Name: ..... Student ID: .....
Name: ..... Student ID: .....

Laboratory 10
Interfacing to SD card and Accelerometer

Introduction to Storage Memory

1. Memory function types

A microprocessor needs memory for two reasons: to hold its program, and to hold the data that it is working with; we often call these *program memory* and *data memory*. To meet these needs there are a number of different semiconductor memory technologies available, which can be embedded on the microcontroller chip.

Memory technology is divided broadly into two types: volatile and non-volatile.

- **Non-volatile memory** retains its data when power is removed, but tends to be more complex to write to in the first place. An example is ROM (Read Only Memory). Non-volatile memory is generally required for program memory, so that the program data is there and ready when the processor is powered up.
- **Volatile memory** loses all data when power is removed, but is easy to write to. Volatile memory is traditionally used for data memory; it's essential to be able to write to memory easily, and there is little expectation for data to be retained when the product is switched off. An example is SRAM (Static Random Access Memory).

Using data files on the LPC1769

We can use standard C library via stdio.h to use the data files on the LPC1769

Function	Format	Summary Action
fopen	`FILE * fopen ( const char * filename, const char * mode);`	opens the file of name filename
fclose	`int fclose ( FILE * stream );`	closes a file
fgetc	`int fgetc ( FILE * stream );`	gets a character from a stream
fgets	`char * fgets ( char * str, int num, FILE * stream );`	gets a string from a stream
fputc	`int fputc ( int character, FILE * stream );`	writes a character to a stream
fputs	`int fputs ( const char * str, FILE * stream );`	writes a string to a stream
fseek	`int fseek ( FILE * stream, long int offset, int origin );`	moves file pointer to specified location

str An array containing the null-terminated sequence of characters to be written.
stream Pointer to a FILE object that identifies the stream where the string is to be written.

The compiler needs to know where to store and retrieve files; this is done using the mbed's LocalFileSystem declaration. This sets up the LPC1769 as an accessible flash memory storage unit and defines a directory for storing local files.

To implement, simply add the following line to the declarations section of a program:

```
LocalFileSystem local("local"); //Create file system named "local"
```

A file stored on the LPC1769 (in this example called "datafile.txt") can therefore be opened with the following command:

```
FILE* pFile = fopen("/local/datafile.txt","w");
```

When we have finished using a file for reading or writing it is essential to close it, for example using fclose(pFile);

The LocalFilesystem has a few restrictions:

- Only 8.3 filenames are supported (8 characters for name and 3 characters for file type)
- Sub-directories are not supported
- fseek is not supported for files opened for writing ("w")
- File access calls (fread, fwrite) will block, including interrupts, as semihosting is effectively a debug breakpoint

Example LPC1769 data file access

```
#include "mbed.h"
LocalFileSystem local("local");  // define local file system
int write_var;
int read_var;             // create data variables

int main() {
 FILE* File1 = fopen("/local/datafile.txt","w"); // open file
 write_var=0x23; // example data
 fputc(write_var, File1); // put char (data value) into file
 fclose(File1);                  // close file
 FILE* File2 = fopen("/local/datafile.txt","r"); // open file for reading
 read_var = fgetc(File2); // read first data value
 fclose(File2);
 printf("input value = %i \n",read_var);
}
```

String file access

```
// Read and write text string data
#include "mbed.h"
LocalFileSystem local("local");  // define local file system
char write_string[64];                  // character array up to 63 characters
char read_string[64];                   // create character arrays (strings)
```

```
int main ()
{
 FILE* File1 = fopen("/local/textfile.txt","w"); // open file access
 fputs("lots and lots of words and letters", File1);// put text into file
 fclose(File1);                                  // close file

 FILE* File2 = fopen ("/local/textfile.txt","r"); // open file for reading
 fgets(read_string,256,File2);          // read first data value
 fclose(File2);
 printf("text data: %s \n",read_string);
}
```

Secure Digital Memory Card

The Secure Digital Memory Card (SDC) has basically a flash memory array and a (micro) controller inside. The flash memory controls (erasing, reading, writing, error controls, and wearleveling) are completed inside the memory card. The data is transferred between the memory card and the host controller as data blocks in units of 512 bytes; therefore, these cards can be seen as generic hard disk drives from the view point of upper level layers. The currently defined file system for the memory card is FAT12/16. The FAT32 is defined for only high capacity (>= 4G) cards. SD Cards are widely used by loads of devices for storage; phones, mp3 players, pc's etc. That means they are a very cheap option for storing large amounts of non-volatile data (i.e. the data is not lost when the power is removed). They should be ideal for data logging and storing audio/images.

SD card consists of a 9-pin interface, a card controller, a memory interface and a memory core. The 9-pin interface allows the exchange of data between a connected system and the card controller. The controller can read/write data from/to the memory core using the memory core interface. In addition, several internal registers store the state of the card. The controller responds to two types of user requests: control and data. Control requests set up the operation of the controller and allow access to the SD card registers. Data requests are used to either read data from or write data to the memory core.

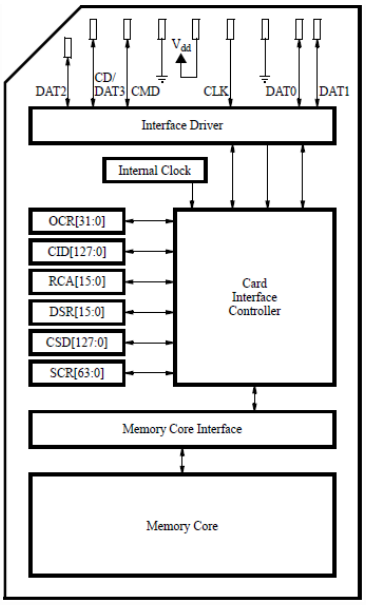


Fig.1 SD card block diagram and pin map

The data on an SD card is organized as a file system – cards below 2GB are typically formatted as FAT16 file systems. In a FAT file system, the first several storage blocks are used to maintain data about the file system – for example allocation tables – while the remaining blocks are used to store the contents of files and directories.

An application, which wishes to access data stored on an SD Card, utilizes file level commands such as open, read, and write to access specific files within the SD card file system. These commands are provided by a FAT file system driver. The FAT file system issues commands at the level of block reads and writes without any knowledge of how these commands are implemented. A separate SD driver implements these commands. Finally, the SD driver utilizes the SPI interface to communicate with the SD Card.

Using external memory with the LPC1769

A flash SD (Secure Digital) card can be used with the LPC1769 via the SPI protocol. It is possible to access the SD card as an external memory. SD Cards are block devices. That means you read/write data in multiples of the block size (usually 512-bytes); the interface is basically "read from block address n", "write to block address m". Note that a filesystem (e.g. FAT) is an abstraction on top of this, and the disk itself knows nothing about the filesystem. The SD card can be connected to the LPC1769 as shown in the following wiring table:

SD Breakout	Xpresso pin
SD CS	P0_16
SD MOSI	P0_18
SD MISO	P0_17

SD CLK	P0_15

`http://developer.mbed.org/teams/mbed/code/SDFileSystem/`

This library supports:

- FAT12 / FAT16 / FAT32
- SD / SDHC cards up to 32GB
- long filenames
- time stamp

Writing data to an SD Card

```
// Writing data to an SD card
#include "mbed.h"
#include "SDFileSystem.h"
SDFileSystem sd(P0_18, P0_17, P0_15, P0_16, "sd");  // MOSI, MISO, SCLK, SD_CS

int main() {
 FILE *File = fopen("/sd/sdfile.txt", "w");   // open file
 if(File == NULL) {
       printf("Could not open file for write\n");
 }
 else{
       printf("SD card file successfully opened\n");
 }
 fprintf(File, "Here's some sample text on the SD card"); // write data
 fclose(File);
}
```

Introducing Accelerometer

The digital accelerometer measures acceleration on 3 axes, using an internal capacitor mounted in the plane of each axis as shown in Fig. 2. Acceleration causes the capacitor plates to move, hence changing an output voltage proportionally to the acceleration or force. It is an example of a Micro-electromechanical system (MEMS). Some accelerometers give its output as a analog voltage while the others converts the analog voltage to digital and outputs this over a digital communication protocol. In this lab we use MMA7455L from Freescale. It gives digital output and can be configured to communicate in both SPI and I^2C modes.

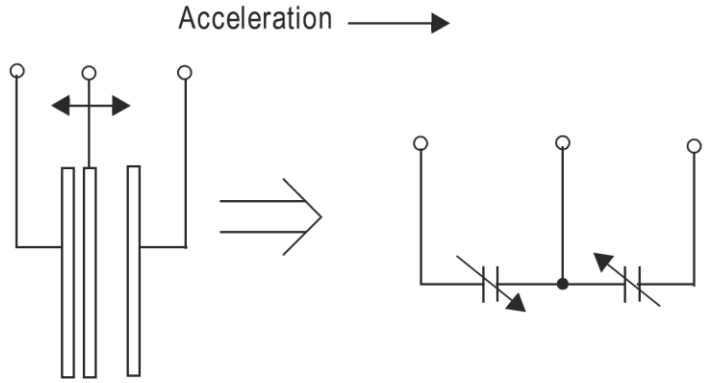


Fig. 2 Conceptual model of MEM capacitor (g-cell beam)

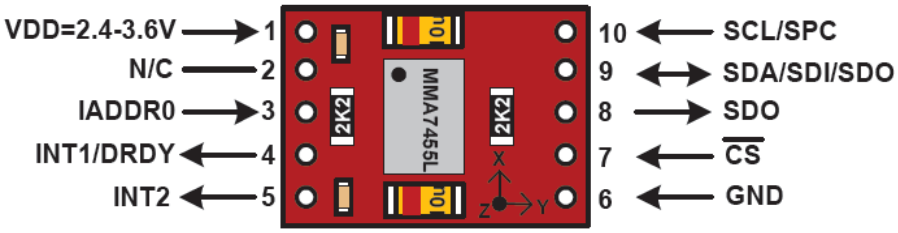


Fig.3 MMA7455L Breakout and its pin map

Range of the acceleration that MMA7455L can measure i.e. its ***measurement sensitivity*** is selectable. When use with 8-bit output mode, its sensitivity can be +/-2g, +/-4g and +/-8g.

Operation Mode: There are four modes of operation that can be configured using register $16 with MODE[1:0]

MODE [1:0]	Function
00	Standby Mode
01	Measurement Mode
10	Level Detection Mode
11	Pulse Detection Mode

1) Standby mode: it is a power saving mode. we can read and write its register but no new measurement. Device output is turned off

2) Measurement mode: The device continues to measure all three axes and convert to digital output. The sampling rate can be is either 125 Hz or 250 Hz. When conversion is complete, DataReady pin (DRDY) is high and DRDY bit of its **Status Register** is flagged. The DRDY pin is kept high until one of the three **Output Value Registers** are read.

3) Level Detection mode: Acceleration level (or Threshold) is set by user, once it is reached the Interrupt pin (INT) will go high and remain high until it is cleared. We can use this mode as a Motion Detection by using a condition of X or Y or Z > Threshold and a Freefall Detection by using a condition of X & Y & Z < Threshold.

4) Pulse Detection: In this mode a time window and a threshold level are set by user. Whenever the measured acceleration reaches the threshold within the time window, the Interrupt signal will be generated.

By applying the appropriate operation mode, several sensing functions that accelerometers are capable of detecting can be set up. These are motion, freefall, shock, vibration, and tilt.

Motion Detection

Motion detection is used to identify if an object is in use or not based on change in acceleration output. Usually, if the RMS value of the acceleration output is greater than a certain threshold (such as 1g) then the motion is detected

Freefall

Freefall is a sensing function that can be used to identify whether the object is falling freely. This is useful in general for many types of electronic equipment to shut down before impact. Freefall can also be used for warranty protection along with shock to identify how high an object has fallen to determine the approximate resultant force. For a robust algorithm there are various different freefall conditions that should be considered. These are linear freefall, projectile fall and rotational fall.

Shock

Shock is a sensing function of the accelerometer that is useful for warranty protection, shipping and handling and to detect the end of a fall condition. It is also used to detect tapping. Shock is a sensing function that can be difficult to detect with the consumer low-g accelerometers because shocks are typically high accelerations. The MMA7455 is capable of detecting up to 8g of acceleration. In some cases freefall can be used to determine.

Vibration

Vibration sensing is limited by the digital filtering in the accelerometer. The MMA7455 has a maximum sampling rate of 250 Hz. Therefore it is capable of detecting from DC to 125 Hz of vibration. The MMA7455 is suitable for these lower frequencies.

Tilt

Tilt is used for a lot of different applications. The cell phone market has exploded with opportunities for accelerometers to perform tilt functions. The most popular features are portrait/landscape orientation detection, scrolling, and menu selection.

In addition, it is a common practice to calibrate any sensor device before using it.
For more information, please read
http://cache.freescale.com/files/sensors/doc/app_note/AN3468.pdf

Basic Start-up procedure

The following are some simple steps to set up the accelerometer using I^2C.

1. Set up the I^2C communication connection between MCU and Accelerometer as in Table1. The device address is 1D
2. Configure the clock speed and all the pins required. The speed is set by adjusting the baud rate.
3. Write simple single byte Read and Write command to communicate to the device.
4. Write to the MMA7455L Register $16, sending in a value of 0x05 to set up the device for measurement mode with ±2g dynamic range.
5. Read the Control Register $16 to ensure that the value is correct (0x05).
6. Read the X, Y and Z registers and watch the outputs change.

The following are four simple rules of the I^2C bus to be aware of:

1. The SDA (data) and SCL (clock) cannot actively be driven high by any I^2C device. I^2C devices must use open-drain drivers. The logic is high by using the recommended external pull-up resistors.
2. The information on the data line is only read on the high phase of the clock.
3. Changing the level of the data is only allowed in the low phase of the clock except during start or stop conditions and this is how these events are signified.
4. When the bus is not busy SDA and SCL lines are pulled back to logic "1".

Table1. Accelerometer Breakout and Xpresso I^2C connections

ETT-MMA7455L signal name	LXpresso pin
VDD	Vcc
SCL	P0.11 (2.2k-4.7k pull-up resistor)
SDA	P0.10 (2.2k-4.7k pull-up resistor)
Gnd	Gnd

ETT-MMA7455L Example

```
I2C accel(P0_10, P0_11); // used with 2.2k-4.7k pull-up resistors.

const int addr_R = 0x3B; // Device Address is 1D hence Address to read:  111011
const int addr_W = 0x3A; // Device Address is 1D hence Address to write: 111010

int read(int n) { // Read 10-bit registers n and n+1
       char msb, lsb;
       accel.start();
       accel.write(addr_W);
       accel.write(n); // Starting register(L)
       accel.start();
       accel.write(addr_R);
```

```
	lsb = accel.read(0);
	accel.stop();
	accel.start();
	accel.write(addr_W);
	accel.write(n + 1); // Next register(H)
	accel.start();
	accel.write(addr_R);
	msb = accel.read(0);
	accel.stop();
	int a = (((int) msb) << 8) | lsb;
	if ((msb >> 1) == 1)
		a -= 1024; // If negative
	return a;
}

void write_8(int reg, int data) { // Write 8-bit data to register
	accel.start();
	accel.write(addr_W);
	accel.write(reg);
	accel.write(data);
	accel.stop();
}

void write_offset(int reg, int data) { // Write 11-bit data to offset registers L
and H
	accel.start();
	accel.write(addr_W);
	accel.write(reg); // Starting register(L)
	if (data < 0)
		data += 2048; // If negative
	accel.write( (char) (data & 0x00FF) );
	accel.stop();
	accel.start();
	accel.write(addr_W);
	accel.write(reg + 1); // Next register(H)
	accel.write((char) ((data & 0xFF00) >> 8));
	accel.stop();
}

void accelerometer_MMA_7455L() {
	accel.frequency(100000);
	wait(1);

	// 0x16: Mode[1:0] Control Register:
	// 00: Standby Mode
	// 01: Measurement Mode
	// 10: Level Detection Mode
	// 11: Pulse Detection Mode

	// 0x16: GLV[1:0] Control Register:
	// 00: 8g
	// 10: 4g
	// 01: 2g

	// DRPD = 1, GLVL = 00(8g), MODE: 01(Measurement).
	write_8(0x16, 0x41);

	// Clear offset registers
	write_offset(0x10, 0); // X axis
```

```
	write_offset(0x12, 0); // Y axis
	write_offset(0x14, 0); // Z axis
	wait(0.1);

	// Start autocalibration. Lay device down horizontal and do not move
	// Offsets must be multiplied by 2

	int offset[3];
	offset[0] = offset[1] = offset[2] = 0;
	for (int i = 0; i < 4; i++) {    // Average 4 readings
		offset[0] -= read(0x00);
		offset[1] -= read(0x02);
		offset[2] -= read(0x04);
		wait(0.2);
	}	// End autocalibration

	// Multiplied by 2 and divided by 4 to calculate the average
	offset[0] /= 2;
	offset[1] /= 2;
	offset[2] = offset[2] / 2 + 2 * 64; //Z axis output for 1g must be 64 in
10-bit mode

	write_offset(0x10, offset[0]);
	write_offset(0x12, offset[1]);
	write_offset(0x14, offset[2]);
	wait(0.1);

	// Read x, y, z with offset correction
	while (1) {
		printf("X = %3d\r\n", read(0x00));
		printf("Y = %3d\r\n", read(0x02));
		printf("Z = %3d\r\n", read(0x04));
		wait(0.1);
	}
}
```

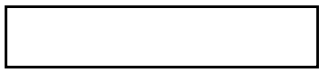
6. Experiment

1. Use LPC1769 create a file with a name “test.txt” to keep student ID and name. Read its content and display on the LCD screen.

2. Create a file on SD card with a name “SDtest.txt” to keep the current date and time. Read its content and display on the LCD screen. Note you need to first format your SD card to be FAT32.

3. Connect accelerometer breakout to the Xpresso board. Set up I^2C communication. Use it to measure the acceleration on X, Y, and Z. Display the data on the LCD screen and save to SD card.

4. Create character “X” on the center of the LCD screen. Make it move on the LCD screen according to its acceleration measured by the accelerometer.



5. Bonus Question: In some operation mode accelerometer can generate interrupt signal. It is of interest to use this to detect a shock or a freefall. Try to program accelerometer to be a freefall detector. It is easier to read Thai document however aware of discrepancy of pin connection between Freescale application note and ETT (Thai version) document.

7. Reference

[1] MMA7455L Technical Data, Freescale Semiconductor, 2009 available online at http://cache.freescale.com/files/sensors/doc/data_sheet/MMA7455L.pdf

[2] AN3458 Application Note**,** Freescale Semiconductor, 2009 available online at http://cache.freescale.com/files/sensors/doc/app_note/AN3468.pdf

[3] Online Document at http://developer.mbed.org/teams/mbed/code/SDFileSystem/