Imperas ARM TrustZone Platform Modeling Application Note

Imperas Software Limited
Imperas Buildings, North Weston,
Thame, Oxfordshire, OX9 2HA, UK
docs@imperas.com

Author:	Imperas Software Limited
Version:	1.1
Filename:	Imperas_ARM_TrustZone_Platform_Modeling_Application_Note.doc
Project:	ARM TrustZone Platform Modeling Application Note
Last Saved:	September 11, 2018
Keywords:	OVP Cortex TrustZone

© 2018 Imperas Software Limited

Copyright Notice

Copyright © 2018 Imperas Software Limited All rights reserved. This software and documentation contain information that is the property of Imperas Software Limited. The software and documentation are furnished under a license agreement and may be used or copied only in accordance with the terms of the license agreement. No part of the software and documentation may be reproduced, transmitted, or translated, in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without prior written permission of Imperas Software Limited, or as expressly provided by the license agreement.

Right to Copy Documentation

The license agreement with Imperas permits licensee to make copies of the documentation for its internal use only. Each copy shall include all copyrights, trademarks, service marks, and proprietary rights notices, if any.

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

IMPERAS SOFTWARE LIMITED, AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Table of Contents

1 Preface

This document describes suggested techniques for modeling ARM TrustZone peripherals in an OVP platform.

1.1 Related Documents

The reader should be familiar with Open Virtual Platforms (OVP) platform and peripheral modeling as described in the following OVP documentation:

- *Writing Platforms and Modules in C User Guide*
- *OVP Peripheral Modeling Guide*
- *OVP BHM PPM Function Reference*

In addition this document shows examples using the Imperas **iGen** tool described in:

- *iGen Platform and Module Creation User Guide*
- *iGen Model Generator Introduction*
- *iGen Peripheral Generator User Guide*

The following ARM documents describe the TrustZone architecture and relevant peripherals:

- *ARM Security Technology (PRD29-GENC-009492C)*
- *ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition (DDI0406C)*
- *ARM TrustZone Protection Controller (BP147) Technical Overview (DTO0015)*

1.2 Trademarks

ARM®, TrustZone®, AMBA®, Cortex™, MPCore™ and any other trademark found on the ARM trademarks list that are referred to or displayed in this document are trademarks or registered trademarks of ARM Ltd or its subsidiaries.

Imperas acknowledge trademarks or registered trademarks of other organizations for their respective products and services.

2 OVP and ARM TrustZone

The ARM TrustZone technology adds a secure mode to the processor architecture and additional security signals to the AMBA bus, which are used by TrustZone-aware peripherals to prevent access to certain bus addresses when the processor (or other bus master) is not in secure mode.

As ARM TrustZone features are used for embedded systems, it becomes important to be able to comprehensively test the software that makes use of the TrustZone features. Key to comprehensive testing is the performance, controllability and visibility that virtual platforms provide.

This document focuses on best known methods for modeling TrustZone-aware peripherals to achieve optimal performance.

2.1 OVP ARM Processor Model

The OVP ARM Processor model implements all the programmer-visible TrustZone features in the ARM architecture, including secure, non-secure and monitor modes, banked registers, secure and non-secure TLBs, etc.

To model an entire TrustZone-aware system using OVP requires additional modeling at the platform level to configure the bus and peripherals to implement the secure/non-secure access restrictions.

This document provides an overview of techniques that may be used to efficiently model TrustZone-aware peripherals in OVP platforms.

2.2 iGen - platform and model builder

The examples also require the use of the Imperas **iGen** platform building tool that converts tcl descriptions into C/C++/SystemC/TLM2 platforms.

2.3 Toolchains

To use the Makefiles provided with the examples you will need to install the armv7 toolchain package available from the download page of the OVPworld website at:

http://www.ovpworld.org/dlp/#arm.toolchains

3 Secure and Non-secure Accesses

From a platform standpoint, all of the details of the TrustZone processor technology boil down to the processor supplying a single bit of information indicating for each bus access whether it is a secure or non-secure access. In AMBA3 bus terms this is the `AxPROT[1]` signal, where `x` is `W` for write accesses and `R` for read accesses.

Secure accesses (`AxPROT[1]` = 0) can access any address, while non-secure accesses (`AxPROT[1]`=1) can only access addresses configured as non-secure (or addresses where the `AxPROT[1]` bit is ignored).

When modeling a platform implementing TrustZone security, this security information must be taken into account when establishing whether an access will succeed.

3.1 Modeling of TrustZone in OVP Platforms

If the security information were simply implemented by adding `AxPROT[1]` as a net output from the processor, each peripheral could monitor this signal and decide for each access whether it is allowed. However this would result in a severe decrease in simulation performance.

The OVP ARM processor model instead takes advantage of the sophisticated address aliasing capabilities built into OVP simulators to support high-speed modeling of virtual memory. The `AxPROT[1]` signal is implemented as an additional address bit for the processor's `INSTRUCTION` and `DATA` ports. This bit is the MSB of the bus address so there are effectively two address spaces, one secure (when the security address bit is 0) and one non-secure (when the security address bit is 1).

At the platform level, this allows control of access to the secure and non-secure address spaces using OVP bus bridge elements. Bus bridges allow connecting an address range on one bus to an address range on another bus; this may have a common or different base address for the bridging (memory address mapping). Once created, these address mappings involve no overhead per access and thus allow very high-performance simulation.

To model a TrustZone system, OVP bus bridges are used to create a mapping between the non-secure address space and the secure address space only for those address regions where non-secure accesses are to be permitted. If no bridge is created in the platform then no non-secure accesses will succeed.

3.2 The TrustZone NS Bit

In OVP ARM Processor models with TrustZone implemented, bit 40 of an address is the NS bit, corresponding to the `AxPPROT[1]` signal, indicating whether an access is secure or non-secure. A value of 0 on the NS bit indicates a secure access, while a 1 indicates a non-secure access.

Note that bit 40 is used because bits 31:0 are the normal ARM 32 bit address bits while bits 39:32 are used when a processor supports Large Physical Address Extension (LPAE). In order to simplify the interface, bit 40 is always used regardless of whether LPAE is present so that platforms need not be modified depending on whether the selected processor variant supports LPAE.

Thus the OVP ARM processor model supports connection to busses with up to 41 bits (bits 40:0) of address. OVP uses sparse memory models which allocate resources only for memory actually used so this does not require significant additional resources to implement.

4 Examples

An OVP platform can support TrustZone security in several different ways:

- Ignore the TrustZone NS bit completely
- Only allow secure accesses to an address region
- Always allow non-secure accesses to an address region
- Configurable during run time to allow/disallow non-secure accesses to an address region

The following examples show how to implement each of these scenarios in an OVP platform, and run a simple application that accesses a peripheral in both secure and non-secure mode, with the access failing when non-secure accesses to the peripheral are not enabled.

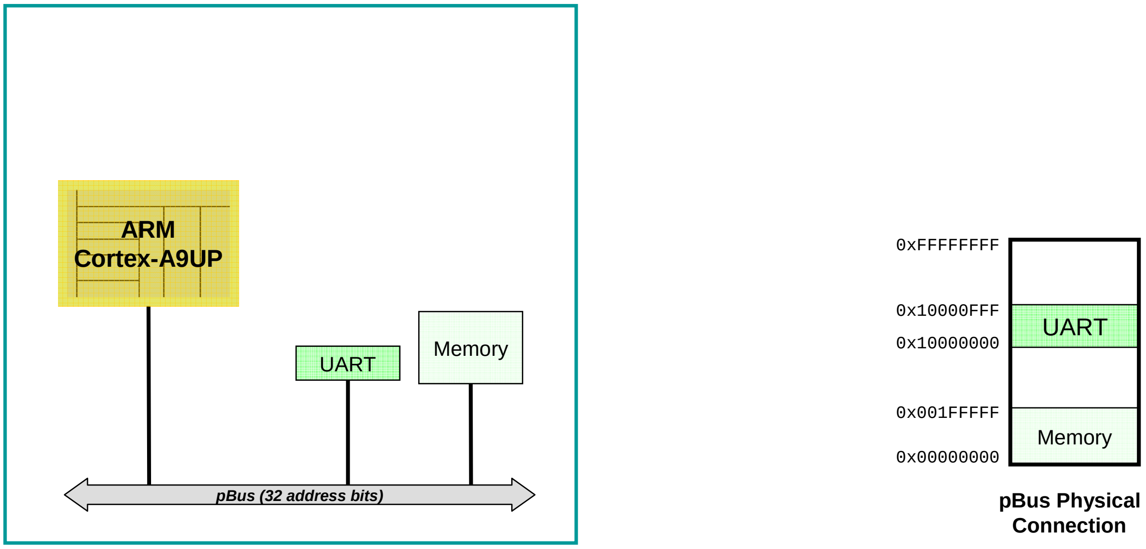
The examples may be found in the following directory in an Imperas or OVP installation:

```
$IMPERAS_HOME/Examples/PlatformConstruction/ARM_specific/trustzone
```

4.1 Ignoring the TrustZone NS Bit in a Platform

If a platform does not have any TrustZone-aware memory or peripherals, the TrustZone bit is easily ignored by connecting a bus of less than 41 address bits to the processor. In this case the simulator ignores address bits outside the range of the connected bus, and no distinction is made between secure and non-secure accesses.

The following example shows a platform that ignores the TrustZone bit and thus makes no distinctions between secure and non-secure accesses to memory or peripherals.



Virtual Platform Block Diagram

Virtual Platform Address Mapping

Figure 1: Platform 'ignoreTrustZone' Block Diagram and Memory Mapping

4.1.1 *ignoreTrustZone* Example Platform

This platform does not implement any TrustZone features. No distinction is made by the memory or peripherals between secure and non-secure reads and writes. This example may be found in:

```
  /ARM_specific/trustzone/1.ignoreTrustZone
```

We will define a simple platform with a Cortex-A9UP processor, memory and an ARM PL011 UART.

The platform is defined as a tcl file, which is processed by the Imperas **iGen** program to generate an OVP platform in C:

```
#
# Bus - 32 bits of address (Secure bit ignored)
#
ihwaddbus -instancename pBus -addresswidth 32

#
# Processor
#
ihwaddprocessor -instancename cpu -type arm -endian little -semihostname armNewlib
ihwconnect      -instancename cpu -busmasterport INSTRUCTION -bus pBus
ihwconnect      -instancename cpu -busmasterport DATA        -bus pBus
ihwsetparameter -handle       cpu -name variant       -value Cortex-A9UP –type enum
ihwsetparameter -handle       cpu -name compatibility -value nopSVC      –type enum

#
# Memory: 0x00000000    0x001fffff  RAM
#
ihwaddmemory -instancename ram0 -type ram
ihwconnect   -instancename ram0 -busslaveport sp1 -bus pBus \
             -loaddress 0x00000000 -hiaddress 0x001fffff

#
# UART:   0x10000000    0x10000fff  UartPL011
#
ihwaddperipheral -instancename uart0   -type UartPL011
ihwconnect       -instancename uart0   -busslaveport  bport1 -bus pBus \
                 -loaddress 0x10000000 -hiaddress 0x10000fff
ihwsetparameter    -handle       uart0   -name outfile -value uart0.log –type string

...
```

A bus is defined with an address width of 32 bits. A processor is also defined and the bus connected to its `INSTRUCTION` and `DATA` ports. Since the bus is defined to have only 32 bits of address all the other address bits from the processor (including bit 40, the NS bit) are ignored in this platform:

```
#
# Bus - 32 bits of address (Secure bit ignored)
#
ihwaddbus -instancename pBus -addresswidth 32


#
# Processor
#
ihwaddprocessor -instancename cpu -type arm -endian little -semihostname armNewlib
```

```
ihwconnect      -instancename cpu -busmasterport INSTRUCTION -bus pBus
ihwconnect      -instancename cpu -busmasterport DATA        -bus pBus
ihwsetparameter -handle       cpu -name variant       -value Cortex-A9UP –type enum
ihwsetparameter -handle       cpu -name compatibility -value nopSVC      –type enum
```

Note that exceptions are NOT enabled on the processor, so any exception will cause the simulation to terminate, which is sufficient for our purposes.

Note also the `armNewlib` semihost library is specified on the processor definition, allowing newlib functions such as `printf()` to be used by applications running on the processor.

Next, 2MB of memory starting at address 0 is defined:

```
#
# Memory: 0x00000000    0x001fffff  RAM
#
ihwaddmemory -instancename ram0 -type ram
ihwconnect   -instancename ram0 -busslaveport sp1 -bus pBus \
             -loaddress 0x00000000 -hiaddress 0x001fffff
```

And finally a UART is defined at address 0x10000000:

```
#
# UART:   0x10000000    0x10000fff  UartPL011
#
ihwaddperipheral -instancename uart0   -type UartPL011
ihwconnect       -instancename uart0   -busslaveport  bport1 -bus pBus \
                 -loaddress 0x10000000 -hiaddress 0x10000fff
ihwsetparameter  -handle       uart0   -name outfile -value uart0.log –type string
```

4.1.2 Example application

Next, let’s define a simple application which will run on the platform and write to the UART from both secure and non-secure modes:

```
#include <stdio.h>

// UART Registers
#define UART_BASE 0x10000000
#define UART_TX   ((volatile unsigned char *) (UART_BASE + 0x0 ))
#define UART_FR   ((volatile unsigned char *) (UART_BASE + 0x18))

// virtual memory translation table
unsigned int translationTable[4096] __attribute__ ((aligned (16384)));

// CP15 register access
#define WR_CP15(_OP1, _CRN, _CRM, _OP2, _VAL) \
    asm volatile("mcr p15, "#_OP1", %0, "#_CRN", "#_CRM", "#_OP2::"r"(_VAL))

#define RD_CP15(_OP1, _CRN, _CRM, _OP2, _VAL) \
    asm volatile("mrc p15, "#_OP1", %0, "#_CRN", "#_CRM", "#_OP2:"=r"(_VAL))

#define WR_SCTLR(_VAL) WR_CP15(0, c1, c0, 0, _VAL)
#define RD_SCTLR(_VAL) RD_CP15(0, c1, c0, 0, _VAL)
#define WR_SCR(_VAL)   WR_CP15(0, c1, c1, 0, _VAL)
#define RD_SCR(_VAL)   RD_CP15(0, c1, c1, 0, _VAL)
```

```
#define WR_TTBR0(_VAL) WR_CP15(0, c2, c0, 0, _VAL)
#define WR_DACR(_VAL)  WR_CP15(0, c3, c0, 0, _VAL)

// Add a section entry to the translation table mapping 1MB
// starting at VA to physical address PA
void addSection(unsigned int VA, unsigned int PA) {

    unsigned int index  = (VA >> 20);
    unsigned int entry  = (PA & 0xfff00000)             |  // Section base address
                          0x3 << 10                     |  // AP = read/write
                          0x2;                             // Section entry, PXN=0

    // Set the entry in the table to be a section
    translationTable[index] = entry;

}

// Setup translation table entries for first 2MB of memory and UART addresses
void setupTranslationTable() {
    addSection(0x00000000, 0x00000000);
    addSection(0x00100000, 0x00100000);
    addSection(UART_BASE, UART_BASE);
}

// Enable the TLB
void enableTLB() {

    // Write the translation table base register
    WR_TTBR0((unsigned int)translationTable);

    // Set DACR to 1 to enable client permissions for domain 0
    WR_DACR(1);

    // Set SCTLR.M (bit 0) to enable TLB
    unsigned int sctlr;
    RD_SCTLR(sctlr);
    WR_SCTLR(sctlr | 1);

}

// Enter non-secure mode by setting SCR.NS (bit 0)
void enterNonSecure() {
    unsigned int scr;
    RD_SCR(scr);
    WR_SCR(scr | 1);
}

// Write a string to the UART TX register
void putString(const char *s) {
    while (*s) {
        while ((*UART_FR & 0x80) == 0) {
            // Wait for Tx Holding Register Empty flag
        }
        *UART_TX = *s++;
    }
}

int main() {

    // Setup and enable the secure mode TLB
    // Note: Processor starts out in secure mode
    setupTranslationTable();
    enableTLB();
    printf ("\n*** Writing to uart from Secure Mode\n");
    putString("Hello from secure mode!\n");

    printf ("\n*** Entering Non-Secure Mode\n");
    enterNonSecure();

    // Enable the non-secure TLB (uses same translation table as secure mode)
    enableTLB();

    printf ("\n*** Writing to uart from Non-Secure Mode\n");
    putString("Hello from non-secure mode!\n");

}
```

The function `main()` first initializes a translation table and enables the TLB (note that we are in secure mode here because the processor always starts out in secure mode, so we are in fact enabling the secure mode TLB here):

```
int main() {

    // Setup and enable the secure mode TLB
    // Note: Processor starts out in secure mode
    setupTranslationTable();
    enableTLB();
...
```

The translation table is configured to describe virtual to physical mappings for the first 2 MB of memory and for 1MB of memory where the UART device is located. Note that identity mappings (where the VA equals the PA) are used:

```
// Setup translation table entries for first 2MB of memory and UART addresses
void setupTranslationTable() {
    addSection(0x00000000, 0x00000000);
    addSection(0x00100000, 0x00100000);
    addSection(UART_BASE, UART_BASE);
}
```

Once the TLB is setup, `printf()` is used to write a message to the simulator console and `putString()` is used to put a string out to the UART, one byte at a time.

```
    printf ("\n*** Writing to uart from Secure Mode\n");
    putString("Hello from secure mode!\n");
```

Next we put the processor into non-secure mode and enable the TLB (this time we are enabling the non-secure TLB):

```
    printf ("\n*** Entering Non-Secure Mode\n");
    enterNonSecure();

    // Enable the non-secure TLB (uses same translation table as secure mode)
    enableTLB();
```

Now we write to the UART again, this time from non-secure mode:

```
    printf ("\n*** Writing to uart from Non-Secure Mode\n");
    putString("Hello from non-secure mode!\n");
```

If the UART is inaccessible from non-secure mode then an exception will be signaled by the processor, which will terminate the simulation, since simulation of exceptions on the processor was not enabled in the platform. This platform has no TrustZone awareness so both secure and non-secure accesses to the UART will succeed.

> ⇒ You may notice that this example is quite contrived - we have not entered non-secure mode using an exception from monitor mode, as is the normal method for entering non-secure mode. Also we have no way of returning from non-secure mode back to secure mode since we haven’t created a handler for the SMC call. We also have not added barrier instructions following CP15 register updates that would be needed when running on real hardware.
>
> By ignoring all these normal requirements we are able to keep the example application quite simple, but still show the effects of secure vs. non-secure accesses.

4.1.3 Building and running the example

The application and platform are in the directory:

`Examples/PlatformConstruction/ARM_specific/trustzone/1.ignoreTrustZone`

Take a copy of that directory and `cd` to it:

```
cp -r
$IMPERAS_HOME/Examples/PlatformConstruction/ARM_specific/trustzone/1.ignoreTrustZone .
cd 1.ignoreTrustZone
```

To compile, build and run the platform, use the following command (see section 2.3 for info on setting up the toolchain needed by the `Makefile`):

```
make run
```

The application will run on the platform and you will see the following output:

```

*** Writing to uart from Secure Mode

*** Entering Non-Secure Mode

*** Writing to uart from Non-Secure Mode
```

And a file `uart0.log` will be created with the following contents:

```
Hello from secure mode!
Hello from non-secure mode!
```

indicating that the processor was able to access the UART from both secure and non-secure modes.

4.2 Always Secure Peripherals

Now we will modify the platform, changing the bus to use all 41 address bits coming from the processor. Now there will be effectively 2 address spaces - a secure space when address bit 40 is 0 and a non-secure space when address bit 40 is 1 (which occurs when the processor is in non-secure mode.) . This example may be found in:

```
$IMPERAS_HOME/Examples/PlatformConstruction/ARM_specific/trustzone/2.alwaysSecure
```

In this example we will show that by not creating a bridge from the non-secure space to the secure space for the UART it will be inaccessible from non-secure mode.

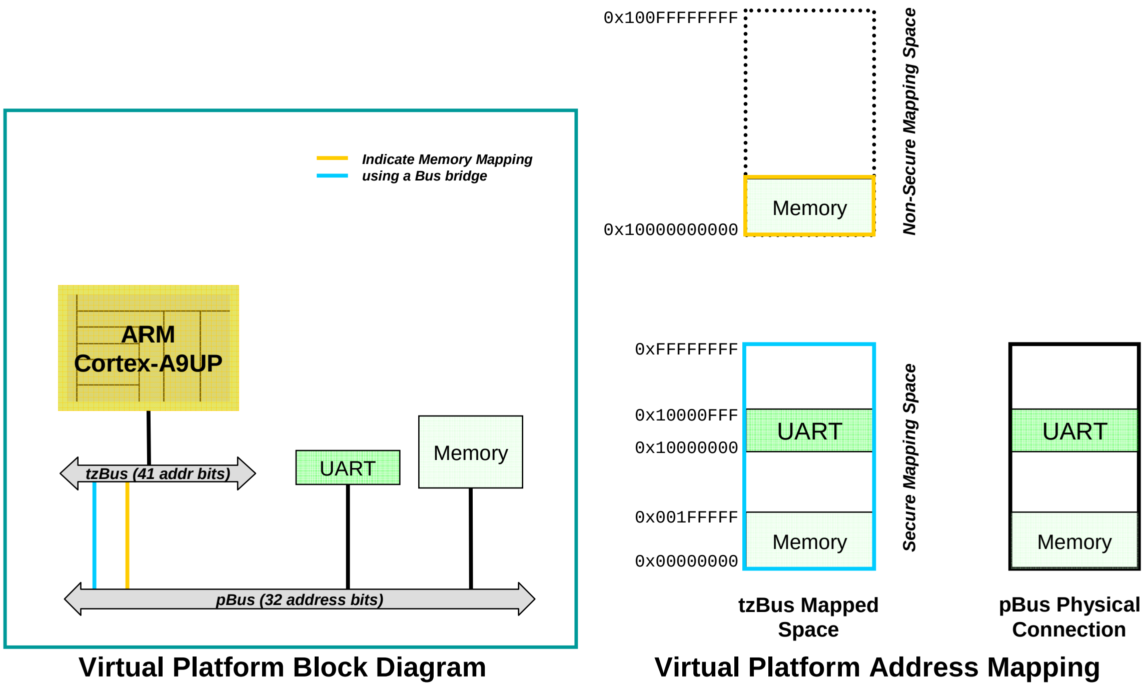


Figure 2: Platform 'alwaysSecure' Block Diagram and Memory Mapping

4.2.1 *alwaysSecure* Example Platform

We need to make two modifications to the previous platform:

1. Increase the address width defined for the bus to 41 bits
2. Add a bridge to map the ram to the secure space so the program will still run after we switch to non-secure mode

The changes from the previous platform are highlighted in **bold**:

```
...

#
# tzBus - 41 bits of address - implements secure and non-secure address spaces
#
ihwaddbus -instancename tzBus -addresswidth 41
```

```
#
# pBus  - 32 bits of address - physical bus connected to devices and memory
#
ihwaddbus -instancename pBus  -addresswidth 32

#
# Bridge - Secure space of tzBus always mapped to pBus
#
ihwaddbridge -instancename secure
ihwconnect   -instancename secure -busslaveport  sp -bus tzBus \
             -loaddress 0x00000000 -hiaddress 0xffffffff
ihwconnect   -instancename secure -busmasterport mp -bus pBus  \
             -loaddress 0x00000000 -hiaddress 0xffffffff

#
# Processor
#
ihwaddprocessor -instancename cpu -type arm -endian little -semihostname armNewlib
ihwconnect      -instancename cpu -busmasterport INSTRUCTION -bus tzBus
ihwconnect      -instancename cpu -busmasterport DATA        -bus tzBus
ihwsetparameter -handle       cpu -name variant       -value Cortex-A9UP –type enum
ihwsetparameter -handle       cpu -name compatibility -value nopSVC      –type enum

#
# Memory: 0x00000000    0x001fffff  RAM
#
ihwaddmemory -instancename ram0 -type ram
ihwconnect   -instancename ram0 -busslaveport sp1 -bus pBus \
             -loaddress 0x00000000 -hiaddress 0x001fffff

#
# Bridge - Memory always accessible from non-secure mode
#
ihwaddbridge -instancename ram0NS
ihwconnect   -instancename ram0NS -busslaveport  sp -bus tzBus \
             -loaddress 0x10000000000 -hiaddress 0x100001fffff
ihwconnect   -instancename ram0NS -busmasterport mp -bus pBus \
             -loaddress 0x00000000 -hiaddress 0x001fffff

#
# UART:   0x10000000    0x10000fff  UartPL011
#
ihwaddperipheral -instancename uart0   -type UartPL011
ihwconnect       -instancename uart0   -busslaveport  bport1 -bus pBus \
                 -loaddress 0x10000000 -hiaddress 0x10000fff
ihwsetparameter  -handle       uart0   -name outfile -value uart0.log   -type string

...
```

Now, in addition to the bus `pBus` with 32 address bits we have an additional bus, `tzBus` with 41 address bits:

```
#
# tzBus - 41 bits of address - implements secure and non-secure address spaces
#
ihwaddbus -instancename tzBus -addresswidth 41
```

A bridge is added to map the entire secure address space (0x000_0000_0000:0x000_ffff_ffff) of `tzBus` to `pBus`:

```
#
# Bridge - Secure space of tzBus always mapped to pBus
#
ihwaddbridge -instancename secure
ihwconnect   -instancename secure -busslaveport  sp -bus tzBus \
             -loaddress 0x00000000 -hiaddress 0xffffffff
ihwconnect   -instancename secure -busmasterport mp -bus pBus  \
             -loaddress 0x00000000 -hiaddress 0xffffffff
```

Thus any address may be accessed from secure mode.

A bridge is also added to make the memory accessible from non-secure mode - without this the program would take an exception immediately after the switch to secure mode:

```
#
# Bridge - Memory always accessible from non-secure mode
#
ihwaddbridge -instancename ram0NS
ihwconnect   -instancename ram0NS -busslaveport  sp -bus tzBus \
             -loaddress 0x10000000000 -hiaddress 0x100001fffff
ihwconnect   -instancename ram0NS -busmasterport mp -bus pBus \
             -loaddress 0x00000000 -hiaddress 0x001fffff
```

4.2.2 Example application

We will use the same application as in the previous example

4.2.3 Building and running the example

The application and platform are in the directory:
      `Examples/PlatformConstruction/ARM_specific/trustzone/2.alwaysSecure`
Take a copy of that directory and `cd` to it:

```
cp -r $IMPERAS_HOME/Examples/PlatformConstruction/ARM_specific/trustzone/2.alwaysSecure
.
cd 2.alwaysSecure
```

To compile, build and run the platform, use the following command (see section 2.3 for info on setting up the toolchain needed by the `Makefile`):

```
make run
```

The application will run on the platform and you will see the following output:

```

*** Writing to uart from Secure Mode

*** Entering Non-Secure Mode

*** Writing to uart from Non-Secure Mode
Processor Exception (PC_PRX) Processor 'alwaysSecure/cpu' 0x823c: e5d33000 ldrb    r3,[r3]
Processor Exception (PC_RPX) No read access at 0x10000018
```

When we attempt to read from the UART from non-secure mode there is an exception on the **ldrb** instruction because the UART's flag register address of 0x1000_0018 is not accessible from non-secure mode.

The file `uart0.log` will be created with the following contents:

```
Hello from secure mode!
```

indicating that the processor was able to access the UART from secure mode, but not from non-secure mode.

4.3 Always Non-secure Peripherals

Next we will modify the platform to make the UART always available in non-secure mode by adding a bus bridge from the non-secure address space for the UART's address region.

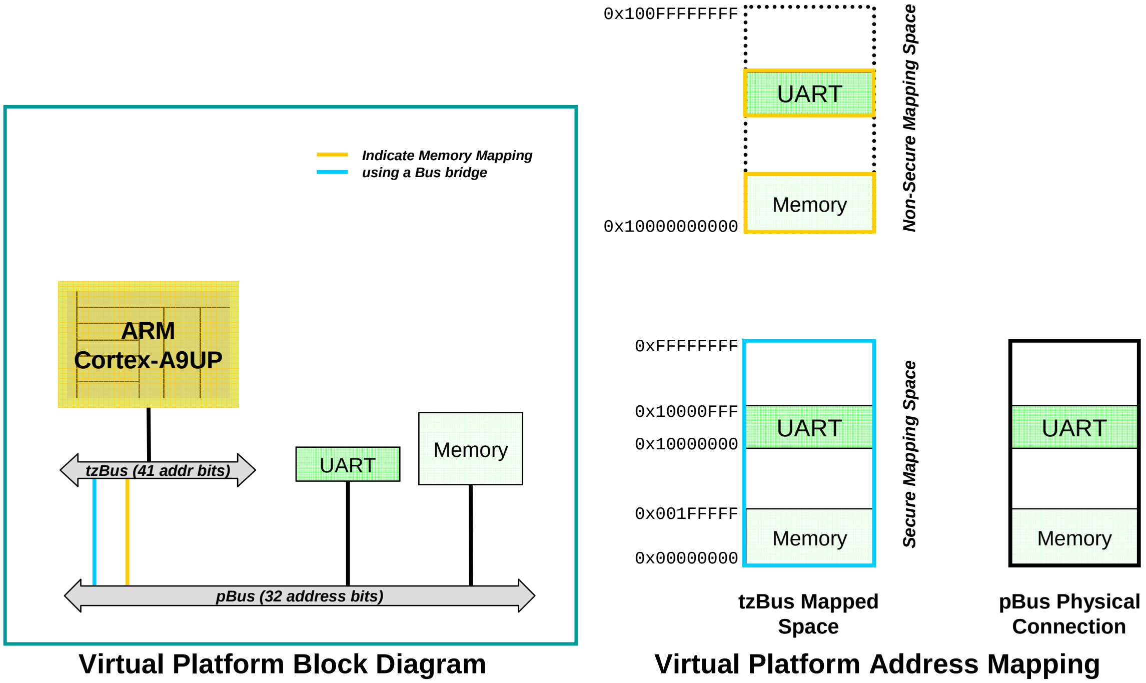


Figure 3: Platform 'alwaysNonSecure' Block Diagram and Memory Mapping

4.3.1 *alwaysNonSecure* Example Platform

We will modify the previous platform and add a non-secure bridge for the UART region. This example may be found in:
`$IMPERAS_HOME/Examples/PlatformConstruction/ARM_specific/trustzone/3.alwaysNonSecure`

The changes from the previous platform are highlighted in **bold** (most of the unchanged code has been omitted for brevity):

```
...

ihwnew -name alwaysSecure

...

#
# UART:   0x10000000    0x10000fff  UartPL011
#
ihwaddperipheral -instancename uart0   -type UartPL011
ihwconnect       -instancename uart0   -busslaveport  bport1 -bus pBus \
                 -loaddress 0x10000000 -hiaddress 0x10000fff
ihwsetparameter  -handle       uart0   -name outfile -value uart0.log   -type string

#
# Bridge - UART always accessible from non-secure mode
#
ihwaddbridge     -instancename uart0NS
ihwconnect       -instancename uart0NS -busslaveport  sp -bus tzBus \
                 -loaddress 0x10010000000 -hiaddress 0x10010000fff
ihwconnect       -instancename uart0NS -busmasterport mp -bus pBus  \
                 -loaddress    0x10000000 -hiaddress    0x10000fff
...
```

The `uart0NS` bridge has been added to make the UART always accessible from non-secure mode.

4.3.2 Example application

We will use the same application as in the previous examples

4.3.3 Building and running the example

The application and platform are in the directory:
`Examples/PlatformConstruction/ARM_specific/trustzone/3.alwaysNonSecure.`
Take a copy of that directory and `cd` to it:

```
cp -r
$IMPERAS_HOME/Examples/PlatformConstruction/ARM_specific/trustzone/3.alwaysNonSecure .
cd 3.alwaysNonSecure
```

To compile, build and run the platform, use the following command (see section 2.3 for info on setting up the toolchain needed by the `Makefile`):

```
make run
```

The application will run on the platform and you will see the following output:

```
*** Writing to uart from Secure Mode

*** Entering Non-Secure Mode

*** Writing to uart from Non-Secure Mode
```

And a file `uart0.log` will be created with the following contents:

```
Hello from secure mode!
Hello from non-secure mode!
```

indicating that the processor was able to access the UART from both secure and non-secure modes.

4.4 Run-time switching between Secure and Non-secure

So far we have seen how to ignore the TrustZone secure bit entirely, and how to create peripherals that are always secure or always non-secure. However it is more common that a peripheral can have its security state modified while running, either at boot time or by a secure monitor program.

This example shows how to define a peripheral with dynamically configurable security. It uses an OVP model of the ARM BP147 TrustZone Protection Controller (TZPC) which implements up to 24 output signals that are used to enable or disable non-secure accesses to specific peripherals. By writing to the registers in the TZPC a program can change the values on these signals and thus modify the security state of a peripheral.

A dummy peripheral called a `DynamicBridge` is implemented using the OVP modeling technology. This can dynamically enable or disable a bridge for a specified address range. By configuring a `DynamicBridge` peripheral to bridge a specific non-secure region to the corresponding secure region and connecting it to an output from the TZPC, the platform can support programmatic control of non-secure accesses to any component or memory region in the specific memory region.

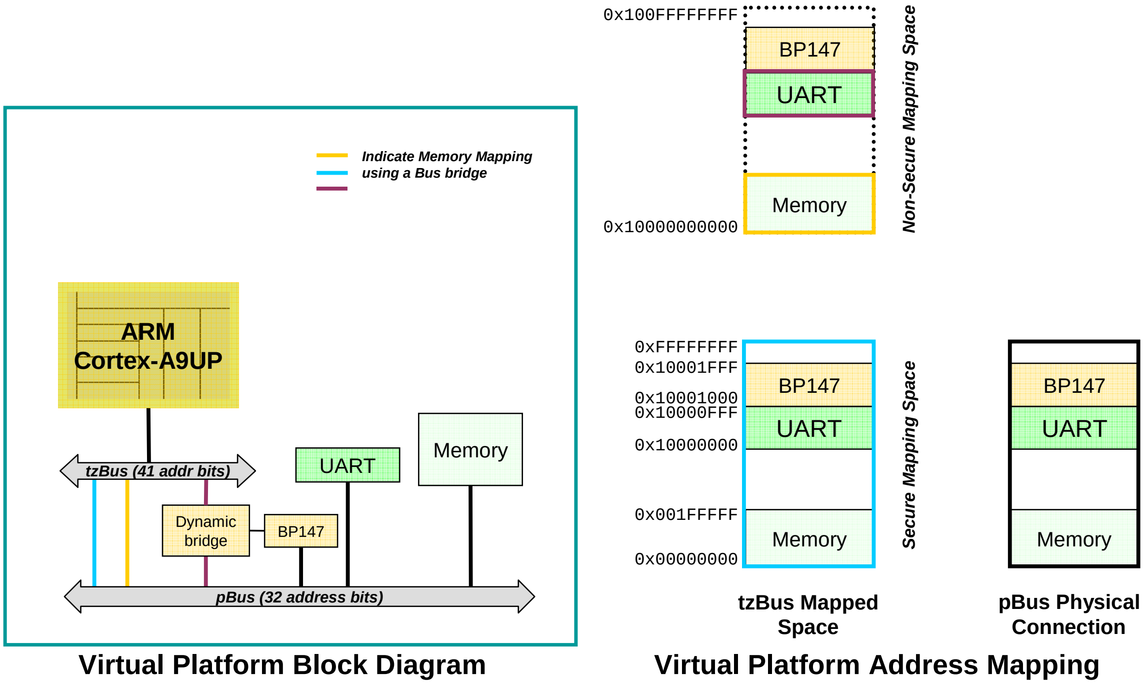


Figure 4: Platform 'programControl' Block Diagram and Memory Mapping

4.4.1 *programControl* Example Platform

We now modify the platform by:

1. Adding a BP147 TrustZone Controller peripheral
2. Adding a `DynamicBridge` peripheral for the UART address region
3. Connecting the enable input of the `DynamicBridge` to the `TZDECPROT0_0` output of the TZPC

This example may be found in:
`$IMPERAS_HOME/Examples/PlatformConstruction/ARM_specific/trustzone/4.programControl`

The changes from the previous platform are highlighted in **bold** (most of the unchanged code has been omitted for brevity):

```
...

ihwnew -name programControl

...

#
# UART:    0x10000000    0x10000fff   UartPL011
#
ihwaddperipheral -instancename uart0   -type UartPL011
ihwconnect       -instancename uart0   -busslaveport  bport1 -bus pBus \
                 -loaddress 0x10000000 -hiaddress 0x10000fff
```

```
ihwsetparameter  -handle       uart0   -name outfile -value uart0.log  -type string

#
# Dynamic Bridge: Mapping from non-secure space for uart0 under program control
#
ihwaddperipheral -instancename uart0NS -type DynamicBridge
ihwconnect       -instancename uart0NS -busslaveport  sp     -bus tzBus
ihwconnect       -instancename uart0NS -busmasterport mp     -bus pBus
ihwconnect       -instancename uart0NS -netport       enable -net tzpcdecprot0_0
ihwsetparameter  -handle     uart0NS -name mpLoAddress -value    0x10000000 –type uns64
ihwsetparameter  -handle     uart0NS -name spLoAddress -value 0x10010000000 –type uns64
ihwsetparameter  -handle     uart0NS -name portSize    -value        0x1000 –type uns64

#
# TZPC: 0x10001000    0x10001fff  BP147 TrustZone Protection Controller
#
ihwaddperipheral -instancename tzpc -type TzpcBP147
ihwconnect       -instancename tzpc -busslaveport bport1 -bus pBus \
                 -loaddress 0x10001000 -hiaddress 0x10001fff
ihwconnect       -instancename tzpc -netport TZPCDECPROT0_0 -net tzpcdecprot0_0

#
# Bridge - TZPC always accessible from non-secure mode
#
ihwaddbridge -instancename tzpcNS
ihwconnect   -instancename tzpcNS -busslaveport  sp -bus tzBus \
             -loaddress 0x10010001000 -hiaddress 0x10010001fff
ihwconnect   -instancename tzpcNS -busmasterport mp -bus pBus \
             -loaddress 0x10001000 -hiaddress 0x10001fff
...
```

Each new device is described in the following.

4.4.1.1 The DynamicBridge Peripheral

The DynamicBridge peripheral has a slave port input and a master port output. When enabled it creates a mapping from the slave port at the address specified by the attribute spLoAddress to the master port at the address specified by the attribute mpLoAddress for the extent specified by the attribute portSize.

The slave port is connected to the tzBus and the master port connected to the pBus:

```
#
# Dynamic Bridge: Mapping from non-secure space for uart0 under program control
#
ihwaddperipheral -instancename uart0NS -type DynamicBridge
ihwconnect       -instancename uart0NS -busslaveport  sp     -bus tzBus
ihwconnect       -instancename uart0NS -busmasterport mp     -bus pBus
```

Note that the slave port can only be connected to an unused region of a bus, while the master port may be connected to a region that is already used, thus creating an alias to that region.

The enable input of the DynamicBridge is connected to an output net from the TZPC, and will control whether the bridge is enabled or not at a given time:

```
ihwconnect       -instancename uart0NS -netport enable   -net tzpcdecprot0_0
```

The `DynamicBridge` must be configured to indicate the address ranges to bridge. This is done by specifying attributes defining the low address for both the master and slave port regions and the size of the port:

```
ihwsetparameter  -handle   uart0NS -name mpLoAddress -value    0x10000000 –type uns64
ihwsetparameter  -handle   uart0NS -name spLoAddress -value 0x10010000000 –type uns64
ihwsetparameter  -handle   uart0NS -name portSize    -value        0x1000 –type uns64
```

4.4.1.2 The TZPC Peripheral

The TZPC peripheral provides configuration to drive 24 output nets that can be used in conjunction with the `DynamicBridge` peripheral to enable mapping of non-secure accesses for memory regions, under program control.

The TZPC is defined and given its own separate address space on the bus:

```
#
# TZPC: 0x10001000    0x10001fff  BP147 TrustZone Protection Controller
#
ihwaddperipheral -instancename tzpc -type TzpcBP147
ihwconnect       -instancename tzpc -busslaveport bport1 -bus pBus \
                 -loaddress 0x10001000 -hiaddress 0x10001fff
```

We are only using one of the output ports from the TZPC, `TZDECPROT0_0`, which we connect to the same net we connected to the `DynamicBridge` enable port:

```
ihwconnect       -instancename tzpc -netport TZPCDECPROT0_0 -net tzpcdecprot0_0
```

Finally, in order to be able to access the TZPC from non-secure mode we add a bridge for its address region:

```
#
# Bridge - TZPC always accessible from non-secure mode
#
ihwaddbridge     -instancename tzpcNS
ihwconnect       -instancename tzpcNS -busslaveport  sp \
                 -bus smbus -loaddress 0x10010001000 -hiaddress 0x10010001fff
ihwconnect       -instancename tzpcNS -busmasterport mp \
                 -bus smbus -loaddress    0x10001000 -hiaddress    0x10001fff
```

⇒ Note that making the TZPC accessible from non-secure mode would make a real system insecure, since only a secure program should be able to program the TZPC, but is necessary for our simplified example which has no secure monitor program.

4.4.2 Example application

We must modify the application to change the `TZDECPROT0_0` setting in the TZPC in order to test the platform.

The changes from the previous application are highlighted in **bold** (most of the unchanged code has been omitted for brevity):

```
...

// TrustZone Protection Controller Registers
#define TZPC_BASE         0x10001000
#define TZPC_DECPROT0_SET ((volatile unsigned char *) (TZPC_BASE + 0x804))
#define TZPC_DECPROT0_CLR ((volatile unsigned char *) (TZPC_BASE + 0x808))
...

// Turn on TrustZone protection for UART
void protectUart() {
    *TZPC_DECPROT0_CLR = 1;
}

// Turn off TrustZone protection for UART
void unprotectUart() {
    *TZPC_DECPROT0_SET = 1;
}

...

int main() {

    // Setup and enable the secure mode TLB
    // Note: Processor starts out in secure mode
    setupTranslationTable();
    enableTLB();

    printf ("\n*** Writing to uart from Secure Mode\n");
    putString("Hello from secure mode!\n");

    printf ("\n*** Entering Non-Secure Mode\n");
    enterNonSecure();

    // Enable the non-secure TLB (uses same translation table as secure mode)
    enableTLB();

    printf ("\n*** Unprotect Uart\n");
    unprotectUart();

    printf ("\n*** Writing to unprotected uart from Non-Secure Mode\n");
    putString("Hello from non-secure mode (unprotected)!\n");

    printf ("\n*** Protect Uart\n");
    protectUart();

    printf ("\n*** Writing to protected uart from Non-Secure Mode\n");
    putString("Hello from non-secure mode (protected)!\n");

}
```

The application program has been changed to first unprotect and then protect the UART before accessing it in non-secure mode by clearing and then setting bit 0 of the `TZPCDECPROT0` register in the TZPC.

This is done by writing a 1 to bit 0 of the registers `TZPCDECPROT0Set` or `TZPCDECPROT0Clr` (at offset `0x804` and `0x808` respectively from the TZPC base address, which is defined in the platform to be `0x1001_0000`):

```
#define TZPC_BASE         ((volatile unsigned char *) 0x10001000)
#define TZPC_DECPROT0_SET ((volatile unsigned char *) (TZPC_BASE + 0x804))
```

```
#define TZPC_DECPROT0_CLR ((volatile unsigned char *) (TZPC_BASE + 0x808))

...

void protectUart() {
    *TZPC_DECPROT0_CLR = 1;
}

void unprotectUart() {
    *TZPC_DECPROT0_SET = 1;
}
```

See the document *ARM TrustZone Protection Controller (BP147) Technical Overview (DTO0015)* for details of the TZPC.

4.4.3 Building and running the example

The application and platform are in the directory:
`Examples/PlatformConstruction/ARM_specific/trustzone/4.programControl`
Take a copy of that directory and `cd` to it:

```
cp -r
$IMPERAS_HOME/Examples/PlatformConstruction/ARM_specific/trustzone/4.programControl .
cd 4.programControl
```

To compile, build and run the platform, use the command (see section 2.3 for info on setting up the toolchain needed by the `Makefile`):

```
make run
```

The application will run on the platform and you will see the following output:

```
*** Writing to uart from Secure Mode

*** Entering Non-Secure Mode

*** Unprotect Uart

*** Writing to unprotected uart from Non-Secure Mode

*** Protect Uart

*** Writing to protected uart from Non-Secure Mode
Processor Exception (PC_PRX) Processor 'programControl/cpu' 0x8284: e5d33000 ldrb    r3,[r3]
Processor Exception (PC_RPX) No read access at 0x10000018
```

Here we see that the attempt to read from the UART from non-secure mode when the protection has been enabled results in an exception.

The file `uart0.log` shows that the UART could be written from secure mode and from non-secure mode when unprotected, but not when protected:

```
Hello from secure mode!
Hello from non-secure mode (unprotected)!
```

5 The `DynamicBridge` Peripheral

This section explains the implementation of the `DynamicBridge` peripheral which dynamically creates or destroys a bus bridge under control of an input net.

The source for the peripheral may be found under:
`$IMPERAS_HOME/ImperasLib/source/ovpworld.org/peripheral/DynamicBridge`

5.1 The `pse.tcl` file

The ports and attributes are defined in the `pse.tcl` file:

```
...

imodelnewperipheral  \
    -imagefile   pse.pse  \
    -name        DynamicBridge \
    -vendor      ovpworld.org  \
    -library     peripheral \
    -version     1.0 \
    -constructor constructor

# Output bus size is 32 bits
set outputBusWidth  32

iadddocumentation -name Description -text $desc
iadddocumentation -name Licensing   -text "Open Source Apache 2.0"
iadddocumentation -name Limitations -text $limitations

# Define bus slave input port
imodeladdbusslaveport  -name sp -remappable -mustbeconnected

# Define bus master output port
imodeladdbusmasterport -name mp -addresswidth $outputBusWidth -mustbeconnected

# Define enable net input port
imodeladdnetport -name enable -type input -updatefunction updateEnable

# Define formals for specifying the output master and slave bus port location and size
imodeladdformal -name mpLoAddress -type integer
imodeladdformal -name spLoAddress -type integer
imodeladdformal -name portSize    -type integer
```

An input slave port named `sp` is defined. It is remappable, meaning that its address is defined by the peripheral source code, and it must be connected:

```
# Define bus slave input port
imodeladdbusslaveport  -name sp -remappable -mustbeconnected
```

An output master port named `mp` is defined. It supports connections to a bus with 32 address bits and must be connected:

```
# Define bus master output port
imodeladdbusmasterport -name mp -addresswidth $outputBusWidth -mustbeconnected
```

If different numbers of address bits are needed a separate peripheral would need to be created.

An input net port named `enable` is defined. The function `updateEnable` is specified to be called when the net connected to it changes. It may be left unconnected:

```
# Define enable net input port
imodeladdnetport -name enable -type input -updatefunction updateEnable
```

Finally, 3 formal attributes are defined for specifying the base addresses for the slave port and master port and the port size to be mapped:

```
imodeladdformal -name mpLoAddress -type integer
imodeladdformal -name spLoAddress -type integer
imodeladdformal -name portSize    -type integer
```

The **iGen** program is used to convert the tcl file into OVP peripheral template source. See the Imperas Peripheral Generator Guide for details.

5.2 The user.c file

iGen will create several `*.igen.c` files and `*.igen.h` files which implement the structure defined in the `pse.tcl` file. It will also create a stubs file to be modified to implement the desired functionality for the peripheral. This is called the `user.c` file by convention.

The `DynamicBridge` `user.c` file, modified to implement the model's behavior is:

```
#include "pse.igen.h"

#define PREFIX "DynamicBridge"

#define DIAG_1  (diagnosticLevel >= 1)
#define DIAG_2  (diagnosticLevel >= 2)
#define DIAG_3  (diagnosticLevel >= 3)

SimAddr mpLoAddress;
SimAddr spLoAddress;
Uns64   portSize;
Bool    bridgeEnabled = False;

//////////////////////////////// Callback stubs ////////////////////////////////

PPM_NET_CB(updateEnable) {

    Bool enable  =  value && !bridgeEnabled;
    Bool disable = !value &&  bridgeEnabled;

    if (disable) {

        // Disable dynamic bridge
        ppmDeleteDynamicBridge("sp", spLoAddress, portSize);
        bridgeEnabled = False;

    }

    if (enable) {
        // Enable dynamic bridge
        ppmCreateDynamicBridge("sp", spLoAddress, portSize, "mp", mpLoAddress);
        bridgeEnabled = True;

    }

    if (DIAG_1) {
        if (enable || disable) {
            bhmMessage(
                "I", PREFIX,
                "%s dynamic mapping for port of size %d at 0x%08llx",
                enable ? "Enable" : "Disable",
                (Uns32) portSize, mpLoAddress
            );
        }
    }
}

PPM_CONSTRUCTOR_CB(constructor) {

    periphConstructor();

    // Note this peripheral is a nop if no net connected to enable
    if (handles.enable) {

        if (!bhmUns64Attribute("mpLoAddress", &mpLoAddress) ||
            !bhmUns64Attribute("spLoAddress", &spLoAddress) ||
            !bhmUns64Attribute("portSize"   , &portSize)) {
            bhmMessage("F", PREFIX, "mpLoAddress, spLoAddress and portSize parameters
must all be defined");
        }
    }
}
```

See the *OVP Peripheral Modeling Guide and OVP BHM and PPM Function Reference* for details on peripheral modeling. The highlights of this model are described in the following.

If no net is connected then the model implements no behavior:

```
    // Note this peripheral is a nop if no net connected to enable
    if (handles.enable) {

        ...

    }
```

If a net has been connected to the enable input the attributes are checked to make sure that all of the `mpLoAddress`, `spLoAddress` and `portSize` have been specified on the model instance. It is a fatal error for these to be missing.

```
        if (!bhmUns64Attribute("mpLoAddress", &mpLoAddress) ||
            !bhmUns64Attribute("spLoAddress", &spLoAddress) ||
            !bhmUns64Attribute("portSize"   , &portSize)) {
            bhmMessage("F", PREFIX, "mpLoAddress, spLoAddress and portSize parameters
must all be defined");
        }
```

The `updateEnable()` function will be called whenever the net connected to the enable port changes value. It first determines whether the bridge is to be enabled or disabled based on the current state of the bridge and the new net value:

```
    Bool enable  =  value && !bridgeEnabled;
    Bool disable = !value &&  bridgeEnabled;
```

If the bridge is to be disabled, the `ppmDeleteDynamicBridge()` function is called to delete the bridge:

```
    if (disable) {

        // Disable dynamic bridge
        ppmDeleteDynamicBridge("sp", spLoAddress, portSize);
        bridgeEnabled = False;

    }
```

If it is to be enabled the `ppmCreateDynamicBridge()` function is called to create the bridge:

```
    if (enable) {

        // Enable dynamic bridge
        ppmCreateDynamicBridge("sp", spLoAddress, portSize, "mp", mpLoAddress);
        bridgeEnabled = True;

    }
```

When creating a bridge it is a fatal error if the slave port overlaps with any other port currently on the bus. The simulator will make this check when the bridge is created. The master port address may overlap, creating an alias.

Once the bridge is created, accesses to the slave port address are mapped to the master port with no additional overhead, allowing high performance simulation even in complex dynamic mapping schemes.

Creating and deleting the bridge does involve significant overhead so these functions should be used as efficiently as possible.

6 Summary

We have shown methods for both ignoring and implementing ARM TrustZone security in an OVP platform.

When implementing TrustZone security we have shown how to model peripherals that are always accessible in non-secure mode, never accessible in non-secure mode or where the accessibility of a peripheral is controlled by the program running on the processor.

We have also shown how to implement an OVP peripheral that uses dynamic bridges to efficiently model TrustZone at the platform level. These same techniques can be used in custom models of TrustZone-aware peripherals.

7 Further Imperas Tools to assist with TrustZone platforms and software

While this document has focused on optimal implementation of TrustZone-aware peripheral models for high simulation performance, additional benefits can be derived from other Imperas tools. Other test and analysis techniques that might be used to more comprehensively understand the behavior of a TrustZone platform or application include using fault injection to confirm that no unintended behavior is observed, using memory access monitoring to understand when and why accesses of secure memory regions are made and tracing of TrustZone behavior.

These tools are available as part of the Imperas M*SDK product. M*SDK includes the SlipStreamer™ API, which provides the basis for non-intrusive, low overhead tools. Using SlipStreamer, Imperas has developed a set of Verification, Analysis and Profiling (VAP™) tools, which have both CPU- and OS-aware capabilities. In addition, users can create custom tools by using SlipStreamer.

For more information on M*SDK, SlipStreamer and the VAP tools, please go to the Imperas website, www.Imperas.com, or email us at info@imperas.com.

##