THE UNIVERSITY OF CAMBRIDGE

Portable FM Radio

Using Atmel ATMEGA644P, Si4703 and Nokia 5100 LCD Display

Martin Marinov

Usage and functions

RDS INDICATOR
If RDS is being received, this icon will show up

VOLUME LEVEL
Shows the output volume level 0 to 15

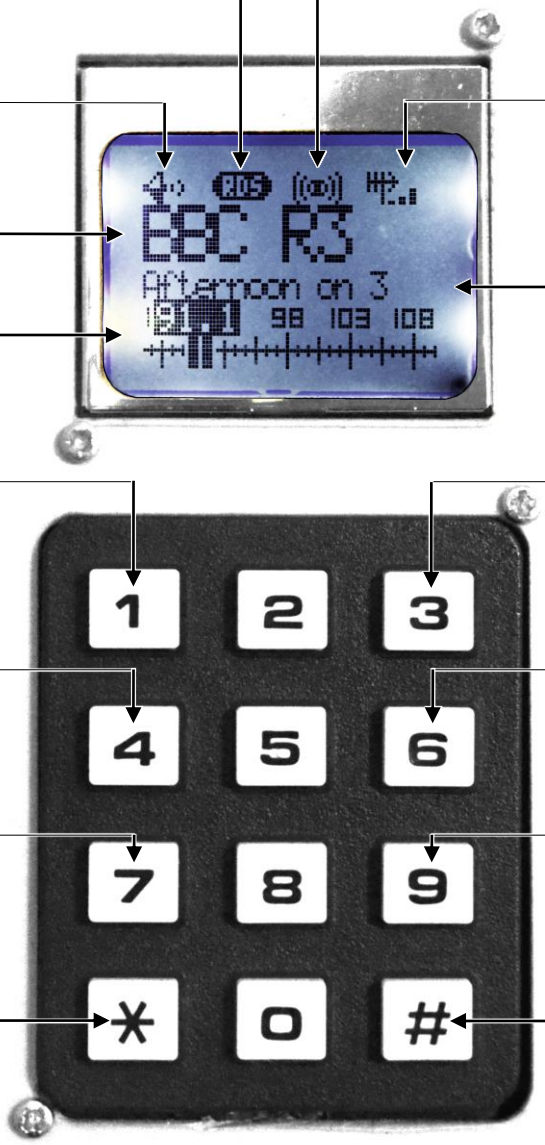
PROGRAM SERVICE
Name of station via RDS

SCALE
Current frequency on the FM scale

STEREO INDICATOR
If STEREO is available, this icon will show up

SIGNAL STRENGTH
Shows the signal strength. The range is 20 to 60 dBµV

RADIO TEXT
Can contain song that is currently playing or show that is running



SEEK LEFT
Finds the previous broadcasting FM station

TUNE LEFT
Lowers the current frequency with 0.1 MHz

SCROLL LEFT
Scrolls the radio text from the RDS one character to the left

LOWER VOLUME
Lower the volume output on the headphones

SEEK RIGHT
Finds the next broadcasting FM station

TUNE RIGHT
Increases the current frequency with 0.1 MHz

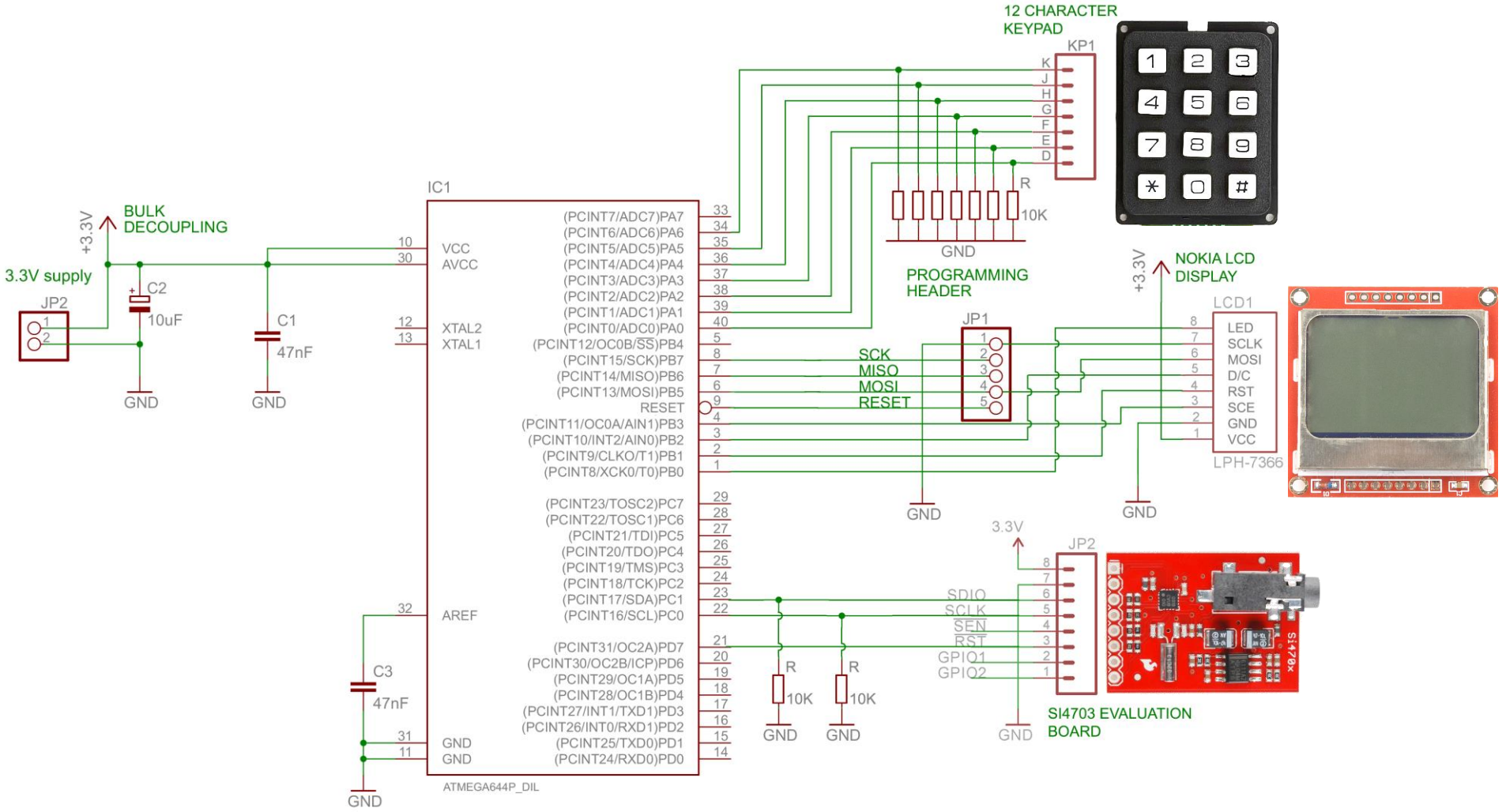
SCROLL RIGHT
Scrolls the radio text from the RDS one character to the right

INCREASE VOLUME
Increases the volume output on the headphones

SPECIFICATIONS

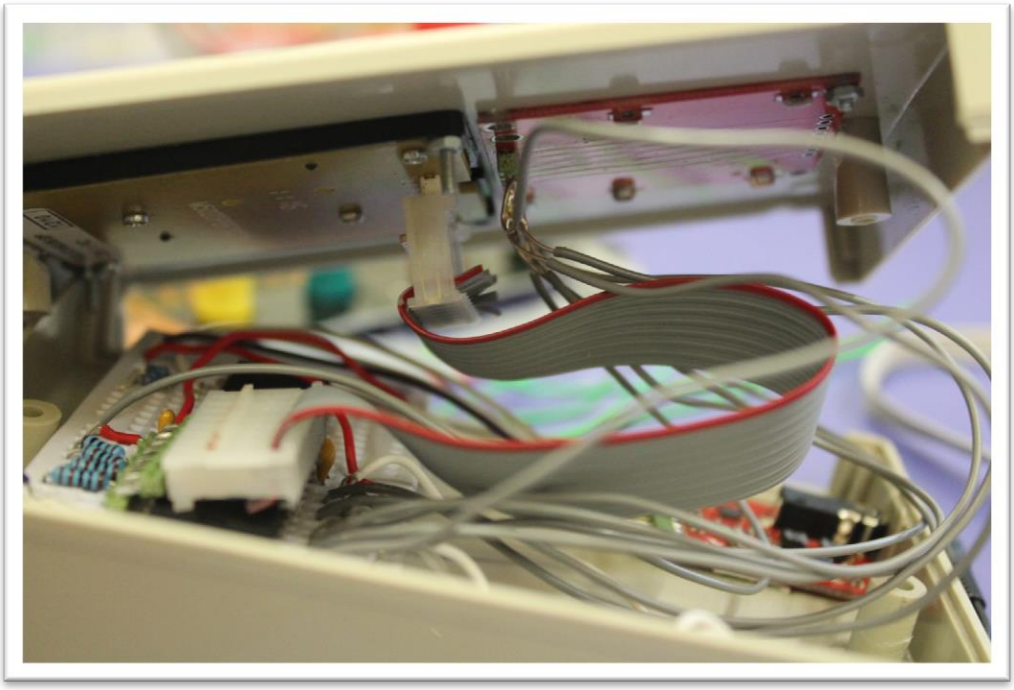
Frequency range	87.5 – 108 MHz
Tuning step	100 KHz
Stereo	Supported
RDS Features	Radio Text and Program Service
Idle power usage	25 mA
Peak power usage	41 mA
Battery life	16 h on two AA batteries
LCD Backlights	Turns off if buttons are not used
Price of hardware parts	£35.14

Hardware


12 CHARACTER KEYPAD
KP1
K
J
H
G
F
E
D
1 2 3
4 5 6
7 8 9
* 0 #
R
10K
GND
PROGRAMMING HEADER
IC1
+3.3V
BULK DECOUPLING
3.3V supply
JP2
C2
10uF
C1
47nF
GND
VCC
AVCC
XTAL2
XTAL1
RESET
AREF
GND
GND
(PCINT7/ADC7)PA7
(PCINT6/ADC6)PA6
(PCINT5/ADC5)PA5
(PCINT4/ADC4)PA4
(PCINT3/ADC3)PA3
(PCINT2/ADC2)PA2
(PCINT1/ADC1)PA1
(PCINT0/ADC0)PA0
(PCINT12/OC0B/SS)PB4
(PCINT15/SCK)PB7
(PCINT14/MISO)PB6
(PCINT13/MOSI)PB5
(PCINT11/OC0A/AIN1)PB3
(PCINT10/INT2/AIN0)PB2
(PCINT9/CLKO/T1)PB1
(PCINT8/XCK0/T0)PB0
(PCINT23/TOSC2)PC7
(PCINT22/TOSC1)PC6
(PCINT21/TDI)PC5
(PCINT20/TDO)PC4
(PCINT19/TMS)PC3
(PCINT18/TCK)PC2
(PCINT17/SDA)PC1
(PCINT16/SCL)PC0
(PCINT31/OC2A)PD7
(PCINT30/OC2B/ICP)PD6
(PCINT29/OC1A)PD5
(PCINT28/OC1B)PD4
(PCINT27/INT1/TXD1)PD3
(PCINT26/INT0/RXD1)PD2
(PCINT25/TXD0)PD1
(PCINT24/RXD0)PD0
ATMEGA644P_DIL
C3
47nF
GND
SCK
MISO
MOSI
RESET
JP1
GND
+3.3V
NOKIA LCD DISPLAY
LCD1
LED
SCLK
MOSI
D/C
RST
SCE
GND
VCC
LPH-7366
GND
R
10K
R
10K
GND
GND
SDIO
SCLK
SEN
RST
GPIO1
GPIO2
3.3V
JP2
GND
SI4703 EVALUATION BOARD

The enclosure of the unit was produced from a project box via laser cutting. The display and the keyboard are centred onto the face of the box and bolted in place with two bolts each. There is a hole on the top of the unit for headphone output (**1**). The Si4703 FM Radio IC has amplified headphones output built in, so there was no need to worry about audio. Furthermore the headphones act as an aerial so there was no need for an external antenna. The IC itself is super glued into place. There is a programming header port on the side of the unit (**2**) so that the Atmel microcontroller could be reprogrammed without opening the enclosure. The battery compartment is bolted to the back side on the outside of the box. The power supply for the project is provided by two AA batteries.

The components are fitted to a breadboard stuck to the back of the inside of the enclosure with a double sided tape. The breadboard hosts the Atmel microcontroller and the small electrical components – the resistors and capacitors. There is a ribbon cable connecting the keyboard and the PORTA of the microcontroller. The Nokia LCD and the FM Radio IC have wires soldered to them and tucked into the breadboard.



This design makes the components easily changeable and all of the expensive ones can be salvaged and returned for use on future projects.

Software

The software could be found on Github[1]. It is released open source with no particular license and without any warranty. The Makefile[2] can be used to compile the code, program the microcontroller and set the fuses. The fuses values for the hardware configuration showed above are *0x99* for *hfuse* and *0xE2* for *lfuse*. Prerequisites are having AVR Libc[3] installed. The Makefile assumes that the USB debugger is connected to the *ttyUSB0* serial port. You need to change that if this is not the case with your setup. The possible targets for the Makefile are:

- make **all** – compiles the executable *main.hex*
- make **program** – sends the executable using *avrdude* over to *ttyUSB0*
- make **fuses** – sets the fuses to *hfuse* = *0x99* and *lfuse* = *0xE2*
- make **clean** – cleans the compilation artifacts

The quickest way to get everything up and running (once the hardware is properly connected) is to type in *make fuses* followed by a *make program*.

The firmware is divided in modules a.k.a. drivers. It allows taking parts of the project and integrating them within different context. The different modules are outlined below.

The Si4703 Driver (fm.c)

The code is based on Sparkfun's Arduino driver from Github[4]. The driver establishes I2C conversation between the microprocessor and the Si4703. The commands are executed by writing and reading the registers of the Si4703 which are transmitted over the I2C connection.

The Hardware Configuration should be written to *config.h*. The driver expects the following pre-processor defines to be present in the file:

- FM_RESET - the pin that the reset is connected to
- FM_PORTRESET - the port of the above pin
- FM_DDRRESET - the DDR of the above pin
- FM_SDIO - the pin where the sdio is connected to
- FM_PORTSDIO - the port of the above pin
- FM_DDRSDIO - the DDR of the above pin

The following defines are required by *i2c.c*. Have a look at it for more information

- TW_DDR, TW_PIN, TW_PORT, TW_SCK, TW_SDA

The pins that the radio is connected to don't need to be pre-initialized, the *fm_turn_on* function will do that for you.

The API itself is:

```
/* Sets up the pins, turns on the radio and starts the i2c conversation  */
uint8_t fm_turn_on(void);
```

[1] https://github.com/martinmarinov/AATMEGA644PandSi4703
[2] Based on http://www.cl.cam.ac.uk/teaching/1314/P31/code/workbook1_Makefile
[3] http://www.nongnu.org/avr-libc/
[4] https://github.com/sparkfun/Si4703_FM_Tuner_Evaluation_Board

```
/* Set frequency. The frequency is an integer of the real frequency in MHz
multiplied by 10. For example if you want to tune to 91.8 MHz, you would set
channel == 918 */
uint8_t fm_setChannel(uint16_t channel);

/* Sets the output volume on the headphones. Values are from 0 to 15 */
uint8_t fm_setVolume(uint16_t volume);

/* Returns the current FM channel, RSSI level in dBµV and a boolean flag
indicating whether stereo signal is currently present. */
uint16_t fm_getChannel(uint8_t * rssi, uint8_t * stereo);

/* Finds the next frequency that has an RSSI higher than a predefined
threshold. This effectively attempts to tune to the next/previous valid FM
broadcast.
If you want to seek up, set seek_up == 1, if you want to seek down, set
seek_up == 0. Function returns 1 on success */
uint8_t fm_seek(uint8_t seek_up);

/* Read the RDS value and return the Program Service (string of 9 chars,
including the last null) and Radio Text (string of 65 chars, including the
last null).
Returns either RDS_NO or RDS_AVALABLE or RDS_FAKE. RDS_FAKE means that the PS
will contain a string that shows the current frequency in MHz to display in
case of no RDS is available. */
int fm_readRDS(char* ps, char* rt);

/* auto discover the address of the device. For debugging purposes
returns the i2c address or 0xFF for an error
radio won't work after this command */
uint8_t fm_get_isc_address(void);

/* A string utility that takes a channel and prints it into the string with
an offset start. */
int str_putrawfreq(char * str, uint16_t freq, int start);
```

The driver was heavily modified in the part concerning the RDS (Radio Data Text) functionality. The original driver had some very buggy support of RDS Program Service. The current implementation could be extended to decode any information that the RDS could be carrying. It provides access to the CRC corrected 8 bytes transmitted. This could be used to obtain the current RDS group that is being broadcast and decode specific features. For reference, see the USING RDS/RBDS WITH THE Si4701/03[5] document and the G Laroche[6] website.

The Keypad Driver (buttons.c)

The keypad has 12 usable buttons. There are 3 control wires (D, E and F) and 4 output wires (G, H, J and K). When a control wire is set to high, a button press can trigger a physical contact between the control wire and one of the output wires. In order to determine which button has been actually pressed, the driver needs to try switching on and off all of the control wires one by one.

The hardware configuration file *config.h* needs to contain the following pre-processor defines:

- BUTTONS_PORT – the port to which the keypad is connected
- BUTTONS_DDR – the DDR of this port
- BUTTONS_PIN – the PIN of this port

[5] https://support.silabs.com/attach/BCA/BCA-764/35383_AN243%20Using%20RDS_RBDS.pdf
[6] http://www.g.laroche.free.fr/english/rds/fonctions/fonctions_rds.htm

The ports that the keypad is connected to don't need to be pre-initialized, the *buttons_init* will do that for you. The driver assumes that the keypad is connected to pins 0 to 6 on the *BUTTONS_PORT* port. Pins 0 to 2 should be connected to the three control wires of the keypad (see section Hardware for an example).

The API is:

```
/* read from config BUTTONS_PORT and BUTTONS_DDR. Buttons should be connected
from 0 to 6 */
void buttons_init(void);

/* returns 1 on success and 0 on failure
val is the char
event is an event BUTTONS_UP or BUTTONS_DOWN
if function returns 0, the value of val and state is undefined */
uint8_t buttons_poll(uint8_t * val, uint8_t * event);
```

The Nokia LCD Driver (lcd.c)

The communication to the Nokia 5100 LCD display is done via SPI. The driver is based on *nokia_lcd.c*[7]. The configuration file *config.h* needs to contain

- SPI_PORT, SPI_DDR, SPI_PIN, SPI_SCK, SPI_MISO and SPI_MOSI

in order to initialize the SPI communication. The *lcd.c* also defines *PIN_LED* which specifies which pin is used to control the display backlight. This is currently hardcoded to *PB0*.

The original driver is heavily modified so that it can support frame buffering. Basically a buffer is kept in memory and different operations are undertaken ontop of it. This allows a smooth synchronized animation. The buffer is sent to the LCD with the *lcd_repaint* function. Therefore the normal way to draw a new frame on the screen (without reusing whatever is in the buffer) is:

```
lcd_clear(); // clear the frame buffer
/* do some drawing here */
lcd_repaint(); // show whatever we have drawn so far on the LCD
```

My experiments show that this allows for smooth animation up until about 10 frames per second. Although updates can go way quicker than that, the LCD cannot keep up updating its pixels fast enough and things become blurry.

The supported API is:

```
/* A bitmap that can store several images to be shown on screen */
typedef struct bitmap bitmap_t;
static struct bitmap {
	uint8_t width;
	uint8_t height; // row number 0 to 5
	uint8_t images; // how many frames does this bitmap have
	uint8_t data[]; // the actual pixel data
};

/* If state == 1, backlight is turned on, otherwise backlight is turned off
*/
void lcd_backlight(uint8_t state);

/* Send the frame buffer to the LCD */
```

[7] http://www.cl.cam.ac.uk/teaching/1314/P31/nokia/nokia_lcd.c

```
void lcd_repaint(void);

/* Show a single char in black of size 5x8 px using the in-built 5x8 px font.
Y is row number (0 to 5). X is a pixel (0 to 83) */
void lcd_char(uint8_t character, uint8_t x, uint8_t y);

/* Show a single char in white of size 10x16 px using the in-built 5x8 px
font (scaled by 2). Y is row number (0 to 5). X is a pixel (0 to 83) */
void lcd_charlargewhite(uint8_t character, uint8_t x, uint8_t y);

/* Show a null terminated string in black. Y is row number (0 to 5). X is a
pixel (0 to 83)  */
void lcd_string(uint8_t *characters, uint8_t x, uint8_t y);

/* Show a null terminated string in white. Y is row number (0 to 5). X is a
pixel (0 to 83)  */
void lcd_stringwhite(uint8_t *characters, uint8_t x, uint8_t y);

/* Show a null terminated string. Each char is 10x16 px (the in-built 5x8 px
font scaled by 2) */
void lcd_stringlarge(uint8_t *characters, uint8_t x, uint8_t y);

/* Show a byte. See remarks for y in lcd_char */
void lcd_uint8(uint8_t val, uint8_t x, uint8_t y);

/* Show a short. See remarks for y in lcd_char */
void lcd_uint16(uint16_t val, uint8_t x, uint8_t y);

/* Show a short in binary. See remarks for y in lcd_char */
void lcd_uint16bin(uint16_t val, uint8_t x, uint8_t y);

/* Show a byte in hexadecimal. See remarks for y in lcd_char */
void lcd_uint8hex(uint8_t val, uint8_t x, uint8_t y);

/* Show a byte in binary. See remarks for y in lcd_char */
void lcd_uint8bin(uint8_t val, uint8_t x, uint8_t y);

/* Clear frame buffer */
void lcd_clear(void);

/* Initialize SPI connection with the LCD */
void lcd_init(void);

/* Draw a bitmap. Y is row number (0 to 5). X is a pixel (0 to 83) */
void lcd_bitmap(bitmap_t * bit, uint8_t id, uint8_t x, uint8_t y);

/* Efficiently fills a rectangle in black. Width is in pixels from 0 to 83
and height is in pixels from 0 to 47. */
void lcd_fillrect(uint8_t x, uint8_t y, uint8_t width, uint8_t height);

/* Efficiently fills a rectangle in white. Width is in pixels from 0 to 83
and height is in pixels from 0 to 47. */
void lcd_clearrect(uint8_t x, uint8_t y, uint8_t width, uint8_t height);

/* Make an individual pixel black */
void lcd_setpixel(uint8_t x, uint8_t y);
```

The frame buffer format is each byte spans 8 vertical pixels where each bit addresses a particular pixel. This means that each pixel takes space of only one bit. This fact is exploited by the *lcd_fillrect* and *lcd_clearrect* functions that do clever bit arithmetic to do as little operations as possible onto the underlying array.

The bitmap format also expects the same pixel format. It also contains some metadata as number of rows and width in pixels of the bitmap that needs to be drawn. A bitmap can contain several

"frames" for an animation or a dynamic icon (such as the sound and RSSI icons). A bitmap could be generated using The BMP Designer that was specifically developed for the project.

The Main Loop (main.c)

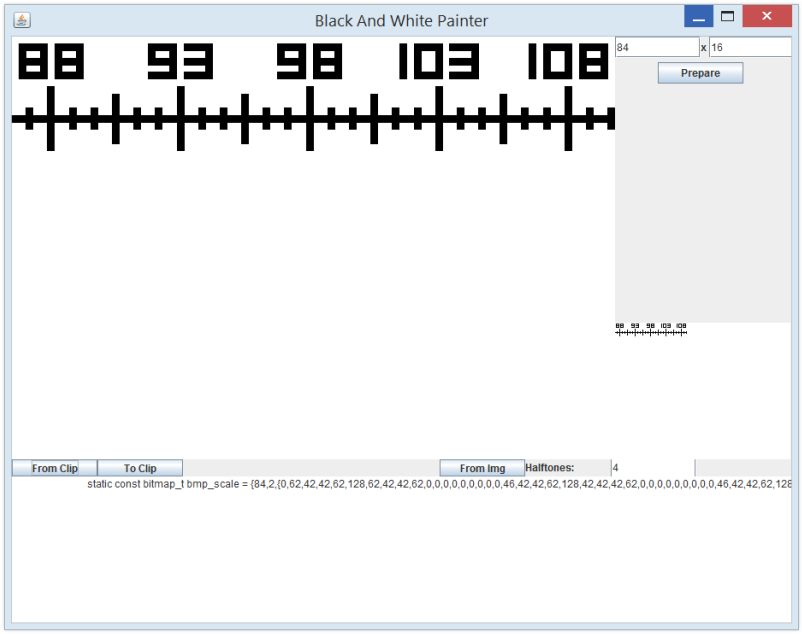
The main loop is the "glue" that holds together all of the parts and utilizes the APIs exposed by the different modules. It constantly polls the keypad driver and the RDS information from the fm_driver and repaints the screen if needed. There is a dedicated method called *drawScreen* that takes care of the actual drawing of the UI content by utilizing the LCD API. Also, the main loop is responsible for toggling the LCD backlight on on a key press and off after a timeout. This is how it works in pseudo code:

```
while (1)
{
	if (counter timeout)
		FM backlight off;
	else
		counter tick;

	if (buttons_poll(&letter, &state) && state == BUTTONS_PRESSED) {
		counter reset;
		FM backlight on;

		switch(letter) {
		case '1':
			FM seek up; break;
		case '3':
			FM seek down; break;
		case '#':
			FM raise volume; break;
		case '*':
			FM lower volume; break;
		case '6':
			FM increase frequency; break;
		case '4':
			FM decrease frequency; break;
		case '9':
			Scroll FM Radio Text right; break;
		case '7':
			Scroll FM Radio Text left; break;
		}

		Redraw screen;

	} else if (fm_readRDS(rdsname, rdsrt))
			Redraw screen;
}
```

The BMP Designer

The bitmap designer was made as a Java application that can produce a *bitmap* structure that could be directly copied/pasted into the c code and rendered with the LCD library function *lcd_bitmap*. The GUI is intuitive, it allows for importing a coloured image and using dithering it makes it look like a grayscale. Alternatively, bitmaps could be designed click by click manually (as most of the graphics used in the project were).

The application does not use external libraries and can be shipped as an executable jar file so it will work on any OS that supports Java just by double clicking on it (or running it with the **java –jar** command).



Problems Encountered

Wrong board

I originally ordered a Si4703 via eBay but instead I received a DTMF decoder. They have a striking visual resemblance since both of them have an audio jack and the same number of inputs/outputs. I only found out about the problem when I started connecting it to the Atmel microcontroller and discovered that the outputs were different. I had to reorder another Si4703 with special delivery, this time via Sparkfun and wait for it to arrive again. In the meantime, I worked on the keyboard and the Nokia LCD display.

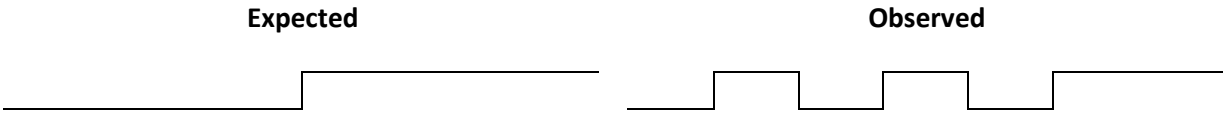
I2C

There was an issue with the I2C that took me about two days to discover. I was having troubles getting any response from the device. I wasn't sure the wiring was correct. After I measured that indeed there was some activity happening over the wires, I thought I was sending the data to the wrong I2C address. I wrote a function that will do auto discovery and try all of the possible I2C addresses (128 in total) to see which one will the Si4703 respond to.

I managed to find the correct one but the tuner still failed to respond properly. The communication was happening and no error codes were produced. The values I was getting from the Si4703 did not make sense so something was corrupting them. I thought the board was not working but then I added an I2C delay after each call to the I2C write and read and this fixed the issue and the communication started working without data being corrupted.

Keypad Bounce

Designing the driver for the keypad was a challenge because of the so-called "bounce" of the keypad buttons. When a button is pressed or released, there change of the state of the output wire is not instantaneous from low to high but rather has a few changes from low to high to low to high as shown in the diagram below.



The *buttons_poll* function logic records the state of the last event and uses it to compare it with the current one in order to return a correct event value – a rising edge (*BUTTONS_DOWN*) or a falling edge (*BUTTONS_UP*) or no change (*BUTTONS_PRESSED* or 0 depending on current state). This means that the main loop can decide to only perform an action when the button is being held (*BUTTONS_PRESSED*) so that even if there is bounce, this will not affect the responsiveness.

For example, if the user wants to turn up the volume, they would hold the volume up button (currently button **#**). If there is any bounce, the volume will still go up – as the user wants to. This is also true for tuning – if they want to arrive at a particular frequency (using button **6**) they will keep the button pressed which will keep generating the *BUTTONS_PRESSED* events. The bounce can only affect some fine gestures required because there is no guarantee, for example, that a quick press of the button won't trigger a double click rather than just one click. Fortunately operating an FM radio does not require such a sensitive control as shown from my experience with the unit.

Fonts

I had a desire to show the station name with a bigger font so it is clearly visible from a distance. This required a larger font. The font that comes with the Nokia LCD screen driver is 5 pixels wide and 8 pixels high. Because of efficiency and the fact that each byte on the frame buffer represents 8 vertical pixels, it is more efficient to have font heights in multiples of 8 (the same applies for bitmaps).

I decided to create my own font with height 16 pixels. For this reason, I modified the BMP designer to use fonts build in the computer to generate C code that could be used as a font on the Nokia LCD screen. Unfortunately if I wanted 10 pixels wide font, each character would take up 2 * 10 = 20 bytes. Some radio stations have non-alphanumeric characters in their names, so I had to span a large number of characters. The built-in font has 96 characters. This would have meant that I had to store 1900 bytes of data (the current font only takes up 450 bytes).

This was too much for the Atmel and the program failed to compile. Keep in mind that there are also other bitmaps as well. So the simplest solution was to use the font that is already built in (the 5 pixels by 8 pixels one) but to scale it on-the-fly to 10x16 pixels by interpolation. This approach was used in the final product.

Further developments

Reducing CPU cycles

The current setup uses the 8MHz clock speed of the Atmel microcontroller. I believe it could be possible to achieve the same level of usability with a lower clock speed. This in theory could lower the power consumption. The challenge would be to change the values that are dependent on the clock speed and test which would be the lowest clock speed that would still support the same level of interactivity as what the product currently has.

Using Interrupts

The current implementation uses continuous polling of the buttons driver and the fm driver to receive key press events and RDS change events. This means that the Atmel microcontroller does not go to sleep at all. It should be possible to redesign the drivers to make use of interrupts and plan sleeping cycles for the CPU so that power could be conserved. Furthermore, the Power Reduction Register could be utilized in this context to conserve even more power.

There is an easy way to set up a pin change interrupt to be triggered on a button pressed. The challenge would be detecting the exact button that was pressed since this would require turning on/off some of the control wires. This would generate further interrupts. Therefore a possible solution would be to use a finite state machine to decide what the current interrupt means. Some care needs to be taken in order to deal with the bounce issue described in Keypad Bounce.

The RDS polling could be also made using interrupts. The Si4703 has two GPIO pins that could be set up to go to a high state when RDS data is available or when the STEREO indicator changes. This could be again detected using a pin change interrupt.

There would be some issues, though. One of them would be that the screen won't refresh, so the RSSI value would be out of date. Another issue would be the procedure to turn off the backlight of the LCD that depends on a counter. Both of these issues could be solved by using a timer to wake up the device from time to time and execute the above procedures.

I did not have time to start implementing the suggestions above. If the product is to be made commercially, an interrupt based implementation could be favourable since it would allow for reduced power usage and better timing of the LCD being left in an on state (which currently depends on the speed at which the main loop executes which varies with the state of the radio and buttons).

Measurements

Power Consumption

The Power Reduction Register is used and pull up registers are used on all inputs in order to attempt to reduce power consumption. There were no measurable difference between having those power optimizations on or off but I kept them on for completeness. One explanation is that those probably make huge difference during sleep modes but since the project is not using any, this does not make a big difference.

The biggest power consumer is the LCD backlight since it is composed of four LEDs. There is therefore logic for turning it off after a few seconds when any keys have not been pressed. My measurements show that that the power consumption is drastically different with the backlight on and off:

	ON	**OFF**
Backlight	41 mA	25 mA

The unit runs on two AA batteries. The most basic (non-alkaline zinc-carbon) ones have capacity between 400 and 900 milliamp-hours. This would give between 11.2 and 25.2 hours of runtime[8] if

[8] According to http://ncalculators.com/electrical/battery-life-calculator.htm

the device runs with its LCD backlight off (assuming 25mA power consumption). My experiment showed that the unit was able to run for about *16 hours and a half* on the cheapest AA batteries I could find (9p each) which includes a few minutes with the LCD backlight on. This still beats my commercial PURE DAB & FM radio (in FM mode) which can run for about 12 hours. As outlined in Using Interrupts, I believe this could be improved even further.

Price

This is the breakdown of the price for the different components (if bought in bulk and just a single unit). It outlines the current setup. If the product is to be released commercially, it could be printed on a board and miniaturized even further which would reduce the numbers below. The prices do not include delivery charges. It does not include labour price nor does it include price for assembling (soldering, hot gluing, using sticky tape, bolting, drilling, laser cutting). Therefore those numbers are just for guideline:

Component	Quantity needed	Price for 1 unit (in £)	Price for 100+ units (in £)
Breadboard	1	2.41	1.93
Resistors and Capacitors	9+3	0.15	0.12
Hook-up Wire	1	1.53	1.22
Project Box	1	2.41	1.93
2AA Battery Holder	1	0.92	0.73
ATMEGA644P	1	5.32	3.02
Evaluation Board for Si4703	1	12.16	9.72
LCD 84x48 - Nokia 5110	1	6.07	4.86
12 Button Keypad	1	2.41	1.93
5pin Break Away Headers	1	0.11	0.09
	Total	35.14	26.87

Conclusion

It looks like the setup has the potential to be brought down to a price range suitable for the consumer market. More research needs to be done on how much the cost could be reduced by printing the whole circuit on a single board. It is a nice product that has a good reception (compared to tuners built in mobile phones) and could be made portable. It has longer battery life than some other competitor products.

Unfortunately there is a tendency to switch to digital DAB radio broadcasting which would render such a product obsolete. But building it is still a nice exercise.

Acknowledgements

I would like to thank Brian Jones[9] for providing me with the required hardware, support, helping out with the laser cutter and providing initial code to base our projects on. Thanks for staying overtime to assist us.

I would also like to thank Dr. Ian Wassell[10] for preparing the documentation for the course.

[9] http://www.cl.cam.ac.uk/research/dtg/www/people/bdj23/
[10] http://www.cl.cam.ac.uk/research/dtg/www/people/ijw24/