University of Technology, Sydney
Faculty of Engineering and Information Technology

Subject: **48434 Embedded Software**

Assessment Number: **2**

Assessment Title: **Lab 2 – MCG and Flash Memory**

Tutorial Group:

Students Name(s) and Number(s)

Student Number	Family Name	First Name

Declaration of Originality:

The work contained in this assignment, other than that specifically attributed to another source, is that of the author(s). It is recognised that, should this declaration be found to be false, disciplinary action could be taken and the assignments of all students involved will be given zero marks. In the statement below, I have indicated the extent to which I have collaborated with other students, whom I have named.

Statement of Collaboration:

Signature(s)

key

Assessment Submission Receipt

Assessment Title:	**Lab 2 – MCG and Flash Memory**	**Mark**
Student Name(s):		
Date Submitted:		
Tutor Signature:		Office use only ☺

Assessment Criteria

Your lab will be assessed according to the following criteria:

Item	Detail	Evaluation	Mark
Opening comments / function descriptions	File headers are correct. Function descriptions are appropriate and correct.	**G A P**	**/0.5**
Naming conventions / code structure	Names and code structure conform to the Software Style Guide.	**G A P**	**/0.5**
Doxygen comments	Comments for all functions, variables and modules are present and informative. Documentation compiles.	**G A P**	**/1**
Initialization	Multipurpose Clock Generator / Processor Expert is set up correctly. LED turns on.	**G A P**	**/1**
Flash HAL	Functionally correct for public functions. Private functions are correct and robust.	**G A P**	**/4**
Protocol implementation	Protocol response is correct, including ACK/NAK.	**G A P**	**/1**
		TOTAL	**/8**

Evaluation

When we evaluate an assessment item, we will use the following criteria:

- **G =** All relevant material is presented in a logical manner showing clear understanding, and sound reasoning. For software - evidence of correct coding style, efficient implementation and / or novel (and correct) code.
- **A =** Most relevant material is presented with acceptable organisation and understanding. For software – some code may be prone to errors under certain operating conditions (e.g. input parameters) or usage, style may have inconsistent sections, occasional inefficient or incorrect code.
- **P =** Little relevant material is presented and/or code displays poor organisation or understanding of the underlying concepts.

Oral Defence

During the demonstration session you will be asked a number of questions based on material which you have learnt in the subject and then used to implement the assignment. You are expected to know exactly how your implementation works and be able to justify the design choices which you have made. If you fail to answer the questions with appropriate substance then you will be awarded **zero** for that component.

Lab 2 – MCG and Flash Memory

Multipurpose clock generator. Flash memory. Serial protocol.

Introduction

The Multipurpose Clock Generator (MCG) module is responsible for the setting up of various system-wide clocks. Flash is a non-volatile memory technology that allows data to be stored when power to the K70 is removed.

Objectives

1. To use "Processor Expert" to safely transition the bus clock from an internal source to an external source when booting.
2. To write a hardware abstraction layer (HAL) for Flash memory.
3. To expand the implementation of the Tower serial protocol.

Equipment

- 1 TWR-K70F120M-KIT – UTS
- 1 USB cable – UTS
- NXP Kinetis Design Studio

Safety

This is a Category A laboratory experiment. Please adhere to the Category A safety guidelines (issued separately).

Cat. A lab

L2.2

Memory Overview

There are two types of memory on board the K70 microcontroller – random access memory (RAM) and non-volatile memory (NVM). The type of RAM used is *static* RAM, which means it is made up of flip-flops and does not need to be refreshed (as opposed to *dynamic* RAM or DRAM, which stores bits of information in capacitors with sensing transistors – eventually the charge on the capacitor leaks away and the DRAM needs to be refreshed). RAM can be read and written to at any time – it is the place where variables are stored, as well as the stack and the heap.

Non-volatile memory is implemented with Flash technology

The NVM is based on *Flash* technology. Flash is a memory technology that allows for bulk erasure, random writing, fast read access, and dense implementation (small silicon area). There are two styles of Flash memory used in the K70 family of MCUs. The first is called "Program flash memory" and consists of large *blocks* of memory – it normally holds the program code and constants. The second style is referred to as "FlexNVM". The FlexNVM memory has the capability to emulate electrically erasable programmable read-only memory (EEPROM). EEPROM refers to a now out-dated memory technology, so its name is a carry-over from previous generations of microcontrollers. FlexNVM has a built-in filing system and provides a high-endurance, byte-writeable, non-volatile memory that is specifically intended to hold non-volatile variables such as modes of program operation, flags, calibration constants, user options, etc.

EEPROM is emulated with Flash

In the MCU used on the Tower board (the MK70FN1M0VMJ12), we only have "Program flash memory". Therefore, we will store our non-volatile variables in a Flash block that we know will not be used by our program (our program is small so it leaves plenty of "Program flash" available for data).

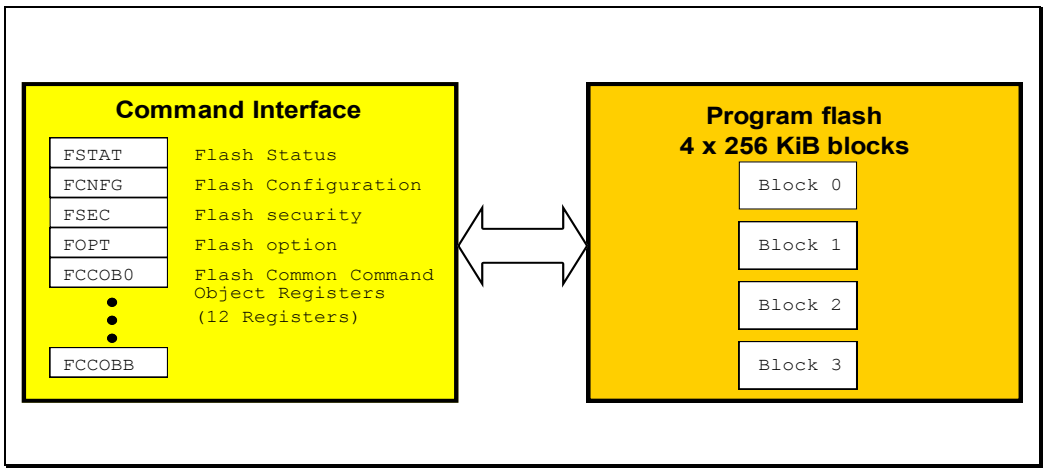
The Flash can be read just like normal RAM – no special procedure is needed. However, unlike RAM, when writing to the Flash a special procedure is required. Writing to the Flash requires various tasks to be carried out under certain timing constraints – if the tasks are not carried out in strict order, at strict voltages, then damage to the silicon can occur. Some of the tasks needed

Flash requires a special procedure when written to

to write to the memory are: applying a high programming voltage to a particular row; selecting a particular cell; pulsing the memory etc. All these tasks are carried out by an on-chip state machine that hides the complexity of this writing process. All we have to do is interact with a few control registers to be able to write to the Flash.

Programming Model of the Flash

A programming model of the Flash is shown below:



Programming model of the Flash

Figure L2.1

The Flash command interface consists of several registers that enable the Flash to be erased and written. In summary, they are:

- `FSTAT` – a status register to indicate the state-machine status.
- `FCNFG` – a configuration register to allow interrupts to be generated under certain conditions.
- `FSEC` – a read-only security register used to indicate the state of the built-in security features such as mass erase enable/disable.
- `FOPT` – a read-only register used to indicate the Flash options applied at boot time, such as whether to boot into low-power mode.
- `FCCOB0-B` – twelve registers that hold the Flash interface's command, address, data and other parameters associated with a particular Flash operation.

A complete description of the Flash memory module can be found in Chapter 30 of NXP's *K70 Sub-Family Reference Manual*.

L2.4

Using the Program Flash for Data Storage

Our device has four 256 KiB Flash blocks labelled Block0-3. You cannot read from a Flash block while it is being erased or written. However, the internal architecture of the K70 allows it to operate (i.e. read instructions from) one "bank" of Flash memory (Blocks0-1) while programming the other "bank" (Blocks2-3). We can therefore use Blocks2-3 as "data flash".

The Flash memory is mapped into the MCU 32-bit address space as follows:

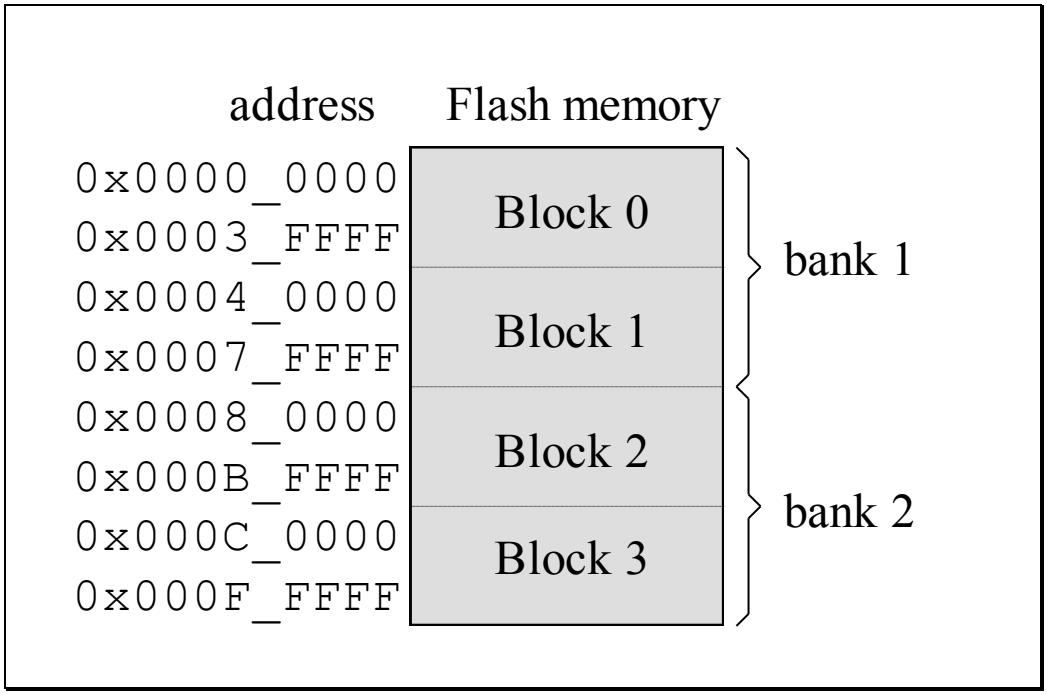


Figure L2.2

All K70 parts have "Program flash memory" in the range 0x0000_0000 to 0x0007_FFFF. The memory starting at address 0x0000_0000, which is in Block0, is actually a "vector table" that is used by the Nested Vectored Interrupt Controller (NVIC) to get the initial stack pointer, initial program counter, and interrupt service routine addresses. Storage for our program, and KDS's startup code, then follows the vector table and resides in Block 0. Since we will execute code from Block0, we have chosen to use Flash Block2 in the other "bank" to store our non-volatile data.

Inside each block, the Flash memory is further divided into *sectors*, which is the smallest unit that can be *erased*. Our device has 4 KiB sectors:

Flash memory is organised into erasable units called sectors

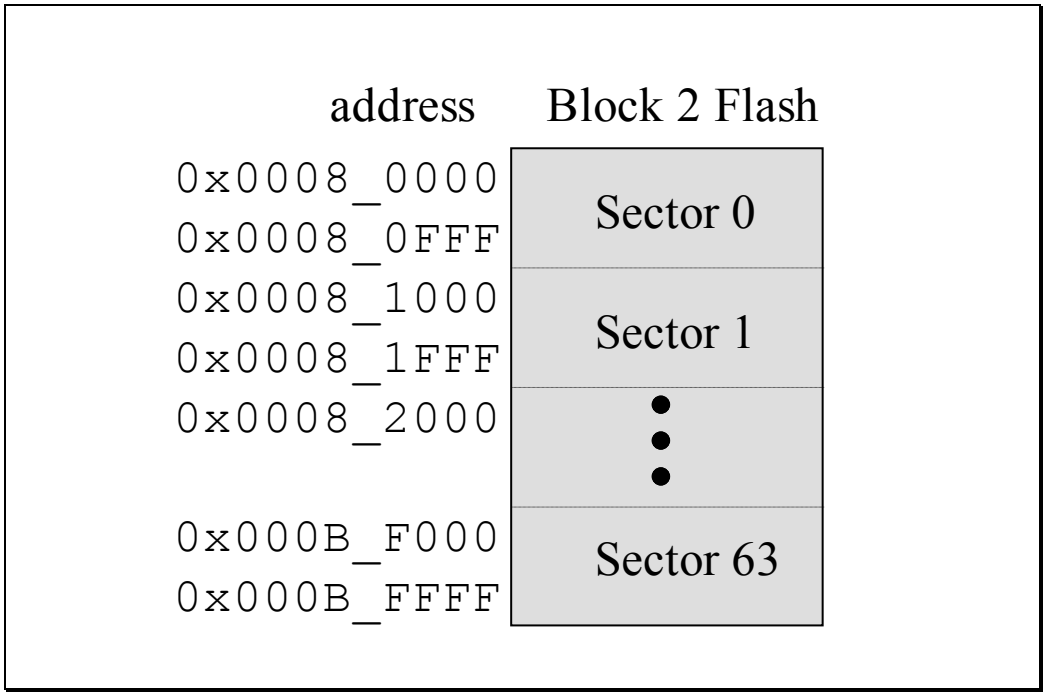


Figure L2.3

One aspect of Flash technology that should be remembered is that it must be *erased* before it is written to. **Failure to do so may damage the Flash array.**

Flash memory must be erased before it is written to!

This means that if we want to change just 1 byte in a 4 KiB sector, the entire sector contents must be read into a RAM buffer, and the 1 byte changed in the RAM buffer. Then the Flash sector must be erased, turning all 0's into 1's. Finally, we must write the entire new 4 KiB sector from the RAM buffer to the Flash sector.

That's why NXP's other K70 part has FlexNVM which emulates single-byte access non-volatile data storage!

Another complicating factor in using Program flash to store data is that the smallest unit of data we can *write* to the Flash array is 8 bytes, which is called a *phrase*. The phrase must be aligned on an 8-byte address, i.e. 0x0008_0000, 0x0008_0010, 0x0008_0018, etc.

Phrases are aligned to 8-byte boundaries

To ease the burden of operating the Program flash memory as a data storage area, we will further restrict ourselves to only operating in the first "phrase" of sector 0 of Flash Block2:

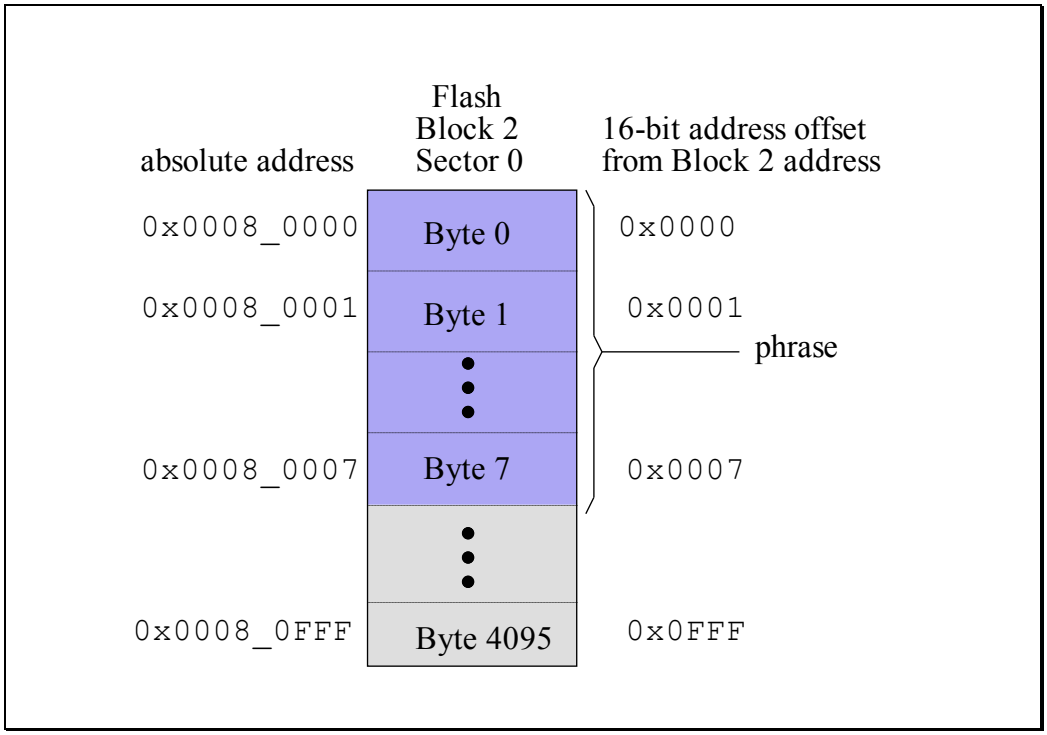


Figure L2.4

The *Tower Serial Communication Protocol* supports commands to program bytes into the Flash memory and to read bytes from the Flash memory. One of the parameters in the packet is used as the address *offset* into our 8-byte storage area. i.e. we can only program and read bytes with an offset between 0 and 7.

Careful reading of the *Tower Serial Communication Protocol* reveals that if the PC sends a packet to program a byte into Flash at address offset 8, then the Tower firmware is to treat this as a command to erase the entire sector, thus erasing all bytes in our phrase.

Software Modularity

We will write a Flash memory hardware abstraction layer (HAL) that is modular. In particular, the HAL will make public functions available for writing data to the Flash, and for bulk erasing. It will also be responsible for allocating Flash memory to non-volatile variables required by the user.

Software Requirements

1. The software is to incorporate all the features of Lab 1.
2. The Tower has a 50 MHz **external** clock which is connected to the EXTAL0 pin of the MCU. Use Processor Expert to configure the CPU clock to the following:

Clock	**Frequency**
Core / system clock	50 MHz
Bus clock	25 MHz
External bus clock	10 MHz
Flash clock	12.5 MHz

   Hint: The clock mode is Bypassed Low Power External (**BLPE)**.

3. The baud rate is to be set to 115200 baud.
4. A HAL is to be written for the Flash for erase and write operations. The write operations to be supported are 8-bit unsigned bytes at any address, 16-bit unsigned half-words at an even address, and 32-bit unsigned words on an address evenly divisible by 4.
5. The HAL should support the allocation of Flash memory at the user's request, up to a **maximum of 8 bytes total**, i.e. we are restricting Flash memory allocation to one phrase.

6. Extra commands of the Tower serial protocol to be implemented are:

Tower to PC	PC to Tower
	0x07 Flash – Program byte
0x08 Flash – Read byte	0x08 Flash – Read byte
0x0B Tower Number	0x0B Tower Number (get & set)
0x0D Tower Mode	0x0D Tower Mode (get & set)

7. The Tower response to a "0x04 Special – Get startup values" packet should be the transmission of 4 packets:

    - 0x04 Special – Startup
    - 0x09 Special – Version number
    - 0x0B Tower Number
    - 0x0D Tower Mode

    Upon power up, the Tower should send the same 4 packets as above.

8. The Tower number and mode are to be stored in Flash. If an *unprogrammed* Tower number or mode are detected on startup (i.e. the Flash has been erased so that the data is 0xFFFF), the application should program the Tower number to the last four digits of your student number and the Tower mode to `1`.

9. If the Tower board is successful in starting up (i.e. Flash is initialized successfully, UART is initialized successfully, etc.) then the orange LED "D7" should be turned on.

10. TortoiseSVN must be used for version control.

Hints

1. `packet.h` has changed to facilitate manipulation of packet parameters.

2. Read section "30.4.10.1.3 Command Execution and Error Reporting" in the K70 Reference Manual. Figure 30-34 on p. 806 is also useful.

3. Here's an example of how the Flash functions would be used:

   I want a `uint16union_t` variable to reside in Flash memory. I therefore declare an uninitialised pointer to a `uint16union_t` :

   ```
   // Prefix "Nv" stands for "non-volatile"
   volatile uint16union_t *NvTowerNb;   /*!< The non-volatile Tower number. */
   ```

   At the moment, my pointer `NvTowerNb` doesn't point to anything. I want it to point to 16-bits of data in the Flash memory, so I call my Flash variable allocation function:

   ```
   success = Flash_AllocateVar(&NvTowerNb, sizeof(*NvTowerNb));
   ```

   Now my pointer has an address (into Flash memory) where my data will actually reside.

   I have to use a special function to write to my non-volatile variable:

   ```
   // Set new Tower number
   success = Flash_Write16((uint16_t *)NvTowerNb, Packet_Parameter23);
   ```

   Reading the variable doesn't require any special functions. Here I am reading my non-volatile Tower Number to send back in a packet:

   ```
   Packet_Put(CMD_TOWER_NB, 1, NvTowerNb->s.Lo, NvTowerNb->s.Hi);
   ```

   Notice how the **`union`** is very handy for handling data in different ways.

4. Make a **`struct`** to encapsulate what you want to put into the FCCOB registers. When using the CCOB registers, note they are big-endian. See the note on p. 790 in the K70 Reference Manual regarding FCCOB Endianness and Multi-Byte Access.

5. Be succinct about terminology in relation to type sizes in the ARM (and the C compiler) and the definitions for the Flash:

Name	C type	Size (in bytes)
Byte	uint8_t	1
Half-word	uint16_t	2
Word	uint32_t	4
Phrase	uint64_t	8

6. Due to the nature of Flash memory, it needs to be bulk erased before being written to. That means any change in data requires the entire contents of the Flash memory to be copied out into RAM, the changes made in RAM, and then the RAM "image" being written back to Flash memory. Thus, any call to one of the "Write" functions in `Flash.c` will eventually lead to erasure of the entire sector, before writing back the entire phrase that we are using.

7. You can structure your software so that `Flash_Write8` calls `Flash_Write16` which calls `Flash_Write32` which calls a private function to write a phrase (after sector erasure).

8. There will be several private functions (not made public by `Flash.h`) that need writing to support the public functions. Some suggested helper functions are:

```
static BOOL LaunchCommand(TFCCOB* commonCommandObject);
static BOOL WritePhrase(const uint32_t address, const uint64union_t phrase);
static BOOL EraseSector(const uint32_t address);
static BOOL ModifyPhrase(const uint32_t address, const uint64union_t phrase);
```

9. The LEDs are connected to Port A. Check the Tower schematic to see which pins they are connected to. Note what logic level (high or low) turns the LEDs on/off. You will need to set up the correct Port A pins to be general purpose outputs, with drive strength enabled.

Marking

The software should be ready for marking on the date specified in the Timetable in the Learning Guide.

Software marking will be carried out in the laboratory, in the format of an oral exam.

Marking criteria are on the front page. Also refer to the document "Software Style Guide".