

UCD30xx

Flash Programming,

Integrity and Security

Literature Number: xxxxxx
8/18/2010

Table of Contents

1	Introduction.....	3
2	Quick Start Summary.....	3
2.1	ROM Bootstrap and Program Flash Checksum.....	3
2.2	JTAG Interface.....	3
2.3	Firmware development setup.....	3
2.4	Production setup.....	4
3	Flash Memory Operations.....	4
3.1	UCD30xx Memory Maps	4
3.2	Flash Programming in ROM Mode	5
3.3	Flash Programming using JTAG	5
3.4	Clearing the flash	6
4	Flash Management for Firmware Development	6
4.1	Best Practice for Firmware Development.....	6
4.2	Firmware development with “backdoors”	6
4.3	I/O line based backdoors.....	7
4.3.1	Serial port based backdoor.....	7
4.3.2	GPIO line based backdoor.....	7
4.3.3	Other options for I/O backdoors.....	8
4.4	Communications backdoors.....	8
4.4.1	Cautions for using Communications Backdoors	8
4.5	JTAG backdoor	8
5	Flash Management in Production.....	8
6	Firmware Examples.....	9
6.1	Checksum Clearing.....	9
6.2	Erasing Flash.....	11

1 Introduction

Most members of the UCD30xx Family offer flash security. They all offer a ROM based PMBus bootstrap program for flash programming and program debugging. Most of them also offer JTAG.

They offer the choice between startup into the bootstrap program and startup into the customer program

This document shows how to make best use of these features. It describes how to enable flash security only when desired, and how to make “back doors” to permit reprogramming of devices with flash security enabled.

This document starts with a quick start summary which gives a recipe for best practices for firmware development and for production.

Next, it provides a detailed view of the UCD30xx Flash programming hardware and Boot ROM as a starting point.

Finally, it goes into detail with code examples for the exact procedures for Flash management for firmware development and for production.

2 Quick Start Summary

This section is a quick summary of the issues and solutions in Flash security. More detailed explanations are available in the sections which follow.

2.1 ROM Bootstrap and Program Flash Checksum

When the UCD30xx is reset or powered up, control goes first to the ROM bootstrap program. The ROM bootstrap initializes the device and then performs a checksum on the Program Flash. If the checksum matches, the ROM turns control over to the customer supplied program in the Program Flash.

If the checksum does not match, control stays in the ROM, and the flash can be reprogrammed.

2.2 JTAG Interface

In addition to the bootstrap, most UCD30xx devices (but not all), offer a JTAG interface. This is intended for debugging programs, and offers memory examine and modify functions as well as breakpoints and single step. If it is enabled, it can be used to erase the program flash, making the checksum invalid.

2.3 Firmware development setup

The best firmware development setup is:

1. Enable JTAG with the first instruction in the main program.
2. Never program the checksum into the program flash.

The disadvantage of this approach is that it requires the use of a PMBus interface every time the UCD30xx is reset. The program won't start on its own, so the start has to be commanded through the PMBus interface.

The second best setup permits auto startup. It is:

1. Enable JTAG with the first instruction in the main program
2. Have an I/O line based backdoor as the second instruction in the main program.
3. Have an additional backdoor from whatever communication port is in use (PMBus, or serial typically)
4. Only program the checksum into a new program version after the backdoors have been tested and verified.

PMBus backdoors are convenient, but unreliable. One bug can kill the PMBus function and the backdoor. A simple backdoor at the very beginning of the program is much more reliable.

2.4 Production setup

The best production setup is:

1. Leave JTAG disabled
2. Have a well tested, secure backdoor which erases program flash. It can be communications port based only, if desired.
3. Don't program the program flash checksum until all other flash is programmed, including calibration and manufacturing data.

The next sections cover these issues in much more detail.

3 Flash Memory Operations

3.1 UCD30xx Memory Maps

The UCD30xx has only one memory bus used for both program and data.

Memories are called "Program" Flash and "Data" Flash, but all memories can be used for both.

The names just show the main use for each memory.

After reset, the UCD30xx starts executing in ROM. In this mode, the ROM is mapped to fill up the entire first 64 Kbytes of memory. The 4 Kbytes of ROM are repeated 16 times.

Here is the memory map for ROM mode:

```
0x00000 - 0x0FFFF => Boot ROM - 4 Kbytes, repeated 16 times.  
0x10000 - 0x17FFF => Program Flash - 32 Kbytes  
0x18800 - 0x18FFF => Data Flash - 2 Kbytes  
0x19000 - 0x19FFF => Data RAM - 4 Kbytes
```

If the ROM finds a valid checksum for the Program Flash, or if a PMBus command is sent telling the ROM to transfer control to the flash, the memory map is changed a little bit.

Here is the memory map during normal operation:

```
0x0A000 - 0x0AFFF => Boot ROM - 4 Kbytes, repeated 16 times.  
0x00000 - 0x07FFF => Program Flash - 32 Kbytes  
0x18800 - 0x18FFF => Data Flash - 2 Kbytes  
0x19000 - 0x19FFF => Data RAM - 4 Kbytes
```

In ROM mode, the reset and interrupt vectors are in the ROM. In normal operation, the reset and interrupt vectors are in the Program Flash.

The two flash memories, "Program" and "Data", are programmed separately. Each has its own programming logic. When a Flash memory is being programmed, it cannot be read.

Flash memories can be programmed word by word, but must be erased a block at the time. The Data Flash has 64 small blocks containing only 32 bytes each. This makes it ideal for storing small blocks of data which need to be changed frequently.

The Program Flash has only 32 blocks, and each one contains 1K bytes.

3.2 Flash Programming in ROM Mode

In ROM mode, the flash is programmed via PMBus commands. The ROM offers several PMBus commands for memory examine and modify, and for controlling program execution. For a complete description, see the UCD30XX Boot ROM Reference Manual.

Normally the PMBus commands are issued by a TI supplied GUI running on a PC, or by a third party Flash programmer, so the process is invisible to the user.

First the Flash is erased using either one Mass Erase command or many Page Erase commands. Next the Flash is written to using Write Block commands. Then the Flash contents can be verified using Read Block commands. Finally an Execute Program command is used to start the Flash program.

At least one third party - System General - offers support for the UCD30xx using PMBus commands.

Their programmers are for units which are not yet soldered into boards.

3.3 Flash Programming using JTAG

Most UCD30xx devices also support JTAG. The UCD3040 in the 80 pin package JTAG is enabled with a reset. With all other devices, it is necessary to write a 0 to the IOMUXCTRL register.

This can be done with the following C language statement in flash memory:

```
MacRegs.IOMUXCTRL.all = 0; //enable JTAG
```

If the device is in ROM mode, it is also possible to enable JTAG by sending a Write Word command with the PMBus – writing a 0 to address 0xFFFF7F028. For detail on the Write Word command, please see the UCD30xx Boot ROM Reference Manual.

There are several JTAG programmers available:

JTAG Technologies – several devices

ASSET Intertech - RIC-1000 USB-1000

These products all work with the TI SVT file generation utility. Contact TI for more information on the SVT tools.

3.4 Clearing the flash

The UCD30xx can erase its own program flash, either in pages or as a single block. A block clear is best used for flash security for production code. In this way, even if the backdoor is activated, the flash security is still maintained.

It is also possible to write all zeroes to the checksum and clear it. That way, the next time the UCD30xx is reset, it will power up into ROM mode and the memory can be examined. This is best for development, so that the memory contents can be analyzed if necessary.

Any changes to program flash must be made from other memories – ROM, RAM, or Data Flash. It is possible to corrupt Data flash without affecting the Program Flash checksum, so it is unwise to put the flash modification program into Data Flash.

It is best to put it into Program Flash and copy it into RAM. In this way, if the program image is corrupted, the checksum will be incorrect, and the device will automatically enter ROM mode on reset.

There are examples of flash erasing and checksum clearing programs in Section 6, Firmware Examples.

4 Flash Management for Firmware Development

During the development phase, the main goal is to avoid activating flash security.

There are several ways to do this, depending on the situation

4.1 Best Practice for Firmware Development

The best practice for firmware development is very simple.

1. Always enable JTAG with the first line of code in main.
2. Never make the program flash checksum correct.

This provides double protection:

1. JTAG can always be used to recover the device
2. The device will always reset into ROM mode, so it can always be reprogrammed.

The disadvantages of this method are:

1. The device must always be connected to the PMBus and told to start executing
2. The JTAG pins cannot be used for other functions.
3. Anyone with the proper tools can read the flash – there is no security.
4. To use JTAG, a JTAG pod and software are required, as well as a JTAG connector on the UCD30xx board.

4.2 Firmware development with “backdoors”

Sometimes there are situations where development level firmware is required to start automatically with no PMBus interface command to start it. In this case, the program flash checksum must be programmed.

There may also be cases where flash security is desired for firmware under development. Again, the checksum must be programmed, preventing the device from going to ROM mode.

In this case, if reprogramming is desired, some kind of backdoor must be provided to clear the checksum. There are several backdoor techniques described below. Any one of them is adequate if used properly. For a robust and easy to use solution, however, the use of 2 or more backdoors is strongly suggested.

The techniques are:

I/O line based backdoors
Communications port based backdoors
JTAG backdoor

4.3 I/O line based backdoors

This backdoor can provide security, if done properly. It starts with the firmware checking an I/O line at startup – before the rest of the system is initialized – and branching to the backdoor if the I/O line is in the proper state.

The big advantage of the I/O line based backdoor is that firmware changes are unlikely to make it stop working. Since it is at the very beginning of the code, changes later in the code should not affect it.

The simplest way involves just branching straight to the code that clears the flash. In this case, the code can erase the entire flash, preventing others from being able to read it. There are several ways to do this:

4.3.1 Serial port based backdoor

The serial port backdoor is most useful if the serial port is being used for primary to secondary communication. The serial port (RX) pin can be programmed as an input and its state read. If the line is high, the serial port is in its normal state. If it is low, then the flash should be cleared. It may be necessary to put in a pull up resistor if chip transmitting is absent or powered down. It is also necessary to ensure that the other chip will not transmit data when the UCD30xx is coming out of reset.

Advantages

1. Doesn't waste an I/O line
2. Can be triggered by other chip via serial port

Disadvantages

1. Pull up may be needed
2. Other chip must avoid transmitting at sampling time (just after reset)

4.3.2 GPIO line based backdoor

Using a general purpose I/O line is simpler, as the line can be dedicated to this function, but it does require a free line available. It will also require a pull up or down and a test point to override the pull up or pull down.

However, it is often useful to have a free I/O line available for development. It is very useful for instrumenting code. It can be used to indicate internal events to monitor timing and trigger oscilloscopes. Since the line is only checked at reset for the backdoor, it can then be used for other functions once the program is started.

4.3.3 Other options for I/O backdoors

There are many other creative options for backdoors if I/O lines are very constrained. An ADC input can be set to an out of normal range value. As suggested above, an I/O line can perform a backdoor function at the beginning of the code and some other function after startup.

4.4 Communications backdoors

Communications backdoors add a message to an existing communications port already used in the application. Typically this is the PMBus interface, but it could be any communications interface.

The standard TI firmware generally supports a simple communications backdoor with a PMBus D9 command used to clear the flash checksum. For added security, this could be changed so that it erases the flash instead. In addition, more bytes could be added to the command sequence, requiring a multi-byte checksum for flash changes.

The serial port could also be used in a similar way.

Advantages:

1. Requires no additional I/O pins
2. Can support password security

Disadvantages

1. If firmware locks up, the backdoor can also stop working

4.4.1 Cautions for using Communications Backdoors

Since a firmware bug can lock up the communications backdoor very easily, always test the backdoor before setting the checksum up for auto startup. Every time the firmware is changed, retest the backdoor.

4.5 JTAG backdoor

The JTAG backdoor simply involves leaving the JTAG interface enabled. This requires dedicating these pins to JTAG, and having a connector, JTAG pod and firmware available. If JTAG is being used during the development process, this is easy.

Anyone doing the development process should know how to use Code Composer Studio and the JTAG, so this is not covered in this document.

The easiest way to clear the checksum is to erase the flash. To do this, simply set the MASS_ERASE bit in the Program Flash Control Register. This is bit 8 at location 0xFFFFFE60. So simply use the JTAG to write a 0x100 to 0xFFFFFE60. This will set the flash to all FFs making the checksum invalid.

5 Flash Management in Production

In production, the goals are different. A secure backdoor is desired, but it must also be reliable. Firmware bugs which prevent backdoor access are less of a concern. All of the firmware, including the backdoor should be well tested before release to production.

The best firmware backdoor is a communications channel based backdoor with a long password, as described above. It could even be a sequence of multiple commands, if desired.

6 Firmware Examples

This section shows short examples of checksum clearing and flash erasing.

6.1 Checksum Clearing

This first code typically goes into the software interrupt, as it must be used in system mode, not in user mode.

```
// clear integrity word.
// copies code from program flash to RAM

DecRegs.PFLASHILOCK = 0x42DC157E;
// Write key to Program Flash Interlock Register
{
    register Uint32 * program_index = (Uint32 *) 0x19000;
    //store destination address for erase checksum program

    register Uint32 * source_index = (Uint32 *)zero_integrity_word;
    //Set source address of PFLASH;

    register Uint32 zoiw_size = (Uint32 *)zoiw_end - (Uint32
    *)zero_integrity_word; //Calculate length

    register Uint32 counter;

    for(counter=0; counter < zoiw_size; counter++)
    //Copy program from PFLASH to RAM
    {
        *(program_index++)=*(source_index++);
    }
}
{
    register FUNC_PTR func_ptr;
    func_ptr=(FUNC_PTR)0x19000;
    //Set function to 0x19000 (start of RAM)
    func_ptr(); //execute erase checksum
}
return;
```

This is the code for actually clearing the flash which is copied into RAM:

```
;zero out integrity word - zoiw
;derived from this C program
#include "..\common\UCD30xx_Device.h" // UCD92xx Headers Include File
#include "..\common\UCD30xx_Defines.h" // UCD92xx Headers Include File
#include "..\common\UCD30xx_specific.h"
#include "..\common\common_function_prototypes.h"
#include "function_prototypes.h"
#include "software_interrupts.h"
;
```

```

#define program_flash_integrity_word *((volatile unsigned long *) 0x7ffc)
//last word in flash, when executing from Flash.  used to store integrity
code

;void zero_out_integrity_word(void)
;{
;    DecRegs.MFBALR1.all = MFBALRX_BYTE0_BLOCK_SIZE_32K; //enable program
flash write
;    program_flash_integrity_word = 0;
;    DecRegs.MFBALR1.all = MFBALRX_BYTE0_BLOCK_SIZE_32K + //expand program
flash out to 4x real size
;
;                                MFBALRX_BYTE0_ONLY;
;
;    while(DecRegs.DFLASHCTRL.bit.BUSY != 0)
;    {
;        ; //do nothing while it programs
;    }
;}
;

.state32
.global    _zero_integrity_word
.global _zoiw_end

;*****
*
;* FUNCTION NAME: zero_out_integrity_word
*
;*
;*
;*    Regs Modified      : A1,V9,SR
;*
;*    Regs Used          : A1,V9,SR
;*
;*    Local Frame Size   : 0 Args + 0 Auto + 0 Save = 0 byte
;*
;*****
*
_zero_integrity_word:
;* -----
*
;    MOV      V9, #96                ; |13|
;    LDR      A1, CON1                ; |13|
;    STR      V9, [A1, #0]            ; |13|
;    MOV      A1, #0                  ; |14|
;    MOV      V9, #32768              ; |14|
;    SUB      V9, V9, #4              ; |14|
;    STR      A1, [V9, #0]            ; |14|
;    MOV      V9, #98                 ; |15|
;    LDR      A1, CON1                ; |15|
;    STR      V9, [A1, #0]            ; |15|
;* -----
*
;* BEGIN LOOP L1
;*
;*    Loop source line          : 18
;*    Loop closing brace source line : 21
;*    Known Minimum Trip Count   : 1

```

```
; * Known Maximum Trip Count : 4294967295
; * Known Max Trip Count Factor : 1
; * -----
; *
L1:
    LDR    V9, CON2                ; |18|
    LDRB   V9, [V9, #0]            ; |18|
    MOV    V9, V9, LSR #3          ; |18|
    MOVS   V9, V9, LSL #31         ; |18|
    BNE    L1                      ; |18|
    BX     LR

    .align 4
CON1: .field _DecRegs+12,32
    .align 4
CON2: .field _DecRegs+102,32
_zoiw_end: ;reference point for end of code.
;*****
**
; * UNDEFINED EXTERNAL REFERENCES
; *
;*****
**
    .global _DecRegs
    .end
```

This code used assembly language generated by the C compiler. This conversion makes it easy to make sure that the `zoiw_end` label goes in the right place at the end of the code, and that the code is all PC relative so that the relocation to RAM causes no problems. There are other ways to achieve the same result.

6.2 Erasing Flash

The code for erasing flash is very similar, except instead of writing to the program flash word, it actually writes to the program flash control register and does a mass erase.

The C calling function is very similar. Only the labels for the code which is copied change.

The parts which change from the code above are shown below:

```
// Execute the function in Data Flash that will erase the PFlash.
//clear program flash();

{
    register Uint32 * program_index = (Uint32 *) 0x19000; //store
destination address for erase checksum program
    register Uint32 * source_index = (Uint32 *)clear_program_flash;
//Set source address of PFLASH;

    register Uint32 cpf_size = (Uint32 *)end_of_clear_program_flash -
(Uint32 *)zero_integrity_word;//Calculate lenght
    register Uint32 counter;

    for(counter=0; counter < cpf_size; counter++) //Copy program
from PFLASH to RAM
```

```

    {
        *(program_index++)=*(source_index++);
    }
}

```

Here is the assembly language code to clear the program flash, along with the C code which created it in the first place:

```

.state32
.global      _clear_program_flash
.global      _end_of_clear_program_flash

.state32
.global      _clear_program_flash
.global      _end_of_clear_program_flash

_clear_program_flash:
    LDR        A1, CON1                ; |5|
    LDR        V9, [A1, #0]            ; |5|
    ORR        V9, V9, #256            ; |5|
    STR        V9, [A1, #0]            ; |5|
L1:
    LDR        V9, [A1, #0]            ; |7|
    TST        V9, #2048                ; |7|
    BNE        L1                      ; |7|

    LDR        V9, CON2                ; |13|
    LDR        A1, [V9, #152]           ; |13|
    ORR        A1, A1, #32              ; |13|
    STR        A1, [V9, #152]           ; |13|

    LDR        A1, [V9, #152]           ; |14|
    BIC        A1, A1, #32256           ; |14|
    ORR        A1, A1, #256             ; |14|
    STR        A1, [V9, #152]           ; |14|

L2:
    B          L2                      ; |17|
    .align     4
CON1: .field    _DecRegs+96,32
    .align     4
CON2: .field    _TimerRegs,32
_end_of_clear_program_flash:
;*****
**
;* UNDEFINED EXTERNAL REFERENCES
*
;*****
**

    .global      _DecRegs
    .global      _TimerRegs

; generated by the following C program:
;
;#include "include.h"
;
;void clear_program_flash(void)

```

```
;{  
;    DecRegs.PFLASHCTRL.bit.MASS_ERASE = 1; //erase it all  
;  
;    while(DecRegs.PFLASHCTRL.bit.BUSY != 0)  
;    {  
;        ; //do nothing while it programs  
;    }  
;  
;    //now reset processor.  
;    TimerRegs.WDCTRL.bit.CPU_RESET_EN = 1; // Make sure the watchdog is  
enabled.  
;    TimerRegs.WDCTRL.bit.WD_PERIOD = 1; // Set WD period to timeout  
faster.  
;  
;    // This will never test true, but it prevents a compiler warning about  
unreachable code.  
;    while(1); // Wait for the watchdog.  
;    //end_of_clear_program_flash needs to go here, go to assembly to put it  
there.  
;    return;  
;}
```