HEX-Five®

MultiZone® Security

Reference Manual

Version	Date	Changes
1.0	May 16, 2020	Initial Release
1.1	May 20, 2020	Review
1.2	May 22, 2020	Update: figures, listings, and tables
1.3	June 8, 2020	Update: Appendix Toolchain Extension (--boot option)

Contents

MultiZone Security Concept

MultiZone® Security is the quick and safe way to add security and separation to Arm® Cortex-M® devices that lack hardware isolation mechanisms like TrustZone® or that need finer granularity than one secure world.

Cortex-M processors are widely used in general purpose microcontrollers and are often embedded in System on Chip (SoC) devices that collectively ship in billions 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. Legacy designs lack 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.

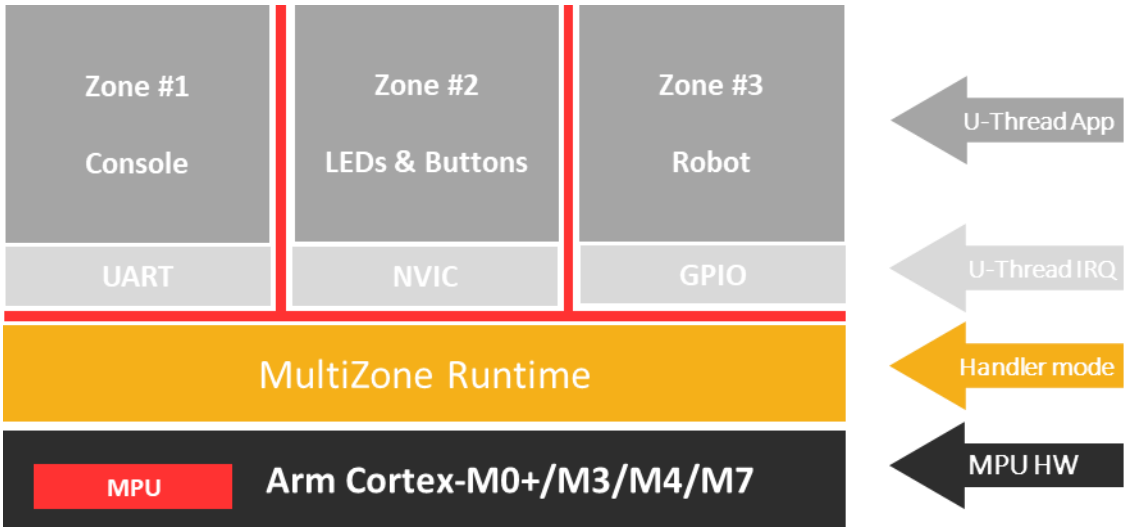
How it works

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 to secure a traditional monolithic application:

STEP 1 - Break 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 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

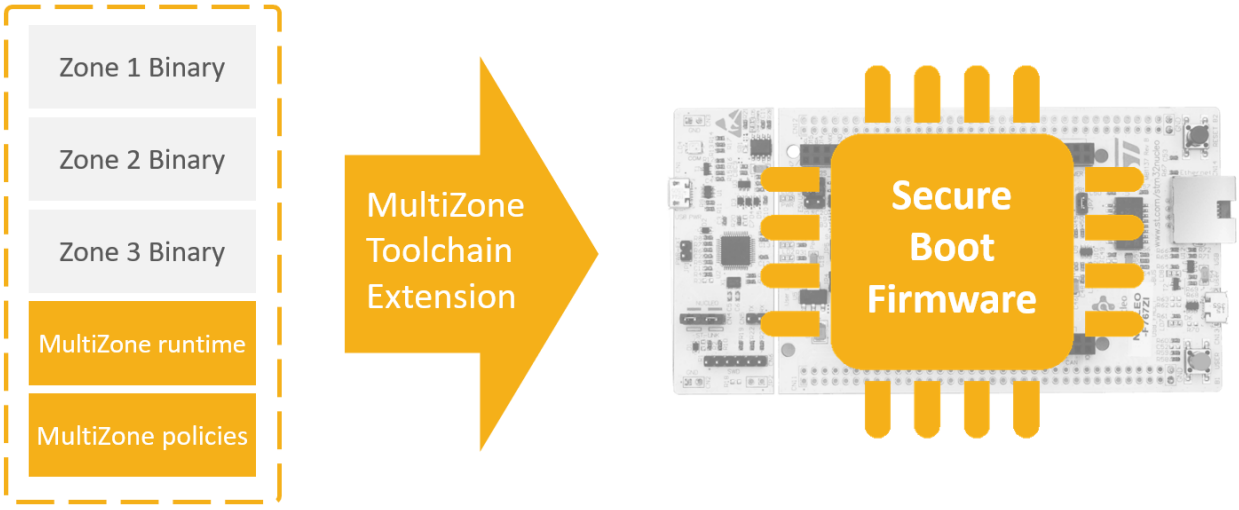
After decomposing the application into separate zones and exposing zones' functionality as message-oriented microservices, the next step is to define the overall hardware separation policies for the whole system. This is done via a simple plain text file named multizone.cfg.

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

# MultiZone reserved memory: 8K @0x08000000, 6K @0x20000000

Tick = 10 # ms

Zone = 1
       irq = 55
       base = 0x08008000; size =   32K; rwx = rx # FLASH
       base = 0x20002000; size =    4K; rwx = rw # RAM
       base = 0x40023800; size =  0x80; rwx = rw # RCC
       base = 0x40020C00; size =  0x40; rwx = rw # GPIOD
       base = 0x40004800; size =  0x40; rwx = rw # USART3

Zone = 2
       irq = 56
       base = 0x08010000; size =   32K; rwx = rx # FLASH
       base = 0x20003000; size =    4K; rwx = rw # RAM
       base = 0x40023800; size =  0x80; rwx = rw # RCC
       base = 0x40020400; size =  0x40; rwx = rw # GPIOB
       base = 0x40020800; size =  0x40; rwx = rw # GPIOC
       base = 0x40013800; size =  0x20; rwx = rw # SYSCFG
       base = 0x40013C00; size =  0x20; rwx = rw # EXTI

Zone = 3
       base = 0x08018000; size =   32K; rwx = rx # FLASH
       base = 0x20004000; size =    4K; rwx = rw # RAM
       base = 0x40023800; size =  0x80; rwx = rw # RCC
       base = 0x40021000; size =  0x40; rwx = rw # GPIOE
       base = 0x40021400; size =  0x40; rwx = rw # GPIOF
```

Listing 1.1. 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 Cortex-M processor that has a Memory Protection Unit including most Cortex-M0+, Cortex-M3, Cortex-M4, and Cortex-M7 microcontrollers and systems on a chip.

This version of the MultiZone SDK is certified for the following hardware targets:

- GigaDevice GD32307C-EVAL evaluation kit (GD32F207 Cortex-M4 @120MHz)
  https://www.gigadevice.com/products/microcontrollers/gd32-development-tools/gd32-evaluation-boards/

- Microchip SMART SAM E70 Xplained evaluation kit (ATSAME70Q21 Cortex-M7 @300MHz)
  http://www.microchip.com/developmenttools/productdetails/atsame70-xpld

- NXP MIMXRT1020-EVK (i.MX RT1020 Cortex-M7 @500MHz)
  https://www.nxp.com/design/development-boards/i.mx-evaluation-and-development-boards/i.mx-rt1020-evaluation-kit:MIMXRT1020-EVK

- Renesas EK-RA6M3 (R7FA6M3AH3CFC Cortex-M4 @120MHz)
  https://www.renesas.com/us/en/products/software-tools/boards-and-kits/eval-kits/ek-ra6m3.html

- ST NUCLEO-F767ZI (STM32F767ZI Cortex-M7 @216MHz)
  https://www.st.com/en/evaluation-tools/nucleo-f767zi.html

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 are specific to a fresh installation of Ubuntu 18.04.4 LTS. Other Linux distros are similar.

Linux prerequisites

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

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 Arm Embedded Toolchain

Hex Five reference version: 8-2019-q3-update Linux 64-bit July 10, 2019

```
cd ~
wget http://developer.arm.com/-/media/Files/downloads/gnu-rm/8-2019q3/RC1.1/gcc-arm-
none-eabi-8-2019-q3-update-linux.tar.bz2
tar -xvf gcc-arm-none-eabi-8-2019-q3-update-linux.tar.bz2
```

OpenOCD on-chip debugger

Hex Five reference build: GNU MCU Eclipse 0.10.0

```
cd ~
wget http://hex-five.com/wp-content/uploads/2020/02/gnu-mcu-openocd-20190827.tar.xz
tar -xvf gnu-mcu-openocd-20190827.tar.xz
```

Linux USB udev rules

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

# STMicroelectronics ST-LINK/V2.1
SUBSYSTEM=="tty", ATTRS{idVendor}=="0483",ATTRS{idProduct}=="374b", MODE="664",
GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="0483",ATTRS{idProduct}=="374b", MODE="664",
```

```
GROUP="plugdev"

# Atmel Corp. Xplained Pro board debugger and programmer
SUBSYSTEM=="tty", ATTRS{idVendor}=="03eb",ATTRS{idProduct}=="2111", MODE="664",
GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="03eb",ATTRS{idProduct}=="2111", MODE="664",
GROUP="plugdev"

# NXP MIMXRT1020-EVK
SUBSYSTEM=="tty", ATTRS{idVendor}=="0d28",ATTRS{idProduct}=="0204", MODE="664",
GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="0d28",ATTRS{idProduct}=="0204", MODE="664",
GROUP="plugdev"

# CMSIS-DAP compatible adapters
ATTRS{product}=="*CMSIS-DAP*", MODE="664", GROUP="plugdev"
```

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

MultiZone SDK

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

Build & load the reference application

Connect the target board to the development workstation as indicated in the board documentation..

'ls bsp' shows the list of supported targets: ATSAME70, EKRA6M3, IMXRT1020, STM32F767, GD32307.

Assign one of these values to the BOARD variable - default is STM32F767.

```
cd ~/multizone-sdk-arm
export GNU_ARM=~/gcc-arm-none-eabi-8-2019-q3-update/bin
export OPENOCD=~/gnu-mcu-openocd-20190827
export BOARD=ATSAME70
make
make load
```

Run the reference application

Connect the board UART port as indicated in the board documentation.

Start the serial terminal console, i.e. GtkTerm, and connect to /dev/ttyACM0 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 show 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.
================================================================
Implementer    : 0x41, Arm.
Variant        : 0x1, Revision 1.
PartNo         : 0xC27, Cortex-M7.
Revision       : 0x0, Patch 0.

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

Z1 >
```

Reference Application

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

Note: This version of the SDK supports up to four zones. The reference application only needs three.

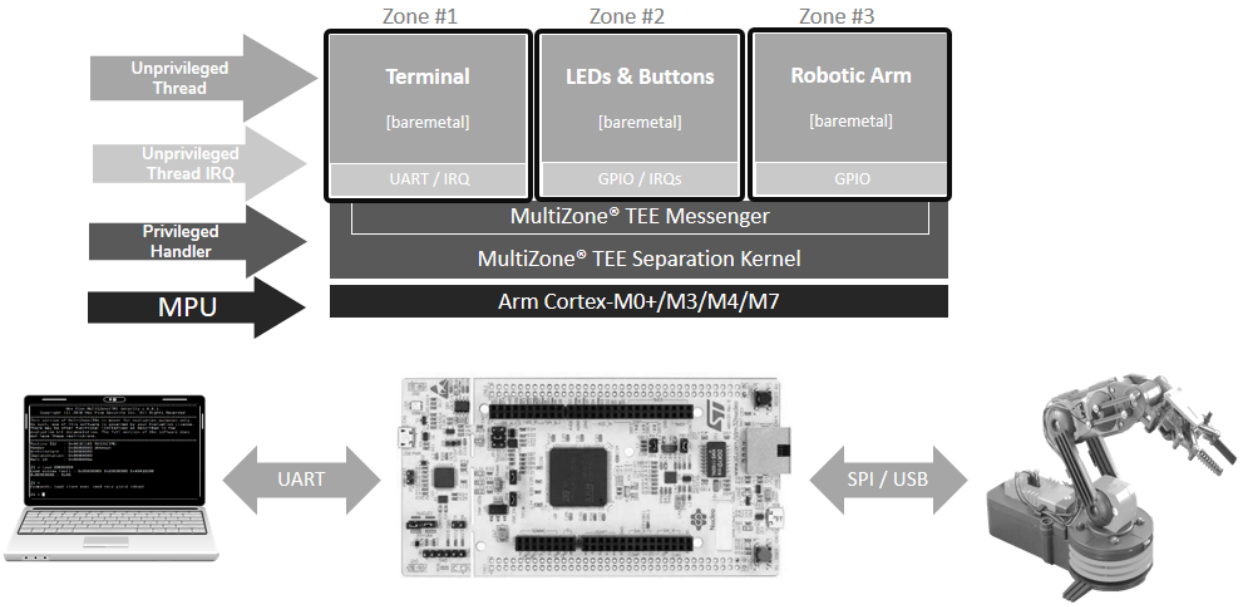


Figure 3.1. MultiZone Reference Application.

Figure 3.1 shows the MultiZone reference application: a typical 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 commands to test security and separation of the system and to measure performance overhead and interrupt latency.

Note: The robotic arm is optional. The only requirement is a serial terminal connected to the board.

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.

Zone 2 blinks an LED and interfaces with a local button to demonstrate real time multi-tasking, secure user-level interrupt handling and secure messaging. It has one interrupt mapped to button BTN1 (USER) that turns on LED LD1 for 3 seconds and sends a notification message to zone 1. In addition, the service listener implements a few message handlers for testing messages and preemptive execution.

Zone 3 operates the 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 the robotic arm commands are depicted in Table 3.1.

Robot Operations

The robotic arm 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 expected, the predefined sequence will likely overextend the arm's motors and potentially result in permanent mechanical damage of the gearboxes. If necessary, you may want to use the manual commands below to adjust the robot home position as indicated in the Figure 3.2.

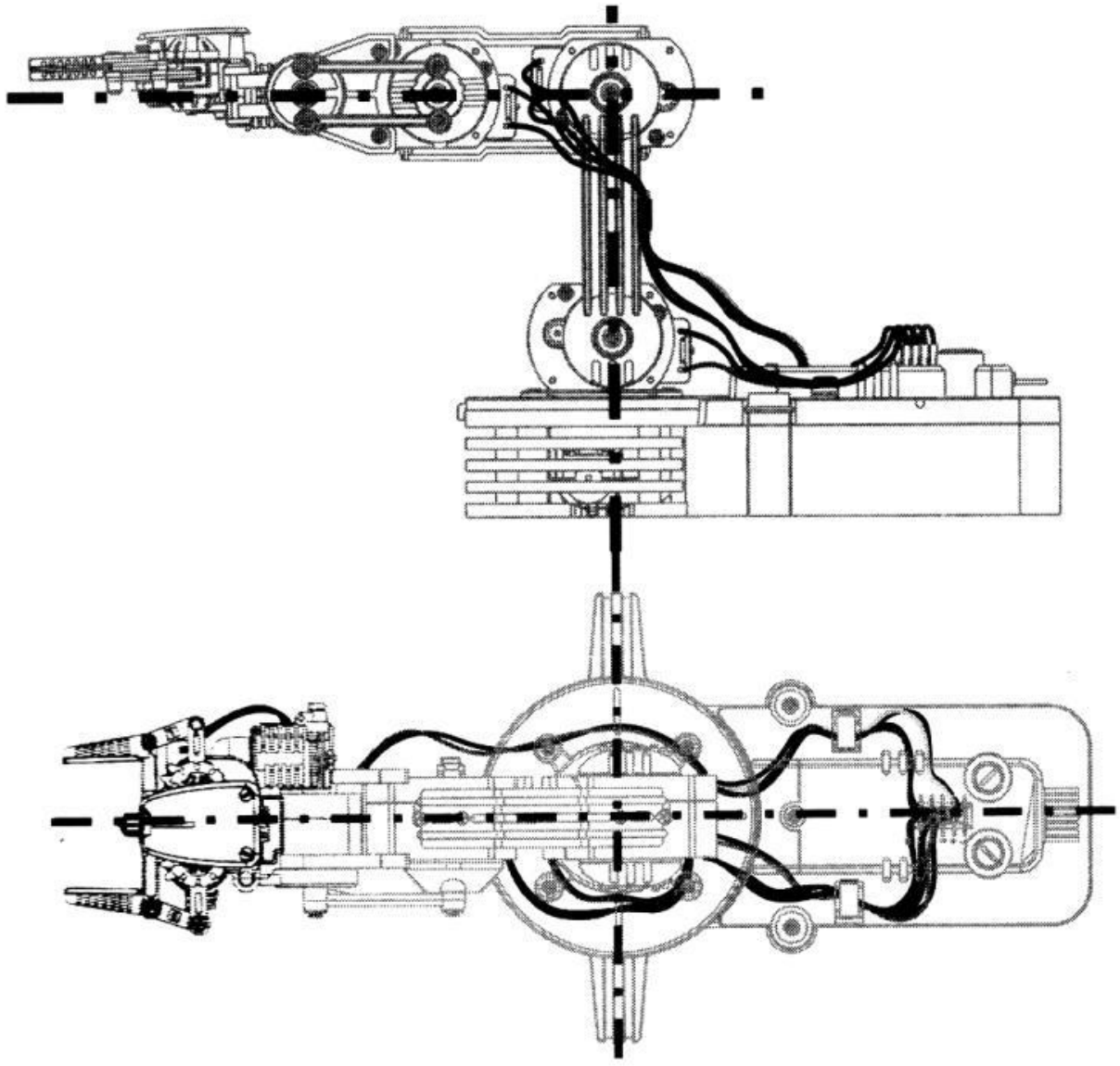


Figure 3.2. Robotic arm home position.

Command	Robot Operation
start	Start the robotic arm operation sequence (assumes unfolded home position)
stop	Stop the robotic arm operation sequence
fold	Fold the robotic arm (assumes unfolded home position)
unfold	Unfold the robotic arm (assumes folded 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 mpu load store exec stats timer restart
```

- "***mpu***": shows the separation policies 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.
- "***stats***": repeats the yield command ten times and prints detailed kernel statistics.
- "***timer***": sets the zone timer to current time plus a time delay expressed in milliseconds.

- "***restart***": jumps the zone execution to its first address 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 arrow to recall previously entered commands.

Secure execution

All zones' code - including interrupt handlers - run securely in Unprivileged Thread mode. You can see this from the splash screen showing the content of privileged registers otherwise not available in Unprivileged Thread mode – CPUID which specifies the Implementer (0x41, Arm), Variant (0x1, Revision 1), PartNo (0xC24, Cortex-M4), and Revision (0x1, Patch 1).

```
Implementer    : 0x41, Arm.
Variant        : 0x0, Revision 0.
PartNo         : 0xC24, Cortex-M4.
Revision       : 0x1, Patch 1.
```

Memory Protection

From any terminal session type "***mpu***" to show the zone isolation policies defined in the MultiZone configuration file (Listing 1.1).

```
Z1 > mpu

0x08008000 0x0800FFFF r-x
0x20002000 0x20002FFF rw-
0x40023800 0x4002387F rw-
0x40020C00 0x40020C3F rw-
0x40004800 0x4000483F rw-
```

A set of read / write / execute commands is provided to assess physical memory separation. Only access compliant with the policies is allowed. 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) Read from a memory address mapped to zone 1:

```
Z1 > load 0x08008001
0x08008001 : 0x30
```

2) Read from a memory address not mapped to zone 1:

```
Z1 > load 0x08007FFF
```

```
Memory protection fault : 0x080096a6
Press any key to restart ...
```

3) Write to a memory address assigned to zone 1 with policy read & write:

```
Z1 > store 0x20002FFF aa
0x20002fff : 0xaa
```

Note: *Depending on target implementation and test address, a successful memory write may corrupt the heap or the stack of the zone leading to an expected application "crash". Note that in this case the other zones are completely unaffected.*

4) Read from the same memory address to verify the value in memory was modified:

```
Z1 > load 0x20002FFF
0x20002fff : 0xaa
```

Note: *Depending on your target address you may have changed a memory address which can be modified by the Zone at runtime, leading a consecutive load to the same address to not display the value that you have stored. The same rationale applies to memory-mapped peripherals that may have read-only or write-only registers.*

5) Write to a read and execute memory address assigned to zone 1:

```
Z1 > store 0x08008000 aa
Memory protection fault : 0x0800970a
Press any key to restart ...
```

6) Read from a memory address belonging to a memory-map peripheral (e.g., USART3):

```
Z1 > load 0x40004800
0x40004800 : 0x2d
```

Note: *Write operations to* memory-mapped peripheral registers may change the configuration of the device and lead to unexpected and unintended behaviours. *Write operation to read-only registers may have no effect.*

Secure Messaging

From any terminal session type "***send***" and "***recv***" to display the messaging-related commands syntax:

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

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

1) Send a message to the local zone and observe the local reply:

```
Z1 > send 1 hex-five

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

2) Send a "***ping***" message to each zone and observe the local reply:

```
Z1 > send 2 ping

Z2 > pong

Z1 > send 3 ping

Z3 > pong
```

3) Block a zone by sending a "***block***" message:

```
Z1 > send 2 block
```

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

```
Z1 > send 2 ping
```

5) Send a second ping. Observe that the inbox is full, because zone 2 is blocked and cannot process incoming messages:

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

Safety Critical Applications

The MultiZone runtime implements a dual policy preemptive / cooperative scheduler. This guarantees that no faulty zone can stop the system while allowing for highly responsive real-time applications with minimal interrupt latency. Zones normally yield context when waiting for external events. If the zone doesn't release the CPU in the maximum allotted time (default is 10ms), the zone execution is preempted and execution continues with the next zone.

1) Enter "***yield***" to measure the current round robin time:

```
Z1 > yield
yield : elapsed time 25us
```

Note: *If the yield displays the "n/a" message, please reset the board.*

2) Block zone 2 by sending a "***block***" message:

```
Z1 > send 2 block
```

3) Enter "***yield***" again. Observe the yield time increase equal to the Tick time (10,000us = 10ms):

```
Z1 > yield
yield : elapsed time 10025us
```

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 Statistics

Zone 1 makes available two specific commands to measure the performance of the system: "***yield***" and "***stats***". The "***yield***" measures the round robin trip time for all zones while the "***stats***" measures and summarizes ten rounds of the yield and provides additional runtime statistics (kernel context switch time and interrupt latency).

1. Enter "***yield***" to measure the current round robin trip time for the 3 zones:

```
Z1 > yield
yield : elapsed time 25us
```

Note: *All zones are in the WFI state.*

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

```
Z1 > stats
200 instr 400 cycles 25 us
```

```
188 instr 376 cycles 23 us
188 instr 376 cycles 23 us
188 instr 376 cycles 23 us
188 instr 376 cycles 23 us
188 instr 376 cycles 23 us
188 instr 376 cycles 23 us
188 instr 376 cycles 23 us
188 instr 376 cycles 23 us
188 instr 376 cycles 23 us
188 instr 376 cycles 23 us
---------------------------------------
instrs  min/med/max = 188/188/200
cycles  min/med/max = 376/376/400
time    min/med/max = 23/23/25 us
```

Note: *If the stats display "0" reset the target.*

3. Observe MultiZone runtime context switch and interrupt latency metrics at the very bottom (Kernel) – i.e. ***cycles*** = 164 and ***irq lat cycles*** = 54.

```
Kernel
---------------------------------------
instrs = 98
cycles = 164
time   = 10 us
---------------------------------------
irq lat instrs = 35
irq lat cycles = 50
time           = 3 us
```

Note: *Considering an operation frequency of 16MHz, the exact context switch time is 10.25 microseconds and the interrupt latency is 3.125 microseconds.*

4) Block zone 2:

```
Z1 > send 2 block
```

5) Run the "***stats***" command again. Observe MultiZone runtime context switch time (Kernel) has dropped to 143 cycles. The reason behind the drop is related to the fact zone 2 is now not going through the WFI (because it is blocked and so looping indefinitely), which imposes less pressure on the scheduler.

```
Kernel
---------------------------------------
instrs = 74
cycles = 143
time   = 8 us
---------------------------------------
```

```
irq lat instrs = 35
irq lat cycles = 50
time           = 3 us
```

Note: *Considering an operation frequency of 16MHz, the exact context switch time is 8.9375 microseconds.*

To exercise a bit the impact of MultiZone runtime in the overall performance of the system let's consider a different system tick values. Considering the CPU is operating at the maximum frequency, i.e. 216MHz, for a Tick time of 10ms, the worst case performance overhead is 0.00759%. For a Tick time of 1ms, the expected worst case performance overhead is 0.0759%. Observe that the overhead is immaterial for real world applications.

Developing Secure Applications

This section explains in detail the source code of the programs running in the zones. It offers a framework for decomposing traditional monolithic firmware in a number of separate message-oriented micro services to implement highly-scalable secure 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 waiting for hardware interrupts and service requests in the form of messages. When an interrupt is received, execution resumes in the respective interrupt handler - if the 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: the main thread of a zone paused waiting for interrupt is always resumed by pending interrupts if the global interrupts flag is enabled. This irrespective of the enabled state of individual interrupt sources. Messages cannot be masked 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. 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 1 - user input - and ignores any other zone's requests.

```
while(1){

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

        // Message Handler
       // ...

    }

    // Wait For Interrupt
    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 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 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 sends 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 MultiZone SEND()/RECV() schema is based on an exception handling mechanism that doesn't expose shared memory across zones.

MultiZone messages are 16-byte fixed length bytes streams. MultiZone does not define any message. It is entirely up to the system designer to define message structure 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 should check the return value of the SEND() and eventually retry or error. Delivery is asynchronous to the sender and will resume the recipient if paused waiting for interrupt. SEND() is non-blocking.

Interrupts and Exceptions

MultiZone implements user-mode 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 and an unacceptable security risk for the MultiZone Trusted Execution Environment. Instead, MultiZone 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 and completely separated from the other zones - unable to compromise the security of the system.

Maskable interrupt sources are mapped to zones on an exclusive basis: only the zone mapped to the interrupt source receives the interrupt and provides the handler. Non maskable exceptions are not mapped to zones; every zone must provide a handler or rely on the MultiZone built-in weak implementation.

Listing 4.2 shows the definition of the zones' vector table. The vector table is defined in the bsp/*crt0.S* file, which is specific to the target platform. Per Armv7-M specification, the vector table defines the MSP (first entry), system exceptions (second to sixteenth entries), and interrupt handlers (remaining entries). In the MultiZone reference application, the vector table definition is common across zones although each zone can define its own vector table according to business requirements.

```
# ------------------------------------------------------------
.section        .text.isr_vector
.align                  2
.globl __isr_vector
# ------------------------------------------------------------
__isr_vector:
        .word   __end_stack             // MSP definition
        .word   reset_handler           // Reset_Handler
        .word   nmi_handler             // NMI_Handler
        .word   hardfault_handler       // HardFault_Handler
        .word   memmanage_handler       // MemManage_Handler
        .word   busfault_handler        // BusFault_Handler
        .word   usagefault_handler      // UsageFault_Handler
        .word   0                       // Not Defined
        .word   0                       // Not Defined
        .word   0                       // Not Defined
        .word   0                       // Not Defined
        .word   svc_handler             // SVC_Handler
        .word   debugmon_handler        // DebugMon_Handler
        .word   0                       // Not Defined
        .word   pendsv_handler          // PendSV_Handler
```

```
        .word   systick_handler         // SysTick_Handler
        .rept   39                      // Repeat for
        .word   _def_handler            // IRQ0_Handler to IRQ38_Handler
        .endr
        .word   uart_handler            // USART3_Handler
        .word   btn_handler             // EXTI15_10_Handler
        .rept   69                      // Repeat for
        .word   _def_handler            // IRQ41_Handler to IRQ109_Handler
        .endr
```

Listing 4.2. Vector table for STM32F767

Each zone implements its exception handler for NMI, HardFault, MemManage, BusFault, UsageFault, SVC, PendSV, and SysTick. For example, listing 4.3 shows the implementation of the MemManage fault handler for zone 1 - the one used to assess memory protection policies. MultiZone provides a generic *weak* implementation of all exception handlers in the *crt0.S* so for zones that don't require specific handler logic - see for example zone 2 and zone 3 that don't provide any handlers for MemManage.

Note: exception handlers names are fixed as defined in the vector table - see identifiers in bold.

```
void memmanage_handler(void) __attribute__ ((interrupt ("irq")));
void memmanage_handler(void){

    register uint32_t *sp __asm ("sp");

    uint32_t pc = *(sp+10);

    printf("Memory protection fault : 0x%08x \n", pc);

    ...
}
```

Listing 4.3. Memory management fault handler

Listing 4.4 shows a code snippet of the implementation of the interrupt handler for the peripheral USART3. The USART3 interrupt source (i.e., irq 55) is mapped via MultiZone configuration policies to zone 1. Therefore USART3 interrupts are routed exclusively to zone 1, which implements the *uart_handler* executed in the context (scope) of zone 1 - see *zone1/main.c*).

These are the steps required to process an interrupt source:

1. map the interrupt source to the target zone in the multizone.cfg file
2. replace the built-in handler name in the crt0.S file
3. provide an interrupt handler implementation for the target zone

```
void uart_handler(void) __attribute__ ((interrupt ("irq")));

void uart_handler(void){
    ...
```

```
}
```

Listing 4.4. USART3 interrupt handler

Exceptions and interrupts are globally enabled by default. Individual sources can be enabled and disabled at runtime using regular privileged instructions, which are silently trapped and emulated, or by replacing them in the source code with more performant MultiZone pseudo instructions. Individual interrupt sources can be set and clear using two well-defined MultiZone APIs: *STORE_NVICISER* and *STORE_NVICICER*. For example, Listings 4.5 and 4.6 show how UART (USART3) and User Button (EXTI15_10) interrupts are managed in zone 1 and zone 2.

```
STORE_NVICISER(UART_IRQn);
```

Listing 4.5. UART interrupt handler

```
static void button_init (void) {
    BTN_INIT(BTN_PIN);
    STORE_NVICISER(BTN_IRQn);
}
```

Listing 4.6. UART interrupt handler

System Timer

ARMv7-M processors provide one 24-bit clear-on-write decrementing counter system timer SysTick. MultiZone provides an enhanced 64-bit virtual instance of the SysTick timer for each zone.

The MultiZone API exposes four interfaces to read and set the virtual timer. The timer compare register is used to trigger single shot interrupts routed to the SysTick handler. There is no need to map the SysTick source to any zone as each zone has its own private virtual copy of the timer completely independent from the others.

For example, the code snippet below shows how the timer is used in zone2/main.c . The system time is stored in T0, incremented by a number of ticks equivalent to 500ms in T1, and then used to set the timer comparator. After 500ms an interrupt is triggered and the zone execution routed to the systick_handler (Listing 4.8).

```
int main (void) {

    // ...

    /* Trigger exception in 500ms */
    const uint64_t T0 = MZONE_RDTIME();
    const uint64_t T1 = T0 + (RTC_FREQ*(500/1000.0)); // 500 ms
    MZONE_WRTIMECMP(T1);

    while(1){
        // ...
```

```
    }

} // main()
```

Listing 4.7. Soft timer initial configuration

Listing 4.8 shows the implementation of the 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.

```
void systick_handler(void) __attribute__ ((interrupt ("irq")));
void systick_handler(void){

    /* Trigger exception in 500ms */
    const uint64_t T0 = MZONE_RDTIME();
    const uint64_t T1 = T0 + (RTC_FREQ*(500/1000.0)); // 500 ms
    MZONE_WRTIMECMP(T1);

    /* Processing */
    // ...

}
```

Listing 4.8. Soft timer interrupt handler

Note: the system timer can be set more efficiently and precisely with the `MZONE_ADTIMECMP()` that atomically reads the system time, increments it, and sets the comparator in one single system call.

Privileged Instructions

The Armv7-M ISA specifies a well-defined set of privileged instructions (e.g., msr, mrs) and a well-defined memory region, called System area (0xE000_0000 - 0xFFFF_FFFF), which is reserved for system-level operations. Within this predefined 512MiB range, there is a special 4KiB subregion named System Control Space (SCS) providing registers for system configuration and status reporting and control (e.g, System Control Block, NVIC, SysTick, MPU).

Access to the system memory is reserved to privileged code. The MultiZone runtime provides complete trap and emulation support to allow zones' programs running in unprivileged mode to read and write protected views of the system memory. Note that this is done transparently to the developer: no source code changes are required to run unmodified binaries - i.e. legacy applications.

Listing 4.9 shows an example of a read operation from a register belonging to the System area - CPUID. Note that the code is unchanged and that this is the normal memory read to the CPUID system register address.

```
void print_cpu_info(void) {
```

```
    /* SCB->CPUID Special register access (read-only) */
    uint32_t cpuid = *(volatile uint32_t *)SCB_CPUID;

    // ...
}
```

Listing 4.9. Unmodified read access to the *CPUID* register.

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 a completely optional alternative, the MultiZone API provides high-performance replacements for these instructions in the form of unprivileged pseudo instructions. These are C-style macro expansions that substitute privileged instructions with optimized inline assembly. Listing 4.10 shows an example of use of interrupt-related privileged pseudo instructions.

```
while(1){

    // Message handler
    char msg[16]; if (MZONE_RECV(1, msg)) {
        // ...
        else if (strcmp("irq=0", msg)==0) __disable_irq();
        else if (strcmp("irq=1", msg)==0) __enable_irq();
        // ...
    }

    // ...
}
```

Listing 4.10. Privileged pseudoinstructions: enable and disable 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	Pseudo instruction	Assembly Implementation
Global Interrupt enable	__enable_irq()	__asm volatile ("cpsie i" : : : "memory")
Global Interrupt disable	__disable_irq()	__asm volatile ("cpsid i" : : : "memory")
Global Fault exception and Interrupt enable	__enable_fault_irq()	__asm volatile ("cpsie f" : : : "memory")
Global Fault exception and Interrupt disable	__disable_fault_irq()	__asm volatile ("cpsid f" : : : "memory")
Assign value to Priority Mask Register	__set_PRIMASK(int primask)	__asm volatile ("msr primask, %0" : : "r" (primask) : "memory")
Return Priority Mask Register	int __get_PRIMASK(void)	__asm volatile ("mrs %0, primask" : "=r" (primask) :: "memory")
Assign value to Fault Mask Register	__set_FAULTMASK(int faultmask)	__asm volatile ("msr faultmask, %0" : : "r" (faultmask) : "memory")
Return Fault Mask Register	int __get_FAULTMASK(void)	__asm volatile ("mrs %0, faultmask" :

		"=r" (faultmask))
Set Base Priority	__set_BASEPRI(int basepri)	__asm volatile ("msr basepri, %0" : : "r" (basepri) : "memory")
Return Base Priority	int __get_BASEPRI(void)	__asm volatile ("mrs %0, basepri" : "=r" (basepri))
Set CONTROL register value	__set_CONTROL(int control)	__asm volatile ("msr control, %0" : : "r" (control) : "memory")
Return CONTROL Register Value	int __get_CONTROL(void)	__asm volatile ("mrs %0, control" : "=r" (control))
Set Process Stack Pointer value	__set_PSP(int psp)	__asm volatile ("msr psp, %0" : : "r" (psp) :)
Return Process Stack Pointer	int __get_PSP(void)	__asm volatile ("mrs %0, psp" : "=r" (psp))
Set Main Stack Pointer	__set_MSP(int msp)	__asm volatile ("msr msp, %0" : : "r" (msp) :)
Return Main Stack Pointer	int __get_MSP(void)	__asm volatile ("mrs %0, msp" : "=r" (msp))
Data Synchronization Memory Barrier	__DSB()	__asm volatile ("dsb 0xF")
Instruction Synchronization Barrier	__ISB()	__asm volatile ("isb 0xF")
Wait for Interrupt	__WFI()	__asm volatile ("wfi")

Table 4.2. MultiZone Pseudo Instructions

Appendix - MultiZone API

This section covers the MultiZone Security API. Consistently with MultiZone zero trust design philosophy, this API is not implemented in the form of a traditional static or dynamic library. Instead, only the 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.

```
/* MultiZone Security API */

#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(cycles);
#define MZONE_ADTIMECMP(cycles);

#define LOAD_SCB(scb_reg);
#define STORE_NVICISER(irq);
#define STORE_NVICICER(irq);
#define LOAD_MPURBAR(val);
#define LOAD_MPURASR(val);
```

Listing 5.1. MultiZone AP header fileI

The API is logically organized in four parts: 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 and cause the scheduler to switch execution to the next zone.
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 16-byte fixed length 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.
int MZONE_RECV(zone num, *char)	Check the inbox for a new 16-byte fixed length message

	from zone num. If a new message is present, it is copied to the local memory pointed by *char and the inbox is made available for a new message. Otherwise returns 0.
CPU Timer	
uint64_t MZONE_RDTIME()	Read the 64-bit time value since reset.
uint64_t MZONE_RDTIMECMP()	Reads the 64-bit timer comparator register.
void MZONE_WRTIMECMP(uint64_t cycles)	Writes the 64-bit timer comparator register.
void MZONE_ADTIMECMP(uint64_t cycles)	Read the time value, increments by cycles, and write to the timer comparator register.
Privileged Access Helpers	
uint32_t LOAD_SCB(uint32_t scb_reg)	Secure read-only operation of specific privileged system registers: e.g., CPUID (SCS_SCB_CPUID) and VTOR (SCS_SCB_VTOR).
STORE_NVICISER(uint32_t irq_num)	Enable interrupt source irq_num. Fail silently if irq_num is not mapped to this zone.
STORE_NVICICER(uint32_t irq_num)	Disable interrupt source irq_num. Fail silently if irq_num is not mapped to this zone.
uint32_t LOAD_MPURBAR(uint32_t mpu_reg)	Read MPU base address configuration for the mpu_reg region. Returns the value of the base address for that specific region. Returns 0 if region is out of bound.
uint32_t LOAD_MPURASR(uint32_t mpu_reg)	Read MPU attribute and size configuration for the mpu_reg region. Returns 0 if region is out of bound.

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.

MZONE_YIELD() is used in a cooperative system to increase system responsiveness. This call is optional as the preemptive scheduler forces a yield() upon tick expiration anyway.

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: *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. *MZONE_WFI()* can potentially lead to a few additional interrupt latency cycles and it is the default option for battery operated devices.

Secure Messaging

MultiZone runtime provides a self-contained facility for inter-zone secure communications. It allows zones to exchange secure messages – 16-byte fixed length protected 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 is resumed if it was paused waiting for interrupt. There is no guarantee of message delivery. It is the responsibility of the sender to retransmit or error. There is one inbox for each other zone. Messages can be sent to and received from the zone itself.

Note: the sender zone is intrinsically bound to the message and cannot be spoofed.

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

Timer Management

The Armv7-M architecture defines one 24-bit System Tick Timer (SysTick). MultiZone provides one enhanced 64-bit virtual instance for each zone and a set of relative APIs.

The virtual timer instance is a 64-bit free-running incremental counting unit operating at the SysTick frequency. It starts to count since the system reset, and allows zones to keep track of the real passage of the time using the ***MZONE_RDTIME*** interface.

Each virtual timer exposes a shadow timer compare register which can be used to trigger an interrupt (to the zone) when the actual time is greater than or equal to the value specified in the timer compare register. The timer compare register can be read and set using the ***MZONE_RDTIMECMP*** and ***MZONE_WRTIMECMP*** APIs, respectively.

Listing 5.2 shows how to combine the *MZONE_RDTIME* and *MZONE_WRTIMECMP* APIs to set the timer expiration in 500ms. In this case, T0 gets the actual time (represented by the number of cycles passed since system reset) and T1 is set by incrementing T0 with the number of cycles equivalent to 500 milliseconds - this depends on the CPU frequency. T1 is then used to set the comparator register.

```
int main (void) {

    /* Set soft-timer */
    const uint64_t T0 = MZONE_RDTIME();
    const uint64_t T1 = T0 + (RTC_FREQ*(500/1000.0)); // 500 ms
    MZONE_WRTIMECMP(T1);
```

```
    // ...

} // main()
```

Listing 5.2. MultiZone timer management APIs use: soft timer configuration.

In addition, MultiZone provides the ***MZONE_ADTIMECMP*** call to execute the above operations atomically in one single system call. ***MZONE_ADTIMECMP*** reads the time value, adds the amount of cycles, and sets the timer compare register. This is the preferred way to set the timer as it is more precise and more performant.

```
int main (void) {

    /* Set soft-timer */
    const uint64_t T1 = RTC_FREQ*(500/1000.0); // 500 ms
    MZONE_ADTIMECMP(T1);

    // ...

} // main()
```

Listing 5.3. MultiZone timer management APIs use: optimized soft timer configuration.

Privileged Access

The Armv7-M ISA defines a well-defined set of privileged instructions. These instructions are intended to be executed at the higher level of privilege (i.e., Privileged Thread or Privileged Handler modes) and typically used to access special registers to control among others interrupt settings and power-down operations.

MultiZone runs zones' code in the Unprivileged Privileged Thread mode. The MultiZone toolchain extension and the MultiZone runtime provide binary translation and trap and emulation to run unmodified binaries in protected mode. Trap and emulation techniques introduce a small performance overhead. As an alternative, MultiZone provides a set of optional high-performance wrappers that minimize this overhead by replacing original instruction with optimized inline assembly macros.

Listing 5.4 shows an example of optimized access to privileged registers. In this case the ***LOAD_SCB*** provides a high-performance wrapper to access memory-mapped system registers in the System Control Block (SCB) area (e.g., CPUID, VTOR).

```
void print_cpu_info(void) {

    /* SCB->CPUID Special register access (read-only) */
    uint32_t cpuid = LOAD_SCB(SCS_SCB_CPUID);

    // ...
```

Listing 5.4. MultiZone high-performance privileged access APIs use: SCB registers

Another important set of memory-mapped privileged registers are the ones related to the Nested Vectored Interrupt Controller (NVIC). MultiZone provides an interface to securely enable and disable interrupts associated with zones.

Listing 5.5 shows how to use the ***STORE_NVICISER*** interface to set the NVIC interrupt enable register corresponding to the UART interrupt source. In this case, the interrupt is enabled after the UART initialization. If the interrupt source is not mapped to the zone via policy file the call fails silently. Similarly, the ***STORE_NVICICER*** provides the interface to disable the interrupt source.

```
int main (void) {

    open("UART", 0, 0);
    STORE_NVICISER(UART_IRQn);

    // ...

    while(1){
        // ...
    }

} // main()
```

Listing 5.5. MultiZone high-performance privileged access APIs use: interrupt controller

The MPU is another privileged hardware block accessible through a memory-mapped privileged register area. Because each zone has its own MPU configuration, MultiZone provides each zone with its own set of shadow MPU registers. The zone configuration (address + policies) is exposed through two well-defined APIs.

Note: access to the MPU is read only and limited to the configuration of the current zone. It is provided only as an example as there should be really no reasons for the application code to read its memory protection configuration, which is statically defined for security reasons and can't be changed at runtime.

Listing 5.6 shows how to use ***LOAD_MPURBAR*** and ***LOAD_MPURASR*** to read regions base address, attributes and size configuration for a given region. Out of bound reads fail silently and return zero.

```
void print_mpu(void){

    // ...

    uint32_t mpu_config[MPU_REGIONS*2];

    for(int i=0, j=0; i<mpu_regions; i++, j=j+2)
    {
        mpu_config[j] = LOAD_MPURBAR(i);
        mpu_config[j+1] = LOAD_MPURASR(i);
    }

    // ...
}
```

Listing 5.6. MultiZone high-performance privileged access APIs use: MPU regions configuration

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 comments and definitions for tick, zones, interrupts, and memory regions.

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

# MultiZone reserved memory: 8K @0x08000000, 6K @0x20000000

Tick = 10 # ms

Zone = 1
       irq = 55
       base = 0x08008000; size =   32K; rwx = rx # FLASH
       base = 0x20002000; size =    4K; rwx = rw # RAM
       base = 0x40023800; size =  0x80; rwx = rw # RCC
       base = 0x40020C00; size =  0x40; rwx = rw # GPIOD
       base = 0x40004800; size =  0x40; rwx = rw # USART3

Zone = 2
       irq = 56
       base = 0x08010000; size =   32K; rwx = rx # FLASH
       base = 0x20003000; size =    4K; rwx = rw # RAM
       base = 0x40023800; size =  0x80; rwx = rw # RCC
       base = 0x40020400; size =  0x40; rwx = rw # GPIOB
       base = 0x40020800; size =  0x40; rwx = rw # GPIOC
       base = 0x40013800; size =  0x20; rwx = rw # SYSCFG
       base = 0x40013C00; size =  0x20; rwx = rw # EXTI

Zone = 3
       base = 0x08018000; size =   32K; rwx = rx # FLASH
       base = 0x20004000; size =    4K; rwx = rw # RAM
       base = 0x40023800; size =  0x80; rwx = rw # RCC
       base = 0x40021000; size =  0x40; rwx = rw # GPIOE
       base = 0x40021400; size =  0x40; rwx = rw # GPIOF
```

Listing 6.1. MultiZone policy definition file

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 assigned to zones and will raise a configuration error if used:

```
Error  : zone 1 range 6 - kernel reserved [0x08000000 - 0x08002000]
```

```
Error  : zone 1 range 6 - kernel reserved [0x20000000 - 0x20001800]
```

Tick

The *tick* parameter drives the preemptive scheduler. It specifies the maximum amount of time in milliseconds that each zone 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. Valid range for the tick value is 0 to 1000. Values outside this range trigger a configuration error:

```
Error  : bsp/STM32F767/multizone.cfg (5) - Invalid tick value 10000, range 0 to 1000.
```

Zones

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

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 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, and read / write / execute flags.

Constraints:

- Zones can be mapped to a maximum of 8 memory regions each. Depending on size and alignment, user defined regions may result in multiple MPU regions - up to 3 each - thus reducing the maximum number available for the given zone. Naturally aligned power of two regions are the most MPU efficient and “consume” only one MPU region each.

- Valid region size range is 32 bytes to 4 GB. Size can be indicated in decimal and hexadecimal notation. The modifiers K, M, G mean the size is given in kilobytes, megabytes, or gigabytes respectively.

- Region access can be any combination of RWX. The first region is for the program text segment and should have RX access.

- Memory regions can overlap across zones. This might be necessary in some particular situation although in general it is considered not secure and will trigger a warning.

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

Interrupts

Interrupts are unique asynchronous events, typically associated with peripherals, indicating a need for immediate attention. MultiZone implements the concept of user-mode interrupts, which enable the secure (unprivileged) execution of hardware interrupt handlers. Interrupts sources must be explicitly assigned to zones.

- Interrupt sources are uniquely assigned to zones. It is not possible to assign the same interrupt source to multiple zones. Each zone has a private virtual timer instance assigned automatically by the system, so there is no need to indicate the timer interrupt source in the configuration.

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

- The maximum number of interrupt sources is 112 corresponding to the range [16-127].

Appendix - MultiZone Toolchain Extension

The MultiZone toolchain extension multizone.jar, aka “configurator”, merges boot code, zones binaries, security policies, and the MultiZone pre-built runtime to produce a signed firmware image for the secure boot. The Multizone SDK invokes this utility in the final step of the Make script. If a Make build system is not 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 --boot parameter is optional. It allows to specify an arbitrary binary file (.HEX or .ELF) 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, etc. IMPORTANT: this optional boot 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

```
$ java -jar multizone.jar -?
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
Report bugs to <bug@hex-five.com>.
```

Listing 7.1. MultiZone toolchain extension syntax