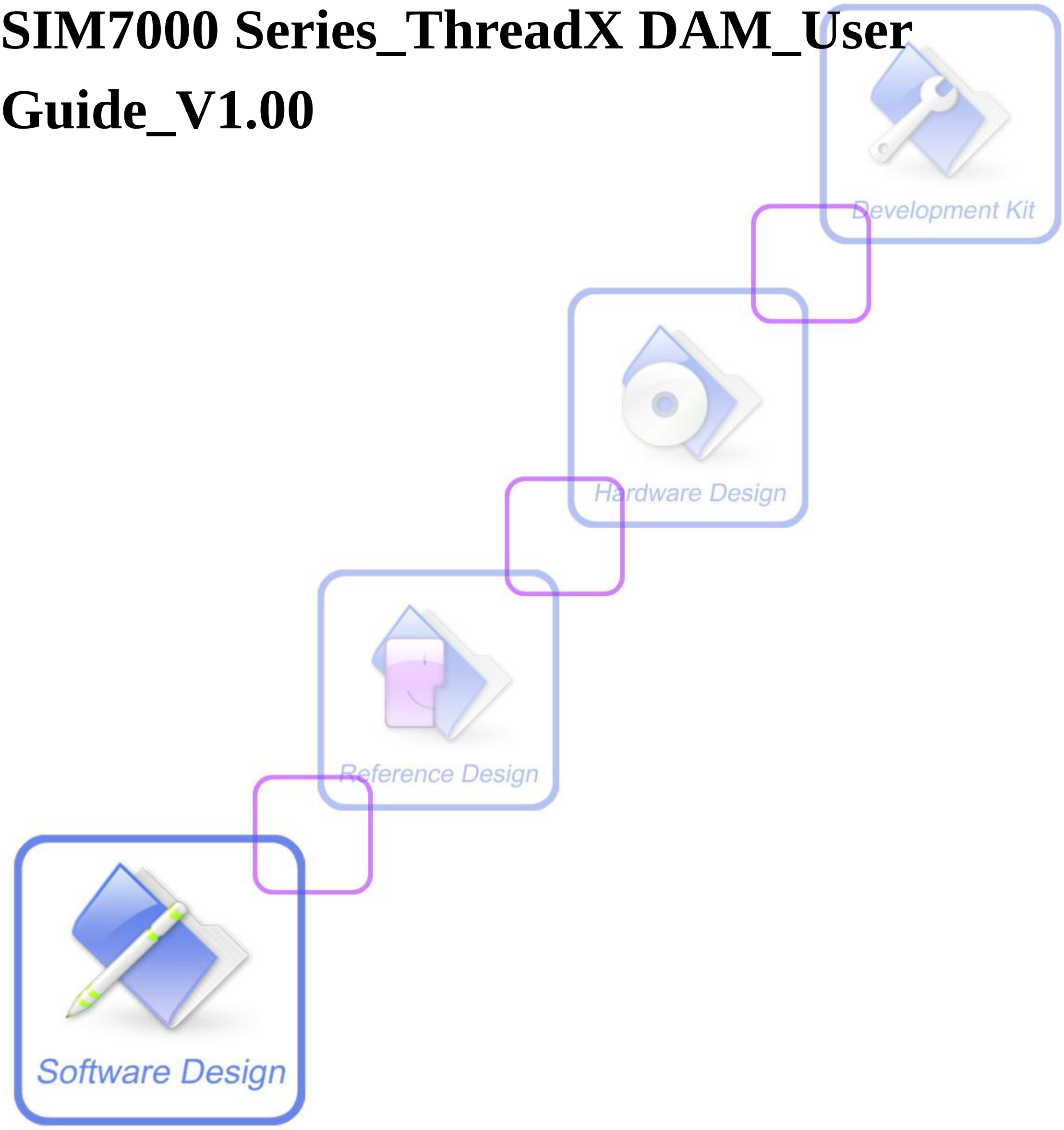
SIM7000 Series_ThreadX DAM_User Guide_V1.00



Document Title	**SIM7000 Series ThreadX DAM User Guide**
Version	**1.00**
Date	**2017-09-06**
Status	**Release**
Document Control ID	**SIM7000 Series_ThreadX DAM_User Guide_V1.00**

General Notes

SIMCom offers this information as a service to its customers, to support application and engineering efforts that use the products designed by SIMCom. The information provided is based upon requirements specifically provided to SIMCom by the customers. SIMCom has not undertaken any independent search for additional relevant information, including any information that may be in the customer's possession. Furthermore, system validation of this product designed by SIMCom within a larger electronic system remains the responsibility of the customer or the customer's system integrator. All specifications supplied herein are subject to change.

Copyright

Contents

Version History

Date	Version	What is new	Author
2017-09-06	1.00	New version	

Scope

This document presents the method of DAM. This document can apply to SIM7000 series modules.

1 Introduction

1.1 Purpose

This document describes the feature support that the ThreadX downloadable application module (DAM) provides to QCT platforms. This document provides the DAM build, load process, initialization, debug, update information, and sample code.

The QCT ThreadX module implementation adheres to the ThreadX DAM implementation specified at http://rtos.com/products/threadx/downloadable_application_modules.

This document assumes that users are aware of the ThreadX module and module manager implementation.

1.2 Conventions

Function declarations, function names, type declarations, attributes, and code samples appear in a different font, for example, #include.

Code variables appear in angle brackets, for example, <number>.

Commands to be entered appear in a different font, for example, **copy a:*.* b:**.

Button and key names appear in bold font, for example, click **Save** or press **Enter**.

2 ThreadX module overview

2.1 Introduction

The ThreadX module provides a framework for applications to load modules dynamically that are built separately from the resident component of the application. The Downloadable Application Module (DAM) is useful in the following scenarios:

- Application code size exceeds the available memory
- New modules are required after the core image is deployed
- Partial firmware updates are required.

The module component relies on the application to provide a memory space where the modules are available for loading. The instruction area of each module is executed in either of the following areas:

□ Memory space that is allocated for loading the module

□ RAM of the module memory

NOTE: The module memory requirements are allocated from the module memory area.

There is no limit on the number of modules that are loaded at the same time (apart from the amount of memory available). However, there is only one copy of the core resident module manager code.

2.2 ThreadX module and module manager

ThreadX module manager is a part of the ThreadX kernel extensions that provide interfaces to load, start, stop, and unload DAM instances. Each module is built independently with a common preamble structure attached to the binary. The preamble contains the basic characteristics and resource requirements of the module including a single thread entry point, required stack size, required data size, priority, module ID, callback thread stack size, and so on. To allocate resources, the module manager reads the preamble while loading the module. The module manager subsequently loads and starts the module.

Each module must have its own instruction area and data area as defined by the application. The module and module manager interact using a software dispatch function through a predefined request ID, which corresponds to services requested by the module.

The module manager creates the module thread and initiates its execution. After execution, the module manager retrieves the ThreadX API requests made by the module. The module has full access rights to the ThreadX API, including the ability to create additional threads within the module.

Figure 2-1 shows the relationship between module and module manager.

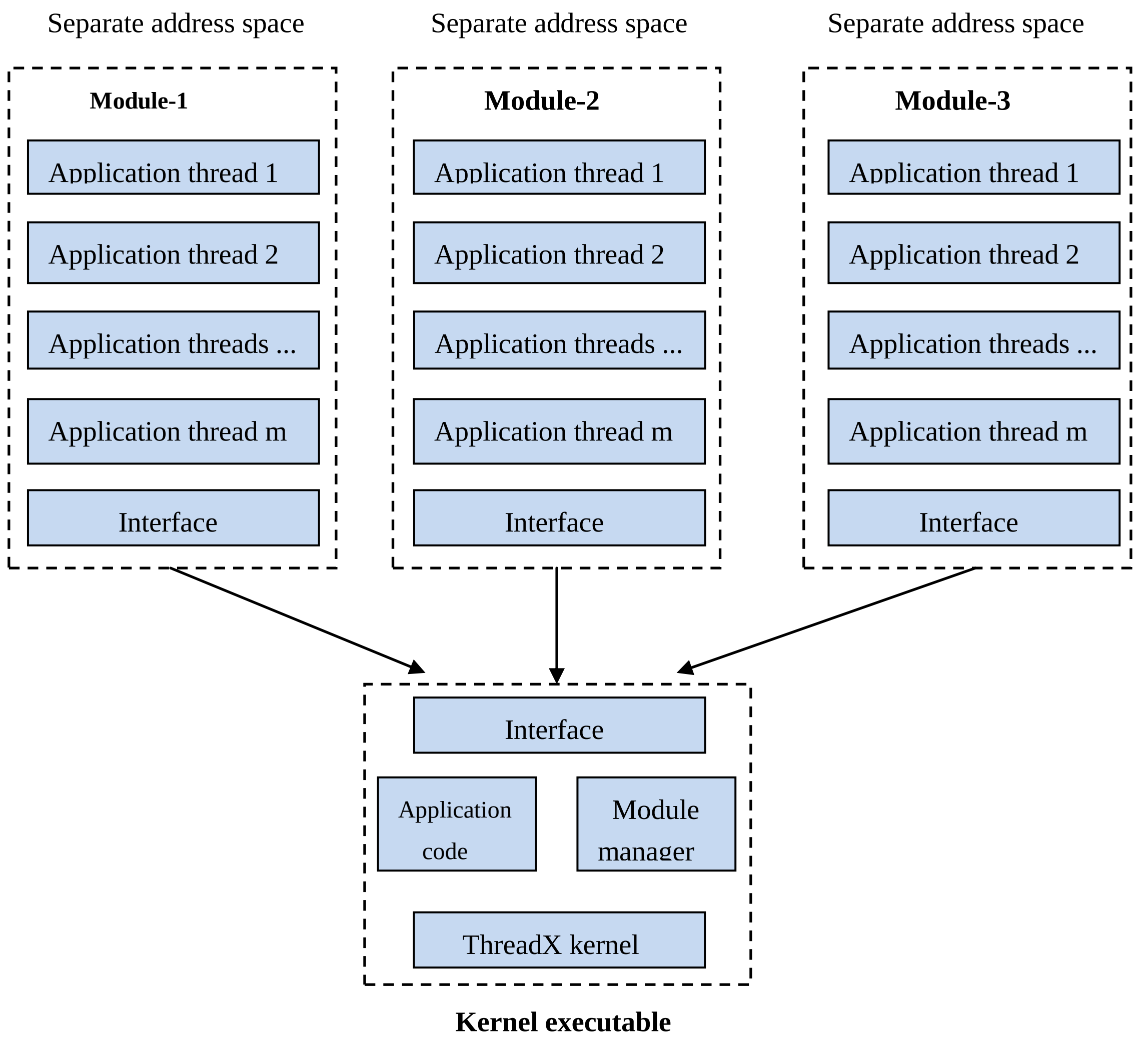


Figure 2-1 Relationship of module and module manager

For more details about ThreadX module, refer to ExpressLogic at:
http://rtos.com/products/threadx/downloadable_application_modules

3 DAM overview

The QCT ThreadX DAM module implementation adheres to the ThreadX module framework. Module manager, Dispatcher, OEM Module loader, QAPI, and MDM9206 platform software all reside in ThreadX Kernel Space. DAM module (code and data) resides in User Space and is separate from Kernel Space. QAPI (bridge) is the interface between DAM module and QTI drivers while standard TX API is the interface between DAM module and Module manager. Each device contains one Module manager and one Dispatcher from MDM9206 platform software. However, the number of OEM Module loaders (which call Module manager APIs to load the module) and module images are depended up on the number of DAM modules. Each DAM module has a unique Preamble and an OEM Module loader.

NOTE: OEM is responsible creating the OEM Module loader and its associated preamble for each DAM module.

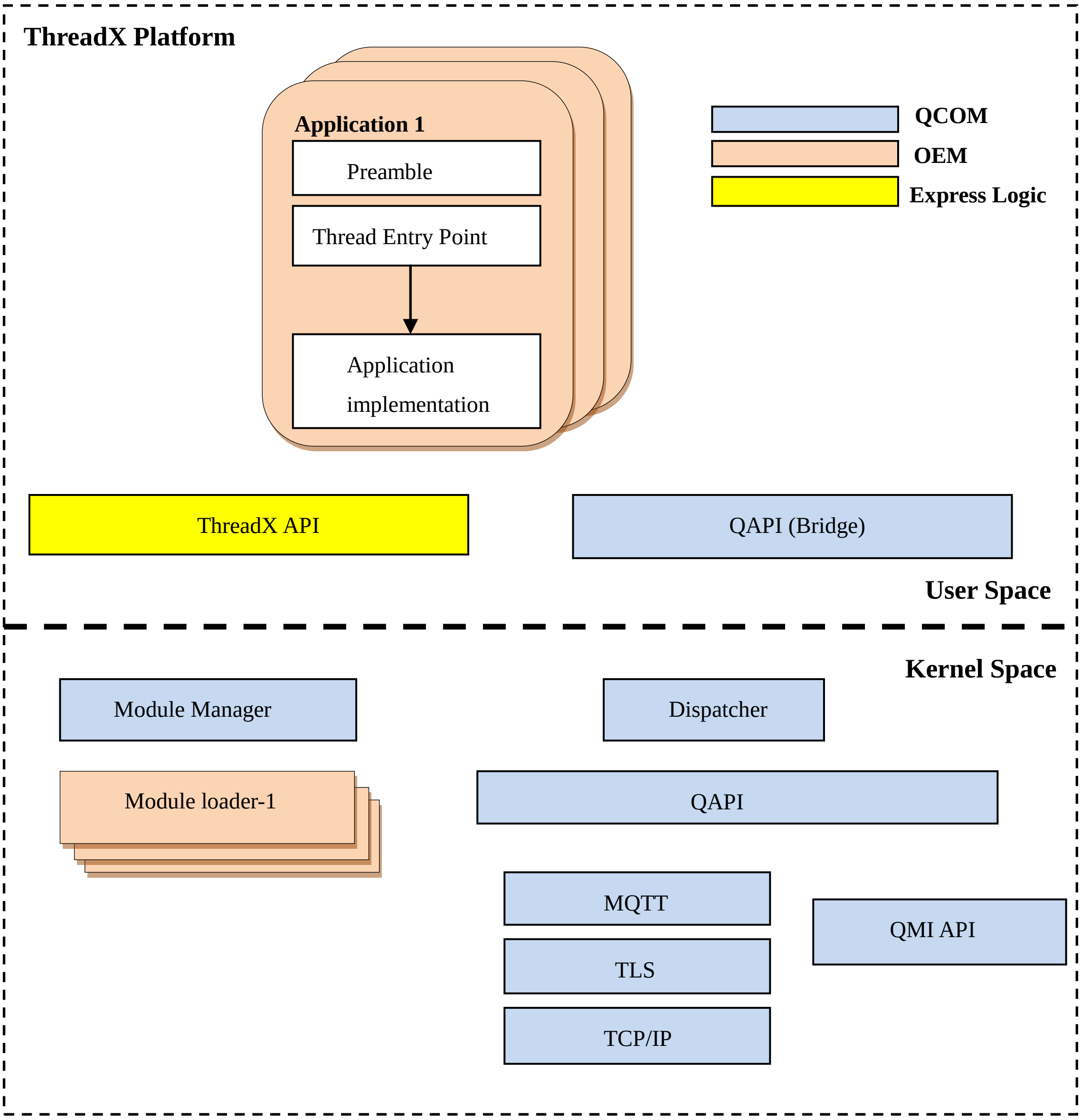


Figure 3-1 QTI ThreadX module overview diagram

3.1 DAM call flow example

Figure 3-2 illustrates an example of ThreadX DAM module call flow.

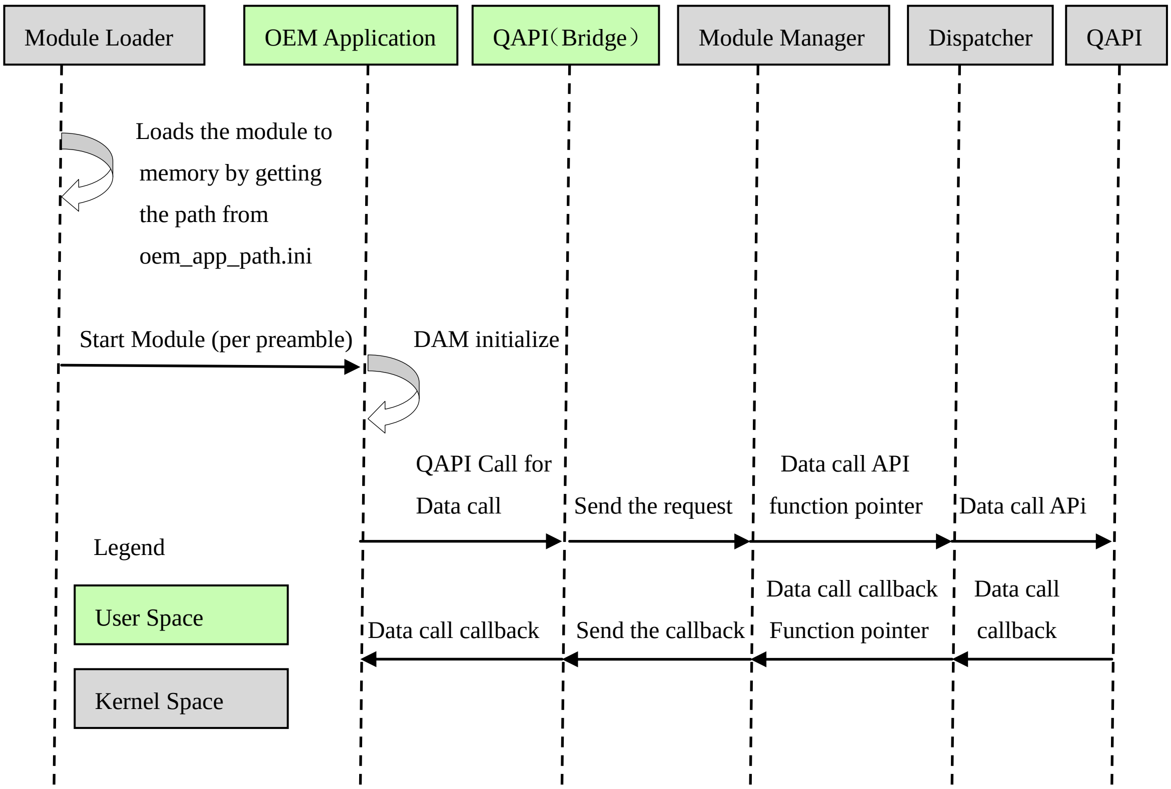


Figure 3-2 Example of ThreadX DAM module call flow

This example of ThreadX DAM module call flow demonstrates loading of a new OEM application ThreadX Module that performs a Data call bringup.

- OEM Application and QAPI (Bridge) are in User Space.
- Module manager, Dispatcher, and QAPI are in Kernel Space.
- Module loader is in Kernel Space and it is developed by OEMs with Kernel software access.
- Module loader loads the application DAM module image to memory by getting the location image from a configuration file (for example, oem_app_path.ini).
- Module loader starts OEM application module with Preamble defined by customers.
- OEM DAM application initialization and setup a Data call via QAPI Bridge.
- Module manager handles the Data call with Dispatcher and QAPI.
- Data call response is established to the OEM application via callback function.

3.2 ThreadX software versions

- ThreadX version on QCT platform – ThreadX Cortex-A7/ARM vG5.7.5.2
- ThreadX module version on QCT platform – ThreadX Module Cortex-A7/ARM v5

4 DAM development key components

4.1 Important points

- C module files must include the following: □
  - TX_MODULE prior to including txm_module.h
  - Remaps the thread APIs that invoke dispatch
- Each module must have a Preamble
  - Defines module characteristics and resources
  - Located at the first instruction area address
  - txm_module_preamble.S-Preamble assembly file□
- Each module must link against a module library (txm_lib.lib) □
  - Module-specific functions to interact with ThreadX

4.2 Preamble

DAM module Preamble defines characteristics and resources of the module. The Preamble in ThreadX module is followed by the instruction area of the module. It is always at the first address of the module.

- Module preamble
- Module instruction area
- Module RAM area

The Preamble is in assembly file format. This file defines various module-specific attributes and is typically linked with the module application code. Key Preamble items are as follows:

- Module ID
- Module single thread entry point
- Module thread stack size
- Module thread priority
- Module callback thread entry point
- Module callback thread stack size
- Module callback thread priority
- Module code size
- Module data size
- More

NOTE: For each DAM application module implemented by a customer, there is a unique Preamble defined by the customer for that DAM module.

Additional information about Preamble is in the “Module Preamble” section of ThreadX Module User Guide document from Express Logic.

4.2.1 Example preamble

custApp\src\txm_module_preamble.S

```
AREA Init, CODE, READONLY CODE32 ; /* Define public symbols. */
EXPORT __txm_module_preamble ;
  /* Define application-specific start/stop entry points for the module. */
IMPORT qcli_dam_app_start ; /* Define common external refrences. */
IMPORT _txm_module_thread_shell_entry
IMPORT _txm_module_callback_request_thread_entry
IMPORT |Image$$ER_RO$$Length|

#ifdef TX_DAM_QC_CUSTOMIZATIONS
IMPORT |Image$$ER_RW$$Length|
IMPORT |Image$$ER_ZI$$ZI$$Length|
#endif

__txm_module_preamble
    DCD     0x4D4F4455          ; Module ID
    DCD     0x5                 ; Module Major Version
    DCD     0x3                 ; Module Minor Version
    DCD     32                  ; Module Preamble Size in 32-bit words
    DCD     0x12345678          ; Module ID (application defined)
    DCD     0x01000000          ; Module Properties where:
                                ; Bits 31-24: Compiler ID
                                ; 0 -> IAR
                                ; 1 -> RVDS
                                ; 2 -> GNU
                                ; Bits 23-0: Reserved
    DCD      _txm_module_thread_shell_entry - . + .      ; Module Shell Entry Point
    DCD      qcli_dam_app_start - . + .                  ; Module Start Thread Entry Point
    DCD      0                                           ; Module Stop Thread Entry Point
    DCD      140                                         ; Module Start/Stop Thread Priority
    DCD      8192                                        ; Module Start/Stop Thread Stack Size
    DCD      _txm_module_callback_request_thread_entry - . + .
                                                         ; Module Callback Thread Entry
    DCD      25                                          ; Module Callback Thread Priority
    DCD      2046                                        ; Module Callback Thread Stack Size
    DCD      |Image$$ER_RO$$Length|                      ; Module Code Size
    #ifdef TX_DAM_QC_CUSTOMIZATIONS
    DCD      |Image$$ER_ZI$$ZI$$Length|                  ; Module data size - get it from ZI
section
    DCD      __txm_module_preamble                       ; Reserved 0
    DCD      |Image$$ER_RW$$Length|                      ; Reserved 1
    #else
    DCD      0x16000                                     ; Module Data Size - default to 16K
```

```
(need to make sure this is large enough for module's data needs!)
DCD     0                                   ; Reserved 0
DCD     0                                   ; Reserved 1
```

4.3 Module library

Each DAM module must link to the module-centric ThreadX library. This library provides DAM module access to ThreadX services in the resident code. This access can be accomplished via macros defined in txm_module.h.

The following example shows how the module can access the resident module manager:

```
    extern VOID (*_txm_module_kernel_call_dispatcher)
    (structTXM_MODULE_KERNEL_REQUEST_STRUCT *);

    #define TXM_MODULE_KERNEL_CALL(r)
    (r) -> txm_module_kernel_request_status = \ TX_FEATURE_NOT_ENABLED; \
    if (_txm_module_kernel_call_dispatcher) \
        (*_txm_module_kernel_call_dispatcher)(r)
```

4.4 Module

The QCLI_DAM_DEMO module Preamble uses the qcli_dam_app_start start method.

Module example

custApp\src\app\src\pal_module.c

```
int qcli_dam_app_start(void)
{
    int   Result = PAL_Initialize();     /*Sleep for 100 ms*/
    tx_thread_sleep(10);                 /* Initialize the platform.*/
    if(Result)
    {
        /* Start the main demo thread. */
        Result = tx_thread_create(&Thread_Handle, "QCLI DAM Thread", QCLI_Thread,
    152, app_stack,QCLI_STACK_SIZE, 152, 152,
        TX_NO_TIME_SLICE, TX_AUTO_START);

        if(Result != TX_SUCCESS)
        {
            PAL_CONSOLE_WRITE_STRING_LITERAL("Failed to start QCLI thread.");
            PAL_CONSOLE_WRITE_STRING_LITERAL(PAL_OUTPUT_END_OF_LIN
            E_STRING);
            PAL_CONSOLE_WRITE_STRING_LITERAL(PAL_OUTPUT_END_OF_LIN
            E_STRING);
        }
    }
    return(TX_SUCCESS);
}
```

in the same file as above, in this function we allocate byte pool for this module.

```
static qbool_t PAL_Initialize(void)
{
        uint8_t Ret_Val=true;
        int filehandle = -1;
        uint32_t bytes;
        uint8 val = 0;
        qapi_UART_Open_Config_t open_properties;
        tx_byte_pool_create(&byte_pool_qcli, "byte pool 0", free_memory_qcli,
CLI_BYTE_POOL_SIZE);
        memset(&PAL_Context_D, 0, sizeof(PAL_Context_D)); memset (&open_properties, 0,
sizeof (open_properties));
        qapi_FS_Open( "/datatx/qcli_config", QAPI_FS_O_RDONLY_E, &filehandle);
        if(filehandle < 0 )
        {
                return 0;
        }
        else
        {
                qapi_FS_Read(filehandle, (void *)&val, sizeof(val), &bytes);
        }       /* ASCII value of '1' */
        if(val != 0x32)
        {
                qapi_FS_Close(filehandle);
                return 0;
        }
        open_properties.parity_Mode=QAPI_UART_NO_PARITY_E;
        open_properties.num_Stop_Bits=QAPI_UART_1_0_STOP_BITS_E;
        open_properties.baud_Rate = 115200;
        open_properties.bits_Per_Char=QAPI_UART_8_BITS_PER_CHAR_E;
        open_properties.rx_CB_ISR = cli_rx_cb;
        open_properties.tx_CB_ISR = NULL;
        open_properties.enable_Flow_Ctrl = false;
        open_properties.enable_Loopback= false;
        if(qapi_UART_Open(&PAL_Context_D.uart_handle,QAPI_UART_PORT_001_E,
&open_properties) != QAPI_OK)
        {
                Ret_Val = false;
        }
        else
        {
                qapi_UART_Receive(PAL_Context_D.uart_handle,(char*)&(PAL_Context_D.Rx_Buf
        fer[PAL_Context_D.Rx_In_Index]), PAL_RECIEVE_BUFFER_SIZE, (void*)1);
        }
```

```
    return (Ret_Val);
}
```

4.5 QCT module memory map changes

4.5.1 OEM memory pool

OEM memory pool is defined for the module manager and modules that are downloaded and executed. The following variables (defined under targacinaaaza.h) control this memory pool:

- OEM_POOL_START – By default, this variable is placed at the end of the modem (MPSS) image. This variable must not be placed in another location without being aware of the system memory map and usage.
- OEM_POOL_SIZE – By default, the pool size lies between ACDB region and MPSS region as indicated in Figure 4-1. It is not recommended to change the size without the required knowledge of the system memory map and module manager + module memory requirements.

4.5.2 Virtual pool (range) for modules

A virtual address space of 512 MB, ranging from 0x40000000 to 0x5FFFFFFF has been reserved for Module Manager and Modules.

Modules must be built within the Virtual Address range 0x4000_0000 to 0x5FFF_FFFF.

Each module must be allocated a unique Virtual Address range (separation among Modules is implementation defined).

Example :

Module-1 VA base address – 0x4000_0000

Module-2 VA base address – 0x4010_0000

Module-3 VA base address – 0x4020_0000

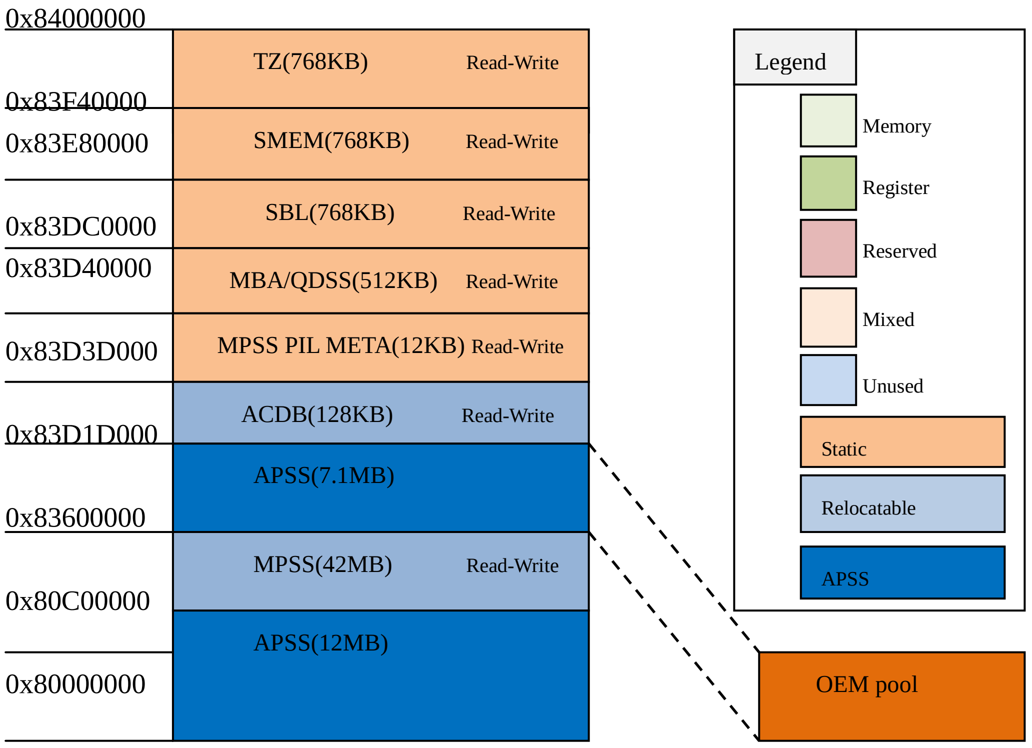


Figure 4-1 Memory map

4.5.3 Run-time mapping of modules by module loader

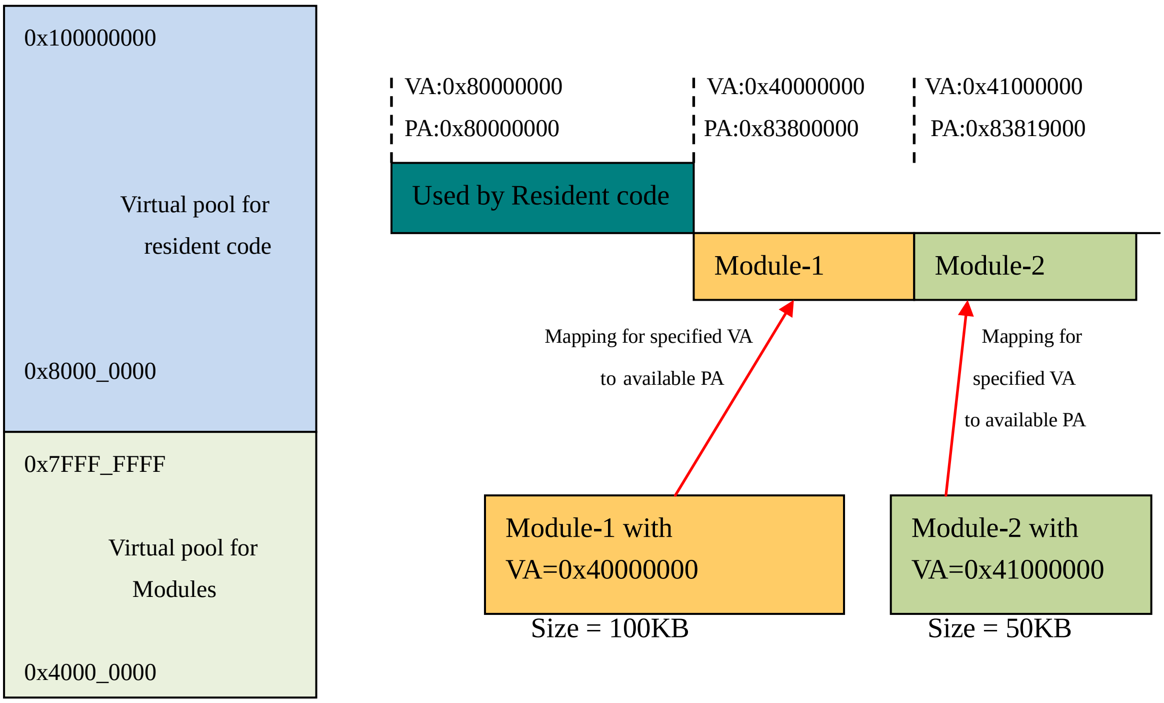


Figure 4-2 Run-time mapping of module

Physical memory pool for module manager and modules is reserved. Memory mapping is done per module for system provided PA to the module compile-time VA.

5 Tools for DAM

Below is a list of tools for DAM module build and load processes. The Comments column specifies the Module OEM responsibilities for their customers (device vendors) including to develop their own tools (flashing tool, log collection tool, file loading to file system tool) and share build compilation tool information in their release notes.

Item	Version	Source	Purpose	Comments
ARM Compiler toolchain (RealView Compilation Tools (RVCT))	ARMCT505B106	ARM Ltd.	Build compilation	Must be listed in module OEM document release to device vendors
Qualcomm Product Support Tool (QPST)	Latest version	QTI	Flash builds and access files	QPST is a QTI license tool that cannot be released by OEMs to device vendors. Module OEM responsibility to create their own flashing tool and EFS load tool.
Qualcomm extensible diagnostic monitor (QXDM)	Latest version	QTI	Log collection	QXDM is a QTI license tool that cannot be released by OEMs to device vendors. Module OEM responsibility to create their own log collection tool.
SCons	2.0.0.final.0 or higher	www.scons.org	Construction tool required to build	Must be listed in module OEM document release to device vendors
Python	ActiveState	ActiveState	Support build	Must be listed in module

			scripts	OEM document release to device vendors
USB network driver combo		QTI	Windows host USB drivers for QTI composite devices	Module OEM responsibility

6 DAM module build process

MDM9206 TX.2.0 has introduced ThreadX DAM, which enables compiling, linking, and loading application modules independently from the main Apps image. DAM application module images must be buildable and linkable with ThreadX module-specific User Space libraries.
This section focuses on the DAM application build process assuming the MDM9206 TX platform image has been completely built.

6.1 Key points of DAM build process

- As part of build process, all QAPI headers are dynamically placed at custApp\include\ and all modules can include QAP as the required public API for these QAPI headers.
- There are no QURT APIs available to application modules. The modules are expected to use the TX module APIs documented in txm_module.h which is available at custApp\include\threadx_api\
- For dynamic allocations, TX byte/block pools interfaces are expected to be used. The AMSS heap interfaces are not exposed to modules.
- Refer to ThreadX Modules User Guide for more information.

6.2 Compile txm_demo module

Run the following command:

```
build.sh
```

After execution, the ELF and binary files are placed at the following locations:

- ELF file –custApp custApp\bin\cust_app.elf
- Binary file –custApp custApp\bin\cust_app.bin

After the Demo module is built, copy custApp\bin\cust_app.bin as \custapp\cust_app.bin the devices alternate file system.

7 Image load process

- Before loading DAM application, MDM9206 TX platform image must be loaded to a device.
- Create a folder /custapp in alternate file system.
- Use EFS Explorer in QPST to copy the created executable (cust_app.bin) to /custapp/cust_app.bin in the file system.
- This method requests a reboot to pick up and start the ThreadX DAM module.
- If the /custapp/cust_app.bin is present during bootup, the module manager initiates the DAM module.

APPENDIX

A. Related Documents

SN	Document name	Remark
[1]	SIM7000 Series_AT Command Manual	

B. Terms and Abbreviations

Abbreviation	Description
DAM	Downloadable Application Moudle

Contact us:

Shanghai SIMCom wireless solutions Ltd.

Address: Building A, SIM Technology Building, No. 633 Jinzhong Road, Shanghai,
P. R. China 200335
Tel: +86 21 3252 3300
Fax: +86 21 3252 3020
URL: www.simcomm2m.com