Movidius
an Intel company

EFuse Programming

Application Note

v0.4

Intel Movidius
2200 Mission College Blvd
M/S SC11-201
Santa Clara, CA 95054
http://www.movidius.com/

Table of Contents

1 Introduction

While the MA2x5x's GPIO Boot feature allows for a simple approach to configuring the boot process, the set of configurations available are limited to a certain set of boot parameters and a safe set of PLL ranges.

In order to boot with the maximum possible clock frequency (which in turn can decrease boot time) and to have more freedom when choosing interface configuration (such as GPIO assignments or transfer modes), the MA2x5x's One-Time-Programmable (OTP) EFuses can be programmed with a custom boot configuration. Additionally, to make use of the MA2155's Secure Boot feature, the security mode and encryption keys must be stored in the EFuse bits.

The MA2x5x has a total of 1024 EFuses. EFuses 0 to 639 are for internal and ROM usage. Supported fields are documented in the table below.

Start Bit	End Bit	Size (bits)	MA2150	MA2155
0	63	64	BOOTIFCFG	
64	79	16	BOOTPLLCFG	
80	83	4	BOOTIFSEL	
84	86	3	BOOTSYSDIV	
87	94	8	Reserved – set to zero	
95	95	1	PADOVERRIDE	
96	96	1	PADORSLEW	
97	98	2	PADDRIVE	
99	99	1	Reserved – set to zero	
100	115	16	Reserved – set to zero	SECURITY_CODE
116	127	12	Reserved – set to zero	
128	255	128	Reserved – set to zero	AES_KEY
256	511	256	Reserved – set to zero	ECC_PUBKEY
512	639	128	Reserved – Do not modify	
640	1023	384	Reserved for customer use	

Table 1: EFuse Bit Assignments

The EFuses also provide an area for customer use in EFuses 640 to 1023. This area can be used for product serial numbers or other information that does not need to change throughout the lifetime of the product. Use of this area does not require use EFuse Boot if GPIO Boot is preferred.

2 Goals and Scenarios

In this Application Note we will introduce an example application that is provided with the Movidius Development Kit (MDK) to program the EFuse bits. This application can be used as-is or as a starting point for integration into custom applications as part of a manufacturing process.

For some products, all units will require identical EFuse values. For this situation, the application could be modified with the required values and executed on each unit via JTAG or GPIO Boot.

Other products may require different EFuse values between units. Examples of this include:

- Different encryption keys for product variants to ensure that firmware from one variant cannot be executed on a different variant.
- Different USB Product Identifier for product variants.
- Serial number stored in customer area.
- Product configuration or calibration data stored in customer area.

For this use-case, the example EFuse programming application can be extended with a communications channel. As the application is a regular MDK application, any available driver can be used.

The EFuse programming operation may have to take place as part of a larger application that performs product testing or other manufacturing processes. In this case, the EFuse operations can easily be integrated into such an application.

Customers must program the parts as per their requirements after receiving the parts but prior to mounting on boards.

EFuses should be programmed by the customer after receipt of the parts, but prior to mounting on boards. Customer programming requires a socket based system with a chip handler capability. The programming board needs the facility to boot the programming application and to apply the programming voltage as required.

3 EFuse Programming Application

The source code and makefile for the EFuse programming application can be found under the MDK installation directory in `examples/util/eFuseProgrammer`.

The application uses functions from the `DrvEfuse.h` header file. This header provides a low level interface, as well as a higher level interface where some of the implementation details are hidden. In order to simplify the example application, the higher level interface is used.

For each EFuse bit, a value of 0 is the default state. In this state, a bit can be programmed to a value of 1 but it is not possible to reset an EFuse bit back to 0. To overcome this, in addition to the main set of EFuses there are a number of 'redundancy EFuses', which allow for limited reprogramming capability in the case of errors or misconfiguration. Four slots are available, each one specifying an address and an overriding value for the address. For each slot, there is an enable/in-use bit, a disable bit, a value bit and 10 bits for the address.

To facilitate programming of the EFuses, the high level EFuse interface uses a structure containing two bit arrays: a value array and a mask array. This allows programming of a selected subset of the EFuses. The `DrvEfuseProgramWithMask()` function is passed an `efuseMaskSet` structure containing the new values and mask. The current EFuse values are read to avoid reprogramming EFuses that already contain the required value. This also allows `DrvEfuseProgramWithMask()` to see if any EFuses are currently set to a value of 1 but are required to have a value of 0. In this case, if the `DRVEFUSE_FLAG_REDUNDANCY` flag is passed to `DrvEfuseProgramWithMask()` then the function will attempt to use any available redundancy slots. If no slots are available, or the flag has not been passed in, then an error will be returned.

If the `DRVEFUSE_FLAG_REDUNDANCY` flag is used, then any Efuses that fail to be programmed in the first pass with a value of 1 are also overridden using the redundancy mechanism. If the flag is not used, then a failure to program an EFuse to a value of 1 results in an error.

Once all EFuses have been programmed, the function will read back the entire set of EFuses to verify that they have the desired values. This behavior can be disabled by passing the `DRVEFUSE_FLAG_NO_VERIFY` flag.

Prior to calling `DrvEfuseProgramWithMask()`, the `efuseMaskSet` structure must be populated with the required values. Several functions are provided to assist in this:

```
void DrvEfuseClearMaskSet(efuseMaskSet *maskSet);
void DrvEfuseSetValueMask1(efuseMaskSet *maskSet, u32 bitIndex, u32
value);
void DrvEfuseSetValueMask32(efuseMaskSet *maskSet, u32 startBit, u32
endBit, u32 value);
void DrvEfuseSetValueMask64(efuseMaskSet *maskSet, u32 startBit, u32
endBit, u64 value);
void DrvEfuseSetValueMaskPtr(efuseMaskSet *maskSet, u32 startBit, u32
endBit, const u8 *value);
void DrvEfuseSetValueMaskPtrRev(efuseMaskSet *maskSet, u32 startBit,
u32 endBit, const u8 *value);
```

The first function simply clears the structure, readying it for new data. Once this is done, data can be fed in using the `DrvEfuseSetValueMaskXXX()` functions. These functions can be provided with a boolean value, a 32-bit value, a 64-bit value or a pointer to a byte array respectively. As well as copying in the provided value at the required bit offset, the relevant bit range is set to all 1s in the mask.

Special care must be taken with the AES key, as this must be stored in reverse byte order (see the AES_KEY section in the Boot Operation chapter of the MA2x5x databook). The function `DrvEfuseSetValueMaskPtrRev()` is provided to ease programming of the AES key.

Helper functions and macros are provided to ease calculation of values for certain EFuse fields. Values for compound fields such as BOOTIFCFG can be calculated using these macros or by hand if necessary. Refer to the MA2x5x databook in the EFuse Boot section of the Boot Operation chapter for more details.

To increase security when using the MA2155 Secure Boot feature, after programming the encryption keys, any unused redundancy slots can be disabled to minimize the changes that can be made to the EFuses. This is done using the `DrvEfuseLockRedundancy()` function.

Note: The standard MDK development board MV0182 does NOT support the ability to program the EFuse contents of the onboard Myriad processor. For this reason and due to the fact that EFuse programming is a once off destructive process, the eFuseProgrammer application contains an EXECUTION_GUARD macro which prevents the application from running until the source is modified. This example is intended as a starting point to allow the user to develop their own EFuse programmer for hardware that supports the feature.

4 EFuse Reader Application

The source code and makefile for the EFuse reader application can be found under the MDK installation directory in `examples/util/eFuseReader`.

This is a simple application which illustrates the process by which the eFuse contents can be read and displays them to the console. It makes use of the `DrvEfuseReadEFuses()` function to perform the read operation.

5 EFuse Limitations

The EFuse hardware has a number of usage constraints which must be respected in order to achieve reliable EFuse operation. For this reason it is very important that the Movidius provided EFuse driver is used for programming and reading EFuse bits as it has been designed to minimize program and read time in order to maximize the EFuse lifetime.

It is also important to note that the MA2x5x ROM will read the EFuse bits when it is configured for EFuse boot on each reboot operation. While the ROM EFuse driver minimizes the accumulative read time, each boot operation needs to be counted against the max number of read operations limit.

In order to achieve the tight timings required the EFuse driver requires a system frequency > 100 Mhz. The driver will return an error if this condition is not met.

The following table describes the key EFuse usage limitations:

Operation	Constraint	Limit

Max Accumulative read time per bit	Time while EFuse
	CSB=L
	LOAD=H
	PGENB=H
	STROBE=H
		2 seconds per bit

MAX accumulative number of read Operations	Number of times where
	CSB=L,
	LOAD=H,
	PGENB=H,
	STROBE=H
		2 million reads per bit

Max time while programming voltage applied	EFUSE_VDDQ = 1.8 V and EFuse PS=H	200 ms lifetime total
Supply Rail Max Ramp	VDDQ Supply Ramp Rate	1.8 V/30 µS max
Recommended programming temperature	Programming Temp Range	25 °C to 125 °C
Recommended programming voltage	VDDQ Voltage	1.8 V → 1.98 V (1.9 V recommended)

Table 2: EFuse Operational Limits

6 Programming Voltage

The MA2x5x package has a separate supply rail for the EFuse programming voltage (EFUSE_VDDQ). While this rail may be in any state (GND, FLOATING or 1.8 V) for EFuse read operations, it must be 1.8 V to 1.98 V during EFuse programming operations.

The EFuse subsystem contains an internal gate which is controlled by the EFUSE PS bit. This gate prevents the connection of EFUSE_VDDQ to the EFUSE array unless the PS bit is set to 1. During programming this gate is used to provide the fine grained programming pulses needed to blow the EFuse bits.

In order to avoid situations where the EFUSE_VDDQ is applied before the PS bit is in a known state it is very important that the EFUSE_VDDQ rail is never present before all other rails have stabilized and the Myriad processor has been reset. As such it is generally advised that the EFUSE_VDDQ rail is only applied to the processor for the period of time while the programming is active. This may be achieved in a number of ways, such as using a GPIO pin to control the output of an external 1.8 V regulator. or by applying the voltage using a bed of nails tester during board assembly. If using a regulator to supply the VDDQ voltage it should support a soft start feature in order to meet the VDDQ supply ramp specification of 1.8 V/30 µS max.

In the example application provided, this is controlled via a GPIO line, defined by the macro EFUSE_VDDQ_GPIO_EN. If the programming voltage is to be controlled in a different manner (e.g. delivered by a test harness) then the `efuseVddqSetState()` function can be amended.

7 Conclusion

The provided application serves to demonstrate the operations and interfaces required to program the EFuses on a MA2x5x in order to make use of EFuse Boot, Secure Boot (MA2155 only) or to store application related information.