Radio Scanner:	Points:	_____ / 4
FMCLick:	Points:	_____ / 6
ADC-Volume:	Points:	_____ / 1
PS/2:	Points:	_____ / 3
Database:	Points:	_____ / 4
clean programming:	Points:	_____ / 2
Theory Tasks:	Points:	_____ / 5
Overall:	Points:	_____ / 25

Microcontroller VU

Application Protocol

Jan Nausner, MatrNr. 01614835
jan.nausner@gmail.com

Tutor:
January 22, 2019

Declaration of Academic Honesty

I hereby declare that this protocol (text and code) is my own original work written in my own words, that I have completed this work using only the sources cited in the text, and that neither this protocol nor parts of it have ever before been submitted to this or any other course.

_____ _____
(Date) (Signature of Student)

Admission to Publish

☐ I explicitly **allow** the publication of my solution (protocol and sourcecode) on the course webpage.

☐ I **do not allow** the publication of my solution (default if nothing is checked).

_____
(Signature of Student)

Contents

1 Overview

1.1 Connections, External Pullups/Pulldowns

Pin Assignment

What	
J12	Connected to VCC
J14	EXT
J15	PF0
LEDs	Off
GLCD backlight	On
PORT switches	Off
PORT jumpers	Pull down

The FMClick, Ethernet and PS2 modules need to be connected as per specification.

1.2 Design Decisions

On startup, the application tries to fetch the channel list from the desktop PC. If it is empty, it automatically performs a band seek and syncs the so obtained channels with the database. On band seek, the channel gets saved as soon as RDS information is available or after a specific timeout, in order to avoid hanging on invalid/weak radio stations. If no RDS information was available at this point, an according meessage is stored in the note. Further design decisions are explained in the respective module sections.

1.3 Specialities

The extended UDP functionality was implemented, which might be a source for some bonus points. Additionally, error handlers for soft and hard errors were implemented in the main application. Soft errors are displayed for a short time and the app then returns to an error free state, while hard errors stop the app after the message was displayed.

2 Main Application

Operation of the main application centers around a channel list datastructure, which contains the respective channelInfo struct and buffers for station name and a note. This list gets filled on reset either by syncing with the database or by performing a band seek operation. Afterwards, the app tunes to the first channel in the list, displays channel information and awaits user input.

The app can be controlled by the PS2 keyboard and with the volume potentiometer. To initiate a seek up/down operation, '.' or ',' have to be pressed. If 't' is pressed, a frequency can be entered (in hundreds of kHz) and after pressing enter, the app tunes to this frequency. By pressing 's', a band seek operation is performed, which purges the local list as well as the database and refills it with stations it found by seeking the band. If 'a' is pressed, the current channel is added to the list and database or updated with new RDS information if already in the list. Pressing 'f' adds the current channel as a favourite, if already added to the list. It is assigned the next free favourite index. A note can be entered by pressing 'n'. It will be stored when enter was pressed. The local list can be stepped through by pressing 'l', which tunes to the next highest channel in the list. After the execution of these commands, the app returns to the main screen, where information on the current radio station is shown, If the current channel is in the list, c[id] is shown in the first line and if it is in the favourites, f[quick dial] is displayed.

3 Communication

3.1 UDP

The provided UDP network stack is lacking some important features, which had to be added manually. In order to support ICMP echo request messages, PingP had to be extended to reflect such requests back to the source, combined with the ICMP echo reply header. To provide an appropriate response to communication requests on non-serviced ports, UdpTransceiverP had to be extenden to reflect such requests as an ICMP port unreachable message.

Information on the ICMP protocol was taken from: https://en.wikipedia.org/wiki/Internet_Control_Message_Protocol

3.2 PS2

In order to support PS2 communication, an ISR has to be installed on the PS2 clock line. Every time a falling edge is detected, the data line is sampled into a shift register. After the reception of all PS2 bits, the data byte is put into a ringbuffer and the decode task gets posted. This task reads scancodes from the buffer until it is empty again. The scancode bytes get decoded with the help of scancode tables for upper and lowercase. In order to achieve maximum performance, the table lookup relies on binary search. Upper-/lowercase tables are switched according to shift key states. After the succesful decoding of the scancode, the obtained character is signaled to the application.

4 FMCLick

This module handles the control of the FMClick radio module. Communication happens over the I2C bus. The FMClick module contains 16 configuration registers, which have to be read and written sequentially over the bus. In order to minimize communication, the module uses a local shadow register, which buffers the current register state of the FMClick module. In order to write to the radio module, the registers are modified in the shadow register, which is then copied to a communication buffer in the right order (starting with the first write address). This buffer is then written sequentially to the bus, up until the highest write address. To perform a read operation, all registers are read sequentially from the bus into the buffer and then written to the communication buffer in the right order. If a driver command is called, the driver gets blocked until the command is completed, as the commands require many different steps which must not be interrupted. Init, seek and tune are implemented according to the state machines provided in the application note. The seek interface was extendedwit a seekmode_t type, where it is possible to specify and additional BAND mode. This mode configures the radio in such way, that it does not wrap around band limits. Reception of RDS information can be en- and disbled with the respective command. If enabled, the RDS interrupt routine posts a decode RDS task, which reads the RDS information from the radio module. These registers are then decoded according to the standard (http://www.interactive-radio-system.com/docs/EN50067_RDS_Standard.pdf). The station name and picode are only signalled to the application, when all blocks have been receive, while the radio text is signalled everytime a block has been received, in order to limit waiting time until information can be displayed. The time is signalled right away, as it only requires one RDS message. To keep things simple, the half-hour timezone offset is not considered. Reception of RDS information is disabled by seek and tune commands in order to avoid wrong interrupts, but reenabled afterwards if it was enabled before.

5 Database

This module handles communication with the database server on the desktop PC. To keep things simple, there are no message queues, thus after the sending of one command, the module blocks until it has received and decoded a response message. Therefore the module only needs to buffer one sended and one received message, leading to a small memory footprint. The udpi_msg_t type in udp_config.h was extended with a len field, for more convenient message handling. If a module command is called, the appropriate UDP message string is composed according to the specified protocol from progmem and input. A send task is posted, which hands the message over to the UDP module. Afterwards the module waits for response, and if we have onie, the message is again buffered and a decode task is started. The message is decoded based on which command we expect a response to. Add and update responses are simple to decode, we just have to check if the command has been executed successfully and then signal the result to the application. A list response requires that all given IDs are stored. Afterwards a new fetchList task is posted, which issues a getChannel request for every ID in the list. After the last database item was received, 0xff is signalled to the application. Get responses

require the more work, as they provide the most information. The decoder iterates over the parameter list, decodes and checks every item accordingly and puts it into a channelInfo struct, which is then signalled to the application.

6 VolumeAdc

This module provides access to the ATMEGA1280 ADC in order to read input from the potentiometer. TinyOS provides the necessary driver modules, so it just required to configure the ADC with the desired settings and make configure the potentiometer pin as an input.

7 Problems

The main problem was the handling of the FMClick module. It requires a lot of effort to understand exactly how it works and how it needs to be configured in order to work in a desired manner. Also in the beginning it was very hard to debug the driver, as the module seemed to not work properly. This led to confusion whether the problems came from the software or the hardware. After finding out that the module is very sensitive and requires stable connections, this got a bit easier. In the rest of the application no major problems were encountered.

8 Work

Estimate the work you put into solving the Application.

Task	Time spent
reading manuals, datasheets	5 h
program design	2 h
programming	30 h
debugging	48 h
questions, protocol	2 h
Total	75 h

9 Theory Tasks

In the theory task we want you to develop argumentation skills that allow you to reason about the problem you have to solve and the solution you are designing. Clear presentation of ideas is crucial for communication with team members, bosses, customers, etc. This time we want you to prove your answers mathematically (by induction). Points are solely awarded for proper

mathematical argumentation.

Setting:

A set of n motes $\{p_1, \cdots, p_n\}$ is scattered in an area to broadcast data. For simplicity, assume that every mote p_i starts with a unique data sample i. That is, mote p_1 starts with 1, mote p_2 with 2, and so on.

Communication is divided into several subsequent rounds and happens wirelessly via broadcasts. Starting with round $r = 1$, in each round every mote i sends a broadcast message $m_i(r)$ to the system. Broadcasting happens in zero time and all motes receive messages just before the next round starts. A message $m_i(r)$ sent in round r is the set of known data samples S_i. For example the message $m_1(1)$ sent by mote 1 in round 1 is $S_1 = \{1\}$. After receiving messages in some round r mote i updates the set of known data samples. E.g., if mote p_i receives a message $m_j(r)$ from mote p_j $S_i = S_i \cup S_j$. This update is done for all received messages in each round.

This round-based communication can be modelled with a communication graph G_r where each node in the graph represents a mote. A directed edge (i, j) in G_r represents that the message sent by mote p_i in round r was sucessfully received by mote p_j in round r.

Tasks

1. **[3 Points]** Strongly connected communication: Assume that, because of message loss, not every message sent by a mote is received by every other mote. The only guarantee the motes have is that in every round the communication graph is strongly connected. Show that after n rounds of communication that data sample 1 is part of every set s_i, i.e., $1 \in \bigcap_{i=1}^{n} S_i$.

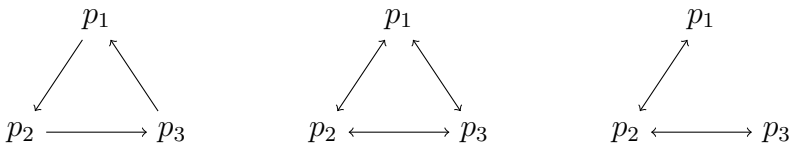

Figure 1: Example for $n = 3$: three strongly connected graphs

2. **[2 Points]** Rooted tree communication: Assume that, because of message loss, not every message send by a mote is received by every other mote. The only guarantee the motes have is that in every round the communication graph is a rooted tree, i.e., there exists at least one mote (the so called root mote) that has a directed path (a sequence of edges) to every other mote in G_r. Note that the root mote can be a different mote in every round. Show that after n^2 rounds of communication there exists a data sample i that is part of every set S_j, i.e., $\{i\} \in \bigcap_{j=1}^{n} S_j$.

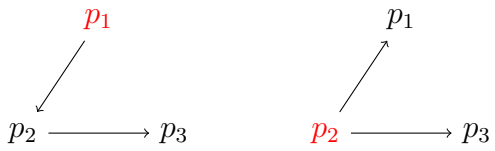

Figure 2: Example for $n = 3$: two rooted trees

9.1 Task 1 solution

Property to prove: After n rounds of communication, sample 1 is part of every set S_i, given the fact that the communication graph is strongly connected after every round.
Base case ($n = 1$): The property holds for $n = 1$, because the only set S_1 already contains sample 1.
Hypothesis: The property holds for n motes.
Induction step: Using the hypothesis, it can be assumed that after n rounds, n out of $n + 1$ motes contain the sample 1. Due to the strongly connected property it is known, that in round $n + 1$ the remaining mote receives at least one message from a mote where 1 is in the sample. Thus after $n + 1$ rounds all $n + 1$ motes have received sample 1. q.e.d.

9.2 Task 2 solution

Property to prove: After n^2 rounds of communication, there exists a sample i which is part of every set S_i, given the fact that the communication graph is a rooted tree after every round.
Base case ($n = 1$): The property holds for $n = 1$, because the only set S_1 already contains sample 1.
Hypothesis: The property holds for n motes.
Induction step: Using the hypothesis, it can be assumed that after n^2 rounds, there is a sample whih is part of n out of $n + 1$ motes. In the worst case, the remaining mote is always the root of a tree degenerated to a list, and thus it would take another n rounds until every mote receives the sample $n + 1$. In all other cases, mote $n + 1$ is the child of another mote containig sample i and thus receives it after one additional step. $n^2 + n \leqslant (n + 1)^2 = n^2 + 2n + 1$. q.e.d.