SIM7070_SIM7080_SIM7090 Series_ThreadX DAM_User Guide

LPWA Module

SIMCom Wireless Solutions Limited
SIMCom Headquarters Building, Building 3, No. 289 Linhong Road, Changning District, Shanghai P.R. China
Tel: 86-21-31575100
support@simcom.com
www.simcom.com

Document Title:	SIM7070_SIM7080_SIM7090 Series_ThreadX DAM_User Guide
Version:	1.00
Date:	2021.09.18
Status:	Released

SIMCom Wireless Solutions Limited
SIMCom Headquarters Building, Building 3, No. 289 Linhong Road, Changning District, Shanghai P.R. China
Tel: +86 21 31575100
Email: simcom@simcom.com

For more information, please visit:
https://www.simcom.com/download/list-863-en.html

For technical support, or to report documentation errors, please visit:
https://www.simcom.com/ask/ or email to: support@simcom.com

About Document

Version History

Version	Date	Owner	What is new
V1.00	2021.09.18	Wenjie.lai	First Release

Scope

This document applies to the following products

Name	Type	Size(mm)	Comments
SIM7080G	CAT-M/NB	17.6*15.7*2.3	N/A
SIM7070G/SIM7070E	CAT-M/NB/GPRS	24*24*2.4	N/A
SIM7070G-NG	NB/GPRS	24*24*2.4	N/A
SIM7090G	CAT-M/NB	14.8*12.8*2.0	N/A

Contents

1 Introduction

1.1 Purpose of the document

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 Related documents

[1] SIM7070_SIM7080_SIM7090 Series_AT Command Manual

1.3 Conventions and abbreviations

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**.

Abbreviation	Description

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.

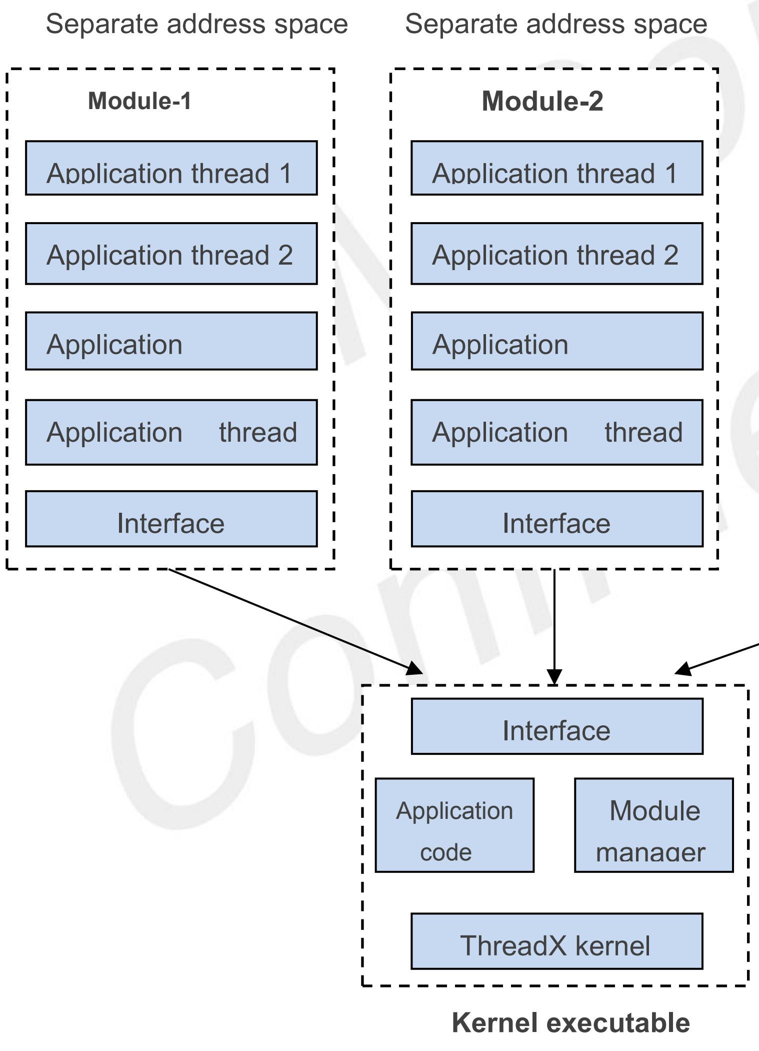


Figure 2-1 Relationship of module and module manager

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 MDM9205 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 MDM9205 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.

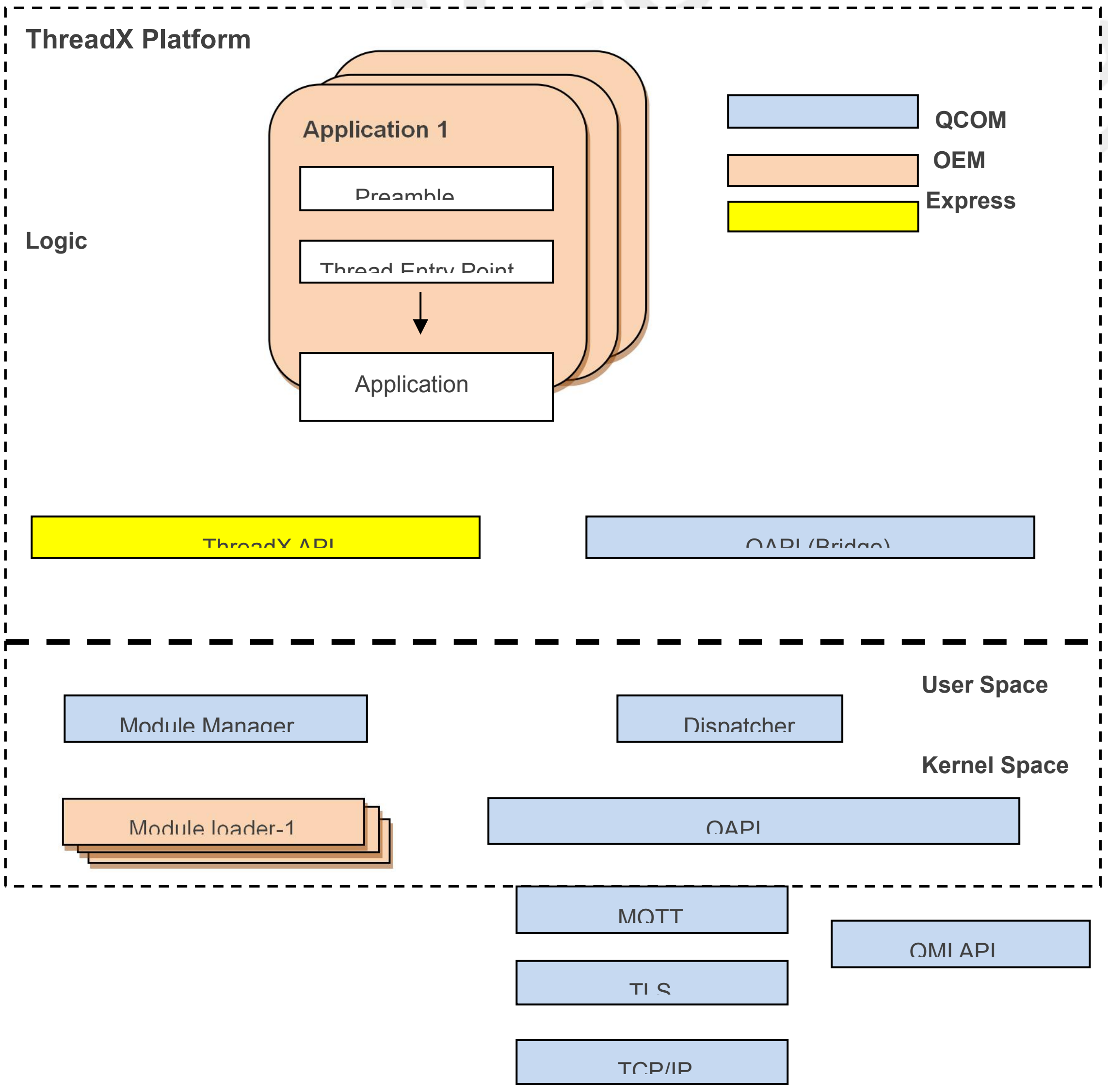


Figure 3-1 QTI ThreadX module overview diagram

3.1 DAM call flow example

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

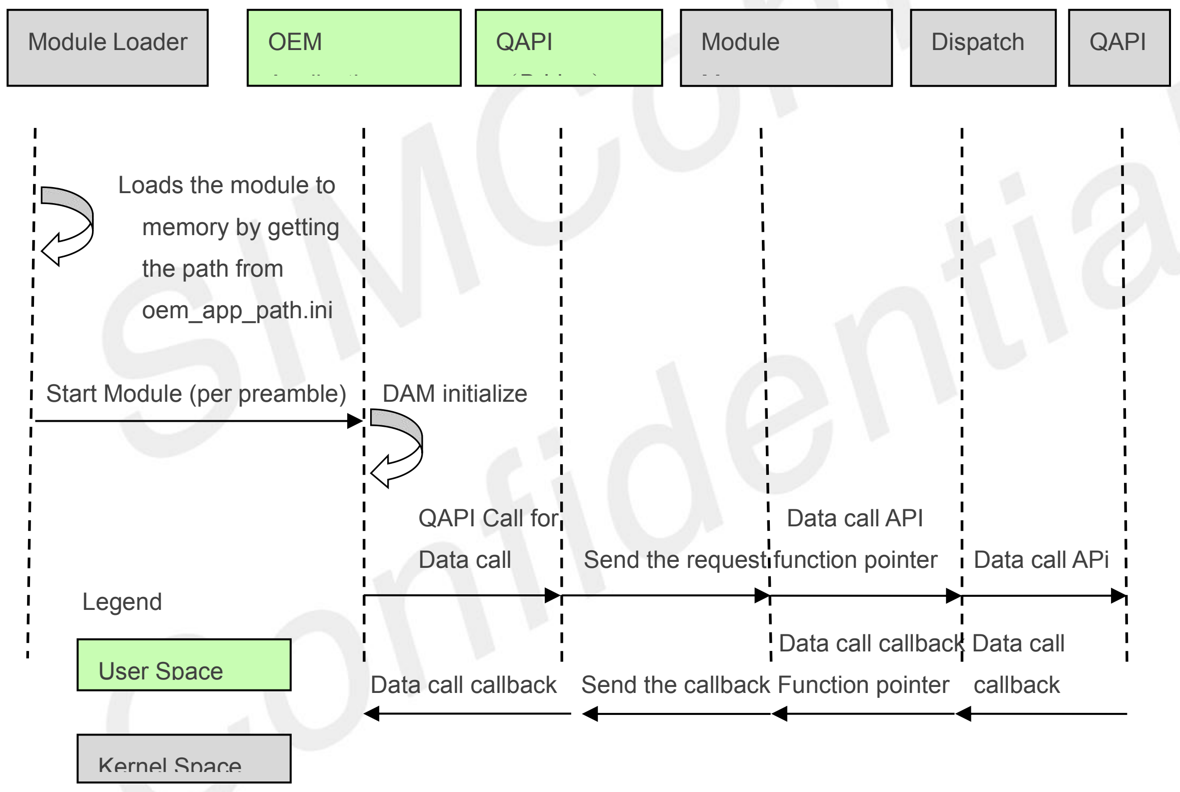


Figure 3-2 Example of ThreadX DAM module call flow

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

- 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.

4.4.1 Module example

```
code\src\app\qcli\pal_module.c
int qcli_dam_app_start(void)
```

```
{
    int                         Result = PAL_Initialize();
    Debug_Printf("demo app is start\r\n");
    /*Sleep for 100 ms*/
    tx_thread_sleep(10);
    /* Initialize the platform.                                          */
    txm_module_object_allocate(&Thread_Handle, sizeof(TX_THREAD));
    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_LINE_STRING);
          PAL_CONSOLE_WRITE_STRING_LITERAL(PAL_OUTPUT_END_OF_LINE_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;
    qapi_UART_Open_Config_t open_properties;
    txm_module_object_allocate(&byte_pool_qcli, sizeof(TX_BYTE_POOL));
    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));

#ifdef UART_PORT_CONTROL

    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 = dam_cli_rx_cb;
    open_properties.tx_CB_ISR = dam_cli_tx_cb;
    open_properties.enable_Flow_Ctrl = false;
    open_properties.enable_Loopback= false;
    if(qapi_UART_Open(&PAL_Context_D.uart_handle, QAPI_UART_PORT_003_E, &open_properties) !=
QAPI_OK){
          Ret_Val = false;
    }else{
```

```
qapi_UART_Receive(PAL_Context_D.uart_handle,(char*)&(PAL_Context_D.Rx_Buffer[PAL_Context_D.R
x_In_Index]), PAL_RECIEVE_BUFFER_SIZE, (void*)1);
    }
#else
        usb_status = qapi_USB_Open();
        if (usb_status == QAPI_OK) {
            static qapi_USB_App_Rx_Cb_t pcall;
            pcall = usb_callback;
            qapi_USB_Ioctl(QAPI_USB_RX_CB_REG_E, (void *)pcall);
        }
#endif
    return (Ret_Val);
}
```

Note the difference between TX 2.0 and TX 3.0 thread API

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

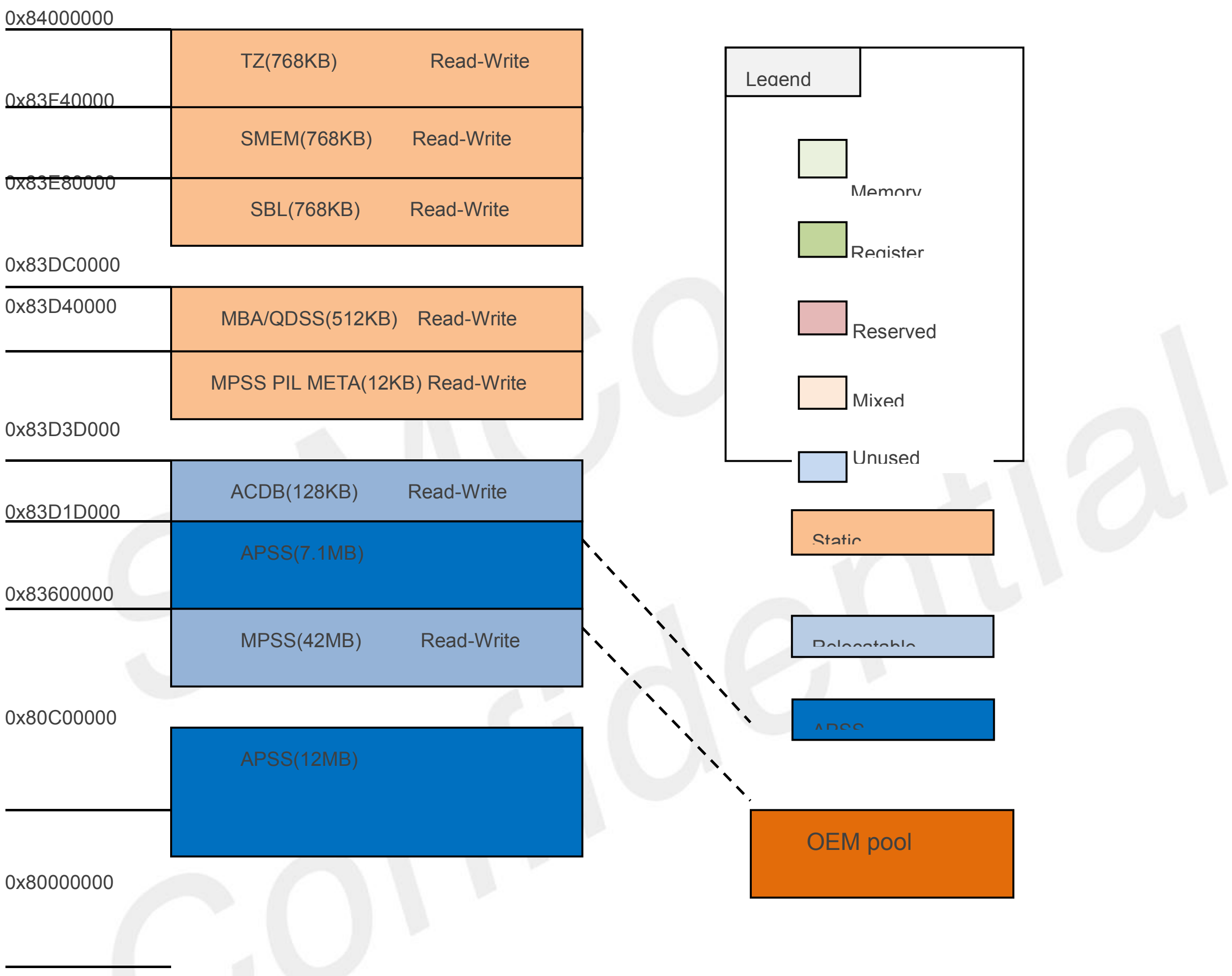
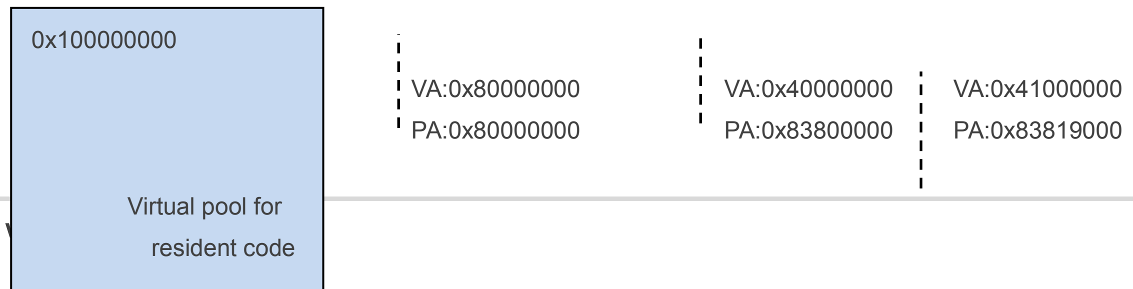


Figure 4-1 Memory map

4.5.3 Run-time mapping of modules by module loader



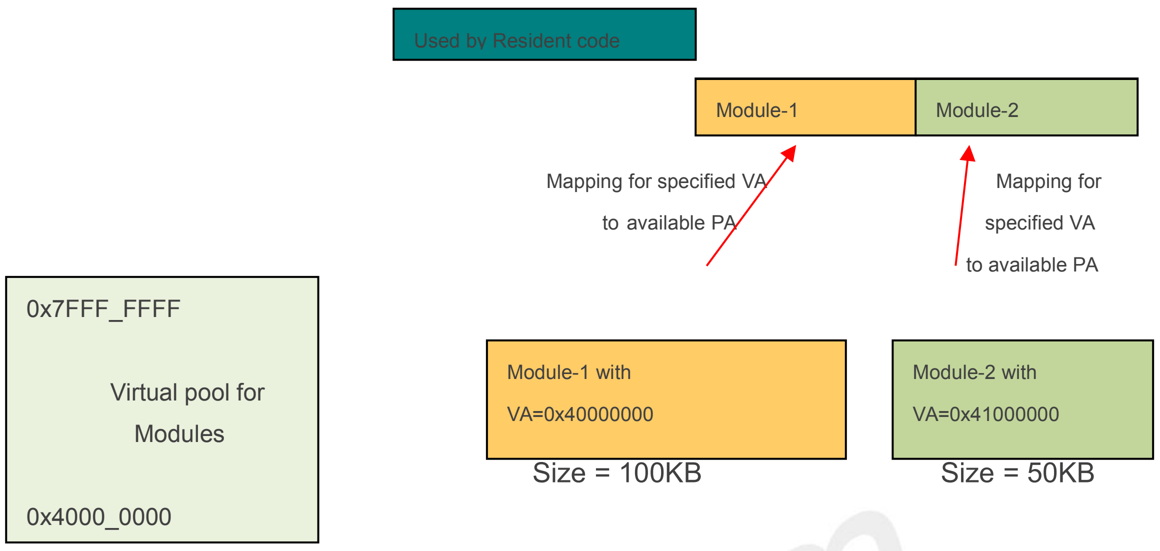

Used by Resident code
Module-1
Module-2
Mapping for specified VA to available PA
Mapping for specified VA to available PA
0x7FFF_FFFF
Virtual pool for Modules
0x4000_0000
Module-1 with VA=0x40000000
Size = 100KB
Module-2 with VA=0x41000000
Size = 50KB

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 DAM with MMU

5.1 Introduction of DAM with MMU

Any DAM application (user space) code that accesses the Kernel space variable results in a crash because of access violation. This breaks the callback handling, if the callback takes pointers in their arguments whose memory is allocated in the kernel space. Kernel code can still access user- supplied variable/pointers.

5.2 Solution for QTI QAPIs

- The DAM application has to initialize a byte pool pointer using ThreadX APIs and pass it as an input to the newly introduced QAPI.
- Kernel (Resident) application need not make any change.
- Steps to allocate Byte pool pointer:
  1. TX_BYTE_POOL * byte_pool; // Should be global variable
  2. txm_module_object_allocate(&byte_pool, sizeof(TX_BYTE_POOL));
  3. tx_byte_pool_create(byte_pool, "byte pool 0", free_memory, BYTE_POOL_SIZE);

Note: Sample implementation for byte pool creation can be seen under PAL_Initialize() in \apps_proc\dataservices\demo\QCLI_DAM_demo\src\qcli\pal_module.c

Note: Chunk of memory from the application byte pool is used for internal module allocation (to move the information/responses from the kernel to the user space). The DAM application must take this memory into account during byte pool allocation. It also depends on how many different modules are used by the application. For example, LwM2M + HTTP + DSS.

5.3 Modules and corresponding QAPIs to allocate byte pool pointer

Module	QAPI to allocate byte pool pointer	Description

DSS	Step1:
	qapi_DSS_Get_Data_Srvc_Hndl(dss_test_cb,(void*)&dss_net_hndl[index],&(dss_net_hndl[index].call_hndl)
	
	Step2:
	qapi_DSS_Pass_Pool_Ptr(dss_net_hndl[index].call_hndl,byte_pool_qcli)
		The Get DSS service handle will be functional only if the corresponding net handle and the byte pool pointer are associated.
		
		Byte_pool_qcli must be initialized as per the steps mentioned to allocate byte pool pointer in the

MQTT	Step1: qapi_Net_MQTT_New(&app_mqttcli_ctx) Step2: qapi_Net_MQTT_Pass_Pool_Ptr (app_mqttcli_ctx,byte_pool_qcli);	The MQTT functionality will be functional only if the corresponding MQTT handle and the byte pool pointer are associated. Byte_pool_qcli needs to be initialized as per the steps mentioned to allocate byte pool

5.4 Modules and corresponding QAPIs to allocate byte pool pointer (cont.)

Module	QAPI to allocate byte pool pointer	Description
HTTPc	Step1: Arg->client=qapi_Net_HTTPc_New_sess(timeout,arg->sslCtx,http_client_cb_demo,(void *)arg,httpc_max_body_length,httpc_max_header_length) Step2: qapi_Net_HTTPc_Pass_Pool_Ptr(arg->client,byte_pool_qcli)	HTTPc connect will be functional only if the corresponding HTTPc session and the byte pool pointer are associated. Byte_pool_qcli needs to be initialized as per the steps mentioned to allocate byte pool pointer in the slide Solution for QTI QAPI

ATFwD	Step1: qapi_atfwd_Pass_Pool_Ptr(at_fwd_callbac k, byte_pool_qcli) Step2: qapi_atfwd_reg(at_cmd_name_list, at_fwd_callback); Once there are no AT commands associated with a callback, the following QAPI can be invoked to release the memory. qapi_atfwd_release_byte_pool(at_fwd_cal lback)	Custom AT command registration is functional only if the corresponding user space callback and byte pool pointers are associated. Byte_pool_qcli needs to be initialized as per the steps mentioned to allocate byte pool pointer in the slide Solution for QTI QAPIs.

LwM2M	Step 1: qapi_Net_LWM2M_Register_App(&app_lwm 2m_handle);
	
	Step 2: qapi_Net_LWM2M_Pass_Pool_Ptr(app_lwm2 m_handle, byte_pool_qcli);
		LwM2M client message communication with the application is only possible after the associated handle is mapped to the byte pool.
		
		Byte_pool_qcli needs to be initialized as per the steps mentioned to allocate byte pool pointer in the slide Solution for QTI

	Step 1: qapi_Net_LWM2M_Register_App_Extended(&app_lwm2 m_handle,user_data, app_callback);
	
	Step 2: qapi_Net_LWM2M_Pass_Pool_Ptr(app_lwm2 m_handle, byte_pool_qcli);
		LwM2M client message communication with the application is only possible after the associated handle is mapped to the byte pool.
		
		Byte_pool_qcli needs to be initialized as per the steps mentioned to allocate byte pool pointer in the slide Solution for QTI

Note: Existing QAPIs, qapi_Net_LWM2M_Encode_App_Payload() and qapi_Net_LWM2M_Decode_App_Payload()are deprecated and new QAPIs will be introduced in future releases.

5.5 Sample Application Code

Module	**File**	**Function**
DSS	apps_proc\dataservices\demo\QCLI_DAM_dem o\src\net\dss_netapp_module.c	dss_get_call_handle()
MQTT	apps_proc\dataservices\demo\QCLI_DAM_dem o\src\net\mqtt_cli_app.c	mqtt_cli_connect()

HTTPc	apps_proc\dataservices\demo\QCLI_DAM_demo\src\net\httpc_module.c	httpc_command_handler()
ATFWD	apps_proc\dataservices\demo\QCLI_DAM_demo\src\atfwd\atfwd_cli_app_module.c	Atfwd()– To map the bytepool pointer at_fwd_callback()– To unmap the bytepool pointer

LwM2M	apps_proc\dataservices\demo\QCLI_DAM_demo\src\lwm2m\d evicecap_app.c
	
	apps_proc\dataservices\demo\QCLI_DAM_demo\src\lwm2m\s w_mgnt_app.c
		devicecap_init()
		
		software_mngt_init()

6 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
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 scripts	Must be listed in module OEM document release to device vendors
USB network driver combo		QTI	Windows host USB drivers for QTI composite	Module OEM responsibility

			devices	
LLVM	4.0.11	QTI	Support compiler	Llvm is a QTI license tool that cannot be released by OEMs to device vendors. Module OEM responsibility to compiler

7 DAM module build process

MDM9205 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 MDM9205 TX platform image has been completely built.

7.1 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.

7.2 Compile txm_demo module

Run the following command:

```
build_llvm.cmd (windows)
build_llvm.sh (linux)
```

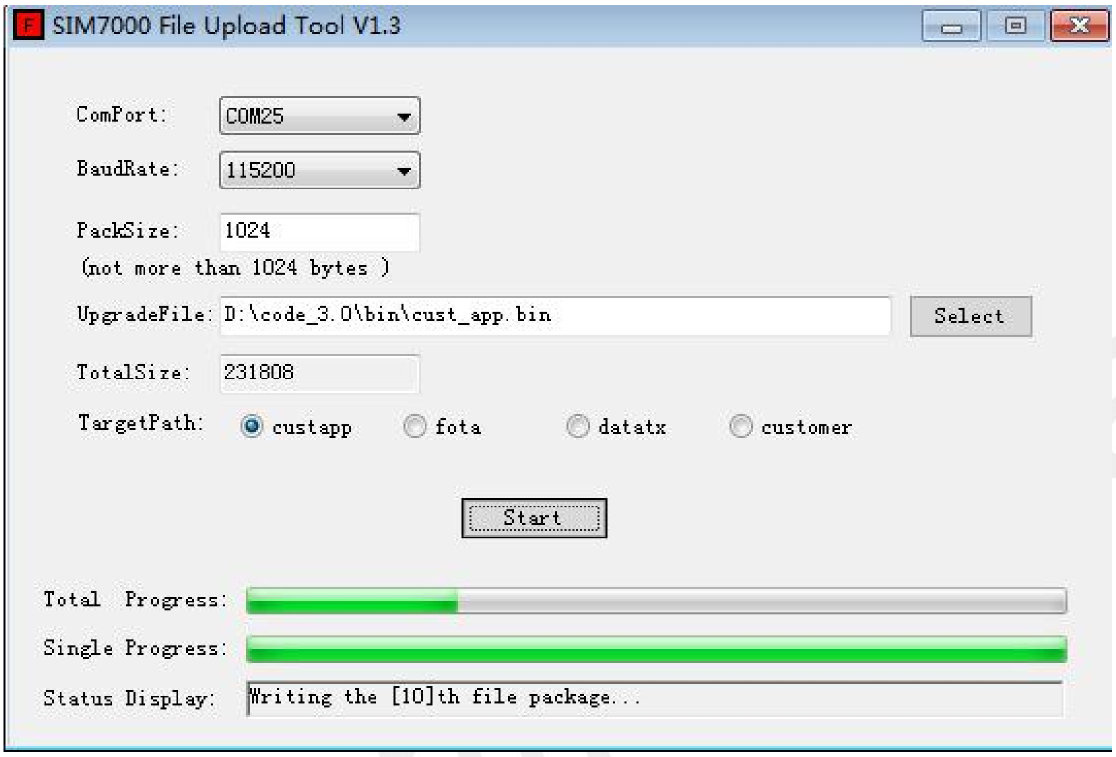
After execution, the ELF and binary files are placed at the following locations:

- 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.

8 Image load process

Before loading DAM application, MDM9205 TX platform image must be loaded to a device.
Use SIM7000 File Upload Tool import cust_app.bin .Reference below screenshots
Note: ComPort is at port


SIM7000 File Upload Tool V1.3

ComPort: COM25
BaudRate: 115200
PackSize: 1024
(not more than 1024 bytes)
UpgradeFile: D:\code_3.0\bin\cust_app.bin
TotalSize: 231808
TargetPath: [✓ custapp] fota datatx customer
Total Progress:
Single Progress:
Status Display: Writing the [10]th file package...

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.