TRUSTONIC

FastCall Hooking Interface

<t-base

PREFACE

VERSION HISTORY

Version	Date	Status	Modification
0.1	17 May 2013	Draft	First version
1.0	12 June 2013	Issued	Doc reviewed and bootloader samples added

TABLE OF CONTENTS

1 Introduction to the FastCall Interface

The SMC instruction [ARM11] in conjunction with a negative (< 0) value in the r0 register is interpreted by the secure monitor as a FastCall. A FastCall accesses a function in <t-base context without performing a complete context switch. FastCalls are always executed in Monitor Privileged Mode [ARM11]. In this regard they shall execute as little code as possible and shall be carefully designed.

FastCall Parameters (Input)

```
r0          FastCallID (always < 0)
r1 - r3     FC parameters depending on the FastCallID
```

Return Values (Output)

```
r0          FastCallID (always < 0) of the FC which has been executed
            (r0 from input data)
r1 - r3     status information / data (depending on FC)
            (r1 returns status "0" or error)
```

<t-base already supports either generic or platform-specific internal FastCalls. They are mainly used for <t-base initialization and common operations that have to be done by the normal world but can only be performed in secure.

FastCalls have the following limitations:

- They cannot call any TlApi or DrApi functions
- They may be executed concurrently on several CPUs
- They must not cause any exception. There is no means to recover in case an exception is triggered in a FastCall.

2 ADDING CUSTOM FASTCALLS

2.1 THE FIRMWARE DRIVER

<t-base allows a unique driver, known as the Firmware Driver, to register an additional FastCall handler that will be called for FastCall IDs that are not known to <t-base. “Firmware Driver” is a naming convention for the first driver to ever install a FastCall handler during <t-base runtime. It is intended to act as a system integration means, compared to other hardware peripherals’ drivers.

As it must always be available as soon as the FastCall handler has been registered, the Firmware Driver **must** be marked as permanent in the driver flags defined in its Makefile:

```
DRIVER_FLAGS := 1 # 0: no flags; 1: permanent; […]
```

Regarding the information shared between the Firmware driver and the Fastcall handlers: the Fastcall handlers get Firmware driver memory mappings in range of 0-2MB at the time handler is installed. As there is absolutely no synchronization mechanism between the Firmware driver and the FastCall handles once installed, it’s mandatory for the Firmware driver to not unmap any of these mappings.

In addition, if new mappings are made by the Firmware driver after FastCall installation, they cannot be relied upon to be visible in Fastcall hook functions.

2.2 ADDITIONAL SECURE DRIVER APIS

The main header file for the Driver API extension is `fastcall.h`.

```
#include "fastcall.h"
```

2.2.1 Types

2.2.1.1 FastCall Registers

```
typedef word_t *fastcall_registers_t;
```

Depending on the platform, a FastCall handler may have access to at least 4 registers (r0 to r3).

Each time an SMC is sent and catch by the Firmware driver, they are forwarded by <t-base to the Firmware driver through the `fastcall_registers_t` table.

2.2.1.2 FastCall Context

```
struct fcContext {
      /* Size of the context */
      word_t size;

      /*Callback to modify L1 MMU mapping */
      void *(*setL1Entry)(word_t idx, word_t entry);

      /* Number of registers available in FastCalls */
      word_t registers;
};
```

The FastCall context structure, `fcContext_t`, is filled when the Firmware driver is initialized into <t-base and forwarded as parameter to the FastCall hook initialization function.

The `setL1Entry` callback can be used at runtime to map additional memory in FastCall context:

- `idx`: The Index of the section in the table of L1 descriptors (maximum of 12 slots available).
- `entry`: The L1 descriptor that will be used for the mapping.

Value returned by function `setL1Entry` is the virtual address of the mapped area (Null in case of error).

However, these mappings, done in FastCall context will not be visible from the Firmware driver.

Note: This context is shared between FastCalls and all processors.

2.2.2 Specific FastCall Entry Points

2.2.2.1 FastCall Handler Initialization

```
typedef word_t (*fcInitHook)(
      struct fcContext *context
);
```

This entry point is called once when FastCall Hooking is enabled through a call to the `drApiInstallFc` driver API (Firmware driver initialization). It is executed in Secure SVC mode.

This function must **never** cause any exception.

Parameters

- `context`: FastCall context structure.

Returns

It must return 0 if initialization goes OK. Else, with any other value, the fastcalls will not be allowed.

2.2.2.2 FastCall Handler

```
typedef void (*fcEntryHook)(
      fastcall_registers_t *regs,
      struct fcContext *context
);
```

This is the actual FastCall handler. It may be executed concurrently on several CPUs. It is executed in Monitor mode.

This function must **never** cause any exception.

Parameters

- `regs`: Normal World registers' values:
  - On entry, `regs[0]` to `regs[context->registers - 1]` contain input parameters. `regs[0]` is always the FastCall identifier.
  - On exit, `regs[0]` to `regs[3]` store output results.
    - `regs[0]` is always the FastCall identifier.
    - `regs[1]` can be used to store a return value.
    - Other registers **must not** be modified. Result of any modification is unpredictable.

    By convention, if the FastCall identifier is unknown the value `MC_FC_RET_ERR_INVALID` should be returned in r1.
- `context`: FastCall context structure.

2.2.3 Specific Firmware Driver APIs

2.2.3.1 drApiInstallFc

```
_DRAPI_EXTERN_C drApiResult_t drApiInstallFc(
      void *entryTable)
```

Install the custom FastCall handler.

Parameters:

- `entryTable`: table of function pointers to FastCall Hooking entry points (see section 2.3, "Firmware Driver Structure").
  - `entryTable[0]` should point to a function of type `fcInitHook`
  - `entryTable[1]` should point to a function of type `fcEntryHook`

Returns:

- `DRAPI_OK` if the FastCall handler has been correctly set
- `E_DRAPI_NOT_PERMITTED` if this function is called from a non-driver context
- `E_DRAPI_INVALID_PARAMETER` if `entryTable` does not point to code in the driver
- `E_DRAPI_CANNOT_INIT` if the driver has not been configured as permanent
- `DRAPI_ERROR_CREATE(E_DRAPI_CANNOT_INIT, E_MAPPED)` if another FastCall handler has already been installed

2.3 FIRMWARE DRIVER STRUCTURE

2.3.1 FastCall Hook Initialization

Initialization of FastCall handling is done by the Firmware driver, it boils down to building a 2-entries table where:

- the first element is a pointer to the FastCall initialization function,
- the second one is a pointer to the actual FastCall handler.

```
void *entryVector[2] = { &fcInit, &_fcMain };

_DRAPI_ENTRY void drMain(
    const addr_t    dciBuffer,
    const uint32_t  dciBufferLen
){
      drApiResult_t ret = drApiInstallFc(entryVector);

      /* Start IPC handler */
      drIpchInit(dciBuffer, dciBufferLen);
}
```

2.3.2 Assembly Glue for FastCall Handler

The FastCall handler, once installed, will be executed in the context of the Monitor and eventually on any CPUs available. While the main Monitor might have a decent stack, it might not be the case for the secondary monitors. For this reason, it's required to setup a new stack before entering in the FastCall Handler

The following piece of assembly code is an example on how to reserve specific stacks for each CPU on which the FastCall handler may run and use the correct one when it is executed (`_fcMain` is the function installed, it then call real FastCall Handler, `fcMain`):

```
      export _fcMain
      import fcMain

CORES_MAX   equ  4
STACK_SIZE  equ  256 * 4

      area stack, noinit, readwrite
fcStackBottom
      space CORES_MAX * STACK_SIZE
fcStackTop

      area text, code, readonly
      preserve8
      arm
```

```
_fcMain
     push {r0, r6-r8, r12, lr}
     mov r12, sp

     ; get affinity level 0 core number in r1
     mrc    p15, 0, r7, c0, c0, 5     ; get mpidr
     ubfx   r6, r7, #0, #4            ; cpu id(1-3)

     ; set own core-specific stack
     ldr    r7, =fcStackTop
     mov    r8, #STACK_SIZE
     ; calculate stack-start for this core
     mul    r6, r6, r8
     ; by sp = top - (core_num * size_core)
     sub    r7, r7, r6
     mov    sp, r7

     ; save the old stack in the new stack
     push {r12}
     blx fcMain
     ; get the old stack back
     pop {r12}
     mov sp, r12
     ; restore the context from the old stack
     pop {r0, r6-r8, r12, lr}
     bx lr

     end
```

2.3.3 FastCall Handler Example

The FastCall Handler is divided in two steps:

- The initialization function, called by <t-base when the Firmware driver install the FastCall hook:

```
word_t fcInit(fcContext_t *context)
{
    /* Initialization code here…
     * Runs in Kernel context */

    /* optionally map things… */
    //context->setL1Entry(0, L1_TYPE_FAULT);

    return 0;
}
```

- The FastCall Handler, called each time an SMC is not recognized by <t-base Monitor:

```
void fcMain(
    fastcall_registers_t *regs,
    fcContext_t *context)
{
    uint32_t mpidr;
    uint32_t fastCallID = regs[0];

    switch(fastCallID){
        case FASTCALL_SOMETHING:
            doFCSomething(regs);
            break;
        default:
            regs[1] = MC_FC_RET_ERR_INVALID;
    }
}
```

3 u-boot Integration Sample

This section describes what could be the integration of the Firmware driver in u-boot, and directly load it into <t-base.

The advantage would be to have the FastCall hook available for u-boot and for the very early boot sequence of the Normal World OS.

3.1 ENTRY POINT IN U-BOOT

The proposed entry point in the bootloader code:

```
ulong mobi_drv_addr = 0x0;
size_t mobi_drv_size = 0x0;

/* Initialize the MobiCore runtime */
if(mci_setup())
    return CMD_RET_USAGE;

if (mc_load_driver(mobi_drv_addr, mobi_drv_size, tci, TCI_SIZE))
    printf("MobiCore Driver loading failed!\n");

// Unmap the MobiCore runtime - otherwise Linux daemon will fail
mci_unmap();
```

It is up to the bootloader developer to specify:

- `mobi_drv_address` – the address in memory where the secure driver blob has been loaded to memory
- `mobi_drv_size` – the size of the blob loaded to memory
- `tci` – a global buffer already allocated to send commands to the driver
- `TCI_SIZE` – the size of the tci buffer previously allocated

NOTE: This guide assumes the bootloader developer has a mechanism to load the secure driver from permanent storage to memory AND has done so before integrating.

NOTE: It is assumed that if TCI is required the bootloader will not release that particular memory for other uses

3.2 <T-BASE COMMUNICATION SETUP

The function `mci_setup()` handles the setup of the communication mechanism between the bootloader and <t-base

The code provided in our example patch is assumed complete and working:

```
static int mci_setup(void)
{
      struct fc_generic fc_init;
      int i;

      uint32_t mci_offset = ((uint32_t)mci) & PAGE_MASK;
      fc_init.cmd = MC_FC_INIT;
      // Set MCI as uncached
      fc_init.param[0] = (uint32_t)mci | 0x1U;
      fc_init.param[1] = (mci_offset << 16 ) | NQ_LENGTH;
      // mcp_offset = 0x118 mcp_length=0x90
      fc_init.param[2] = ((NQ_LENGTH + mci_offset) << 16) | MCP_LENGTH;
      _smc(&fc_init);
      printf("MobiCore INIT response = %x\n", fc_init.param[0]);
      if(fc_init.param[0]) {
            printf("MobiCore MCI init failed!\n");
            return -1;
      }

      // MCI is setup, do a nsiq to give RTM a chance to run
      for(i = 0; i < MC_MAX_SIQ; i++)  {
            uint32_t state, ext_info;
            mc_nsiq();
            mc_info(0, &state, &ext_info);
            // Check if initialized
            if(state == MC_STATUS_INITIALIZED) {
                  printf("MobiCore RTM has initialized!\n");
                  break;
            }
      }
      if (i == MC_MAX_SIQ) {
            printf("MobiCore RTM failed to initialize\n");
            return -1;
      }
      mcp = (mcpBuffer_ptr)((uint8_t*)mci + mci_offset + NQ_LENGTH);
      nq = (uint8_t*)mci + mci_offset;
      printf("MobiCore IDLE flag = %x\n", mcp->mcFlags.schedule);

      return 0;
}
```

The code assumes the buffer MCI is already allocated in memory and has a size of MCI_SIZE:

```
#define MCI_SIZE 512
uint8_t mci[MCI_SIZE];
```

NOTE: This buffer must be globally defined as it is used for communication throughout the code!

NOTE: This code execute several smc calls to give <t-base time to initialize. It does also have a maximum number of calls(MC_MAX_SIQ) defined so if something goes wrong it can handle errors correctly!

3.3 FIRMWARE DRIVER LOADING

The function for driver loading is assumed complete and working.

The return value of the function is 0 for success – driver has been loaded and has initialized correctly.

```
static int mc_load_driver(void *buf, size_t size, void *tci, size_t tci_size)
{
	int ret = 0;
	int i;
	// now we have the driver in memory, setup the MCP
	mclfHeaderV2_ptr header = buf;
	printf("MobiCore driver address %x, size = %u!\n", buf, size);
	mcp->mcpMessage.cmdOpen.cmdHeader.cmdId = MC_MCP_CMD_OPEN_SESSION;
	mcp->mcpMessage.cmdOpen.uuid = header->uuid;
	mcp->mcpMessage.cmdOpen.wsmTypeTci = WSM_CONTIGUOUS | WSM_WSM_UNCACHED;
	mcp->mcpMessage.cmdOpen.adrTciBuffer = ((uint32_t)tci) & ~(PAGE_MASK);
	mcp->mcpMessage.cmdOpen.ofsTciBuffer = ((uint32_t)tci) & PAGE_MASK;
	mcp->mcpMessage.cmdOpen.lenTciBuffer = tci_size;

	// check if load data is provided
	mcp->mcpMessage.cmdOpen.wsmTypeLoadData=WSM_CONTIGUOUS|
WSM_WSM_UNCACHED;
	mcp->mcpMessage.cmdOpen.adrLoadData = ((uint32_t)buf) & ~(PAGE_MASK);
	mcp->mcpMessage.cmdOpen.ofsLoadData = ((uint32_t)buf) & PAGE_MASK;
	mcp->mcpMessage.cmdOpen.lenLoadData = size;
	memcpy(&mcp->mcpMessage.cmdOpen.tlHeader,header, sizeof(mclfHeader_t));

	put_notification(0);
	for (i = 0; i < MC_MAX_SIQ; i++) {
		mc_nsiq();
		udelay(2000);
		if(get_notification() == 0) {
			printf("MobiCore RTM Notified back!\n");
			break;
		}
	}
	if (i == MC_MAX_SIQ) {
		printf("MobiCore RTM did not ack the open command!\n");
```

```
            ret = -1;
      }

      for (i = 0; i < MC_MAX_SIQ || mcp->mcFlags.schedule; i++) {
            mc_nsiq();
            break;
      }
      if (i == MC_MAX_SIQ) {
            printf("MobiCore   is   not   yet   IDLE   -   driver   is   probably
missbehaving!\n");
            return -1;
      }
      printf("MobiCore Driver loaded and RTM IDLE!\n");
      return ret;
}
```

Parameters

- `buf` – the address in memory where the secure driver blob has been loaded to memory
- `size` – the size of the blob loaded to memory
- `tci` – a global buffer already allocated to send commands to the driver
- `tci_size` – the size of the tci buffer previously allocated

NOTE: As stated before please note that if the secure driver will use the TCI after initialization you MUST ensure the memory allocated for TCI is not released to Android!

NOTE: Please remember the driver must not have long initialization times or endless loops. The code above has checks that the driver will not take too long to initialize but since there might not be any time source interrupts in the bootloader <t-base might not relinquish control to bootloader at all there is nothing the code can do!

NOTE: It is always assumed the driver developer understands the system!

3.4 DATA DEALLOCATION

Data deallocation is very important in the bootflow. Without proper deallocation subsequent calls to <t-base from the Android daemon will fail.

The code is considered complete and working:

```
static int mci_unmap(void)
{
      int i;
      mcp->mcpMessage.cmdHeader.cmdId = MC_MCP_CMD_CLOSE_MCP;
      put_notification(0);

      mc_nsiq();

      for(i = 0; i < MC_MAX_SIQ; i++)  {
            uint32_t state, ext_info;
            mc_info(0, &state, &ext_info);
            mc_nsiq();
            // Check if initialized
            if(state != MC_STATUS_INITIALIZED) {
                  printf("MobiCore RTM has been uninitialized!\n");
                  break;
            }
            mc_nsiq();
      }
      if (i == MC_MAX_SIQ) {
            printf("MobiCore RTM failed to uninitialize\n");
            return -1;
      }
      return 0;
}
```

NOTE: Error handling is important here as any error returned by this code will result in an unusable <t-base system for Android

3.5 <T-BASE FILES NEEDED

Our u-boot example code consists of the following files:

Name	Comment
mcLoadFormat.h	<t-base header with structures of the load format
mcUuid.h	<t-base header with structure about uuid format
mcVersionInfo.h	<t-base header with versioning information
mcimcp.h	<t-base header with structures related to bootloader->t-base communication (MCI format)
mcinq.h	<t-base header with structure related to command queue format needed for communication with <t-base