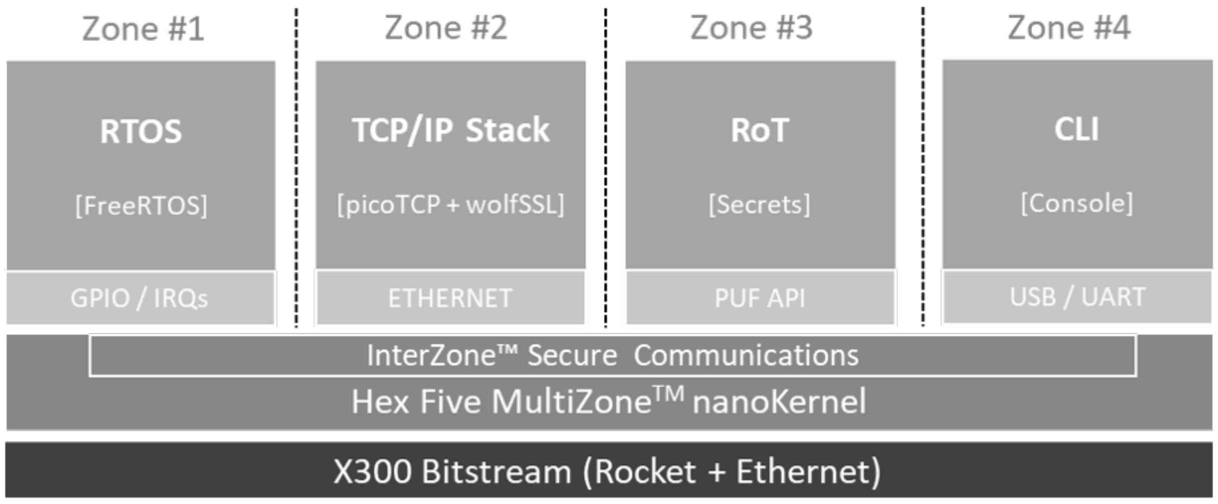


Fig. 2 MultiZone Secure IoT Stack support FreeRTOS separated across 3 Zones plus an additional Zone for a UART console.

All communication between Zones is implemented using the secure InterZone Messenger function of MultiZone Security, which itself uses no shared memory.

The overall system presents a compelling reference platform for IoT devices requiring FreeRTOS but needing a more secure implementation which prevents exploits from propagating from one component to the full OS.

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 MultiZone Security - the first Secure IoT Stack for RISC-V

The purpose of this reference design is to illustrate how a rich operating system can be decomposed into its ingredient parts and implemented across multiple Zones with complete separation thereby materially improving its cyber-resilience.

Specifically, in Zone 1 the TCP/IP stack was removed from FreeRTOS and replaced with a messaging interface using InterZone Messenger. In Zone 2, PicoTCP and wolfSSL were implemented to provide a TLS termination for the ethernet port and their APIs wrapped with the InterZone Messenger pointing incoming and outgoing packets at Zone 1 and secrets in Zone 3.

By creating this separation, exploitable vulnerabilities in the TCP/IP and TLS stack and contained to Zone 2 - they can neither propagate to Zone 3 to access the root of trust, now to Zone 1 to access the various tasks implemented in FreeRTOS.

All of this is implemented without ANY shared memory - using the messenger function of MultiZone Security alone. The overhead for this implementation is extremely small:

- 1kB of SRAM and 4kB of flash for the MultiZone nanoKernel
- 0.01% of cycles for context switching
- Minor library duplication between Zones

The largest performance impact comes from solely using the InterZone Messenger interface to move data between Zones 1 and 2; it is also possible to enable a variety of shared memory interfaces between these Zones - secure split buffers or a ring buffer. However, for this IoT endpoint demo even operating at 65MHz on an low cost FPGA board, the performance more than meets the application requirements.

In this case, MultiZone Security is setup with the following configuration file (see multizone.cfg in the BSP/X300/ folder.

For more information on the design and use of MultiZone Security, consult the MultiZone Security SDK manual - available on at https://github.com/hex-five

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

Tick = 10 # ms

Zone = 1 #
    irq  = 16, 17, 18 # BTN0 BTN1 BTN2
    base = 0x20410000; size =      64K; rwx = rx # FLASH
    base = 0x80001000; size =      16K; rwx = rw # RAM
    base = 0x10025000; size =    0x100; rwx = rw # PWM
    base = 0x10012000; size =    0x100; rwx = rw # GPIO
    base = 0x0C000000; size = 0x400000; rwx = rw # PLIC

Zone = 2 #
    base = 0x20420000; size =   64K; rwx = rx # FLASH
    base = 0x80005000; size =   16K; rwx = rw # RAM
    base = 0x60000000; size =    8K; rwx = rw # XEMACLITE

Zone = 3 #
    base = 0x20430000; size =   64K; rwx = rx # FLASH
    base = 0x80009000; size =    4K; rwx = rw # RAM
    base = 0x0200BFF8; size =   0x8; rwx = r  # RTC

Zone = 4 #
    base = 0x20440000; size =   64K; rwx = rx # FLASH
    base = 0x8000A000; size =    4K; rwx = rw # RAM
    base = 0x10013000; size = 0x100; rwx = rw # UART
```

Fig. 1 This is the multizone.cfg used in multizone-secure-iot-stack, it is located in the /bsp/X300 directory. This version can be fully modified as desired as changes are live, the only item that cannot be changes are the base addresses for each board hard coded into each BSP.
It is possible to overlap memory regions in order to share resources, however this increases the potential attack surface and it allows user code in Zones to directly interact with each other. In this implementation, shared memory is not required and thus all communicaiton between Zones has been restructed to the InterZOne Messenger.

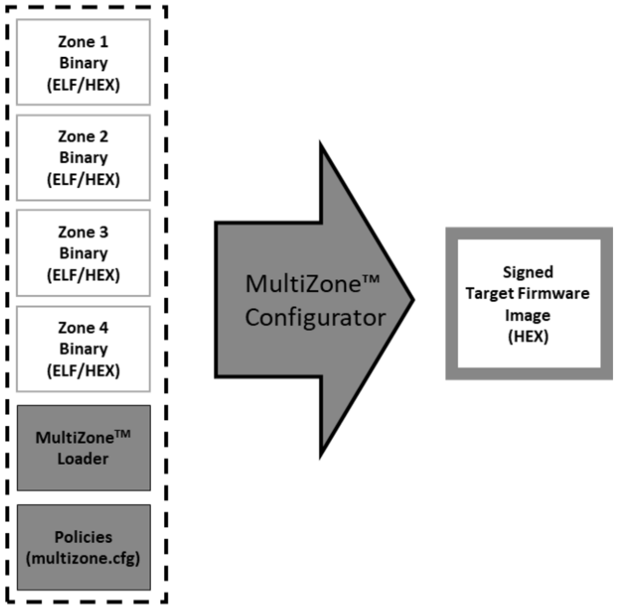


Fig. 2 MultiZone Configurator flow showing (4) pre-compiled and linked Zone binaries coming together with the multizone.cfg file into a signed target firmware image (HEX).

Chapter 2

Secure IoT Stack - Demo Application

This chapter describes the MultiZone Security Secure Iot Stack implementation through 4 zones - the design decisions that were taken and tradeoffs incurred:

1. FreeRTOS Zone with TCP stack removed
2. TCP/IP Zone with picoTCP
3. Key Management Zone with wolfSSL
4. TEE Console Zone - UART Interface

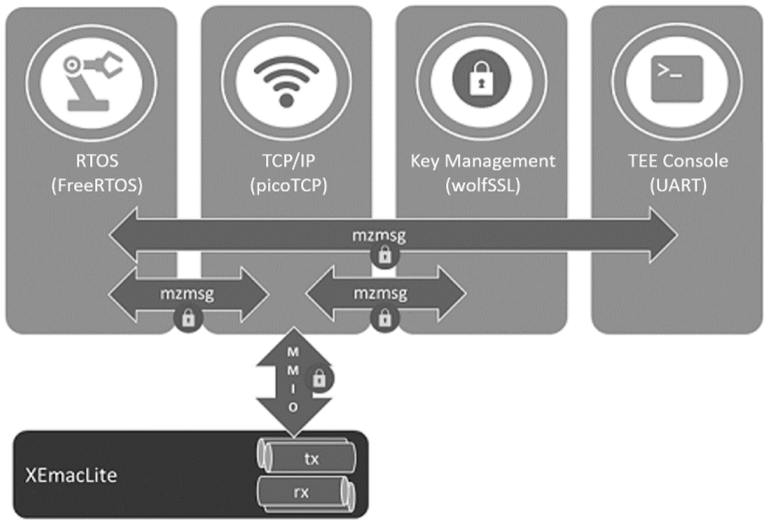


Fig. 4 MultiZone Secure IoT Stack data flow showing the Ethernet peripheral communicating with the TCP/IP + TLS stack in Zone 2, then the InterZone Messenger being used for the balance of communication.

Data flow in this system is the following

- Ethernet terminates on Zone 2 - picoTCP

- TCP/IP packets are decoded and the contents routed to/from FreeRTOS Zone
- For TLS 1.3 sessions, the public key and encode/packet decode libraries exist in Zone 2, but the private key exists only in Zone 3 - during TLS 1.3 session negotiation, the client key is signed in Zone 3 and passed back to Zone 2
- The TEE Console provides a second interface into the system to evaluate performance, look at isolation boundaries of a different Zone and send messages back and forth to FreeRTOS (Zone 1)

2.1 Configuring the toolchain

To configure the toolchain, build and upload the software starting from a fresh installation of Ubuntu 18.02 LTS (other Linux versions require a subset of these commands)

Hex Five created a modified version of the Rocket SoC called teh X300 with an additional Ethernet port and improved performance

Upload the X300 Bitstream to a Xilinx Artik-7 35T Arty FPGA board prerequisites: Xilinx Vivado, Olimex ARM-USB-TINY-H Debugger

Download the X300 bitstream .mcs file from https://github.com/hex-five/multizone-fpga/releases Push the .mcs file to the Arty board using Vivado Install the reference RISC-V toolchain for Linux - directions specific to a fresh Ubuntu 18.04 LTS, other Linux distros generally a subset

```
sudo apt update
sudo apt upgrade -y
sudo apt install git make default-jre libftdi1-dev
sudo ln -s /usr/lib/x86_64-linux-gnu/libmpfr.so.6 /usr/lib/x86_64-linux-gnu/libmpfr.so.4
wget https://github.com/hex-five/multizone-sdk/releases/download/v0.1.0/riscv-gnu-toolchain-20181226
tar -xvf riscv-gnu-toolchain-20181226.tar.xz
wget https://github.com/hex-five/multizone-sdk/releases/download/v0.1.0/riscv-openocd-20181226.tar.x
tar -xvf riscv-openocd-20181226.tar.xz
git clone https://github.com/hex-five/multizone-secure-iot-stack
cd multizone-secure-iot-stack
git update --init --recursive
sudo apt-get install libusb-0.1-4
sudo apt-get install screen
```

If you have not already done so, you need to edit or create a file to place the USB devices until plugdev group so you can access them without root privileges:

```
sudo vi /etc/udev/rules.d/99-openocd.rules
```

Then place the following text in that file Detach and re-attach the USB devices for these changes

```
# These are for the HiFive1 Board
SUBSYSTEM=="usb", ATTR{idVendor}=="0403",
ATTR{idProduct}=="6010", MODE="664", GROUP="plugdev"
SUBSYSTEM=="tty", ATTRS{idVendor}=="0403",
ATTRS{idProduct}=="6010", MODE="664", GROUP="plugdev"
# These are for the Olimex Debugger for use with E310 Arty Dev Kit
SUBSYSTEM=="usb", ATTR{idVendor}=="15ba",
ATTR{idProduct}=="002a", MODE="664", GROUP="plugdev"
SUBSYSTEM=="tty", ATTRS{idVendor}=="15ba",
ATTRS{idProduct}=="002a", MODE="664", GROUP="plugdev"
```

to take effect.

Add environment variables and a path to allow the Makefiles to find the toolchain

edit /.bashrc and /.profile and place the following text at the bottom of both files. Close and

```
export RISCV=/home/<username>/riscv-gnu-toolchain-20181226
export OPENOCD=/home/<username>/riscv-openocd-20181226
export PATH="$PATH:/home/<username>/riscv-gnu-toolchain-20181226/bin"
```

restart the terminal session for these changes to take effect.

Compile and Upload the Project to the Arty Board This will result in a HEX file that is now ready

```
cd multizone-secure-iot-stack/
make clean
make
```

to upload to the Arty board. The first time you push this HEX file up it takes about 2 minutes, on subsequent passes it goes much faster.

2.2 Operating the Demonstration Applications

Zone 1 is configured with FreeRTOS running three tasks as shown in the splash screen on power-up.

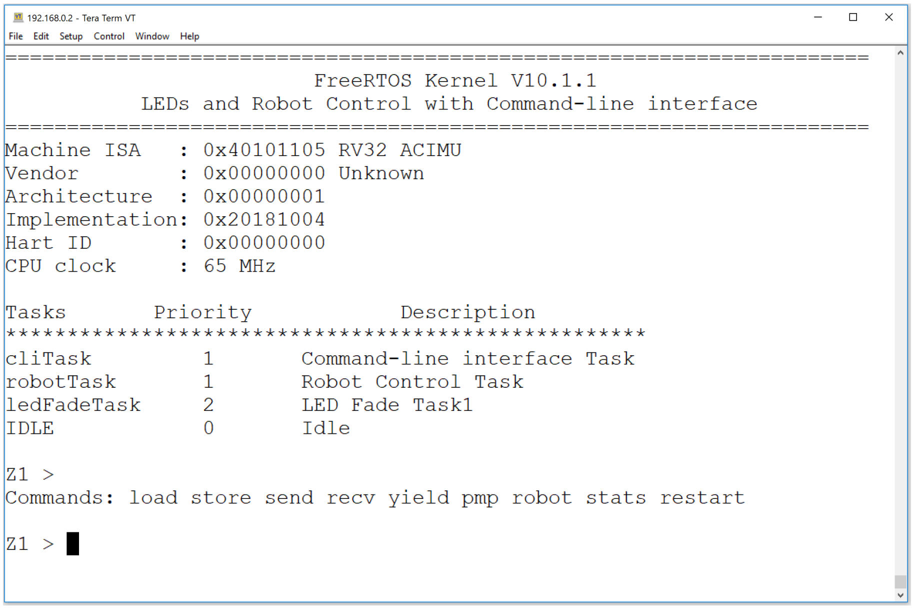
1. Realtime Robot Operation - High Priority
2. LED Fade Operation - High priority
3. Command Line Interface (CLI) - Medium priority

```
make load
```

The default IP address as specified in the Makefile is 192.168.0.2. You can either telnet into this or use ssh to enter this zone:

```
telnet 192.168.0.2
stty -icanon -echo && openssl s_client -crlf -connect
```

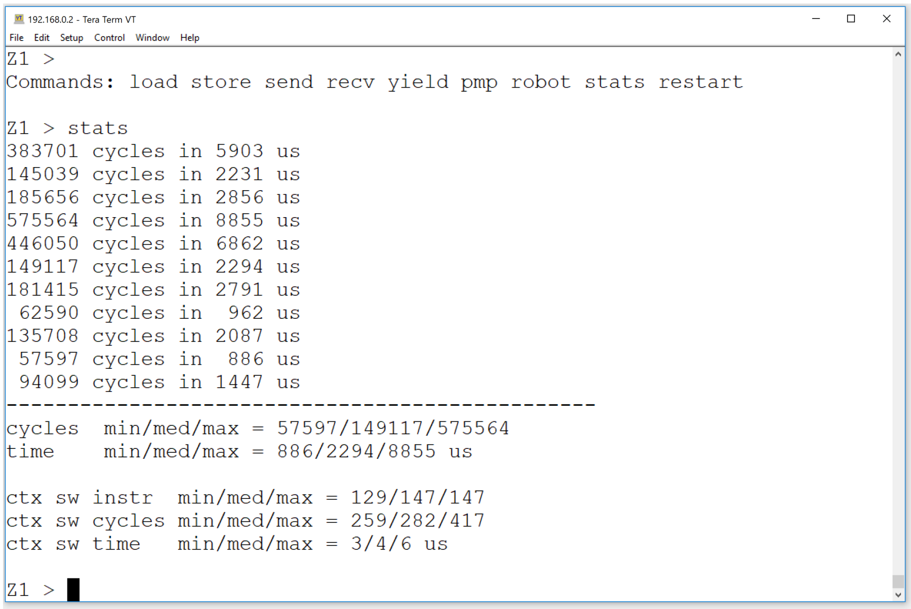
Once you enter the zone, you will be presented with the following Splash screen, press enter and you'll get a list of available commands as shown.


192.168.0.2 - Tera Term VT
File Edit Setup Control Window Help

```
======================================================================
                         FreeRTOS Kernel V10.1.1
              LEDs and Robot Control with Command-line interface
======================================================================
Machine ISA   : 0x40101105 RV32 ACIMU
Vendor        : 0x00000000 Unknown
Architecture  : 0x00000001
Implementation: 0x20181004
Hart ID       : 0x00000000
CPU clock     : 65 MHz

Tasks        Priority            Description
****************************************************
cliTask           1       Command-line interface Task
robotTask         1       Robot Control Task
ledFadeTask       2       LED Fade Task1
IDLE              0       Idle

Z1 >
Commands: load store send recv yield pmp robot stats restart

Z1 >
```

If you run the stats command, you'll see the performance of this system. As the FreeRTOS scheduler is invoked each time FreeRTOS runs a task, the stats are slightly slower in Zone 1 than in than in Zone 4 where no scheduler overhead exists.

The robot can be controller by sending messages to zone 1 itself in the format:


192.168.0.2 - Tera Term VT
File Edit Setup Control Window Help

```
Z1 >
Commands: load store send recv yield pmp robot stats restart

Z1 > stats
383701 cycles in 5903 us
145039 cycles in 2231 us
185656 cycles in 2856 us
575564 cycles in 8855 us
446050 cycles in 6862 us
149117 cycles in 2294 us
181415 cycles in 2791 us
 62590 cycles in  962 us
135708 cycles in 2087 us
 57597 cycles in  886 us
 94099 cycles in 1447 us
------------------------------------------------
cycles  min/med/max = 57597/149117/575564
time    min/med/max = 886/2294/8855 us

ctx sw instr  min/med/max = 129/147/147
ctx sw cycles min/med/max = 259/282/417
ctx sw time   min/med/max = 3/4/6 us

Z1 >
```

```
Z1> send 1 > (unfolds robot)
Z1> send 1 1 (begins timed dance)
Z1> send 1 0 (ends timed dance
Z1> send 1 < (folds robot)
```

You can also enter Zone 4 via the USB UART at 115200 board, 8N1 by using a suitable serial terminal to /dev/ttyUSB* (where * varies system to system - typically 0, 1 or 2.

The available commands in Zone 4 follow the same format as Zone 1 and are documented in detail in the Zone 4 section. You an also send ping/pong messages from Zone 1 to Zone 4 and Zone 1 to Zone 4:

Zone 1 Console

```
Z1> send 4 ping
```

Zone 4 Console

```
Z1> ping
```

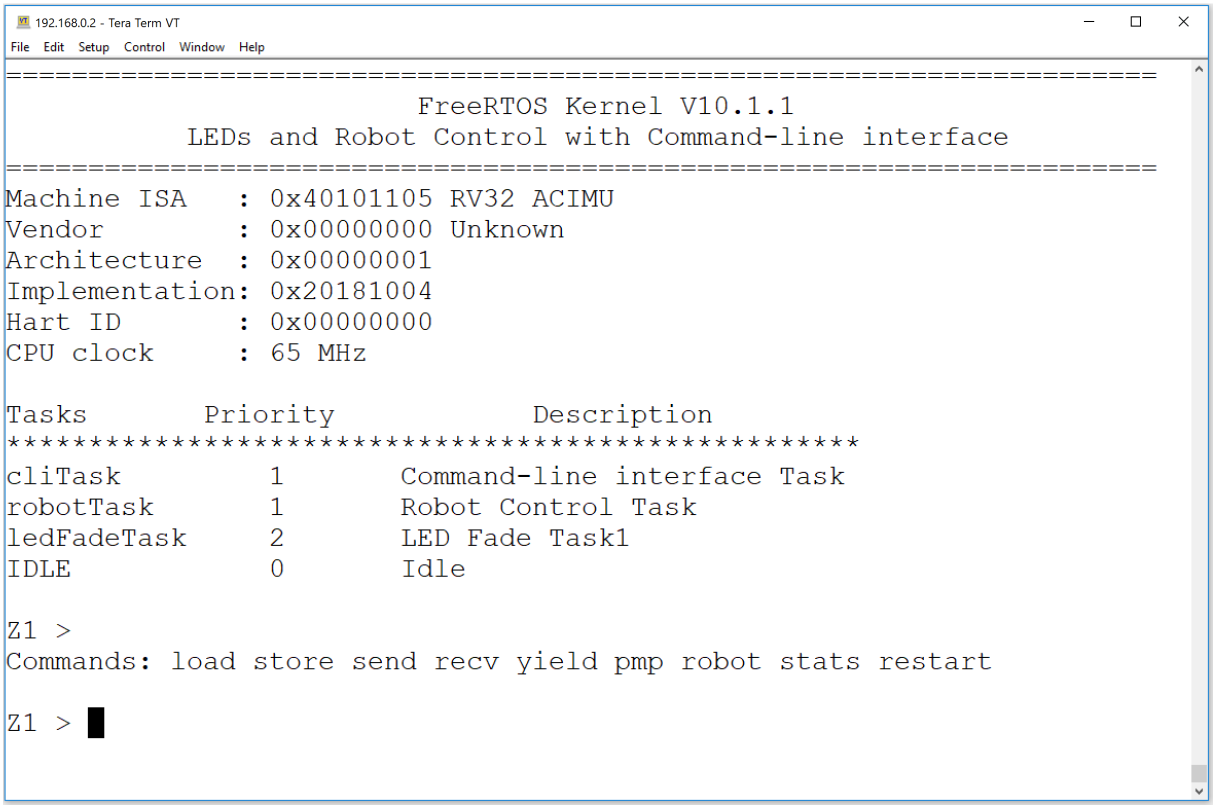
Chapter 3

FreeRTOS Implementation Details

This chapter describes how FreeRTOS is implemented in the MultiZone Security Secure IoT Stack - the design decisions that were taken and tradeoffs incurred.

3.1 Operating the FreeRTOS Zone

Zone 1 is configured with FreeRTOS running three tasks as shown in the splash screen on power-up.


192.168.0.2 - Tera Term VT
File Edit Setup Control Window Help

```
======================================================================
                         FreeRTOS Kernel V10.1.1
            LEDs and Robot Control with Command-line interface
======================================================================
Machine ISA    : 0x40101105 RV32 ACIMU
Vendor         : 0x00000000 Unknown
Architecture   : 0x00000001
Implementation: 0x20181004
Hart ID        : 0x00000000
CPU clock      : 65 MHz

Tasks        Priority            Description
****************************************************
cliTask          1       Command-line interface Task
robotTask        1       Robot Control Task
ledFadeTask      2       LED Fade Task1
IDLE             0       Idle

Z1 >
Commands: load store send recv yield pmp robot stats restart

Z1 >
```

The commands available to the FreeRTOS Zone are

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]`

send	send [Zone #] message
	Sends a message to another zone – in this application, only zone 3 is actively listening for messages and these only relate to the robot.
		`Z1> send 4 a`
		`(sends and 'a' to Zone 4 UART CLI)`

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 3159us`
		`[Zones 2, 3 & 4 are configured to yield by default]`

Command	Syntax and function	Example

stats	Stats – completes a set of 11 yield commands, measures can calculates context switch time in cycles and microseconds	`Z1> stats`
		`84955 cycles in 1307 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 = ...`

robot	`robot [command] { control the`
	`     robot attached the GPIO pins`
	
	`valid commands are:`
	` > - unfold`
	` < - fold`
	` 1 - begin timed dance`
	` 0 - end timed dance`
	
	`minor adjustments`
	` q - open pincher`
	` a - close pincher`
	` w - lift wrist`
	` s - drop wrist`
	` e - lift shoulder`
	` d - drop should`
	` r - lift waist`
	` f - drop waist`
	` t - rotate c-clockwise`
	` g - rotate clockwise`
		`Z1> robot >`
		`(robot unfolds)`
		`Z1> robot 1`
		`(robot begins timed dance)`
		`Z1> robot >`
		`(robot ends timed dance)`
		`Z1> robot <`
		`(robot folds)`

3.2 FreeRTOS Zone Implementation Details

FreeRTOS is initialized with a 20ms tick rate by using the soft timer function API built into MultiZone Security.

Every 20ms, the vPortSysTickHandler is called which Increments the RTOS Task handler and sets the next soft timer event.

```
void vPortSetupTimerInterrupt()
{
/* Calculate first tick timer compare */
const uint64_t ullCurrentTime = ECALL_CSRR_MTIME();
const uint64_t ullNextTime = ullCurrentTime + (configRTC_CLOCK_HZ /
configTICK_RATE_HZ);
/* Setup mtimer handler */
ECALL_TRP_VECT(0x3, _timer_handler);
/* Request first tick interrupt */
ECALL_CSRW_MTIMECMP(ullNextTime);
}
```

```
void vPortSysTickHandler()
{
/* Calculate next compare value */
const uint64_t ullCurrentTime = ECALL_CSRR_MTIME();
const uint64_t ullNextTime = ullCurrentTime + (configRTC_CLOCK_HZ /
configTICK_RATE_HZ);
/* Increment the RTOS tick. */
if ( xTaskIncrementTick() != pdFALSE ){
ulPortYieldRequired = pdTRUE;
}
/* Request next timer interrupt */
ECALL_CSRW_MTIMECMP(ullNextTime);
}
```

Interrupts are handled using a low-level user mode interrupt handler which saves context and pushes the interrupt source into a register, then calls the handle interrupt routine to service the interrupt.

Interrupt Initialization

```
void b0_irq_init()
{
...
/* Enable the interrupt */
ECALL_IRQ_VECT(16+LOCAL_INT_BTN_0, _interrupt_entry);
localISR[IRQ_M_LOCAL + LOCAL_INT_BTN_0] = button_0_handler;
}}
```

Low-level user mode interrupt handler

```
_interrupt_entry:
...
/* Save RegFile context */
pushREGFILE
/* Save mcause */
LOAD s0, MCAUSE_OFFSET(sp)
STORE zero, MCAUSE_OFFSET(sp)
...
/* Save Task context*/
portSAVE_CONTEXT
/* Call IRQ handler (a0 = s0 = mcause) */
mv a0, s0
jal handle_interrupt
...
```

Interrupt Handling

```
void handle_interrupt(unsigned long mcause)
{
/* Check if global*/
if(!((mcause & MCAUSE_CAUSE) == IRQ_M_EXT)){
localISR[mcause & MCAUSE_CAUSE]();
}
```

Synchronous traps are handled in the same manner has interrupts, registering a syncexception entry against all of the traps, pulling exception information from the Zone's own mailbox then passing that information to the interrupt handler.

Define Synchronous Interrupt Handler

```
_syncexception_entry:
/* Receive own zone's mailbox to get exception information */
li a0, MULTIZONE_ZONE
la a1, _msg
jal ECALL_RECV
la t0, _msg
/* pass the obtained register information to the handler */
LOAD a0, 0(t0) // mcause
LOAD a1, 1*portWORD_SIZE(t0) // mtval
LOAD a2, 2*portWORD_SIZE(t0) // mepc
jal handle_syncexception
...
```

Register Synchronous Interrupt Handler

```
BaseType_t xPortStartScheduler ( void )
{
...
/* 0x0 Instruction address misaligned */
ECALL_TRP_VECT(0x0, _syncexception_entry);
/* 0x1 Instruction access fault */
ECALL_TRP_VECT(0x1, _syncexception_entry);
...
/* 0x7 Store access fault */
ECALL_TRP_VECT(0x7, _syncexception_entry);
...
}
```

Define Interrupt Handler

```
__attribute__((weak))
void handle_syncexception(unsigned long mcause, unsigned long mtval,
unsigned long mepc)
{
switch(mcause){
case 0x0: // Instruction address misaligned
...
break;
...
}
...
}
```

The TCP/IP has been removed from this implementation of FreeRTOS and replaced with a wrapper to send and received InterZone Messages to the picoTCP + wolfSSL implementation in Zone 2. A data structure is implemented to add flow control ontop of the basic InterZone Messenger such that 4 characters are sent at a time between Zone 1 and Zone 4 and further transmission is held until and ACK is received from the recipient Zone indicating that the mailbox is clear and another message is clear to send.

```
typedef struct {
    int zone;
    int out[4];
    int in[4];
    int ack_pending;
    int ack_index;
    int last_index;
} mzmsg_t;
```

Messages received from Zone 2 are parsed for backspace, delete and arrow keys then directed to the cli task via the readline function as if they were being typed in locally:

```
void cliTask( void *pvParameters){
    char c = 0;
mzmsg_init(&zone2, 2);

    mzmsg_write(&zone2, (char *) welcome_msg, sizeof(welcome_msg));
    print_cpu_info();
    mzmsg_write(&zone2, (char *) taskinfo_msg, sizeof(taskinfo_msg));

    char cmd_line[CMD_LINE_SIZE+1]="";
int msg[4]={0,0,0,0};

    while(1){

    mzmsg_write(&zone2, "\r\nZ1 > ", 7);
    readline(cmd_line);
        mzmsg_write(&zone2, "\r\n", 2);

...

if (tk1 != NULL && strcmp(tk1, "pmp")==0){
            print_pmp_ranges();
        } else if(tk1 != NULL && strcmp(tk1, "robot")==0){
...
```

Chapter 4

TCP/IP - picoTCP

This chapter describes how picoTCP is implemented in Zone 2 of MultiZone Secure IoT Stack - the design decisions that were taken and tradeoffs incurred.

picoTCP is implemented in Zone 2 to terminate a telnet sessions on port 232 and TLS sessions on port 443. The contents of packets are then sent and received to/from the FreeRTOS CLI session in Zone 1 using InterZone Messenger with no shared memory. An alternate configuration would be to use a shared memory buffer structure for a zero copy TCP/IP approach, however this would increase the attack surface and was not necessary for the level of performance targeted in this demo.

4.1 Messaging Protocol

To facilitate communications between Zones, an ACK protocol is implemented ontop of InterZone messenger between Zones 1 and 2. If a message has been sent and no ACK received back, Zone 2 will yield until it sees the ACK, then s1 implements the mirror of the protocol.

InterZone messenger allows for an array of 4 integers to be sent, in this case a simple data structure called mzmsg is implemented to send up to MZMSG_CHARS (default 4) characters plus control signals per message:

```
typedef struct {
    int zone;
    int out[4];
    int in[4];
    int ack_pending;
    int ack_index;
    int last_index;
} mzmsg_t;
```

picoTCP handles all aspects of TCP/IP transport in Zone 2 entirely transparently from FreeRTOS in Zone 1.

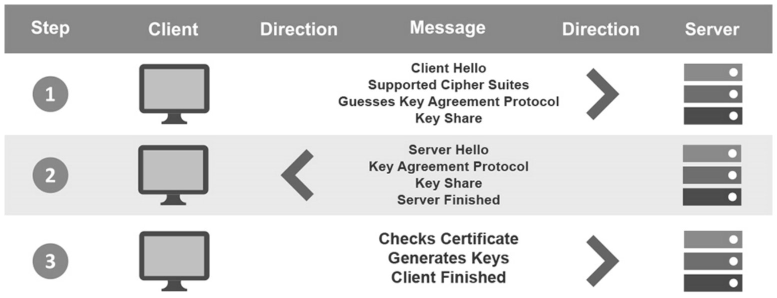
PicoTCP sends a packet for each set of MZMSG_CHARS received via InterZone messenger, in the case of an unencrytped telnet session, the bottleneck to performance is picoTCP's ability to wrap the message characters in an full telnet packet and send them. However, for an edge IoT M2M application, the resulting performance is more than satisfactory.

4.2 TLS 1.3 Protocol

To establish a TLS 1.3 session, the PC issue a command to connect to the Arty baord as a server:

```
void vPortSetupTimerInterrupt()
stty -icanon -echo && openssl s_client -tls1_3 -crlf -nocommands -connect 192.168.0.2:443
```

One of the major advantages of TLS 1.3 vs 1.2 is the key negotiation process is simplified as shown below.



Step	Client	Direction	Message	Direction	Server
1			Client Hello Supported Cipher Suites Guesses Key Agreement Protocol Key Share	>	
2		<	Server Hello Key Agreement Protocol Key Share Server Finished		
3			Checks Certificate Generates Keys Client Finished	>	

Fig. 6. TLS 1.3 Handshake process courtesy of TheSSLStore.com

Step 1: Similar to the TLS 1.2 handshake, the TLS 1.3 handshake commences with the "Client Hello" message – with one significant change. The client sends the list of supported cypher suites and guesses which key agreement protocol the server is likely to select. The client also sends its key share for that particular key agreement protocol.

Step 2: In reply to the "Client Hello" message, the server replies with the key agreement protocol that it has chosen. The "Server Hello" message also comprises of the server's key share, its certificate as well as the "Server Finished" message. The "Server Finished" message, which was sent in the 6th step in TLS 1.2 handshake, is sent in the second step. Thereby, saving four steps and one round trip along the way.

Step 3: Now, the client checks the server certificate, generates keys as it has the key share of the server, and sends the "Client Finished" message. From here on, the encryption of the data begins.

Read more at: https://www.thesslstore.com/blog/tls-1-3-handshake-tls-1-2/

In this implementation, the server is the Arty board and the client is the PC terminal; the server's public share is hard coded in zone 2 and the client key share is transferred to zone 3 for signing (see Zone 3). The signed key is returned to Zone 2 and the client. The client checks the server certificate, determines it is self signed and proceeds with the TLS 1.3 session.

Zone 2 is responsible for realtime encryption and decryption of packets when the TLS 1.3 session is implemented. As the wolfSSL library is implemented in C, there is no hardware acceleration in the implementation and the soft core processor is operating at only 65MHz, this causes encryption to be a significant bottleneck of performance in the TLS 1.3 session type. This can be viewed by comparing performance of a telnet session to the tls 1.3 session.

Multiple opportunities exist to improve the performance of the system:

- Incoming characters are sent one per packet to facilitate editing with backspace, arrow and delete keys, they could easily be sent one line at a time to improve performance
- Outgoing characters are sent 4 to a packet (by default) to align with the messaging protocol; they could also easily be sent one line at a time.
- There is no hardware crypto acceleration in this bitstream - this could readily be added
- This implementation of wolfSSL is entirely in C; a commercial implementation would likely include a port to RISC-V assembly language materially improving performance
- The softcore processor used in the X300 bitstream is operating at only 65MHz, a hardcore processor would likely operate at 10x+ this frequency

Chapter 5

Key Management - wolfSSL

Zone 3 is responsible for secure storage of the private key and signing client keys to support the TLS 1.3 session handshake.

The private key for the TLS 1.3 session is stored in the key_der struct and a proxy for a true random number generator is seeded with MCYCLE - the number of cycles since system reset.

```
unsigned int my_rng_seed_gen(void)
{
    uint64_t cycles = ECALL_CSRR_MCYCLE();
    rand_seed += cycles;
    rand_seed ^= 0x67452301;
    rand_seed += 0xEFCDAB89;
    rand_seed ^= cycles << 16;
    rand_seed ^= 0x98BADCFE;
    rand_seed += 0x10325476;
    rand_seed += cycles >> 16;
    rand_seed ^= 0xC3D2E1F0;
    return rand_seed;
}
```

During the TLS 1.3 handshake, the client transferred from Zone 2 to Zone 3 and signed with the wolfSSL crypto library using 256 bit Eliptic Curve Cryptography (ECC) and the TLS_AES_128_GCM_SHA256 cyper. This process takes approximately 7 seconds on the 65MHz soft core with the current library implementation.

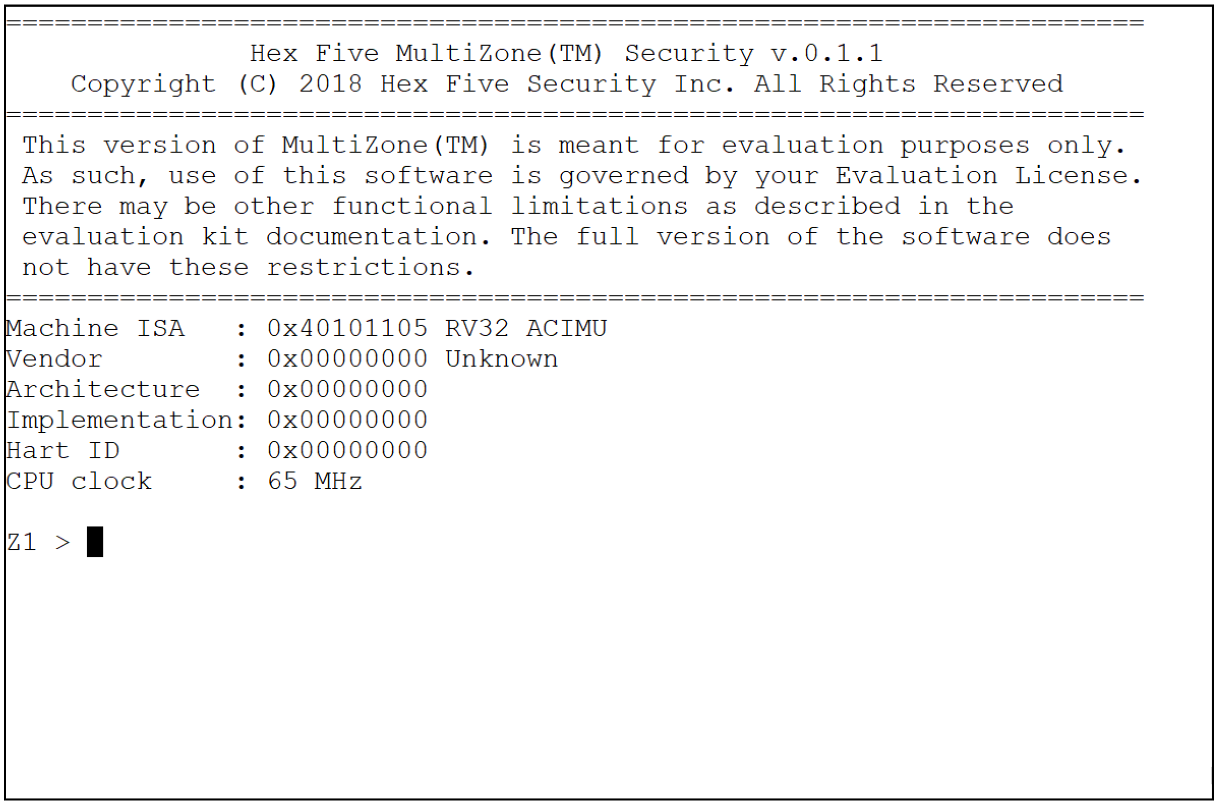
As the maximum context time in the reference application is 10ms, the MultiZone nanoKernel preempts Zone 3 approximately 700 times during this process; as all elements of context are preserved when the nanoKernel preempts a Zone the library is unaware of this preemption.

Once the client key is signed, it is transmitted back to Zone 2 over INterZone messenger and the balance of the TLS 1.3 negotiation can proceed.

Chapter 6

TEE Console Zone

6.1 Operating the UART - CLI Zone



```
======================================================================
               Hex Five MultiZone(TM) Security v.0.1.1
    Copyright (C) 2018 Hex Five Security Inc. All Rights Reserved
======================================================================
 This version of MultiZone(TM) is meant for evaluation purposes only.
 As such, use of this software is governed by your Evaluation License.
 There may be other functional limitations as described in the
 evaluation kit documentation. The full version of the software does
 not have these restrictions.
======================================================================
Machine ISA   : 0x40101105 RV32 ACIMU
Vendor        : 0x00000000 Unknown
Architecture  : 0x00000000
Implementation: 0x00000000
Hart ID       : 0x00000000
CPU clock     : 65 MHz

Z1 > █
```

Fig. 6. Serial terminal window connected to Zone 1. Note the Press Enter on the window to get a list of commands that you can issue to Zone 1.

The available command formats are the same as for Zone 1 with the exception of the robot command being removed (as the robot is only attached two Zone 1) and three new commands being added.

Command	Syntax and function	Example

exec	exec [address] – executed a jump to the address specified	`Z4> exec 20440000`
		`(Zone 4 reboots)`

restart	restart - restarts Zone 4, equivalent to exec to flash base address	`Z4> restart`
		`(Zone 4 restarts)`

exec	timer [milliseconds] – sets a soft timer that expires in specified milliseconds.	`Z4> timer 5000`
		`timer set T0=1447, T1=6447`
		`(wait 5 seconds)`
		`Z4> Timer Expired : 6447`

6.2 UART - CLI Implementation Details

One of the features of Zone 4 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 a single 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 4 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 1 and prints them

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

ECALL_RECV(1, 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....
    break;
    case 2   : write(1, "\rZ3 > USB DEVICE DETACH\r\n", 25); break;
    ...
}
```

Checks for incoming messages from Zone 2 and prints them

```
ECALL_RECV(1, 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 1)

```
ECALL_YIELD();
```

In main() two test options are shown and commented out The first one simulates a locked up Zone and forces the nanoKernel to preempt Zone 4 and force a context switch based on the defined tick time (10ms). The second one immediately yields Zone 4 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 accessed via the MultiZone API

```
} else if (tk1 != NULL && strcmp(tk1, "yield")==0){
    uint64_t C1 = ECALL_CSRR_MCYCLE();
    ECALL_YIELD();
    uint64_t C2 = ECALL_CSRR_MCYCLE();
    const int T = ((C2-C1)*1000000)/CPU_FREQ;
    printf( (T>0 ? "yield : elapsed time %dus \n" : "yield : n/a \n"), T);
```

The softtimer is implemented with a timer command in milliseconds. At the expiry of the timer, trap 0x3 is issued.

```
#include <libhexfive.h>

...

void trap_0x3_handler(void)__attribute__((interrupt("user")));
void trap_0x3_handler(void){
    const uint64_t T = ECALL_CSRR_MTIME();
    ...

    printf("\rZ1 > timer expired : %lu", (unsigned long)(T*1000/RTC_FREQ));

    ....
}

...

main () {

    ECALL_TRP_VECT(0x3, trap_0x3_handler); // register 0x3 Soft timer
    while(1){

        // do many things

    }
}
```

Chapter 7

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_WFI();

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

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

void ECALL_CSRS_MIE();
void ECALL_CSRC_MIE();

void ECALL_CSRW_MTIMECMP(uint64_t);

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 privileged 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.

For example

```
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.

Function	Syntax and function	Example
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.
CSRS_MIE	ECALL_CSRS_MIE(void) Secure user-mode emulation of the Machine Status Register (mstatus) MIE bit. Enables all interrupts (PLIC + CLINT) mapped to the zone including the soft timer (trap 0x3). The operation is atomic with respect to the context of the zone.	ECALL_CSRS_MIE();
CSRC_MIE	ECALL_CSRC_MIE(void) Secure user-mode emulation of the Machine Status Register (mstatus) MIE bit. Disables all interrupts (PLIC + CLINT) mapped to the zone including the soft timer (trap 0x3). The operation is atomic with respect to the context of the zone.	ECALL_CSRC_MIE();

Function	Syntax and function	Example

ECALL_CSRW_MTIMECMP	ECALL_CSRW_MTIMECMP (uint64_t)
	Secure user-mode emulation of the machine-mode timer compare register (mtimecmp). Causes a trap 0x3 exception when the mtime register contains a value greater than or equal to the value assigned. Each zone has its own secure instance of timer and trap handler. Per RISC-V specs this is a one-shot timer: once set it will execute its callback function only once. Note that mtime and mtimecmp size is 64-bit even on rv32 architecture. Registering the trap 0x3 handler sets the value of mtimecmp to zero to prevent spurious interrupts. If the timer is set but no handler is registered the exception is ignored.
		// Set the the timer
		uint64_t T = 10; // ms
		uint64_t T0 = ECALL_CSRR_MTIME();
		uint64_t T1 = T0 + T*32768/1000;
		ECALL_CSRR_MTIMECMP(T1);
		// Receive a TRP_0x3 when timer expires
		See further example code in Zone 1

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();
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 8

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

Tick = 10 # ms

Zone = 1 #
    base = 0x40410000; size =   64K; rwx = rx # FLASH
    base = 0x80001000; size =    4K; rwx = rw # RAM
    base = 0x20000000; size = 0x100; rwx = rw # UART

Zone = 2 #
    irq  = 11, 21, 22 # BTN0 BTN1 BTN2
    base = 0x40420000; size =      64K; rwx = rx # FLASH
    base = 0x80002000; size =       4K; rwx = rw # RAM
    base = 0x20005000; size =    0x100; rwx = rw # PWM
    base = 0x20002000; size =    0x100; rwx = rw # GPIO
    base = 0x0C000000; size = 0x400000; rwx = rw # PLIC

Zone = 3 #
    base = 0x40430000; size =   64K; rwx = rx # FLASH
    base = 0x80003000; size =    4K; rwx = rw # RAM
    base = 0x0200BFF8; size =   0x8; rwx = r  # RTC
    base = 0x20002000; size = 0x100; 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().
fence	FENCE – this determines whether fencing is enabled when this zone comes into and leaves context. If no fence command is present at the top of a zone definition 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.
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: irq = [interrupt numbers assigned to zone, separated with a comma]
base	Each zone has (6) available ranges of memory, including mapped peripherals, that can be uniquely assigned to that zone. • The first base is for ROM; size that is a multiple of 4 Bytes, base address 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. • The second base 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. • base 3-6 are for other memory mapped peripherals – base address must be a multiple of the size (NAPOT) or 3 Byte Aligned (4BL) Arguments are: [where x is a number from 1-6]: • base = [physical base address] • size = [It parses byte, K or M] or you can use a hex size such as 0x100 • rwx = any combination of [RWX] – defines memory range permissions Read, Write and Execute

Chapter 9

MultiZone Configurator Command line Options

The MultiZone Configurator is invoked automatically when you run the make command, 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: java -jar multizone.jar [OPTION...] file.hex... [-o file.hex]
Hex Five MultiZone(TM) Configurator

 -c, --config=file.cfg      Config file. Default: multizone.cfg
 -o, --output=file.hex      Output file. Default: multizone.hex
 -a, --arch={E31|E51|...}   Architecture. Default: E31
 -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 10

Errata

Issue
Compatibility with gcc compiler optimizations – gcc optimizations for the risc-v toolchain appear to have the opposite result and often result in errors. el.

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

Issue
When you establish a new serial connection to Zone 4, the telnet / TLS session to Zone 2 is terminated

Work Around
This appears to be an issue in the bitstream which we are working to debug.

The work-around is to re-establish the telnet / TLS session to Zone 2.