DICE III Boot

and Flash Memory Initialization

Revision 1.0.0-41559

June 27, 2015

Contents

1 Overview

The DICE III has a primary (on-chip) boot ROM from which the ARM will execute code after a reset. This is also referred to as 'ROM BOOT' in this document. This ROM has code to load programs into internal memory from an external serial flash memory device.

The TCAT DICE III Firmware SDK provides a template for secondary boot programs (BOOT2) that are loaded from flash address zero by the on-chip boot loader.

On-chip ROM BOOT Features

- Load from industry standard SPI Flash using either 0x03 or 0x0b.
- Load a variable size image
- Support reprogramming of the DICE PLL for faster load
- Basic image integrity checks
- If the boot image is invalid, default to UART loader

DICE III firmware programs are generally separated into 'Secondary Boot Loaders' and 'Applications.' A secondary boot loader is a minimal program which validates an application image before loading it to RAM and transferring control to it. Applications are fully functional implementations of end-product (or factory test) firmware.

DICE III programs are copied from serial flash memory to RAM and executed in RAM.

2 SPI Operation

The ROM BOOT loader is designed to support as many different SPI devices as possible and to support various clock speeds as well.

2.1 Operating Speed

The DICE III starts up running from an external crystal which is nominally 12MHz max 20MHz. The maximum SPI speed supported is one-eighth of this frequency when the PLL is not programmed.

The boot loader will always read the header from the flash at this speed using opcode 0x03. The header contains information which will optionally program the DICE III PLL and select a specific SPI divider. This allows the image to configure the remaining loads for higher speeds.

2.2 Memory Use

The boot code uses a small part of the 320k internal memory. The upper 1k is reserved for use so it is not possible to load an application which is larger than 319k. As most applications will use a large amount of .bss this should never pose a problem.

2.3 SPI Format

The ROM BOOT loader uses the following SPI protocol parameters:

Word size	8 bits
CPOL	0
CPHA	0
Order	MSB First

2.4 Header Format

The first 16 bytes in flash memory contain the header. The header is read using opcode 0x03.

Table 1, SPI Header format

Byte Offset	Value	Comment
0	'T'	Start Identifier
1	'C'	Start Identifier
2	'A'	Start Identifier
3	'T'	Start Identifier
4	0x03	Dice 3 Family
5	FLAGS	Flags for enabling PLL etc. (see below)
6	PLL_MUL	Multiplier for PLL (See the 'CPU PLL Control Register' definition in *DICE_III_User_Guide_SysCtrl)*
7	PLL_DIV	Pre-divider for PLL (See the 'CPU PLL Control Register' definition in *DICE_III_User_Guide_SysCtrl)*
8	LADDR0	Address to load to (7:0)
9	LADDR1	Address to load to (15:8)
10	LADDR2	Address to load to (23:16)
11	LADDR3	Address to load to (31:24)
12	NBWORD0	Number of 32 bit words (7:0)
13	NBWORD1	Number of 32 bit words (15:8)
14	NBWORD2	Number of 32 bit words (23:16)
15	SPI_BAUDR	Divider for SPI Baud rate (See the 'SPI Baud Rate Select' register definition in *DICE_III_User_Guide_SPI*)

The FLAGS byte is defined as follows in the table below.

Table 2, Header Flags

Bit	Field	Comment
0:3	CPU_CLK_N	Divider for CPU clock (See 'CPU Clock Control Register' definition in *DICE_III_User_Guide_SysCTRL*)
4	USE_PLL	If set the PLL will be programmed
5	CHKSUM	The image is protected by a checksum
6	USE_FAST_READ	If set use Fast Read opcode 0x0b
7	0	Reserved

2.5 Payload Format

The payload follows after the header. The size of the payload in 32 bit words is as defined by NBWORD in the header, this includes the checksum.

The crc32 is calculated over the entire payload and the result must be 0x00000000. This is typically accomplished by putting the crc32 value at the end of the payload and increasing the payload by one DWORD.

The crc16 is only computed and checked if the CHKSUM flag is set.

The polynomial used is:

X^32+X^26+X^23+X^22+X^16+X^12+X^11+X^10+X^8+X^7+X^5+X^4+X^2+X^1+X^0

It is calculated on each byte as they are ordered in the payload. The bit order is MSB first.

The DWORDS in the payload are placed in accordance with the endianness of the ARM which is 'little-endian'. This means that the first byte is the least significant. The CRC word is placed that way as well.

2.6 SPI Load Procedure

The boot sector will follow these steps

1. Remap ROM so RAM resides at 0
2. Boot sector reserves the top 1k of internal ram
3. Program SPI0 to comply with format as specified, clk=$F_{crystal}/8$
4. Issue opcode 0x03 and read 16 byte header to scratch area in RAM
5. Check header, if invalid GOTO **hdr_err**
6. If USE_PLL flag is not set GOTO **skip_pll**
7. Program PLL according to header

8. Wait 500us for PLL to lock
9. Switch CPU to run from PLL
10. **skip_pll:**
11. Reprogram SPI0 to SPI_BAUDR
12. Load the payload into RAM from LADDR, Compute checksum while loading
13. If not CHKSUM flag GOTO **no_crc**
14. If crc!=0 GOTO **crc_err**
15. **no_crc:**
16. Load PC= LADDR (execute code, never return)
17. **crc_err:**
18. **hdr_err:**
19. return

3 UART Operation

The primary boot code will first initiate SPI operation and attempt to find a valid secondary boot (or Application) image at the start of flash memory. If SPI boot fails with a header or CRC error, the UART operation will be entered. This assumes a 12MHz clock crystal is driving the on-chip oscillator.

3.1 UART Load Procedure

Before starting, the CPU is switched back to running from the 12MHz crystal. After that the UART is programmed to run 115,200 baud, 8 N 1, with no hardware flow control.

The editor is built to work with single CR or LF or a combination. It will always send CR/LF pairs.

The prompt character used is '$' in order to distinguish it from the firmware Command-Line Interpreter (CLI), which uses a '>' prompt character. This will not be shown if the loader finds a valid boot image in flash memory.

When the loader is entered it shows the following splash screen:

```
TCAT-BOOT
Reason  : Header
Version : 1.0.0.2477
Device  : TCD30<b><r>
$
```

<b> represents a single-digit chip bond-type (integer number)

<r> represents a single-digit revision (integer number)

The available commands are summarized in the table below.

Table 3, UART Loader Commands

Command	Description
S	Show the splash screen
L <address>	Start an X-Modem load to <address>
R <address>	Execute code from <address>

3.2 Parameter and white-space format

The <address> parameter must be a hex value starting with '0x' and any number of hex characters.

Only the space (' ') character is allowed for white-space.

3.3 X-Modem

The X-Modem implementation supports the X-Modem-1K transfer protocol with crc16. There is no support for simple checksum as 'C' is always used for negative acknowledge and not NAK. Both 128 byte and 1k packets are supported through use of SOH or STX.

Once a transfer is initiated from the DICE III with the 'L' command, there will be approximately a 160 second timeout period allowing the serial terminal program to be configured for the X-Modem transfer.

A failed load due to timeout:

```
$L 0x0
CCCCCCCCCCCCCCCCCCCCCerror: Load Error #01 - Size = 0x00000000
$
```

A failed load due to interrupted transfer:

```
$L 0x0
Cerror: Load Error #03 - Size = 0x00011f80
$
```

Successful X-Modem application load:

```
$L 0x0
CLoad Completed - Size = 0x00023b80
$R 0x0
```

3.4 Warm Boot

In some applications, and when debugging, it is often necessary to be able to reset the application without reloading the code from SPI or UART. If the boot code detects that it is entered due to a software reset or a watchdog reset, it will check the reserved vector at 0x00000014 in RAM. If that vector is set to 0x20120710 it will immediately call address 0x00000000 in RAM.

4 Secondary Boot

While it is possible to have the on-chip ROM loader start an application at flash memory address zero, it is often desirable to program a secondary boot loader to flash address zero. The secondary boot loader can then load the application.

There are several advantages having a secondary boot loader:

- Ability to handle Golden and Upgrade images so if field loading of the application fails the device will simply revert to the golden image.
- Ability to customize the secondary loader to the actual target, which could include configuring GPIO pins, etc.
- Ability to load complex images to external SDRAM

The DICE III SDK contains a template secondary loader project called tcd3xxx_boot2. This secondary boot handles golden and upgrade images. It will run the upgrade if valid and if not it will load the golden image. If none of the images are valid it will fault back to the ROM UART loader described above.

4.1 Defining sections in the external Flash

The tcd3xxx_boot2 template assumes a certain layout of the serial flash. It must match the layout defined by the Application in the myAppFlash.cpp implementation file in order for the application supplied load options, such as FSS DCP and FSS CLI, to work properly.

These definitions in boot2.c specify the size of the secondary boot and the two images:

```
#define APPSIZE_MAX (5*64*1024)

#define GOLD_START  (1*64*1024)
#define CURR_START  (GOLD_START + APPSIZE_MAX)
```

In the template by default, the secondary boot uses the first 64k sector, which is followed by the 320k Golden image which is then followed by the 320k Upgrade image. The corresponding FSS definition in myAppFlash.cpp looks like this:

```
static const FSS_SEGMENT myFSSSegments[] =
{
	sFSS_SEGMENT
	(	// the boot sector - using boot2 which is 64k
		/*flags      */ eFSS_FLAG_BOOT | eFSS_FLAG_PROT,
		/*flashBase  */ 0x00000000,
		/*flashSize  */ 64*1024,
		/*lockFn     */ NULL,
		/*name       */ "boot"
	),
	sFSS_SEGMENT
	(	// the golden application image
		/*flags      */ eFSS_FLAG_APP_GOLD | eFSS_FLAG_PROT,
		/*flashBase  */ 64*1024,
		/*flashSize  */ 320*1024,
		/*lockFn     */ NULL,
		/*name       */ "App_Gold"
```

```
        ),
        sFSS_SEGMENT
        (       // the upgrade application image
                /*flags         */ eFSS_FLAG_APP,
                /*flashBase     */ (64+320)*1024,
                /*flashSize     */ 320*1024,
                /*lockFn        */ NULL,
                /*name          */ "App_Upgrade"
        ),
        // ...
};
```

Other sections may be present, for example if the application stores settings that should be recalled when the device is powered-on (eFSS_FLAG_SPS2), if environment variables are used in the application (eFSS_FLAG_ENV), or for any other use the developer may require.

4.2 BOOT2 Application header

The secondary boot loader uses a specific header for the loadable images:

```
typedef struct
{
        uint32 magic;
        uint32 size;
        uint64 version;            // application version
        uint32 vendorOUI;          // vendor OUI
        uint32 productID;          // product ID
        uint32 custom;             // for any vendor use
        uint8  reserved[16];       // reserved -- set to zero
} BOOT2_HEAD;
```

This header contains information about the product type and the version of the application. This enables firmware loader applications to determine if the image is compatible with the product simply by comparing the vendorOUI and productID to the one currently running. When loading from a host using DCP it is possible to enquire about the vendorID and productID of the running application and use that information to validate the new image before loading it to flash. The version field also allows loader applications to prevent downgrading an application.

Field	size	comment
magic	uint32	must be 0x20120710
size	uint32	size in 32 bit words
version	uint64	TCAT version type
vendorOUI	uint32	Vendor OUI (low 3 bytes)
productID	uint32	product ID
custom	uint32	vendor specific

The CRC works the same way as for the ROM BOOT so the CRC over the payload not including the header must result in a value of 0.

Images with this type of boot header will have the file extension of 'tcd' by convention.

4.3 Secondary Boot memory usage

The secondary boot loader is linked to be loaded to RAM at address 0x4f000 so the upper 4096 bytes of the RAM are reserved. This means that the largest application that can be loaded is 316KB, but as most applications have .bss at the end this is not a practical problem.

4.4 Complete boot sequence

1) ROM BOOT will load the tcd3xxx_boot2 application to 0x4f000 as specified in its linker definition file (tcd3xxx_boot2.ld), and because the boot application is less than 3KB in size it will not overwrite the memory used by ROM BOOT (top 1KB).
2) ROM BOOT will execute the tcd3xxx_boot2 application.
3) The secondary boot will find a valid image, first checking the Upgrade image and then the Golden image. If neither of them is valid it will invoke the ROM BOOT UART loader.
4) The valid image is loaded into RAM from address 0
5) The newly loaded image is executed from address 0

Note that the secondary boot does not change the PLL parameters so it is simply taking over whatever the ROM BOOT programmed based on the header.

5 Preparing a bootable image

There are two kinds of bootable images which are created by the SDK: those which are bootable by the on-chip ROM boot loader (tcd file extension) and those which are bootable by a secondary loader (tca file extension). The relevant distinction between the two is the header which is prepended to the binary executable at the end of the build process.

- A tca file has the BOOT2 header described above. This is generally a full firmware application, factory test program, etc. and is loaded and executed by a secondary loader. The header of a tca file has the BOOT2 format as described in section 4.1 above.
- A tcd file has a 16-byte header containing the information in the table in section *2.4*. This is usually a secondary boot loader, but may also be a full application, both of which are programmed to the first flash segment.

The Linux command line application d3hdr can generate these file headers as necessary and append them to the raw binary build output file. This utility is located in the utils/linux/ directory of your subversion tag. The build process will do this for you, however, so it is not necessary to run the d3hdr program yourself.

The project template **tcd3xxx_boot2**, which is used to create secondary boot loaders, has a step in its makefile which creates a tcd3xxx_boot2.tcd file using the raw binary output file. This type of project does not create a tca file since it is always programmed at flash address zero and will always need a tcd header.

In addition to creating a <application _name>.tca file at the end of the build process, the Application project template makefiles include a step to create <application _name>.tcd files, which can be programmed to the first sector in flash memory (and therefore be loaded by the on-chip ROM loader) in case a secondary boot loader is not used. The tca files should be programmed to flash memory at locations that are similarly defined in both the secondary loader and the application's flash memory layout definition as described in section 4.1 above.

5.1 Creating the <name>.tcd file using d3hdr

d3hdr [options] <srcfile> <dstfile>

Option	Comment
-tb	Generate ROM BOOT header
-cpundiv <num>	Divider for CPU clock, default 1
-usepll	If not defined do not reprogram PLL
-chk	If defined the CRC is checked when loading
-fst	If defined Flash FastRead (0x0b) is used
-pmul <num>	PLL multiplier, default 100
-pdiv <num>	PLL division, default 3
-load <num>	Address in RAM to load and run from
-spid <num>	SPI clock divider, default is 2

Please note that there are restrictions to –pmul and –pdiv which are described in the See the 'CPU PLL Control Register' definition in *DICE_III_User_Guide_SysCtrl*If a 12MHz crystal is used the defaults will make the ARM run 200MHz and the APB bus run 50MHz. As the SPI sits on the APB the default will make the SPI run 25MHz.

To create a loadable .tcd file for tcd3xxx_boot2 the command would be:

```
./d3hdr -tb -usepll -chk -load 0x4f000 tcd3xxx_boot2.bin tcd3xxx_boot2.tcd
```

This command is done as part of the build process in your project's makefile, so you don't have to explicitly run `d3hdr`.

5.2 Creating the <name>.tca file using d3hdr

d3hdr [options] <srcfile> <dstfile>

Option	Comment
-ta	Generate tcd3xxx_boot2 header
-oui <num>	24 bit vendor ID (eg. 0x000166)
-pid	Specify product ID
-ver <maj>.<min>.<sub>.<bld>	Specify version
-bs	Specify build by build server
-beta	specify beta version
-cust <num>	specify customer specific uint32

As mentioned before, the <filename>.tca files are normally created by the template makefile's. A Python script will inspect the myProductDefs.h file and extract the version and product information to create the header.

6 Programming an uninitialized device using BOOT2

All of the firmware application template projects in the DICE III SDK provide the ability to program the flash through the CLI (i.e. using X-Modem via serial terminal) and in most cases also using a communication protocol (DCP) from a host (USB and FireWire currently). In this document, the CLI method is described. In this example, a secondary boot loader is programmed to flash address zero, and additional Application image(s) are loaded to other areas in flash memory. The secondary loader will look for, validate and run the appropriate Application image.

6.1 Prerequisites

Build the application that needs to be loaded, and build the tcd3xxx_boot2 project. The files <filename>.tca and tcd3xxx_boot2.tcd are the ones you will need. You can find them in the build/app/bin directory in each of the projects.

You will also need to have a terminal program hooked up to UART0 running 115,200 8N1. Your terminal program must support X-Modem 1k.

Make sure your Application project and boot2 project are created with matching flash layouts.

6.2 Loading

From Eclipse run the application in the debugger. You will now see the splash screen and a CLI prompt.

You can use the `fss.list` CLI command to see your flash sections:

```
>fss.list

============================================================
FSS Structure:
Flash Base:       0x00000000
Flash Size:       0x000e0000
Flash Segments:   5
Flash Prg. Time:  2 ms/64kB
Flash Ers. Time:  1000 ms/64kB
============================================================
Segment[0]: boot
Base Address: 0x00000000, Size: 0x00010000 Good
Flags: 0x00000009
Segment[1]: App_Gold
Base Address: 0x00010000, Size: 0x00050000 Good
Flags: 0x0000000c
Segment[2]: App_Upgrade
Base Address: 0x00060000, Size: 0x00050000 Good
Flags: 0x00000002
Segment[3]: App_Env
Base Address: 0x000b0000, Size: 0x00010000 Good
Flags: 0x00000040
Segment[4]: App_Settings
Base Address: 0x000c0000, Size: 0x00020000 Good
Flags: 0x00000030
============================================================
>
```

6.2.1 Loading the secondary boot

At the prompt write:

```
>fss.load boot yes yes
Erasing flash...
start XModem-1k transfer ...
C
Loaded 2084 bytes
```

After the flash is erased you will then see one or more ‘C’ characters indicating that the firmware is ready to receive new image data. In the terminal program start an X-Modem 1k upload of the tcd3xxx_boot2.tcd file.

6.2.2 Loading an Application

Now the boot sector is in place but we don’t have an application image yet. To load the application, write:

```
>fss.load App_Gold yes yes
Erasing flash...
start XModem-1k transfer ...
CC
Loaded 141536 bytes
>
```

After you see the first ‘C’ character, start an X-Modem 1k upload of the <appname>.tca file. At this point the system can boot from power up.

```
>reset
**********************************************************
* Secondary Boot - ver 0.9.0                             *
* Application Invalid, found Golden - execute            *
**********************************************************

**********************************************************
* TC Command Line Interface System                       *
* Copyright 2004-2015 TC Applied Technologies            *
**********************************************************
* Organization    : TC Applied Technologies              *
* Product         : TCD3000 EVM001                       *
* Serial no.      : 124                                  *
* SDK Version     : 1.0.2.0 local beta                   *
* Application Ver.: 1.0.0.0                              *
*                   - built on 14:54:43, Jun 15 2015     *
**********************************************************
```

Note that since there is not App_Upgrade application programmed yet, the secondary loader did not find an App_Upgrade image, and loaded the App_Gold image. You can program the same application to App_Upgrade to verify that you’ve configured the segment properly in firmware. In that case, the splash screen should show the following:

```
**********************************************************
* Secondary Boot - ver 0.9.0                             *
* Application found - execute                            *
**********************************************************
```

Future loads should be done to the App_Upgrade sector. This will prevent the Golden image from being destroyed and the system will always have something to fall back to.

7 Programming an uninitialized device using ROM BOOT only

All of the firmware application template projects in the DICE III SDK provide the ability to program the flash through the CLI (i.e. using X-Modem via serial terminal) and in most cases also using a communication protocol (DCP) from a host (USB and FireWire currently). In this document, the CLI method is described. In this example, the application is loaded to flash address zero, and a secondary loader is not used.

7.1 Prerequisites

Build the application that needs to be loaded. The file <appname>.tca is the one you will need. You can find it in the build/app/bin directory in the project directory.

You will also need to have a terminal program hooked up to UART0 running 115,200 8N1. Your terminal program must support X-Modem 1k.

As the templates are created for loading by a secondary loader (BOOT2) you will need to modify your myAppFlash.cpp to indicate that the application is at the beginning of the Flash.

```
static const FSS_SEGMENT myFSSSegments[] =
{
        sFSS_SEGMENT
        (       // the golden application image
                /*flags       */ eFSS_FLAG_APP_GOLD | eFSS_FLAG_PROT,
                /*flashBase   */ 0,
                /*flashSize   */ 320*1024,
                /*lockFn      */ NULL,
                /*name        */ "Application"
        ),
        sFSS_SEGMENT
        (       // the environment variable sector, used for serial number by the
base class
                /*flags       */ eFSS_FLAG_ENV,
                /*flashBase   */ 320*1024,
                /*flashSize   */ 64*1024,
                /*lockFn      */ NULL,
                /*name        */ "App_Env"
        ),
        sFSS_SEGMENT
        (       // the user storage area
                /*flags       */ eFSS_FLAG_SPS2 | eFSS_FLAG_LOCK,
                /*flashBase   */ (64+320)*1024,
                /*flashSize   */ 128*1024,
                /*lockFn      */ NULL,
                /*name        */ "App_Settings"
        )
};
```

Note that only the first segment is relevant to this example. The other sections are typical of a usual firmware application, but are not required. When you run this application, the flash layout will determine where the uploaded data will be placed when using the `fss.load` CLI command.

7.2 Loading the Application

From Eclipse run the application in the debugger. You will now see the splash screen and a CLI prompt. The debugger will be configured to load and run the **tca** image.

You can use the `fss.list` command to see your flash sections:

```
>fss.list

============================================================
FSS Structure:
Flash Base:      0x00000000
Flash Size:      0x000e0000
Flash Segments:  3
Flash Prg. Time: 2 ms/64kB
Flash Ers. Time: 1000 ms/64kB
============================================================
Segment[0]: Application
Base Address: 0x00000000, Size: 0x00050000 Good
Flags: 0x0000000c
Segment[1]: App_Env
Base Address: 0x00050000, Size: 0x00010000 Good
Flags: 0x00000040
Segment[2]: App_Settings
Base Address: 0x00060000, Size: 0x00020000 Good
Flags: 0x00000030
============================================================
>
```

Note that the 'Application' segment is located at address zero.

7.2.1 Loading an Application at the start of flash memory

In this case, a full application is loaded to address zero. This means the application was created from one of the application templates, and not the tcd3xxx_boot2 loader template. The upload image is therefore the **tcd** file that was created by the build process.

At the prompt write:

```
>fss.load Application yes yes
Erasing flash...
start XModem-1k transfer ...
C
Loaded 141504 bytes
>reset
***********************************************************
* TC Command Line Interface System                        *
* Copyright 2004-2015 TC Applied Technologies             *
***********************************************************
* Organization    : TC Applied Technologies               *
* Product         : TCD3000 EVM001                        *
* Serial no.      : 0                                     *
* SDK Version     : 1.0.2.0 local beta                    *
* Application Ver.: 1.0.0.0                               *
*                   - built on 14:33:19, Jun 15 2015      *
***********************************************************
>
```

After the flash is erased you will then see one or more 'C' characters indicating that the firmware is ready to receive new image data. In the terminal program start an X-Modem 1k upload of <appname>.tcd

At this point the system can boot from power up. Because the secondary boot is not used, a failed upload will render the device unable to run and only the ROM BOOT UART interface will be available.

8 Revisions

Table 8.1 Document revision history

Date	Rev.	By	Change
June 27, 2015	1.0.0-41559	BK	Initial draft, based on source doc