

# SAKCIC61508 SPI Communications Examples

## Application Note

Examples of how to use the standard and secure SPI modes to access SAKCIC61508 on-chip resources plus using the DFLASH erase and programming features

Released

## Using The CIC61508 SPI Interface

V1.0 2011-08

**Edition 2011-08**

**Published by**

**Hitex (UK) Ltd.**

**University Of Warwick Science Park, Coventry UK**

**© 2015 Hitex (UK) Ltd.**

**All Rights Reserved.**

**Legal Disclaimer**

The information given in this document shall in no event be regarded as a guarantee of conditions or characteristics. With respect to any examples or hints given herein, any typical values stated herein and/or any information regarding the application of the product, Hitex (UK) Ltd. hereby disclaims any and all warranties and liabilities of any kind, including without limitation, warranties of non-infringement of intellectual property rights of any third party.

**Information**

For further information on technology, delivery terms and conditions and prices, please contact the nearest Hitex Office ([www.hitex.com](http://www.hitex.com)).

**Document Change History**

<b>Date</b>	<b>Version</b>	<b>Changed By</b>	<b>Change Description</b>
24/6/11	0.1	M Beach	First version for external review
14/7/11	0.2	M Beach	Authorising version
9/8/2011	1.0	M Beach	Released version

**We Listen to Your Comments**

Is there any information in this document that you feel is wrong, unclear or missing?  
Your feedback will help us to continuously improve the quality of this document.  
Please send your comments (including a reference to this document) to:

[comments@hitex.co.uk](mailto:comments@hitex.co.uk)



## Table of Contents

<b>1</b>	<b>Overview .....</b>	<b>9</b>
1.1	Installing The Example Applications .....	12
1.1.1	Importing The Examples Into Tasking Eclipse .....	13
1.2	Running The Example Applications .....	17
1.3	TC1782 Setup .....	17
1.3.1	SSC0 Configuration .....	17
1.3.2	GPTA0 Configuration .....	17
<b>2</b>	<b>App0: Simple message exchange with the CIC61508 .....</b>	<b>18</b>
<b>3</b>	<b>App1 &amp; App2: Message exchange with the CIC61508 .....</b>	<b>20</b>
<b>4</b>	<b>App3: Practicable message exchange with the CIC61508.....</b>	<b>23</b>
4.1	Providing A SPI Message Timebase.....	23
4.1.1	Further Refinement Of SPI Driver .....	25
<b>5</b>	<b>App4: Adding a SFR WRITE function.....</b>	<b>26</b>
<b>6</b>	<b>App5: Using the SFR WRITE function to load the SPI error counter .....</b>	<b>29</b>
<b>7</b>	<b>App6: Entering Secure SPI Mode .....</b>	<b>30</b>
7.1	Secure SPI Mode Example Messages .....	30
7.2	How To Get Into Secure SPI Mode.....	31
7.3	First Steps In Secure SPI Mode .....	33
<b>8</b>	<b>App7: Using Secure SPI Mode Features.....</b>	<b>35</b>
8.1	CIC61508 Memory Spaces .....	35
8.2	Creating A Function To Read Any Secure Mode Address .....	36
8.3	Writing Of Secure Mode Addresses.....	37
<b>9</b>	<b>App8: Erasing And Programming The DFLASH .....</b>	<b>40</b>
9.1	Erasing The DFLASH.....	40
9.1.1	Erase Function Error Codes.....	41
9.1.2	Creating The Erase Delay.....	41
9.1.3	Using The uiCIC_EraseDFLASH() Erase Function .....	42
9.2	Programming The DFLASH .....	43
9.3	Exiting Secure SPI Mode After DFLASH Reprogramming .....	45
<b>10</b>	<b>App9: Programming The DFLASH From A TARDISS Export .....</b>	<b>46</b>
10.1	Restoring Corrupted DFLASH.....	47
<b>11</b>	<b>App10: Typical Usage Example .....</b>	<b>48</b>
<b>12</b>	<b>Troubleshooting .....</b>	<b>50</b>
<b>13</b>	<b>Appendices .....</b>	<b>51</b>
13.1	Appendix 1 - CIC61508 SPI Functions .....	51
13.1.1	uint16 uiCIC_SPI_SendMessage(uint16 uiTx_data) .....	51
13.1.2	void vCIC_SPI_Driver(void) .....	51
13.1.3	CIC_ErrorType uiCIC_SPI_ReadSFR(uint8 uiSFR_addr, uint8 uiSFR_data).....	51
13.1.4	CIC_ErrorType uiCIC_SPI_WriteSFR(uint8 uiSFR_addr, uint8 uiSFR_data).....	51
13.1.5	CIC_ErrorType uiCIC_SPI_EnterSecureMode(void) .....	52
13.1.6	uint16 uiCIC_ReadSecureAddress(uint16 uiAddress, uint16 uiMspace) .....	52
13.1.7	CIC_ErrorType uiCIC_WriteSecureAddress(uint8 uiData, uint16 uiAddress, uint16 uiMspace).....	52
13.1.8	CIC_ErrorType uiCIC_EraseDFLASH(void) .....	52
13.1.9	CIC_ErrorType uiCIC_WriteDFLASH_Wordline(uint8 uiInputData, uint16 uiWordlineBaseAddr, uint16 uiOffset) .....	53
13.1.10	CIC_ErrorType uiCIC_WriteEntireDFLASH(uint8 uiInputData, uint16 uiLength) .....	53
13.1.11	CIC_ErrorType uiReset_CIC(void).....	53
13.1.12	CIC_ErrorType uiReset_CIC_Immediate(void).....	53
13.1.13	uint16 uiCIC_SPI_FastWriteSFR(uint8 uiSFR_addr, uint8 uiSFR_data) .....	54
13.2	Appendix 2 - CIC61508 SPI Functions Memory Usage.....	55

## List of Figures

Figure 1	Example Applications In Tasking Eclipse .....	10
Figure 2	Example Application Directory Layout .....	10
Figure 3	uiCIC_SPI_ReadSFR Flowchart .....	21
Figure 4	GPTA_LTC01 SRN22 interrupt handler.....	23
Figure 5	Entering Secure SPI Mode.....	32
Figure 6	Read any address sequence .....	34
Figure 7	Write Any Address Sequence .....	37
Figure 8	Erase DFLASH Sequence .....	40
Figure 9	DFLASH Wordline programming procedure .....	44
Figure 10	Breaking up the 0x1000 DFLASH image into 32-byte wordlines for programming .....	45
Figure 11	PRO-SIL TestBench NVM Tables Tab.....	46
Figure 12	Real Application Example – Programming Corrupted Or Blank DFLASH .....	49

## **List of Tables**

Table 1	Troubleshooting.....	50
Table 2	ROM Requirement For SPI Functions .....	55

## List of Code Listings

Code Listing 1	Reading The CIC61508 SVER SFR.....	19
Code Listing 2	Writing The CIC61508 OTRH SFR .....	19
Code Listing 3	Continually read the ERRORSYSTEM SFR .....	20
Code Listing 4	CIC_SPI_Driver Interrupt Provides Message Timebase.....	24
Code Listing 5	Beginning a SPI transmission .....	25
Code Listing 6	System tick timeout cases .....	27
Code Listing 7	Initialising the SPI Error Counter Using The SFR Write Function .....	29
Code Listing 8	Reading the CIC61508 CRC8 at C:0x2FFF in secure SPI mode .....	33
Code Listing 9	Secure SPI Command Set .....	34
Code Listing 10	The uiCIC_ReadSecureAddress() Function .....	36
Code Listing 11	Writing a XDATA Location .....	37
Code Listing 12	The uiCIC_WriteSecureAddress() Function .....	38
Code Listing 13	Cause A CIC61508 Reset .....	39
Code Listing 14	Causing CIC61508 Reset .....	39
Code Listing 15	Possible Erase Error Codes .....	41
Code Listing 16	Creating the 102ms erase delay .....	41
Code Listing 17	Reading the entire DFLASH into an array prior to erasing .....	42
Code Listing 18	Creating the 2.6ms programming delay .....	43
Code Listing 19	DFLASH Image Export From TARDISS/PRO-SIL TestBench .....	46

## **Introduction**

This application covers the use of the CIC61508 standard and secure SPI modes. It uses a step-by-step approach to showing how to access the CIC SFRs and DFLASH programming modes from the SPI interface. A TC1782 SafeTkit board is used as a basis as this is a stable and well-known platform that is also used for PRO-SIL applications proving and development.

The example applications developed in this application note can be downloaded from:

[ftp://internal.hitex.co.uk/pub/hitex/CIC61508/CIC61508\\_SPI.zip](ftp://internal.hitex.co.uk/pub/hitex/CIC61508/CIC61508_SPI.zip)

The unzip password is "hitex2013".



## 1 Overview

It is accompanied by example applications which are built using the Tasking Eclipse VX tools v3.5r1 and executed using the evaluation version of the Hitex HiTOP54-TC debugger.

The final application contains all the techniques set out in earlier programs and provides a complete set of CIC61508 standard and secure mode driver functions.

Each application builds on the previous one. In all cases, the objective is to end up with a simple to use function with a clear API that does something useful. For example, to read a CIC61508 SFR step-by-step requires us to:

1. Wait for any previous transmission to complete
2. Reset transmit interrupt request
3. Write the SFR offset to the SSC Transmit buffer
4. Loop until the transmission buffer is available again
5. Loop until the bit transmission takes place
6. Reset transmit interrupt request
7. Do steps 3 to 6 and then go to 8
8. Check that the CIC61508 returned the SFR address in the top byte

Clearly in a real application it would be much more convenient to just do:

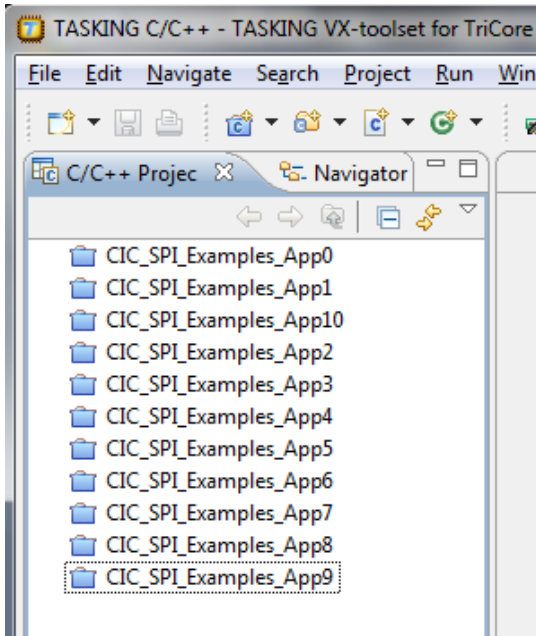
```
/* Read CIC61508 SFR at address 0x2C (SVER) */  
CIC_Status = uiCIC_SPI_ReadSFR(0x2C, (uint16 *) &uiSFR_value);
```

Thus where possible, the steps to perform an action are placed into a single function. Future examples can then make use of the earlier functions at will.

By developing a series of simple functions, eventually we will be able to read and write any SFR, enter secure SPI mode, erase the DFLASH, programme the DFLASH and reset the CIC61508.

This after all is the whole basis of modular programming!

The 11 example applications are pre-installed in the Eclipse IDE:



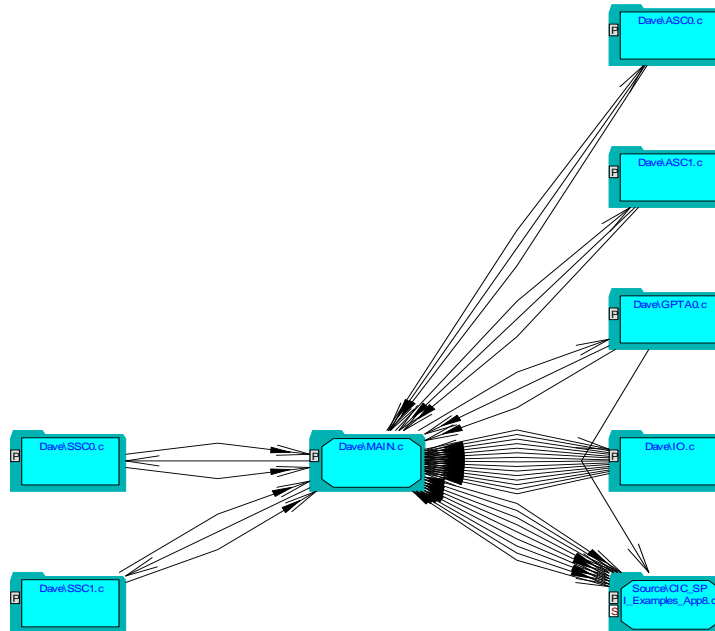
**Figure 1 Example Applications In Tasking Eclipse**

Each “AppX” project has a directory structure similar to the one shown below.

Name	Date modified	Type	Size
Dave	22/06/2011 11:47	File folder	
Debug	22/06/2011 14:16	File folder	
Debugger	22/06/2011 11:55	File folder	
include	22/06/2011 11:47	File folder	
Source	22/06/2011 11:47	File folder	
.cproject	22/06/2011 14:15	CPROJECT File	35 KB
.project	22/06/2011 11:46	PROJECT File	1 KB
CIC_SPI_Examples_App0.lsl	22/06/2011 11:46	LSL File	4 KB
CIC_SPI_Examples_App0.simulator.launch	22/06/2011 11:46	LAUNCH File	3 KB
cstart.c	22/06/2011 11:46	C File	39 KB
cstart.h	22/06/2011 11:46	H File	11 KB
DConfig	22/06/2011 11:46	File	1 KB
MConfig	22/06/2011 11:46	File	1 KB

**Figure 2 Example Application Directory Layout**

The SOURCE directory contains a C module “CIC\_SPI\_Examples\_AppX.C. All the source code that communicates with the CIC61508 is place in here. The module DAVE\MAIN.C simply acts as a caller and manages the top-level sequence of operations.



The files CSTART.C and CSTART.H are generated by the Tasking Eclipse and take care of the TC1782 CPU initialization. The .LSL file tells the Tasking linker about the TC1782 memory map. The MCONFIG and DCONFIG are again automatically generate by the tools and should not be altered by the user.

## 1.1 Installing The Example Applications

The examples must be imported into the Tasking Eclipse environment. This is done from an import directory.

Run the installer “CIC61508\_SPI.exe” and follow the on-screen instructions. The result will be a directory structure on driver C:.

Name	Date modified	Type
CIC61508_SPI_Examples	24/06/2011 15:18	File folder
CIC61508AppletExamples	18/03/2011 13:59	File folder
CIC61508FW	08/06/2011 11:16	File folder
Config.Msi	24/06/2011 08:54	File folder

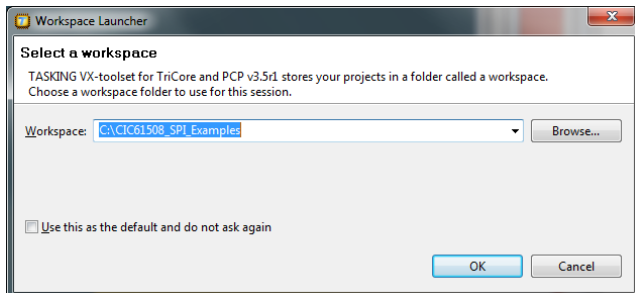
The subdirectory “CIC\_SPI\_Examples” contains further directories as shown:

Name	Date modified	Type
CIC_SPI_Examples_App0	24/06/2011 15:18	File folder
CIC_SPI_Examples_App1	24/06/2011 15:18	File folder
CIC_SPI_Examples_App2	24/06/2011 15:18	File folder
CIC_SPI_Examples_App3	24/06/2011 15:18	File folder
CIC_SPI_Examples_App4	24/06/2011 15:18	File folder
CIC_SPI_Examples_App5	24/06/2011 15:18	File folder
CIC_SPI_Examples_App6	24/06/2011 15:18	File folder
CIC_SPI_Examples_App7	24/06/2011 15:18	File folder
CIC_SPI_Examples_App8	24/06/2011 15:18	File folder
CIC_SPI_Examples_App9	24/06/2011 15:18	File folder
CIC61508	24/06/2011 15:18	File folder
Docs	24/06/2011 15:18	File folder

The directories CIC\_SPI\_Examples\_App0,1,2 etc. contain separate Tasking Eclipse projects. These need to be imported into Eclipse before we can make use of them.

### 1.1.1 Importing The Examples Into Tasking Eclipse

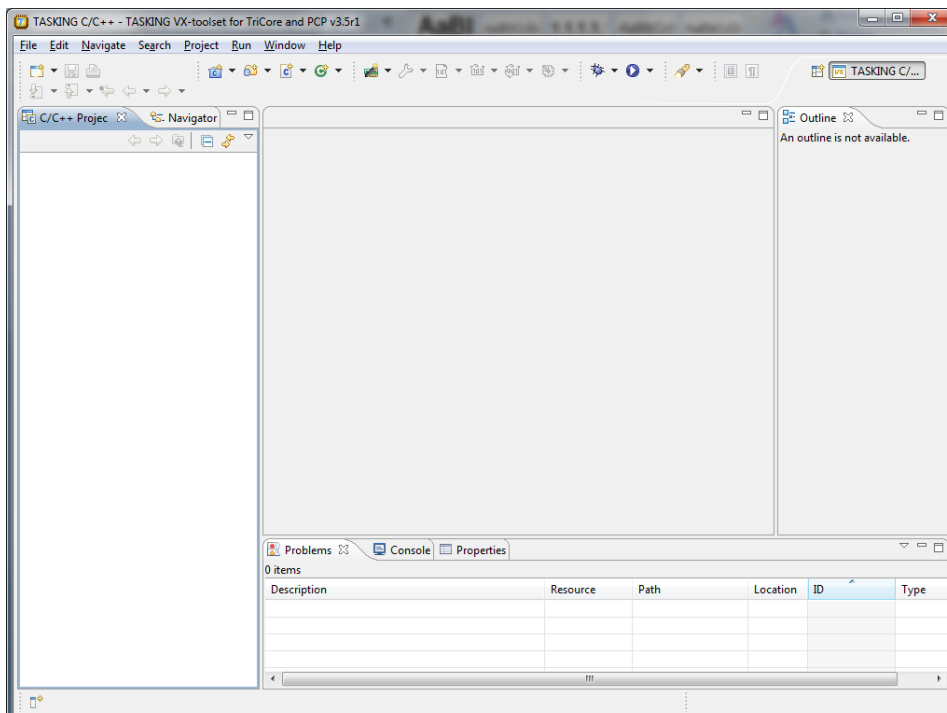
Start the Tasking Eclipse. When prompted for a workspace location, browse for the directory just created by the installer.



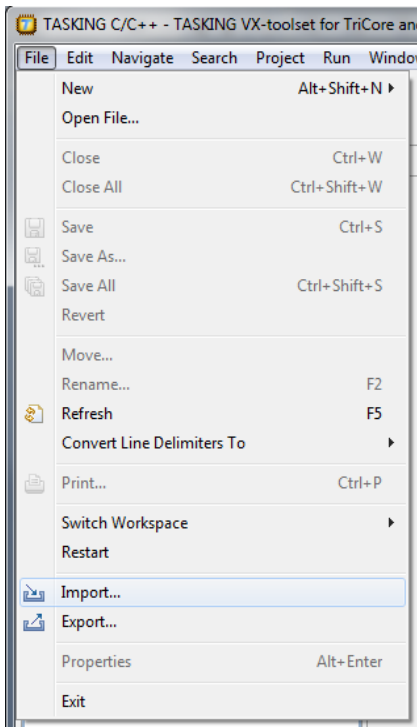
Eventually, the Eclipse will open with a Welcome screen.



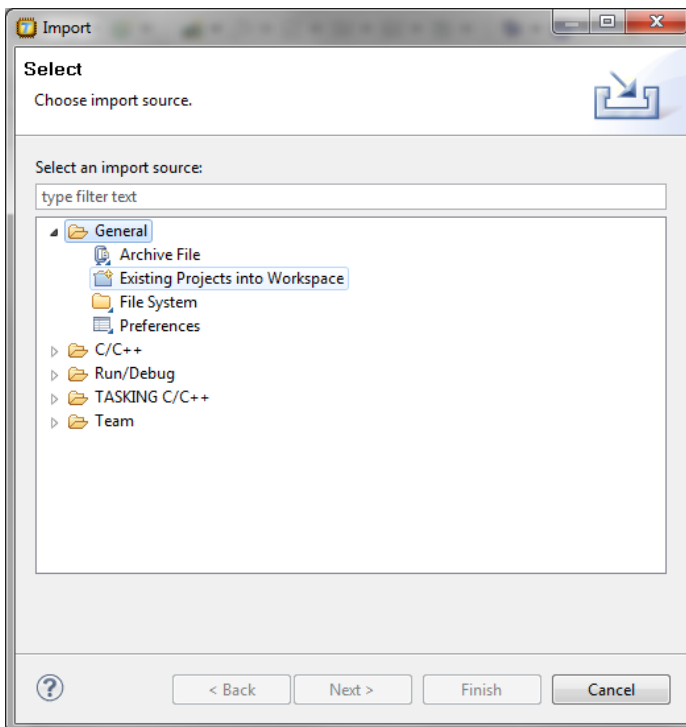
Close this by clicking on X to the right of "Welcome" on the tab to reveal:



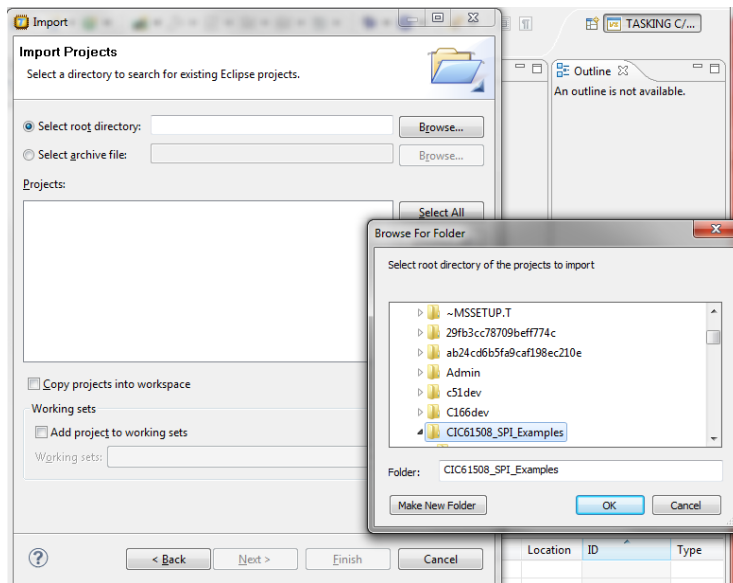
Next we need to import the 11 example applications. This is done by choosing File-Import.



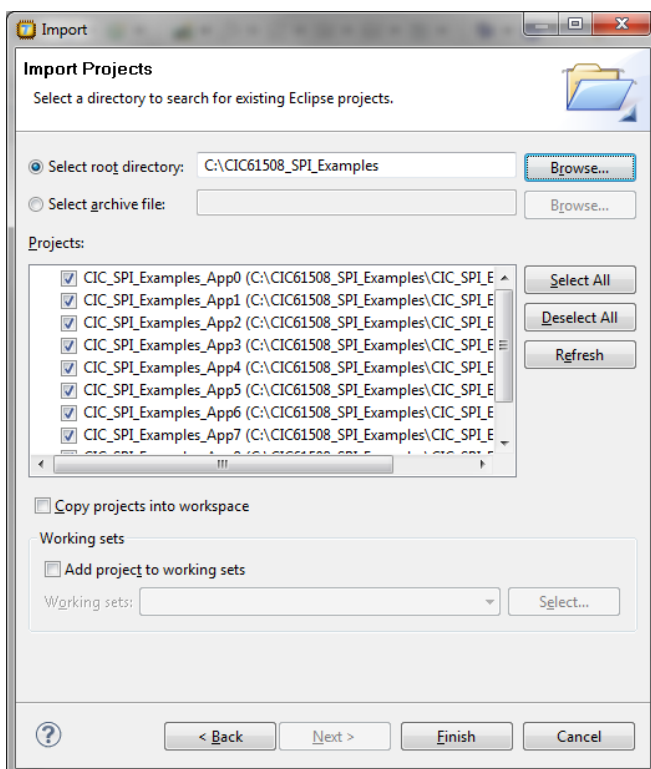
And then General-Existing Projects Into Workspace.



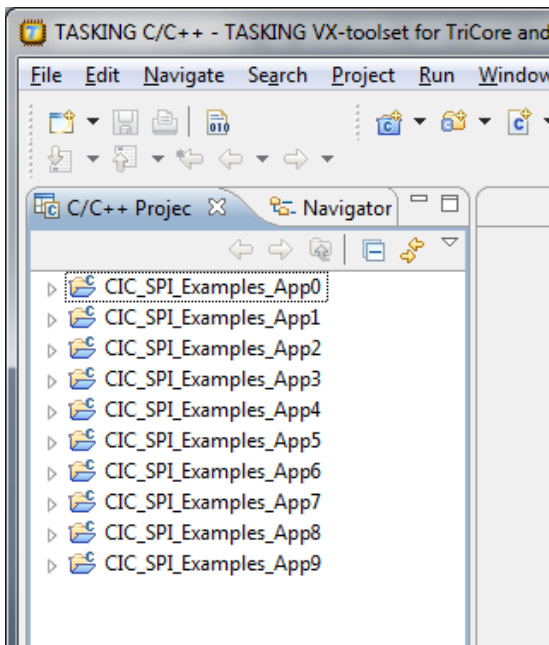
Then click "Next" and then the CIC61508\_SPI\_Examples directory:



Click OK and Eclipse will list all 10 example applications, with the select boxes ticked. As we want all the applications, just click finish.



The Eclipse project window will list all 11 projects.



The example applications are now ready for use.



## 1.2 Running The Example Applications

A line-by-line explanation of the examples is not provided in this document. Instead, it is recommended to load the examples into a TriCore debugger. Project files for the HiTOP54-TC debugger are supplied as this is what is provided in the Hitex TC1782 SafeTkit but other debuggers can be used. The only requirement is that they can connect to the USB-JTAG interface and load programs created using Tasking VX TriCore v3.5r1.

The examples are fully commented so what they are doing should be clear. Where there is a critical action or timing required, this is highlighted and a flow chart is provided in the section devoted to that example application later on in this document.

For each application described in this document, it is recommended to load the associated example application and single-step to gain an appreciation of what is happening. The operation of each significant function is described with a combination of code listings, explanations and flowcharts.

## 1.3 TC1782 Setup

The 10 examples are based on the DaVE tool which provides the TC1782 configuration and sets up the SSC0 and GPTA0 for use in the SPI examples. The TC1782 oscillator is 20MHz and the actual CPU frequency is 180MHz. The Tasking standard TC1782 SFR header file is used rather than the DAVE-generated one. This makes the functions developed in the examples more easily transferrable to PRO-SIL applications.

### 1.3.1 SSC0 Configuration

fractional divider mode  
required baud rate = 1500000 baud  
SSC0 master mode  
transfer data width is 16 bit  
transfer/receive LSB first  
shift transmit data on the leading clock edge, latch on trailing edge  
idle clock line is low, leading clock edge is low-to-high transition  
check receive error is enabled  
check phase error is enabled  
ignore receive parity errors  
parity transmit enable bit is disabled  
parity receive enable bit is disabled  
even parity is selected  
P3.6 is used as SSC0 slave select output signal 1 ( SLISO01)  
slave output select leading delay: 3 SCLK periods  
slave output select trailing delay: 2 SCLK periods  
slave output select inactive delay: 3 SCLK periods  
P3.2 is used as SSC0 clock output signal ( SCLK0)  
P3.4 is used as SSC0 master transmit slave receive output signal (MTRSR0)  
P3.3 is used as SSC0 master receive slave transmit input signal (MRST0)

### 1.3.2 GPTA0 Configuration

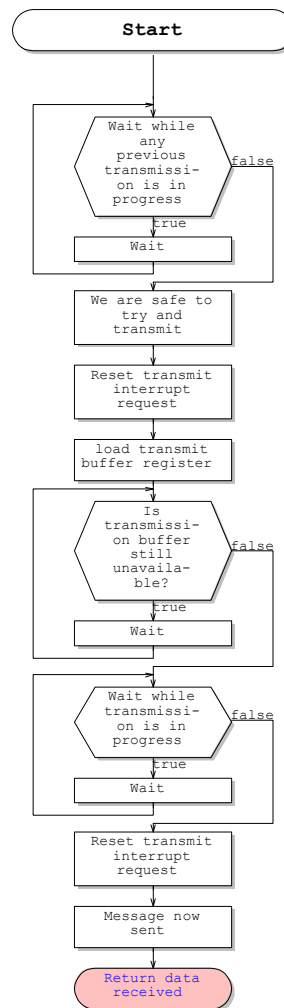
The basic GPTA0 clock is 51.24 MHz which is divided down by 128 by the LTC prescaler to give a timebase of 2.5us/count (4MHz). LTC00 is used as a free running timer and LTC01 is configured as a compare register, able to generate an interrupt on SRN22. The period of the interrupt is set by the SPI handler interrupt which is called from this SRN.

The blue LED on TC1782 Port 5.0 is toggled on each interrupt so show that something is happening.

## 2 App0: Simple message exchange with the CIC61508

This is the simplest possible way to communicate with the CIC61508.

The first step to establish contact with the CIC is to send a read SFR command. At this early stage, several steps are required to do this.



The code written to implement this is in `CIC_SPI_Examples_App0.C`. It is called from `MAIN.C`.

One important point about low-level CIC SPI communications is that as a 16-bit SPI slave, the CIC only emits its 16 bits of data when it receives data and that this data is sent in parallel to the transmission by the master (TC1782). Thus the reply to any master transmission belongs to the previous message. This means that to get the value of a particular SFR, two read commands must be issued by the master. The first one sends the address of the SFR to read while the second one sends the same data but importantly, the reply to this message contains the actual SFR contents. The first reply is simply discarded. This is illustrated in App0.

```
/* Read CIC61508 SFR at address 0x2C (SVER) */  
uiCIC_value = uiCIC_SPI_SendMessage(0x002C);  
  
/* 0x5555 returned */  
  
/* Read CIC61508 SFR at address 0x2C (SVER) */  
uiCIC_value = uiCIC_SPI_SendMessage(0x002C);  
  
/* Data returned will be 0x2C28 */
```

#### **Code Listing 1     Reading The CIC61508 SVER SFR**

In this code fragment, the address of the SVER SFR (0x2C) is sent as the low byte of the 16-bit message, As the CIC is assumed to have just powered-up, the first reply it always sends is "0x5555". Sending the address of the SVER a second time results in the CIC sending back the contents of the SVER as the low byte and the address of the SVER (previously sent) as the high byte. In more advanced examples, this high byte is compared with the SFR address to detect possible errors. Here we just take the data as received.

To write a CIC SFR, the process is slightly different. Continuing on directly from the first code fragment, we will attempt a write to the OTRHH SFR whose address is 0x00.

```
/* Write 0xAA to OTRHH SFR at location 0x00 in CIC61508 */  
uiCIC_value = uiCIC_SPI_SendMessage(0xAA00 | 0x0000 | 0x0080U);  
  
/* Data returned will be 0x2C28 */  
  
/* Read back the value of OTRHH */  
uiCIC_value = uiCIC_SPI_SendMessage(0x0000);  
  
/* 0x80AA returned. OTRHH address now in the upper byte */
```

#### **Code Listing 2     Writing The CIC61508 OTRH SFR**

To cause a write, the data to write to the SFR is in the high byte and the address of the SFR is bit0-6 of the low byte, However to cause the write, bit 7 of the low byte is set i.e.

Message = ((DataToWrite << 8) | SFR address | 0x80 )

This simple write system does not check whether the write was successful, which is undesirable in a real system. This will be addressed in a later example.

### 3 App1 & App2: Message exchange with the CIC61508

This will give us a CIC61508 read function with error checking.

App1 is an intermediate step to App2. App1 creates the function “CIC\_ErrorType uiCIC\_SPI\_ReadSFR(uint8 uiSFR\_addr, uint8 \*uiSFR\_data)” which has a more usable API than just the simple “uiCIC\_SPI\_SendMessage(0x0000)” from App0.

The new function reads a CIC SFR by sending the address of that SFR and the address of a variable in which to put the SFR value. The operation is repeated so that the CIC will return the data from the first access. This means that a single call to this function returns the data in the requested SFR. This means two accesses are required but it is more convenient for the caller.

In reality, the function will only work if it is single-stepped in a debugger. As the function uiCIC\_SendMessage() is very simple and contains no message pacing, the SPI messages sent to the CIC61508 would be spaced every 19us. This is illegal as the CIC61508 only accepts SPI messages on a 75us timebase i.e. 75us between message starts. You can see the effects of not having this message separation by running App1 in a debugger as follows:

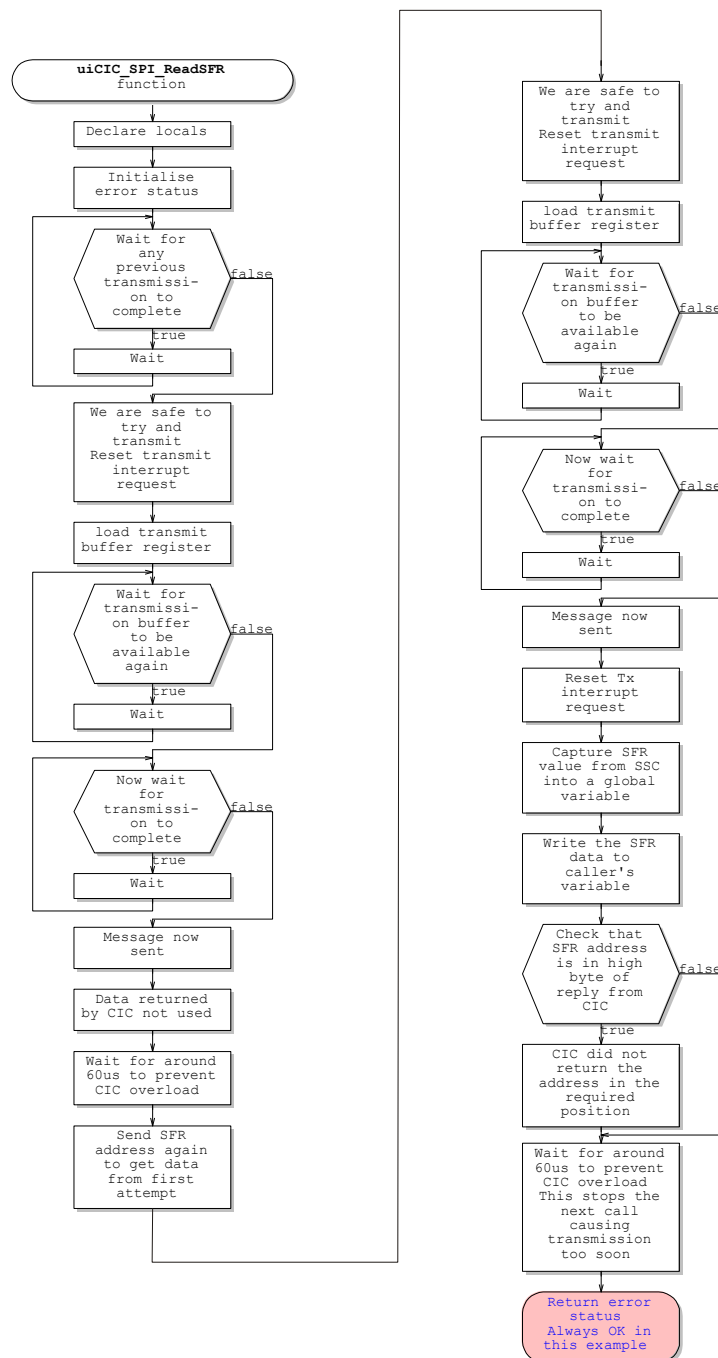
Single step main() up until the while(1) loop. Make sure that the variable uiCIC61508\_ERRORSYSTEM is in a watch window and that the real time updates are enabled. Now run at full speed. iCIC61508\_ERRORSYSTEM will change to 0x2D (DISABLED state). This is because the SSC will be sending messages every 18-19us. This is too fast for the CIC61508 and it will go to DISABLED state. The minimum time permitted between messages is 75us so some sort of pacing mechanism will be required. App2 adds this pacing.

```
// Loop forever
while(0x01U)
{

    /* Read CIC61508 SFR at address 0x07 (ERRORSYSTEM) */
    uiCIC_SPI_ReadSFR(0x07, (uint8 *)&uiCIC61508_ERRORSYSTEM);

}
```

#### Code Listing 3 Continually read the ERRORSYSTEM SFR



**Figure 3** uiCIC\_SPI\_ReadSFR Flowchart

The format of the data returned by the CIC is checked to make sure that the SFR address is returned in the upper byte. As two accesses are made to the CIC, a delay is required to avoid going below the 75us inter-character spacing limit.

It checks the value of the upper byte returned by the CIC and if it is not the SFR address then an error is returned.

The parameters are:

uint8 uiSFR\_addr: address of CIC SFR to read

uint8 \*uiSFR\_data: pointer to variable where SFR value should be placed.

It returns:

CIC\_ErrorType: error code

This function relies on a simple timing loop in vCIC\_InterMessageSpacing().

The effects of having the message pacing can be seen by running App2. Enable the real-time watch window update and single step around this while() a few times. Make sure CIC\_Status and uiCIC61508\_ERRORSYSTEM are in a watch window. uiCIC61508\_ERRORSYSTEM will = 0x78 (NOTREADY state) and CIC\_Status = No\_Error (0).

```
// Loop forever
while(0x01U)
{
    /* Read CIC61508 SFR at address 0x07 (ERRORSYSTEM) */
    CIC_Status = uiCIC_SPI_ReadSFR(0x07, (uint8 *)&uiCIC61508_ERRORSYSTEM);
}
```

Now run at full speed. After a few seconds, press and hold down the CIC61508 reset button to prevent it replying. The value of CIC\_Status will change to "CIC\_Bad\_Reply\_Format" and the value of uiCIC61508\_ERRORSYSTEM will go to 0xFF.

The simple timing loop-based delay used here is clearly unsuitable for real applications as it ties up the whole CPU for 60us at a time. The next example overcomes this by using a GPTA interrupt for message pacing.

## 4 App3: Practicable message exchange with the CIC61508

App3 results in workable CIC61508 SFR read function with error checking and message spacing based on a GPTA0 timer. This is useful for real applications.

### 4.1 Providing A SPI Message Timebase

Many aspects of the CIC61508's SPI interface relies on accurate timing on the part of the master device (TC1782). In `uiCIC_SPI_ReadSFR()` in App3, the second read of the SFR address is delayed by 75us, based on a state machine called from the GPT0\_LTC01 interrupt (SRN22). LTC00 is configured as a 2.5us/count free running timer and LTC01 as a compare register with interrupt on match enabled.

```
void INTERRUPT (GPTA0_SRN22INT) GPTA0_viSRN22(void)
{
    if(GPTA0_SRSS2.B.LTC1)          // LTC1 event (= compare with last timer)
    {
        GPTA0_SRSC2.U = 0x00000002; // reset LTC1 service request bit

        // Call the CIC SPI handler every 75us
        vCIC_SPI_Driver();

        // FLASH LED0
        IO_P5_0 ^= 1;
    }
} // End of function GPTA0_viSRN22
```

**Figure 4** GPTA\_LTC01 SRN22 interrupt handler.

The function `CIC_SPI_Driver()` is located in `CIC_SPI_Examples_App3.c` and is basically a `switch()` statement.

```
/* Determine this action */
switch(CIC_SPI_state)
{

    case Start_SPI_Transmission: /* Background has requested SPI transaction */

        /* Send the next data */
        SSC0_TSRC.B.CLRR = 0x01U; /* Reset transmit interrupt request */
        SSC0_TB.U = uiSPI_Tx_data; /* load transmit buffer register */

        /* Wait for transmission buffer to be available again */
        while(SSC0_TSRC.B.SRR == 0x00U)
        {
            /* Wait */ ;
        }

        /* Now wait for transmission to complete */
        while(SSC0_STAT.B.BSY == 0x01U)
        {
            /* Wait */;
        }

        /* Get message just received from CIC */
        uiSPI_Rx_data = SSC0_RB.U ;

        /*!! Next interrupt will be on the next 75us tick
        * GPTA0_LTC clock is 2.5us so 30 x 2.5 = 75us
        * As the compare register GPTA0_LTCXR01 contains the
        * GPTA0_LTCXR00 count at the last 75us, we just need to
        * add another 30 counts to create the next 75us tick.
        * */

        /* Schedule next 75us interrupt by incrementing compare register by 30 */
        GPTA0_LTCXR01.U += 30 ;

        /* Next time wait for end of 75us slot */
        CIC_SPI_state = CS_Idle ;

        /*!! Note: A new transmission request from the background will
        * not be actioned until 75us has elapsed
        * */

        break;

    case CS_Idle : /* SPI transaction completed */

        /* Schedule the next 75us tick */
        GPTA0_LTCXR01.U += 30 ;
        break ;
}
```

#### **Code Listing 4      CIC\_SPI\_Driver Interrupt Provides Message Timebase**

When the SPI interface is not in use, the variable "CIC\_SPI\_state" has the value 0x00 (typedef enum CS\_Idle). To begin a transmission sequence, the `uiCIC_SPI_ReadSFR()` function sets it equal to "Start\_SPI\_Transmission".



```
/* Load Tx variable with SFR address */  
uiSPI_Tx_data = (uint16) uiSFR_addr ;  
CIC_SPI_state = Start_SPI_Transmission ;
```

#### **Code Listing 5     Beginning a SPI transmission**

Within 75us, the LTC01 interrupt will occur and the case Start\_SPI\_Transmission will execute. This case schedules the next interrupt 75us later by adding the number of LTC00 counts in 75us to the current LTC01 value. As the GPTA0\_LTC clock is 2.5us/count,  $30 \times 2.5 = 75\text{us}$ . Thus we just need to add another 30 counts to create the next 75us tick. The state is changed to CS\_Idle, allowing the uiCIC\_SPI\_ReadSFR() to continue. However as the next interrupt is 75us in the future, any further transmissions will not be auctioned immediately. Thus the 75us message spacing demanded by the CIC61508 is maintained.

```
/* Wait for transmission to complete (next 75us interrupt) */  
while(CIC_SPI_state != CS_Idle)  
{ /* Wait */ ; }
```

The CS\_Idle state continues, with the interrupt being rescheduled for 75us spacing until the next transmission is requested.

In the later examples, timeouts are required for other aspects of CIC61508 operation so this function will be added to considerably.

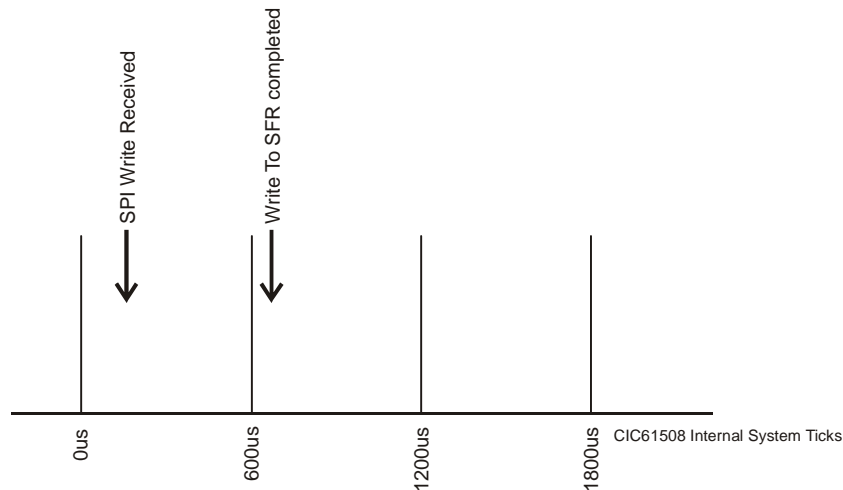
#### **4.1.1     Further Refinement Of SPI Driver**

The CIC\_SPI\_Driver() function here is quite simplistic in that it waits for the SPI transmission to complete before proceeding to read the received data from the CIC61508. In a real system, the reading of the CIC returned data could be done in an additional case that is scheduled around 25us after the transmission is begun.

Alternatively, the SSC RX interrupt could be enabled and the loading of data between the receive and transmit buffers to the globals "uiSPI\_Tx\_data" and "uiSPI\_Rx\_data" could be made there. However for the sake of clarity, these have not been implemented in the example.

## 5 App4: Adding a SFR WRITE function

To write to a CIC61508 SFR requires bit7 of the low byte sent to be set. The data returned by the CIC contains the address of the SFR in the high byte. This needs to be checked. Thus two transactions per write are needed, as with the proper SFR read function developed in App3 and App4 above, (uiCIC\_SPI\_ReadSFR ()). A complication with the write function is that unlike READ, the actual update of the SFR can take up to 600us to complete. This is because internally, the CIC61508 only checks for new write messages every 600us.



Thus to avoid confusion, the write function implemented here inserts a 600us delay before attempting to read back the reply. This means that the CIC\_SPI\_Driver() interrupt function now needs some additional cases "Start\_CS\_Wait\_For\_CIC\_System\_Tick" and "CS\_Wait\_For\_CIC\_System\_Tick". In reality, due to system oscillator tolerances, the system tick time is assumed to be up to 10% slow so we actually use a 660us system tick time.

```
/* Insert a 600us delay */
case Start_CS_Wait_For_CIC_System_Tick:

    /* Schedule a 600us tick period */
    /* In fact to accommodate the potential clock differences between the
     * CIC and the host CPU, we need to allow a 10% safety margin so
     * the delay is really 660us.
     */
    /* Increment LTC01 compare register by 660us/2.5us = 240 counts */
    GPTA0_LTCXR01.U += 264U;

    /* Wait in next state */
    CIC_SPI_state = CS_Wait_For_CIC_System_Tick ;

break;

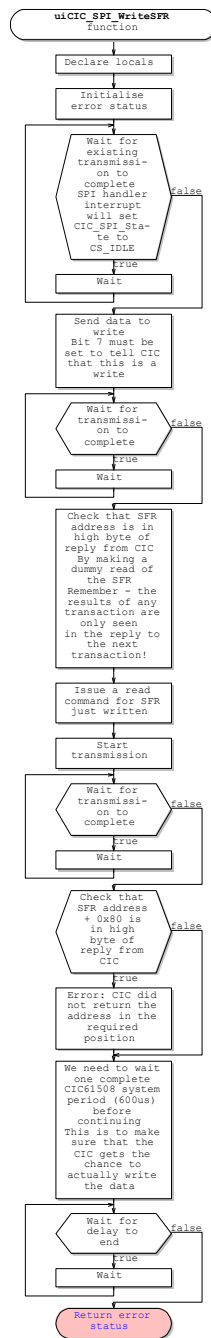
/* Stay in this state until timeout ends */
case CS_Wait_For_CIC_System_Tick :
    /* Go back to idle state */
    CIC_SPI_state = CS_Idle ;

    /* Schedule the next 75us tick */
    GPTA0_LTCXR01.U += 30;

break ;
```

**Code Listing 6     System tick timeout cases**

The sequence of operations for a SFR write are:



It should be noted that the function will hang for 660us while the system tick timeout.

## 6 App5: Using the SFR WRITE function to load the SPI error counter

Now that we have a workable SFR write function, we can perform an useful CIC61508 action. To get the CIC61508 into the ACTIVE state, all the error counters need to be  $\geq 0x40$ . The SPI error count (ERRORCNTCOMM) has be manually loaded. This is done by writing the SPI RESET command to the MODE SFR. The CIC will then load 0x80 into ERRORCNTCOMM. We can then read this back to check that it really is 0x80.

```
/* Now set the SPI Error counter to 0x80 using SPI RESET command */
CIC_Status = uiCIC_SPI_WriteSFR(CSFR_ESTM_MODE, ESTM_SPI_RESET);

/* Now read back the SPI Error counter to see whether it contains 0x80 */
CIC_Status = uiCIC_SPI_ReadSFR(CSFR_ESTM_SPICNT, (uint8
*)&uiCIC61508_ERRORCNTCOMM);
```

### Code Listing 7 Initialising the SPI Error Counter Using The SFR Write Function

It is best to single step this in your debugger to see what happens.

## 7 App6: Entering Secure SPI Mode

Now that we have functions that can read and write CIC61508 SFRs reliably, the next stage is to enter the secure SPI mode. This uses a completely different SPI protocol (still 16-bit messages) and has a special set of entry conditions to ensure that it is not accidentally entered. This security is required as in secure mode, we have the ability to erase and reprogramme the calibration data in the DFLASH. In addition, we can read and write addresses that are not normally accessible, cause the CIC61508 to jump to any address plus make unconditional device resets. Within the context of a safety system, these actions must only be performed intentionally.

### 7.1 Secure SPI Mode Example Messages

Here are some low-level examples of secure SPI mode reads and writes.

*READ PDATA location 0xF001*

F001, 0004, CIC reply: 01F0,<bytedata>00

*READ IDATA location 0x00FE*

00FE, 0002, CIC reply: FE00,<bytedata>00

*READ CODE location 0xA000*

A000, 0008 CIC reply: 00A0,<bytedata>00

*WRITE 0x11 to PDATA location 0xF000*

F000, 1184 CIC reply: 00F0,8411

*WRITE 0x22 to IDATA location 0x00FE*

00FE, 2282 CIC reply: FE00,8222

To program the DFLASH:

1. Write 0xA000 to 0xA01F wordline with data, 0x01 to 0x1F
2. Define the wordline address to be programmed.
3. Send A000,0088
4. Send data to be programmed
5. Send A000, <byte0>88, A000,<byte1>88, ....., A000,<byte31>88
6. Wait for completion of Dflash programming using simple 102ms timeout.

Easy-to-use 'C' functions will be created now to undertake these actions.

## 7.2 How To Get Into Secure SPI Mode

Entering the secure SPI mode requires that the CIC61508 is in either the NOTREADY or DISABLED state (ERRORSYSTEM = 0x78 or 0x2D).

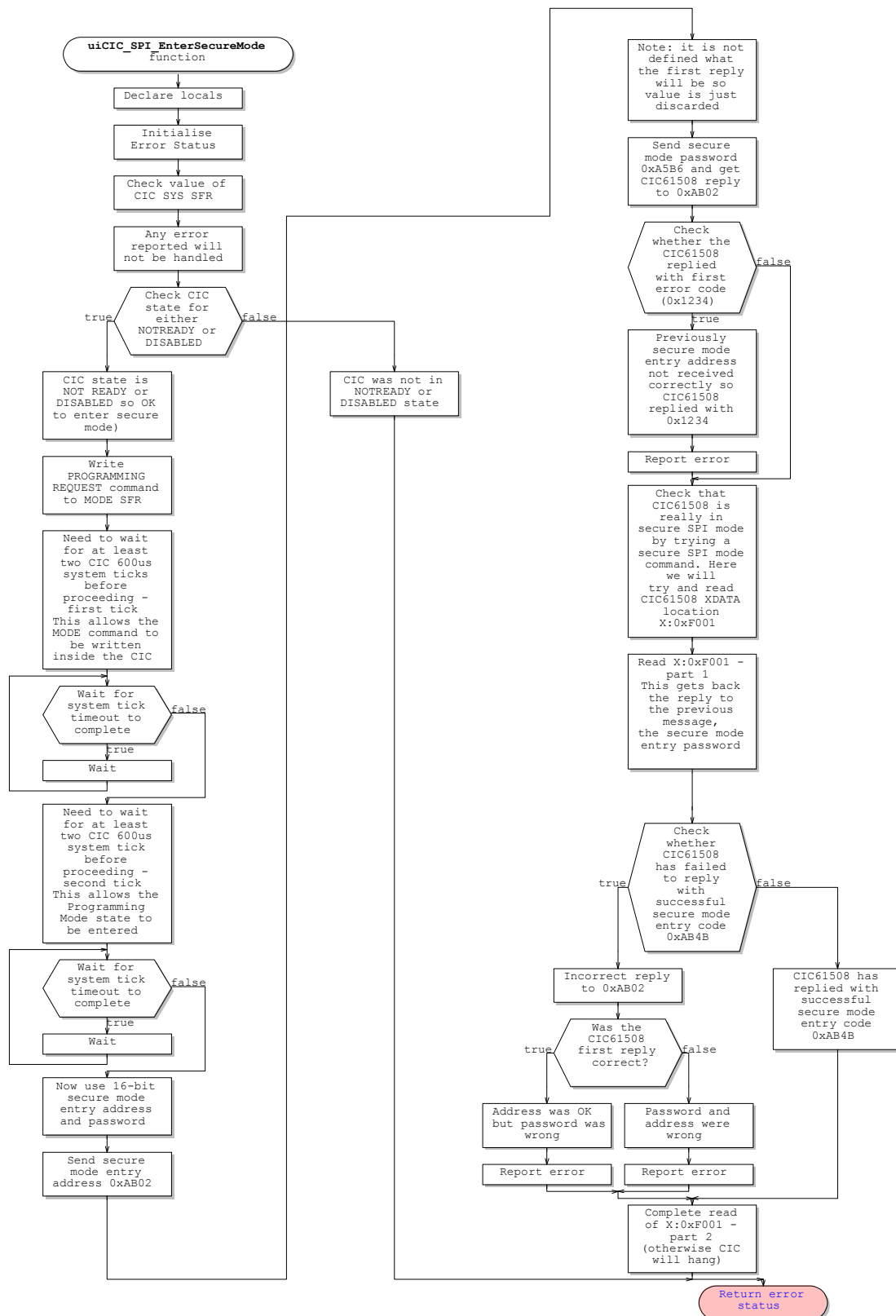
To enter the secure mode, an address and password approach is used as per:

1. TC1782 sends secure mode address "0xAB02"
2. CIC61508 replies with random data (this is OK!)
3. TC1782 sends secure mode password "0xA5B6"
4. CIC61508 replies with "0xAB02"
5. TC1782 sends any secure mode address
6. CIC61508 replies with "0xAB4B"
7. TC1782 sends a read command

We are now in secure SPI mode.

Note that the steps 5 and 7 are not obvious and are required because the reply from the CIC61508 to any message is in fact the reply to the previous one, just as in standard SPI mode. Thus to get the reply to the secure password 0xA5B6, we make a dummy read of the XDATA location 0xF001. The 0xA5B6 is returned after the sending of the 0xF001 address so this where the 0x1234 and 0xAB4B replies are checked for. However to stop the CIC61508 getting out of step with the TC1782, the second part of the read of X:0xF001 is then sent. Secure SPI messages to the CIC61508 must always be made in pairs. The timing requirements for messages is the same i.e. 75us minimum between transactions. However there is no need to ever wait 600us between address write and read operations as the normal CIC61508 system is disabled in secure mode.

The exact details of this are set out in the CIC61508 User Manual v1.0, section 2.8. What follows here is a practical implementation that works.



**Figure 5 Entering Secure SPI Mode**



### 7.3 First Steps In Secure SPI Mode

Now we are in secure SPI mode, we can try some things are that not possible in standard SPI mode. The first thing to do is read the CRC8 of the CIC61508 internal ROM at location C:0x2FFF. In the App6 main(), this is implemented using the basic uiCIC\_SPI\_SendMessage() function.

```
/* Try to read CRC8 value from CIC61508 ROM at C:0x2FFF */
secure SPI mode READ function */

/* Send address in secure mode */
uiDummy = uiCIC_SPI_SendMessage(0x2FFFU) ; /* Throw away reply */

/* Send READ command */
/* Parameters: (READ Command | (CODE Memory Space & 0x00FFU)) */
uiDummy = uiCIC_SPI_SendMessage(0x0000U | (0x0008U & 0x00FFU)) ;
/* Throw away reply */

/* Note: dummy ought to contain Address previously sent (0x2FFF) */

/* Send address in secure mode again */
/* This will cause the data requested above to be sent by CIC */
uiCIC_value = uiCIC_SPI_SendMessage(0x2FFFU) ; /* Reply contains value at C:0x2FFF
*/

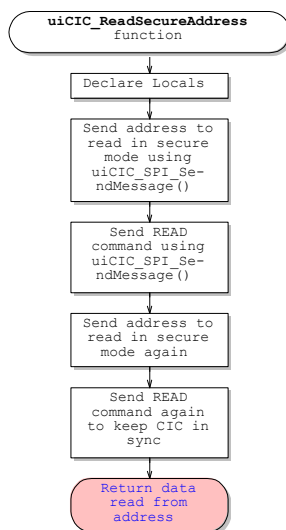
/* For the SAKCIC61508 ROM device, the returned value = 0xDA
* This this the device CRC8.
*/

/* Send READ command again to keep CIC61508 in sync */
uiDummy = uiCIC_SPI_SendMessage(0x0000U | (0x0008U & 0x00FFU));
```

#### Code Listing 8 Reading the CIC61508 CRC8 at C:0x2FFF in secure SPI mode

To make a read in secure mode, the messages required are:

1. Send 16bit address message (0x2FFF)
2. CIC61508 returns random data
3. Send 16 bit READ command message that specifies which memory are the address is in.
4. CIC61508 returns 0x2FFF
5. Send 16bit address message (0x2FFF) again
6. CIC61508 returns the value



**Figure 6 Read any address sequence**

The secure mode READ command uses the lower byte of the 16-bit transmission to specify the memory space in which the address lies. There are 3 memory spaces, each with an unique memory space code. Here we are reading the CODE space so the memory space code is 0x0008.

```

/* Secure SPI Mode Commands */
#define Secure_WRITE 0x0080U
#define Secure_READ 0x0000U

#define CODE 0x08U // Access code space C:0x000 - 0x2FFF
#define XDATA 0x04U // Access xdata space X:0xF000- 0xF1FF
#define IDATA 0x02U // Access idata space I:0x0000 - 0x00FF

/* Secure mode actions */
#define Erase_DFLASH 0x03U // Erase the DFLASH
#define Program_DFLASH 0x05U // Program a wordline in the DFLASH
#define Jump_Address 0x06U // Cause the MIL to call an absolute address
#define Reset_CIC61508 0x07U // Cause the MIL to RESET
  
```

**Code Listing 9 Secure SPI Command Set**

In reality, we need a simple functions that will perform the necessary steps to read and write a secure mode address and this will be done in App7 that follows.

## **8 App7: Using Secure SPI Mode Features**

Now we have the means to enter secure SPI mode and know how to read data, we can create a simple function to read any address in any CIC61508 memory space. We can then go on to create a function to write any address or invoke a secure SPI mode action (reset, jump, erase DFLASH etc.)

### **8.1 CIC61508 Memory Spaces**

The CIC61508 has three memory spaces:

CODE: ("C:") this is where the CIC61508 ROM is located and the DFLASH calibration data area

XDATA: ("X:") this is where the CIC61508 stores non-critical variables

IDATA: ("I:") This is where the CIC6158 SFRs are located and other critical data items.

Note: port pins are not directly accessible from secure SPI mode. If you need to read or write specific CIC61508 port pins, you will need to use the "Applet" concept, covered in a separate application note.

Some useful secure SPI mode addresses are:

C:0x1000: Secure SPI mode SSC interface

C:0x2000: Reset hook (legacy use only)

C:0x2FFF: CRC8 of CIC61508 program ROM

X:0xF06A: BIST memory test results

X:0xF061: BIST opcode test results

I:0x87: ERRORSYSTEM SFR absolute address

Addresses of data items in the DFLASH can be gleaned from the CIC61508 Buildsheets spreadsheet.

## 8.2 Creating A Function To Read Any Secure Mode Address

In App6 the steps required to read a secure mode address were covered. In this App, these have been incorporated into a new function:

```
uint16 uiCIC_ReadSecureAddress(uint16 uiAddress, uint16 uiMspace)
```

The uiMspace parameter specifies which memory space in the CIC61508 is to be read, according to:

```
#define CODE          0x08U // Access code space C:0x000 - 0x2FFF
#define XDATA         0x04U // Access xdata space X:0xF000- 0xF1FF
#define IDATA         0x02U // Access idata space I:0x0000 - 0x00FF
```

The function consists of the following steps:

```
uint16 uiCIC_ReadSecureAddress(uint16 uiAddress, uint16 uiMspace)
{
    /* Declare Locals */
    uint16 uiDummy ;
    uint16 uiReadData ;

    /* Send address to read in secure mode using uiCIC_SPI_SendMessage() */
    uiDummy = uiCIC_SPI_SendMessage(uiAddress) ;

    /* Send READ command using uiCIC_SPI_SendMessage() */
    uiDummy = uiCIC_SPI_SendMessage(Secure_READ | (uiMspace & 0x00FFU)) ;

    /* Send address to read in secure mode again */
    uiReadData = uiCIC_SPI_SendMessage(uiAddress) ;

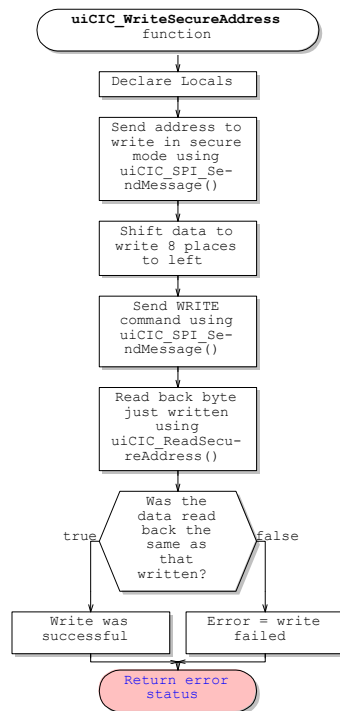
    /* Send READ command again to keep CIC in sync */
    uiDummy = uiCIC_SPI_SendMessage(Secure_READ | (uiMspace & 0x00FFU)) ;

    /* Return data read from address */
    return(uiReadData) ;
}
```

### Code Listing 10 The uiCIC\_ReadSecureAddress() Function

### 8.3 Writing Of Secure Mode Addresses

In the App7 main(), we use secure SPI mode to perform some commonly-used commands via proper secure mode read and write functions, located in CIC\_SPI\_Examples\_App7. The first action is writing to a memory location. The low-level sequence of operations to make a write are:



**Figure 7 Write Any Address Sequence**

Basic example of secure mode writes are:

```

/* Write the location X:0xF0FF */
CIC_Status = uiCIC_WriteSecureAddress(0xAAU, 0xF0FFU, XDATA);

/* Read the data back from X:0xF0FF */
uiTestValue = uiCIC_ReadSecureAddress(0xF0FFU, XDATA);
  
```

#### Code Listing 11 Writing a XDATA Location

From the foregoing, a simple function is created to write secure mode addresses:

```

CIC_ErrorType uiCIC_WriteSecureAddress(uint8 uiData, uint16 uiAddress, uint16
uiMspace)
  
```

This uses the basic `uiCIC_SPI_SendMessage()` function to send the write command and data. It also uses the previously created `uiCIC_ReadSecureAddress()` to check that the write was successful. The memory space parameters are identical to those in the read function except that the CODE space has no effect as this is not writable (see later sections for how to write to the DFLASH in the CODE space).

```
CIC_ErrorType uiCIC_WriteSecureAddress(uint8 uiData, uint16 uiAddress, uint16 uiMspace)
{
    /* Declare Locals */
    uint16 uiDummy ;
    uint16 uiReadData ;
    uint16 uiWriteData ;
    CIC_ErrorType uiResult;

    /* Send address to write in secure mode using uiCIC_SPI_SendMessage() */
    uiDummy = uiCIC_SPI_SendMessage(uiAddress) ;

    /* Shift data to write 8 places to left */
    uiWriteData = ((uint16)uiData << 8U) ;

    /* Send WRITE command using uiCIC_SPI_SendMessage() */
    uiDummy = uiCIC_SPI_SendMessage(uiWriteData | Secure_WRITE |
                                    (uiMspace & 0x00FFU)) ;

    /* Read back byte just written using uiCIC_ReadSecureAddress() */
    uiReadData= uiCIC_ReadSecureAddress(uiAddress, (uiMspace & 0x00FFU));

    /* Was the data read back the same as that written? */
    if(uiReadData == uiData)
    {
        /* Write was successful */
        uiResult = CIC_No_Error;
    }
    else
    {
        /* Error = write failed */
        uiResult = CIC_SecureWriteError;
    }

    /* Return error status */
    return(uiResult);
}
```

**Code Listing 12 The uiCIC\_WriteSecureAddress() Function**

Other forms of secure mode write cause specific actions. One example is using the write to cause a CIC reset using the “Reset CIC61508” action (0x07). Actions that are supported are:

```
/* Secure mode actions */
#define Erase_DFLASH      0x03U // Erase the DFLASH
#define Program_DFLASH    0x05U // Program a wordline in the DFLASH
#define Jump_Address      0x06U // Cause the CIC to call an absolute address
#define Reset_CIC61508    0x07U // Cause the CIC to RESET
```

To send an action request, the message is of the form:

```
uiCIC_SPI_SendMessage (0x0080 | 0x0007)   i.e.

uiCIC_SPI_SendMessage (Secure_WRITE | Reset_CIC61508)
```

### Code Listing 13 Cause A CIC61508 Reset

In reality, this would appear as:

```
/* Cause a CIC61508 reset */

/* Send dummy address */
uiCIC_SPI_SendMessage(0x0000U) ;

/* Send a reset command */
uiCIC_SPI_SendMessage(Secure_WRITE | Reset_CIC61508) ;

/* Wait for CIC to restart */
```

### Code Listing 14 Causing CIC61508 Reset

The reset command will cause the CIC61508 to behave as it does at power-on. This means that it will run all its startup self-tests. This takes up to 60ms to complete. Any attempt to communicate with the CIC during this time will get a reply of 0xFF. This is because its MRST pin is inactive and floating high. The SYSDIS LEDs on the board will blink once as the CIC restarts. Once this has completed, the CIC61508 will be back in standard SPI mode.

The 60ms reset delay will be incorporated into the CIC\_SPI\_Driver() interrupt function in the next example, along with a proper CIC61508 reset function.

## 9 App8: Erasing And Programming The DFLASH

This example adds the ability to read the entire DFLASH into a RAM array, erase the DFLASH and re-program the array contents back again. It also shows standard and secure SPI mode reset commands.

Note: if this example fails, it is likely that the DFLASH in your CIC61508 will be left in an erased state or corrupted. In both cases, you will need to use either the TARDISS or PRO-SIL TestBench tools to recover the situation. Alternatively, App9 or App10 of this document could be used.

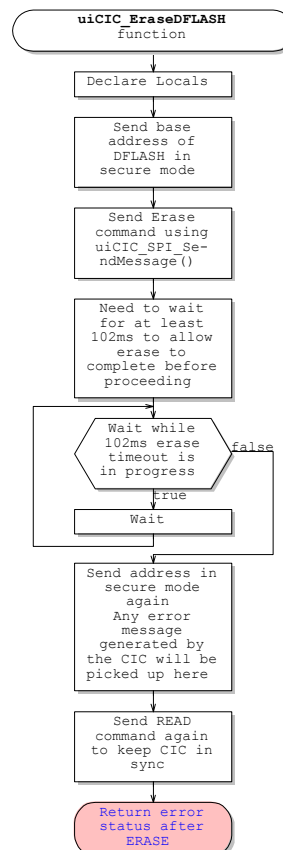
### 9.1 Erasing The DFLASH

Erasing the DFLASH relies on the CIC61508's bootROM FLASH erase function but with a wrapper function accessed from the secure SPI mode.

The sequence of operations to erase the DFLASH are:

1. Enter secure SPI mode
2. Send the base address of the DFLASH (X:0xA000)
3. Write the erase command Secure\_WRITE | Erase\_DFLASH . This is in reality 0x80 | 0x03.
4. Wait for 102ms (see CIC61508 Data Sheet)
5. Send the DFLASH base address again
6. Send a read command of the CODE memory space. This will get back any error status from the erase function.

This can be summarized as:



**Figure 8 Erase DFLASH Sequence**

From this flowchart, the function `CIC_ErrorType uiCIC_EraseDFLASH(void)` has been created.



### 9.1.1 Erase Function Error Codes

The `uiCIC_EraseDFLASH()` erase function is makes use of the CIC61508 bootROM erase function that is built in the CIC61508. This is able to return a number of errors. In addition, the CIC61508 itself can identify a number of errors related to the addresses supplied. The error codes returned are listed below.

```
#define FLASH_TYPE1_ERASE_FAILED    0x0200U
#define Address_Not_DFLASH_Base    0x0300U
#define Address_Above_DFLASH       0x0400U
#define Address_Below_DFLASH       0x0400U
```

#### Code Listing 15 Possible Erase Error Codes

The error “FLASH\_TYPE1\_ERASE\_FAILED” is generated by the CIC61508 bootROM and if this occurs, it is likely that an attempt was made to erase the DFLASH while a previous attempt was in still progress i.e. no erase delay was used. If the functions given in `CIC_SPI_Examples_App8.c` are used unmodified, then none of these errors should be encountered.

### 9.1.2 Creating The Erase Delay

To create the 102ms erase delay, an extra pair of cases have been added to the `CIC_SPI_Driver` function:

```
/* Setup a 102ms delay to allow erase to complete */
case Start_CS_Wait_For_ERASE_Cycle :

    /* Schedule the next tick 102ms later */
    /* At 2.5us/count, we need to add 102000/2.5 = 40800
     * to the compare register */
    GPTA0_LTCXR01.U += 40800;

    /* Go to wait for end of erase time state */
    CIC_SPI_state = CS_Wait_For_ERASE_Cycle ;

break ;

case CS_Wait_For_ERASE_Cycle : /* End of ERASE cycle */

    /* Go back to idle state */
    CIC_SPI_state = CS_Idle ;

    /* Schedule the next 75us tick */
    GPTA0_LTCXR01.U += 30;

break ;
```

#### Code Listing 16 Creating the 102ms erase delay

### 9.1.3 Using The uiCIC\_EraseDFLASH() Erase Function

In the main.c for example App8, the entire DFLASH is read into a local array using the secure mode read command in a for() loop.

```
/* Read the entire DFLASH contents into an array */
for(uiI = 0x00; uiI < DFLASH_Length ; uiI++)
{
    /* Read the DFLASH byte-by-byte into array */
    uiDFLASH_Image[uiI] = uiCIC_ReadSecureAddress((uiI + DFLASH_Base_Addr), CODE);
}
```

#### Code Listing 17 Reading the entire DFLASH into an array prior to erasing

When running in a debugger, if you put uiDFLASH\_Image into a watch window and you should see the contents of the DFLASH. For most CIC61508 configurations, this should start with “0xF8, 0x0D, 0xE9, 0x74, 0x01.....”

The DFLASH is then erased:

```
CIC_Status = uiCIC_EraseDFLASH();
```

After erasure, the DFLASH will contain only zeros, this being the usual state of erased Infineon FLASH memory.

The example then goes on to program the previously read DFLASH contents back into the CIC61508. This covered in the next section.

## 9.2 Programming The DFLASH

Programming of the DFLASH relies on the CIC61508's bootROM FLASH programming function but with a wrapper function accessed from the secure SPI mode.

The CIC61508 DFLASH is arranged in "wordlines" of length 32 bytes. This represents the smallest programmable area. Thus to program the entire 0x1000 bytes of the DFLASH we have to break it into chunks of length 0x20.

At the lowest level, the bytes in the wordline are sent one-by-one as part of a write-to-CODE space command. The programme wordline sequence is:

1. Get the first byte of the next wordline from an array
2. Move it to the high byte of the message
3. OR in the WRITE to CODE command to make a 16-bit value
4. Send base address current wordline
5. Send programme DFLASH command
6. Send the 16 value composed of <data><write, CODE>
7. Repeat from step 1 another 31 times until all wordline bytes have been sent
8. Wait 2.6ms for programming to complete (see CIC61508 Data Sheet)
9. Make a dummy read to see if any programming errors have been reported by the CIC61508.
10. Add 0x20 to the wordline base address
11. Get the next wordline
12. Go back to step 1

As with the erase function, two extra cases have been added to the CIC\_SPI\_Driver() function to allow the 2.6ms programming delay to be inserted:

```
/* Set up a 2.6ms delay to allow programming to complete */
case Start_CS_Wait_For_PROG_Cycle :

    /* Schedule the next tick 2.6ms later */
    /* At 2.5us/count, we need to add 2600/2.5 = 1040
     * to the compare register */
    GPTA0_LTCXR01.U += 1040U;

    /* Go to wait for end of programming time state */
    CIC_SPI_state = CS_Wait_For_PROG_Cycle ;

break ;

/* End of 2.6ms programming delay */
case CS_Wait_For_PROG_Cycle :

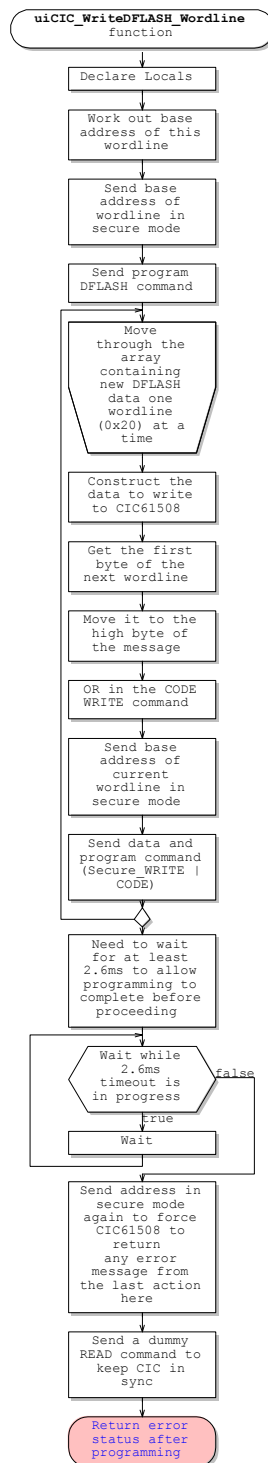
    /* End of Programming cycle */
    CIC_SPI_state = CS_Idle ;

    /* Schedule the next 75us tick */
    GPTA0_LTCXR01.U += 30;

break ;
```

**Code Listing 18 Creating the 2.6ms programming delay**

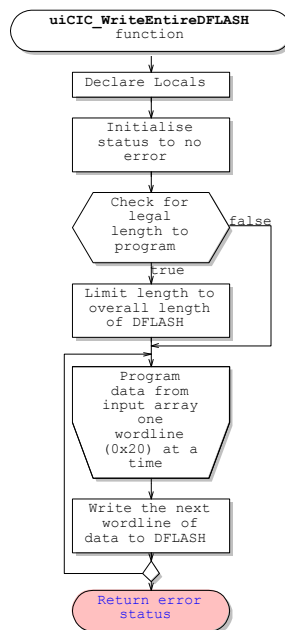
The wordline programming procedure may be summarized as:



**Figure 9 DFLASH Wordline programming procedure**

The wordline programming is undertaken by the function `uiCIC_WriteDFLASH_Wordline()`. This receives a pointer to the 32bytes of the current wordline, the base address of the DFLASH and the offset of the current wordline from the base of the DFLASH.

In the `main()` for App8, we actually want to programme the entire DFLASH with the data previous read from it, prior to the erase. This means that a wrapper function is required to break up the 0x1000 DFLASH image into 0x20 byte chunks. This is done by `uiCIC_WriteEntireDFLASH()`.



**Figure 10 Breaking up the 0x1000 DFLASH image into 32-byte wordlines for programming**

It is called as per:

```

/* Program the image array back into DFLASH */
CIC_Status = uiCIC_WriteEntireDFLASH(uiDFLASH_Image, DFLASH_Length);

```

As a confidence check, the second location of the DFLASH is read to make sure it contains something other than 0x00, the value returned by an erased DFLASH. In most cases, the second DFLASH byte is 0xF8.

```

/* Read a DFLASH location (should return 0xF8) */
uiTestValue = uiCIC_ReadSecureAddress((DFLASH_Base_Addr + 1), CODE);

```

### 9.3 Exiting Secure SPI Mode After DFLASH Reprogramming

After reprogramming, the CIC61508 will still be in secure SPI mode. To get back to standard SPI mode, a Reset action command must be sent.

If the SYSDIS\_C LED on the board is not illuminated after this then a problem has occurred during programming. To fix this, you will need to use the TARDISS or PRO-SIL TestBench tools to restore the DFLASH contents. If you are using a FLASH CIC61508, the CIC61508.hex file included in the CIC61508 directory can be blown into FLASH using the FLOAD programmer. Alternatively, you can use the final example, App9 to program a DFLASH image held in the TC1782 ROM into the CIC61508 DFLASH.

## 10 App9: Programming The DFLASH From A TARDISS Export

This example takes a DFLASH image created by the TARDISS (or PRO-SIL TestBench) tools and programs it into DFLASH. The TARDISS and PRO-SIL TestBench tools are able to import CIC61508 DFLASH data from a spreadsheet (“BuildSheet”) or a real device and then export them as a compilable C file, containing the calibration data in the form of a C const array.



**Figure 11 PRO-SIL TestBench NVM Tables Tab**

This kind of operation may be used on a production line where the CIC’s DFLASH data is programmed by a production line test program running on the TC1782.

The DFLASH image is held in the file CIC\_DFLASH.C and in a const array “DFLASH\_data”.

```
/* Autogenerated File Template */

/* Include the necessary include files */
#include "CIC_DFLASH.h"

unsigned char CONST CIC_DFLASH_data[CIC_DFLASH_SIZE] =
{
    0x00, 0xF8, 0x0D, 0xE9, 0x74, 0x01, 0xB7, 0x42,
    0xA6, 0x3B, 0x02, 0xEE, 0x1B, 0xFF, 0x62, 0x03, ....
}
```

### Code Listing 19 DFLASH Image Export From TARDISS/PRO-SIL TestBench

In the main() in App9, the programming is now carried via:

```
/* No error occurred so we are in secure SPI mode */
CIC_Status = uiCIC_EraseDFLASH();

/* Program DFLASH data exported from the TARDISS or PRO-SIL TestBench tools */
CIC_Status = uiCIC_WriteEntireDFLASH(CIC_DFLASH_data, DFLASH_Length);
```

## **10.1 Restoring Corrupted DFLASH**

App9 can be used to re-programme corrupted DFLASH with a standard image that will function with PRO-SIL TriCore. To do this, just load the App9 into the debugger, reset the CIC61508 and run the program.

## 11 App10: Typical Usage Example

The SAKCIC61508 is supplied with no data in the DFLASH. This will cause it to go straight to the DISABLED mode at power-up. During the manufacture of a board containing the SAKCIC61508, the DFLASH must be programmed with a valid DFLASH calibration data set. This dataset usually takes the form of a C const array produced as an export from the TARDISS NVM Tables menu, as covered in App9 above.

This application is slightly different in that it checks for a valid data set in the DFLASH. If the data is invalid, then it proceeds to program the correct data set into the DFLASH automatically.

```
/* Check that the upper half of the DFLASH contents is the inverse
 * of the lower half */

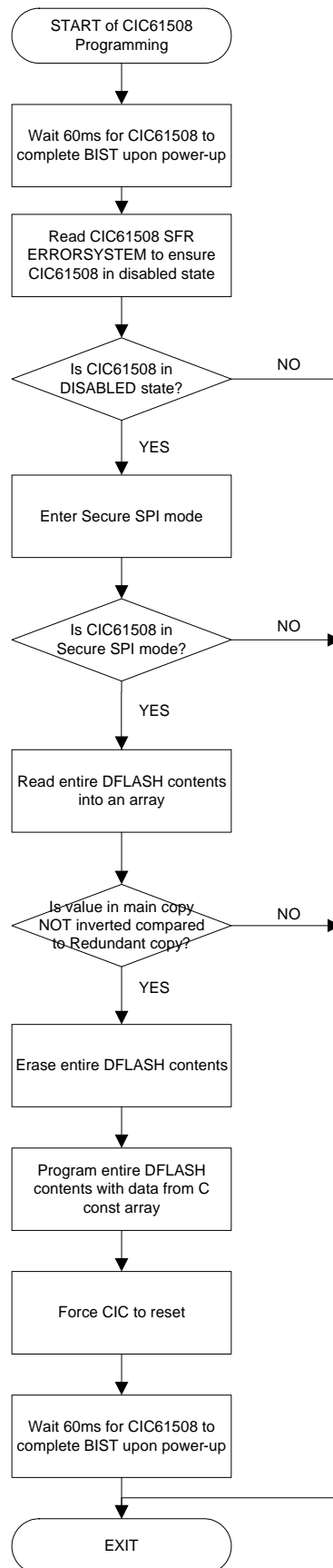
/* Set initial error status */
uiCIC_DFLASH_Error = DFLASH_Integrity_OK;

/* Loop through all locations */
for(uiI = 0x00; uiI < DFLASH_Length/2U ; uiI++)
{
    if(uiDFLASH_Image[uiI] != (uiDFLASH_Image[uiI + DFLASH_Length/2U] ^ 0xFF))
    {
        /* Upper half not the inverse of lower half so flag an error */
        uiCIC_DFLASH_Error = DFLASH_Integrity_Error;
    }
}
```

It then resets the CIC61508 and waits 60ms for it to restart before exiting.

This strategy could also be used in an application where the DFLASH is checked before the main system starts.





**Figure 12 Real Application Example – Programming Corrupted Or Blank DFLASH**

## 12 Troubleshooting

The table below lists some common problems encountered when using the example applications.

**Table 1 Troubleshooting**

Symptom	Cause/Workaround
CIC61508 SPI read functions always return 0xFF	CIC61508 not running 2k2 pull-resistor not fitted to CIC61508 MRST pin
CIC61508 SPI functions report errors	CIC61508 in standard SPI mode when secure SPI mode commands are sent. CIC61508 in secure SPI mode when standard SPI mode commands are sent.
CIC61508 is always in DISABLED mode.	It is likely that the DFLASH has been erased or corrupted. Run App10 to restore it.

## 13 Appendices

### 13.1 Appendix 1 - CIC61508 SPI Functions

All the functions in the Hitex CIC61508 SPI driver library are listed here, along with a brief description.

#### 13.1.1 uint16 uiCIC\_SPI\_SendMessage(uint16 uiTx\_data)

Description	Lowest level SPI driver. Sends 16bit pattern to CIC via SPI
Input Parameters	uiTx_data: Word to send via SPI
Output Parameters	None
Return	Word received from CIC via SPI. Data is the reply to the preceding message.
Preconditions	None
Post Conditions	None
attention	None

#### 13.1.2 void vCIC\_SPI\_Driver(void)

Description	SPI driver to generate chipselect and drive SPI peripheral.
Input Parameters	None
Output Parameters	None
Return	None
Preconditions	None
Post Conditions	None
attention	Called from SPI transmission timebase interrupt (>=75us) off GPTA0 LTC01. Next 75us interrupt scheduled from here.

#### 13.1.3 CIC\_ErrorType uiCIC\_SPI\_ReadSFR(uint8 uiSFR\_addr, uint8 uiSFR\_data)

Description	Reads a CIC SFR by sending the address of that SFR and the address of a variable in which to put the SFR value. The operation is repeated so that the CIC will return the data from the first access. This means that a single call to this function returns the data in the requested SFR. This means two accesses are required but it is more convenient for the caller
details	The format of the data returned by the CIC is checked to make sure that the SFR address is returned in the upper byte. As two accesses are made to the CIC, a delay is required to avoid going below the 75us inter-character spacing limit.
Input Parameters	uint8 uiSFR_addr: address of CIC SFR to read uint8 uiSFR_data: pointer to variable where SFR value should be placed.
Output Parameters	None
Return	CIC_ErrorType: error code
Preconditions	CIC61508 just powered-up.
Post Conditions	-
attention	This checks the value of the upper byte returned by the CIC and if it is not the SFR address then an error is returned.

#### 13.1.4 CIC\_ErrorType uiCIC\_SPI\_WriteSFR(uint8 uiSFR\_addr, uint8 uiSFR\_data)

Description	Write a CIC61508 SFR in standard SPI mode
Input Parameters	uiSFR_addr Address of SFR
Input Parameters	uiSFR_data Value to write to SFR
Output Parameters	None
Return	Error status
Preconditions	None

Post Conditions      None  
attention              CIC must already be in standard SPI mode. Does not check that write was successful

### 13.1.5      **CIC\_ErrorType uiCIC\_SPI\_EnterSecureMode(void)**

Description              Put the CIC61508 SFR into secure SPI mode  
Input Parameters        None  
Input Parameters        None  
Output Parameters      None  
Return                    Error status after operation  
Preconditions            None  
Post Conditions         None  
attention                CIC must be in standard SPI mode.

### 13.1.6      **uint16 uiCIC\_ReadSecureAddress(uint16 uiAddress, uint16 uiMspace)**

Description              Reads byte from an address in CIC secure SPI mode  
Input Parameters        uiAddress: Address in CIC to read  
Input Parameters        uiMspace: Code specifying 8051 memory space to access  
Output Parameters      None  
Return                    Word received from CIC via SPI. Byte read is in LSB  
Preconditions            None  
Post Conditions         None  
attention                CIC must already be in secure SPI mode. Makes 4 accesses to SPI  
attention                Needs to check that previous transmission was returned

### 13.1.7      **CIC\_ErrorType uiCIC\_WriteSecureAddress(uint8 uiData, uint16 uiAddress, uint16 uiMspace)**

Description              Writes a byte to an address in CIC secure SPI mode  
Input Parameters        uiAddress: Address in CIC to write  
Input Parameters        uiMspace: Code specifying 8051 memory space to access  
Output Parameters      None  
Return                    Byte received from CIC via SPI.  
Preconditions            None  
Post Conditions         None  
attention                CIC must already be in secure SPI mode. Makes 4 accesses to SPI  
attention                Needs to check that previous transmission was returned

### 13.1.8      **CIC\_ErrorType uiCIC\_EraseDFLASH(void)**

Description              Erases the entire CIC61508 DFLASH  
Input Parameters        None  
Output Parameters      None  
Return                    Error status resulting from ERASE attempt  
Preconditions            None  
Post Conditions         None  
attention                CIC must already be in secure SPI mode.

### 13.1.9 **CIC\_ErrorType uiCIC\_WriteDFLASH\_Wordline(uint8 uiInputData, uint16 uiWordlineBaseAddr, uint16 uiOffset)**

Description	Write a wordline in the CIC61508 DFLASH
Input Parameters	uiInputData: Pointer to array contain wordline data
Input Parameters	uiWordlineBaseAddr: Base address of the DFLASH
Input Parameters	uiOffset: Offset of the current wordline in multiples of 0x20
Output Parameters	None
Return	Error status resulting from programming attempt
Preconditions	None
Post Conditions	None
attention	CIC must already be in secure SPI mode.

### 13.1.10 **CIC\_ErrorType uiCIC\_WriteEntireDFLASH(uint8 uiInputData, uint16 uiLength)**

Description	Write the DFLASH with large block of data
Input Parameters	uiInputData: Pointer to array contain data destined for DFLASH
Input Parameters	uiLength: Length of data to write (must be a multiple of 0x20)
Output Parameters	None
Return	Error status resulting from programming attempt
Preconditions	None
Post Conditions	None
attention	CIC must already be in secure SPI mode.

### 13.1.11 **CIC\_ErrorType uiReset\_CIC(void)**

Description	Cause the CIC61508 to reset using new command
Input Parameters	None
Input Parameters	None
Input Parameters	None
Output Parameters	None
Return	Error status after operation
Preconditions	None
Post Conditions	None
attention	CIC must be in secure SPI mode.

### 13.1.12 **CIC\_ErrorType uiReset\_CIC\_Immediate(void)**

Description	Cause the CIC61508 to reset in standard SPI mode
Input Parameters	None
Input Parameters	None
Input Parameters	None
Output Parameters	None
Return	Error status after operation
Preconditions	None
Post Conditions	None
attention	CIC must be in standard SPI mode. Uses new WakeUp timer immediate reset command.

**13.1.13 uint16 uiCIC\_SPI\_FastWriteSFR(uint8 uiSFR\_addr, uint8 uiSFR\_data)**

Description	Write a CIC61508 SFR in standard SPI mode but with no read-back or error checking.
Input Parameters	uiSFR_addr: Address of SFR
Input Parameters	uiSFR_data: Value to write to SFR
Output Parameters	None
Return	Last data from SPI RX buffer
Preconditions	None
Post Conditions	None
attention	CIC must already be in standard SPI mode.

## 13.2 Appendix 2 - CIC61508 SPI Functions Memory Usage

**Table 2 ROM Requirement For SPI Functions**

<b>Function</b>	<b>Code size (hex)</b>
uiCIC_EraseDFLASH	0x0000003a
uiCIC_ReadSecureAddress	0x00000022
uiCIC_SPI_EnterSecureMode	0x0000009e
uiCIC_SPI_FastWriteSFR	0x00000034
uiCIC_SPI_ReadSFR	0x00000032
uiCIC_SPI_SendMessage	0x00000030
uiCIC_SPI_WriteSFR	0x00000062
uiCIC_WriteDFLASH_Wordline	0x00000060
uiCIC_WriteEntireDFLASH	0x0000002c
uiReset_CIC	0x0000001e
uiReset_CIC_Immediate	0x00000026
vCIC_SPI_Driver	0x000000e4
<b>Total ROM Size For All SPI Driver Functions:</b>	<b>0x3D8 (984 bytes)</b>

[www.hitex.com](http://www.hitex.com)

Published by Hitex (UK) Ltd.