Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

3G Datalink for Wireless GPS Sensors

Semester Thesis

Schnetzler Christoph
cschnetz@ethz.ch

Computer Engineering and Networks Laboratory
Department of Information Technology and Electrical Engineering
ETH Zürich

Supervisors:
Dr. Jan Beutel
Tonio Gsell
Prof. Dr. Lothar Thiele

June 11, 2018

Acknowledgements

I thank Jan Beutel, Tonio Gsell and Akos Pasztor for their support and their helpful inputs. I thank Thomas Kleier helping me vapor phase soldering the u-blox SARA-U270 modules.

Abstract

This semester thesis aims to extend the existing wireless GPS rev2 sensor system with a software interface for the cellular module. The goal is to exchange data directly and reliably between the sensor system and the PermaSense data backend system. The result is a working, well-documented parser which handles the communication between the cellular module and the GPS sensor system. It is able to establish an internet connection and can send and receive data over it. It is though not yet integrated in the existing wireless GPS sensor code.

Contents

Chapter 1

Introduction

Avalanches, rockfalls and receding permafrost boundaries are just a few examples of what is going on in the mountains. This leads us to the question if these events are predictable at all.

The PermaSense project has been working for years to record data such as temperature, humidity, brightness, rock movements, etc., to create a better understanding of the processes in the high alpine locations and to react early to possible natural hazards.

In order to collect data, sensors have been installed at some chosen locations in the Alps which take measurements at regular intervals.

One of these sensors is the wireless GPS rev2 sensor. It collects ground motion data and stores it either locally or sends it via a cellular module to the PermaSense data backend system. Up to now, these sensors can only store their data locally as the software interface for the cellular module is not implemented.

Therefore, the goal of this work is to implement an interface which is able to establish a direct 3G datalink between sensor and PermaSense data backend system to exchange data.

Chapter 2

Objectives

- Extend the wireless GPS rev2 sensor system with a dedicated 3G datalink.
- Design an integration concept which guarantees
  - efficient operation and monitoring of the system.
  - reliable data transfer into the PermaSense data backend system.
- Implement and evaluate a prototype system based on the ARM/FreeRTOS software framework of the wireless GPS rev2 sensor system.
- Document the work in a manner that follow-up projects can be built upon.

Chapter 3

Overview

3.1 PermaSense

The PermaSense project aims to observe and to understand the phenomena in high-alpine environments better. This can be, for instance, rockfalls, receding permafrost boundaries or slope instabilities. For a reliable observation of the phenomena, a sensing network has been developed. This network consists of ultra-low power wireless sensor nodes which are sensing a diverse range of parameters, such as temperature, humidity, light, ground motion, etc. The nodes allow long term autonomous operations in range of multiple years. Therefore, the devices are strictly duty cycled and gather data in low rate (e.g. 1/120sec). Figure 3.1 shows the basic system architecture consisting of a Wireless Sensor Network, Core Stations and the Data Backend.

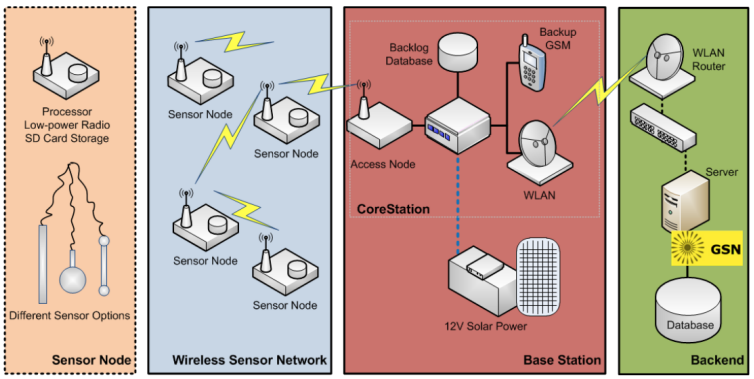


Figure 3.1: PermaSense System Architecture [19]

3.2 Wireless GPS rev2 sensor system

The wireless GPS rev2 sensor system (WGPS) has been developed to measure ground motion in high-alpine regions. Tracking ground motion helps to observe rockfall and debris flow critical regions as well as the movement of boulders at steep mountainsides. Having long time data over critical regions allows to detect suspicious behaviors, such as accelerating motion or sudden movements. For instance, with such information, mountain villages located in danger zones can be warned or evacuated at an early stage.

3.2.1 System overview

The WGPS device is driven by an ultra-low power STM32L496VG microcontroller [2], based on an FPU ARM Cortex-M4 MCU. Its equipped with GPS, GSM, Radio (TinyNode), an electronic compass, a tilt, a temperature and a humidity sensor (figure 3.2). The cellular module (called GSM) and the TinyNode are connected over a plug connection. Therefore, they can be plugged in if needed. It's powered over a solar panel and needs between 11 and 20 voltage input. The WGPS device is able to track ground motion with high precision in range of millimeters. This precision is achieved by using a reference GPS sensor mounted as closly as possible to the other sensors but at a position which is known to be static (e.g. solid rock). If the other ones move, we can calculate the relative movement which is much more precise than just the GPS data itself. The GPS data is recorded in raw format and no data processing is performed on the WGPS devices itself. The postprocessing is done in the PermaSense data backend system.

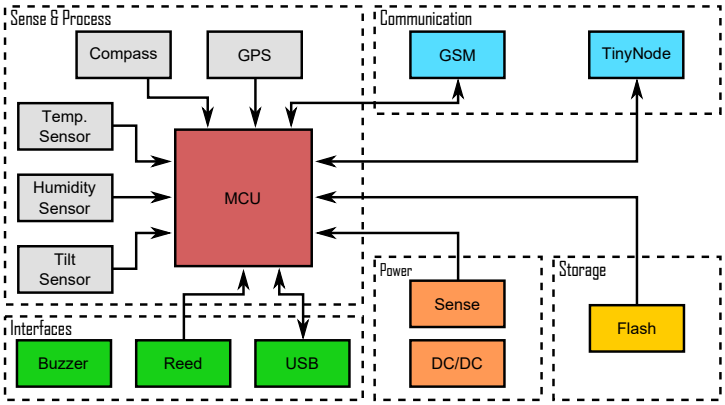


Figure 3.2: WGPS System overview [14]

3.2.2 Operating modes

Compared to the basic system architecture (figure 3.1), the WGPS devices do not need any core stations, they directly connect to the PermaSense data backend. This allows a simpler installation in the mountains. The WGPS has three different operating modes.

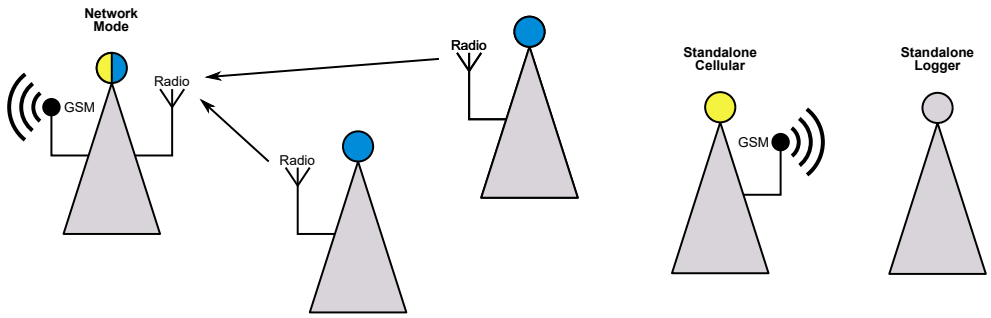


Figure 3.3: WGPS operating modes [14]

Standalone Logger Mode

In this operating mode, the WGPS device has no GSM and Radio module attached. Therefore, the device is not able to communicate with the PermaSense backend system. It stores the measurements on its internal storage. The data has to be extracted during on-site maintenance. This configuration is for areas which do not have cellular coverage or which are not intended to be observed online.

Standalone Cellular Mode

The WGPS device is equipped with a cellular module but has no radio module attached. With the cellular module, the device is capable of communicating with the backend system. It can send the recorded measurements and receive commands from the backend system.

Wireless Network Mode

This is the full configuration which uses radio and GSM. With the radio the device is able to communicate between multiple WGPS devices. One device acts as sink and collects all the date. This device is equipped with GSM and can therefore forward all data to the backend system. The sink can also receive commands from the backend which are forwarded over radio to the other devices.

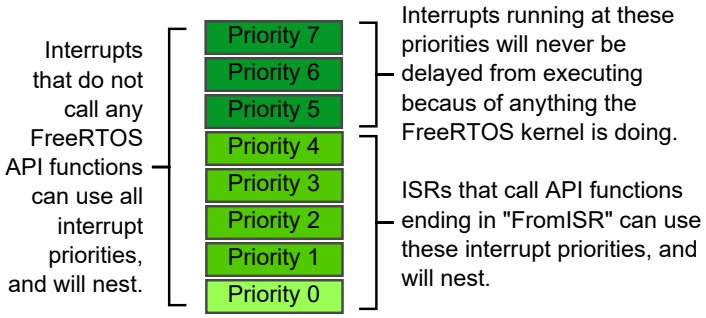


Figure 3.4: Example interrupt priority configuration [16]

3.2.3 FreeRTOS

The WGPS device needs to take measurements in periodic time steps. But not all measurements have to be done in the same periodic interval. Sometimes, a measurement takes some time and we would like to do other work until it has finished and we can continue. We also want to be able to shut off the peripherals independently to save energy. We need something which organizes this large amount of work for us. Therefore, the WGPS system uses FreeRTOS.

FreeRTOS is a real time operating system which is small enough to run on microcontrollers. It provides real time scheduling functionality, inter-task communication, timing and synchronization primitives. It allows us to write code in a set of independent tasks. Tasks reduce the dependencies between software modules and increase the code maintainability. Each task and interrupt is assigned to a priority which allows the scheduler to decide which task or interrupt needs to be executed first. As interrupts need to response very fast to events, they have always the higher priority than tasks. Means, the lowest interrupt priority is still higher than the highest task priority. Figure 3.4 shows a possible interrupt priority configuration where the maximal system call interrupt priority is set to four and the kernel interrupt priority is set to zero. FreeRTOS has special API functions for interrupts, ending with ”FromISR”. These functions are located in the interrupt code, but not executed inside an ISR because FreeRTOS claims to be deterministic and therefore doesn’t allow to call functions which could cause non-deterministic behavior.

The FreeRTOS primitives used in this project are introduced in section 5.3.

Chapter 4

Cellular module

The used cellular module is the SARA-U270 from u-blox [4].

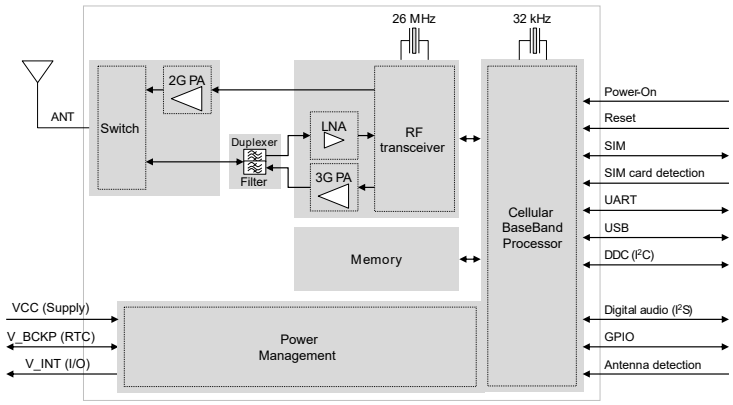


Figure 4.1: SARA-U270 block diagram [7]

4.1 Interface

The used serial communication interface is UART. The module features the complete functionality conforming to the ITU-T V.24 recommendation [5]. In the project, only the data lines (Rx, Tx) and the hardware control lines (CTS, RTS) are used. The modem status and control lines (DTR, DSR, DCD, RI) are provided but unused. By default, the HW flow control is enabled, the DSR line is set to ON in data mode and OFF in command mode and the module enters the online command mode after an ON-to-OFF transition of the DTR line, issuing an "OK" result code. Result codes are discussed in chapter 4.2.

The used frame format is 8N1 (8 data bits, no parity, 1 stop bit). 8N1 is already the default frame configuration and doesn't need to be changed.

4.2 AT commands

This section gives a short overview about AT commands. All information about u-blox AT commands can be found in the document "u-blox Cellular Modules - AT Commands Manual" [6].

4.2.1 Operating modes

The SARA-U270 module has three different operating modes:

- Command mode
- Data mode
- Online command mode

Command mode

In the command mode, the module interprets every received character as a command to execute. Any communication in the command mode is terminated with the command line termination character <s3_ character>. By default, this is the carriage return '\r' character. Every command has the following generic syntax:

```
"AT"<command_name><string><s3_character>
```

For instance,

```
"AT+IPR=9600\r"
```

but <string>can also be empty,

```
"AT+CGMM\r"
```

The AT command response comprises of an echo, an optional information text response string (ITR) and a final result code (FRC).

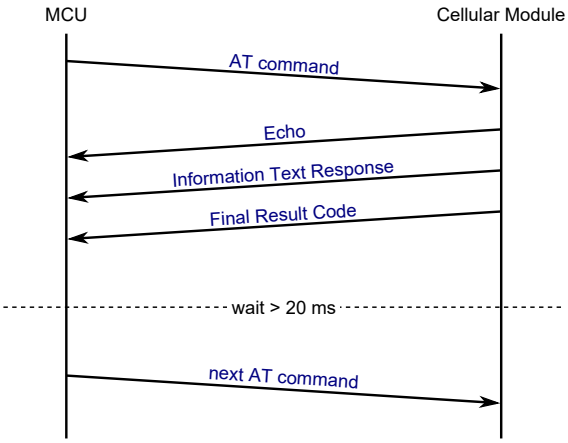


Figure 4.2: AT command response flow chart [17]

The echo is an exact copy of the command which has been sent. The ITR contains intermediate outputs as well as descriptive outputs. This can be the manufacturer's name, the firmware version, stored settings, etc. An ITR disposes of the following syntax:

```
<text><s3_character><s4_character>
```

The <s4_character>is by default the linefeed character '\n'.

The FRC completes the response and informs about the result of the command. It has the following syntax:

```
<numerical_code><s3_character>
```

The most common FRC's are "OK" and "ERROR". All allowed result codes are listed in table 4.1. A possible response is, for instance:

```
AT+URAT?\r +URAT: 1,2\r\n 0\r
```

The module supports verbose or numeric responses. By default, the verbose response mode is enabled, but parsing text instead of numbers is inefficient. Therefore, we use the numeric response mode. After reception of the FRC one should wait for at least 20ms before issuing a new AT command. This allows the module to transmit buffered URC's, otherwise collisions between URC's and AT commands are possible. URC's are introduced in section 4.3.

Verbose	Numeric	Description
OK	0	Command executed correctly
CONNECT	1	Data connection established
RING	2	Incoming call signal from the network
NO CARRIER	3	Connection terminated from the remote part or attempt to establish a connection failed
ERROR	4	General failure
NO DIALTONE	5	No dialtone detected
BUSY	6	Called number is busy
NO ANSWER	7	No hang up detected after a fixed network timeout
Command aborted	18	Command execution aborted

Table 4.1: Allowed result codes

Data mode

The Data mode is entered as soon as a connection to a remote host is established or the result code "CONNECT" has been received, respectively. In this mode, the module doesn't interpret characters to be commands. Instead, every character received from the serial interface is being forwarded to the remote host. Every character received from the remote host is being forwarded to the serial interface. To exit the data mode, one needs to send +++. This triggers the module to send "OK". Only after this message, the data mode is left successfully.

Online command mode

The online command mode is if the module is in data mode but can read and process AT commands. This mode can be entered over a ON-to-OFF transition on the DTR line. An "OK" result code indicates that the online command mode has been entered.

4.2.2 Errors

Every AT command has an error response type. The most common error type is the "+CME ERROR". This error occurs if the issued command couldn't be processed successfully by the cellular module. If an error occurs, the module sends the echo followed by the FRC. By default, the FRC "ERROR" is displayed without information about what caused the failure exactly. The numeric response for an error is 4. To enable the entire error message, one needs to send the AT command "+CMEE" after the module has booted.

Next there is an example of an error response with +CMEE disabled,

```
AT+CPIN?\r ERROR\r
```

and with +CMEE enabled,

```
AT+CPIN?\r +CME ERROR: SIM NOT INSERTED\r
```

For a better understanding the example shows the verbose and not the numeric response. In the numeric response "ERROR" would be replaced with 4 and "SIM NOT INSERTED" with 10. All possible errors can be found in the "AT Commands Manual, Appendix A" [6].

4.2.3 Response time

The cellular module usually replies immediately to an AT command. This means that in average it takes 10ms and in any case less than one second to respond. However, there are commands which will need more time to answer. For instance, the network registration can take up to three minutes until a response is issued.

4.3 URC - Unsolicited Result Code

Beside the normal response message, the module can as well send unsolicited result codes - URC's (figure 4.3). URC's are string messages which are completely uncorrelated to an AT command execution. The module uses URC's to inform if a status changes or a specific event has occurred. This happens when, for instance, the module has lost its network coverage and needs some time to register again to the network. URC's can occur at any time except if the module is in data mode or if the module is already executing an AT command. URC's need to be enabled with AT commands. Most of the URC's have the same name as the command that enables it. For instance,

```
AT+CREG=1
```

enables the network registration URC with the following appearance,

```
+CREG: 1
```

The One is just one possible value of CREG and indicates that the module is now registered to the home network.

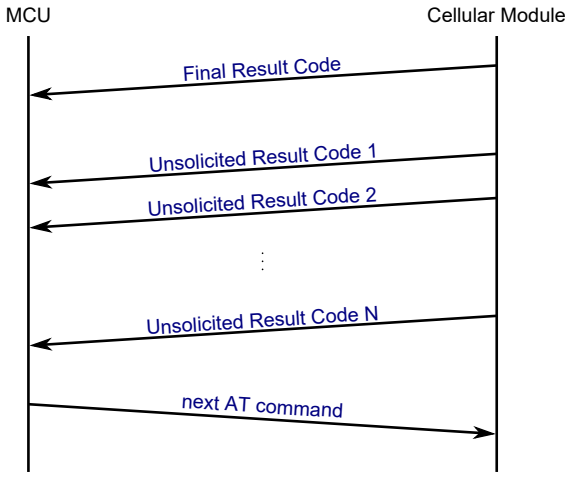


Figure 4.3: URC flow chart [17]

4.4 Boot phase

The boot phase of the module can take several seconds. In order to signal the successful end of the boot phase, the module sends a greeting text. The module will not response to any AT commands before the greeting text has been sent. By default, the empty string is sent. The greeting text can be set to any string with a maximal length of 49 characters.
If the greeting text is used, one needs to consider that the module has an autobauding feature which is enabled by default and allows the module to detect the baudrate automatically as soon as the first characters have been received. Therefore, as long as the module hasn't received anything, it won't send anything. This means that if the greeting text is used to detect the end of the booting phase, one needs to set a fix baudrate. Otherwise both sides wait for an event that will never happen.
In this project, the greeting text has been changed to "ready\r".

Chapter 5

Code implementation

5.1 Overview

If the wireless GPS sensor system is not exactly operating in datalogger mode, it needs the cellular module to transfer the data to the PermaSense data backend system. Up to now, the entire hardware is provided and also the software is complete except for the software interface between the cellular module and the WGPS device. This means that actually the WGPS device if fully working but only in data logger mode. In this project we aim to close this gap and to implement the interface. As mentioned in the objectives, the interface should allow efficient operation and give the opportunity to monitor the system.
This chapter is split up in three major parts. In the first one, the development environment and the FreeRTOS primitives used in the implementation are introduced. In the second part, the interface will be explained more explicitly. And finally, in the third part, we consider how to establish a reliable connection between cellular module and the PermaSense backend data system.

5.2 Development environment

The used development and debugging IDE is the Atollic® TrueSTUDIO® [1] for STM32 microcontrollers, based on Eclipse, CDT, GCC and GDB. It supports version control such as Git and the SEGGER J-Link [3] which has been used as debugger. The entire project is version controlled using Git and the repository is stored on GitLab.

5.3 FreeRTOS primitives

As the rest of the already existing software relies on FreeRTOS, we need to implement our interface within this framework. Our implementation uses FreeRTOS

tasks, software timers and event groups which are now introduced briefly. More details can be found in the FreeRTOS reference manual [12].

5.3.1 Tasks

Tasks are the basic elements, handled by the scheduler. Each task can be in one of four states (running, ready, blocked or suspend) as shown in figure 5.1. If a task is in blocking state, then he is not ready to run and waits for an event to happen. If the event happens, the task is put in ready state by the scheduler. If it is the only task in ready state and no task is in running state, then the task is immediately put in running state. Only in the running state, CPU time is consumed. If more than one task is in ready state, the priority decides which task is put in running state first. If a task is suspended, it cannot resume by itself, instead, another task needs to do that. In our implementation we never use suspend as we always wait for some events to happen and therefore wait in blocked state.

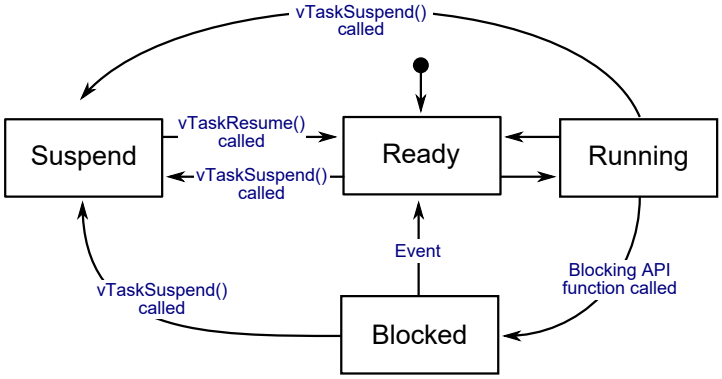


Figure 5.1: Valid task state transitions [15]

5.3.2 Software timer

FreeRTOS software timers are used to trigger an event in a certain time in the future. If one creates a timer, then one also needs to create a callback function. The callback function is called after the timer has fired and the code within this function is executed. Timers can be used for various occasions. For instance, if one starts to execute a measurement, but one does not know if the execution is deterministic. A timer can be started before starting the execution and specify the maximum allowed time the task has. If the task exceeds the specified time, the timer fires and can abort the execution.

5.3.3 Event groups

Event groups belong to the synchronization primitives and are used to communicate events to tasks. They allow a task to stay in the blocked state until a combination of events has occurred. For instance, if we send an AT command, we need to wait for a response. This can be either the event "OK" or the event "ERROR". Event groups allow us now to block the task until one of them occurs. In order to indicate if a certain event has occurred, event 'flags' are used. A set of events 'flags' is called event 'group'. Setting an event flag or a certain combination of them, unblocks the task.

5.4 Cellular module software interface

In the second step, we will now step through our implementation. First, we explain two important control structures which are used in the entire code. Secondly, we show how we initialized the serial interface between cellular module and MCU. Thirdly, we consider the command parser and explain how it is thought to be used. Finally, we introduce our task.

5.4.1 The AtCommand struct and the GSM struct

AtCommand struct

AT commands have different error types, response types, response times, sizes and names. This means that we need to handle the AT commands differently accordingly to the given parameters. For instance, some take longer to respond than others, some have only a FRC as response, some have an ITR followed by a FRC and there are also special cases where only an ITR is sent. In order to keep track of all of these different characteristics, we created the AtCommand struct. This struct stores the error type, the response type, the response time, the name and its size. Before sending an AT command, this struct has to be filled such that the parser (see section 5.4.3) is aware of what to expect and to react accordingly.

GSM struct

This struct is the heart of our implementation. It contains the PARSER_REG variable. This variable is used as a register as every bit indicates a different parser state. Table 5.1 lists all used flags. This register keeps track of the actual system state. At the very beginning, no bit is set. If we start to listen on the serial interface, we activate the interrupt and then the first bit *GSM_IT_ENABLED* is set. Next, if we power on the cellular module, the second bit *GSM_RUNNING*

Bit	Name	Description
0	GSM_IT_ENABLED	Set if parser is started, interrupts enabled
1	GSM_RUNNING	Set if module is activated
2	GSM_GREETING	Set if greeting text has been received
3	GSM_COMMAND	Set if response of a command is expected
4	GSM_URC	Set if a URC is being received
5	GSM_ECHO_OK	Set if command echo from GSM Module is ok
6	GSM_FIRST_AFTER_ECHO	Cleared if first character after echo is received
7	GSM_ERROR	Set if error in the parser pipeline occurred

Table 5.1: PARSER Register

is set to indicate that from now on characters can be received. As we use the greeting text to indicate a successful startup, we first need to wait for it to be received. The third bit *GSM_GREETING* is set if we have received it. If the first three bits are set, we are ready to start sending AT commands.

During transmission and reception of an AT command, several errors can occur. For instance, there can be an UART error, an error in the parser or an error response from the cellular module. Therefore, we store the error type and its corresponding error code in the GSM struct. If an error is present, the *GSM_ERROR* bit in the PARSER_REG is set, also. This bit is set as long as the error is not handled, and an error handling is pending.

The AT command responses and URC's need to be stored somewhere. This is also done in the GSM struct by using two pointers. One points to the beginning of the ITR string and the other one to the beginning of the URC string. They both have stored their string size, too. The URC and the ITR pointer are used in the *URC Handler* and in the *Command Handler*.

5.4.2 UART and pin initialization

All information about the STM32L496VG relies upon the information provided by the STM32L496 datasheet [8], the STM32L496 Reference Manual [9] and the STM32 HAL driver [10] .

According to the wireless GPS rev2 sensor schematics, see Appendix, the cellular module is connected over USART2. Therefore, Tx and Rx are on pin PA2 and PA3 respectively. To initialize these pins, we use the UART_HandleTypeDef struct from the STM32 HAL driver. Because USART2 belongs to the alternate functions of a pin, we need to change its mode to AF7. In the next step, we need to initialize the UART itself. As mentioned in chapter 4.1, the cellular module uses the 8N1 frame format and therefore we set it accordingly to this. Table 5.2 shows the entire setup. The baudrate has been set to 9600 bps but 115200 bps

UartHandle->Instance	= USART2;
UartHandle->Init.BaudRate	= 9600;
UartHandle->Init.WordLength	= UART_WORDLENGTH_8B;
UartHandle->Init.StopBits	= UART_STOPBITS_1;
UartHandle->Init.Parity	= UART_PARITY_NONE;
UartHandle->Init.HwFlowCtl	= UART_HWCONTROL_RTS_CTS;
UartHandle->Init.Mode	= UART_MODE_TX_RX;
UartHandle->Init.OverSampling	= UART_OVERSAMPLING_16;

Table 5.2: UART initialization

works as well.

Further, we need to select the clock source and enable the peripheral clock for the USART2. Four different clocks can be used for USART2, PCLK, HSI16, LSE or the system clock SYSCLK. The clock source is set in the "Peripherals independent clock configuration register (RCC_CCIPR)" and enabled with the "APB1 peripheral clock enable register (RCC_APB1ENR1)". In fact, one does not have to deal with these registers directly. The clock can be enabled with the HAL macro, *__HAL_RCC_USART2_CLK_ENABLE()* and the clock source can be selected with the RCC_PeriphCLKInitTypeDef struct. In the last step we set the interrupt priority of the USART2. This priority has to be chosen very carefully to ensure that the entire system still works. The interrupt priority is set with *HAL_NVIC_SetPriority()* and enabled with *HAL_NVIC_EnableIRQ()*.

5.4.3 AT command parser

The AT command parser (from now on called parser) is actually only an interrupt handler. Every time the USART2 interrupt is triggered, the parser is called. Figure 5.2 shows the entire structure. It can be split up in four parts as colored in the figure. Every part will be explained briefly starting with the white section.

Entering interrupt handler

If the interrupt is triggered, we check if an UART error has occurred. If an error is detected, we check if the cellular module (in figure 5.2 referred as GSM) is already running. The *GSM_RUNNING* flag indicates if the module is running. The flag is set as soon as the SW_GSM pin (PE10) is set HIGH. This enables the voltage supply for the cellular module. If the module is not running, this error can be ignored. If it is running, we call the UART error callback function. The way how this error is handled needs to be adapted to the already existing code of the wireless GPS sensor system. For now, the callback function just restarts the module. Without an UART error, we check if the cellular module is running and if there are unhandled errors from the parser or from the cellular module itself.

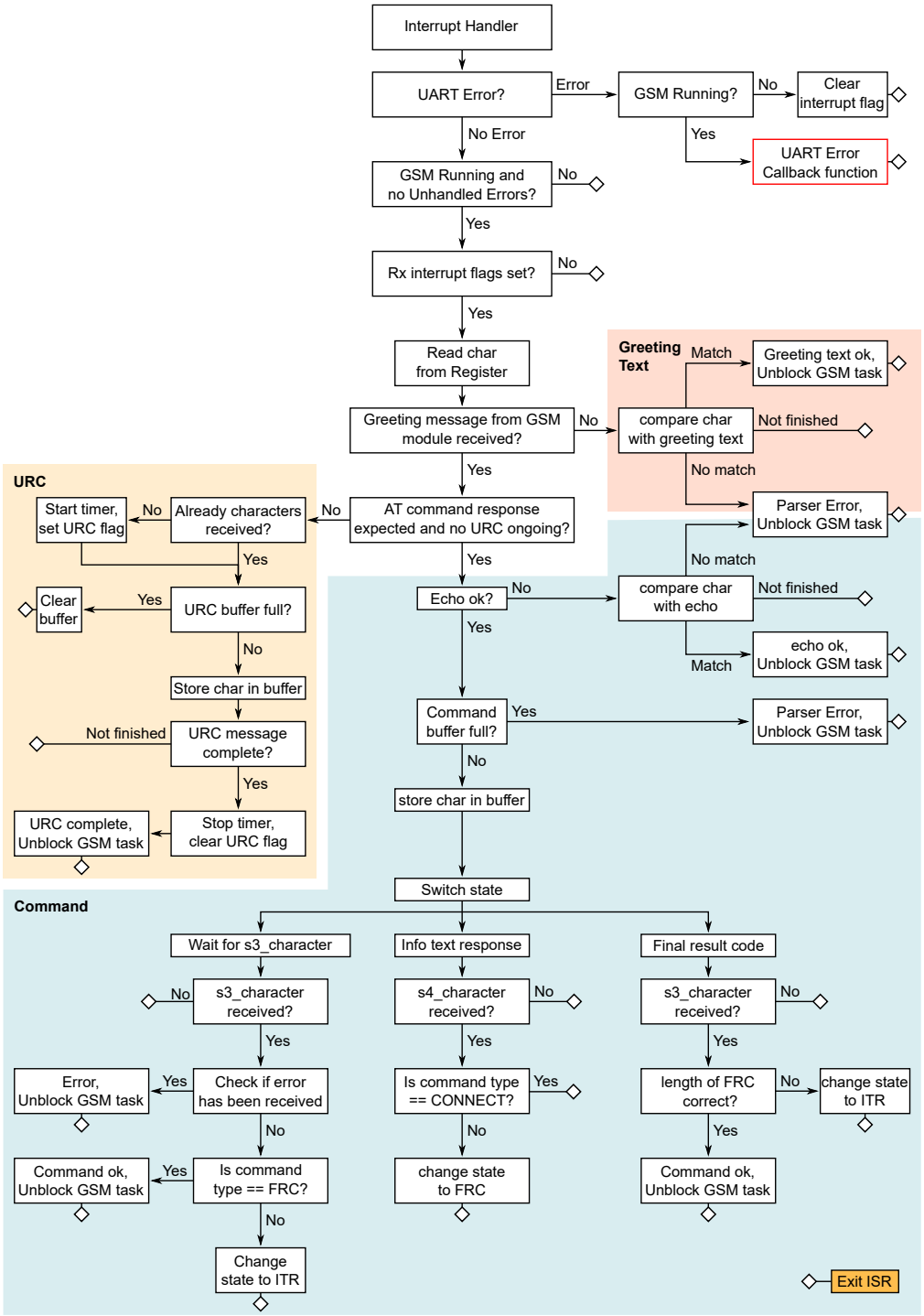


Figure 5.2: AT command parser

Unhandled errors are indicated by the *GSM_ERROR* flag. We only proceed if the module is running and no errors are pending. In the next step we check the Rx interrupt flag to be sure that a new character has been received. If the flag is set, we read the character from the RDR register.

Now we have to decide where we need to compare this char. If the greeting text of the cellular module hasn't been received, then we enter the greeting text section. If we have sent an AT command, then we expect the cellular module to response to it and the command section is entered. If no command response is expected, we have an unsolicited result code (URC) from the module and the URC section is entered.

Greeting text

We use the greeting text to signal a successful boot (see chapter 4.4). It is sent as soon as the module is ready to receive AT commands. Why do we not handle it in the URC section as the greeting text is an URC? The reason why, is that the greeting text is the very first string we receive and therefore we check it at the very beginning. This avoids sending AT commands before the greeting text has been received.

In this part, we compare the received character with the corresponding character from the greeting text. That means, if we receive, for instance, the 5th character, then we compare it with the 5th character of our greeting text string. If it is correct and we are not at the end of our string, we leave and wait for the next character to be received. If it doesn't match, we set the *GSM_ERROR* event flag to unblock our GSM task. We also set the *GSM_ERROR* flag in our PARSER register to signal an unhandled error. If the last character of the greeting text has been received successfully, we set the *GSM_GREETING* event flag to unblock the GSM task as well as the *GSM_GREETING* flag in our PARSER register. This flag is set as long as the module is not powered off or reset. If one of these events take place, this flag needs to be cleared.

URC

URC's have been explained in chapter 4.3. If no command response is expected, we enter this part of the implementation. First, we check if this is the first character we have received for the coming message. If this is true, we start a Free RTOS software timer and set the *GSM_URC* flag in our PARSER register to indicate that an URC is ongoing. The timer is set to ensure that the URC will finish after some time and not hang up the parser. If the flag is already set, we don't need to do it again and can continue. Before we store the character in our buffer, we check if it's full or not. If so, we just clear the buffer and continue. Why are we doing this? URC's are short messages with a length of about 7 to

15 characters and therefore predictable and an overflow should never happen. If it happens, then an overflow is avoided. Every URC message ends with the <s3_character>and until we haven't received this we leave the ISR. After we received it we stop the timer, clear the *GSM_URC* flag and unblock the GSM task by setting the *GSM_URC* event flag.

Important note: If the cellular module is in data mode, all sent characters from the remote host are directly forwarded to the serial interface. This means, these characters are considered to belong to an URC and need to be handled in the URC handler of the GSM task. The maximal length of data which can be received, depends on the buffer size. If this limit is exceeded, then the buffer is silently cleared, as mentioned above. Every message sent from a remote host to the cellular module, needs to end with the <s3_character>, otherwise the parser won't be able to recognize a finished message.

Command

Every issued AT command has by default an echo which is sent first, followed by the optional ITR and ends with the FRC. Therefore, we compare the received character with our corresponding command character. It's the same procedure as with the greeting text. If one character doesn't match, we raise an error and unblock the GSM task. If it's not finished, we wait for the next character. If it finishes successfully, we set the *GSM_ECHO* flag and the *GSM_FIRST_AFTER_ECHO* flag in the PARSER register. The GSM_FIRST_AFTER_ECHO flag is used to set the switch statement to its initial state. After the echo we check if the buffer is full or not. If it's full, we set the error and unblock the GSM task with the *GSM_ERROR* flag. To avoid such an overflow, one needs to adjust the buffer size to the largest possible output of the used AT commands. But for the sake of completeness, the overflow is handled. In the normal case, when it is not full we store the character in the buffer.

The next state is more complicated as the possible answers of the cellular module are quite different. Normally, the response is composed of an ITR followed by a FRC or the response contains only a FRC. But if the cellular module responses an error then the ITR and FRC are dropped and only the error is sent. There is also a special case where only an ITR is sent without any FRC's. This happens when the module changes from command mode to data mode by issuing a "CONNECT". This would not be a problem but "CONNECT" is send in verbose format and therefore with full header and footer:

```
<s3_character><s4_character> CONNECT <s3_character><s4_character>.
```

Finally, there are also responses which have more than one ITR line. Means, the <s3_character>is sent multiple times inside the ITR instead of only one time at the end.

All these cases need to be handled correctly and in order to this a switch statement is used. The first state which is entered is always the "wait for s3_character". As long as this character hasn't been received, we leave the ISR. After reception, we know that one part is fully received. This can either be a FRC, an error, an entire ITR or one part of a multiline ITR. If the received response matches one of the possible errors, then we raise an error and unblock the GSM task by setting the *GSM_ERROR* flag. If the expected response simply consists of an FRC, we read out the value and unblock the GSM task by setting the *GSM_COMMAND* event flag. To signal that the AT command is finished and URC's are accepted again we clear the *GSM_COMMAND* flag in the PARSER register. The FRC value is stored in the GSM struct in the "final_result" parameter. If we expect an ITR, we change the state to ITR. As mentioned in chapter 4.2 the ITR ends with <s3_character><s4_character>, therefore we need to wait for the s4_character to be received. This should be the consecutive character of the s3_character. If the expected response is the special case "CONNECT", as mentioned above, we simply stay in the ITR state and wait for the next s4_character to be received. If it is not a special case, then we change the state to FRC. Here we wait again until the s3_character is received as the FRC ends with this character. After reception, we check how long the response is. The maximal length of an FRC is three. If it's longer than three, we have an ITR with multiple lines and therefore we change the state back to ITR. If the length is within the accepted boundaries, we read out the value, unblock the GSM task by setting the *GSM_COMMAND* event flag and clear the *GSM_COMMAND* PARSER register flag.

5.4.4 GSM task

In this chapter, we consider the used FreeRTOS task called GSM task. The goal of this task is to establish an TCP/IP connection to the PermaSense data backend system, sending all pending data, listen if someone is sending anything he has to react to, tear down the connection and shut down the cellular module.

This task is thought to be used in the following manner: After a certain time period, another task calls *GSM_powerOn()*, leading the GSM task to enter the ready state. The GSM task then performs all the steps mentioned in the goal of this task. After he has done everything, he enters the blocking state and waits until he is set in ready state again.

Figure 5.3 shows the GSM task flow graph. If the cellular module (called GSM) is not running, the task enters the block state and waits to be unblocked by the event flag *GSM_RUNNING*. If it's running, then we check if the greeting text has already been received or not. If not, we block and wait for the *GSM_GREETING* event flag to be received. If the flag hasn't been received within a certain time, the task is unblocked due to a timeout and we enter the error handler which

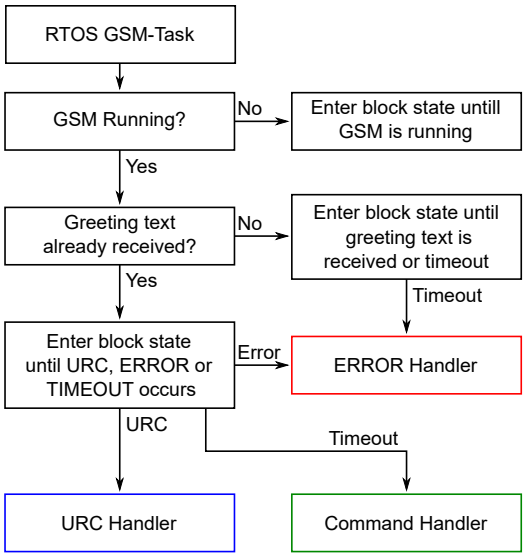


Figure 5.3: GSM task

then needs to take a decision what to do. For now, we restart the module. If the task is unblocked by the *GSM_GREETING* event flag, we continue. According to the AT command manual, one should wait at least 20ms before issuing a new AT command. Therefore, we enter the blocking state for at least 20ms and wait if an URC or an error is responded by the parser. If the *GSM_ERROR* event flag unblocks the task we enter the error handler. If we are unblocked by a *GSM_URC* event flag, we enter the URC handler instead. If a timeout occurs, then nothing is pending, and we can continue sending AT commands.

In the following the three used handlers will be introduced:

Error handler

All errors are stored in the GSM struct. The variable *Error_type* holds the information about the kind of error (UART error, Parser error or GSM error) which has occurred and the variable *Error_code* contains more precise information about it. All errors need to be handled and this is thought to be done in the error handler. For instance, one can log the error and then continue or reset the parser and restart or one can just ignore the error. What is done is strongly depending on the already existing error handling in the WGPS device code.

URC handler

If an URC has been received, this handler is entered. The URC message is stored in the aRxURCBuffer. The GSM struct contains a pointer on this buffer and stores its length. A possible URC is for instance

```
+CREG: 1
```

First, one needs to decide which URC's has been received. The URC handler checks the length of the URC to already exclude URC's with the wrong length. Then, we compare all possible URC's with the received one and if we find the right one, we read out the value and decide what action needs to be done.

The URC handler is also entered if the cellular module is in data mode and the server sends data to the module. This can be a command or similar. Note that everything sent to the module needs to end with the <s3_character>, otherwise the parser will fail to detect the end of the message.

Command handler

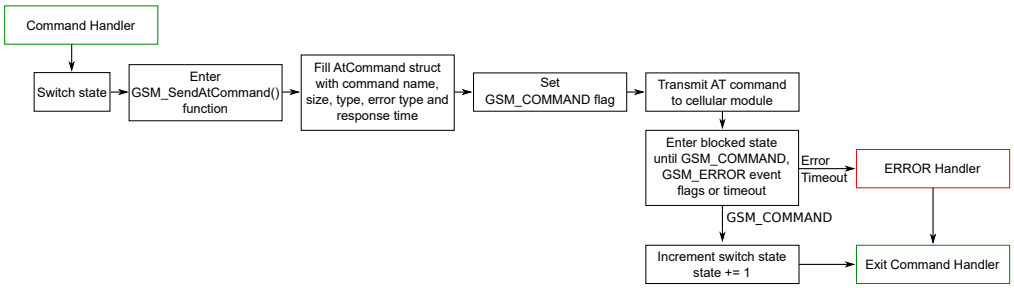


Figure 5.4: Command handler flow graph

In the *GSM_CommandHandler()* all AT commands are send and handled. To be able to leave the function after a finished command and to guarantee that URC's are handled in time we use a struct statement. This allows us to jump to the command we want to send. Up to now the switch is split into six sections (Initialization, Cellular network registration, Internal PDP context activation, Data connection, Data mode, Close data connection and End). The initialization is used to enable the numeric error response such that more precise error responses are send by the module. The cellular network registration, the internal PDP context activation and the data connection are explained in a separate section, see chapter 5.5. In the data mode we send our data to the remote host we have established a connection with. If we have send and received everything we need to enter the close data connection part, where we exit the direct link mode and

close the used socket. Entering the "End" state turns off the cellular module and the UART interface.

To send an AT command we use the *GSM_SendAtCommand()* function which takes the command name, it's size, type, error type and maximal response time as input. These parameters can differ between AT commands. In the function these values are stored in the AtCommand struct and we set the *GSM_COMMAND* flag to indicate that a response is expected. Next, we use the HAL driver function *HAL_UART_Transmit()* to send the command to the cellular module. If this was successful we enter the block state and wait to be unblocked. This can be through the *GSM_ERROR* event flag, the *GSM_COMMAND* event flag or by a timeout. If we get a timeout or an error, we enter the error handler, else we increment the switch state and leave the function. The switch state can be set or read with the *GSM_CommandHandlerState()* function.

5.5 Datalink

In this part, we derive a solution how we can establish a reliable datalink between cellular module and a remote host. To test the datalink, YAT has been used as dummy remote host. YAT is a terminal which can listen and send data to a defined port and therefore acts as a simple server.

There are two possible ways how the cellular module could be used on the wireless GPS rev2 sensor system. Either it's always on or its duty cycled. Obviously, duty cycling will be much more energy efficient. The problem is the reachability. The server won't be able to send commands to the wireless GPS sensor system if the cellular module is in sleep mode or turned off. How does the server know when to send commands if needed? One possible solution is that after the cellular module established a connection it sends all data and then sends a special preamble to indicate that it now waits a certain time to receive commands from the server. The server can now send any commands it wants to. After a certain command or time, the cellular module disconnects and turns off. This way it's like a receiver indicated communication between server and cellular module.

What can be done if its' kept always on? The nice way would be if the server could establish a TCP/IP connection with the cellular module. The problem is that normally the network operator allows connectivity only in one direction. Means, only the module can establish a TCP/IP connection not the other way around. If there are special contracts or the possibility at all providing both direction hasn't been investigated due to lack of time. A possible solution is that the module uses its keep alive mode which keeps the TCP connection open such that both can send any time they want. Disadvantage is that the server has occupied sockets and the module wastes lot of energy to keep it open. Another

way could be to call the module or send a SMS to it to inform that it should establish a connection to the server.

5.5.1 GPRS system architecture overview

Before we talk about establishing a TCP/IP connection we first need to understand how the mobile network works and what is different to the normal IP based internet. Figure 5.5 gives an overview of the mobile network which is interposed between the cellular module and the IP based internet network.

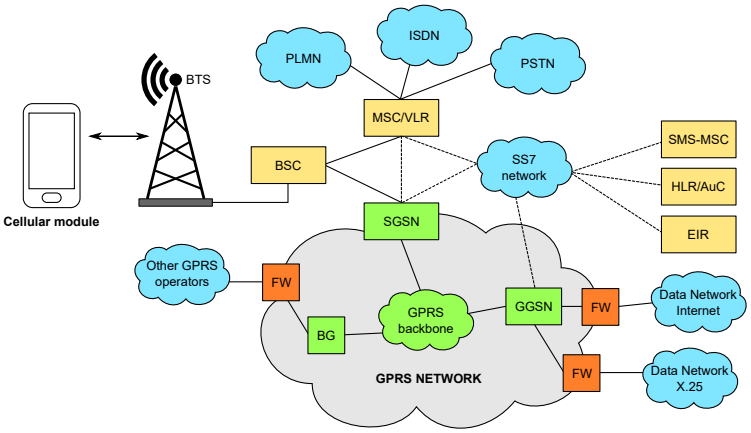


Figure 5.5: GSM and GPRS system architecture [18]

The cellular module needs to connect to a Base Transceiver Station (BTS). The BTS forwards the received signals to the Base Station Controller (BSC). Depending if the received signal is circuit switched traffic or packet switched data the BSC forwards it to the Mobile Switching Center (MSC) respectively the Serving GPRS Support Node (SGSN). The BSC is responsible for setting up and disconnecting Circuit Switched (CS) and Packet Switched (PS) connections to MSC respectively SGSN. The SGSN handles the packet data protocol (PDP) contexts. The GPRS backbone is now connected to other GPRS operators or via a Gateway GPRS Support Node (GGSN) to an external data network. For instance, the IP based internet data network. The GGSN assigns IP addresses to mobile devices as the cellular module. This allows now to communicate over the internet and to establish a TCP/IP connection.

5.5.2 Cellular network registration

This section provides an overview about the GSM/UMTS network registration on the cellular module. First, we need to decide which Radio Access technology (RAT) we want to use. This can either be GSM, UMTS or GSM/UMTS in

dual mode. By default, it's in dual mode and we let it like this as it switches automatically between the best available. Next, we can choose which operating bands we want to use (800, 900, 1900, 2100 MHz etc.). The cellular module can store the network settings in its non-volatile memory (NVM) and therefore only needs to be changed once. The module is able to perform the network registration automatically based on the stored settings. Automatic registration is enabled if the command "AT+COPS=0" is send. The automatic registration is stored in the NVM. If the module is registered in GSM either GPRS or EDGE is available depending on the signal quality. If in UMTS then WCDMA, HSDPA, HSUPA or HSDPA/HSUPA can be available.

In our implementation we have the automatic registration enabled. Therefore, if the module is switched on it tries to establish a network registration either in GSM or in UMTS mode. If both are available, UMTS is preferred. We enable the "+CREG" URC which informs us if the registration was successful. If we have received the corresponding URC, we continue with the Internal PDP context activation. Figure 5.6 shows how the network registration should be performed and how the +CREG URC's need to be handled.

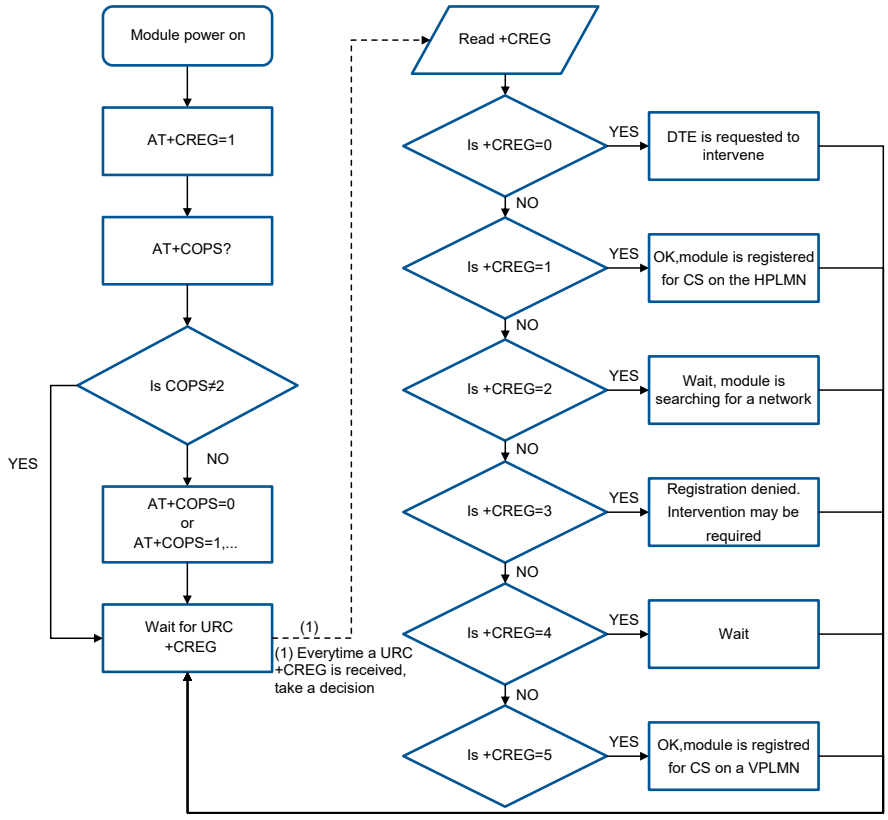


Figure 5.6: CS and PS network registration flow chart [17]

5.5.3 Internal PDP context activation

Before we can use TCP/IP we need to create a packet switched data (PSD) profile. This profile is used to set PDP context parameters for an internal context. This is, the protocol type (IPv4, IPv6), the APN together with username and password, the DNS address, the authentication, the IP address (dynamic, static) and the Quality of Service parameters (precedence, delay, reliability, peak rate, mean rate, delivery order, etc.). The profile can be stored in the NVM. Once configured one can activate the PDP context with the specified profile.

In our implementation we first register the cellular module to the general packet radio service (GPRS). GPRS allows 2G and 3G networks to transmit IP packets to networks such as the internet. Next, we load our PSD profile and activate the PDP context. After the activation we have an IP address assigned to our module.

5.5.4 Data connection

In order to transfer data reliable, we use TCP/IP. Therefore, we create a socket on the cellular module. For the next step we need the IP address and the port from the remote host we want to connect to. We can do that with a DNS lookup or we hardcode the IP address and the port. As we used YAT as dummy server to test the connection we hardcoded the IP and the port and established the connection. If the connection is successful we get an OK as response. In the next step we enter the direct link mode. This pushes the module in data mode (see chapter 4.2.1). After "CONNECT" is received everything send to the module is forwarded to the server. If we want to close the connection, we send +++ to exit the data mode and then we close the socket. Figure 5.7 shows how the cellular module boots and establishes a TCP/IP connection with YAT.

```
=============
START
=============
Cellular Module Running

-
AT command: AT+CGMM

Response:
SARA-U270
0
-
-
AT command: AT+CGMI

Response:
u-blox
0
-
-
AT command: AT+CMEE=1

Response:
0
-
-
AT command: AT+CREG=1

Response:
0
-
-
AT command: AT+CREG?

Response:
+CREG: 1,0
0
-

URC: +CREG: 1

-
AT command: AT+CREG?

Response:
+CREG: 1,1
0
-
-
AT command: AT+COPS?

Response:
+COPS: 0,0,"Swisscom",0
0
-
-
AT command: AT+UREG?

Response:
+UREG: 0,2
0
-
-
AT command: AT+URAT?

Response:
+URAT: 1,2
0
-
-
AT command: AT+CGATT?

Response:
+CGATT: 1
0
-
-
AT command: AT+UPSND=0,8

Response:
+UPSND: 0,8,0
0
-
-
AT command: AT+UPSDA=0,2

Response:
0
-
-
AT command: AT+UPSDA=0,3

Response:
0
-
-
AT command: AT+UPSND=0,8

Response:
+UPSND: 0,8,1
0
-
-
AT command: AT+UPSND=0,0

Response:
+UPSND: 0,0,"10.185.2.127"
0
-
-
AT command: AT+USOCR=6

Response:
+USOCR: 0
0
-
-
AT command:
AT+USOCO=0,"82.130.102.220",
10000

Response:
0
-
-
AT command: AT+USODL=0

Response:
CONNECT
-

URC: COMMAND FROM REMOTE
HOST

-
AT command: +++
Response:
-

URC: 0

-
AT command: AT+USOCL=0

Response:
0
-
----------
Cellular Module OFF
----------
```

Figure 5.7: Boot, network registration, PDP context activation, socket creation and connection

Chapter 6

Further work

In this chapter we will briefly summarize what has been done and what hasn't, as well as hints how to proceed with this work.

6.1 Achieved goals

The software interface between cellular module and MCU has been implemented. The interface is able to transmit all kind of AT commands and to react to the necessary requirements. It is able to detect URC's and provides an interface which allows an easy handling of them. The interface is also able to detect data transmitted by the remote host. Up to now there are no action taken if such data is received. The interface just outputs the received message and continues. All kind of errors are caught and are forwarded to an error handler. This error handler takes decisions how to handle the specific error.

As the goal was to implement and to include the interface in the already existing WGPS software we only managed to implement the interface. It's not included yet although everything is prepared such that the integration is the next logical step.

6.2 Follow-up

According to the objectives, this work is designed in a way that follow-up projects can be built upon. Therefore, special care has been taken such that everything is implemented in a structured and generic way so that modification can be done as simply as possible.

Defines

The code uses no hard-coded values for AT command settings which can be changed. All of these values are mapped with defines. Therefore, if one changes for instance, the s3_character, the only thing needed to adapt to is the corresponding define. All defines are located in the header file "gsm.h".

TODO's

If one wants to integrate the provided code into the wireless GPS sensor system code, there are several places where one needs to adapt the code. For sake of simplicity, the TODO's mark these places. Every TODO contains also a short text which explains what is thought to be adapted.

README's

The repository on Gitlab contains README's which explains what needs to be done in more details.

Code comments

The code contains comments everywhere where it could cause understanding problems.

Inclusion into the wireless GPS rev2 sensor system code

This part will give a short summary of what needs to be done in order to integrate the code into the GPS sensor system code. The code is already implemented with FreeRTOS. The most difficult part will be to adjust the interrupt and task priority so that it harmonizes with the other interrupts and tasks. Then, one needs to select the appropriate clock source for the USART2.

As the wireless GPS sensor system can enter the sleep mode, one needs to consider that some variables may need to be volatile such that the value is stored before entering sleep mode. Next, one needs to define how the data is provided to the task and how it needs to be sent such that the PermaSense data backend system is able to process the data.

Chapter 7

Conclusion

In this work we proposed an integration concept which allows efficient communication with the cellular module. Also, we proposed an efficient monitoring based on the FreeRTOS task which allows us to monitor unsolicited result codes (URC's), all kind of errors and the result of sent AT commands. Although never tested with the PermaSense data backend system itself, we were able to establish a reliable TCP/IP connection with a dummy server, hosted on a computer and to exchange data in both directions. As this work is thought to be continued, the entire code is designed in a manner that follow-up projects can be easily build upon. Comments and TODO's help to understand what needs to be done and how the code should be understood. No hard-coded values are used for values which can be changed on the cellular module. Defines are used instead, to apply the changes globally.

The proposed code is completely embedded in the ARM/FreeRTOS software framework and has been developed and tested on the wireless GPS rev2 sensor system. Therefore, the integration in the already existing wireless GPS rev2 sensor system code is straight forward.

Bibliography

[1] Atollic TrueStudio, http://www.st.com/en/development-tools/truestudio.html

[2] STM32L496VG microcontroller, http://www.st.com/en/microcontrollers/stm32l496vg.html

[3] SEGGER J-Link EDU, https://www.segger.com/products/debug-probes/j-link/models/j-link-edu/

[4] U-blox SARA-U270 GSM Module,https://www.u-blox.com/de/product/sara-u2-series

[5] ITU-T Recommendation V24, 02-2000. List of definitions for interchange circuits between Data Terminal Equipment (DTE) and Data Connection Equipment (DCE)

[6] u-blox Cellular Modules, Data and Voice Modules, AT Commands Manual, Document number: UBX-13002752, Revision R56, 22-Nov-2017

[7] SARA-U2 series, HSPA modules with 2G fallback, Data Sheet. Document number: UBX-13005287, Revision R19, 26-Feb-2018

[8] STM32L496xx Datasheet. Document number: DS11585, Revision 8, May 2018

[9] STM32L496 Refrence Manual. Document number: RM0351, Revision 6, April 2018

[10] Description of STM32L4/L4+ HAL and low-layer drivers. User manual. Document number: UM1884, Revision 7, September 2017

[11] Mastering the FreeRTOS Real Time Kernel, A Hands-On Tutorial Guide. Richard Barry, 2016

[12] The FreeRTOS Refrence Manual, API Functions and Configuration Options. Version 10. Amazon Web Services, 2017

[13] Shockfish SA. Tinynode 184. http://www.tinynode.com/?q=product/tinynode184/tn-184-868

[14] Functional Specification of the next-gen WGPS device, wgps_rev2_fs.pdf. Akos Pasztor, ETH Zurich TIK, 2016

[15] https://www.freertos.org/RTOS-task-states.html, online, 08.06.2018

[16] https://www.freertos.org/a00110.html, online, 08.06.2018

[17] AT Commands Examples, Examples for u-blox cellular modules, Application Note. Document number: UBX-13001820, Revision R11, 22-Sep-2016

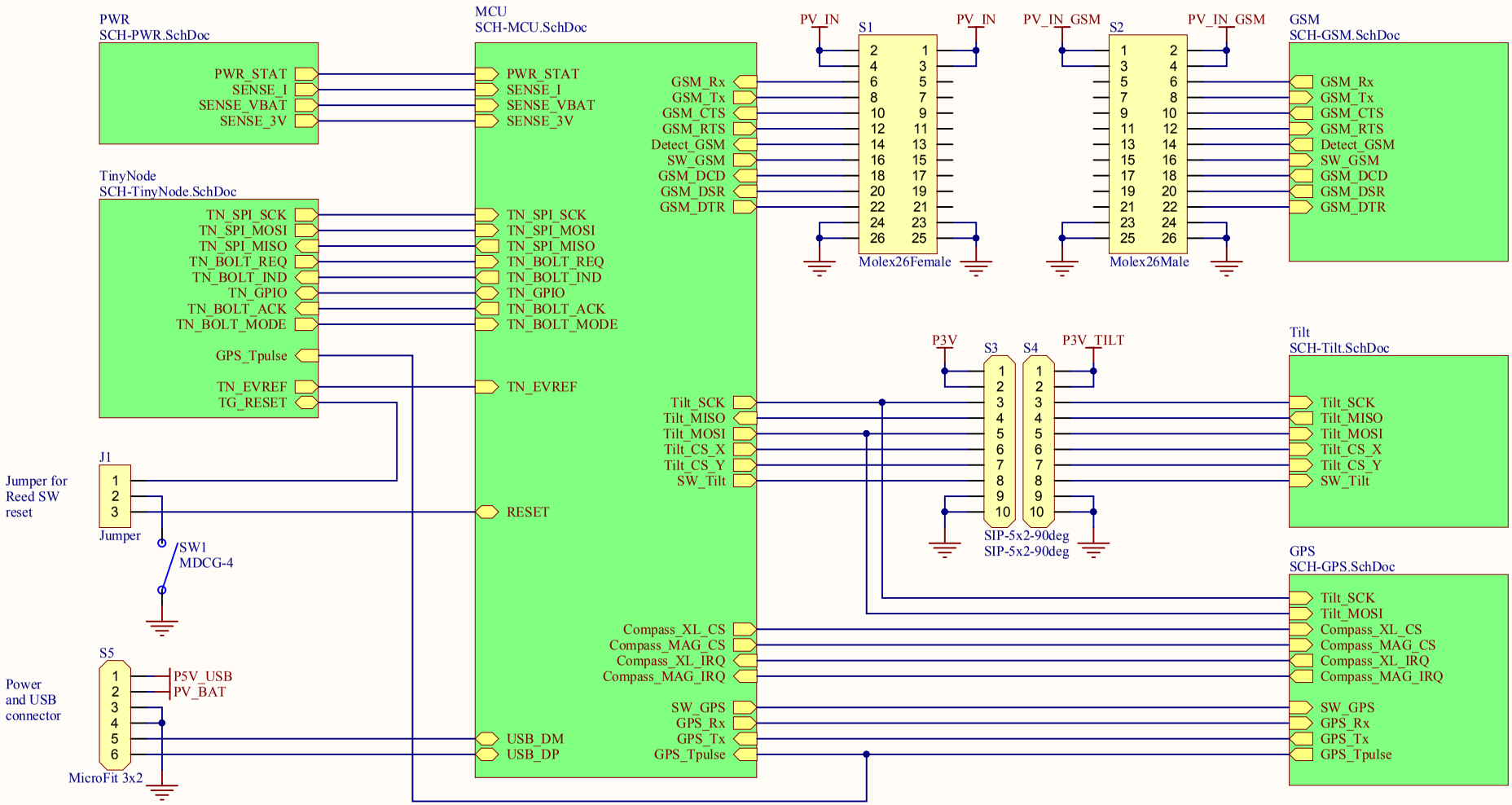
[18] http://www.telecom.tuc.gr/~perak/speech/DSR-issues/report-html/, online, 10.06.2018

[19] PermaSense Data Management: System documentation and tutorial for online data access. Samuel Weber, Jan Beutel, Christoph Walser, 2014

List of Figures

Appendix A

WGPS Schematics



Permasense Logo | ETHZürich | ETH | ETH | ETH

H1 H2 H3 H4

ETH

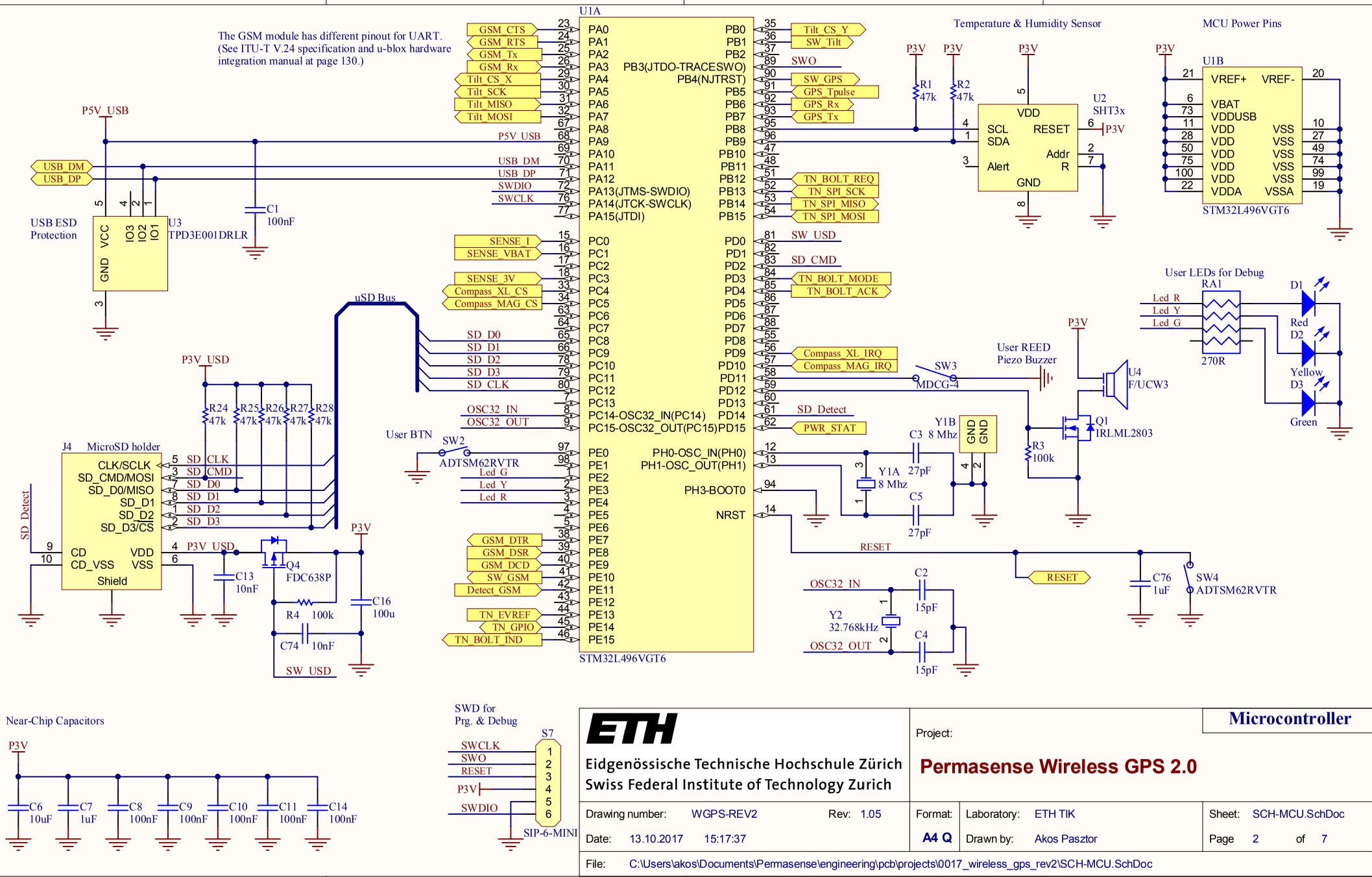
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

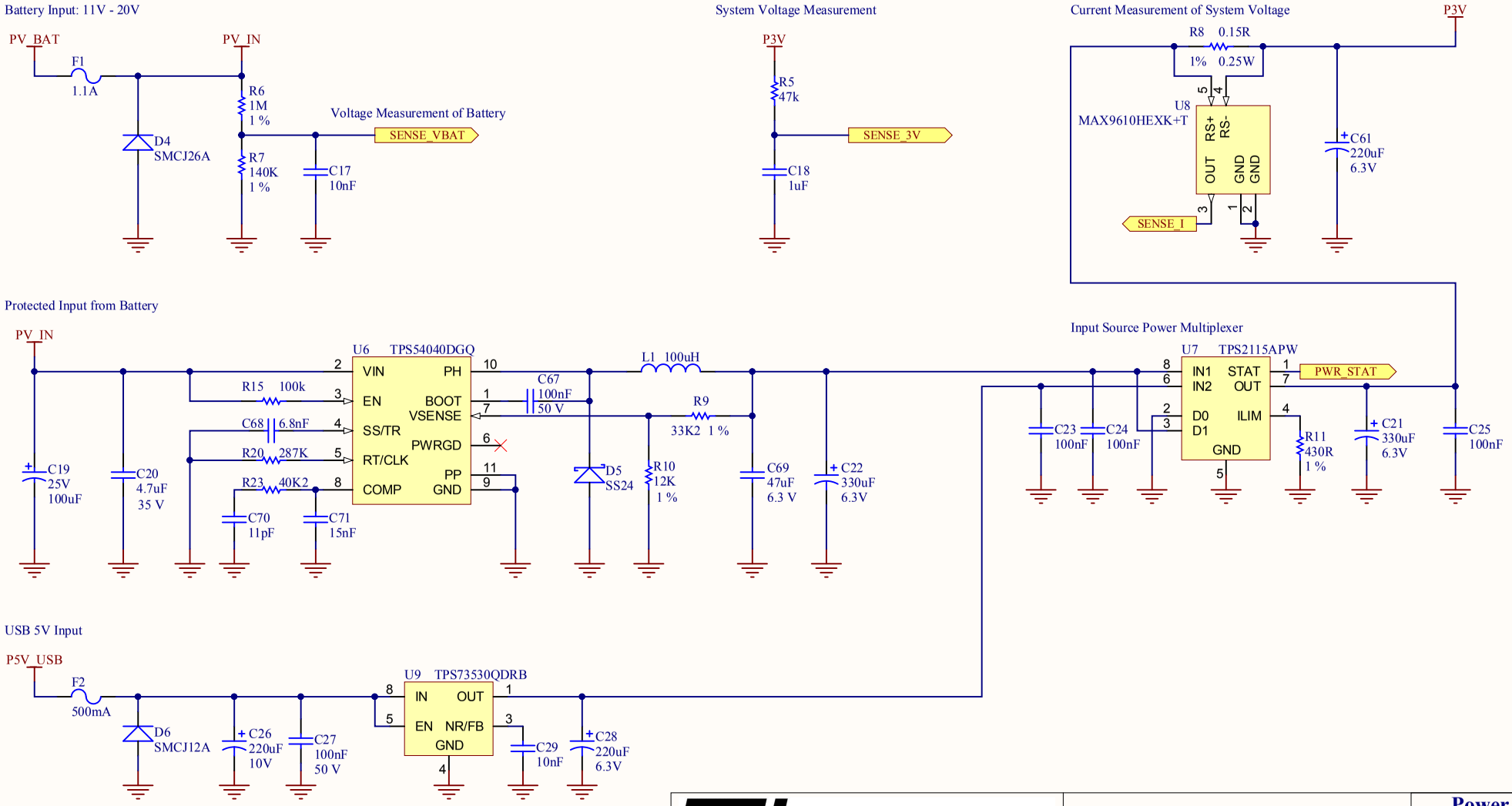
Project:

Permasense Wireless GPS 2.0

Overview

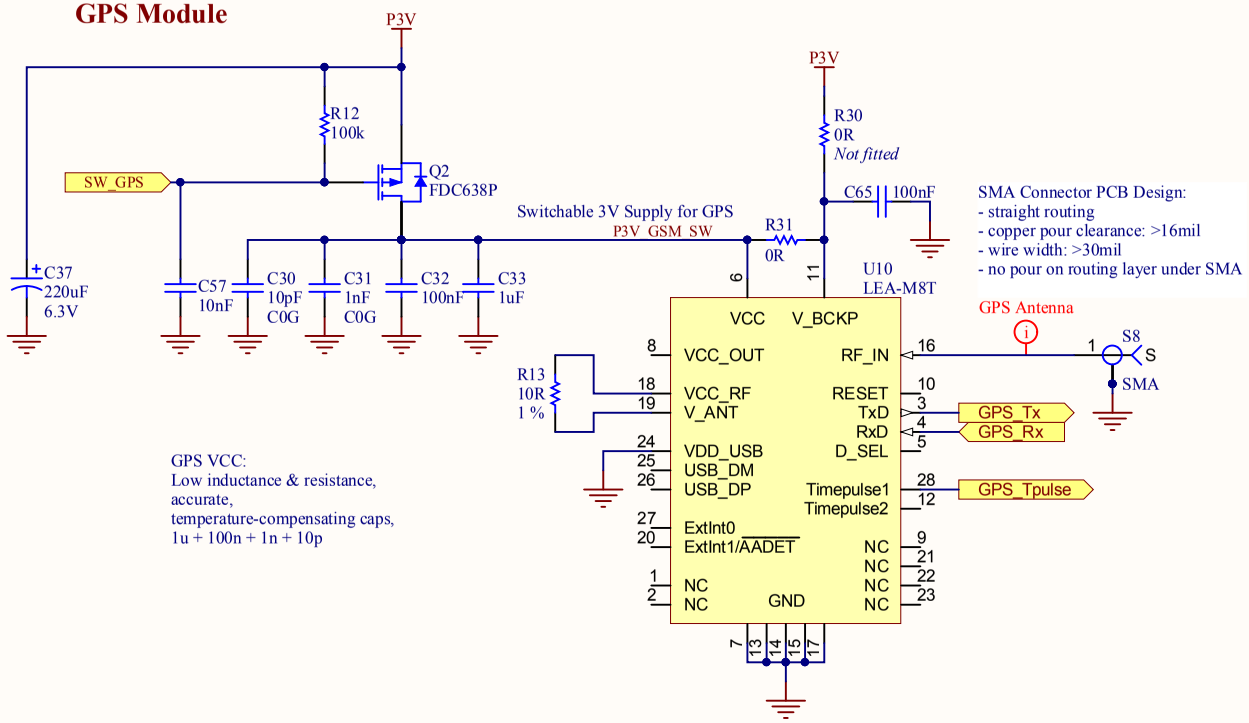
Drawing number:	WGPS-REV2	Rev: 1.05	Format: A4 Q	Laboratory: ETH TIK	Sheet: SCH-Connectivity.SchDoc
Date:	13.10.2017	15:17:37		Drawn by: Akos Pasztor	Page 1 of 7
File:	C:\Users\akos\Documents\Permasense\engineering\pcb\projects\0017_wireless_gps_rev2\SCH-Connectivity.SchDoc				


The GSM module has different pinout for UART.
(See ITU-T V.24 specification and u-blox hardware integration manual at page 130.)
U1A
STM32L496VGT6
Temperature & Humidity Sensor
U2
SHT3x
MCU Power Pins
U1B
STM32L496VGT6
P5V_USB
USB ESD Protection
U3
TPD3E001DRLR
C1
100nF
uSD Bus
P3V_USD
J4
MicroSD holder
SD_Detect
C13
10nF
Q4
FDC638P
R4
100k
C74
10nF
C16
100u
SW_USD
User BTN
SW2
ADTSM62RVTR
User LEDs for Debug
RA1
270R
D1
Red
D2
Yellow
D3
Green
User REED
Piezo Buzzer
SW3
MDCG-4
U4
F/UCW3
Q1
IRLML2803
R3
100k
Y1B
8 Mhz
C3
27pF
Y1A
8 Mhz
C5
27pF
RESET
C76
1uF
SW4
ADTSM62RVTR
OSC32_IN
OSC32_OUT
Y2
32.768kHz
C2
15pF
C4
15pF
Near-Chip Capacitors
C6
10uF
C7
1uF
C8
100nF
C9
100nF
C10
100nF
C11
100nF
C14
100nF
SWD for
Prg. & Debug
S7
SIP-6-MINI
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich
Project:
Permasense Wireless GPS 2.0
Microcontroller
Drawing number: WGPS-REV2
Rev: 1.05
Date: 13.10.2017 15:17:37
Format: A4 Q
Laboratory: ETH TIK
Drawn by: Akos Pasztor
Sheet: SCH-MCU.SchDoc
Page 2 of 7
File: C:\Users\akos\Documents\Permasense\engineering\pcb\projects\0017_wireless_gps_rev2\SCH-MCU.SchDoc



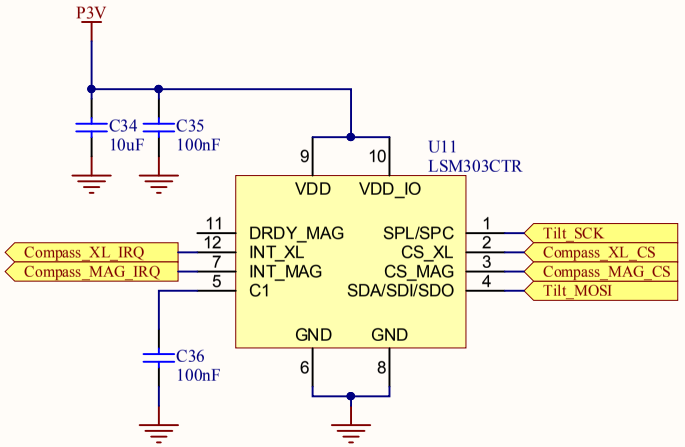
ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich	Project: **Permasense Wireless GPS 2.0**		**Power Supply**
Drawing number: WGPS-REV2 Rev: 1.05 Date: 13.10.2017 15:17:37	Format: **A4 Q**	Laboratory: ETH TIK Drawn by: Akos Pasztor	Sheet: SCH-PWR.SchDoc Page 3 of 7
File: C:\Users\akos\Documents\Permasense\engineering\pcb\projects\0017_wireless_gps_rev2\SCH-PWR.SchDoc			

GPS Module



E-Compass 3D accelerometer & 3D magnetometer

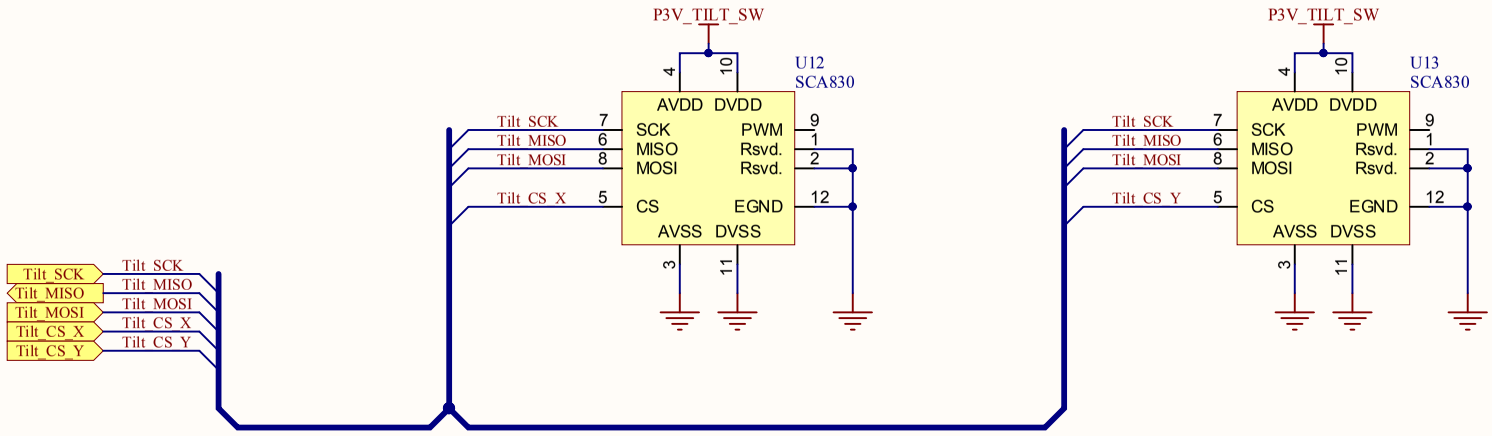
Supply for Ultra-Low Power E-Compass is not needed to be switchable; always ON



ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich	Project: Permasense Wireless GPS 2.0			GPS & Compass
Drawing number: WGPS-REV2 Rev: 1.05	Format: A4 Q	Laboratory: ETH TIK	Sheet: SCH-GPS.SchDoc	
Date: 13.10.2017 15:17:37		Drawn by: Akos Pasztor	Page 4 of 7	
File: C:\Users\akos\Documents\Permasense\engineering\pcb\projects\0017_wireless_gps_rev2\SCH-GPS.SchDoc				

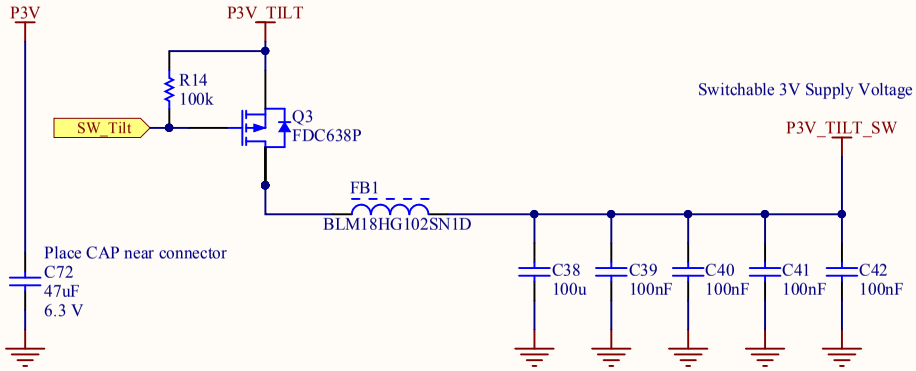
Tilt Sensors

Place Tilt Sensors with 90° different alignment

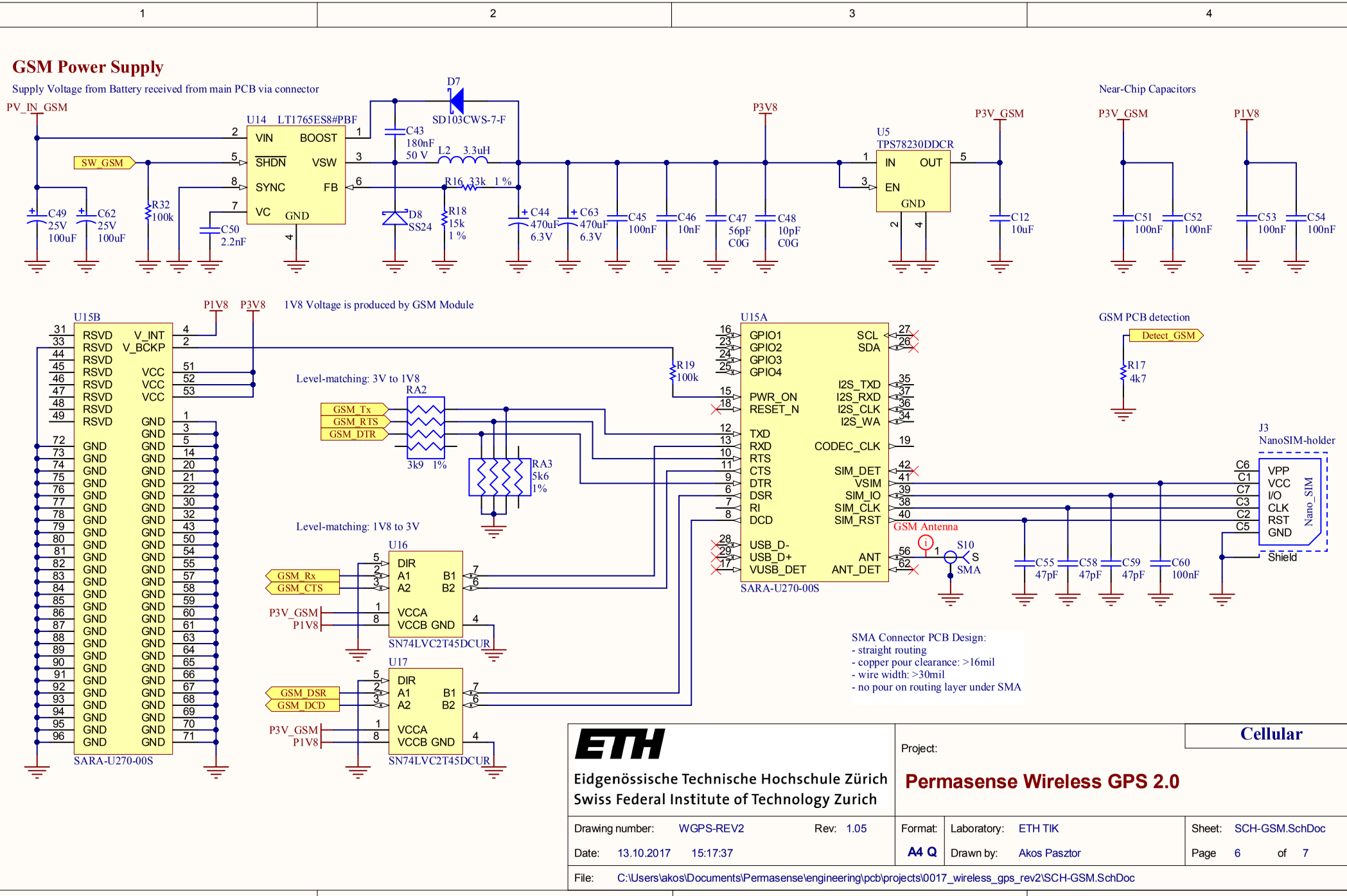


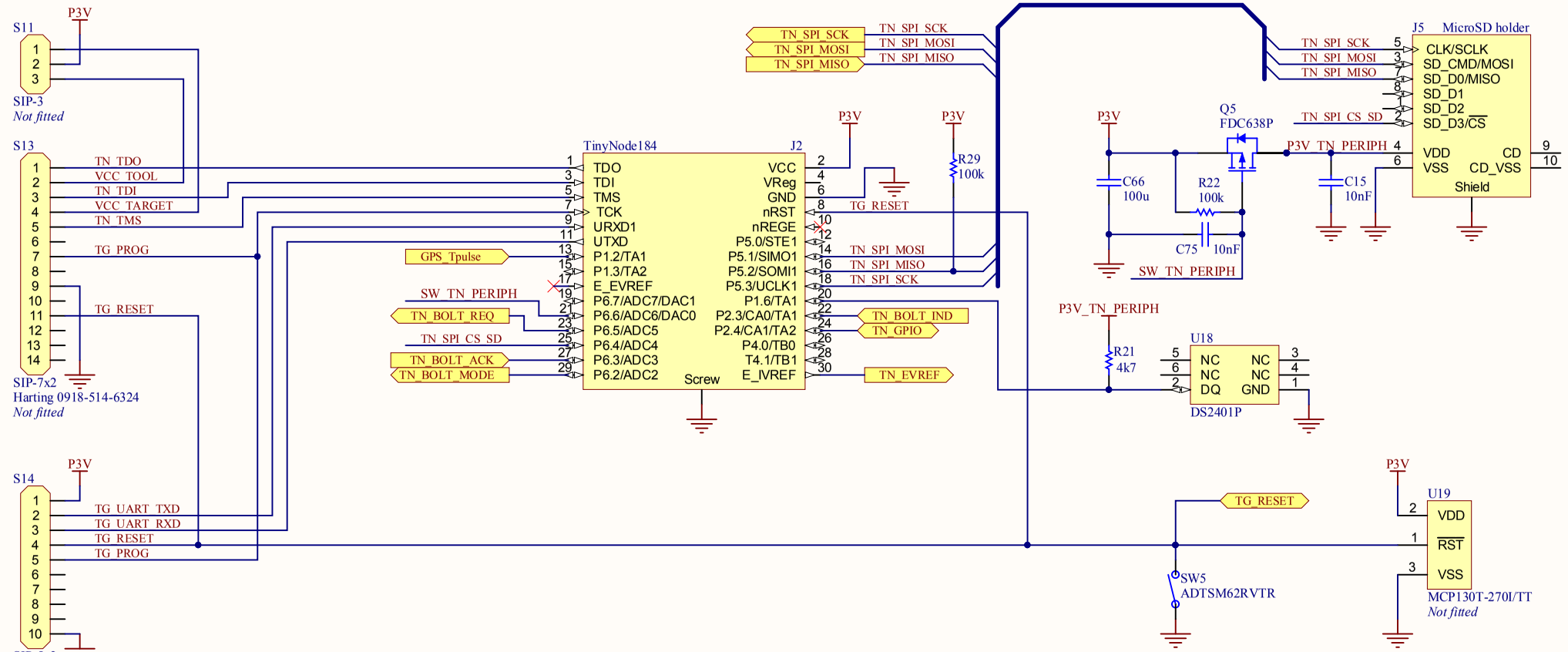
3V received from main PCB via connector

Switchable 3V Supply Voltage for Tilt Sensors



ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich	Project: Permasense Wireless GPS 2.0			Tilt Sensors
Drawing number: WGPS-REV2 Rev: 1.05	Format: A4 Q	Laboratory: ETH TIK	Sheet: SCH-Tilt.SchDoc	
Date: 13.10.2017 15:17:37		Drawn by: Akos Pasztor	Page 5 of 7	
File: C:\Users\akos\Documents\Permasense\engineering\pcb\projects\0017_wireless_gps_rev2\SCH-Tilt.SchDoc				


GSM Power Supply
Supply Voltage from Battery received from main PCB via connector
PV_IN_GSM
U14 LT1765ES8#PBF
VIN
BOOST
SHDN
VSW
SYNC
FB
VC
GND
SW_GSM
D7
SD103CWS-7-F
C43 180nF 50 V
L2 3.3uH
R16 33k 1 %
D8 SS24
R18 15k 1 %
C49 25V 100uF
C62 25V 100uF
R32 100k
C50 2.2nF
C44 470uF 6.3V
C63 470uF 6.3V
C45 100nF
C46 10nF
C47 56pF C0G
C48 10pF C0G
P3V8
U5 TPS78230DDCR
IN
OUT
EN
GND
P3V_GSM
C12 10uF
Near-Chip Capacitors
P3V_GSM
C51 100nF
C52 100nF
P1V8
C53 100nF
C54 100nF
U15B
SARA-U270-00S
P1V8
P3V8
1V8 Voltage is produced by GSM Module
R19 100k
Level-matching: 3V to 1V8
RA2
3k9 1%
RA3 5k6 1%
GSM_Tx
GSM_RTS
GSM_DTR
Level-matching: 1V8 to 3V
U16
SN74LVC2T45DCUR
GSM_Rx
GSM_CTS
U17
SN74LVC2T45DCUR
GSM_DSR
GSM_DCD
U15A
SARA-U270-00S
GSM PCB detection
Detect_GSM
R17 4k7
J3
NanoSIM-holder
VPP
VCC
I/O
CLK
RST
GND
Shield
Nano_SIM
GSM Antenna
S10
SMA
C55 47pF
C58 47pF
C59 47pF
C60 100nF
SMA Connector PCB Design:
- straight routing
- copper pour clearance: >16mil
- wire width: >30mil
- no pour on routing layer under SMA
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich
Project:
Permasense Wireless GPS 2.0
Cellular
Drawing number: WGPS-REV2
Rev: 1.05
Date: 13.10.2017 15:17:37
Format: A4 Q
Laboratory: ETH TIK
Drawn by: Akos Pasztor
Sheet: SCH-GSM.SchDoc
Page 6 of 7
File: C:\Users\akos\Documents\Permasense\engineering\pcb\projects\0017_wireless_gps_rev2\SCH-GSM.SchDoc



ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich	Project: Permasense Wireless GPS 2.0		TinyNode
Drawing number: WGPS-REV2 Rev: 1.05	Format: A4 Q	Laboratory: ETH TIK	Sheet: SCH-TinyNode.SchDoc
Date: 13.10.2017 15:17:37		Drawn by: Akos Pasztor	Page 7 of 7
File: C:\Users\akos\Documents\Permasense\engineering\pcb\projects\0017_wireless_gps_rev2\SCH-TinyNode.SchDoc			