UTS

University of Technology, Sydney

Faculty of Engineering and Information Technology

Subject:	**48434 Embedded Software**
Assessment Number:	**2**
Assessment Title:	**Lab 2 – EEPROM and CRG**
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)

Marks

Opening comments / function descriptions	/0.5
Naming conventions / code structure	/0.5
EEPROM HAL	/4
CRG implementation	/2
Protocol implementation	/1
TOTAL	/8

Office use only ☺

key

Assessment Submission Receipt

Assessment Title:	**Lab 2 – EEPROM and CRG**	**Mark**
Student's Name:		
Date Submitted:		
Tutor Signature:		Office use only ☺

Lab 2 – EEPROM and CRG

EEPROM. Phase-locked loop. Computer operating properly. Serial protocol.

Introduction

The EEPROM is a non-volatile memory technology that allows data to be stored when power to the 68HC12 is removed. The Clocks and Reset Generator (CRG) module is responsible for the setting up of various system-wide clocks and the generation of resets based on the Computer Operating Properly (COP) clock and oscillator stability. The CRG module can also be used for the generation of real-time interrupts.

Objectives

1. To write a hardware abstraction layer (HAL) for an EEPROM on a microcontroller.
2. To set up a phase-locked loop.
3. To set up a Computer Operating Properly timer.
4. To expand the implementation of the ModCon serial protocol.

Equipment

- 1 ModCon microcontroller board – UTS
- 1 HCS12 USB programmer / debugger – PEMicro
- 2 USB cables
- 1 5V DC power supply, 1 12V DC power supply, PC connector
- Freescale CodeWarrior IDE for the 68HC12

Safety

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

Cat. A lab

Memory Overview

There are two types of memory on board the 68HC12 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 on the 68HC12. The first consists of large *blocks* of memory and is usually referred to simply as "Flash" – it normally holds the program code and constants. The second style is referred to as EEPROM (electrically erasable programmable read-only memory). EEPROM normally refers to a now out-dated memory technology, so it's use is a carry-over from previous generations of microcontrollers. The EEPROM consists of small 4-byte *sectors* of memory, and is specifically intended to hold non-volatile variables and constants, such as modes of program operation, flags, calibration constants etc.

EEPROM is Flash with a very small block size

EEPROM requires a special procedure when written to

The EEPROM can be read just like normal RAM – no special procedure is needed. However, unlike RAM, when writing to the EEPROM a special procedure is required. Writing to the EEPROM 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 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 EEPROM.

Programmers' Model of the EEPROM

A programmer's model of the EEPROM is shown below:

Programmer's model of the EEPROM

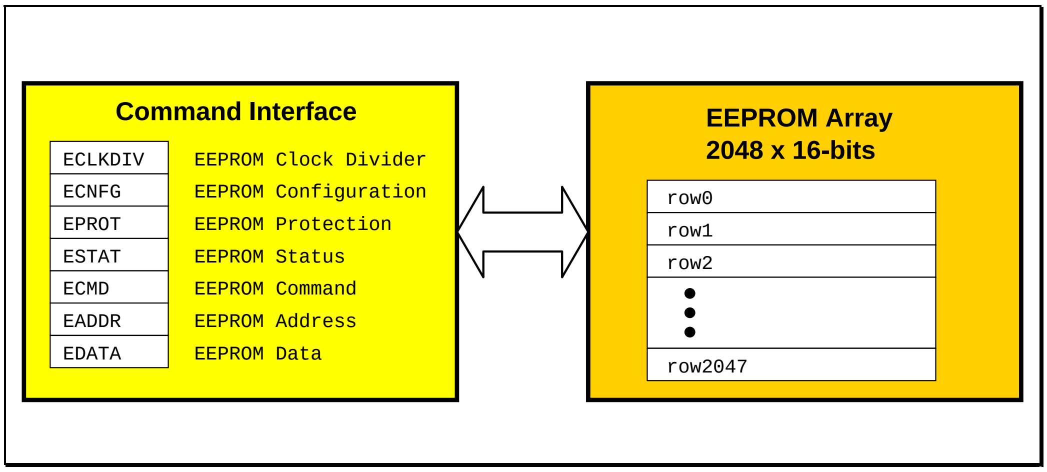

Figure L2.1

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

- `ECLKDIV` – a clock divider to derive a 200 kHz clock from the oscillator clock.
- `ECNFG` – a configuration register to allow interrupts to be generated under certain conditions.
- `EPROT` – a protection register used to indicate sectors that are prevented from being erased or written.
- `ESTAT` – a status register to indicate the state-machine status.
- `ECMD` – a command register used to indicate erase, program or verify.
- `EADDR` – an address register to store the address of the EEPROM to modify (we do not have *direct* access to this register).
- `EDATA` – a data register to store the data to be written to the EEPROM (we do not have *direct* access to this register).

Some of these registers are written to only once (for example, the EEPROM clock divider). A complete description of the EEPROM block can be found in the Freescale document *EETS4K Block User Guide*.

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!

Clocks and Reset Generator Module

The Clocks and Reset Generator (CRG) module is responsible for the generation of various clocks used around the various microcontroller modules. It provides the circuitry to interface with an external crystal, it has a phase-locked loop (PLL) that is used to divide the external frequency up or down, it has a system reset generator and it can generate real-time interrupts.

A simplified block diagram of the CRG module is shown below:

A simplified CRG block diagram

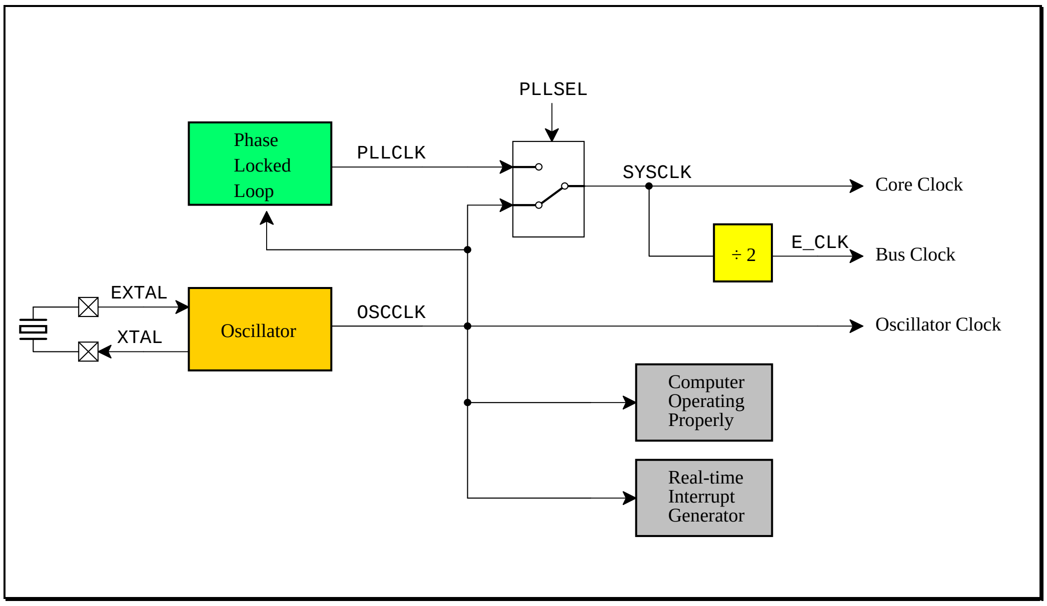

Figure L2.2

As can be seen from the diagram, an external crystal is used to derive a clock called `OSCCLK`. This clock drives other modules as well as the COP sub-module and RTI sub-module. It is also fed into the phase-locked loop, which develops a new clock called `PLLCLK`. The `PLLCLK` clock can be selected, via the `PLLSEL` bit, to make the system clock, `SYSCLK`. The system clock goes directly to the CPU, where it used for the fetching and decoding of instructions. The system clock is also divided by 2, to create the bus clock, with the non-intuitive name `E_CLK`.

The `E_CLK` is a very important clock as it drives most peripherals.

The `OSCCLK` drives the Flash and EEPROM memory.

Computer Operating Properly Sub-Module

The Computer Operating Properly (COP) sub-module is sometimes called a watchdog timer. It is a timer which, once started, needs to be continually reset by the user's program. If the timer times-out, then the computer is reset. This is a security against runaway situations caused by faulty software, erroneous operation due to electromagnetic interference or perhaps failure of the silicon itself (e.g. radiation, such as gamma rays in space, can change the memory content of chips and cause other failures).

Real-Time Interrupt Generator Sub-Module

The Real-Time Interrupt (RTI) generator can be used to generate a hardware interrupt at a fixed periodic rate. This is useful when implementing digital schemes that need to sample an analog waveform at a particular rate, or when implementing a pre-emptive real-time operating system.

Default Condition of the CRG Module After a Reset

After a reset, the CRG defaults to a state where the `PLLSEL` bit is `0`. Thus, if nothing is done to control the generation of the various clocks, the microcontroller will default to using `OSCCLK` as the `SYSCLK`. The `SYSCLK` is then divided by 2 to generate the bus clock, `E_CLK`. Therefore, the default `E_CLK` is just `OSCCLK / 2`.

For a 16 MHz crystal, the bus clock will default to 8 MHz.

Programmer's Model of the CRG

A programmer's model of the CRG is shown below:

Programmer's model of the CRG

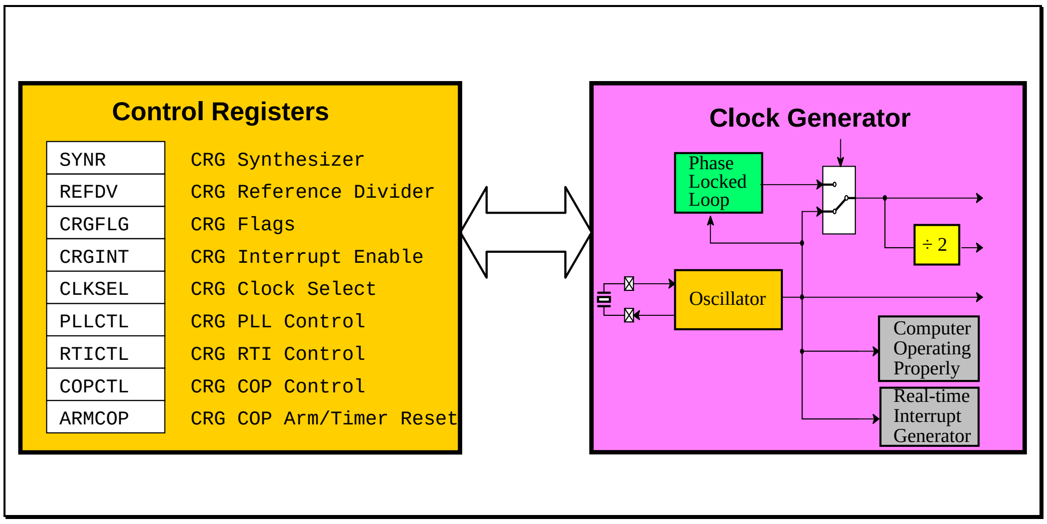

Figure L2.3

The CRG module consists of several registers that are used to setup and provide the status of the CRG functions. In summary, they are:

- `SYNR` – controls the multiplication factor of the PLL.
- `REFDV` – controls the division factor of the PLL.
- `CRGFLG` – provides status bits and flags.
- `CRGINT` – enables CRG interrupt requests.
- `CLKSEL` – controls CRG clock selection.
- `PLLCTL` – controls the PLL functionality.
- `RTICTL` – selects the timeout period for the RTI.
- `COPCTL` – controls the COP watchdog.
- `ARMCOP` – used to restart the COP time-out period.

Most of these registers are written to only once for setup purposes. A complete description of the CRG block can be found in the Freescale document *CRG Block User Guide*.

Software Requirements

1. The software is to incorporate all the features of Lab 1.
2. The oscillator clock is 16 MHz, and the bus clock (`E_CLK`) is to be set to 24 MHz using the PLL.
3. The baud rate is to be set to 115200 baud.
4. The COP is to be armed for a timeout period of approximately 1 ms. It should be restarted by writing a `0x55` at the start of the main loop, and a `0xaa` at the end of the main loop. Other modules, such as the EEPROM module, may also need to reset the COP if they introduce a delay (such as waiting for the EEPROM to write data).

   Note: If you enable the COP you will not be able to debug your program via the Background Debug Module (since the COP will time out) unless you also set the `RSBCK` bit in the `COPCTL` register.
5. A hardware abstraction layer is to be written for the EEPROM for erase and write operations. The write operations to be supported are 8-bit unsigned bytes at any address, 16-bit unsigned words at an even address, and 32-bit unsigned long words on an address evenly divisible by 4.

   Note: Due to a mask set erratum, it is not possible to use the sector modify command. Therefore, erase and write operations should be issued separately.
6. Extra commands of the ModCon serial protocol to be implemented are:

ModCon to PC	PC to ModCon
	0x07 EEPROM – Program byte
0x08 EEPROM – Stored data	0x08 EEPROM – Get byte
0x0D ModCon Mode	0x0D ModCon Mode (get & set)

7. The ModCon response to a “0x04 Special – Get startup values” packet should be the transmission of 4 packets:
    - 0x04 Special – Startup
    - 0x09 Special – Version number
    - 0x0B ModCon Number
    - 0x0D ModCon Mode

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

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

9. If the ModCon board is successful in starting up (i.e. PLL is setup, SCI is setup etc.) then the red LED “D11” should be turned on.

10. TortoiseSVN must be used for version control.

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”.

Software Specification

The following header files are suggested, but not mandatory.

EEPROM.h

```
// ----------------------------------------
// Filename: EEPROM.h
// Description: Routines for erasing and
//   writing to the EEPROM
// Author: (your name)
// Date: (the date)

#ifndef EEPROM_H
#define EEPROM_H

// new types
#include "types.h"

// EEPROM data access
#define _EB(EEPROM_ADDRESS) *(UINT8 volatile *)(EEPROM_ADDRESS)
#define _EI(EEPROM_ADDRESS) *(INT16 volatile *)(EEPROM_ADDRESS)
#define _EW(EEPROM_ADDRESS) *(UINT16 volatile *)(EEPROM_ADDRESS)
#define _EL(EEPROM_ADDRESS) *(INT32 volatile *)(EEPROM_ADDRESS)
#define _ES(EEPROM_ADDRESS) *(UINT32 volatile *)(EEPROM_ADDRESS)

// ----------------------------------------
// EEPROM addresses
// ----------------------------------------

// ModCon parameters
#define    sModConNb        _EW(0x400)
#define    sModConMode      _EW(0x402)

// ----------------------------------------
// EEPROM_Setup
//
// Sets up the EEPROM with the correct internal clock
// Input:
//  oscClk is the oscillator clock frequency in Hz
//  busClk is the bus clock frequency in Hz
// Output:
//   none
// Conditions:
//   none

BOOL EEPROM_Setup(const UINT32 oscClk, const UINT32 busClk);

// ----------------------------------------
// EEPROM_Write32
//
// Writes a 32-bit number to EEPROM
// Input:
//   address is the address of the data,
//   data is the data to write
// Output:
//   TRUE if EEPROM was written successfully
//   FALSE if address is not aligned to a 4-byte boundary
//   or if there is a programming error
// Conditions:
//   Assumes EEPROM has been initialized

BOOL EEPROM_Write32(UINT32 volatile * const address,
  const UINT32 data);
```

```
// ----------------------------------------
// EEPROM_Write16
//
// Writes a 16-bit number to EEPROM
// Input:
//   address is the address of the data,
//   data is the data to write
// Output:
//   TRUE if EEPROM was written successfully
//   FALSE if address is not aligned to a 2-byte boundary
//   or if there is a programming error
// Conditions:
//   Assumes EEPROM has been initialized

BOOL EEPROM_Write16(UINT16 volatile * const address,
  const UINT16 data);

// ----------------------------------------
// EEPROM_Write8
//
// Writes an 8-bit number to EEPROM
// Input:
//   address is the address of the data,
//   data is the data to write
// Output:
//   TRUE if EEPROM was written successfully
//   FALSE if there is a programming error
// Conditions:
//   Assumes EEPROM has been initialized

BOOL EEPROM_Write8(UINT8 volatile * const address, const UINT8 data);

// ----------------------------------------
// EEPROM_Erase
//
// Erases the entire EEPROM
// Input:
//   none
// Output:
//   TRUE if EEPROM was erased successfully
// Conditions:
//   Assumes EEPROM has been initialized

BOOL EEPROM_Erase(void);

#endif
```

CRG.h

```
// ----------------------------------------
// Filename: CRG.h
// Description: Routines for setting up the
//   clock and reset generator
// Author: (your name)
// Date: (the date)

#ifndef CRG_H
#define CRG_H

// new types
#include "types.h"

typedef enum
{
  COP_DISABLED  = 0,
  COP_RATE_2_14 = 1,
  COP_RATE_2_16 = 2,
  COP_RATE_2_18 = 3,
  COP_RATE_2_20 = 4,
  COP_RATE_2_22 = 5,
  COP_RATE_2_23 = 6,
  COP_RATE_2_24 = 7
} TCOPRate;

// ----------------------------------------
// CRG_SetupPLL
//
// Sets up the PLL to generate a certain bus clock
// Input:
//   busClk is the desired bus clock rate in Hz,
//   oscClk is the oscillator clock in Hz,
//   refClk is the reference clock in Hz
// Output:
//   TRUE if the bus clock was setup successfully
// Conditions:
//   Assumes that refClk divides oscClk evenly
//   Assumes that refClk divides busClk evenly

BOOL CRG_SetupPLL(const UINT32 busClk, const UINT32 oscClk,
  const UINT32 refClk);

// ----------------------------------------
// CRG_SetupCOP
//
// Sets up the COP to reset within a certain
//   number of milliseconds
// Input:
//   Desired COP rate, corresponding to
//   Table 3.3 in the CRG Block User Guide
// Output:
//   TRUE if the COP was setup successfully
// Conditions:
//   none

BOOL CRG_SetupCOP(const TCOPRate aCOPRate);

#endif
```