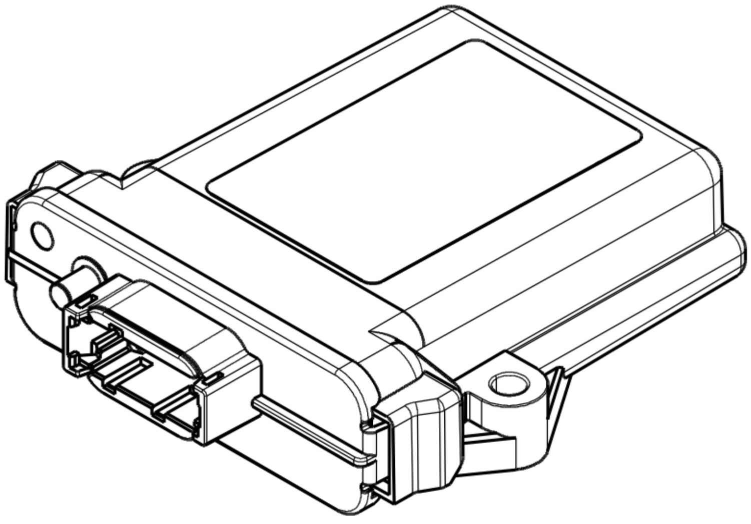


Microcontroller Adaptation within the Telematics Domain

Master of Science Thesis in Computer Systems and Networks

MARCUS LAURILA
JOHNNY VÄYRYNEN

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden 2014

Gothenburg, Sweden 2014

MARCUS LAURILA
JOHNNY VÄYRYNEN

Examiner: ROGER JOHANSSON

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden June 2014

Abstract

The ever-growing field of telematics spans over a wide array of heterogeneous peripherals such as GPS units, accelerometers, alarms and smart cards. Gathering, handling and storing data or transmitting it to a server over the air introduces demands on adaptability, robustness and configurability of the central unit being used.

The MX-3 telematics platform series developed by *Host Mobility AB* has proven to meet the functional demands placed upon it but features an unnecessary burden in the form of a heavy and expensive GPRS modem running the customer application using Java ME (Micro Edition) besides the C-flavoured core functionality of the contained PIC microprocessor.

This project aims to improve the robustness, energy and cost efficiency of the system by moving all of the custom application code to the microprocessor, eliminating the need for an expensive modem able to support Java. Challenges to be faced include creating a complete and robust C code base as well as identifying demands on the microcontroller and modem in the new setup and matching them with appropriate hardware.

During this project we have made appropriate hardware changes as well as adapted and augmented the former code base in order to develop a prototype for a new telematics platform series based on the MX-3 called the MX Mini. The results section of this paper presents the achieved architectural improvements and projected production cost savings.

Acknowledgements

We would like to express our gratitude towards Host Mobility AB and all of its employees for providing us with this very work opportunity and patiently providing us with information and assistance throughout. Our workplace supervisor Albin Dennevi deserves a special mention for always being there and ascertaining expected functionality from the many pieces of hardware included throughout.

Furthermore, our examiner Roger Johansson has been ever so helpful by sharing with us words of wisdom from his many years of experience within the area and making us retain a keen eye to details whilst not having the bigger picture fall into oblivion.

Last but not least, we would like to thank our families for always being there and ascertaining that our fridges retain adequate quantities of delicious yet nutritious contents through times of personal poverty induced by ingesting excessive amounts of malt beverages.

The Authors, Göteborg 2014

Acronym Listing

ACK	Acknowledgement
ANSEL	Analog Select
API	Application Programming Interface
AT	Attention
CAN	Controller Area Network
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
ECAN	Enhanced Controller Area Network
FTP	File Transfer Protocol
GPRS	General Packet Radio Service
GPS	Global Positioning System
GSM	Global System for Mobile Communications
HTTP	Hypertext Transfer Protocol
I/O	Input/Output
IDE	Integrated Development Environment
IP	Internet Protocol
LED	Light-emitting diode
MIPS	Million Instructions Per Second
MITM	Man-In-The-Middle
NMEA	National Marine Electronics Association
PHP	PHP Hypertext Preprocessor
PIC	Peripheral Interface Controller
POP3	Post Office Protocol v3
PPP	Point-to-point Protocol
RAM	Random Access Memory
ROM	Read-only Memory
RP	Remappable Peripheral
RX	Receive
SD	Secure Digital
SMS	Short Message Service
SMTP	Simple Mail Transfer Protocol
SPI	Serial Peripheral Interface
SQL	Structured Query Language
TCP	Transmission Control Protocol
TX	Transmit
UART	Universal Asynchronous Receiver/Transmitter
UDP	User Datagram Protocol
URC	Unsolicited Result Code

Contents

1

Introduction

The Global Positioning System (GPS) offers possibilities to localize objects with high precision anywhere on Earth. In combination with General Packet Radio Services (GPRS) it is possible to provide real time tracking of vehicles through for example Short Message Service (SMS). The system to be developed will be able to track a vehicle's position in real time or for a given time frame by analyzing data received from the GPS module. In addition to location tracking, the system can also provide information about for example velocity and temperatures depending on the peripherals being used.

Host Mobility AB is a company located in Gothenburg, Sweden specializing in developing telematics platforms used in ground based vehicles and naval vessels. There exists today a platform series called MX-3, communicating over the MX protocol which is a proprietary protocol courtesy of Host Mobility. The MX-3 unit is displayed in Figure 1.1 and a high level schema is illustrated in Appendix A. Presently, a Java-flavoured *midlet platform* in the GPRS modem contains customer-specific software handling the connection to the server and the prioritization of data exchanged through this. Alongside the modem there is a *co-processor* in the form of a PIC microcontroller based on C code. This microcontroller handles all peripherals and I/O of the unit such as memory cards, GPS units and accelerometers.

The microcontroller communicates with the modem using the MX protocol and the customer application is made to facilitate the API featured by the modem.

1.1 Purpose

No standardized architectures exist for the applications targeted by MX-3 and the current approach is limited in its flexibility. The MX-3 solution as a whole suffers from decreased cost efficiency due to having to use a specialized GPRS modem in order to both handle communication and arbitrary customer application code. It would instead be desirable to make use of a simple and inexpensive modem only handling the communication and

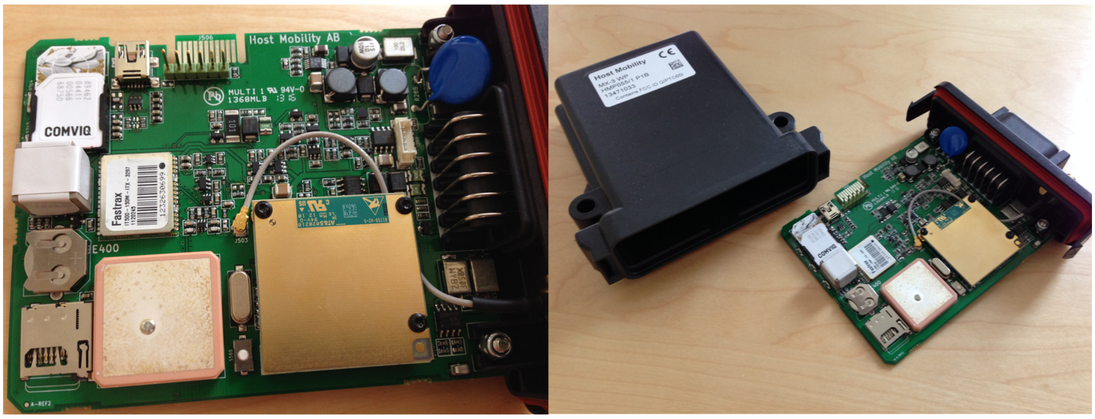


Figure 1.1: The MX-3 unit.

not the application. Furthermore, a less complex modem would consume less energy as well as simplify the *sleep mode* of the entire module. In a solution where the modem only acts as a slave unit with no control over decision-making, the issue about synchronizing sleeps and wake-ups between two processors vanishes, trivializing methods of handling low energy modes. Low energy consumption is crucial when the only source of power is a battery.

Allocating application logic to modem software not only enforces the use of an expensive and specialized modem, but also burdens its processing capabilities and complicates the overall architecture. Incorporating this unfitting burden in its entirety within the microcontroller would be a preferable approach. The microcontroller will then act as a *master unit* to the modem, as opposed to using the previous, codependent architecture. Future customer applications will thus be run in the microcontroller, and make use of the C programming language rather than Java.

Unfortunately, the lack of a preceding setup gives rise to multiple uncertainties. One challenge paralleling the entire design process will be to assess if a PIC will be able to handle the composite workload with regard to memory and processing capacity. Handling both the customer application and the acquisition of data from peripherals might give rise to a need for more customized scheduling of the clock cycles. While all of the PIC's processing power could previously be used for peripheral data connection and responding to requests from the application in the GPRS modem, the very same clock cycles will now be actually running the customer application. The program flow thus needs to be carefully planned to ensure that mundane tasks are executed only on demand, and the customer application may utilize processing power to the largest possible extent.

1.1.1 Technical goals

The microcontroller needs to be able to interoperate with differing slave GPRS modems in order to promote independence from specific vendors. The customer application shall via the use of well-defined function calls be able to operate with all the peripherals the MX-3 supports, and forward the data gathered from these to an external server. Writing the server application receiving the aforementioned data also belongs to the set of goals, and will be performed in a Linux environment using *GPRS sockets*. The architecture needs to support arbitrary protocols for communicating with the PIC, with a default implementation using a derivative of the MX protocol included in the product. If the PIC proves to be inadequate, the type of microcontroller to be used as the central unit shall be re-evaluated to fit the final needs.

1.2 Objectives

Ultimately, the aim of the project is to decrease the production cost of a complete MX unit along with increasing its energy efficiency and thus increase its areas of application.

The energy consumption of the preceding MX-3 unit is not of highest concern. There is support for Li-ion batteries, but the majority the customers run it with a power source and hence use the battery as a backup source for short periods of possible power failures. The challenge of the former system, in terms of power efficiency, was to synchronize the sleeps between the Java ME[1] application running in the modem and the C code running in the microcontroller. When it comes to power efficiency, having the module sleep during periods of inactivity is very important.

The new implementation, on the other hand, will make it possible to replace the modem with an inexpensive and less power demanding variant. This will lead to lower production costs and decreased power consumption which opens up for a whole new market. Such a lightweight product used in detached small or hand-held tools like assault or moped/motorcycle alarms would be feasible.

1.3 Related work

El-Medany et al. [1] and Lita et al. [2] discuss in their similarly themed papers how a *low cost localization system* consisting of microcontroller, GPS and GPRS modem can be implemented. El-Medany's system communicates with a monitoring server connected to a SQL database, plotting vehicle position information on Google maps. The latter puts more effort in communication via SMS, e.g. notifications from a vehicle leaving a delimited area, having its engine started/stopped or car alarm go off. Al-Khadher presents in his paper [3] another similar solution featuring a GSM modem and tracking via Google maps. Nevertheless, the emphasis of Al-Khadher's paper lies in how a Kalman filter may improve the accuracy of GPS coordinates through *error correction*. Yet another implementation is presented in the paper of Zhang et al [4]. The authors compare the data transmission protocols TCP and UDP and prefer UDP due to larger throughput albeit at the expense of reliability. In the same paper, an analysis was also made about transmission delays over GPRS.

[1] Java ME (Micro Edition) is a Java platform designed for embedded systems.

As in El-Medany's and Al-Khadher's projects, Google maps is used in our implementation in order to plot GPS coordinates. An advantage of Google maps is that it is free and supports most GPS devices' data formats directly. When it comes to transmission protocol, our choice was TCP, in contrast with Zhang's implementation. None of these unfortunately investigate the scenario of having several peripheral units connected to the microcontroller. There is a certain complexity in handling the input, including TCP packets, from the GPRS modem while simultaneously keeping track of the connected peripherals. This brings another dimension to the project and the implementation is subject to the challenges of parallelism and hardware assessment.

1.4 Report outline

The report is further outlined as follows:

- **Chapter 2** describes the actual development process and its subcomponents in detail.
- **Chapter 3** clarifies gains from the old to the new product.
- **Chapter 4** summarizes the findings of the project in its entirety as well as describes items that were considered, but did not fit within our given time frame.

2

Implementation

Throughout the project, MPLAB[1] [5] in conjunction with the XC16 C-compiler[2] [6] has been used as the primary IDE with which coding for the PIC microcontroller has been done and the compiled code burnt into the flash memory. Git has been used as *version control system*, see Appendix B.

AT-command testing via Tera Term[3] and a serial-USB converter has been used to complement the communication between the GPRS modem and the microcontroller. Section 2.8.3 provides more detailed information about debugging.

A server managing the raw data sent from the MX Mini unit has been created, and peripherals have been adapted to use one at a time and their respective data added to the communication between the MX Mini and the server, see section 2.6.1. This data has been continuously inspected to assure the correctness of each peripheral functionality, and to verify the microcontroller's ability to properly handle simultaneous use of them all. The server is written in Java and Eclipse has been used as IDE.

2.1 GPRS Modem

The modem being used in the implementation, Cinterion TC65i, is based on GPRS technology. GPRS is a packet-data technology that allows GSM operators to provide wireless data services such as Internet access. GPRS is built on the *GSM network platform*, so operators can keep but enhance their existing infrastructure. The GPRS core network is based on Internet Protocol (IP) standards, yielding support for connection to

[1] MPLAB is a free Integrated Development Environment (IDE) for development of embedded applications on Microchip's microcontrollers.

[2] The MPLAB XC16 C-compiler fully supports all Microchip 16-bit devices. The compiler optimizes and translates standard ANSI C programs into 16-bit device assembly language.

[3] Tera Term is an open-source terminal emulator that emulates different kinds of computer terminals. There is support for telnet, SSH and serial port connections.

IP-based networks. There is support for PPP, which is used to tunnel IP to the device, allowing an IP address to be dynamically assigned to the MX unit.

GPRS is a *best-effort service*, implying variable throughput and latency that depend on the number of users sharing the service at the time. It supports download rates of up to 115 kbps, which is adequate for the system being developed. GPRS provides an always-on data connection, so the user does not need to reconnect every time they want to access new data. Due to the packet-switched architecture, the users will only need to pay for the data itself rather than the airtime to establish a connection and download data.

Furthermore, GPRS is widely supported in the world and there is support for international roaming, making it possible to access data services at most places in the world. Even though an area has not yet been upgraded to GPRS, many data services may still be accessed through *circuit-switched GSM*[4].

2.1.1 Types of modem messages

Besides echo characters, there are two kinds of message to receive from the modem: AT-replies and URCs. Every AT-reply starts with an echo of the request message, followed by a response. URCs, on the other hand, are unexpected messages sourcing from the modem, which have to be handled separately. URCs may arrive spontaneously, as with the case of RING / SMS commands, or as asynchronous responses to certain AT requests, primarily vendor specific such.

2.1.2 AT commands

With the modem being the slave unit in the new architecture, it no longer presents a Java application capable of handling communication. Instead, the communication between the PIC master unit and the modem will be done using an extension of the *Hayes command set*[5] for GPRS modems and alike, i.e. AT commands. The TC65i AT Command Set Manual offers detailed information about the specific set of AT commands [7].

AT commands are specific instructions used to control a modem. The ”at” prefix must be provided in the beginning of each command line. The commands may be used to get basic information about the GPRS modem such as name of manufacturer, model number, IMEI number and software version. AT commands can also be used to change modem configuration, setting up socket connections as well as getting current status messages.

In order to terminate a command a `<CR>`, i.e. carriage return, must be present at the end of the string. Since every response has to be parsed out and handled accordingly, one needs to know where a specific message starts and ends. In most cases a command is followed by a response following the format `<CR><LF><response><CR><LF>`. The modem is expected to send a response to almost all requests it receives. These responses can

[4] Circuit-switched data is usually billed per minute rather than by data volume.

[5] The Hayes command set is a command language originally developed for the Hayes Smartmodem 300 in 1981.

AT command	Description
ATI	Display product identification information
AT+CGMI	Request manufacturer identification
AT+COPS	Operator Selection
AT^SICI	Internet connection information
AT^SISC	Internet service close
AT^SISR	Internet service read data
AT^SISW	Internet service write data

Table 2.1: Examples of AT commands.

either be in the form of strings or numeric values. In most cases the response also ends with either `OK` or `ERROR`.

AT commands include a standard set, characterized by having a '+' postfix or none at all after the initial 'at', for instance 'at+cops' and 'ate0'. Furthermore, different vendors may provide implementations with extended functionality commands. Cinterion[6] uses the '^' character to denote their custom commands, for instance in 'at^siss'. There are also commands that accept parameters. See Table 2.1 for a few examples of AT commands.

Although it would be possible to turn off echo-replies to simplify parsing, it is recommended to have them enabled in order to achieve a truly robust structure where every echo is verified.

2.1.3 Unsolicited Result Codes

URCs (Unsolicited result codes) are messages sent from the GPRS modem that provide certain information about the occurrence of an event. In contrast with regular AT replies, those messages are not immediate results of an AT command and may therefore arrive unexpectedly at any time. A few examples of URCs are call, SMS and (optionally) Internet Service related commands[7].

The difficulty of handling URCs lies mainly within the parser. Firstly, there are plenty of different URCs that may arrive. Secondly, one does not know when to expect a certain URC. A solution would be to provide support for a set of URCs and handle them specifically. Then, URCs that are not present in this list will be ignored.

Another solution is to turn off the majority of URCs, although it is not possible to turn off every URC. The benefit would be that they do not need to be handled in the parser, but instead checks have to be done manually with periodic intervals, e.g. if an SMS has arrived.

[6] Cinterion, or its Machine-To-Machine division Gemalto M2M, is a digital security company. Their products allow vehicle and equipment communication over cellular networks.

[7] Internet Service URCs can be manually enabled or disabled.

Unless somehow handled as special cases directly in the AT parser, the current state of the MX unit necessitates action from the customer to handle URCs correctly. Received AT data seemingly not belonging anywhere else is automatically sent to the customer application's URC handler function.

2.2 Microcontroller summary

Initially, the MX module was running a PIC24HJ128GP506A microcontroller. At one point in time, a decision was made to replace it with the superior DSPIC33EP256MU806 model due to an inadequate amount of memory in the former. Both microcontrollers feature a 16-bit Modified Harvard architecture[8] CPU. See section 2.2.9 for further information about the PIC upgrade.

2.2.1 Parallel I/O Ports

The majority of the device pins are shared between peripherals and the *general purpose* I/O ports. The general purpose I/O ports allow the PIC to monitor as well as control devices. Many I/O ports are multiplexed with various functions. The multiplexing depends on the peripheral features and type of device. A pin diagram of dsPIC33E is illustrated in Figure 2.1.

Every I/O port has four registers which are directly associated with the operations of respective ports. The four registers are TRISx, PORTx , LATx and ODCx where 'x' denotes the specific I/O port. Furthermore, each I/O pin on the device has a corresponding bit in the TRIS, PORT and LAT registers.

The TRIS registers are used in order to determine whether each pin associated with the I/O port is an *input* or *output*, e.g. setting the TRIS bit to '1' for a specific I/O pin means that the pin is an input and the other way around. By default, all port pins are defined as inputs.

To access data on an I/O pin one may perform a read from the PORT register. Writing to the PORT register writes the data to the port data latch. One should be careful when performing a *read-modify-write* operation on the PORT register. A write to a port implies that the port pins are read, this value is modified and then it is written to the port data latch. If a read-modify-write operation is done on the PORT registers, with the associated I/O pins set as input, and then at a later stage changed to output, an unexpected value may be output on the I/O pin.

The advantage of using the LAT registers associated with I/O pins is to avoid the issues that may occur when performing read-modify-write instructions. This is because a read from the LAT registers returns the values in the port output latches rather than the actual values on the I/O pins. Hence, read-modify-write operations on the LAT registers eliminate the possibility of writing the input pin values into the port latches. The LAT registers behave in the same way as PORT registers when it comes to writes.

[8] The Modified Harvard architecture is a derivative of the Harvard computer architecture, which allows access to the instruction memory as if it were data.

The ODC registers are used for data control purposes. Every port pin can be individually configured for digital or open-drain output.

Furthermore, there are ANSELx registers controlling the operation type of port pins that may be used as analog instead of digital. According to the datasheet the default value of the ANSELx register is 0xFFFF, meaning that all pins that share analog functions are analog by default [8].

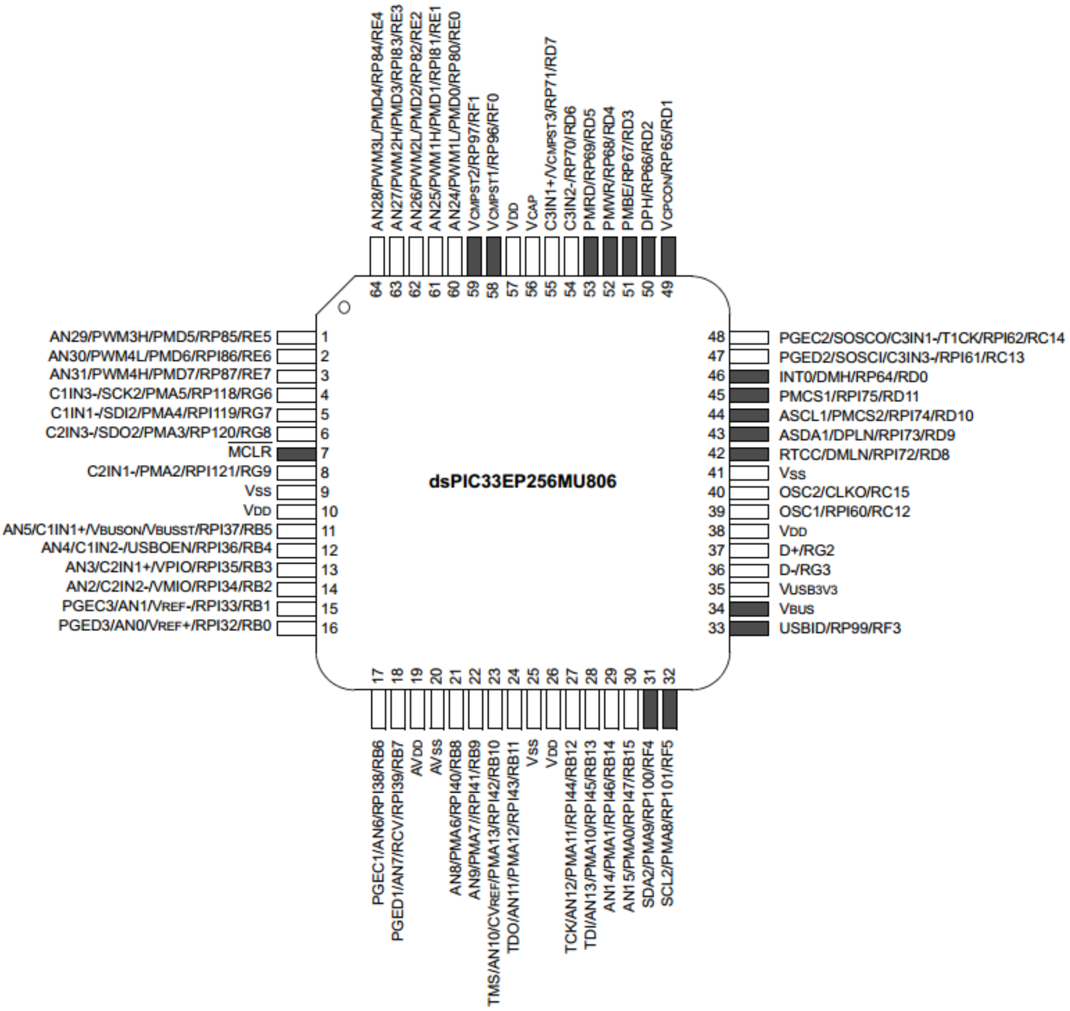

dsPIC33EP256MU806
1 AN29/PWM3H/PMD5/RP85/RE5
2 AN30/PWM4L/PMD6/RPI86/RE6
3 AN31/PWM4H/PMD7/RP87/RE7
4 C1IN3-/SCK2/PMA5/RP118/RG6
5 C1IN1-/SDI2/PMA4/RPI119/RG7
6 C2IN3-/SDO2/PMA3/RP120/RG8
7 MCLR
8 C2IN1-/PMA2/RPI121/RG9
9 VSS
10 VDD
11 AN5/C1IN1+/VBUSON/VBUSST/RPI37/RB5
12 AN4/C1IN2-/USBOEN/RPI36/RB4
13 AN3/C2IN1+/VPIO/RPI35/RB3
14 AN2/C2IN2-/VMIO/RPI34/RB2
15 PGEC3/AN1/VREF-/RPI33/RB1
16 PGED3/AN0/VREF+/RPI32/RB0
17 PGEC1/AN6/RPI38/RB6
18 PGED1/AN7/RCV/RPI39/RB7
19 AVDD
20 AVSS
21 AN8/PMA6/RPI40/RB8
22 AN9/PMA7//RPI41/RB9
23 TMS/AN10/CVREF/PMA13/RPI42/RB10
24 TDO/AN11/PMA12/RPI43/RB11
25 VSS
26 VDD
27 TCK/AN12/PMA11/RPI44/RB12
28 TDI/AN13/PMA10/RPI45/RB13
29 AN14/PMA1/RPI46/RB14
30 AN15/PMA0/RPI47/RB15
31 SDA2/PMA9/RP100/RF4
32 SCL2/PMA8/RP101/RF5
33 USBID/RP99/RF3
34 VBUS
35 VUSB3V3
36 D-/RG3
37 D+/RG2
38 VDD
39 OSC1/RPI60/RC12
40 OSC2/CLKO/RC15
41 VSS
42 RTCC/DMLN/RPI72/RD8
43 ASDA1/DPLN/RPI73/RD9
44 ASCL1/PMCS2/RPI74/RD10
45 PMCS1/RPI75/RD11
46 INT0/DMH/RP64/RD0
47 PGED2/SOSCI/C3IN3-/RPI61/RC13
48 PGEC2/SOSCO/C3IN1-/T1CK/RPI62/RC14
49 VCPCON/RP65/RD1
50 DPH/RP66/RD2
51 PMBE/RP67/RD3
52 PMWR/RP68/RD4
53 PMRD/RP69/RD5
54 C3IN2-/RP70/RD6
55 C3IN1+/VCMPST3/RP71/RD7
56 VCAP
57 VDD
58 VCMPST1/RP96/RF0
59 VCMPST2/RP97/RF1
60 AN24/PWM1L/PMD0/RP80/RE0
61 AN25/PWM1H/PMD1/RPI81/RE1
62 AN26/PWM2L/PMD2/RP82/RE2
63 AN27/PWM2H/PMD3/RPI83/RE3
64 AN28/PWM3L/PMD4/RP84/RE4

Figure 2.1: Pin diagram of dsPIC33E [8].

2.2.2 Peripheral pin select

In order to independently map the input/output of digital peripherals to digital I/O pins, the *Peripheral Pin Select* configuration feature should be applied. There is a fixed subset of digital I/O pins available for this purpose. The Peripheral Pin Select is done in software.

The actual amount of available pins depends on the particular device and its pin count. Pins that support this feature are named ”RPn”, where ”RP” means *remappable peripheral* and ’n’ is the number of the remappable pin.

In the MX mini configuration, the GPS communicates via UART1, the modem via UART2 and the SD-card via SPI. CAN could be used for various purposes such as communication with other computers and peripherals within the vehicle, but also for dealing with vehicle data via the CAN bus, for example velocity, revolution numbers and error codes. The mentioned interfaces were mapped to remappable pins according to Peripheral Pin Select configuration. See Figure 2.2 for an illustration of the mapping.

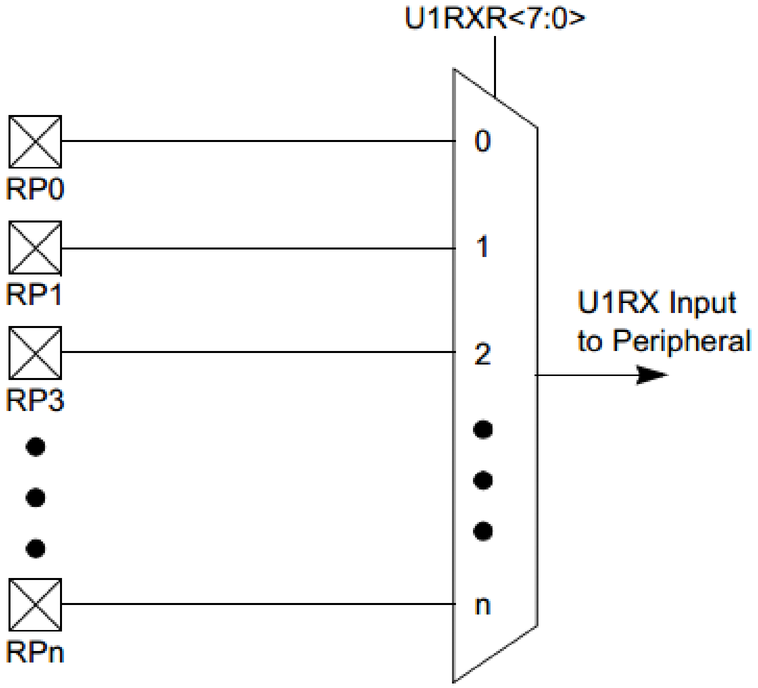


Figure 2.2: Example of remappable input for UART1 Receive [8].

2.2.3 UART

The Universal Asynchronous Receiver Transmitter (UART) is *full-duplex*, asynchronous communication channel[9] that is used for communication with peripheral devices [8]. Various communication protocols are supported by the UART, e.g. RS-232 is used for communication with the modem. Figure 2.3 demonstrates a simple overview over the communication between the modem and the PIC, using UART.

[9] In asynchronous communication, a start signal is sent before each block of data and a stop signal denotes the end of data.

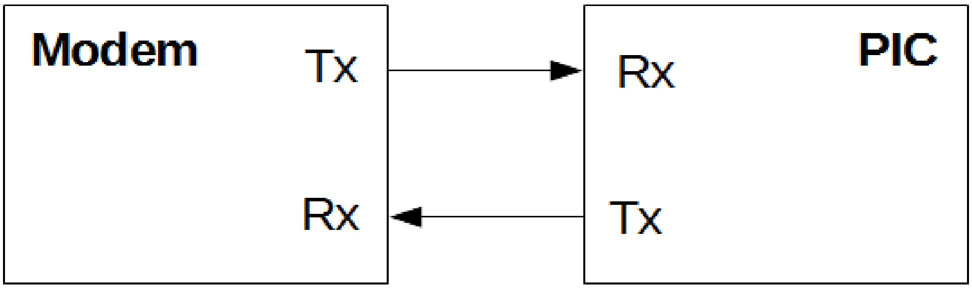


Figure 2.3: Modem UART communication.

2.2.4 SPI

Serial Peripheral Interface is a synchronous serial data link[10], that operates in full duplex mode. SPI is useful for communication with other peripheral or microcontroller devices. The peripherals may for example be EEPROMs, display drivers, sensors and SD cards [8].

2.2.5 CAN

The dsPIC33E supports Enhanced Controller Area Network (ECAN), that implements the CAN specification 2.0B [8]. This implementation is used mainly in industrial and automotive applications. Benefits of this asynchronous serial data communication protocol include reliable communications in an electrically noisy environment. See Figure 2.4 for an illustration of a CAN bus network.

2.2.6 Digital and Analog I/O

Contrary to analog transducers sensing continuous variables such as temperature and pressure, many transducers provide an output of two different states; *high* or *low*. This could for example be useful if the temperature gets too high or the pressure reaches below a level. The digital and analog I/O are connected to the MX unit via a *12-pin connector*, as exhibited in Appendix C.

Due to the *binary language* of computers, digital (discrete) signals of inputs and outputs are much easier for the microprocessor to deal with in contrast to analog signals where an *analog-to-digital converter* is needed in order to be able to interpret the signals.

Digital I/O

Digital Inputs allow the microcontroller to detect logical states while digital outputs allow output of logical states. A digital input compares the voltage level to a certain *threshold*. If the voltage is above the threshold, the computer will interpret the digital input as '1' or high. On the other hand, if the voltage is below the threshold, the digital

[10] In synchronous serial communication, data is sent in a continuous stream at constant rate. A clock is required and the transmitting and receiving devices have to be synchronized, using the same rate.

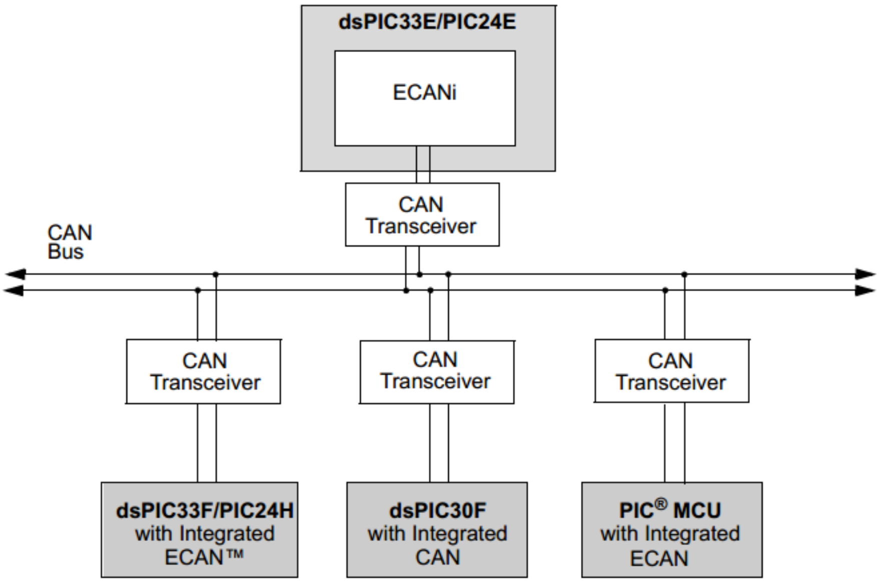

dsPIC33E/PIC24E
ECANi
CAN Transceiver
CAN Bus
CAN Transceiver
CAN Transceiver
CAN Transceiver
dsPIC33F/PIC24H with Integrated ECAN™
dsPIC30F with Integrated CAN
PIC® MCU with Integrated ECAN

Figure 2.4: Example of a CAN bus network [9].

input is set as '0' or low. When it comes to digital outputs, one is able to control the voltage of the pin. Assigning a high value to a pin will produce a certain voltage to it, while setting it to low implies connecting it to zero volt ground value.

Analog I/O

As computers only speak binary language consisting of ones and zeroes, interpreting analog signals is a bit more complicated than digital such. However, manufacturing processes and natural phenomena tend to vary gradually over time, rather than alternating between two distinct states. Hence, such analog signals have to be translated into a digital representation. The smooth continuous values must be converted to discrete numbers, which is done by the Analog-to-digital converter. Essentially, the conversation involves *quantization*[11] of the input, implying that a small amount of errors unavoidably will be introduced. The input is then sampled and the result is a sequence of digital values. Having a sufficiently high sample rate is important in order to avoid errors such as *aliasing*. The PIC processor features a built-in A/D converter able to measure the input voltage on any analog input pins. More information about ADC can be found in the ADC section of the DSPIC33E datasheet [10].

[11] Quantization is mainly applied in mathematics and digital signal processing. It involves a process of mapping a large set of values into a smaller set.

2.2.7 Baud rate

Baud rates are used to measure the transmission speed in *asynchronous communications*. Units sharing a bus must agree on the same speed of information, i.e. symbols per second. Otherwise communication will become jittery, or more likely, completely dysfunctional. Typically, a device will function either with a set baud rate or be configurable within a certain range. To exemplify, the outgoing debug port has been used with a configured rate of 115200 baud, while the incoming data from the GPS UART is read at a rate of 9600 baud.

2.2.8 Interrupts

Several functions in the code body of the microcontroller are dependent on *interrupts*. Hereafter follow descriptions of a few such.

Timed interrupts

There is a timed interrupt function which is called every 10ms. This serves to increment an internal clock value used in a number of locations, and cascades onto a 100ms subroutine every tenth iteration, which in its turn cascades onto a 1000ms subroutine. These subroutines periodically toggle LEDs and instruct the main program flow to perform certain peripheral data acquisition tasks. If there are subscriptions to the gathered data, outgoing TCP packets may be queued here. Clock cycle heavy tasks such as actually sending TCP packets are not performed within timed interrupt routines.

UART interrupts

The UARTs generate interrupts upon transmitting and receiving data [8]. These can be configured to trigger in a variety of ways by setting the corresponding bits in the UxSTA (UART x Status and Control register) registers accordingly, see Figure 2.5.

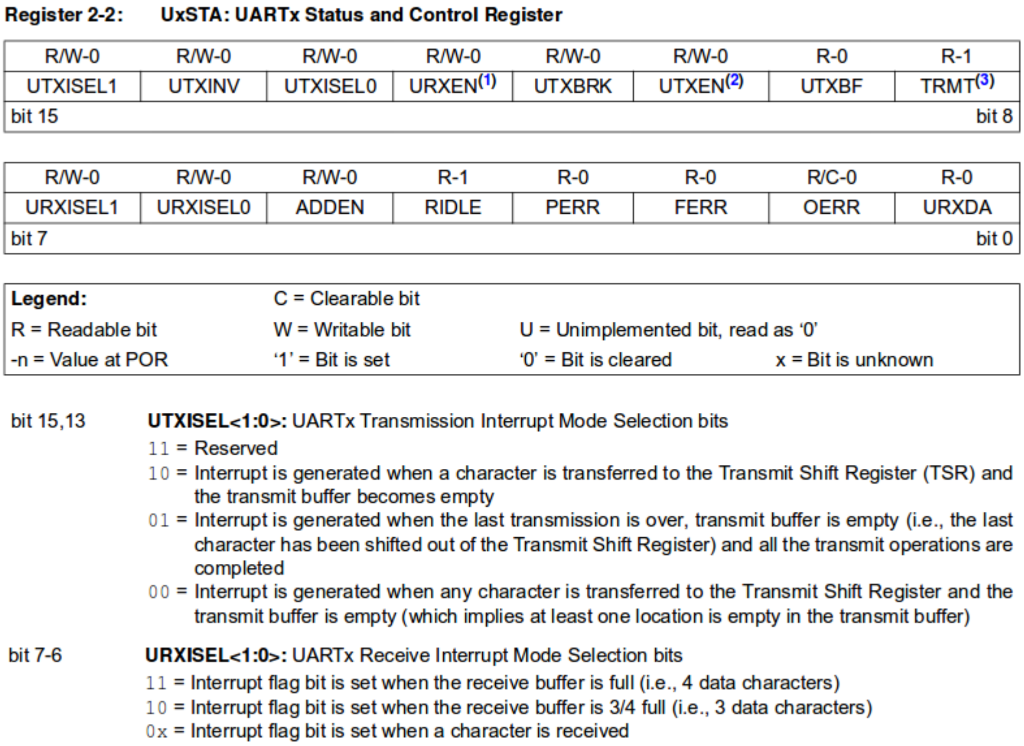
While *polling*[12] for UART data is possible, facilitating interrupts and performing reads within interrupt handlers helps to simplify the code somewhat, and per extension, improve performance.

Trap interrupts

A trap interrupt is triggered as the cause of an exception or fault such as a division by zero, invalid memory address access attempt or stack related issues. In such an event, a specific interrupt bit is set and the corresponding *trap handler* is called. An example of a trap handler is illustrated in Figure 2.6. Notice that the last line of the function runs an endless loop, causing the watchdog[13] to kick in, which results in a reboot of the device after a certain amount of time.

[12] Polling means continuously checking if the device is ready, at which point it is accessed. Often used synonymously with busy-waiting.

[13] A watchdog is a built-in feature running in parallel with other code that may react to special circumstances.



Register 2-2: UxSTA: UARTx Status and Control Register

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R-0	R-1
UTXISEL1	UTXINV	UTXISEL0	URXEN(1)	UTXBRK	UTXEN(2)	UTXBF	TRMT(3)
bit 15							bit 8

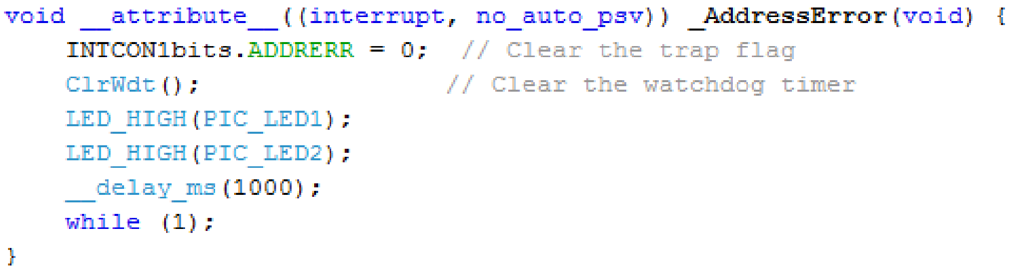
R/W-0	R/W-0	R/W-0	R-1	R-0	R-0	R/C-0	R-0
URXISEL1	URXISEL0	ADDEN	RIDLE	PERR	FERR	OERR	URXDA
bit 7							bit 0

Legend:	C = Clearable bit		
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as ‘0’	
-n = Value at POR	‘1’ = Bit is set	‘0’ = Bit is cleared	x = Bit is unknown

bit 15,13 **UTXISEL<1:0>:** UARTx Transmission Interrupt Mode Selection bits
11 = Reserved
10 = Interrupt is generated when a character is transferred to the Transmit Shift Register (TSR) and the transmit buffer becomes empty
01 = Interrupt is generated when the last transmission is over, transmit buffer is empty (i.e., the last character has been shifted out of the Transmit Shift Register) and all the transmit operations are completed
00 = Interrupt is generated when any character is transferred to the Transmit Shift Register and the transmit buffer is empty (which implies at least one location is empty in the transmit buffer)

bit 7-6 **URXISEL<1:0>:** UARTx Receive Interrupt Mode Selection bits
11 = Interrupt flag bit is set when the receive buffer is full (i.e., 4 data characters)
10 = Interrupt flag bit is set when the receive buffer is 3/4 full (i.e., 3 data characters)
0x = Interrupt flag bit is set when a character is received

Figure 2.5: UARTx Status and Control Register [8].



```
void __attribute__((interrupt, no_auto_psv)) _AddressError(void) {
    INTCON1bits.ADDRERR = 0;  // Clear the trap flag
    ClrWdt();                // Clear the watchdog timer
    LED_HIGH(PIC_LED1);
    LED_HIGH(PIC_LED2);
    __delay_ms(1000);
    while (1);
}
```

Figure 2.6: Interrupt handler of address error.

2.2.9 Microcontroller upgrade

Throughout the project, observations were made about the initial microcontroller being inadequate for the new implementation. In particular, stack errors were encountered as a consequence of an insufficient amount of RAM. There was a need for a microcontroller with larger amounts of memory, and for this we consulted our workplace supervisor. We found the DSPIC33EP256MU806 to be a fitting match given the fairly similar pin setup in combination with superior RAM and Flash memory. Obviously, upgrading to a superior microcontroller comes with a price. See section 4.1 for an elaboration regarding

	PIC24HJ128GP506A	**dsPIC33EP256MU806**
Architecture	16-bit	16-bit
CPU Speed (MIPS)	40	70
Memory Type	Flash	Flash
Program Memory (kB)	128	256
RAM Bytes	8192	28,672
Internal Oscillator	7.37MHz, 32.768 kHz	7.37MHz, 32.768 kHz
I/O Pins	53	51
Pin Count	64	64
Communication Peripherals	2 UART, 2 SPI, 2 I2C	4 UART, 4 SPI, 2 I2C
CAN	1	2

Table 2.2: specification comparison between PIC24HJ128GP506A and dsPIC33EP256MU806.

costs of components.

Upgrading the microcontroller implied effort in terms of porting the old code. The I/O pins had to be remapped and the communication interfaces such as UART had to be set up accordingly. The boot sector, oscillator and baud rates had to be checked over. Furthermore, the TRIS and ANSEL bits needed to be re-configured. Port pins functioning as analog inputs must have their corresponding ANSEL and TRIS bits set. In order to use port pins for digital I/O functionality, the corresponding ANSEL bit has to be cleared, according to the datasheet of DSPIC33EP256MU806. See Table 2.2 for a specification comparison between the two microcontrollers.

2.3 PIC-communication

The PIC controller can communicate through different channels. The internal communication can preferably be done via UART and SPI as well as programmable I/O pins. The PIC is, for example, communicating with the modem via a UART port. This UART interface is also the PIC's primary channel to outside communications via the modem's capabilities.

The TC65i modem has an embedded *TCP/IP stack* that is driven by AT commands and enables the host application to easily access the Internet. The advantage of this solution is that the system engineers do not have to implement their own TCP/IP and PPP stacks. The modem has support for usage as a TCP socket client/server, UDP client, FTP client, HTTP client, SMTP client and POP3 client.

There are basically two strategies for using Internet Services AT commands, URC mode or polling mode.

In URC mode the progress of an Internet session is URC driven. This means that URCs notify whether data can be sent or received, when the transfer is finished or when the service can be closed. If an error occurs it will be reported as well. The benefits of this method are that the PIC does not need to continually poll the service until a result has appeared.

Polling mode, on the other hand, disables the URCs related to Internet Services. The responsibility of checking status information and errors lies on the host. The host has to send a sequence of commands periodically in order to poll for newly arrived information. The benefit of this approach is to get better control of what is happening. One may for instance assume that there is no need to check for data or errors more often than e.g. a few times per minute during times when no data is sent from the PIC, and hence can plan to do other tasks during this time. This also reduces workload when it comes to specifying possible URCs and the PIC's responses to them.

Other ways of communication include calling or sending SMS with the modem as receiver.

2.4 Setting up TCP/IP connection

Initially one has to decide which mode to use, URC or polling. The next step is to create a GPRS *connection profile* with the AT-command `at^sics`. The command takes various parameters to set e.g. the type of connection. When a connection profile has been set, proceed to to set up a *service profile* using the command `at^siss`. This profile is based on one of the previously defined connection profiles. Once the profiles are set, a connection can be established with `at^siso`. In URC mode, the `^SISW` and `^SISR` indicate that the service is ready to receive or send data. Afterwards the data can be read or written with ˆsisw or atˆsisr. In polling mode one may have to run `at^sisw` (receive) or `at^sisr` (send) repeated times until a confirmation message has been received. A flow chart is exhibited in Figure 2.7, where parts of the of the procedure is demonstrated.

During a TCP connection, the information retrieved from SISR and SISW commands is useful for tracking the connection's status. They return the number of bytes that can be read or written, respectively - and a return value of -2 indicates that the connection has been terminated in one way or another. Continuous reads and writes thus provide a solid basis for real-time connection status information.

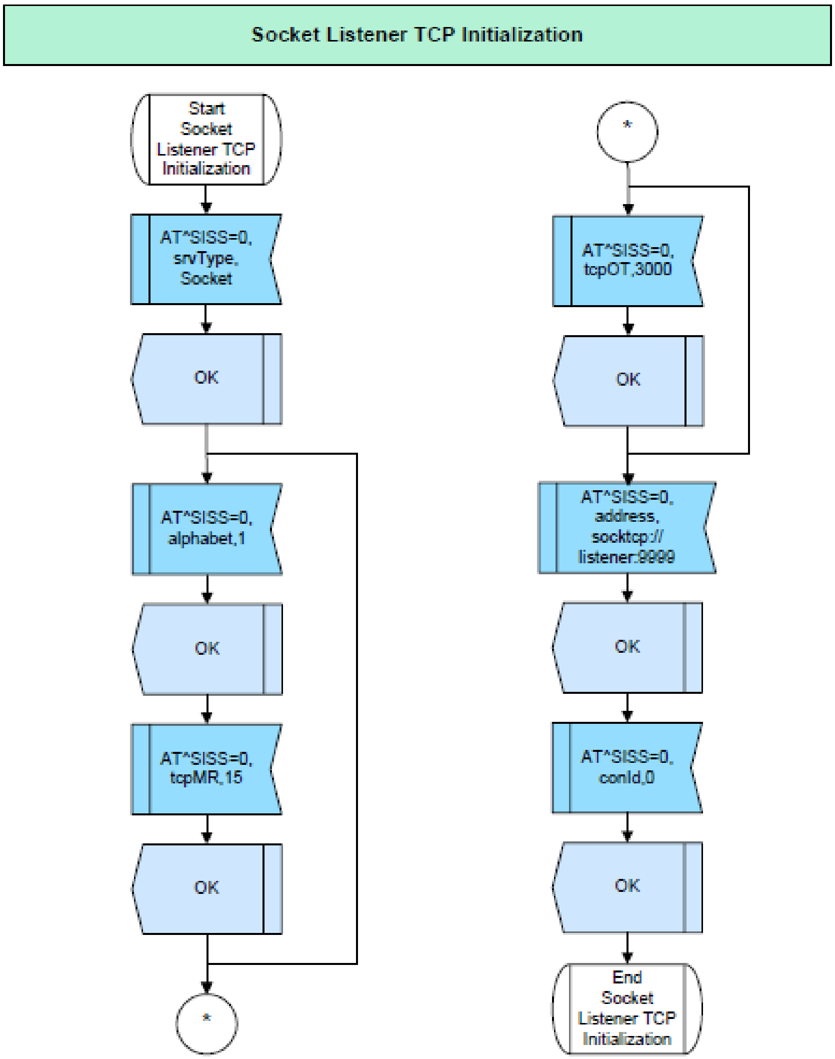

Socket Listener TCP Initialization
Start Socket Listener TCP Initialization
AT^SISS=0, srvType, Socket
OK
AT^SISS=0, alphabet,1
OK
AT^SISS=0, tcpMR,15
OK
*
*
AT^SISS=0, tcpOT,3000
OK
AT^SISS=0, address, socktcp:// listener:9999
OK
AT^SISS=0, conId,0
OK
End Socket Listener TCP Initialization

Figure 2.7: Socket Listener TCP Initialization [11].

2.5 The MX protocol

2.5.1 Protocol overview

The protocol that has been used by default in the communication between MX units and other endpoints such as servers is a proprietary protocol called the MX protocol.

All in all, the premises are quite simple. The protocol includes a short fixed size header with a length indicator, see Table 2.3, and a variable size data payload.

A MX unit receiving packets will then be able to respond properly according to the given parameters. Available commands are:

- **Read** - Perform a single read of the requested information.
- **Write** - Perform a single write to the specified target.
- **Function call** - Call the specified function.
- **Subscribe** - Inform that you want unsolicited information of this type to be sent from now on.
- **Unsubscribe** - Stop subscribing to an event type.
- **Event** - The unsolicited packet type subscriptions return.
- **Error** - Informs about failed commands.

It should be noted that all of these, except Event and Error, also have a corresponding Reply type that is used to return the status of each operation. For example, performing a successful `CMD_WRITE` will return a `CMD_WRITE_REPLY` packet.

Name	Length (bytes)	Description
LOM	2	Length of message including complete header and body. Big-endian format is used.
FLAGS/SEQ	1	FLAGS (4 high bits) are reserved. SEQ (4 low bits) represent an incremented sequence number, wrapping back to 0 after 15.
SRC	1	Source of packet.
DST	1	Destination of packet.
CRC	1	A checksum that is calculated by adding all bytes in the message (including LOM, excluding CRC). This sum, modulo 256, is the CRC. A sending node generates the CRC which is then verified by the receiver.

Table 2.3: MX protocol header description.

The message bodies vary for each command, and may stretch from zero bytes to the maximum packet size allowed by the implementation. The default value is 256 bytes but the two byte LOM field allows for a size up to 64 kilobytes, provided that the TCP and AT layers are also configured to handle this. In practice, with the RAM of the PIC limited to a few ten kilobytes to be shared by all of the code, the actual packet size limit lands in the order of a few kilobytes. On the other hand, very few MX commands ever require a message size of more than 256 bytes. An example of a command requiring quite some data is the `Micro Code Over The Air` command - but this is handled by splitting up the data into multiple individual MX packets. Table 2.4 displays a list of all of the possible endpoints commands may be received from or sent to.

Do note that while many endpoints are defined, not all actions are applicable on all destinations - one may for example not perform a write to the CAN module, or subscribe to PIC information.

Function name	**Identifier**	**Description**	**Comment**
FUNC_APPLICATION	0x01	TC65i application	DEPRECATED
FUNC_PIC	0x02	PIC controller	Read, write
FUNC_CAN	0x03	CAN interface	Read, (un)subscribe, function
FUNC_GPIO	0x04	Input, output, temperature reading	Read, write, (un)subscribe
FUNC_ANALOG	0x05	Input voltage and PB battery voltage	Read, (un)subscribe
FUNC_GPS	0x06	GPS interface	Read, (un)subscribe
FUNC_MEMORY	0x07	SD-card	Read, write, function
FUNC_TILT	0x08	Accelerometer	Read, (un)subscribe
FUNC_AT	0x09	AT communication	Communication between the PIC and the modem.
	0x0A-0xFF	Reserved	

Table 2.4: MX protocol endpoints.

2.5.2 Projected changes

Previously, with the application residing in the modem, MX packets were necessary to propagate data from the PIC and peripherals to the application in the modem. With the new architecture, the data can instead be directly delivered to the application in the microcontroller. Thus, the `FUNC_APPLICATION` endpoint becomes obsolete. On the other hand, as the external server gains access to commands and receives various types of data from the MX unit, it might be reasonable to make use of this and add a `FUNC_EXTERNAL` endpoint to denote that type of communication.

2.6 System architecture

A top level view of an architecture featuring a server is displayed in Figure 2.8. A server will typically be preconfigured to send subscriptions to the relevant types of data flows, likely at least GPS data. The MX units in vehicles gather the data and send it to the server. Users may then view the data in a frontend. A PHP frontend plotting routes based on GPS data from the old MX-3 units using the Google Maps API[14] is displayed in Figure 2.9.

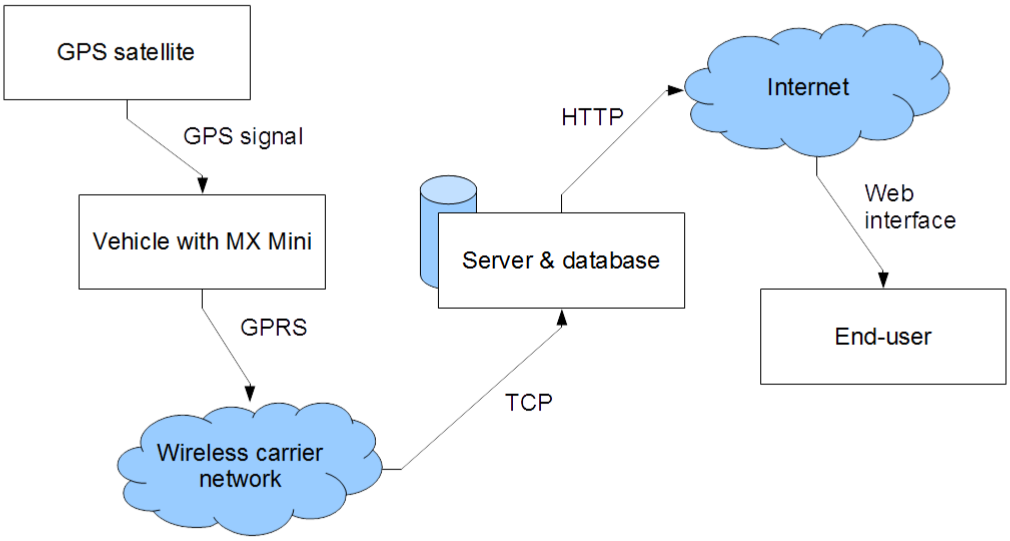
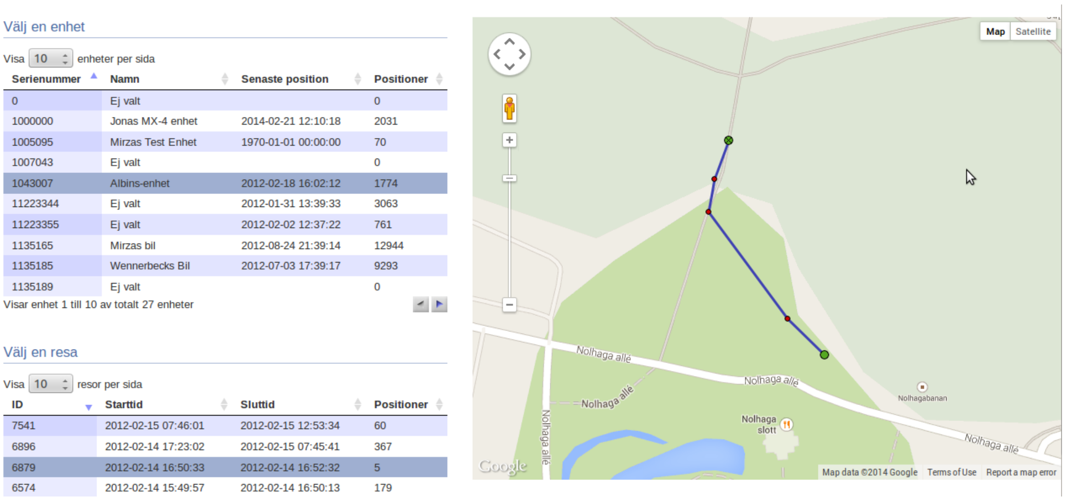


Figure 2.8: Information flow chart.

2.6.1 The test server

It makes sense to have a multifaceted server to display to a customer rather than a simplistic one. To the list of goals to be met belong sufficient performance and robustness

[14] Google provides an interface where coordinates may be graphically displayed on a map view.



Välj en enhet

Visa 10 enheter per sida

Serienummer	Namn	Senaste position	Positioner
0	Ej valt		0
1000000	Jonas MX-4 enhet	2014-02-21 12:10:18	2031
1005095	Mirzas Test Enhet	1970-01-01 00:00:00	70
1007043	Ej valt		0
1043007	Albins-enhet	2012-02-18 16:02:12	1774
11223344	Ej valt	2012-01-31 13:39:33	3063
11223355	Ej valt	2012-02-02 12:37:22	761
1135165	Mirzas bil	2012-08-24 21:39:14	12944
1135185	Wennerbecks Bil	2012-07-03 17:39:17	9293
1135189	Ej valt		0

Visar enhet 1 till 10 av totalt 27 enheter

Välj en resa

Visa 10 resor per sida

ID	Starttid	Sluttid	Positioner
7541	2012-02-15 07:46:01	2012-02-15 12:53:34	60
6896	2012-02-14 17:23:02	2012-02-15 07:45:41	367
6879	2012-02-14 16:50:33	2012-02-14 16:52:32	5
6574	2012-02-14 15:49:57	2012-02-14 16:50:13	179

Figure 2.9: A trip plotted on a PHP frontend.

to handle thousands of concurrent connections proficiently as well as demonstrating functionality of the currently existing features in the MX API.

While the protocol to be used for the interaction on top of the TCP/IP stack will vary from customer to customer, it is likely that many will choose the MX protocol or something closely related. The test server is shipped with a default implementation of the MX protocol.

The choice of programming language for the server included a number of different possibilities. As it is expected that customers' needs will vary, it is out of scope for this project to create a one-size-fits-all server. Rather, heed was be taken to portability, scalability and ease of setting it up. Some important differences between a select few language possibilities we chose from are as follows:

- **C:** Efficient and eases up MX protocol usage since the counterpart in the MX Mini unit is already implemented in C.
- **Java:** Highly portable with easily modifiable functionality. **(Our choice)**
- **Python:** Easy to get working fast on a small scale.

Due to Java's portability and our familiarity with socket programming in the language, this was our language of choice. It should be noted that while using C would have freed up resources from having to implement and maintain the MX protocol in a separate language, it might be appreciated by some customers to have multiple versions readily available. It is not infeasible to imagine releasing several versions in the future from which a customer may choose a version in their preferred language.

Furthermore, a database shall be used in connection with the server. This eases up extensive testing of units both on the customer and the developer side, and also enables for connecting a PHP frontend where results (e.g. trips) can neatly be displayed in a browser. Presently, the environment is setup to use PostgreSQL[15]. Java provides a driver for easy interfacing with PostgreSQL database handling.

2.6.2 Design overview

With the customer application no longer being handled by the modem, planning the code architecture inevitably becomes more challenging. Not only must the PIC be able to handle the built-in customer application and external (TCP) commands, the bus through which AT communication with the modem is handled is the very same we have to use to read in TCP packets to the PIC.

The program flow can roughly be divided into three parts: The main loop, the customer application loop, and interrupt routines. No true parallelism is used, so in practice the customer application code is explicitly called within every iteration of the main loop. An interrupt function is entered every 10ms, and cascades to 100ms and 1000ms functions respectively. While placing heavy functions within the timed interrupt routines is a poor idea due to possible program flow obstruction, these perform well for periodical lightweight data reads from peripherals and setting flags for timed events to be handled within the main flow.

Customer application

A user-friendly interface is provided, making it straightforward for the customer to specify what is done in the customer application. The interface provides features such as server port and IP address configuration, periodic task setup, peripheral data subscriptions and enabling or disabling the GPRS functionality.

AT parsing

Parsing the AT communication is essential to make the system function as a whole. The parser interprets incoming messages, forwards them to buffers read by the user application and provides an interface for sending AT commands. The send commands can be *blocking* or *non-blocking*[16], and nonblocking such may include a pointer to a callback function to handle the expected data. To exemplify, it may be prudent to send a SISI AT command with a callback function that then parses out and handles the presently interesting parts of connection status information. See Figure 2.10 for a flow chart of how the AT parsing is done.

The parser presents to the outside a status that needs to be checked before AT commands are sent, and that may be used to view the status of the last request if it finished. `at_check_status()` returning `AT_STATUS_READY` implies that data can be sent, `AT_STATUS_SENDING` or `AT_STATUS_PENDING` imply commands currently being processed. Reading `AT_STATUS_OK` or `AT_STATUS_ERR` reveals that the last command has finished

[15] PostgreSQL is a widely used open source database.

[16] A blocking request will halt the program flow until the request is served, whereas non-blocking requests may be handled asynchronously.

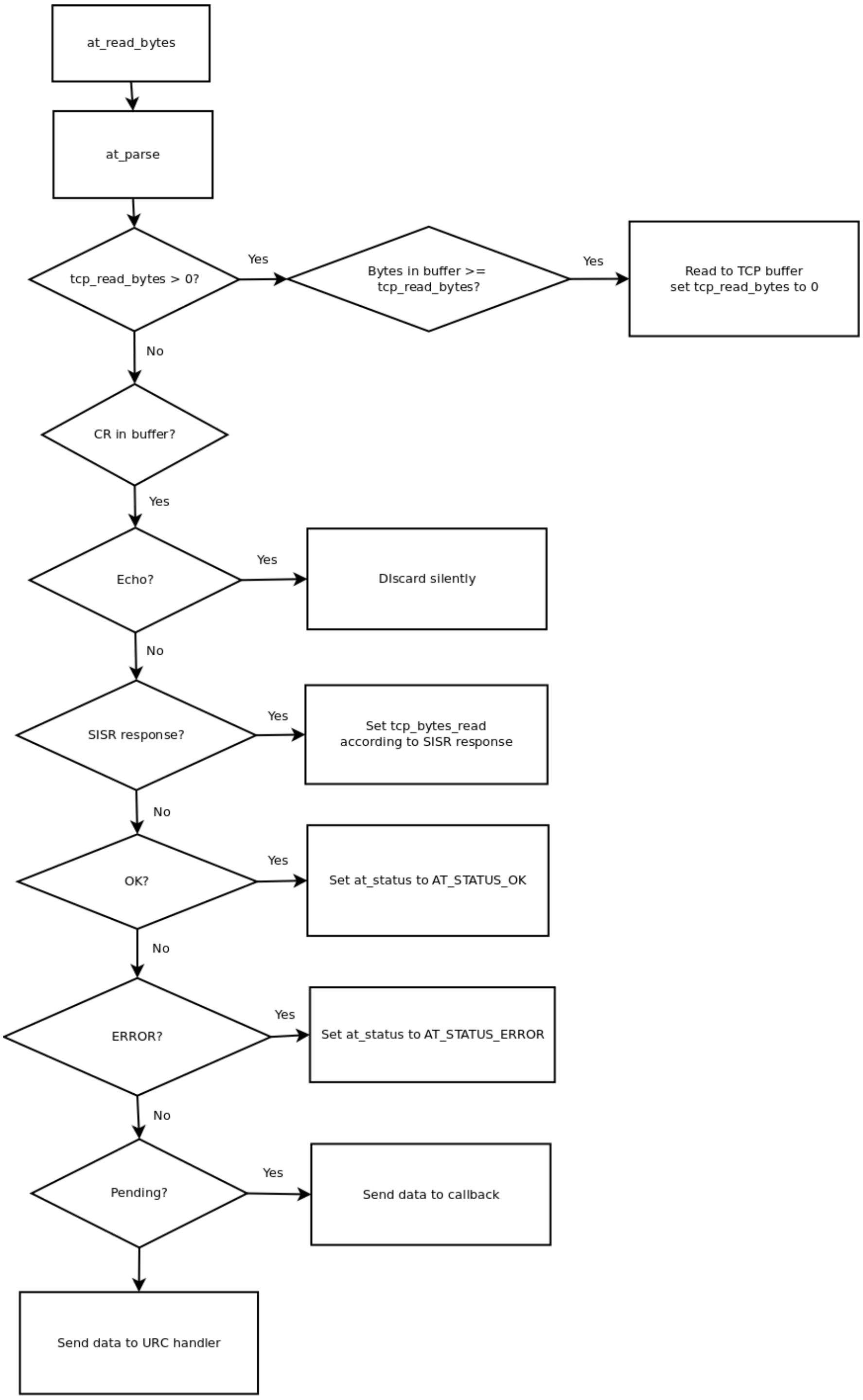


Figure 2.10: AT parser flow chart.

executing, and the status read also resets the flag to `AT_STATUS_READY`. Thus, continuous checking of an `if (at_check_status() == AT_STATUS_OK)` will silently discard `OK` or `ERROR`, which may be preferable in some cases.

A function that is continuously called from the main loop reads available bytes, if any, from the modem bus and places them into a designated buffer. Upon reaching a carriage return indicating the end of a single AT line or having read an explicitly stated amount of TCP bytes, the data is parsed. TCP data is sent directly to the application's protocol handler (using the MX protocol by default), while AT lines have a number of possible cases:

Echoes from sent AT commands are silently discarded. `OK` or `ERROR` indicate that the command is finished, and change the status of the parser to `AT_STATUS_OK` or `AT_STATUS_ERR`. If there is a pending command and a callback function was given, send all data in between the echo and `OK/ERROR` to the callback. If data is received with no command presently pending, it is treated as an URC and sent to the customer application's *URC handler*.

Evidently, not all commands may behave as expected. The send commands must thus always include a timeout value. A command is treated as finished either upon receiving an OK or ERROR from the modem, or if its timer runs out. Furthermore, while data is continuously read from the modem, sending AT commands necessitates that the application first makes sure the parser indicates it is in an `AT_STATUS_READY` state.

Callbacks

The basic premise is that supporting all possible cases of responses within the AT parser would be infeasible due to the multitudes of possible such. It may even be impossible due to dynamic responses, and it is bad practice to assume that only a subset of functions will be used. Because of this, it makes sense to instead make the function call dynamic. A caller of `at_send_command_callback` can include any correctly typed function to be called later when data perceived to belong to the sent AT command is received.

Protocol handler

Besides having a functional TCP layer, there needs to be a specified protocol through which the application may communicate with external units. Presently, the system is based on the aforementioned MX protocol. This protocol, however, stands separated from the other code. The handler works on top of the tcpsock functionality using its interface to send data as arbitrary byte sequences, abstracting the protocol from the underlying functionality. Thus replacing the protocol would be a straightforward process - merely making sure the protocol handler can correctly parse out information from the raw data the TCP layer hands to it and react to it accordingly. Of course, the correspondent needs to be using the same protocol. There is no inherent support for using multiple protocols simultaneously, but this could be achieved either within a large, dynamic protocol handler or a 'routing' function of sorts, identifying protocol types and sending data to the correct individual protocol handlers.

NMEA protocol

The GPS receiver is one of the most central components in the MX module. NMEA (National Marine Electronics Association) is the protocol which the GPS communicates over [12]. Every message consists of a maximum of 80 characters, not including line terminators. The message begins with a '$' and ends with a line feed. The data is contained in a single line and every data item is separated by commas. The amount of data items may vary in each message, e.g. if data is unavailable the corresponding field remains blank. Hence, one should determine the field boundaries based on commas rather than character position. The asterisk is followed by a checksum consisting of a two-digit hexadecimal number. The *checksum* is calculated through bitwise exclusive OR operation over the ASCII codes of all characters between the dollar ($) and asterisk (*) character. The checksum is optional for most data messages although there are cases where it is compulsory, e.g. RMA and RMB.

An example of an NMEA message would look like:

```
$GPGGA,161229.487,3723.2475,N,12158.3416,W,1,07,1.0,9.0,M, , , ,0000*18
```

where GPGGA means Global Positioning System Fix Data. The second field holds a timestamp, third field the latitude, fourth field the longitude and so on.

2.6.3 Internet connectivity

Choice between TCP / UDP

The basic characteristics of TCP and UDP are clear; the *connection-oriented*[17] TCP offers a more reliable delivery but at the cost of larger overhead, especially for data traffic consisting of many small-sized packets. Let us have a look at what the protocols have to offer in the context of this project.

TCP is easy to set up in both directions - with a known endpoint (i.e. a server to which the MX unit sends its data), the unit can simply open a TCP connection to it and have access to full-duplex[18] communication. If connection problems arise, it is easy to notice errors from the TCP layer.

UDP provides simple transportation with considerably less overhead. The MX units' movement may prove to be troublesome when it comes to *dynamic IP addressing*[19], though, as the server communicating with the MX unit cannot ascertain that it knows

[17] In contrast to a connectionless protocol where messages are handled individually, establishes sessions where sequences of messages can be sent.

[18] Data can freely flow in both directions simultaneously. In half-duplex communication, only one direction can be active at a time.

[19] Instead of having one statically assigned IP address, the address may change based on geography and timing.

the current IP address, making only MX to server communication stable. Furthermore, UDP connectivity issues are harder to track as there are no built-in ACKs.

It should be noted that regardless of the choice, the MX protocol will have its own ACK system in place on the application layer. However, the customer is free to use any protocol, and thus UDP use would require such an implementation for the given upper layer protocol. Thus, TCP is our favoured option here.

TCP send queue & Packet acknowledgements

Every time a peripheral wants to send data to the server, e.g. GPS sending position coordinates, an MX packed is created, which is enqueued in a circular buffer. The elements in the queue are fetched periodically. In order to ensure that a packet has arrived at its destination, there is a desire to apply acknowledgements. Note that the implementation uses the TCP protocol when sending and receiving data, where acknowledgements are already used. However, it was desirable to customize various parameters such as timeout and number of resends, hence these features were re-implemented.

When a packet is sent, a timer is started. If an ACK is received before the timer expires, the next enqueued packet will be sent. On the other hand, if no ACK has arrived when the timer ends, a re-transmit will be performed. In the scenario of the maximum number of re-transmissions has been reached, the packet will be discarded, and the next packet in the queue is going to be sent, if there is any. An ACK includes the acked packet's sequence number and a CRC. This combination should be sufficient to uniquely identify a packet.

The downside of this so called ”ping pong approach” may imply low throughput of data. Another approach would be to asynchronously send packets one after another and keep track of incoming ACKs for every packet sent, similar to how TCP is implemented. Such an implementation would provide speed, but this will not be a bottleneck in our system as the volume of the data traffic will be minimal.

2.7 Power saving modes

There are several ways to reduce power consumption on the PIC, as explained in further detail in the datasheet [13].

It is possible to reduce the system clock frequency, statically or dynamically and achieve savings roughly proportional to the frequency decrease. This may, especially within the field of telematics, not be so useful due to the problems this introduces with avoiding peripheral communication errors due to baud rate discrepancies.

Perhaps more interestingly, the PIC can be put to sleep through the use of simple instructions. Using the `PWRSAV #SLEEP_MODE` and `PWRSAV #IDLE_MODE` assembly commands will make the PIC enter Sleep and Idle mode, respectively. **Sleep** mode disables both the system clock source and all peripherals operating on it, effectively making it the lowest power mode available. **Idle** mode disables the PIC's CPU but leaves the system clock source on. In idle mode, all peripheral modules are on by default but can optionally be disabled.

There is also Doze mode, which means reducing the CPU and Flash memory clock rate while retaining the system clock frequency for peripherals. Doze mode is set through manipulating the CLKDIV register:

- **CLKDIV<11>** is the DOZEN, i.e. Doze Enable bit.
- **CLKDIV<14:12>** are also known as the DOZE<2:0> bits, and decide the ratio between the clock speeds. The eight possibilities represent ratios of 1:1 to 1:128.
- **CLKDIV<15>** is the Recover On Interrupt bit. As opposed to default behavior where interrupts do not affect Doze mode, setting this bit and receiving any interrupt makes the CPU clock resume normal operation speed.

There are also the PMD (Peripheral Module Disable) bits, allowing for peripheral modules such as SPIs and UARTs to be deprived of all power and clock input.

2.8 Robustness

2.8.1 Hardware

Obviously, robustness is crucial in such a system. The unit does not operate with any redundant units, which implies that a hardware collapse will most likely result in a failure of service delivery. A remedy would be to connect several identical units via the CAN bus and continuously verify that the master unit is still functioning, if not, the malfunctioning unit will simply be replaced by one of the other units. There are several known solutions that could be applied if availability is of high concern, clustering being the most common. Depending on requirements, there are alternatives including cold standby, warm standby, hot standby and TMR (Triple Modular Redundancy) configurations. In his book, Storey discusses considerations about obtaining high availability in a system [14].

Evidently, having such a redundant setup would raise the price of the entire product. One has to assess the risk of failure, compare it to the price and then determine if it is worth the cost. Consequently, the trade-off between availability and cost has to be taken into account.

2.8.2 Software

While hardware robustness is of some concern, in this context the software is arguably the most vital part. It would not be acceptable for the code to end up stuck in an infinite loop. To prevent that from happening, there is a *watchdog* interrupt handler that performs a reboot if the program gets stuck in a place for too long. Another important task is to keep the TCP connection up and reconnect in case of a lost connection, for example due to a server going down or a mobile unit having poor connectivity in the current location. Furthermore, in order to ensure that a correct packet arrives at its correct destination mechanisms such as acknowledgements, source- and destination fields and error codes are applied. As every packet is sent or received, a CRC is calculated based on the header fields and payload to ensure that the packet is not corrupt. Upon discovering multiple subsequent CRC errors, the software clears the modem's TCP buffer. Were this to happen multiple times, the modem is rebooted following the assumption that an internal error has occurred.

2.8.3 Debugging

When it comes to programming in general, debugging the code has always been a central ongoing process. Debugging is needed in order to help determine the reason of an error.

During the startup period of the project the buzzer and LEDs were utilized as simple debug tools. A LED lighting up could be used to indicate that the code flow was working as expected. However, this approach is hardly verbose enough for effectively debugging large systems.

A more convenient debugger is the built-in debugger in the MPLAB IDE. In order to run the debug mode in MPLAB, small modifications had to be made to the hardware. In practical terms, a capacitor had to be removed from the board. By using the debug mode in MPLAB one could e.g. set *breakpoints*, step into code and check values of variables. Sadly, the downside of the debugger was that it would occasionally provide ambiguous results. Hence, there was a desire for an additional debugging tool.

In the next approach one of the UART TX registers was facilitated, exhibited in Figure 2.11. A RS232 - USB adapter was used to read output from the PIC UART on a PC, see Figure 2.12. By doing this, it was made possible for us to use a terminal emulator supporting serial port connections, e.g. HyperTerminal or Tera Term to output data directly from the microcontroller. A logger was created which made it possible to debug the code using prints - a major aid indeed. For an example of how the debugging prints could look like, see Figure 2.13.

2.9 Peripheral testing

While certain features such as motion triggered sleeps and alarms have not been feasible to test within the confines of our workspace, peripherals have been tested to the extent permitted by the equipment at hand. Tera Term has been used to display values the PIC receives directly, and having passed the first checks the test server was modified to subscribe to the various functions to verify their correctness over time. Below follow accounts of the various tests performed.

2.9.1 GPS

The GPS module was tested by simply checking the incoming values from the pin assigned to GPS to the PIC. While initial tests displayed a valid incoming timestamp, the module reported a zeroed out location. An external antenna was connected to the

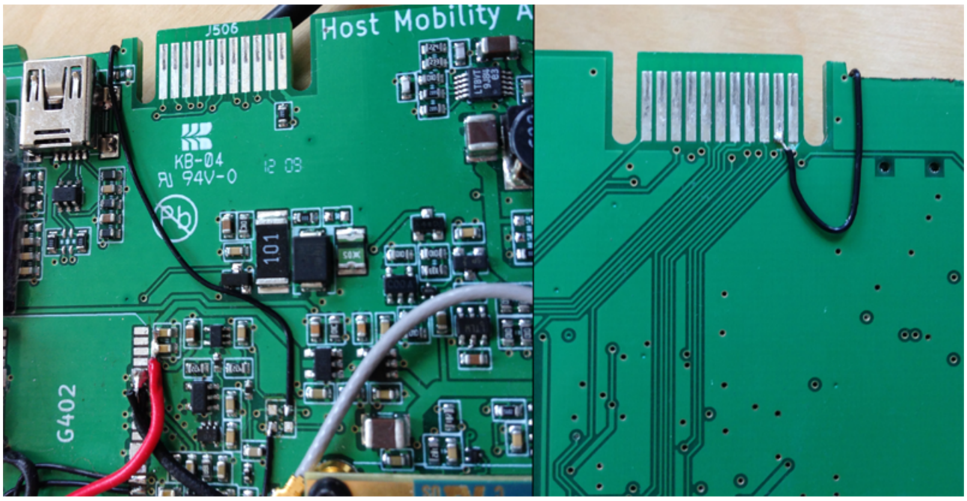


Figure 2.11: UART1 TX (black wire) connected to a RS232 serial cable.

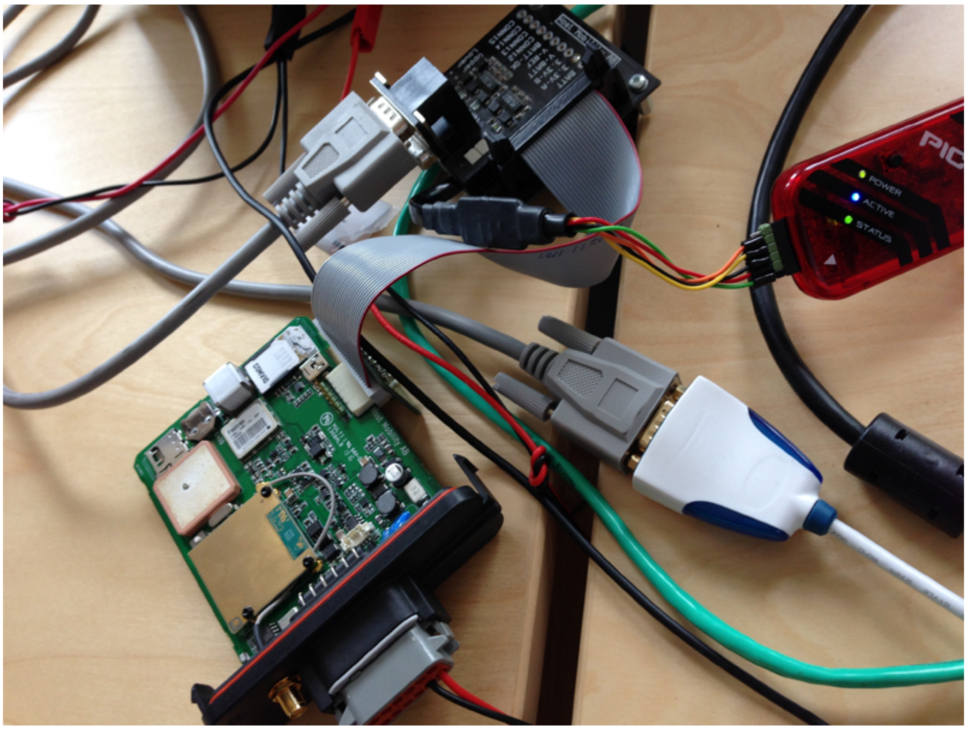
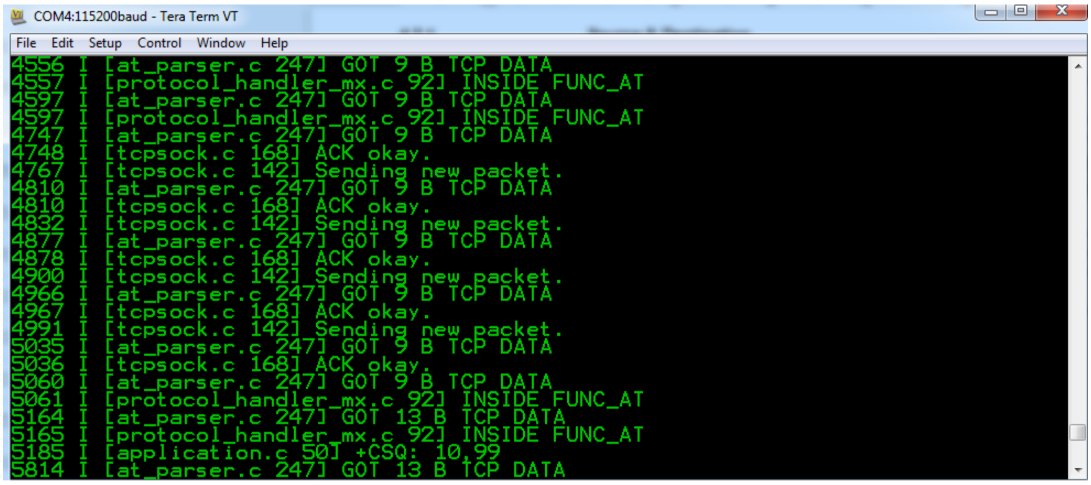


Figure 2.12: A demonstration of the serial-USB adapter debugging setup.


COM4:115200baud - Tera Term VT
File Edit Setup Control Window Help

```
4556 I [at_parser.c 247] GOT 9 B TCP DATA
4557 I [protocol_handler_mx.c 92] INSIDE FUNC_AT
4597 I [at_parser.c 247] GOT 9 B TCP DATA
4597 I [protocol_handler_mx.c 92] INSIDE FUNC_AT
4747 I [at_parser.c 247] GOT 9 B TCP DATA
4748 I [tcpsock.c 168] ACK okay.
4767 I [tcpsock.c 142] Sending new packet.
4810 I [at_parser.c 247] GOT 9 B TCP DATA
4810 I [tcpsock.c 168] ACK okay.
4832 I [tcpsock.c 142] Sending new packet.
4877 I [at_parser.c 247] GOT 9 B TCP DATA
4878 I [tcpsock.c 168] ACK okay.
4900 I [tcpsock.c 142] Sending new packet.
4966 I [at_parser.c 247] GOT 9 B TCP DATA
4967 I [tcpsock.c 168] ACK okay.
4991 I [tcpsock.c 142] Sending new packet.
5035 I [at_parser.c 247] GOT 9 B TCP DATA
5036 I [tcpsock.c 168] ACK okay.
5060 I [at_parser.c 247] GOT 9 B TCP DATA
5061 I [protocol_handler_mx.c 92] INSIDE FUNC_AT
5164 I [at_parser.c 247] GOT 13 B TCP DATA
5165 I [protocol_handler_mx.c 92] INSIDE FUNC_AT
5185 I [application.c 50] +CSQ: 10,99
5814 I [at_parser.c 247] GOT 13 B TCP DATA
```

Figure 2.13: Debugging output from Tera Term.

GPS, whereafter the reported latitude and longitude corresponded to the location of the workplace.

2.9.2 Digital I/O

Digital I/O was tested using a multimeter and an adaptable power supply as aids. In this context, observed values of 5V or higher represent a logical one and lower values a zero. For the input, the adaptable power supply was connected to a digital input pin and set to output a voltage of 20V. During the testing, it was soon discovered that the application would display a steady zero. Software inspection revealed no errors. It was observed through the use of an oscilloscope that the incoming signal was not only distorted but also as low as in the range of millivolts rather than the expected 20V, leading to the discovery of faulty soldering in the MX test unit. Having repaired this, the application consistently reported correct logical input values corresponding to the manually alternately enabled and disabled input voltage.

Output was tested by connecting a multimeter to a digital output pin and a reference ground from the MX unit's power supply. The tests consistently displayed a near-zero voltage from logical zero output and 20V from logical one, implying correct functionality.

2.9.3 Analog I/O

A 100Ω resistor is situated at every analog input. This is where the voltage is measured by the ADC and represents the input current of the analog input port. Furthermore, the components of the board scale down the input voltage, ensuring a value that can be measured by the 12-bit ADC (A/D Converter) within the interval of 0V and the *reference voltage* (V_{ref}). The reference voltage of the MX-3 unit is 2.5V, hence the maximum value that the ADC can convert. The voltage range is divided into 4096 values (i.e. 12 bits) where each value is given by $V_{ref}/4096$. In this particular case, with a voltage reference of 2.5V, each value is 2.5V/4096 = 0.6 mV. This is the resolution of the ADC, meaning the smallest voltage change that can be measured by the ADC. Any input voltage below 0.6mV will be converted to 0. Voltage values between 0.6mV and 1.2mV results in a 1 as output. Effectively, values between 1.2mV and 1.8mV result in a 2 and so on.

Tests of the analog input included voltage measurement of the power source and current measurement of an external adaptable power supply. Additionally, the voltage of the Li-ion battery was measured.

2.10 Security concerns

While care has been taken throughout the project when it comes to maintaining high code quality, fully securing the system from malicious attackers remains outside the scope of this project. Still, even a brief overview reveals multiple interesting *attack vectors* worth mentioning.

2.10.1 Physical access

The MX unit provides multiple auxiliary physical interfaces in forms of UARTs, SPIs and USB. While their usage is highly dependent on the customer application handling these respective input sources, an attacker with physical access could potentially be able to mount a stealth attack using any of these.

2.10.2 TCP/IP

The default implementation uses the TCP/IP stack of the modem as is, without any additional security layer used. The PIC connects to an address set in the custom application code, and keeps communicating using TCP. Security is limited to what GPRS in itself can offer. Furthermore, it cannot be guaranteed that the PIC is speaking with the intended recipient as an attacker may be spoofing her IP address and acting as the endpoint, or performing a *MITM (Man-In-The-Middle) attack*[20]. All of this could potentially be remedied by applying a *public key cryptography scheme*[21].

2.10.3 SMS/RING

Customer applications can choose to offer functionality through calling the modem, or sending an SMS to it. Presently, no checks are in place for validating the initiator of such communications. It might be prudent to ascertain that only specific, known numbers are able to perform actions in these ways. Looking into the possibilities of spoofing these types of communication remains outside the scope of this project.

[20] When an adversary is able to intercept and potentially modify information flowing between legitimate participants.

[21] A commonly used, strong form of cryptography that does not require secret keys to be shared beforehand.

2.10.4 Implications

Assuming an attacker gets through and manages to send malformed TCP data the PIC accepts, what is the potential impact? The first thing to watch out for is presuming data is valid. Not doing this can easily result in unpredictable behavior. It has been our aim to always validate data and discard malformed such - not only is this necessary from a security point of view, but also protects from erroneous input sent by malfunctioning soft- or hardware. Furthermore, bound checking needs to be performed in all places handling data derived from user input. Failure to do this may end up in buffer overflows[22], which have the potential of revealing data or denying service by making the processor crash.

[22] A security flaw where supposedly protected memory locations may be overwritten.

3

Results

3.1 Architecture

The entirety of the custom user application has been removed from the GPRS modem and instead incorporated in the microcontroller. The previously used PIC24 microcontroller has been replaced with a more powerful dsPIC33 variant. While the actual modem in use remains the same, everything is set up for a smooth transfer to a more economically viable alternative. Peripherals in the form of UART modules, digital and analog I/O have been tested and confirmed to work within a controlled, stationary environment.

The new architecture brings with itself a welcome improvement in the form of reduced code size. The modem now carries with it no Java code at all, and the code interfacing to the peripherals that now lies in the microcontroller is smaller in size thanks to its now closer proximity to the hardware and the inherently more compact nature of the C language.

Sleep modes have not been used throughout this project, but we acknowledge their existence in the forms described in section 2.7 and would like to emphasize their importance in the future.

3.2 Cost efficiency

The two significant factors when it comes to cost are expectedly the PIC and the modem. The price largely depends on the size of the order, i.e. buying a quantity of one thousand microcontrollers yields a great cut of the price per unit. Therefore, the prices listed in this chapter are based on a standard order size consisting of between 200 and 500 units. Also, the cost of the components will most likely differ over time, hence the listings in this chapter will only reflect the current (2014) prices on the market. When it comes to modems there are several alternatives to choose from, however, in this section only two of them recommended by our workplace supervisor are considered.

Microcontroller	Price per unit	Price difference	
PIC24HJ128506A-I/PT	4.40$	-	-
DSPIC33EP256MU806-I/PT	5.73$	+1.33$	+30.2%

Table 3.1: Microcontroller price comparison.

Modem	Price per unit	Price difference	
TC65	39.78$	-	-
BGS2	20.75$	-19.03$	-47.8%
M72	14.29$	-25.49$	-64.1%

Table 3.2: Modem price comparison.

Referring to Table 3.1, upgrading the microcontroller resulted in an increased cost of 1.33$. The change of modem would, assuming the TC65 is replaced by a BGS2 from the same manufacturer, decrease that particular cost by 19.03$. There are also possibilities to change manufacturer from Cinterion to Quectel, replacing the modem with a M72 and resulting in a decreased cost of 25.49$. See Table 3.2 for modem pricings.

Let us assume that the final product makes use of the superior microcontroller DSPIC33EP256MU806 in conjunction with the cheapest alternative of modem explored, i.e. Quactel M72. These changes would, all in all, sum up to a 24.16$ save per MX unit. With a total cost of around 300$ for a MX-3 unit depending on the exact model, the MX Mini unit would be roughly eight per cent cheaper.

3.3 Physical characteristics

As the MX unit as a whole is delimited by its protective case, the hardware changes we have made will not impact its dimensions.

The TC65i modem weighs in at 7.5 grams, while the two modem alternatives explored both weigh just over 4 grams. The microcontrollers themselves have negligible weight difference, and indeed, weight altogether. Considering the MX unit's total weight of 280 grams, this roughly one percent change from the three grams' difference is rather miniscule.

3.4 Component power consumption

The currently used modem, TC65i [15], consumes 1.5mA in *GPRS sleep mode* while the M72 [16] and BGS2 [17] modems consume 1.2mA respectively. Changing the modem to one of the latter would therefore decrease the power consumption by 20% in sleep mode. Furthermore, with the new master-slave architecture, the issue about synchronizing sleeps and wake-ups between the two processors has vanished. This would imply that the overall power consumption of the unit can be decreased even further. However, due to a limited time frame, no measurements have been done regarding the difference in sleep modes.

The microcontroller DSPIC33EP256MU806 will be run at the same clock frequency as the previously used PIC and will therefore not affect the power consumption.

3.5 Limitations

3.5.1 Testing the equipment

While testing has been performed throughout the development process, it has been quite limited in scope. There is still a need to perform lengthy testing of the system. Not only need the single modules be checked for errors, the system in its entirety must be made certain to perform efficiently and robustly in a real world environment (vehicular use).

3.5.2 Change of modem

At the time of writing, the TC65i modem used in the previous MX-3 model remains in the MX Mini prototype. Alternatives have not been explored in practice due to time limitations. Thus, a change may facilitate unforeseen difficulties. That being said, changing to a modem from some other manufacturer than Cinterion/Gemalto will most likely end up with more work as the set of AT-commands differ between manufacturers.

4

Conclusions

Upon the closure of this project, we have acquired an architecture and code base that promotes vendor independency with regard to the communications GPRS modem. This remedies the issues of hardware choice limitations, and per extension, cost efficiency, since the simplest possible modem can be used according to specific needs. Furthermore, we have concluded that the new dsPIC33 microcontroller is sufficient for simultaneously handling all peripheral I/O, operating the GPRS modem and running a customer application.

Another important task during the project has been to improve the energy efficiency. While a new lightweight modem will consume less energy per se, there are now possibilities to trivialize low energy modes as a result of the new architecture, which opens up for a significant reduction of power consumption. This is preferably done via sleep and idle management, but has been left outside the scope of this project due to limited amount of time.

4.1 Discussion

All in all, our project has touched upon a large number of differing areas related to both software and hardware where we've honed our skills or even transgressed from having no prior experience to actually being able to make qualified decisions. While not focusing as much on GPS and Google Maps as El-Medany [1] and Al-Khadher [3], the inclusion of this aspect within our work (See Chapter 2.6) has given us a great real life example of telematics device usage involving multiple processing steps and subcomponents. It is worthwhile to notice that Zhang's [4] findings about UDP throughput retain their veracity pertaining to applications primarily focusing on network communication, but with our lesser sending frequency and packet sizes typically in the order of a hundred bytes we concluded the slight overhead increase from TCP was more than well offset by the stability and ease of two-way communication offered.

Upon embarking on the project, it was thought that the capability of the PIC microcontroller to handle all of the peripherals, GPRS modem control as well as running a typical user application simultaneously was to be evaluated in a rather late phase of the whole process. Indeed, it was uncertain if the PIC change and code migration would even be handled within the project's scope. It was however to be discovered that as early as a week into the actual work, we could derive from the MPLAB memory usage bars that RAM was almost exhausted. At first, code optimization was considered but projected gains were very limited and a PIC upgrade instead was made very appealing not only by the much larger memory amounts but also the rather insignificant price increase, as evidenced by section 3.2. This lead to quite the change in the time plan as items got switched around - but simultaneously, the early access to extra RAM simplified testing significantly for the remainder of the project and gave leeway to much more available memory for customer applications.

Initially, we had concerns about the distribution of clock cycles between the customer application and the remainder of code. But time would show that no complicated solutions such as interrupt and timer controlled scheduling were needed - the well designed program flow where care is taken to ensure that computations are only performed when necessary (e.g. when flags are set) was sufficient. Early in the development process, we carelessly executed certain heavy tasks in the timed interrupt routines due to convenience. This led to bad overall performance and erroneous timings, giving us an important lesson to carry with us throughout the remainder of the project.

4.2 Future work

All in all, the features present in the unit are just tested in a safe office environment. Releasing the product into production will necessitate meticulous testing in real world circumstances, with mobile vessels facilitating the MX platform with various peripherals and also taking heed to energy consumption in cases where only battery power is used. It would of course be highly desirable to provide a simple interface for all sleep mode management.

The new architecture will be able to support more peripherals, most notably related to the new usages a solely battery-powered approach will support. These will not be outlined nor tested in the scope of this project, but examples of interesting such may be motion detector triggered alarms, automatic emergency call devices in vehicles and assistance buttons for the elderly.

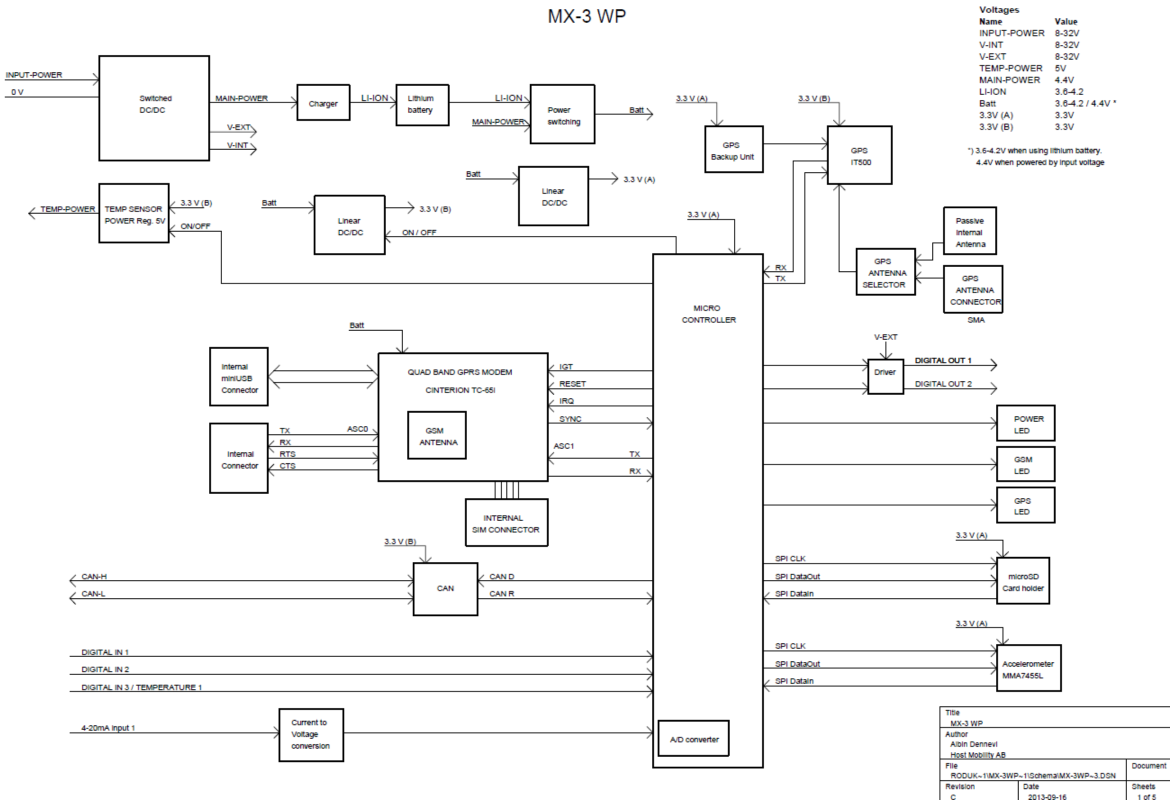
A more than necessary feature for the product is the possibility to patch the software over the air. This feature is crucial before releasing the product on market. If a major bug is discovered to be present in thousands of units on the roads, there will be a need to easily find a way to apply a fix. Doubtless, in such a big system bugs will appear in one way or another. Having the company manually upgrade the software in every unit would be infeasible in terms of workload, time and subsequently cost. A solution could be to install an EEPROM or SD-card and then periodically check if there is a new patch available. The patch will be downloaded into the memory and the device will reboot and apply the patch through the bootloader upon finishing the download.

In addition, the SMS and RING response capabilities have not been explored in practice. Inclusion of these should be straightforward; setting the AT parser to recognize these commands and having the customer application react accordingly, for instance by enabling or disabling power saving mode.

Furthermore, it would be desirable to support the SPI and CAN communication interfaces. For instance, abroad usage of GPRS connections often carries with it high roaming fees. This could be alleviated by having support for memory cards through SPI on which peripheral data could be stored. A GPRS connection would not have to be up all the time but could be used on demand, triggered by for example reaching a predefined quantity of saved data.

A

MX-3 high level schema



Voltages

Name	Value
INPUT-POWER	8-32V
V-INT	8-32V
V-EXT	8-32V
TEMP-POWER	5V
MAIN-POWER	4.4V
LI-ION	3.6-4.2
Batt	3.6-4.2 / 4.4V *
3.3V (A)	3.3V
3.3V (B)	3.3V

*) 3.6-4.2V when using lithium battery.
4.4V when powered by input voltage

Title MX-3 WP		
Author Albin Dennevi Host Mobility AB		
File RODUK~1\MX-3WP~1\Schema\MX-3WP~3.DSN		Document
Revision C	Date 2013-09-16	Sheets 1 of 5

B

Version control system

It is highly recommended to use a version control system when working on larger project with several people involved. Git has been a helpful version control tool throughout the development process.

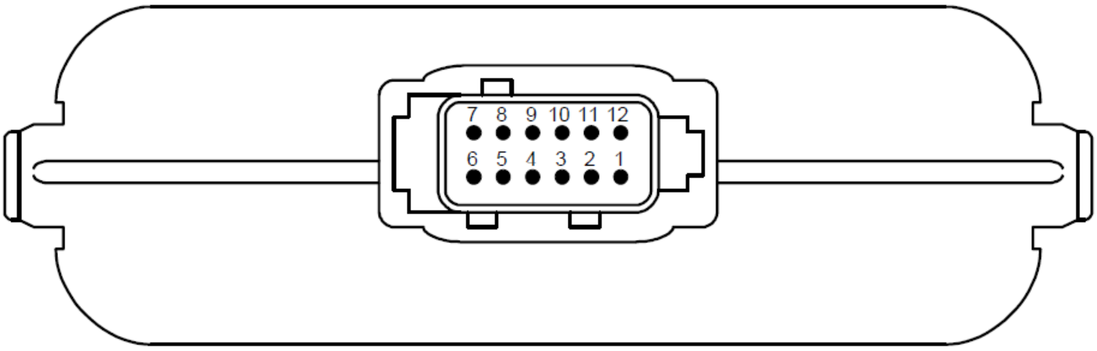
A version control system is a software that is able to track changes that occur to any file in a given project. It has an ability to take “snapshots” of the current state of your files and one will be able to return to that state whenever you wish. A history is also provided which includes the person making the change, a timestamp of when the changes were done and what changes that was made.

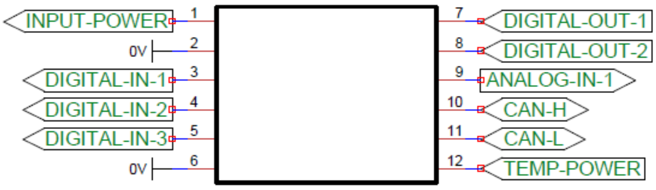
There are many advantages of using a version control system. First, it provides possibilities to create test branches from a certain state. If the changes appear to be successful one may merge the newly created branch with the original one. On the other hand, if the changes ended up with a poor result, it is easy to go back to the original branch with a fully working state. Second, the tool is handy when tracking bugs, knowing what person made a change and for what reason. Another benefit would be that you can work on the project locally, i.e. in a “sandbox”, without interfering with other peoples' work on the same project. Furthermore, you do not have multiple backup files to handle and keep track of.

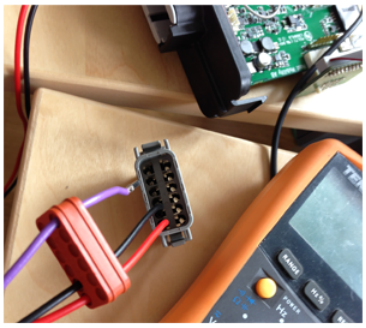
Additionally, by using a version control system, the supervisors at Host Mobility may easily track and monitor progress that has been made. Hence, it provides a natural way of getting feedback of the actual state of the project.

C

12-pin connector schema







References

[1] E.-M. W, A.-O. A, A.-H. R, A.-I. S, A cost effective real-time tracking system prototype using integrated gps/gprs module, 6th International Conference on Wireless and Mobile Communications (ICWMC), 2010 : 20 - 25 Sept. 2010, Valencia, Spain ; proceedings (2010) 521–525.

[2] I. Lita, I. Cioc, D. Visan, A New Approach of Automobile Localization System Using GPS and GSM/GPRS Transmission, ISSE 2006 29th International Spring Seminar on Electronics Technology : nano technologies for electronics packaging : May 10-14, 2006, International Meeting Centre, St. Marienthal, Germany : conference proceedings (2006) 115–119.

[3] M. A. Al-Khedher, Hybrid gps-gsm localization of automobile tracking system, International Journal of Computer Science & Information Technology (IJCSIT) Vol 3, No 6, Dec (2011) 75–85.

[4] Z. P.C, S. Z.K, X. M, Design and implementation of vehicle monitoring system based on gprs, Proceedings of 2005 International Conference on Machine Learning and Cybernetics, Vol.6, Aug (2005) 3574–3578.

[5] Microchip Technology Inc, Mplab x ide user's guide (2011).
URL `http://ww1.microchip.com/downloads/en/DeviceDoc/52027B.pdf`

[6] Microchip Technology Inc, Mplab xc16 c compiler user's guide (2012).
URL `http://ww1.microchip.com/downloads/en/DeviceDoc/50002071C.pdf`

[7] Gemalto, Tc65i at command set, rev. 02.004 (2008).
URL `http://m2m.gemalto.com`

[8] Microchip Technology Inc, dspic33epxxxmu806/810/814 and pic24epxxxgu810/814 data sheet (2009).
URL `http://ww1.microchip.com/downloads/en/DeviceDoc/DS70616C.pdf`

[9] Microchip Technology Inc, Section 21. enhanced controller area network (ecan) (2011).
URL `http://ww1.microchip.com/downloads/en/DeviceDoc/70353C.pdf`

[10] Microchip Technology Inc, Section 16. analog-to-digital converter (adc) (2013).
URL `http://ww1.microchip.com/downloads/en/DeviceDoc/70621c.pdf`

[11] Gemalto, Application developer's guide, rev. 07 (2012).
URL `http://m2m.gemalto.com`

[12] National Marine Electronics Association, Nmea (2014).
URL `http://www.nmea.org`

[13] Microchip Technology Inc, Section 9. watchdog timer and power-saving modes (2011).
URL `http://ww1.microchip.com/downloads/en/DeviceDoc/S9.pdf`

[14] N. Storey, Safety-critical computer systems, Addison-Wesley, Harlow, England Reading, Mass, 1996.

[15] Gemalto, Tc65i/tc65i-x datasheet (2014).
URL `http://m2m.gemalto.com`

[16] Quactel, Quectel m72 dual-band gsm/gprs module (2012).
URL `http://www.quectel.com`

[17] Gemalto, Bgs2 datasheet (2014).
URL `http://m2m.gemalto.com`