MultiZone® Security

Reference Manual

RISC-V

Version	Date	Changes
1.0	July 10, 2020	Initial
1.1	July 29, 2020	Minor edits
1.2	Aug 15, 2020	Clarify PLIC interrupt mapping
1.3	Aug 28, 2020	Add protected DMA test command
1.4	Sep 28, 2020	Add configurator option “load” to run programs in ram

Contents

MultiZone Security Concept

MultiZone® Security is the quick and safe way to add security and separation to RISC-V processors that lack hardware isolation mechanisms and that need finer granularity than one secure partition.

RISC-V processors are increasingly used in general purpose microcontrollers and often embedded in System on Chip (SoC) devices that collectively ship in millions of units annually. Securing these devices has become increasingly difficult as complex new requirements are often met with the addition of readily available third-party software. The RISC-V standard ISA lacks the physical resources necessary to provide separation of trusted and untrusted functionality, thus leading to larger attack surface and increased likelihood of vulnerability. In response, Hex Five has created a software only solution in MultiZone providing security and separation without the need to redesign existing hardware and software, and eliminating the complexity associated with managing a hybrid hardware/software security scheme.

MultiZone provides hardware-enforced software-defined separation of multiple functional areas within the same chip. MultiZone is completely self-contained, exposes an extremely small attack surface, and it is policy-driven, meaning that no coding or security expertise are required. With MultiZone Security open source software, third party binaries, and legacy code can be configured in minutes to achieve unprecedented levels of safety and security.

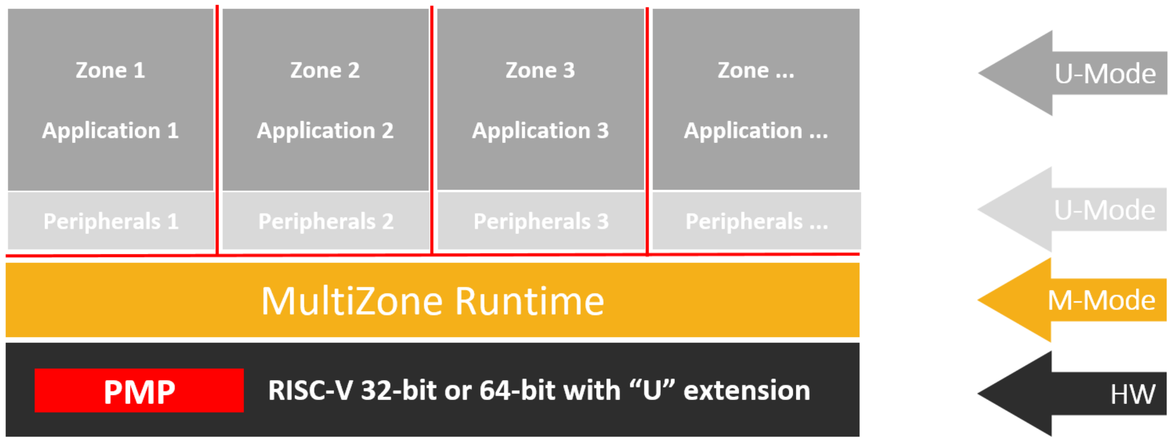
How it works

MultiZone main components include:

- ***MultiZone Runtime*** - a small binary providing separation kernel and secure communications.
- ***MultiZone Configurator*** - a development utility that extends the GNU toolchain.
- ***MultiZone API*** - a free and open API providing static wrappers for system calls.

Unlike traditional system software, no compilation, linking or debugging is required - and in fact even allowed. Instead, these are the three logical steps necessary to secure an existing monolithic application:

STEP 1 - Decompose the monolithic firmware into separate binaries



Decompose the traditional monolithic firmware into a few distinct functional modules called "zones". Good candidates for a typical connected device may include: one zone for the RTOS and its tasks, one zone for the communications stack – by definition exposed to remote attack, one zone for the crypto libraries that interact with keys, certificates, and Root of Trust, and a few bare metal zones to protect access to various system resources like peripherals and I/O.

Each zone is compiled and linked individually, with no cross-reference to other zones, and results in its own self-contained binary. Zones' programs can be written in any language, built with different toolchains, different versions of compilers and libraries, and by different developers at any point in the hardware and software supply chain. Zones expose their functionality as micro-services that communicate with each other via a secure communication layer provided by the MultiZone Runtime.

MultiZone microservices are the secure asynchronous equivalent of traditional synchronous APIs. By design, zones are completely separated hardware threads and don't share any memory, so there is no stack, hype, buffers or pointers for calling functions and passing values and/or references back and forth. Traditional APIs can be easily exposed as microservices by wrapping their code into a simple listener loop that receives input messages from other zones (request), processes the input according to some internal logic, sends back a return message with the output of the call (response), and goes back to sleep waiting for the next request – MultiZone messages are unstructured fixed-length sequences of 16 bytes.

STEP 2 - Define hardware separation policies

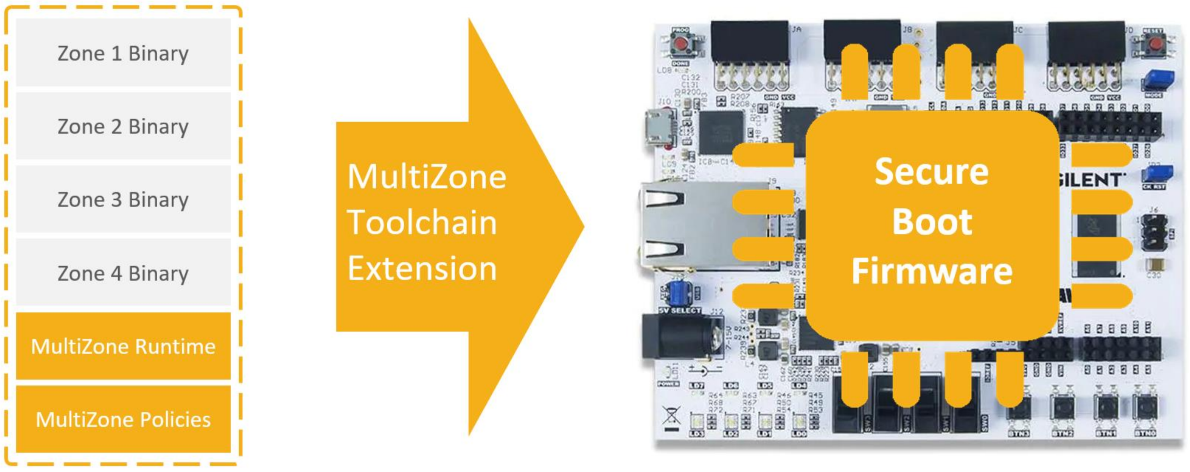
After decomposing the application into separate zones and exposing zones' functionality as message-oriented microservices, the next step is to define the overall hardware separation policies for the whole system. This is done via a simple plain text file named *multizone.cfg*.

```
# Copyright(C) 2020 Hex Five Security, Inc. - All Rights Reserved

# MultiZone reserved memory: 4K @0x20400000, 4K @0x80000000

Tick = 10 # ms

Zone = 1
      plic = 3 # UART (PLIC)
      irq  = 3 # DMA
      base = 0x20402000; size =      32K; rwx = rx # FLASH
      base = 0x80002000; size =       4K; rwx = rw # RAM
      base = 0x10013000; size =    0x100; rwx = rw # UART

Zone = 2
      irq  = 16, 17, 18 # BTN0 BTN1 BTN2 (CLINT)
      base = 0x2040A000; size =    8K; rwx = rx # FLASH
      base = 0x80003000; size =    4K; rwx = rw # RAM
      base = 0x10025000; size = 0x100; rwx = rw # PWM
      base = 0x10012000; size = 0x100; rwx = rw # GPIO

Zone = 3
      base = 0x2040C000; size =    8K; rwx = rx # FLASH
      base = 0x80004000; size =    4K; rwx = rw # RAM
      base = 0x10012000; size = 0x100; rwx = rw # GPIO

Zone = 4
      base = 0x2040E000; size =    8K; rwx = rx # FLASH
      base = 0x80005000; size =    4K; rwx = rw # RAM
```

Listing 1.1. Example of MultiZone policy definition file.

The syntax of the policy file is minimal and intuitive. Each zone is allocated a number of memory-mapped resources identified by start address, size, and any combination of read / write / execute attributes. Resources include contiguous regions of memory for programs, data, peripherals, I/O, and interrupt sources. The configuration file also defines the tick time for the preemptive kernel – default value is 10ms. By default, each zone has transparent access to its own virtual instance of the cpu timer and to all non-maskable software traps. Maskable interrupt sources can't be shared across zones and must be explicitly assigned to the zone responsible for the safe execution of their unprivileged handlers. See chapter 5 for a detailed description of syntax and semantics of the MultiZone separation policies.

STEP 3 – Generate the secure boot image



Run the MultiZone Configurator utility (mutizone.jar) to merge zones binaries with the MultiZone Runtime and to apply the separation policies. This is typically done as the final step of the build process by invoking the configurator utility from the Make file. The output of the configurator is a signed firmware image in standard Intel HEX file format. This SDK simplifies the upload of the firmware to flash via free OpenOCD drivers.

Supported Hardware

MultiZone works with any 32-bit or 64-bit RISC-V standard processors with Physical Memory Protection PMP and U mode.

This version of the GNU-based SDK supports the following development kits:

- Xilinx Artix-7 Arty 35T FPGA Evaluation Kit
- Microchip PolarFire SoC FPGA Icicle Kit

The Arty FPGA Evaluation Kit is certified for the following softcore SoC bitstreams:

- Hex Five X300 RV32ACIMU - Permissive open source free for any use.
- SiFive E31 v19_02 RV32ACIMU - Proprietary. Evaluation license required.
- SiFive S51 v19_02 RV64ACIMU - Proprietary. Evaluation license required.

For instructions on how to upload the bitstream to the ARTY board and how to connect the Olimex debug head ARM-USB-TINY-H see Arty FPGA Dev Kit Getting Started Guide

Installation

The GNU-based MultiZone SDK works with any version of Linux, Windows, and Mac capable of running Java 1.8 or greater. The directions in this document have been verified with fresh installations of Ubuntu 20.04, Ubuntu 19.10, Ubuntu 18.04.5, and Debian 10.5. Other Linux distros are similar. Windows developers may want to install a Linux emulation environment like Cygwin or run the SDK in a Linux VM guest (2xCPU, 2GB Disk, 2GB Ram).

Linux prerequisites

```
sudo apt update
sudo apt install make default-jre gtkterm
```

Ubuntu 18.04 LTS additional dependency:

```
sudo add-apt-repository "deb http://archive.ubuntu.com/ubuntu/ focal main
universe"
sudo apt update
sudo apt install libncurses-dev
```

Note: *GtkTerm is optional and required only to connect to the reference application via UART. It is not required to build, debug, and load the MultiZone software. Any other serial terminal application of choice would do.*

GNU RISC-V Toolchain

Hex Five reference build: RISC-V GNU Toolchain Linux 64-bit June 13, 2020

```
cd ~
wget https://hex-five.com/wp-content/uploads/riscv-gnu-toolchain-
20200613.tar.xz
tar -xvf riscv-gnu-toolchain-20200613.tar.xz
```

OpenOCD on-chip debugger

Hex Five reference build: RISC-V openocd Linux 64-bit June 13, 2020

```
cd ~
wget https://hex-five.com/wp-content/uploads/riscv-openocd-20200613.tar.xztar
-xvf gnu-mcu-openocd-20190827.tar.xz
```

Linux USB udev rules

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

# Future Technology Devices International, Ltd FT2232C Dual USB-UART/FIFO IC
SUBSYSTEM=="tty", ATTRS{idVendor}=="0403",ATTRS{idProduct}=="6010",
MODE="664", GROUP="plugdev"
SUBSYSTEM=="usb", ATTR{idVendor} =="0403",ATTR{idProduct} =="6010",
MODE="664", GROUP="plugdev"

# Olimex Ltd. ARM-USB-TINY-H JTAG interface
SUBSYSTEM=="tty", ATTRS{idVendor}=="15ba",ATTRS{idProduct}=="002a",
MODE="664", GROUP="plugdev"
SUBSYSTEM=="usb", ATTR{idVendor} =="15ba",ATTR{idProduct} =="002a",
MODE="664", GROUP="plugdev"
```

Depending on your system configuration you may need to reboot for these changes to take effect.

MultiZone Security SDK

```
cd ~
wget https://github.com/hex-five/multizone-sdk/archive/master.zip
unzip master.zip
```

Build & load the MultiZone reference application

Connect the target board to the development workstation as indicated in the user manual.

'ls bsp' shows the list of supported targets: X300, E31, S51, PFSOC.

Assign one of these values to the environment variable BOARD - default target is X300.

```
cd ~/multizone-sdk
export RISCV=~/riscv-gnu-toolchain-20200613
export OPENOCD=~/riscv-openocd-20200613
export BOARD=X300
make
make load
```

Note: *With some older versions of the ftdi libraries, the first "make load" after powering the board may take a bit longer than normal. If you don't want to wait, the simple workaround is to reset the FPGA board to abort the openOCD session. If you do this, make sure to kill the openocd process on your* *computer. Subsequent loads will work as expected and should take approximately 10 seconds.*

Run the reference application

Connect the UART port (ARTY micro USB J10) as indicated in the user manual.

On your computer, start a terminal session (GtkTerm) and connect to /dev/ttyUSB1 at 115200-8-N-1.

Hit the enter key a few times until the cursor 'Z1 >' appears on the screen.

Enter 'restart' to display the splash screen.

Hit enter again to print the list of available commands

```
=====================================================================
                       Hex Five MultiZone® Security
    Copyright© 2020 Hex Five Security, Inc. - All Rights Reserved
=====================================================================
This version of MultiZone® Security is meant for evaluation purposes
only. As such, use of this software is governed by the Evaluation
License. There may be other functional limitations as described in
the evaluation SDK documentation. The commercial version of the
software does not have these restrictions.
=====================================================================
Machine ISA   : 0x40101105 RV32 ACIMU
Vendor        : 0x0000057c Hex Five, Inc.
Architecture  : 0x00000001 X300
Implementation: 0x20181004
Hart id       : 0x0
CPU clock     : 64 MHz
RTC clock     : 16 KHz

Z1 > Commands: yield send recv pmp load store exec dma stats timer restart

Z1 >
```

Important: *make sure that switch SW3 is positioned close to the edge of the board.*

Important: *open jumper JP2 (CK RST) to prevent system reset upon UART connection*.

Reference Application

This section describes the reference application included in the MultiZone Security SDK. The system architecture consists of four separate bare-metal applications, each running in its own hardware thread and mapped to its own set of hardware resources. The MultiZone separation kernel enforces hardware-level separation of CPU and memory, policy-based access to I/O peripherals, and unprivileged secure execution of interrupt handlers. The MultiZone messenger provides secure communications across the four zones to allow the system to operate as a whole.

Note*: this reference application is a simplified example to understand the basics of the MultiZone Trusted Execution Environment. A more comprehensive example for advanced user is available under permissive licensing free of charge for any use at https://github.com/hex-five/multizone-secure-iot-stack. It provides a fully functional state-of-the-art secure IoT device including free and open RISC-V microcontroller, MultiZone TEE, TCP/IP, TLS, ECC, MQTT, and OTA firmware updates. The MultiZone Secure IoT Stack works with commercial cloud services like AWS and with self-hosted mqtt brokers like Eclipse Mosquitto.*

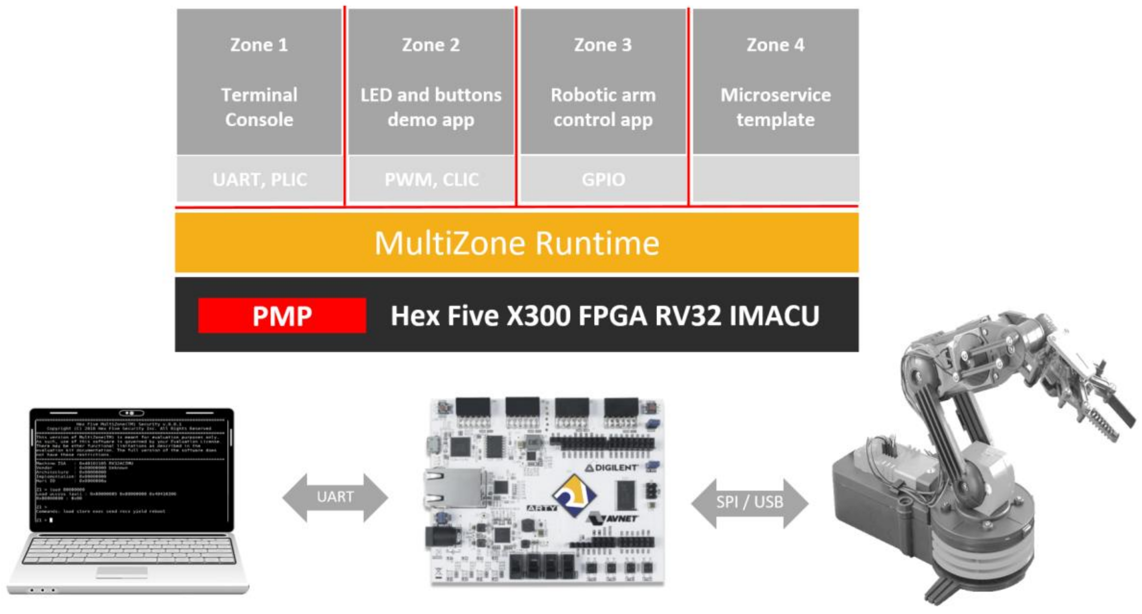


Figure 3.1. MultiZone Reference Application.

Figure 3.1 shows the MultiZone SDK reference application: a typical real time MCU-based industrial application controlling the movements of a robotic arm via a local terminal console. It also includes a set of built-in bare-metal utilities to asses security and separation of the system and to measure performance overhead and interrupt latency of the TEE.

Note: The robotic arm OWI-535 is optional. The only requirement to fully evaluate all the capabilities of the MultiZone Trusted Execution Environment is a serial terminal connected to the target via UART/USB.

Zone 1 connects to the host PC via UART over USB at 115200/8/N/1. Operating in zone 1 is a simple bare metal ANSI terminal application written in C. It presents the user with a command line interface to send and receive messages, to assess the enforcement of the separation policies, and to measure performance overhead of the TEE. To allow for low-power suspend mode, the UART port is driven by interrupt mapped to the PLIC controller.

Zone 2 demonstrates real time multi-tasking, secure user-level interrupt handling and secure messaging. This zone uses RTC timer and PWM peripheral to continuously drive LED1 through the RGB color pallet. It also has three CLIC local interrupts mapped to buttons (BTN0, 1, 2) that cause the LED1 to change color and send a message back to Zone 1. In addition, the service listener implements a few message handlers for testing messages and safety-critical preemptive execution.

Zone 3 operates the OWI robotic arm connected via GPIO lines. User commands are received from zone 1 and the status of the robot reported back via secure messaging. The full list of commands for the robot are listed in Table 3.1.

Zone 4 offers and empty microservice template available for additional user tests.

Robot Operations

The OWI-535 robot has no servomotors or feedback mechanisms. The control application running in zone 3 has no way to detect the initial position of the arm. If the initial position is not the one showed in Figure 3.2, the predefined sequence will not work as expected and will likely overextend the arm’s motors potentially resulting in permanent mechanical damage of the gearboxes. If necessary, once unfolded you may want to use the manual commands listed below to adjust the robot home position as indicated in Figure 3.2.

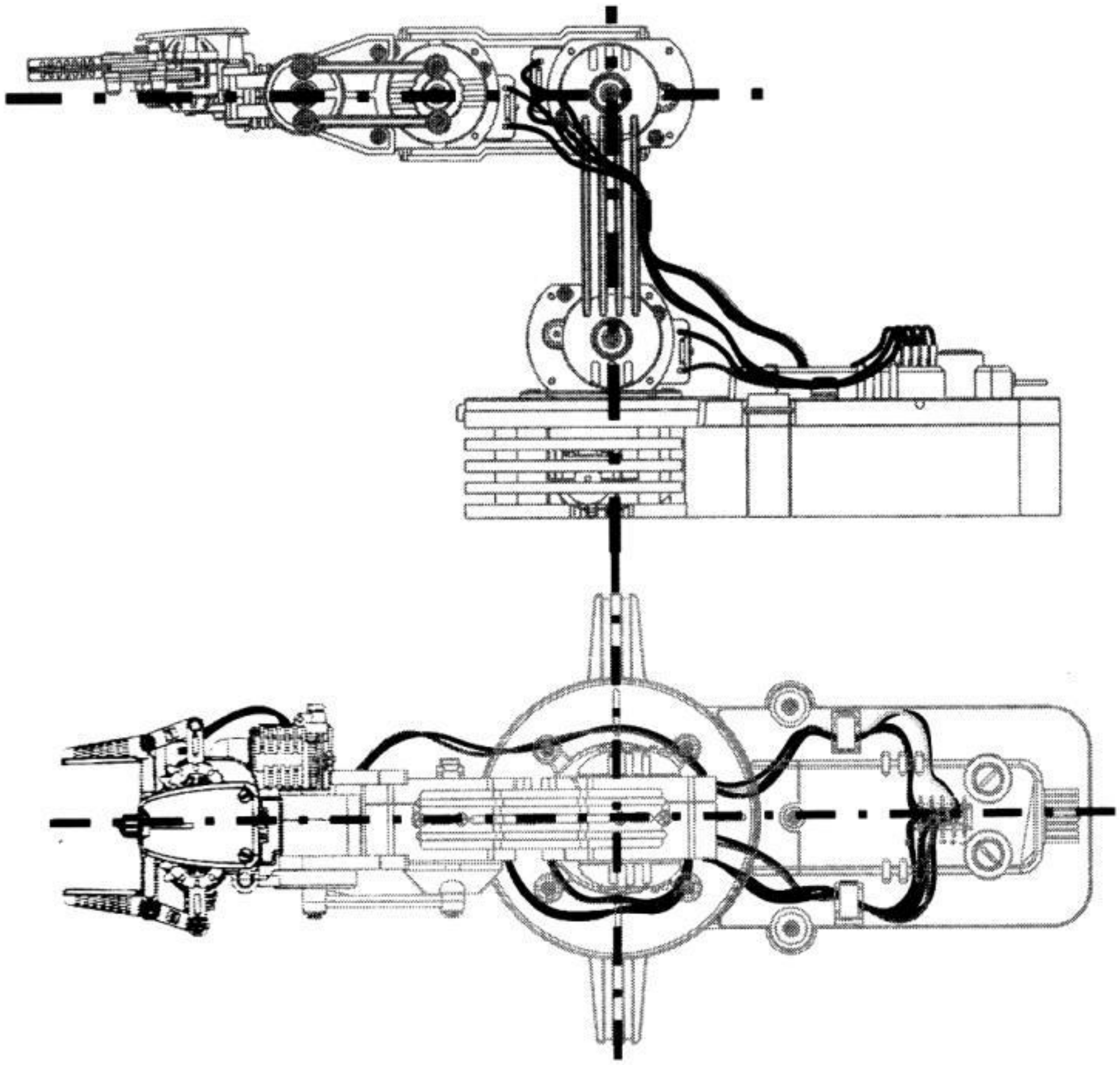


Figure 3.2. Robotic arm home position (unfolded).

Command	Robot Operation
unfold	Deploy the arm for operation – extend to home position
fold	Retract the arm for transport – from home position
start	Start the robotic arm sequence – from home position
stop	Stop the robotic arm sequence – return to home position
q	Close the grip of the robotic arm

a	Open the grip of the robotic arm
w	Rotate the robotic arm wrist up
s	Rotate the robotic arm wrist down
e	Rotate the robotic arm elbow up
d	Rotate the robotic arm elbow down
r	Rotate the robotic arm shoulder up
f	Rotate the robotic arm shoulder down
t	Rotate the robotic arm base clockwise
g	Rotate the robotic arm base counterclockwise
y	Turn the robotic arm light on

Table 3.1. Robotic arm manual commands.

Security and Performance Assessment

At any time, it is possible to enter an empty line to show the list of commands available:

```
Z1 >
Commands: yield send recv pmp load store exec dma stats timer restart
```

- "***pmp***": shows the physical memory protection separation policies in effect for zone 1.
- "***load***" and "***store***": read and write data from/to any physical memory location.
- "***exec***": jumps the zone execution to an arbitrary memory location.
- "***send***" and "***recv***": sends and receives messages to/from any zone.
- "***yield***": yields the CPU to the next zone showing the time taken to loop through all zones.
- "***dma***": submit a protected DMA transfer request
- "***stats***": repeats the yield command ten times and prints detailed system statistics.
- "***timer***": sets the zone timer to current time plus a time delay expressed in milliseconds.
- "***restart***": jumps the zone execution to its first instruction in memory restarting the zone.

At any time, it is possible to type the command “restart” to restart the individual zone and refresh the splash screen – this will not affect in any way the other zones. At any time, it is also possible to use the UP and DOWN arrows to browse and recall up to 10 previously entered commands.

Testing trap & emulation of privileged instructions

All zones' code, including interrupt handlers, runs securely in unprivileged user mode. You can see this from the splash screen showing the values of some privileged CSRs registers otherwise not available in unprivileged user mode: misa, mvendorid, marchid, mimpid, mhartid. Read more about trap & emulation in the "Privileged Instruction" section.

```
Machine ISA   : 0x40101105 RV32 ACIMU
Vendor        : 0x0000057c Hex Five, Inc.
Architecture  : 0x00000001 X300
Implementation: 0x20181004
Hart id       : 0x0
```

Testing hardware-enforced separation

From the terminal session connected to zone 1 type "***pmp***" to show the configuration of the Physical Memory Protection unit as defined by the separation policies defined in the MultiZone configuration file (Listing 1.1).

```
Z1 > pmp
0x20402000 0x20409fff r-x TOR
0x80002000 0x80002fff rw- NAPOT
0x10013000 0x100130ff rw- NAPOT
```

A set of read / write / execute commands is provided to assess the enforcement of the hardware separation policies. Only access compliant with these policies will succeed. Attempts to violate the security boundaries are blocked by the memory protection unit and result in hardware exceptions, which are trapped and displayed on the terminal screen.

1) Valid read from a memory location mapped to zone 1 (hex address):

```
Z1 > load 20402000
0x20402000 : 0x97
```

2) Invalid read from a memory location not assigned to zone 1:

```
Z1 > load 2040A000
Load access fault : 0x00000005 0x20402b1c 0x2040a000
0x2040a000 : 0x00
```

Note*: the trap handler displays the fault code (0x5), the address of the instruction that triggered the fault (0x20402b1c), and the address of the memory location that could not be loaded (0x2040a000).*

3) Valid write to a memory location mapped to zone 1 consistent with read / write policy:

```
Z1 > store 80002000 aa
```

```
0x80002000 : 0xaa
```

Note: *Depending on the memory location chosen for this test, a successful memory write may result in a heap or stack corruption leading to a zone crash. As with buffer overflows and similar out-of-memory situations, this is to be expected. However, observe that the fault is contained in zone 1 and that the other zones continue to operate completely unaffected.*

4) Verify that the written value is in fact stored at the chosen memory location:

```
Z1 > load 80002000
0x80002000 : 0xaa
```

Note: *Depending on the chosen target location, the memory content may change as the result of the code running in zone 1, leading a consecutive load at the same address to display a different value. The same applies to memory-mapped peripherals that may have read-only or write-only registers.*

5) Invalid write to a read-only memory location assigned to zone 1:

```
Z1 > store 20402000 aa
Store access fault : 0x00000007 0x20402b7c 0x20402000
0x20402000 : 0xaa
```

6) Invalid attempt to execute code from a no-execute memory address mapped to zone 1:

```
Z1 > exec 80002000
Instruction access fault : 0x00000001 0x80002000 0x80002000
Press any key to restart
```

Testing protected DMA transfers

The RISC-V Physical Memory Protection mechanism is a facility local to the hart that can't protect access to memory-mapped resource from any bus masters other than the RISC-V core itself. This would represent a major avenue for attack as the PMP schema is completely bypassed in the presence of unprotected bus masters like common DMA controllers. To enforce system separation policies, MultiZone built-in support for protected DMA transfers traps all DMA requests and emulates the PMP logic in software. This is completely transparent to the developer, doesn't require any modifications to existing software, and doesn't add any performance overhead to the transfer itself. In a single-channel DMA configuration, only the zone mapped to the DMA irq can submit transfers and receive transfer-complete interrupts. In a multi-channel configuration, DMA channels are assigned to zones similar to shared PLIC sources. Source, destination, size, and R/W access are checked by the MultiZone runtime. Non-compliant transfer requests are silently ignored.

From the terminal session connected to zone 1 type "***pmp***" to show the configuration of the Physical Memory Protection unit as defined by the separation policies defined in the MultiZone configuration file

(Listing 1.1).

```
Z1 > pmp
0x20402000 0x20409fff r-x TOR
0x80002000 0x80002fff rw- NAPOT
0x10013000 0x100130ff rw- NAPOT
```

The "**dma**" command allows to interactively submit a DMA transfer request for zone 1, which is mapped to the single-channel DMA controller interrupt source 3. Only transfers compliant with the separation policies for zone1 will succeed. Non-compliant requests are silently dropped and ignored. Upon successful completion of the transfer, an interrupt service request is triggered that displays the status of the DMA registers on the terminal screen according to the specs of the DMA block.

1) Valid DMA transfer request from FLASH (read access) to RAM (write access) for a block size of 4 byte. Source, destination, and size values are all in hex format:

```
Z1 > dma 20402000 80002000 4
DMA transfer complete
source : 0x20402004
dest   : 0x80002004
size   : 0x00000000
```

Note: *Depending on source, destination, and size values chosen for the test, a successful DMA transfer may result in heap or stack corruption leading to a zone crash. As with buffer overflows and similar out-of-memory situations, this is to be expected. However, observe that the fault is contained to zone 1 and that the other zones continue to operate completely unaffected.*

Note: *Upon transfer complete, the DMA registers source and destinations are incremented by the size of the transfer (width = 1 byte) and the size register decremented to zero. This may not appear intuitive but it is in fact the specified behavior of the DMA controller implementation.*

2) Verify that the transfer did in fact copy the memory block from source to destination:

```
Z1 > load 0x20402000
0x20402000 : 0x97

Z1 > load 0x80002000
0x80002000 : 0x97

... repeat for the remaining values
```

3) Invalid DMA transfer requests: source and destination regions not accessible from zone 1 and thus request silently ignored

```
Z1 > dma 20401000 80002000 4

Z1 > dma 20400000 80003000 4
```

Testing secure messaging

From the terminal session type “***send***” and “***recv***” to show the syntax of the message-related commands:

```
Z1 > send
Syntax: send {1|2|3|4} message

Z1 > recv
Syntax: recv {1|2|3|4}
```

1) Send a message to zone 1 own inbox and then read the message:

```
Z1 > send 1 hex-five

Z1 > recv 1
msg : hex-five
```

2) Send a “***ping***” message to the other zones to verify they are up and running:

```
Z1 > send 2 ping

Z2 > pong

Z1 > send 3 ping

Z3 > pong

Z1 > send 4 ping

Z4 > pong
```

3) Block a zone by sending a “***block***” message:

```
Z1 > send 2 block
```

4) Send a “***ping***” message to the now blocked zone. Observe that no reply comes back.

```
Z1 > send 2 ping
```

5) Send a second "***ping***" message. Observe that the inbox is now full, as zone 2 is blocked it is not processing incoming requests:

```
Z1 > send 2 ping
Error: Inbox full.
```

Testing safety-critical applications

The MultiZone runtime scheduler implements a dual policy preemptive / cooperative. This guarantees that no faulty zone can bring the system to a halt while allowing for highly responsive real-time applications with minimal interrupt latency. Good behaving zones pause or yield cpu execution while waiting for external interrupts events, timer expiration, or incoming messages. However, if a zone doesn't release the CPU in the maximum allotted time (user configurable, default is 10ms), the zone execution is preempted and execution continues with the next zone.

Note: MultiZone fully support system suspend mode and low-power states. If all zones are waiting for interrupt, the scheduler brings the physical cpu to a low power state by mean of the RISC-V wfi instruction.

1) Enter "***yield***" to release cpu execution and to measure the actual round-trip time for the 4 zones:

```
Z1 > yield
yield : elapsed cycles 319 / time 4us
```

2) Force zone 2 to misbehave (loop idle) by sending a "***block***" request:

```
Z1 > send 2 block
```

3) Enter "***yield***" again. Observe that the yield time has increased by the Tick time (10,000us = 10ms) and that the system is still fully functional despite the unrecoverable zone 2 failure:

```
Z1 > yield
yield : elapsed cycles 647530 / time 10117us
```

4) Change the Tick parameter to 1ms. Rebuild and load the new image. Repeat step number 3 and observe the change in yield time (1,000us = 1ms):

```
Z1 > send 2 block

Z1 > yield
yield : elapsed time 1025us
```

Performance assessment

Two commands are available in zone 1 to measure the system performance of the trusted execution environment: “***yield***” and “***stats***”. The “***yield***” utility measures the round robin trip time to traverse all zone at the current level of computational load. The “***stats***” utility collects the results of 10 consecutive iterations of the yield utility and provides additional real-time statistics about kernel context switch time and interrupt latency.

1. Enter “***yield***” to measure the current system load:

```
Z1 > yield
yield : elapsed cycles 319 / time 4us
```

Note: *A properly designed embedded system should spend most of the time in low power mode awaiting to respond to external events - WFI. In this case, yield cycles and time are indicative of 4 consecutive traversals of the kernel – approx. 80 cycles.*

2. Enter “***stats***” to iterate the measurement and summarize ten rounds of yield – min/med/max:

```
Z1 > stats
313 cycles 139 instr  4 us
319 cycles 134 instr  4 us
319 cycles 134 instr  4 us
319 cycles 134 instr  4 us
319 cycles 134 instr  4 us
319 cycles 134 instr  4 us
319 cycles 134 instr  4 us
319 cycles 134 instr  4 us
319 cycles 134 instr  4 us
319 cycles 134 instr  4 us
-----------------------------------------
cycles min/med/max = 313/319/319
instrs min/med/max = 134/134/139
time   min/med/max = 2/2/2 us
```

3. Observe the MultiZone kernel statistics at the bottom of the screen.

```
kernel
-----------------------------------------
cycles min/max = 62/382
instrs min/max = 35/189
time   min/max = 0/5 us
-----------------------------------------
irq lat cycles = 116
irq lat instrs = 50
time           = 1 us
```

Note: *values smaller than zero are rounded up to one.*

Let’s run a bit of math now: with a tick time of 10ms and the above measured context switch time max of 5 microseconds, the worst-case performance overhead amounts to 5/10,000 = 0.0005 or 0.05%. With a tick time of 1ms, the worst-case overhead is 5/1,000 = 0.005 or 0.5%. Note how this worst-case scenario figures are immaterial for real world applications.

Developing Secure Applications

This section explains in detail the source code of the sample programs running in the four zones. The reference application offers a framework for decomposing traditional monolithic firmware in a number of separate message-oriented microservices to implement high-security applications.

Zones implement a request / response servlet-like pattern: the zone main thread initializes hardware peripherals, enables interrupt sources, starts eventual parallel tasks, and then loops indefinitely in low-power mode waiting for hardware interrupts and/or service requests in the form of messages. When an interrupt is received, execution resumes in the context of the relative interrupt handler - if the individual interrupt source is enabled - and continues with a new iteration of the main listener loop. When a new request message is received, execution resumes with a new iteration of the main listener loop.

Depending on the complexity of the business requirements, message handlers can either be implemented directly in the main loop or in one or more separate functions - similar to interrupt handlers. Either way, message handlers perform some business logic, similar to traditional APIs and return a response in the form of a new message for the requesting zone.

Note: consistently with RISC-V privileged specs, the main thread of a zone paused waiting for interrupt is always resumed by pending interrupts if the global interrupts flag is enabled. This irrespectively of the enabled state of any individual interrupt source. Incoming message events are not maskable and always resume the main thread.

```
/* interrupt handler */
// ...

/* interrupt handler */
// ...

int main (void) {

    /* hardware initialization */
    // ...

    /* listener loop */
    while(1){

        /* message handler */
        msg_handler();

        /* suspend waiting for interrupts and messages */
        MZONE_WFI();
    }

}
```

Listing 4.1. MultiZone microservice reference Implementation.

Listener

As an example, the code snippet below shows the implementation of the listener in *zone2/main.c*. This zone listens only for requests coming from zone number 1 - user input - and ignores other zones, requests.

```
while(1){

    /* Message Listener */
    char msg[16];
    if (MZONE_RECV(1, msg)) {

       /* Message Handler */
       if (strcmp("ping", msg)==0) ... ;
       else if () ...;
       else if () ...;
       else MZONE_SEND(1, msg);

    }

    /* Pause waiting for irqs & mesgs */
    MZONE_WFI();

}
```

Note that in this example the message handler logic is quite simple and implemented directly in the main loop - see next section. More complex functionality is better handled in one or more separate event handler functions outside the listener loop as implemented in zone1/main.c.

The zone suspension block is the last part of the loop. For applications that require continuous processing of the main thread - and that therefore don't go into a low power state, the `MZONE_WFI()` should be replaced with the equivalent `MZONE_YIELD()`. See the Thread Scheduling section in the MultiZone Security API chapter for more detail about these two MultiZone system calls.

Messages

MultiZone implements the service request / response pattern via secure messages. The event handler receives a message in input, processes the message according to its business logic, and eventually sends back a response message to the requesting zone.

For example, the code snippet below shows the implementation of a simple ping / pong service in zone2/main.c.

```
char msg[16];

if (MZONE_RECV(1, msg)) {

    if (strcmp("ping", msg)==0) {
        MZONE_SEND(1, "pong");
    }
```

If a new request message "ping" is received from zone 1 - `MZONE_RECV(1, msg)`, a response message "pong" is sent to the requesting zone - `MZONE_SEND(1, "pong")`. Note that the MultiZone SEND() and RECV() schema is based on an exception handling mechanism that doesn't expose shared memory across zones. MultiZone messages are fixed length 16-byte long data streams. The '\0' terminator is not required although it makes sense if the messages represent strings as in the case of this reference application. MultiZone doesn't define any default message and doesn't require any default listeners. Depending on the business requirements, it is absolutely fine to have sealed zones that exchange no messages at all. It is entirely up to the system designer to define messages and semantic for the target application. Message delivery is not guaranteed - i.e. if the recipient inbox is full. If required by the application, the sender can check the return value of the SEND() and eventually retry or error. Delivery is asynchronous with respect to the sender and will resume the recipient if paused waiting for interrupt. SEND() and RECV() are always non-blocking.

Interrupts and Exceptions

MultiZone implements user-mode secure interrupts in accordance to its zero-trust model. Traditional monolithic operating systems execute interrupt handlers at the highest level of privilege, typically in "kernel mode" drivers. This constitutes a major attack vector for the system as a whole and an unacceptable security risk for high-security safety-critical applications. MultiZone is immune from privilege level escalation attacks as it provides a framework for secure unprivileged execution of interrupt handlers: interrupt sources and their handlers are mapped to zones and executed in the context of the respective zone at the lowest level of privilege, completely separated from kernel and other zones, thus unable by design to compromise the security of the system.

Maskable interrupt sources – aka external interrupts - are mapped to zones on an exclusive basis: only the zone mapped to the interrupt source receives the interrupt and provides the relative handler code. Individual PLIC sources are mapped to zones in the same way as local interrupts. The PLIC external interrupt 11 is assigned automatically by the system to the zones mapped to PLIC sources. Non maskable interrupts – aka software traps - are not mapped to zones: each zone must provide a handler or rely on the MultiZone built-in weak implementation. Note that there is no need to map the timer comparator external interrupt 7 as each zone has its own private instance of the multiplexed cpu – more about the timer in the next section

MultiZone built-in trap & emulation engine transparently supports vectored and non-vector interrupts as defined in the RISC-V privileged specs – both PLIC and CLIC/CLINT.

The code in zone 1 shows an example of non-vectored interrupts. The MTVEC CSR register is initialized in main.c to point to a single trap handler function that manages all maskable and non-maskable interrupts.

```
int main (void) {

       ...
```

```
        /* register machine trap handler */
       CSRW(mtvec, trap_handler);

       ...
```

```
__attribute__(( interrupt(), aligned(4) )) void trap_handler(void){

       const unsigned long mcause = MZONE_CSRR(CSR_MCAUSE);
       const unsigned long mepc   = MZONE_CSRR(CSR_MEPC);
       const unsigned long mtval  = MZONE_CSRR(CSR_MTVAL);

       switch(mcause){

       case 0 : printf("Instruction address missaligned : 0x%08x 0x%08x 0x%08x \n",
                break;

       case 1 : printf("Instruction access fault : 0x%08x 0x%08x 0x%08x \n", mcause,
                break;

        ...
```

Listing 4.2. Zone 1 single trap handler example.

The code in zone 2 shows an example of vectored interrupts. The MTVEC CSR register is initialized in main.c to point to a vector table. The first entry of the table corresponds to the NMI handler function that manages non-maskable software traps. The remaining entries point to a number of maskable interrupt handlers for the external interrupt sources – i.e. PLIC, TIMER, CLIC. Note that according to RISC-V specs, the lsb bit of the vector table base address must be set to '1' to enable the vectored mode.

```
int main (void){

       // vectored trap handler
       static __attribute__ ((aligned(4)))void (*trap_vect[32])(void) = {};

       trap_vect[0] = trp_handler;
       trap_vect[3] = msi_handler;
       trap_vect[7] = tmr_handler;
       trap_vect[BTN0_IRQ] = btn0_handler;
       trap_vect[BTN1_IRQ] = btn1_handler;
       trap_vect[BTN2_IRQ] = btn2_handler;

       CSRW(mtvec, trap_vect);
       CSRS(mtvec, 0x1);

       ...
```

```
__attribute__((interrupt())) void trp_handler(void) {...} // NMI trap handler (0)
__attribute__((interrupt())) void msi_handler(void)  {...} // Software interrupt (3)
__attribute__((interrupt())) void tmr_handler(void)  {...} // Machine timer (7)
__attribute__((interrupt())) void btn0_handler(void) {...} // CLIC button 0
```

Listing 4.3. Zone 2 vectored trap handler example.

System Timer

The RISC-V ISA defines one 64-bit one-shot timer per hart for both rv32 and rv64 architectures. The MultiZone built-in timer engine creates a number of multiplexed private instances of the timer, one for each zone. The MultiZone API exposes four interfaces to read and set the timer. The timer comparator register is used to trigger single shot interrupts routed to irq number 7. Note that there is no need to map the timer interrupt source to any zone as each zone has its own private copy of the timer completely independent from the others.

As an example, the code snippet below shows how the timer is used in zone 2 to continuously change the color of the LED. The timer comparator is initialized to fire after 25ms and the interrupt source enabled. Inside the timer handler the comparator is set again to trigger the next iteration.

```
int main (void) {

    ...

    /* set & enable the timer */
    MZONE_ADTIMECMP((uint64_t)25*RTC_FREQ/1000);
    CSRS(mie, 1<<7);
```

Listing 4.7. Timer initialization.

Listing 4.8 shows the implementation of the timer handler. The exception handler is typically made of two building blocks: one responsible to re-configure the timer compare register value to generate a synchronous periodic interrupt (since the timer generates an asynchronous single-shot interrupt) and the other responsible to implement the expected processing functionalities.

```
__attribute__((interrupt())) void tmr_handler(void)  {

    /* some business logic goes here */
    ...

    /* reset timer (clears mip) */
    MZONE_ADTIMECMP((uint64_t)25*RTC_FREQ/1000);

}
```

Listing 4.8. Soft timer interrupt handler

Note: *according to RISC-V specs, setting the RISC-V timer requires a first read of the 64-bit value of the real-time clock, one 64-bit addition to add the delay, and one 64-bit write to the comparator. In* *addition to discrete read and write APIs for real time clock and comparator, MultiZone provides the optimized `MZONE_ADTIMECMP()` that atomically reads the real time clock registers, adds the desired delay in real-time clock ticks, and then writes the result to the comparator register in one convenient single system call. This is the recommended way to set the timer.*

Privileged Instructions

To guarantee noninterference and temporal and spatial separation, MultiZone executes all programs and interrupt threads in secure unprivileged user mode "U". A key feature of MultiZone is the ability to execute unmodified binaries so that existing applications, libraries, and system level software - like operating systems and their drivers - run in secure unprivileged mode without and change to source code and binary, even if they were designed, compiled, and linked to run at the highest level of privilege in machine mode "M".

To support this key requirement, the MultiZone runtime provides complete trap and emulation of privileged instruction including read / write access to privileged CSRs registers. Note that this is done transparently to the developer: no source code changes ever are required to run unmodified binaries - i.e. legacy applications.

Transparent trap & emulation is great for code reusability, portability and quality. However, it comes at the cost of a few extra cycles necessary to trap the exception and to emulate its unprivileged access. As an entirely optional alternative, the free and open MultiZone API defines high-performance wrappers for these privileged instructions in the form of unprivileged pseudo instructions. These are static C-style macro expansions that translate privileged instructions into faster inline assembly. Listing 4.10 shows an example of use of interrupt-related privileged pseudo instructions.

```
int main (void) {

    ...

    CSRW(mtvec, trap_handler); /* register trap handler                */
    CSRS(mie, 1<<11);          /* enable external interrupts (PLIC)    */
    CSRS(mstatus, 1<<3);       /* enable global interrupts (PLIC, TMR) */
```

Listing 4.10. Privileged pseudo instructions: read / write access to CSRs to enable interrupts.

MultiZone pseudo instructions are defined in the *multizone.h* header file. The full list of privileged pseudo instructions is shown in Table 4.2.

Description	MultiZone macro	RISC-V mnemonic
Atomic R/W	CSRRW(csr, rs)	csrrw rd,csr,rs
Read CSR	CSRR(csr)	csrr rd, csr
Write CSR	CSRW(csr, rs)	csrw csr, rd

Atomic R/W	CSRRW(csr, rs)	csrrw rd,csr,rs
Set bits in CSR	CSRS(csr, rs)	csrs csr, rs
Clear bits in CSR	CSRC(csr, rs)	csrc csr, rs

Table 4.2. MultiZone Pseudo Instructions

Appendix - MultiZone API

This section covers Hex Five's implementation of the free and open MultiZone Security API definition. Consistently with the MultiZone zero trust design philosophy, this API is not implemented in the form of a traditional static or dynamic library that would require shared memory structures like stack and hype. Instead, only a static C header file is provided containing macro expansions into assembly code. Note that this guarantees complete separation as there are no cross-references between zones and MultiZone runtime.

```
#define MZONE_YIELD()
#define MZONE_WFI()

#define MZONE_SEND(zone, msg)
#define MZONE_RECV(zone, msg)

#define MZONE_RDTIME()
#define MZONE_RDTIMECMP()
#define MZONE_WRTIMECMP(val)
#define MZONE_ADTIMECMP(val)

#define MZONE_CSRR(csr)
```

Listing 5.1. MultiZone API header file multizone.h

The API is logically organized in four groups: hardware thread scheduling, secure messaging, timer management, and high-performance access to privileged registers.

Thread Scheduling	
void MZONE_YIELD()	Indicate to the scheduler that this zone has nothing pressing to do. Cause the scheduler to switch execution to the next zone. Note that there is no guarantee that the "next" zone selected equal current zone+1.
void MZONE_WFI()	Pause this zone waiting for interrupts or messages. If all zones are waiting for interrupt, the scheduler puts the CPU in a suspended low power state.
Secure Messaging	
int MZONE_SEND(zone num, *char)	Send a fixed length 16-byte long message to zone num. The return value is 1 if the message is delivered or 0 if the receiving mailbox is full. Upon successful delivery the receiving zone execution is resumed if paused.
int MZONE_RECV(zone num, *char)	Check the inbox for a new message from zone num. If a new message is present, it is copied into the local memory

	pointed by *char mark the inbox ready to receive a new message. Otherwise returns 0.
CPU Timer	
uint64_t MZONE_RDTIME()	Read the 64-bit real time clock value since reset.
uint64_t MZONE_RDTIMECMP()	Read the 64-bit timer comparator register.
void MZONE_WRTIMECMP(uint64_t cycles)	Write the 64-bit timer comparator register.
void MZONE_ADTIMECMP(uint64_t cycles)	Read the time value, increments by ticks, and write to the timer comparator register.
CSRs Read	
unsigned long MZONE_CSRR(csr)	Read the value of the privileged system register csr for this zone. Note that the size of the return value is 32-bit for rv32 and 64-bit for rv64.

Table 5.1. MultiZone Security API signature and description.

Thread Scheduling

These APIs affect the current zone execution. Both yield execution to the next zone according to the scheduler internal policy - fair round robin. Note that the "next" zone is not necessarily current zone + 1.

MZONE_YIELD() is used in a cooperative system to provide optimal system response time. This call is optional as the preemptive scheduler forces the zone thread to yield upon expiration of the tick timer – if set. Note that *MZONE_YIELD()* doesn't pause the zone thread and thus prevents the system from reaching the low-power state.

MZONE_WFI() pauses the execution of the zone indefinitely until an interrupt or a message is received. If all zones are waiting for interrupt, the scheduler puts the core in a suspended low power state suitable to battery operated applications. Note that upon reaching low-power state, zone 1 is automatically put into context to minimize interrupt latency. As a consequence, low-latency interrupt sources should be mapped to zone 1.

Important: *MZONE_YIELD()* imposes less pressure on the scheduler and has a near-native interrupt latency. It is recommended for highly responsive applications where low-power consumption is not a requirement. On the other hand, *MZONE_WFI()* can potentially lead to a few additional interrupt latency cycles for zones different than zone1 and it is the default option for battery operated devices.

Secure Messaging

MultiZone runtime provides a self-contained facility for secure inter-zone communications. It allows zones to exchange fixed length 16-byte long bytes streams on a non-shared memory basis. Delivery is synchronous to the execution of the sender and non-blocking. Upon successful delivery, the recipient zone execution is resumed if paused waiting for interrupt. There is no guarantee of message delivery. It is the responsibility of the sender to retransmit or error. To guarantee noninterference, every zone has on set of inboxes providing one entry for each other zone. These include one inbox for the zone itself that can thus send and receive message to its local loop. Inboxes are statically allocated thus there is no need to open or close these communications streams.

Note: by design, the *MZONE_SEND()* API has no parameter for the sender. The sender zone is intrinsically bound to the message to prevent spoofing.

Note: secure messaging is intended as a replacement for traditional stack-oriented calls. High speed transfers of large amounts of data across zones are better implemented via secure DMA or secure split buffers.

Timer Management

The RISC-V ISA defines one 64-bit one-shot timer per hart for both rv32 and rv64 architectures. The MultiZone built-in timer engine creates a number of multiplexed private instances of the timer, one for each zone. The MultiZone API exposes four interfaces to read and set the timer. The timer comparator register is used to trigger single shot interrupts routed to irq number 7. Note that there is no need to map the timer interrupt source to any zone as each zone has its own private copy of the timer completely independent from the others.

Note: according to RISC-V specs, setting the RISC-V timer requires one read of the 64-bit value of the real-time clock, one 64-bit addition to add the delay, and one 64-bit write to the comparator. In addition to discrete read and write APIs for real time clock and comparator, MultiZone provides the optimized *MZONE_ADTIMECMP()* that atomically reads the real time clock registers, adds the desired delay in real-time clock ticks, and then writes the result to the comparator register in one convenient single system call. This is the MultiZone recommended way to set the timer.

Appendix - MultiZone Policies

This section explains syntax and semantics of the MultiZone policies. The MultiZone SDK stores these policies in the bsp/<platform>/multizone.cfg file. This is a plain text format file that can be modified with any text editor of choice. The content of this file is case insensitive. White space characters are ignored. A typical policy configuration file includes definitions for scheduler and zones including memory mapped resources and interrupts mappings.

```
# MultiZone reserved memory: 4K @0x20400000, 4K @0x80000000

Tick = 10 # ms

Zone = 1
       plic = 3 # UART (PLIC)
       irq  = 3 # DMA (single channel)
       base = 0x20402000; size =      32K; rwx = rx # FLASH
       base = 0x80002000; size =       4K; rwx = rw # RAM
       base = 0x10013000; size =    0x100; rwx = rw # UART

Zone = 2
       irq  = 16, 17, 18 # BTN0 BTN1 BTN2 (CLINT)
       base = 0x2040A000; size =    8K; rwx = rx # FLASH
       base = 0x80003000; size =    4K; rwx = rw # RAM
       base = 0x10025000; size = 0x100; rwx = rw # PWM
       base = 0x10012000; size = 0x100; rwx = rw # GPIO

Zone = 3
       base = 0x2040C000; size =    8K; rwx = rx # FLASH
       base = 0x80004000; size =    4K; rwx = rw # RAM
       base = 0x10012000; size = 0x100; rwx = rw # GPIO

Zone = 4
       base = 0x2040E000; size =    8K; rwx = rx # FLASH
       base = 0x80005000; size =    4K; rwx = rw # RAM
```

Listing 6.1. MultiZone policy definition file multizone.cfg

Comments

Comments are marked with the '#' symbol. The first comment line after the copyright notice is a reminder of the memory regions reserved to the MultiZone runtime. These memory regions cannot be changed or assigned to zones.

Preemptive Scheduler Tick

The *tick* parameter drives the preemptive scheduler. It specifies the maximum amount of time in milliseconds that each zone thread can hold the CPU. If a zone exceeds this limit, the preemptive scheduler suspends the zone execution and moves to the next according to a fair round robin schema. If the *tick* value is set to 0, the scheduler policy becomes fully cooperative: zones must explicitly yield execution via *MZONE_YIELD() or MZONE_WFI()* APIs for the other zones to run. Fully cooperative policy is mainly intended for testing and should not be used in production. Valid range for the tick value is 0 to 1000. The default value of 10ms is appropriate for most embedded application.

Zones

Zones are equivalent to hardware threads as their execution is bound to specific hardware resources. Zones define a logical partitioning of the system including contiguous memory regions, memory-mapped peripherals, and respective interrupt sources.

Zone sections constraints:

- Zones are identified with consecutive numbers starting from 1.
- The maximum number of zones supported by this version of the SDK is 4.
- Zones must be mapped to at least one memory region.
- There must be a zone definition for each binary processed by the multizone.jar toolchain extension.

Memory Regions

Memory regions represent contiguous blocks of memory space and must be explicitly mapped to zones on a white-list basis. Memory region attributes include: start address, size, access control flags, and optional load address for loading programs in ram.

Memory regions constraints:

- On most RISC-V processors, each zone can be mapped to a maximum of 8 memory regions depending on a combination of base address and size of each regions. The RISC-V standard ISA defines three types of regions: naturally aligned four-byte 4-byte (NA4), naturally aligned power-of-two (NAPOT), and top of range (TOR). NA4 and NAPOT impose base address and size constrains and consume one PMP register each. TOR regions allow for any combination of base and size but consume two PMP register each. Although MultiZone hides all the complexity related to the proper implementation of the underlying Physical Memory Protection schema, it is ultimately up to the system designer to choose base addresses and sizes that meet both business requirements and system resources.
- Valid region size range is 4 bytes to 2^{32} bytes for rv32 or to 2^{64} bytes for rv64. Size can be indicated in decimal and hexadecimal notation. The modifiers 'K', 'M', 'G' express sizes in kilobytes (2^{10}), megabytes (2^{20}), and gigabytes (2^{30}).
- Region access policy can be any combination of read (r), write (w), execute(x), or no access (---).

- Zone execution starts at the base address of the first memory region, which should contain the program text segment and the ‘rx’ policy. An optional “load” address allows to load programs in ram for faster execution - see example below.

- Memory regions can overlap across zones. This might be required in some particular use cases that require sharing of peripherals. In general, regions overlapping is not recommended and triggers a warning.

```
Warning: zone 3 range 2 overlaps zone 2 range 3.
```

Example: running zone 4 in ram for faster execution.

- Source address in flash: 0x2040E000
- Destination address in ram: 0x80006000

File multizone-sdk/bsp/X300/multizone.cfg:

```
Zone = 4
      base = 0x80006000; size = 8K; rwx = rx; load = 0x2040E000 # PROG IN RAM
      base = 0x80005000; size = 4K; rwx = rw                    # DATA IN RAM
```

File multizone-sdk/zone4/linker.lds:

```
MEMORY {
  flash (rxai!w) : ORIGIN = flash + 0xE000, LENGTH = 8K
  prog  (rxai!w) : ORIGIN = dtim  + 0x6000, LENGTH = 8K
  ram   (wa!xri) : ORIGIN = dtim  + 0x5000, LENGTH = 4K
}
```

Interrupt Sources

Interrupts are unique asynchronous events, typically associated with peripherals, indicating the need for immediate attention. MultiZone implements the concept of user-mode interrupts enabling the secure (unprivileged) execution of hardware interrupt handlers. Interrupts sources must be explicitly assigned to zones. Note that cpu timer source 7, which is private and thus always available to each zone, and PLIC source 11, which is available to any zone mapped to PLIC sources, can't be explicitly assigned to any zones.

Interrupts constraints:

- Interrupt sources are uniquely assigned to zones. It is not possible to assign the same interrupt source to multiple zones.

- External interrupts 7 (timer) and 11 (PLIC) cannot be assigned to any zones as they are automatically assigned by the system.

- Multiple interrupt sources can be assigned to a zone as a comma separated list.

- Valid IRQ values include: 3 (user software) and 16 to 31 (local CLIC/CLINT).

- Valid PLIC values include 0 to 31 on RV32 and 0 to 63 on RV64.

Appendix - MultiZone Toolchain Extension

MultiZone provides the multizone.jar toolchain extension to merge zones binaries, security policies, and MultiZone runtime into a signed firmware image. The Multizone SDK invokes this utility in the final step of the Make script. If a build system other than make is used, the utility can be executed as a standalone command line.

Note: the toolchain extension multizone.jar is provided as a portable signed jar file compatible with any development environment capable of running Java 1.8 or higher.

Note: the optional parameter --boot allows to specify an arbitrary binary file in HEX format to be executed at system boot before starting the zones threads. This code may be provided as part of the Board Support Package or generated dynamically by various IDE tools. It is intended to bring up and configure hardware blocks like PLLs, IO, IRQs/IC, watchdog, FPGA fabric, etc. Important: this code is executed in privileged mode, with no memory protection constraints, and with preemptive scheduler disabled. If present, it should be considered integral part of the trusted code base.

Syntax

```
Usage: java -jar multizone.jar [OPTION...] file.hex... [-o file.hex]
Hex Five MultiZone Configurator

 -c, --config file.cfg  Config file.  Default: multizone.cfg
 -o, --output file.hex  Output file.  Default: multizone.hex
 -b, --boot   file.hex  Boot file.    Default: none
 -a, --arch {X300|...}  Architecture. Default: X300
 -q, --quiet            Don't produce any output
 -?, --help             Give this help list
 -V, --version          Print version info

Example: java -jar multizone.jar zone1.hex zone2.hex zone3.hex zone4.hex
Report bugs to <bug@hex-five.com>
```

Listing 7.1. MultiZone toolchain extension syntax