Lab 4

ET014G Programming of Embedded Systems

Erlend Røed Myklebust

Spring 2016

Mittuniversitetet
MID SWEDEN UNIVERSITY

NTNU – Trondheim
Norwegian University of
Science and Technology

Task) This lab is to use the potentiometer, ADC, TC, USB with CDC task and SD card with FAT file system, to control the logging of potentiometer raw ADC reading to a file in SD card by COM port software interface on PC.

This lab contains several elements from Lab 3, and because of this, virtually all of the initialization in this solution is identical to the one from Lab 3. It would therefore seem that most of the code "wrote itself". However, there were a number of significant challenges involved in developing the final code for this lab. Most notably of which was how to correctly configure and use the FAT file system driver. This was complicated for two reasons. One, the FAT driver utilizes another driver called Memory Access Control for interfacing between memory modules, and it was not at all obvious how to configure this to interface with the SD/MMC over SPI. Two, the only available example code in ASF v.3.29.0 is very comprehensive and hard to read. There is also a compatibility issue between the USB CDC driver and PM Power Manager in ASF v.3.29.0. Luckily, I had already discovered this in Lab 2 and knew the solution was to use System Clock Control instead. Besides all of this, the challenge was mainly a question of developing a proper code structure capable of handling everything correctly and efficiently. Incidentally, that is why I ended up with two separate solutions to this task, one with a good structure, version 2.0, and then the *other* one.

The project name for this solution is Lab_4_Task_v.2.0. It was developed using Atmel Studio 6.2 with ASF v.3.29.0, AVR32/GNU compiler and linker v.4.4.7, and relies on the following ASF drivers: *ADC, CPU Cycle Counter, FAT file system, Generic board support, GPIO, INTC, LCD Display DIP204B, Memory Control Access, SD/MMC, System Clock Control, TC Timer/Counter* and *USB CDC.* See Appendix A for the full code.

Note: functions for *ADC, USB CDC, FAT, INTC, LCD Display, SD/MMC* and *TC* are included in separate .h-files to improve the readability of *main.c.*

Before compiling the code, a number of settings has to be defined to make the necessary functions available from the FAT file system and the Memory Access Control drivers. Depending on the ASF version, either LUN_2 or LUN_4 needs to be enabled in *conf_access.h* to set SD/MMC over SPI as target for the Memory Access Control. Next in *conf_explorer.h*, FS_FAT_32 must be set to *true* to allow FAT32 and FS_LEVEL_FEATURES must contain both *FSFEATURE_READ* and at least *FSFEATURE_WRITE* to enable reading and writing of FAT formatted files. If these settings are not made, and the program still tries to access the functions associated with them, the program will not compile successfully.

The program starts by configuring the CPU clock to 48 MHz and the PBA clock to 12 MHz. These settings are defined in the file *conf_clock.h* and are activated by the function call *sysclk_init()*. Next is the usual initialization of the board, LCD display, INTC, SD/MMC and ADC. Then the TC is configured to generate an interrupt every 20 ms, but unlike in Lab 3, the counter is not started right away. In *conf_usb.h*, the configuration of the CDC serial port is defined with 115200 baudrate, 8 data bits, no parity and 1 stop bit, and is then activated. Finally, the ADC is started and the program enters the main loop.

The flowchart in Figure 1 describes the control structure in the main while loop and gives a short explanation of which actions are performed in various states of the program. Five identifiers govern the structural flow:

- *commence_program* – responsible for starting the "actual" program. Its intention is to ensure that a user has opened a terminal window on his/her PC before the program writes anything to the CDC. It is initially set to *false* and once it is set to *true,* it is never reset to *false.*
- *loop_identifier* – main indicator for the loop, responsible for determining what actions the program should perform next. *IDLE* is the initial and default state, *INT_TC* is set by the TC interrupt, *UPDATE_INITIAL* is set once by a push button interrupt, *UPDATE_START* is set by a valid start-command and *UPDATE_STOP* is set by a valid stop-command.
- *mounted* – indicates whether a FAT partition is mounted and a measurement file has been created or not. Valid values are *true* and *false.*
- *running* – indicates whether an ADC measurement and logging session is active or not. Valid values are *true* and *false.*
- *command* – contains the currently active command. *NO_COMMAND* is the initial and default command, *START_COMMAND* is set when a user types 'start' followed by a break-line (enter key) in the terminal window and *STOP_COMMAND* is set when a user types 'stop' followed by a break-line in the terminal window.

Figure 2 gives a simplified description of the joint functionality offered by the *build_cmd()* – function and the *decode_command()* – function, that is performed at the end of each main loop execution. Figure 3 describes the interrupt routines for the TC interrupt and the push button interrupt. Figure 4 shows the code functions for mounting the FAT partition, creating a new file and appending a given file with data.

In short, a typical use case would happen like this: When a user first powers on the EVK1100, he/she is met with a message saying "Open PC terminal. Press PB0 to start". Upon pushing PB0, the *commence_program* is set to *true* and *loop_identifier* is set to *UPDATE_INITIAL*, activating the main loop switch structure. Because *loop_identifier* is set to *UPDATE_INITIAL*, the first thing that happens is FAT partition mount, followed by activating the TC and writing an initial message to the CDC in which the user is prompted to enter a valid command. By writing start + enter, an ADC measurement run is started by first creating a measurement text file and then relying on the TC interrupt to trigger an ADC measurement every 20ms. For each TC interrupt, a single ADC value is measured, the current file is opened, the data is appended to this file and the file is closed. The user then types stop + enter into the terminal window, causing the measurements to stop by simply denying the TC interrupt routine the possibility of setting the *loop_identifier*, hence making it impossible to reach the *INT_TC* state in the main loop switch structure. The user can now power down and extract data from the SD/MMC.

Finally, Figure 5 and Figure 6 depicts the terminal window output and the measurement results for an example measurement run.

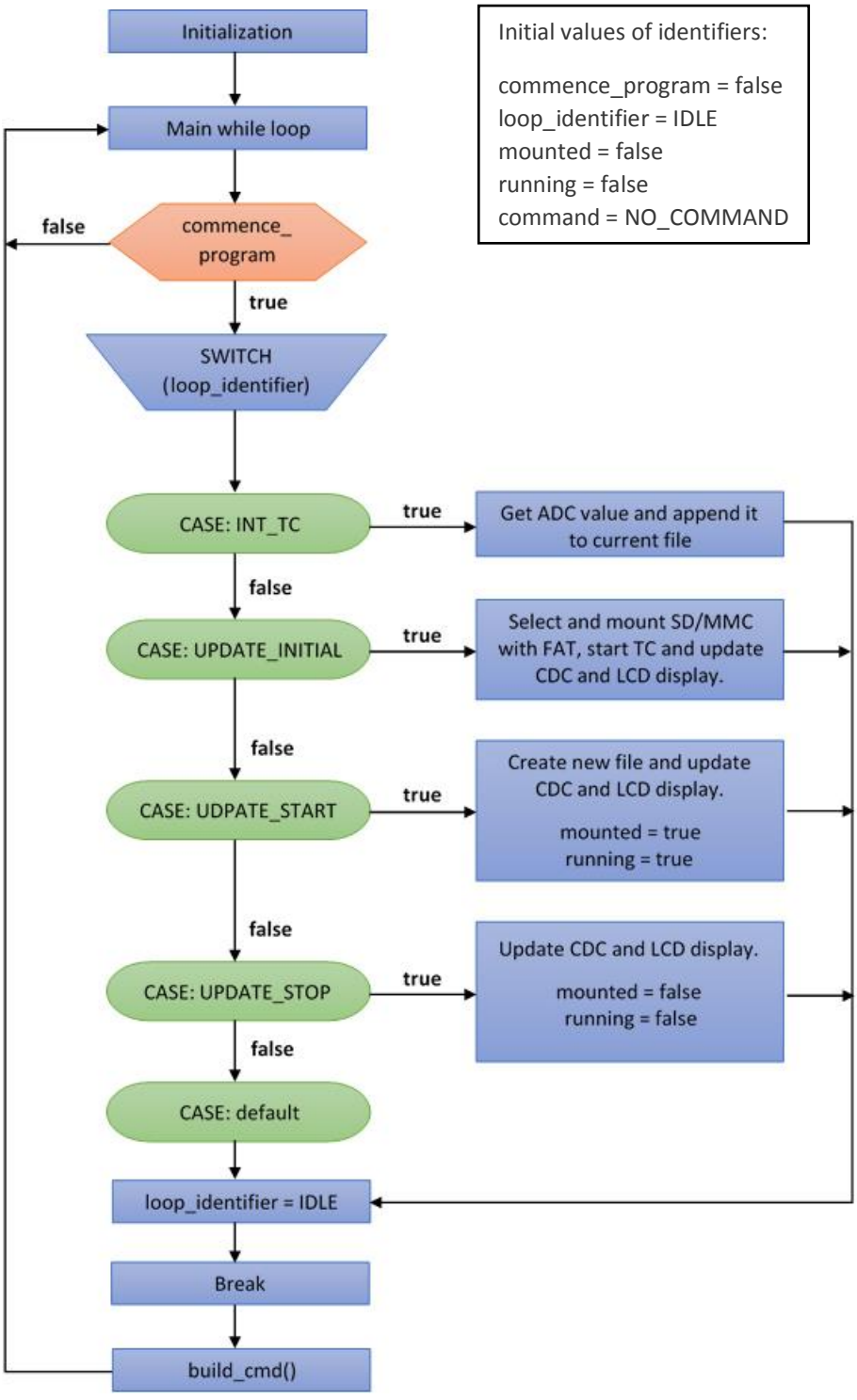


Figure 1. Flowchart describing the main loop of the program

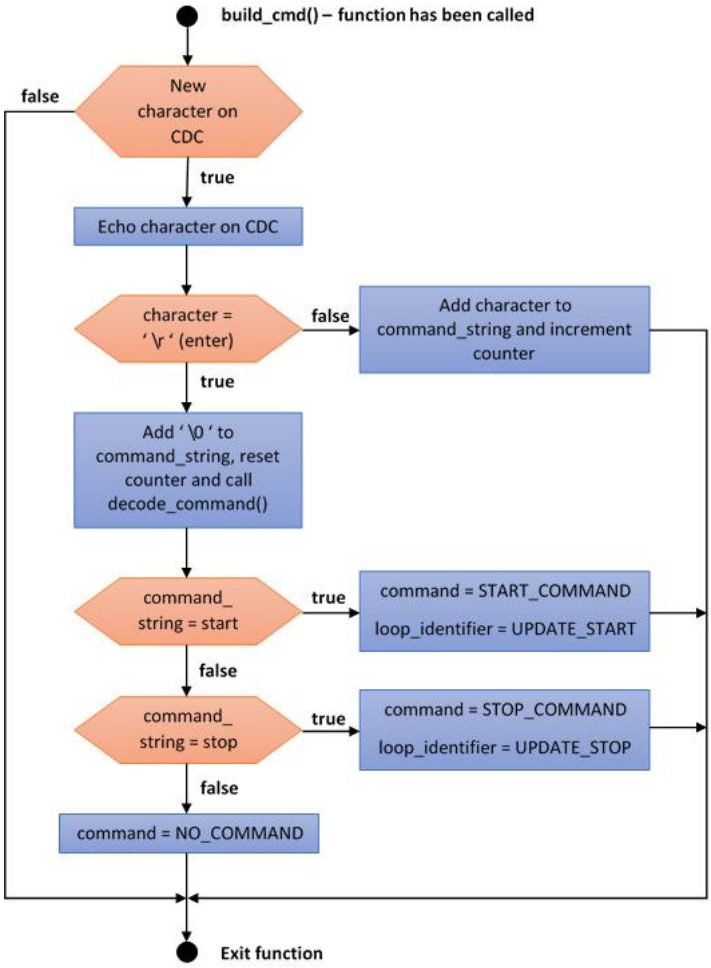


Figure 2. Simplified flowchart describing the build_cmd() and decode_command() functions

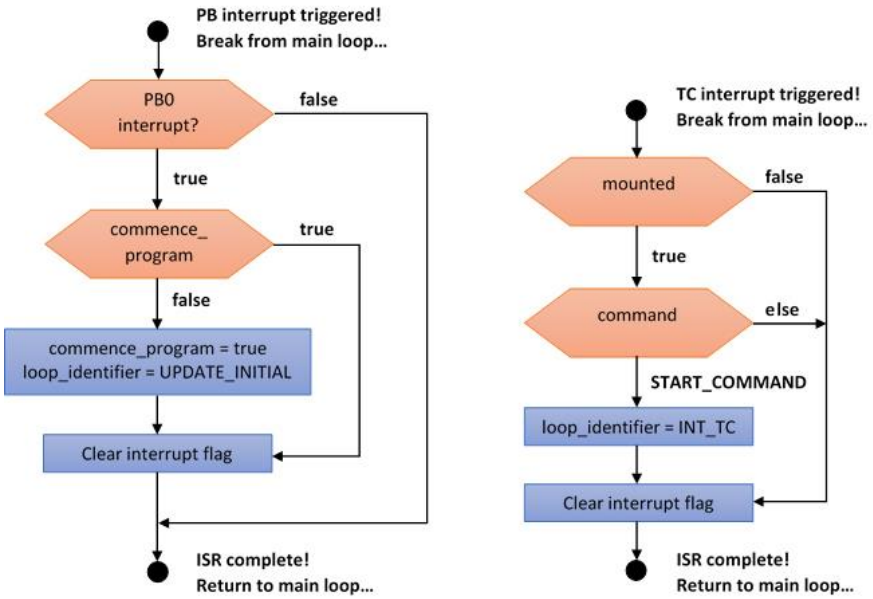


Figure 3. Flowcharts for the push button and the TC interrupt routines

```
static void erm_fat_mount_sdmmc(void)
{
	// Select and mount the FAT partition on the SD/MMC
	nav_reset();
	nav_drive_set(0);
	nav_partition_mount();
}


static void erm_fat_create_file(char* file)
{

	sprintf(file, "Measurement_%lu.txt", erm_current_file_counter);

	nav_filelist_reset();

	while (nav_filelist_findname(file, false))
	{
		erm_current_file_counter++;
		sprintf(file, "Measurement_%lu.txt", erm_current_file_counter);
	}

	// Create the file
	nav_file_create((FS_STRING) file);

}

static void erm_fat_file_append(char* file, uint32_t input)
{
	uint8_t i;
	char    temp_string[8];

	// Set navigator to desired file
	if (!nav_setcwd((FS_STRING) file, true, true))
	{
		erm_cdc_println("error selecting sd/mmc");
	}
	else
	{
		// Open file
		file_open(FOPEN_MODE_APPEND);
		// Convert pot_value to a string
		sprintf(temp_string, "%lu\x03", input);
		// Write value to file
		i = 0;
		while ((int) temp_string[i] != 0x03)
		{
			file_putc((int) temp_string[i]);
			i++;
		}
		// Add CR and LF
		file_putc('\r');
		file_putc('\n');
		// Close file
		file_close();
	}
}
```

Figure 4. Functions for mounting FAT partition, creating files and appending data to a file

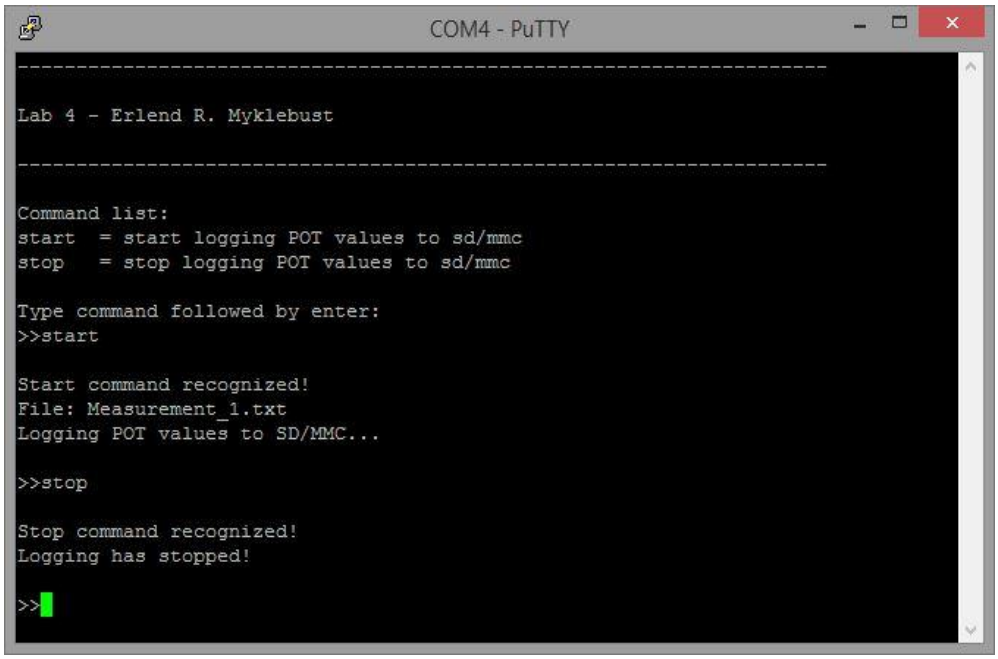


Figure 5. Example terminal output

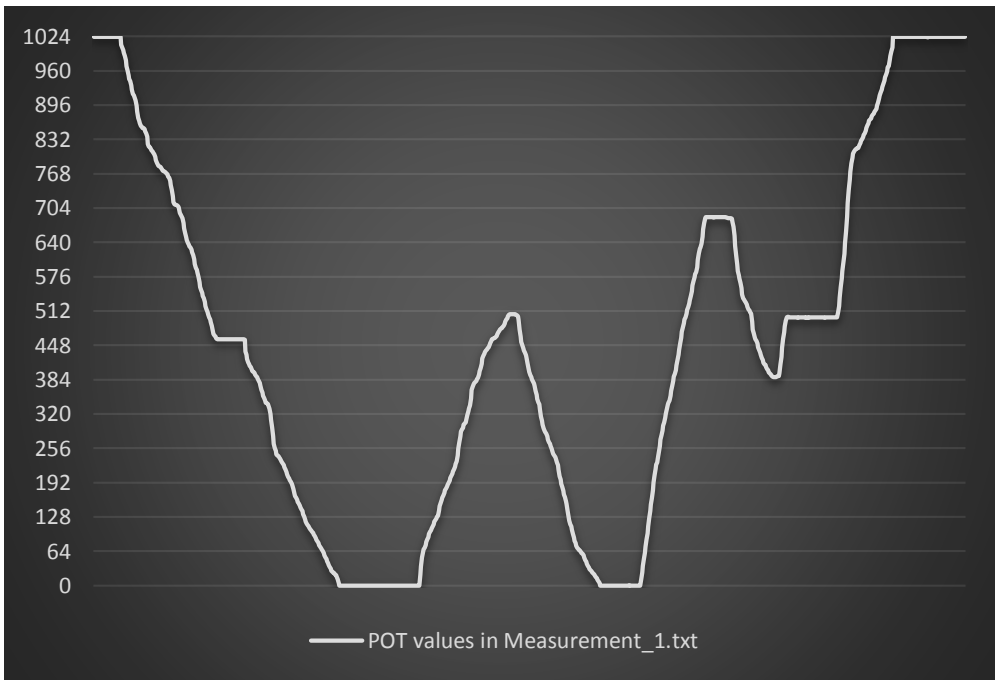
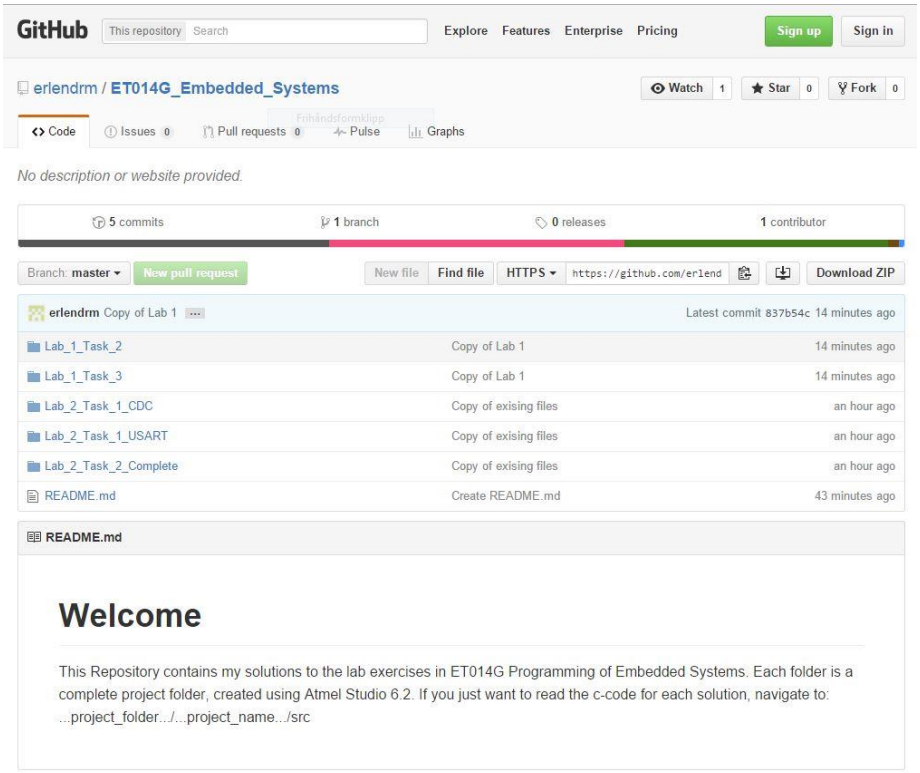


Figure 6. Resulting POT values from example measurement

END

Appendix A: Code Repository (GitHub)



The GitHub repository "ET014G_Embedded_Systems" contains my code solutions for this laboratory exercise in form of complete project folders.

If you only wish to read the code files for each solution, simply navigate to *Lab_4_Task_v.2.0/Lab_4_Task_v.2.0/src* from the main repository. There you will find the *main.c* and all the additional include files.

Please use the following static link to reach the repository:

github.com/erlendrm/ET014G_Embedded_Systems