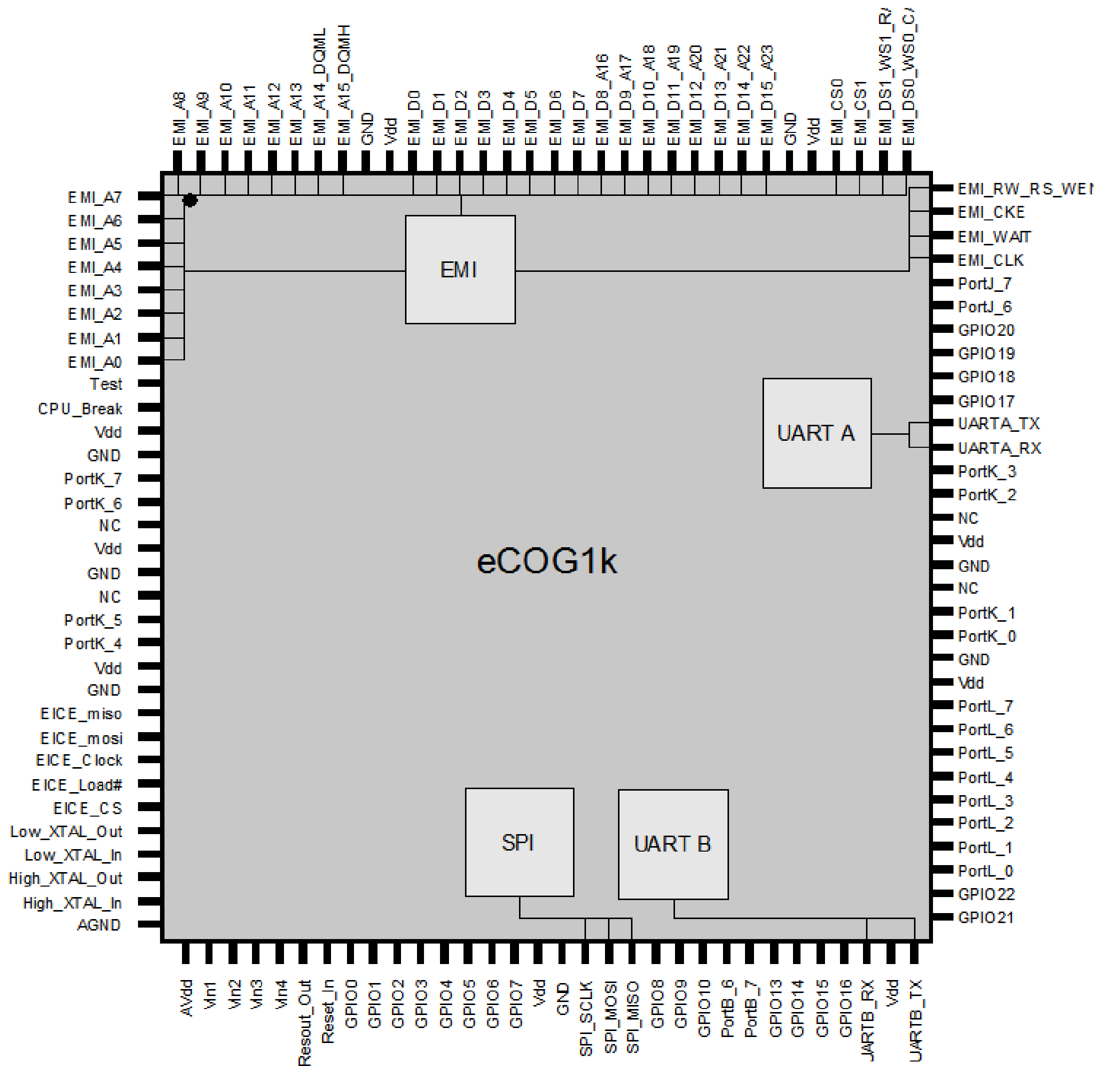
cyan www.cyantechnology.com

ESSENTIAL TEAMWORK FOR WINNING DESIGNS

AN032 – Using the Flash Information Block for Small EEPROM Replacement

Version 1.0

This application note demonstrates how the eCOG1k flash memory information block can be used as a non-volatile storage area for application data with full read-write access.



Confidential and Proprietary Information

©Cyan Technology Ltd, 2008

This document contains confidential and proprietary information of Cyan Technology Ltd and is protected by copyright laws. Its receipt or possession does not convey any rights to reproduce, manufacture, use or sell anything based on information contained within this document.

Cyan Technology™, the Cyan Technology logo and Max-eICE™ are trademarks of Cyan Holdings Ltd. CyanIDE® and eCOG® are registered trademarks of Cyan Holdings Ltd. Cyan Technology Ltd recognises other brand and product names as trademarks or registered trademarks of their respective holders.

Any product described in this document is subject to continuous developments and improvements. All particulars of the product and its use contained in this document are given by Cyan Technology Ltd in good faith. However, all warranties implied or expressed, including but not limited to implied warranties of merchantability, or fitness for purpose, are excluded.

This document is intended only to assist the reader in the use of the product. Cyan Technology Ltd shall not be liable for any loss or damage arising from the use of any information in this guide, any error or omission in such information, or any incorrect use of the product.

This product is not designed or intended to be used for on-line control of aircraft, aircraft navigation or communications systems or in air traffic control applications or in the design, construction, operation or maintenance of any nuclear facility, or for any medical use related to either life support equipment or any other life-critical application. Cyan Technology Ltd specifically disclaims any express or implied warranty of fitness for any or all of such uses. Ask your sales representative for details.

Revision History

Version	Date	Notes
V1.0	26/10/2005	First release

Contents

1 Introduction 5
2 Glossary 5
3 The Information Block 5
4 Limitations 6
5 Program Operation 6
6 Example Application 7
Application Program Interface 8
Appendix A Software Listings 9
A.1 flashIB.h 9
A.2 flashIB.c 9
A.3 main.c 14

1 Introduction

There are many applications where users require a small amount (tens of bytes) of non-volatile memory for the storage of variables, such as a network address, calibration data or version numbers. In such applications, often it is necessary to use an external memory device such as a serial EEPROM. The eCOG1k contains a separate section of flash memory designed for this purpose, termed the information block, and this application note demonstrates how the eCOG1k flash memory information block can be used as a non-volatile storage area for application data with full read-write access. Example software is provided that reads, writes and verifies the information block using the provided API.

2 Glossary

A table of abbreviations used in this document.

eCOG1	Cyan Technology target micro controller
EEPROM	Electrically Erasable Programmable Read-Only Memory
MMU	Memory Management Unit
UART	Universal Asynchronous Receiver/Transmitter

3 The Information Block

The eCOG1k contains 64Kbytes of non-volatile internal flash memory, organised as 32Kwords x 16 bits. The information block is separate from the main flash memory block and is organised as an additional 64 x 16 bit words. The first four words of the information block, at byte addresses 0-7, contain the read/write protection control bits for the entire flash memory, including both the main and information blocks. These locations should not be written unless the user requires explicit control of the read/write protect status of the flash memory. The included application software ignores the status of the read/write protect mechanism, and addresses only those memory locations above the flash memory block, using the address range 0-120 (actual address range is 8-127).

The information block is programmed in the same way as the main flash memory, while asserting the internal ***infren*** signal. Application notes *AN001 - eCOG1k Internal Flash Memory* and *AN018 - Writing Data to Flash Memory* contain detailed descriptions and example code for writing data to the flash memory and the information block. This application note describes the implementation using general methods of the ideas discussed in these application notes, and provides a user API for easy incorporation into a CyanIDE project.

Information block reads do not use the MMU; the information block is read via dedicated registers, as shown below:

flash.inf.rd_adr	0xFFCC
flash.inf.rd_data	0xFFCD

Writing to the ***flash.inf.rd_adr*** register starts the fetch from the information block. The most significant bit of the ***flash.inf.rd_adr*** register contains a ***bsy*** field to indicate that the device is fetching the contents of the specified address. When the ***flash.inf.rd_adr.bsy*** field is low, the contents of the ***flash.inf.rd_data*** register are valid.

4 Limitations

When programming the flash memory, it is necessary to ensure that there are no read accesses to the memory being programmed. Therefore, the code that performs the programming must be run from a different memory. This also requires that any interrupts that would usually result in execution from within the flash memory be disabled.

The programming code uses the *tmr* timer to generate precise time delays during programming. User code that also uses this timer must reconfigure the timer after any calls to the flash programming interface.

The software is designed to use the HEAP segment for execution of the flash programming code. The user must ensure that this segment is large enough to contain the entire programming function and a copy of the information block. A heap size of 0x200 words is ample for the application.

5 Program Operation

Three functions are available for writing to the information block, and are described in the API section. The read function simply programs the correct registers for an information block read, while respecting the necessary timing requirements.

The programming code relies on a read-modify-write sequence, as the entire information block must be erased at once. Thus for each call to the `Write_IB_Byte()` routine, the complete information block is copied to a buffer area in the heap, the specified byte is modified, and the entire block is reprogrammed using the updated buffer. The buffer (in the *HEAP* segment) is assigned using `malloc()` and has a static size of 120 bytes; the read/write protect bits are not programmed.

For programming, it is necessary to copy the code for the programming function to another memory device prior to execution. The application uses dynamic allocation of internal SRAM for executing the programming code, using the following method.

- Assign enough *HEAP* memory for the programming function
- Copy the programming function to the assigned *HEAP* segment
- Map the *HEAP* segment to code space
- Execute the programming code from the heap
- Free the assigned memory segment

Using this method, the application needs only to be guaranteed that the *HEAP* segment is large enough to contain the programming code. It is not necessary for the application explicitly to assign SRAM (or other) memory for executing the programming code, or to position the programming code in a particular flash memory location.

The `Write_IB_Byte_Confirm()` routine is built on the read and write functions described above, and confirms that the specified data has been written successfully after writing. A flag is passed to indicate the status of the operation.

6 Example Application

The application software (*AN032SW.zip*) contains the software required for writing to the information block, and an example application that performs read/write testing using the three API functions. The software requires that a terminal program (such as HyperTerminal) is connected to UART A on the evaluation/development board with the following settings.

- 9600 baud
- 8 data bits, no parity, 1 stop bit
- No flow control

The example project can be downloaded to the target system and executed. Every location in the flash memory information block is written with different data and read back, and the progress is reported via UART A.

Application Program Interface

`unsigned char Read_IB_Byte(unsigned char address)`

Read and return a byte from the information block at the location specified.

Parameters

address	The information block address of the byte to be read

Returns

The contents of the information block address.

Example

```
Out_Byte = Read_IB_Byte(Read_Address);
```

`Void Write_IB_Byte(unsigned char address, unsigned char data)`

Write the specified byte to the specified address.

Parameters

address	The information block address of the byte to be read
data	The data byte to write to *address*

Returns

Nothing.

Example

```
Write_IB_Byte(Write_Address, Write_Byte);
```

`unsigned int Write_IB_Byte_Confirm`
`(unsigned char address, unsigned char data)`

Write the specified byte to the specified address, and read the byte back to confirm. The function is built upon *Read_IB_Byte()* and *Write_IB_Byte()*.

Parameters

address	The information block address of the byte to be read
data	The data byte to write to *address*

Returns

`TRUE` if the data read is the same as the data written, otherwise `FALSE`.

Example

```
if (Write_IB_Byte_Confirm(Write_Address, Write_Byte) != TRUE)
{
      error = 1;
}
```

Appendix A Software Listings

A.1 flashIB.h

```
/*=============================================================================
Cyan Technology Ltd
Example application software for eCOG1

FILE
    flashIB.h

DESCRIPTION
    Tony Ward, October 2005
    Header file for using the information block for non-volatile storage.
=============================================================================*/

// Function prototypes
unsigned char Read_IB_Byte (unsigned char address);
void Write_IB_Byte (unsigned char address, unsigned char data);
int Write_IB_Byte_Confirm (unsigned char address, unsigned char data);

enum
{   // Return codes for write and confirm
    TRUE = 0,
    FALSE
};
```

A.2 flashIB.c

```
/*=============================================================================
Cyan Technology Ltd
Example application software for eCOG1

FILE
    flashIB.c

DESCRIPTION
    Tony Ward, October 2005
    Source file for using the information block for non-volatile storage.
    The routines used for programming the flash memory are borrowed heavily
    from AN018 - eCOG1 Writing Data to Flash memory.
=============================================================================*/

// Include files
#include <ecog.h>
#include <ecog1.h>
#include <stdlib.h>
#include "driver_lib.h"
#include "flashIB.h"

// Local function prototypes
void Program_IB(unsigned int *buffer);
void Write_IB(unsigned int *buffer);

typedef void (*FUNC_PTR)(unsigned int *);
FUNC_PTR fptr;

// This function performs byte read access on the information block
unsigned char Read_IB_Byte(unsigned char address)
{
    unsigned char data;

    // Set the read period
    fd.flash.prg_cfg.period = 0x3;

    // Offset address, start writing above RW protect bits
    fd.flash.inf_rd_adr.adr = ((address + 8) >> 1);
    while (fd.flash.inf_rd_adr.bsy)
        ;        // Wait for read to complete
    data = (address & 1) ? (rg.flash.inf_rd_data & 0xff) : (rg.flash.inf_rd_data >> 8);
    return(data);
}
```

```
/* This function must perform a read-modify-write cycle on the entire flash
 * info block.  We must therefore have 60 words of RAM available, and for
 * this we will use the heap
 */
void Write_IB_Byte(unsigned char address, unsigned char data)
{
    // Flash memory variables
    unsigned char *flash_buffer;
    unsigned int *flash_buffer_ptr;
    unsigned char IB_addr;

    // Get some memory for the read
    flash_buffer_ptr = (unsigned int *)malloc(128);
    flash_buffer = (unsigned char *)flash_buffer_ptr;

    /* READ */
    // Read in the entire flash here, rather than use repeated calls to the
    // Read_NV_Storage function
    for (IB_addr = 0; IB_addr < 64; IB_addr++)
    {
        fd.flash.inf_rd_adr.adr = IB_addr;
        while (fd.flash.inf_rd_adr.bsy)
            ;         // Wait for read to complete
        *flash_buffer_ptr++ = rg.flash.inf_rd_data;
    }

    /* MODIFY */
    address += 8;         // R/W protect bit offset
    *(flash_buffer + address) = data;

    /* WRITE */
    // Program the information block
    Program_IB((unsigned int *)flash_buffer);

    free(flash_buffer);
}


/* Write the data to flash, then read back and confirm
 * The function is built on the above functions, and will only return when
 * the written data is verified
 */
int Write_IB_Byte_Confirm(unsigned char address, unsigned char data)
{
    unsigned char temp;

    Write_IB_Byte(address, data);
    temp = Read_IB_Byte(address);

    return ((temp == data) ? TRUE: FALSE);
}

/* This function runs the programming interface.
 * The entire RAM memory is mapped to code side above the flash space. Heap is then
 * allocated and the function is copied to the heap, before being executed from there
 */
void Program_IB(unsigned int *buffer)
{
    unsigned int *heap_ptr;
    FUNC_PTR ptFun = &Write_IB;
    unsigned char *code_ptr = (unsigned char *)ptFun;    // code_ptr must be 24-bit

    int i;

    // Translator backup variables
    unsigned int tran_backup[7];

    // We need to set up the timers here
    // The ripple value is set by the application for 5 (/64), from H PLL
    // gives .65 usec period
    fd.ssm.clk_dis.tmr = 1;
    fd.tim.ctrl_dis.tmr_cnt = 1;
    fd.ssm.rst_set.tmr = 1;
    fd.ssm.tap_sel3.tmr = 5;    // divide by 64
    fd.ssm.div_sel.tmr = 1;    // High PLL
```

```
    fd.ssm.rst_clr.tmr = 1;
    fd.ssm.clk_en.tmr = 1;
    fd.tim.ctrl_en.tmr_cnt = 1;

    //    BACK UP THE FLASH DATA TRANSLATOR PARAMS
    tran_backup[0] = rg.mmu.flash_data_log;
    tran_backup[1] = rg.mmu.flash_data_phy;
    tran_backup[2] = rg.mmu.flash_data_size;
    tran_backup[3] = rg.mmu.ram_data0_log;    // This should be above the flash mapping
    tran_backup[4] = rg.mmu.ram_data0_phy;
    tran_backup[5] = rg.mmu.ram_data0_size;
    tran_backup[6] = rg.mmu.translate_en;

    // ALLOCATE SOME HEAP FOR PROGRAMMING CODE
    heap_ptr = (unsigned int *) malloc (0x300);
    // MAP IN FLASH TO DATA SPACE
    rg.mmu.flash_data_log = 0x00;            // top half of data space
    rg.mmu.flash_data_phy = 0x00;            // At the beginning
    rg.mmu.flash_data_size = 0x7f;           // All of it
    // Translator is always enabled
    // COPY CODE TO HEAP
    ptFun = (FUNC_PTR) heap_ptr;
    code_ptr = (unsigned char *)((unsigned long)code_ptr << 1);
    for (i = 0; i < 0x180; i++)
    {
        *heap_ptr++ = *(unsigned int *)code_ptr;
        code_ptr += 2;
    }
    // MAP THE RAM SEGMENT TO CODE SPACE
    // Mimics the RAM data translators, for the function pointer to be accurate,
    // we need to have the heap mapped in the same place in code space
    // This shouldn't be a problem as flash memory is only 0x8000 words long
    rg.mmu.ram_code_log = rg.mmu.ram_data0_log;    // should be above flash mapping
    rg.mmu.ram_code_phy = rg.mmu.ram_data0_phy;
    rg.mmu.ram_code_size = rg.mmu.ram_data0_size;
    rg.mmu.translate_en |= MMU_TRANSLATE_EN_RAM_CODE_MASK;
    // RUN THE CODE (FROM RAM)
    (*ptFun)(buffer);

    free ((unsigned int *) ptFun) ;

    // RESTORE THE ORIGINAL STATE OF THE TRANSLATORS
    //    BACK UP THE FLASH DATA TRANSLATOR PARAMS
    rg.mmu.flash_data_log = tran_backup[0];
    rg.mmu.flash_data_phy = tran_backup[1];
    rg.mmu.flash_data_size = tran_backup[2];
    rg.mmu.ram_data0_log = tran_backup[3];        // should be above flash mapping
    rg.mmu.ram_data0_phy = tran_backup[4];
    rg.mmu.ram_data0_size = tran_backup[5];
    rg.mmu.translate_en = tran_backup[6];
}


/* Write the information block.  We pass a pointer to a buffer containing the contents
 * and the entire flash block is written after erasing
 */
void Write_IB(unsigned int *buffer)
{
    int usec_count;         // variables required for timing delays
    int msec_count;

    int count;
    int adr = 0;            // Flash access parameters

    msec_count  = 1563;     // = .0010032
    usec_count  = 2;        // = .00000128 (not accurate, but adequate)

    // Begin Erase procedure
    adr = 0;            // Only 1 page in information block
    buffer += 4;        // Increment this, we need to skip writing the info block

    // Need a non-zero value in here, defines the access time in ticks
    fd.flash.prg_cfg.period = 0x1f3;
    // Enable erase/write, no command
    rg.flash.prg_ctrl = 0x290;
    // Set the address for the flash erase/write
```

```
rg.flash.prg_adr = adr ;

// Set the page erase command
// Enable erase_op, xe and erase signals (on IB)
rg.flash.prg_ctrl = 0x2974;

// This loop generates a 5us delay
rg.tim.tmr_ld = usec_count;
for (count = 0; count < 5; count++)
{
    rg.tim.cmd = TIM_CMD_TMR_LD_MASK;
    while (rg.tim.tmr_ld != usec_count)
        ;        // Wait for load value to be clocked in
    rg.tim.ctrl_en = TIM_CTRL_EN_TMR_CNT_MASK;
    while (rg.tim.tmr_cnt)
        ;
    rg.tim.ctrl_dis = TIM_CTRL_DIS_TMR_CNT_MASK;
}

// Enable the nvstr signal
rg.flash.prg_ctrl = 0x2975;
// This loop generates a 10ms delay
rg.tim.tmr_ld = msec_count;
for (count = 0; count < 10; count++)
{
    rg.tim.cmd = TIM_CMD_TMR_LD_MASK;
    while (rg.tim.tmr_ld != msec_count)
        ;        // Wait for load value to be clocked in
    rg.tim.ctrl_en = TIM_CTRL_EN_TMR_CNT_MASK;
    while (rg.tim.tmr_cnt)
        ;
    rg.tim.ctrl_dis = TIM_CTRL_DIS_TMR_CNT_MASK;
}

// Remove the erase signal
rg.flash.prg_ctrl = 0x2971;

// This loop generates a 5us delay
rg.tim.tmr_ld = usec_count;
for (count = 0; count < 5; count++)
{
    rg.tim.cmd = TIM_CMD_TMR_LD_MASK;
    while (rg.tim.tmr_ld != usec_count)
        ;        // Wait for load value to be clocked in
    rg.tim.ctrl_en = TIM_CTRL_EN_TMR_CNT_MASK;
    while (rg.tim.tmr_cnt)
        ;
    rg.tim.ctrl_dis = TIM_CTRL_DIS_TMR_CNT_MASK;
}

// Remove all control signals
rg.flash.prg_ctrl = 0x2900;
// Disable the flash memory
rg.flash.prg_ctrl = 0x0000;
// End Erase procedure

// Begin Write procedure
fd.flash.prg_cfg.period = 0x1f3;

// Enable the flash memroy
rg.flash.prg_ctrl = 0x2900;

// Set the programming address
rg.flash.prg_adr = 4;

/* The following code is responsible for programming data
 * Programming is carried out in 2 blocks of 32 words, ignoring W/R protect
 */
for (adr = 4; adr < 64; adr++)
{
    // Set the flash control to write mode
    // enable program_op, xe and prog
    rg.flash.prg_ctrl = 0x29b2;

    // Wait 5us
    rg.tim.tmr_ld = usec_count;
```

```
        for (count = 0; count < 5; count++)
        {
            rg.tim.cmd = TIM_CMD_TMR_LD_MASK;
            while (rg.tim.tmr_ld != usec_count)
                ;        // Wait for load value to be clocked in
            rg.tim.ctrl_en = TIM_CTRL_EN_TMR_CNT_MASK;
            while (rg.tim.tmr_cnt)
                ;
            rg.tim.ctrl_dis = TIM_CTRL_DIS_TMR_CNT_MASK;
        }

        // Enable nvstr
        rg.flash.prg_ctrl = 0x29b3;

        // Wait 10us
        rg.tim.tmr_ld = usec_count;
        for (count = 0; count < 10; count++)
        {
            rg.tim.cmd = TIM_CMD_TMR_LD_MASK;
            while (rg.tim.tmr_ld != usec_count)
                ;    // Wait for load value to be clocked in
            rg.tim.ctrl_en = TIM_CTRL_EN_TMR_CNT_MASK;
            while (rg.tim.tmr_cnt)
                ;
            rg.tim.ctrl_dis = TIM_CTRL_DIS_TMR_CNT_MASK;
        }

        fd.flash.prg_cfg.last_wr = 1;
        rg.flash.prg_adr = adr ;
        rg.flash.prg_data = *buffer++;
        while (fd.flash.prg_adr.bsy)
            ;
        // Remove prog signal
        rg.flash.prg_ctrl = 0x29b1;
        // Remove all control signals
        rg.flash.prg_ctrl = 0x2900;
    }

    // Disable the flash memory
    rg.flash.prg_ctrl = 0x0000;

    // End Write procedure
}
```

A.3 main.c

```
/*============================================================================
Cyan Technology Limited

FILE - main.c

DESCRIPTION
    This is a test function for reading and writing the information block
    on the eCOG1. A test byte is written to each location and read back
    to confirm.

============================================================================*/
#include <ecog.h>
#include "flashIB.h"
#include "driver_lib.h"

#define NUM_BYTES 0x78

int main(int argc, char* argv[])
{
    unsigned char i;
    unsigned int error = 0;        // No error
    const char *start_msg = "Testing Information Block...\r\n";
    const char *success_msg = "Information block written and verified!\r\n";
    const char *error_msg = "Information block read/write error!\r\n";

    // The information block is 128 bytes long, organised as 64 words
    // The first 8 bytes are used for read/write protect of the entire flash
    // The software therefore uses only the addresses above the protect bits
    // and accepts addresses of 0x0 - 0x77 inclusive

    while (duart_a_tx((int)*start_msg++) != '\0');
    // Write using routines
    for (i = 0; i < NUM_BYTES; i++)
    {
        Write_IB_Byte (i, i);
    }
    // Read using routine
    for (I = 0; i < NUM_BYTES; i++)
    {
        if (i != Read_IB_Byte(i))
        {
            error = 1;
        }
    }
    // Write and confirm using routines
    for (i = 0; i < NUM_BYTES; i++)
    {
        if (FALSE == Write_IB_Byte_Confirm(i, i))
        {
            error = 1;
        }
    }

    if (error)
    {
        while (duart_a_tx ((int)*error_msg++) != '\0');
    }
    else
    {
        while (duart_a_tx ((int)*success_msg++) != '\0');
    }

    while (1);        // Stop here

    return 0;
}
```