Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Computer Engineering and
Networks Laboratory

Master Thesis

Power Aware Communication Platform for an Autonomous Sailboat

by David Frey

March 2009 – October 2009

Supervisors: Andreas Schranzhofer
Dr Clemens Moser

Professor: Prof. Dr. Lothar Thiele

Abstract

Building autonomous sailing boats is a difficult and not yet well understood problem. A group of mechanical engineers of ETH Zurich committed to build such a vessel to participate in Microtransat, a contest to cross the Atlantic ocean.

The communication platform developed in the scope of this thesis not only provides communication services to the existing control system of the boat, it also adds robustness by supervising it and employs a power management system which prolongs the responsiveness of the system in low energy situations.

The platform was assembled from its basic building blocks that include an embedded computer, voltage and temperature measurement, a GPS receiver as well as a GSM and an Iridium modem. The power consumption of all parts was measured and the main energy consumers were connected such that they can be powered off when not used.

The intricacies of high-sea communication and different transmission devices are dealt with by the platform and hidden from the sailing system, which only has to implement a high-level interface over an internal link. Thus it can send status and control messages in both directions and it is reachable in emergency situations.

Also using the internal link, the sailing system is supervised for responsiveness by the communication platform and rebooted, should an error condition occur.

To be prepared for periods on the sea with little energy input from the solar panels, a power management system has been developed that can influence the energy consumption of the communication platform as well as the sailing system. Using a forecast of the expected solar power input an algorithm optimizes the power consumption such that the minimal service level of the boat is maximized. Several different algorithms have been compared in simulation based on statistical and historical data and the most competitive was implemented to run on the boat.

Table Of Contents

1 Introduction

The first chapter will provide information about the context of this project and explain the general problem setting.

1.1 The Microtransat challenge

The Microtransat is a competition designed for completely autonomous sailing boats with a goal of crossing the Atlantic Ocean. With this easily explainable and understandable challenge, it tries to encourage the development of autonomous sailing and navigation technique for the open sea.

The basic rules of the challenge are simple: Every participating vessel must be completely autonomous with respect to navigation and energy supply. Additionally, the only form of propulsion allowed is the wind. The only constraint on the design of the boat is a maximum length of four meters; the type and number of sails as well as the height of the mast and depth of the keel are not restricted (see Section 6.3.3).

The only guidelines for the route across the Atlantic are the starting point off of the southern coast of Ireland and the finish line in the Caribbean. However, given the difference in climate between northern and southern sea routes and the location of the trade winds, there is one obvious choice for the route. The boat first sails south, leaving the adverse weather of the northern hemisphere with its abundant precipitation and minimal sunshine. Then, once the boat hits the trade winds around thirty degrees north, it turns west and crosses the ocean with a tail wind and stable, sunny weather.

Motivations to drive the development of technology for autonomous sailing boats are diverse. In the case of this project it presented a challenging, well defined problem ideal for a university project. It combines challenges in the area of mechanical engineering, computer science and electrical engineering. As there is hardly any existing work in this field, new ground can be covered both in the mechanical design of a new boat as well as in the control system and the sailing and navigation logic.

A good motivation for each competing team is to build the first boat to autonomously cross the Atlantic. Comparable to the first moon landing, it encompasses a technically complex topic in a very simple yet descriptive project that is easily explained to anybody. As such it is also sure to capture the attention of the media, which loves topics with a nimbus of pioneering spirit.

Additionally, other practical uses of the technology developed for this contest have been proposed. [1] One day, small, fully autonomous sailing boats could serve scientific purposes by collecting data in the sea. Unlike scientific buoys, they can collect water samples and measure environmental data at more than one location and deliver their findings back to a home base. For this application, the technology must be extremely reliable and robust lest all data be lost with an irretrievable vessel.

Other, less direct applications could include assisting technology for skippers. Knowledge gained from building completely autonomous sailing boats could be used to build autopilot-like systems for sailing boats that support the yachtsman while on duty or completely control a ship while the skipper is resting. However, boats built for the Microtransat usually use a simplified sailing technique to limit possibilities for mechanical failure. To employ the controlling systems developed for the challenge in existing, possibly much larger sailing ships, they would need to be completely re-engineered.

1.1.1 Avalon, an autonomous sailing boat

1.1.1.1 The team

The SSA project (Students Sail Autonomously) was initiated in Summer 2008 by a group of students studying mechanical engineering at ETH Zürich. Initially, two teams existed, both planning to build a sailing boat, but with different goals. After uniting these two teams, a common goal was found: to build a vessel that was able to participate in Microtransat in addition to being optimal for long distance, autonomous sailing. SSA qualifies as a *Fokusprojekt,* which is a mandatory part in the study plan of mechanical engineering.

Several students of mechanical engeneering wrote their bachelor thesis within the scope of this project [2][3][4][5][6][7].

The SSA team acted independently in terms of organization and financing while always coordinating with the university. The entire project was financed by one major sponsor and several minor sponsors that were all acquired by the team itself. While the main sponsor contributed financially, other companies donated work and material to the project. During the run time of one year, connections to different members of the media could be established. Good media coverage, in the form of newsletter articles and television and radio segments, resulted [8].

The team structure had a very flat hierarchy. While every member had his own specialization and role, no real team leader existed. Decisions were made with the general consent of the people interested and competent in the respective field. This allowed for a very flexible development and facilitated coordination of this thesis with the rest of the project. For every interface between the software developed the scope of this project and the rest of the system, be it hardware or software, the individual to coordinate with was also the person with the competence to know if it could actually be implemented.

On the other hand, every specification or agreement was always work in progress and subject to all kinds of changes. For example, the interface between the communication platform and the control computer changed almost completely from the first version agreed upon by both parties to the final version presented in this paper. A significant requirement of functionality, no longer needed by the navigation algorithm, was dropped while others emerged.

1.1.1.2 The boat

The Avalon sailing boat was newly designed from scratch, focusing on reliability and durability in every step. Fabrication was done partly by external companies, partly in collaboration with other parties, and partly by the team itself.

To reduce the chance of mechanical failure, the boat has as few movable parts as possible. Only three actors in the form of motors are on board, while sensors deliver data for the control system. The boat is equipped with only one sail, rigidly mounted to mast and boom, which are one and the same. Unlike traditional sailing ships, the mast is not one straight pole with the boom loosely attached, but the two are one rigid L-forming shape. This part is then mounted on the boat by an extension of the boom near its center. As such, the whole construction, sail and mast-boom can be rotated using one motor inside the boat. After some problems with the initial design, rotation is finally possible in 360 degrees with no need to limit the number of turns in one direction as all electrical connections to the mast are done via a slip ring.

The remaining two motors control the two rudders of the boat, located symmetrically on both sides in the back of the vessel. The advantages of having two rudders are redundancy and the availability of a ‘breaking mode’ where both rudders can be set at a 90 degree angle to the boat axis, thus holding the boat at the same spot in the water. The sail can then be set parallel to the wind direction, minimizing any drift. This is highly useful during testing if access to the boat is needed while on the water in windy conditions. Also, the boat will go into this mode in case of winds stronger than 7 Beaufort. Above this wind speed, sailing is too dangerous for the boat; using this ‘handbrake’ technique, the boat will wait for the storm to blow over.

Other important factors guaranteeing reliability for a long-distance sailing boat are the measures required against intruding seawater. The boat was developed as a waterproof hull incorporating two parts. The lower part consists of everything that is normally under water, and the only openings in it are the shafts of the two rudders and a nozzle for the bilge pumps. The deck section is screwed onto the first part and sealed with waterproof glue. Openings of the top part include the mounting of the mast, electrical connections to the solar panels, and two antennas. As a second barrier to intruding salt water, all electrical components inside the hull are once again in IP68 waterproof plastic boxes. Interconnections are done using watertight plugs and cables.

The boat uses four panels of mono-crystalline solar cells mounted towards the back of the deck as the main source of electrical energy. On sunny days, the surface area of around two square meters produces more than enough energy to run the boat for twenty-four hours. A lithium-manganese battery with a capacity of 2400 Wh stores the energy over night an can supply the boat for almost three days without solar input. To cover for longer periods of bad weather, a fuel cell running on ethanol was also included, supplying energy for about 30 days.

1.2 Problem Setting

The challenges in autonomous sailing are diverse. On one hand, power input from the solar panels is unpredictable and the battery installed in the boat is only able to supply power for roughly 2.5 days. With that in mind, power is a scarce resource and its consumption should be well planned. On the other hand, the boat will be on open sea for three months where it will only be reachable by satellite communication. This makes reliable communication of diagnostic data indispensable. Not only does communication need to be intelligent and handle failures, but the whole system must avoid both crashes and lockups.

The spectrum of tasks in this project reached from hardware integration over software design to the conception of a power management system. We will now discuss the problem setting in these three areas.

A hardware platform suitable as communication platform had to be assembled and integrated in the existing system of the Avalon sailboat. It had to include GSM and Iridium satellite phone as external communication devices and a means to communicate internally with the existing system. Moreover, it should measure environmental data, namely all supply voltages and current consumption of the most important parts, temperature and humidity inside the system. In addition, it should be able to determine the boat's position using GPS technology. Moreover provisions have to be taken to enable power-aware operation of the platform; that is, we should be able to turn off all devices that are not in use.

The software running on the communication platform has a number of diverse requirements: For the existing system, several communication services must be provided. The sailing computer must be able to send status messages back to a control center at ETH Zurich. Also it must be reachable by command messages and in case of emergency provide a means of remote control. For this purpose internal as well as external communication is needed, including software for the control center on land. An abstraction layer has to be constructed over the two hardware communication interfaces on the platform so that switching from one to the other is transparent. For external communication, short message service and data connections have to be supported. For all data sources on the hardware platform, measurements needed to be collected, recorded and archived. A message management part needs to keep track of status messages coming from the sailing computer and the log data and send both as needed. The responsiveness of all processes needs to be monitored in order to guarantee that the platform is always in a working state. Lastly, every part of the software system is required to adapt its current consumption according to a power management system.

The power management concept additionally has to optimize the usage of available energy when it becomes scarce. Using hardware measurements the platform can compute the current energy content of the battery and it can obtain a forecast of the expected energy input of the solar cells over the communication link. Also it must be able to influence the energy consumption of every part except for the main processor of the platform. Given this information, the power management has to plan ahead and only use so much energy each day as to maximize the minimal service level on any day the boat is operating.

1.3 Approach

To find an optimal algorithm for power management we formalized the problem by making a number of simplifications and assumptions so that different algorithms could be compared conveniently by simulation. For one we assumed to have available a prediction on the future energy input from an online source. Also we simplified the battery to have ideal charge and discharge characteristics.

Parameters that can be influenced are the various devices of the communication platform that can be used less frequently to save power. Also the power consumption of the whole boat, including motors can be cut down by switching for a certain fraction of time.

We then identified the desired characteristics of a power management algorithm and specified a metric which translated these into a number and lets us rank the algorithms according to their performance. We rank algorithms higher that switch off the boat the least during the period of a simulation run.

Several promising algorithms, mostly based on mixed integer linear programs, were implemented and simulated with different scenarios to compare their performance in different environments.

1.4 Results

Using the BaseBoard as a starting point the hardware system for a communication platform was assembled in waterproof PVC box. It includes an Iridium and a GSM modem for wireless communication ashore and on open sea. Furthermore it can record a set of environmental information: All supply voltages, current consumption of many devices, temperature and humidity in the system, the current position and it is even ready to connect a camera as soon as it is integrated on board. Internal communication works over a serial line and not only can the existing sailing computer be restarted if needed but the power consumers of the whole boat can be disconnected should power management require it.

Software-wise the platform is able to open data connections and send and receive SMS messages on either communication device. Over a TCP interface theses services are offered to the sailing computer. Data from the different sensors is recorded and sent back to a computer at ETH together with diagnostic messages. A power management component can inhibit certain energy-intensive tasks or reduce their frequency and so has control over the power consumption of the communication platform. All processes are monitored in software for reactivity and also the hardware watchdog is used to guarantee maximal availability of the communication services.

For every device that can be switched off by the communication platform the power consumption was measured. Then a list of tasks using these devices was identified. Each of them can be varied in frequency to influence the energy consumption of the platform, making power management possible.

For the power management we found an algorithm that outperforms even far more complex competitors in scenarios with weather forecasts that probabilistic inaccuracy, which best matches the conditions in the real world. This algorithm, a simple MILP program was implemented on the communication platform to perform the power management.

1.5 Contributions

This power aware communication platform contributes to the competitiveness of the Avalon project by:

- Removing complexity: The communication platform frees the mechanical engineering team from having to deal with low-level communication issues by providing a stable high-level interface.
- Conserving energy: The platform makes informed decisions on how much energy to spend based on the currently available energy and the expected future solar power thus maximizing the minimal service level of the sailing boat.
- Increasing safety: The communication platform also acts as an external watchdog to the sailing infrastructure, reseting the system upon detecting error conditions.

2 Hardware

This chapter initially presents an overview of all the electronic hardware in use on the sailing boat. While most systems presented in the first section are beyond the scope of this document, knowing what devices are deployed on the boat gives an understanding of how and with which parts of the overall system our subsystem interacts.

Later, the chapter will give a more detailed overview of the communication platform itself, which encompasses a number of devices that are interconnected using different high or low level protocols. All of the major features which usually correspond to devices, are then discussed in detail.

2.1 Overview of the Boat

Before the communication platform was added to the system, the overall architecture was completely hierarchical as seen in Figure 1: every single device on board was connected, directly or through hubs, to the controlling sailing PC. This embedded computer, running Debian Linux gathers the sensor information from all devices and controls every actor in the system. It runs programs responsible for sailing and navigation, all developed by mechanical engineers. To discern it from other subsystems we call it *sailing computer* throughout this document.

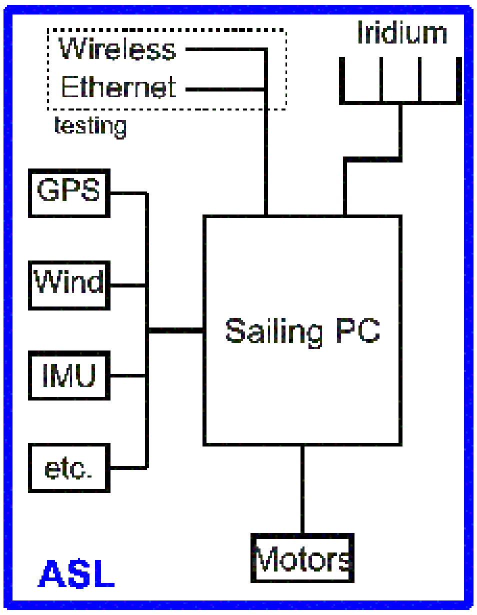

Figure 1: Simplified system overview of the sailing boat before the start of this project.

With the incorporation of the communication platform into the system, the strictly hierarchical topology was broken up. As shown in Figure 2, the newly introduced component is on the same hierarchical level as the sailing computer, with which it communicates over a network link. Still, the failure of either of those systems is fatal to the mission of the boat. Should the sailing computer fail, the boat would not be maneuverable. In case of a failure on the communication part, the boat will still sail on course, but it will not be reachable for intervention. Additionally, no information about its position will be available. However, because there are two different devices on board, they can both be programmed to supervise each other's functionality. According to criteria, if one component, decides that the other is non-functional, hardware is in place to allow for it to restart the non-functional part.

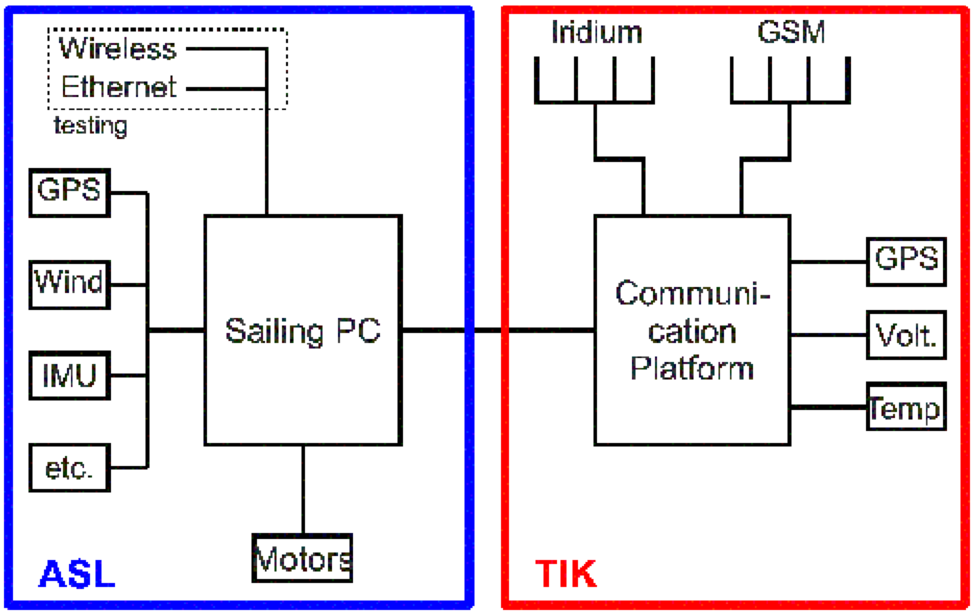

Figure 2: Simplified system overview of the sailing boat including the Communication Platform developed in this project.

The purpose of the communication platform is to keep the sailing computer from dealing with low-level communication issues and allow it to gather as much environmental information as possible. It will also take care of energy management on the boat, as the input from the solar panels is limited and the system will barely run for three days on battery power only. The platform communicates with the sailing computer over a network link on a very high level of abstraction. All details dealing with the GSM or Iridium modems, reestablishing lost connection, or automatically fetching command messages are hidden from the sailing computer.

As mentioned, the number of devices that make up the system that is Avalon is almost too big to put into one diagram. Nevertheless, by simplifying things as much as possible it was accomplished in Figure 3. The diagram only shows the active components (no antennas), communication buses (thin lines), and voltage supply (bold lines). Only the positive power lines are shown, while connection to the (negative) ground panel is implicitly assumed.

The two wireless routers are only used during testing and will be removed before the actual crossing of the Atlantic Ocean. The need for two routers arose because the network antenna needed to be placed at the top of the mast for acceptable signal quality. Because the mast needs to be freely rotatable, it could not be wired with an antenna or network cable. Hence one WLAN router in the main hull connects to the sailing computer while the second one sits on the top of the mast and only acts as a repeater.

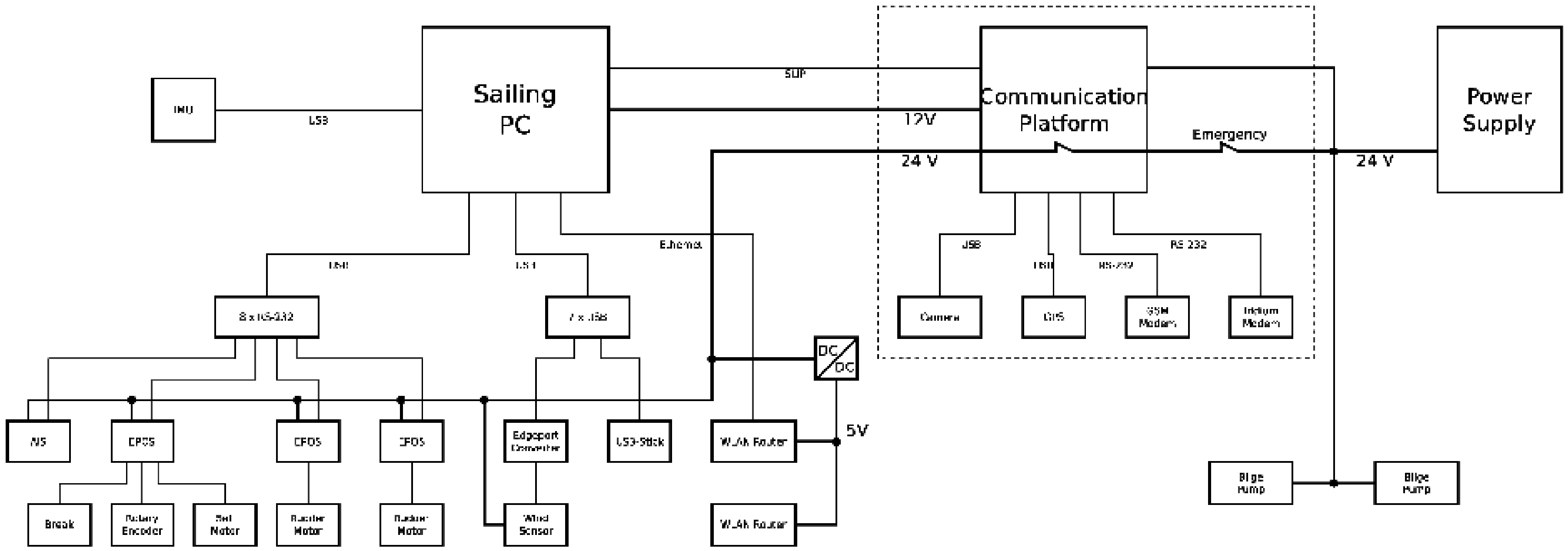

Figure 3: System overview of the sailboat. The subsystem developed for this thesis is framed with a dashed border

2.2 Overview of the Communication platform

As a starting point and core of the platform the Gumstix BaseBoard was used. It had been developed for the PermaSense project [9][10] to serve as a base station in high-alpine sensor networks. The very core of the system forms a Gumstix Verdex Pro, a single-board embedded computer, featuring a main processor from Marvell, the PXA270, running at 600 MHz with 128 MB of RAM and 32 MB of non-volatile Flash memory. The slightly modified BaseBoard acts as a break-out board and adds additional peripherals and a regulated voltage supply. The TinyNode [11] connector also included on the board was not used in this application; instead, some of its pins were interconnected to get access to an additional serial port and to connect the temperature sensor directly to the Gumstix.

Features of the BaseBoard include:

- 5.5 - 18V unregulated power input
- Generated voltages: 3.3V, 4.2V, 5V
- 3 user-switchable power outputs with the same voltage level as the power input
- RTC with battery backup (keep datetime over power cycles)
- two RS-232 ports
- GSM connector for Siemens MC75 and MC75i
- 3 USB connectors with user-switchable 5V supply
- voltage and current measurements on board (10 channels in total, 3 channels free)
- various General Purpose Input Output (GPIO) ports

The Gumstix board itself disposes of a microSD slot which we use with an 8 GB card to store data such as pictures, videos and log files.

To this central building block

Two main devices for external communication are included in the platform: a GSM modem, mainly used for testing at high data rate and small cost, and an Iridium satellite modem, apt for data transfer on high seas. Both devices use a serial line for data transmission with the main processor, and both devices can be switched off to reduce power consumption.

To provide a source of information independent from the main sailing computer, the system also includes several devices for data acquisition. Besides current and voltage measurement of the BaseBoard power supply, a daughter board can measure the total power consumption of the entire boat as well as cut off power to most devices on board. Additionally, a GPS receiver for independent location data and a web-cam (because an image says more than a thousand words) are connected and powered via USB.

Figure 4 shows the interconnections between the devices constituting the communication platform. The BaseBoard acts as a main board to which routes all data lines to the Gumstix computer. The A/D converter to measure voltages and currents is directly integrated on the BaseBoard, as is the temperature sensor SHT11. The BaseBoard is assembled inside a PVC box together with the GSM

Modem, connected via a serial line and a GPS module, connected over USB. Outside the physical enclosure but still part of the communication platform is the Iridium modem.

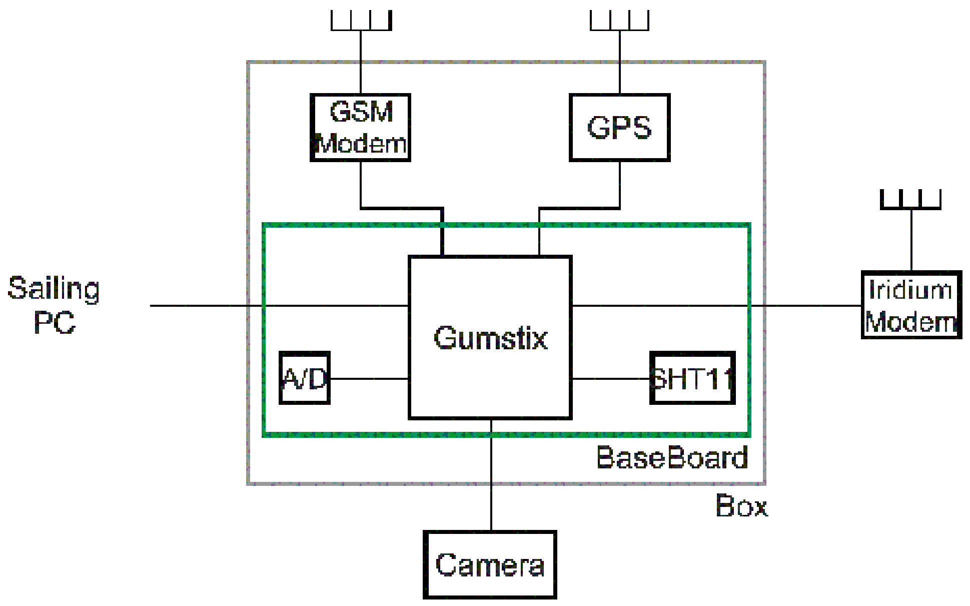

Figure 4: Overview of the communication platform showing only communication lines.

2.3 Platform Components

We will now present the main features of the platform hardware. Most features map to one or several devices used in the implementation; they will be introduced in this context, additionally covering power supply, internal and external data connection.

2.3.1 Main Switch

The most important precondition of doing power management in any system is to actually have a means to influence the current power consumption. While the sailing PC can stop the actual sailing process, bring the sail and rudders in a save and stable position, and stop all additional actions, it can not actually switch off the devices it controls by switching off their supply voltage.

Therefore, the system contains a daughter board which routes the main power supply to most devices on board, can measure, and most notably, switch it off. Devices which cannot be switched off by this method are the two bilge pumps which are of vital importance to the survival of all electronic equipment and to continue working no matter what the state of the power management.

The daughter board used is the same as the BaseBoard, however only a fraction of the board responsible for switching the input voltage for external users and measuring current consumption is utilized. The rest of the board is left unassembled. Reusing an existing, tested circuit board saved the time otherwise spent developing, manufacturing and testing a new, single-use piece of hardware.

The power-switching circuitry is laid out for a continuous current of 20 A, about ten times more than the boat will use in average. To make sure that the main switch stays closed (conducting), even when the connection between BaseBoard and daughter board fails, a pull up resistor was installed on the daughter board, which has to be overridden by the BaseBoard to switch off the main consumers.

2.3.2 Rebooting of Sailing Computer

As a system that has to run unattended for months, an autonomous sailing boat has to completely avoid any system lockups, crashes, hangs or any similar issues. Of course every software and hardware is designed not to break or crash, but as long as software is written by humans it will contain bugs. Besides making programs provably deadlock free, a second method deploys a system of mutual surveillance and restarting. The communication platform has to supervise the sailing computer's functionality and can, should an error condition occur, reset the other system into a working state.

The functionality for resetting the sailing computer has to be implemented in hardware, as any 'reset' software-command is useless if the process that is supposed to handle it crashed long ago. However the sailing computer, an industrial grade embedded computer system, does not have an electrical reset input. It only disposes of a press-button soft-power switch, so the only possibility to forcibly reboot it is to cut and reapply power. This is done by powering the sailing computer by using one of the switchable power outputs of the BaseBoard. One of the positive side effects of this is that the current consumption of the sailing computer can now be measured independently of the other consumers. An appropriate pull-up resistor has been applied, so as to ensure the only condition of power being cut to the sailing computer is when the Gumstix drives the responsible port low. While the processor is off or booting, this output is in high impedance state, making any interference with the switching process impossible.

Ideally, supervision and rebooting would also work in the opposite direction. Because this option has not been implemented, we only mention here that the BaseBoard has an electrical reset input. The sailing computer could then use an output, such as a control signal from the parallel port to drive this input high when its software detects a system lockup on the Gumstix.

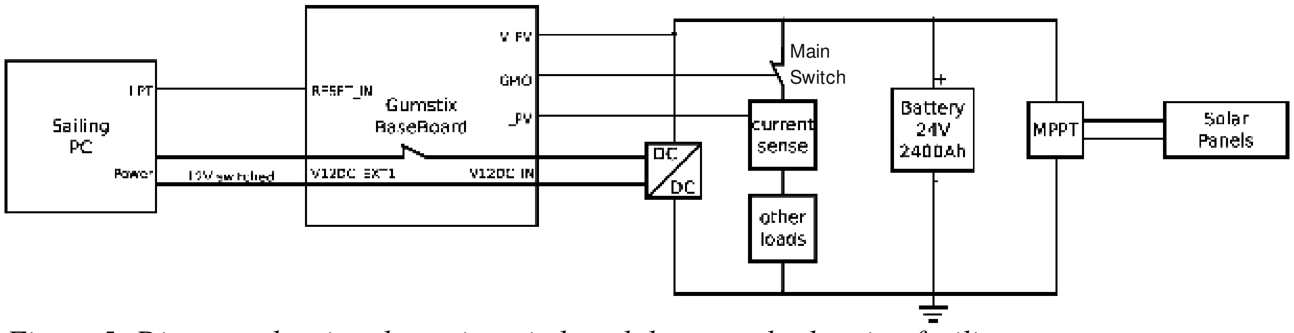

Figure 5: Diagram showing the main switch and the mutual rebooting facility

Figure 5 shows the outline of the power supply for the communication platform and the sailing computer. On the right hand side, solar panels input power and load the battery. To the left the main switch controls the current through all other loads except for the communication platform and the sensor platform. the latter two are connected over a DC/DC converter, because the input voltage to the BaseBoard is limited to 18 V. V_PV and I_PV are inputs to the A/D converter for voltage and current measurement. On the Gumstix BaseBoard itself the supply current for the sailing PC is switched to allow rebooting the latter. The connection from the parallel port of the sailing PC to the reset input of the BaseBoard is a possible solution to allow for rebooting the communication platform but not actually implemented.

2.3.3 Iridium

An integral part of the communication platform is the connection of the system to the outside world. As the sailing boat will be on high seas for months, no earth-based communication media is suited for reliable transmission. The only medium with sufficient reliability are satellites. These days several commercial satellite networks are available for data transfer on the Atlantic ocean. The SSA team chose to use the Iridium network.

The Iridium satellite constellation is a group of sixty-six satellites, providing worldwide voice and data connections in the L-band. The satellites are in low earth orbit, resulting in a high velocity over ground and therefore short time of visibility. At the equator, any satellite is visible for a maximum of seven minutes. When a call is in progress to a satellite that is currently going out of sight, an attempt is made to hand over the call to another satellite. If the handover fails, the call is dropped.

The Iridium network supports voice calls, data connections with up to 2400 baud, SMS and Short Burst Data (SBD) sending. SMS is the short message service known from the GSM network designed text messages, while SBD is a service to send short packets of up to 2 kb of binary data. Both these services are connectionless, and charged by message or amount of data respectively. Voice and data connections are circuit switched and billed by connection time.

The Iridium modem is connected with an RS-232 serial connection to the Gumstix. Because the first two serial interfaces were in use by the communication connection to the sailing computer and the GSM modem, the only remaining interface, originally devised to connect to a TinyNode, was used. To this end we desoldered the existing connector and interconnected several pins to route the serial signal to an unused six pin header. Then, a small auxiliary board holding a RS-232 driver/receiver and a 26-pin ribbon cable connector matching the one on the modem was fabricated. Due to the limitation of the third serial interface of the (Gumstix, only a 3-wire RS-232 connection was used, without the possibility of hardware flow control. [12]

Also on the auxiliary board is a small transistor circuit to drive the power switch input of the modem. If this input is brought low and high again the modem power state is toggled, meaning it switched on if it was off before and vice versa. However this method of power switching exhibited several disadvantages:

- The power consumption didn't go to zero but lingered at about 0.2 Watt.
- There is no way to make sure what power state the modem is by hardware, instead the serial line has to be used to check if the modem is responsive.
- The switching actually proved unreliable; sometimes more than one high-low-high transition was needed to switch the modem off.

The decision was then made to instead use one of the switched power outputs of the BaseBoard. This ensured that the modem used absolutely no power when switched off and is much more reliable.

2.3.4 GSM

Global System for Mobile communications (GSM) is a popular mobile phone standard, allowing besides phone calls, short message sending (SMS) and packet switched data connections using General Packet Radio Service (GPRS). Data Rates range up to 474 kbit/s and data transmission is billed by amount of data rather than connection time. The almost complete coverage of the GSM network on lakes and near sea costs makes it an ideal substitute for Iridium while testing, with lower cost and higher bandwidth. Furthermore, a GSM connector was already present on the BaseBoard, thus facilitating the deployment.

The Siemens MC75 GSM modem [13], which offers quadband GSM support, GPRS and EDGE, was used. Communication with the modem takes place either via a serial or an USB port. The BaseBoard implements both alternatives and the Serial port was used. Also a number of GPIOs are connected to the modem and used to monitor the power state and power off the module.

2.3.5 GPS

An indispensable part of monitoring an autonomously moving device is to know the current location. This is important for tracking the progress and to recover the vessel in case of emergency. Global Positioning System (GPS) is the state of the art technology to pinpoint any device on the earth's surface and won't be discussed in detail here. We use a USB GPS module LEA-5H-0-005 from u-blox [14].

Our GPS device is mounted inside the box of the communication platform and connected to the BaseBoard via power switched USB. Time to first position fix is thirty seconds in perfect conditions, so switching off the device can drastically reduce the power consumption as positioning data for monitoring purposes is not needed more often than once every thirty minutes. Satellite visibility is expected to be ideal in our case, as the antenna is mounted on the boat surface with no mentionable metallic parts protruding higher to block any signals.

2.3.6 Serial Communication

The first serial line of the Gumstix is wired to a 4-Pin Souriau connector on the box which also includes the switched power output for the sailing computer. Both are routed through one and the same cable to the sailing computer. It includes a ground line and a power line and the two serial data wires for both directions.

2.3.7 Temperature and Humidity Sensors

Temperature and humidity are useful environmental data to assess general system health. The BaseBoard has a designated connector for the SHT11 temperature and humidity sensor [15]. However, it is designed to connect to a TinyNode that was not used in the setup. Therefore four contacts of the TinyNode connector were interconnected to route the data- and clock-line of the sensor directly to two GPIO pins of the Gumstix.

The sensor provides temperature measurements in 14-bit resolution from -40°C to +124°C. The humidity measurements cover the full relative range from 0% to 100% with a resolution of 12 bit. The hardware interface consists of a clock input and a bidirectional data line and uses a custom protocol. As there was no Linux driver available, a small kernel module was developed that offers temperature and relative humidity values via the proc interface.

2.3.8 Voltage and Current Sensors

The BaseBoard features a ten-channel sigma-delta ADC (AD7708B) that is used to sample voltage and current measurements of internal and external power supply lines. Conversions are done with 16-bit resolution, which is more than enough to get a rough idea of system health and battery charge state. The converter chip is connected to the gumstix via SPI bus. A driver was already available from the permasense project which exports the measurements via the proc interface.

For voltage measurements the inputs are converted to the ADC's input range using voltage dividers (in the magnitude of 100 kΩ). Each input channel has a specific range, defined by the dividing ratio and the input range of the ADC.

Current measurement is done using a small shunt resistor in the power supply line and current sense amplifier with voltage output (MAX4372). The Maxim chip amplifies the voltage drop over the shunt by a constant factor, allowing for small resistor values at a decent resolution. Shunt resistors are in the range of 0.5 to 5 mΩ, so waste power is in the order of microwatts. Again, each channel has a specific input range depending on the size of the shunt resistor and the gain of the amplifier.

The voltage and current measurement of the entire boat is done with the same technique. The shunt and current amplifier are located on the daughter board and the analog signals are then routed to the external measurement inputs of the BaseBoard

2.3.9 Camera

During the course of the project, provisions have been taken to include a digital camera on the sail-boat. However, because work on the boat was stalled by the SSA team before the camera case had been delivered, it has only been tested on the table, but not integrated into the boat.

A customary webcam in a custom waterproof case was used. Like other parts of this project, it was possible to use a camera case developed for the permasense project. This IP68 waterproof camera case was already in use in heavy weather conditions. The camera is a QuickCam Pro 9000 from Logitech with an optical sensor of two megapixels. It can record still images and video clips with up to thirty frames per second. The connection to the BaseBoard is a standard high-speed USB port which is power switchable, so the camera will not consume power unless in use.

2.3.10 External LED

As a minimalistic status indicator the platform has an external LED. Even though it is an embedded system that is not meant to be interactively, a status report by a blinking green light is very useful when testing and assembling the platform in the boat. Also different blinking codes can be used to indicate certain operation conditions.

The LED is connected to a designated output on the BaseBoard. It is controlled by a GPIO of the Gumstix and driven by a amplifying transistor.

2.4 Waterproof casing

As a second-line defense mechanism, every component on the board of Avalon is in a waterproof containment. In addition, the whole boat is built to be watertight. Moreover, any large amount of water entering the main boat hull will be pumped out by two bilge pumps.

The main part of the communication platform, the BaseBoard (including the GSM modem), the daughter board and the GPS receiver, are installed in one waterproof PVC-box. The Iridium modem is in another box together with other communication components such as a wireless router and the Automatic Identification System (AIS). The camera is obviously, also in its own box outside the main hull. All wires leaving the box are either IP68 watertight Souriau connectors for data and power lines or threaded coupling connectors for antennas.

3 Software

A large part of the this project's time was spent designing and implementing the software of the communication platform. The work encompassed everything from the top level design down to dutifully dealing with every return value of every system call. For the temperature and humidity sensor, even a small kernel module was written, as this device had never before been used with the Gumstix.

In this chapter we are first giving you a brief overview of the different software components and how they interact by means of listing the functional requirements. We then present the methods of inter process communication used, as they play an essential role in constructing a deadlock free software system.

The main programs, running as system daemons, have been written in C++ making heavy use of the standard Standard Template Library (STL) and the IOstream libraries. Furthermore many configuration files and shell scripts have been written to customize other, standard system programs and to generally integrate our programs into the Linux operating system. Care has been taken to follow UNIX standards and common practice wherever possible. Configuration and program files are installed in standard location using common package-management tools, serial devices are locked using the standard lock file scheme and we use commonplace tools wherever possible, such as *logrotate* or *pppd*.

3.1 Overview

Already in the early planning phase it became clear that the software system would have to meet four main requirements, two of which are clear feature requirements, where as the other two provide restrictions on the operation and implementation of these features and are therefore non-functional requirements.

Firstly, the platform has to provide a means of communication according to the needs of the sailing part of the vessel. This will include message handling and queueing, fault tolerant transmission of data and a means to control the boat in case of emergency.

The second requirement is to monitor data from GPS, temperature and humidity sensor as well as voltages and currents of the power supply. This data has to be recorded and stored, as well as triaged for sending back home over the communication part.

The first big restriction is to support energy aware operation. The system has to be able to adapt its energy consumption to the energy available at the moment and in the future according to some algorithm defined in the next chapter. Therefore any task that requires a mentionable amount of energy has to be executed according to some dynamical schedule.

Last but most importantly, the system has to detect and handle deadlocks by supervising every process providing one of the required functionalities. Should any of them fail, they must be reinitialized to a working state. The hardware watchdog of the Gumstix platform has to be used as the top-level supervisor.

3.1.1 Programming Language and Naming Conventions

All software presented in this chapter is implemented in C++ unless stated otherwise. With the exception of the usual main routine in every program and a few utility functions, all code is object oriented. In particular we also encapsulated threads in a C++ class. In accordance with object oriented vocabulary we will usually call functions and variables belonging to a class *methods* and *properties* respectively.

In this document we will capitalize names for software entities to distinguish them from general uses of the same words.. Examples are Shared Memory, Power Manager or Software Watchdog. Whenever we refer to a specific class or instance thereof we will from now on use these capitalized names. The same naming conventions apply to names of processes, however we try to avoid processes and classes with identical names.

3.1.2 Software Requirements

We will now list both functional and non-functional software requirements including their source and state of implementation. A few terms should be defined here for the sake of clearness:

Status message:

> A message in form of an ASCII string providing a minimal amount of information about the state of the sailboat. It is initiated by the sailing computer and the communication platform appends its own messages before sending it to the control center.

Command message:

> A message sent from the control center to the boat. It contains control commands either for the communication platform or the sailing computer. This represents an active intervention and is therefore considered a means of last resort, as it will violate the competition rules.

Each requirement is characterized by three properties:

Source: The requirements were either brought forward by SSA, as a service provided for the existing system, or by the project description provided by the Computer Engineering and Networks Laboratory, TIK. A few requirements also emerged during the course of the project, indicated with a P.

Type: Each requirement is either functional (F) or non-functional (NF). Functional requirements translate directly into functions of the software systems, while all other requirements are non-functional. These are usually requirements that specify a desired quality of the software system or impose restrictions on how certain functionalities are implemented. [16]

Status: All requirements are either implemented (I), partially implemented (PI) or not implemented (N). For partially implemented requirements the degree of completion is detailed in footnotes.

Description	Source	Type	Status
Provide an abstraction layer for the following communication devices available on the platform: Iridium, GSM and Ethernet.	TIK	NF	I
Support opening an IP connection to the Internet using a communication device.	TIK	F	I
Support sending short messages, either using a special messaging service or emulation it over an IP connection.	SSA	F	I

Provide means to switch on and off the communication devices.	TIK	F	I
Receive status messages from the sailing computer over a TCP interface.	SSA	F	I
Append the current values for position, voltages and temperature to status messages	TIK	F	N
Send status messages as short messages depending on the available service and retry message sending in case of failure.	SSA	F	I
Make the sending of status messages dependant on the current schedule.	TIK	NF	I
Let sailing computer poll command messages over a TCP interface.	SSA	F	I
Periodically poll the current means of communication for command messages.	SSA	F	I
Make the polling of command messages dependent on the current schedule.	TIK	NF	I
Periodically send monitoring and log data to a server at the control center.	TIK	F	I
Make sending of log and monitoring data dependent on the current schedule.	TIK	NF	I
Periodically poll a weather forecast from an internet service.	SSA	F	PI[1]
Allow remote access to the system with a minimal and predictable reachability.	TIK; SSA	F	I
Collect measurements of voltage and current from the A/D converter on the BaseBoard.	TIK	F	I
Collect measurements of temperature and humidity from the Sensirion sensor.	TIK	F	I
Collect measurements from the u-blox GPS receiver.	TIK	F	I
Switch the power of GPS receiver off when it is not in use.	TIK	F	I
Make the acquisition of GPS data dependant on the current schedule.	TIK	NF	I
Make the GPS data, voltage, current, temperature and humidity measurements available to other processes using shared memory	P	F	I
Take pictures using the USB camera.	TIK	F	I
Take videos using the USB camera.	TIK	F	PI[2]
Switch off the camera when not in use	TIK	F	I
Make the acquisition of pictures and videos dependant on the current schedule.	TIK	NF	I
Calculate the number of execution of a number of tasks according to the current battery voltage and a forecast.	TIK	F	I[3]

1 The functions for polling files from any internet service are in place, as is the scheduling of the transmission. However no particular weather data provider was selected and parsing and processing of the data is not implemented.

2 Videos are to be recorded using the ffmpeg program. This could be demonstrated to work on a standard PC, but so far the program could not be compiled for the Gumstix to include v4l support, which is needed to access the camera.

3 Implementation is complete, but no forecast data provider is available at the moment.

Share the current schedule with all processes using shared memory.	TIK	NF	I
Check every working thread for responsiveness.	TIK	F	I
To prove it is still alive, every process periodically updates a counter in shared memory.	P	F	I
For unresponsive threads, kill the owning process and restart it.	TIK	F	I
The process in charge of checking the responsiveness of the system has to be supervised by the hardware watchdog in turn.	ME	F	I
Supervise the sailing computer using the same 'counter in shared memory' technique over a HTTP interface.	ITET; SSA	F	NI
Reboot the sailing computer if it is unresponsive.	ITET; SSA	F	I
Lock all inter-process shared memory with a semaphore	ME	F	I
Attribute all inter-process shared memory to one owning thread, which is responsible for initialization of memory and semaphore	ME	NF	I
Never access a shared resource from more than one thread (except for shared memory)	ME	NF	I
Log progress of all actions to file so modes of operation and error sources can later be reconstructed.	ITET	F	I
Rotate, compress and archive log files and data files daily	ME	F	I
Configuration of all parts is possible using plain-text configuration files.	ME	F	I

3.1.3 Software structure

The aforementioned requirements had to mapped to a number of processes and threads that would guarantee as much independence as possible between different parts of the system while still being implementable with reasonable effort.

We will now first quickly introduce the terms *process* and *thread* to set the ground for further discussion of the software structure.

3.1.3.1 Process

Developing for a Linux system, we use the traditional notion of an UNIX process, which is an running instance of a computer program. A process is created by the operating system (in our case the Linux kernel) by allocating a virtual memory space, loading the program code into memory and executing it. Linux processes have several properties, and we will shortly list the ones important to our system:

– Address Space: Each process is assigned its own virtual address space that is protected against interference with other processes or the operating system kernel. Thus, no process can, deliberately or unintentionally, compromise the data or program code of any other process. On the other hand, this has also the effect that data cannot be copied easily from one process to another, creating the need for shared memory (see Section 3.2.1).

- Process ID: Every process running on the system can be identified by a unique number. These identifiers can be used to send signals to single processes using the `kill()` system call. To publish their ID, all programs developed in this project write it to a *pidfile*. These files reside in a unique location that can be computed from the program name (`/var/run/programm.pid`). We use these pidfiles to detect and avoid duplicate instances of single programs and to find and terminate processes that stopped working (see Section 3.3.1).
- Switching: On a Linux system, many processes are executed seemingly in parallel. To achieve this, the kernel uses time sharing, the only technique available on a single-core platform such as the Gumstix. Processes are executed on the processor in turn, requiring intervention from the operating system whenever control is handed from one process to another. This action, called context switching is an expensive operation which includes storing and restoring the state of the CPU and updating the current mapping of virtual memory. Also context switches mostly render useless the content of caches found on most modern processors.

3.1.3.2 Thread

A thread denotes one point of execution inside a process as defined before. While a process may only have a single thread and therefore a single point of execution, it is often useful to partition the work done by one program so that it can be executed concurrently by several threads inside one and the same process. We now point out some important properties that set threads apart from processes.

- Address space: All threads belonging to one process share the same virtual address space. They can therefore use normal variables that are for example written by one thread and read by another to transfer data. However, it is almost always necessary to take further precautions when using variables in different threads. For one, a compiler might optimize the program code such that the variable is kept in a processor register and not written to memory every time it changes. Other threads will then read an old value when reading the variable. Also read and write operations are typically not atomic in most computer systems, leading to data corruption when one write operation is interrupted by a read operation.
- Another consequence of the common address space is that one malfunctioning thread can corrupt the state of all other threads in a process. Thus it is usually wise to terminate and restart the whole process when one thread experiences a fatal error condition or even blocks execution.
- Thread ID: Every thread is assigned a unique ID within the process it is running. Again, we can use this identifier to send signals to single threads. This is mainly useful to have the thread return from blocking system calls when new work arrives or when it should exit.
- Switching: As opposed to processes, threads are not created by the operation system kernel, but by library functions. In particular, we use the POSIX compliant `pthread` implementation. As a consequence, switching execution between single threads does not require the intervention of the operation system and is therefore much less resource intensive.

We will later in this document often use the term *working thread*. This has the following background: Every process except for the Watchdog uses the thread that is created at startup as a maintenance thread. It will instantiate and initialize several variables and objects, create the process wide log file and start all threads doing all the work. These will be called *working threads*. The initial thread will then simply wait for the working threads to exit, after which it will clean up and terminate the process.

3.1.3.3 Partitioning

To combine all all functions in one monumental process is unwise, because any fault in even the most unimportant part of the system could crash the whole system. Be it an error in an external resource or a bug in the software, any fatal error that can terminate or seriously corrupt a process will affect all tasks executed by that process. Either they are directly compromised by e.g. memory corruption, or they are unnecessarily restarted as the watchdog recognises a deadlock in one part of the system and has to restart every single process.

The other extreme of putting each thread into its own process will minimize the possibility of unnecessary reinitialisations. However it will make implementation much harder in cases where two threads need to communicate frequently, as this will have to happen through means of inter-process communication. In such cases it makes sense to incorporate several threads in one process and make the use of simple shared variables. Also threads with such a high degree of collaboration are far more likely to compromise each other, should one of them malfunction. Restarting all of them will then be even desirable.

In light of these considerations we decided to split the whole functionality into four processes:

1. The Message Manager comprises all communication related functionality. It includes a little TCP server for the interface to the sailing PC, all message and data handling and retransmission logic, as well as the management of the different communication interfaces.
2. The DataLogger runs a dedicated thread for each data source. Their output goes to separate log files.
3. A third process does nothing but power management. It creates a section of shared memory and where it publishes the current schedule for other processes to read. This schedule is recalculated at predetermined intervals according to measurements and forecast data.
4. Lastly, a watchdog process supervises every single thread of the aforementioned entities. It also checks the responsiveness of the sailing computer and engages and serves the watchdog device on the Gumstix platform.

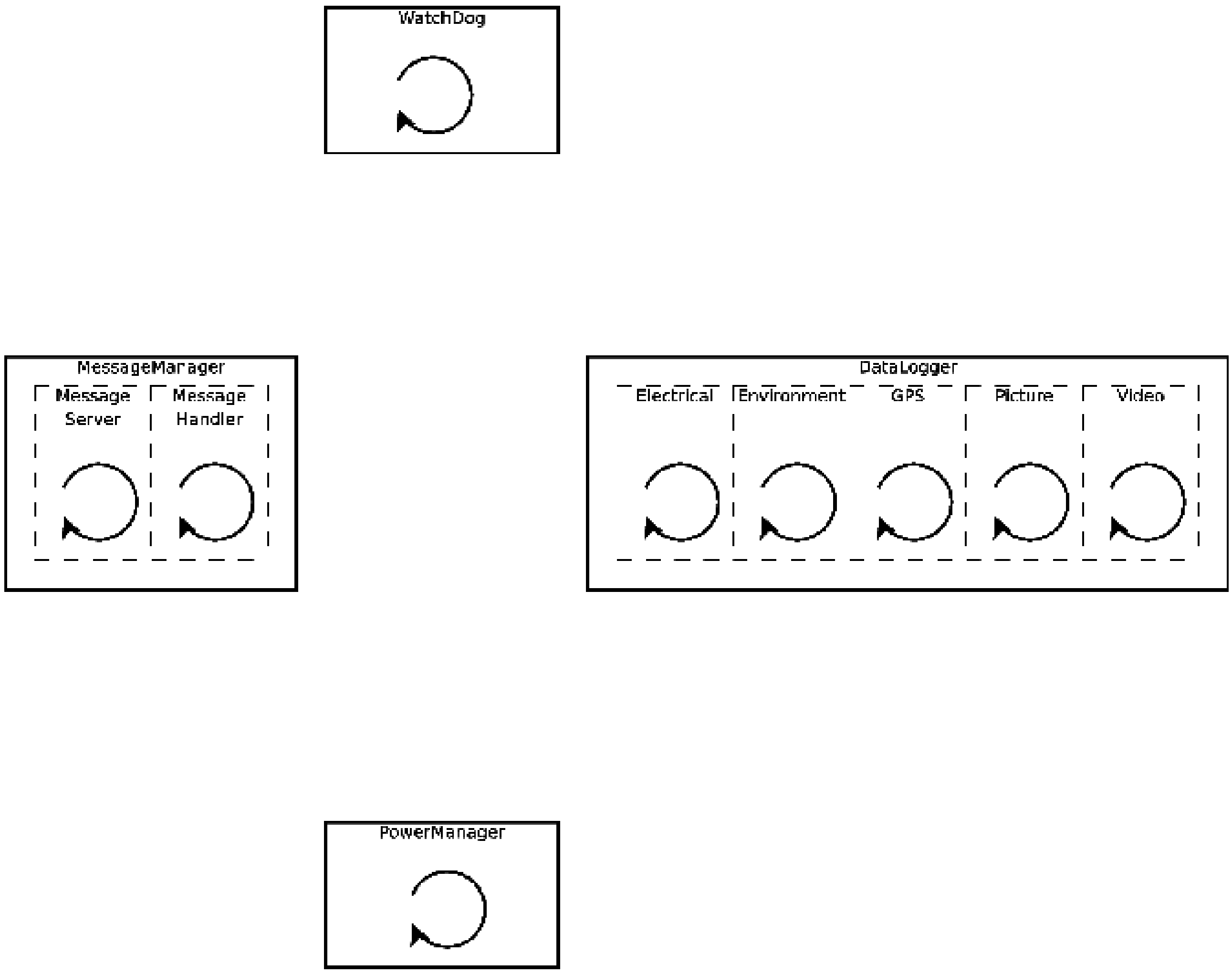

Figure 6: Overview of the Software building blocks. Solid boxes represent Unix processes while dashed boxes stand for single threads.

3.2 Inter-Process Communication

When splitting up our application into several processes we need means to share data in between them. In particular the schedule computed by the Power Manager must be accessible for all components and each one of them has to notify the watchdog of its state of operation.

Inter-process communication bears a few known pitfalls. The completely asynchronous execution of processes combined with preemptive multitasking on the Linux system has to be considered in the design of every component. Any thread can be interrupted by any other at any point of execution. Signals, for example, might arrive while a process is holding a lock for a shared variable. Reacting to it immediately by eg. stopping the current action without releasing the lock could result in a deadlock.

When designing the different components i.e. processes great care was taken not to create any shared resources. In particular for hardware devices there is one sole thread that may interact with the device. Switching the power supply of the Iridium modem for example happens in one thread only. So does polling the GPS device or writing to the serial port of the GSM modem. The only real shared resources in our system are shared memory as explained below.

3.2.1 Shared Memory

While it is easy to share data between different threads of execution inside a single process, doing so in between different processes is only possible with the help of the operating system. Because modern OS' provide each process with its own address space, simply referencing the same memory location won't let processes share data. Instead they have to ask the operating system to map an area of real memory into the virtual address space of both processes.

We use the POSIX compliant Linux system calls `shmget`, `shmat` and `shmdt` for creating, mapping and detaching shared memory segments.

All shared data is only distributed in a broadcast fashion: There is exactly one process that writes to each segment and all other processes will only read. The process writing to the shared memory is also in charge of constructing and initializing it, and must therefore be started before any process wishing to access the shared data. To prevent data corruption that could occur if shared memory is read and written at the same time, we use semaphores (see Section 3.2.2) to protect read- and write-access.

3.2.2 Semaphores

A semaphore is a useful primitive to protect access to a shared resource. Using two operations, `wait` and `signal`, and an internal counter value it can be used to restrict the number of processes executing a critical section to a certain number (often one). The counter value initially represents the number of processes allowed in the critical section. Before entering such a section, a process executes the `wait` operation. This will wait for the counter to be larger than zero, and, when that is the case, decrement it. Upon leaving the critical section, the `signal` operation has to be used to once again increment the counter.

The nontrivial aspect of semaphores is, that the operations have to be atomic. For the `signal` operation in particular, the test if the counter is greater than zero and the following decrementation must not be interrupted by any other thread or process doing the same.

Semaphores are normally provided as library functions by the runtime environment. We use the POSIX compliant implementation available on Linux, particularly the functions `sem_open`, `sem_post` and `sem_timedwait`. By convention, every semaphore is owned by exactly one process, which is responsible to initialize it to the correct value. Every other process will fail to access it unless it has been initialized by the owner.

As POSIX semaphores are created by the kernel, they won't be destructed if the creating process exits. Instead they remain usable and accessible by their name until explicitly destructed by their owner. If the owning process exits unexpectedly without cleaning up, it will simply reinitialize the semaphore on restart, recognizing it had already been created.

We exclusively use semaphores to protect shared memory sections from simultaneous access. Memory access is a fast operation that will, in normal operating conditions, not suspend the execution of a program for a longer period. We can therefore define any call of the `wait` operation that takes longer than ten seconds an error, because no other process should be accessing the memory for such a long time. Using `sem_timedwait` we then make the call return after this fixed timeout, thereby detecting situations where the semaphore was erroneously left locked and avoid any deadlocks.

3.2.3 Signals

POSIX signals are a means to notify a process of an asynchronous event. Common signals are SIGINT or SIGTERM, sent by other programs or the terminal respectively, to notify a program that it should terminate. Another signal widely used among system demons is SIGHUP. It will cause these programs to close and reopen their log files, so they can be rotated.

Every program can register a callback function for most signals that is called by the operating system upon delivery. Because these signal handlers are called completely asynchronously and without knowledge of what code is currently executed by the program, they can only safely call reentrant [17] functions. Any calls to functions performing IO or access semaphores could cause data corruption or deadlocks.

For these reasons signal handlers should be kept short and simple, usually just changing a flag value to indicate the reception of the specific signal. Most system calls return with an error value whenever a signal is handled, making it easy to immediately check whether a flag has been set.

We use signals for two purposes: A few commonly used signals are handled to provide the commonly expected reactions and SIGUSR1 is used for inter-thread communication.

The two signals SIGINT and SIGTERM raise a flag and notify every thread in the program to let them clean up and exit gracefully. Should one of these signals be received a second time, the default action is invoked, which is to terminate the program. As previously mentioned we also handle SIGHUP by closing and reopening any log files the program is writing to. This allows *logrotate* to move, rename and compress these files.

To alert single threads that an event has occured, we use the user defined signal SIGUSR1. This will cause a system call the thread might be waiting on (such as `sigtimedwait()`, `select()` or the like) to return. The thread is then responsible to check what event has occurred. This could be a request to terminate or rotate log files or the arrival of a message .

3.2.4 Files

Regular files are a very simple and platform independent means of inter process communication. We have, hover, not chosen to use them for data that is changing often, as we are developing for an embedded platform whose only permanent storage is Flash memory. Repeated rewrites of the same location, which is to be expected when using small file with often changing content, can cause the flash memory to wear out and fail.

Files are used for saving log data. These files are daily moved to a separate directory and renamed according to the current day. Whenever the energy budget allows it, the Message Manager will reread these files and send them back to the control center.

3.2.5 Network protocol to Sailing Computer

For the communication between the communication platform and the sailing computer a simple function interface was devised. The communication platform always acts as a server and the sailing computer is the client, initiating every request, i.e. function call. The original specification of the interface contained three more methods which were dropped in the course of the project, as it became clear that the sailing algorithms wouldn't need data for wind forecast.

The specification is here first presented in a implementation independent way, specifying input and output arguments for every function as seen from the communication platform. An input argument therefore stands for data traveling from the sailing computer to the platform.

Function name:	send_status
Description:	Send a status message from the sailing computer to the control center.
Input argument:	Status message as ASCII string
Output argument:	Execution status (OK if there message was handled without error, FAIL otherwise)

Function name:	receive_message
Description:	Poll for a command message that has been sent from the control center.
Input argument:	none
Output argument:	Execution status (OK if there is a message pending, FAIL if there isn't)
	Message received as ASCII string, if any

Function name:	read_forecast
Description:	Return the last forecast that was received by email.
Input argument:	none
Output argument:	Execution status (OK if a forecast has been received, FAIL if it hasn't)
	Complete content of email as data array

The interface was finally implemented in three times: as a Python function interface, to be called by the control scripts running on the sailing computer, as a TCP interface over the link between the two systems and as a C++ interface inside the Message Manager, to be called by the TCP server.

The Python and C++ interface descriptions are one to one representation of the above implementation independent description[4]. The protocol we use over TCP is also very simple and shall be described here.

Every function call over TCP consists of two phases: First, the client sends the function identifier, which is one character, over the TCP stream, followed by the input argument, if any. It then calls shutdown for writing on the socket, resulting in a FIN packet being sent. At this point the server knows it won't receive any more data and internally handles the function call. Then it writes the return argument to the stream and closes it.

4 The Python implementation is called message_manager and contains the implementation of the TCP client. For C++ there is an interface description in form of a pure abstract class called Message Handler.

The advantage of this method is that we don't need to explicitly specify the length of input or output arguments, as we signal the end of both using TCP flags. This method however only works as long as we don't have multiple input- or output-arguments of non-fixed length.

We use the following constants:

Function	Function Identifier
send_status	“s”
receive_message	“r”
read_forecast	“f”

Name	Value
OK	“1”
FAIL	“0”

3.3 Components

After covering some common aspects of the different software components we will discuss their functionality in detail.

3.3.1 Watchdog

A high priority requirement for the hole communication platform comes from the fact that it will be used on an autonomous sailboat, which will be on the sea for months without interruption. It must therefore work for a indefinite period of time without crashing or hanging. While the absence of crashes and lockups is of course a requirement for all software with reasonable functionality, errors leading to such events are hard to detect even in moderately complex systems. So while we took action to conceptually avoid deadlocks and write error-tolerant code wherever possible, this system should furthermore be able to guarantee that any such fatal situation is detected and corrected.

The Watchdog process plays the most important part in fulfilling this challenging requirement. It acts as a software watchdog for every thread of our software system and supervises the reactivity of the sailing computer. When detecting errors in either case, it has to take appropriate action and keep track of them. On the other hand it has to be watched by the hardware watchdog of the Gumstix platform. This guarantees that also errors that cause the watchdog itself to freeze or crash are caught.

We will now first discuss the workings of the software watchdog we implemented, then describe the action taken when a lockup is detected. We will also explain how the hardware watchdog works and lastly state what kind of guarantees we can give using this approach.

3.3.1.1 Software Watchdog

Software Watchdog is a C++ class that mimics the behavior of a hardware watchdog. It supports two main operations: `serve()` and `check()`. The first one is to be executed by the thread being supervised to signal that it still alive and doing work. The second one is called by the Watchdog process and checks if the `serve()` operation has been executed within a certain time.

Software Watchdog relies on shared memory as discussed in Section 3.2.1. When the class is instantiated in the supervised thread, it creates a shared memory section of the size of one integer number and initializes the value to 1. We will from now an refer to this variable as the heart value. In the Watchdog process, the constructor then tries to open the shared memory segment of the heart value. If the open operation fails, this indicates that the thread being watched has not started or crashed during initialization. For some threads this is a clear error condition and has to be acted upon, while other threads can be disabled in configuration and this way the watchdog can detect such a situation.

The supervised thread will then go to its main loop and do whatever work it has to do. This main loop has the same structure for all working threads in our system: In a first part, the thread waits for an event to happen. For example, it could be waiting for a TCP connection to be opened or for a signal to arrive. In the second part it will then process whatever task has arrived. We now define the maximal time it may take the thread to execute the second part, i.e. the actual work, as this thread's *heartbeat*. To make the thread correctly serve the Software Watchdog, the fist part (where it mostly waits) has to be designed such that it will always finish after at most one heartbeat, no matter if the event happened it has been waiting for. The second part will then have to check for that condition. We can then insert a call to `serve()` before and after both parts, and be sure that they are called at least once every heartbeat in normal operation.

In the Watchdog process `check()` is executed periodically (every 10 seconds). If an error is recognised, it is then acted upon as described in the next section. We now define the two operations in detail:

The `serve()` operation keeps track of the last value written to the heart in a data member of the Software Watchdog class. Upon execution, it increments that value and once again writes it to the shared memory. As we will see in the next paragraph, it is not important what value is written as the heart value, as long as it is different from the last one.

The `check()` operation uses two data members: one also holds the last value read from the heart, and the other one holds the timestamp of that read operation. Whenever it is executed, `check()` reads the current heart value and compares it to the old one. If it differs, the old value and the timestamp are updated and the function returns `true`. Is the value identical to the last one however, the timestamp is checked. Is it older than one heartbeat, the supervised thread hasn't called `serve()` for too long, which is signaled by a return value of `false`.

Using the information that `check()` is called every ten seconds during normal operation, we can make the following statements about `check()`:

- The timestamp is updated at most ten seconds after the heart value has been changed by the watched thread.
- `check()` returns false at most ten seconds after the timestamp is older than one heartbeat.
- Therefore, `check()` returns false at most twenty seconds after the heart value is older than one timestamp, which is at most twenty seconds after the watched thread has entered an error state.

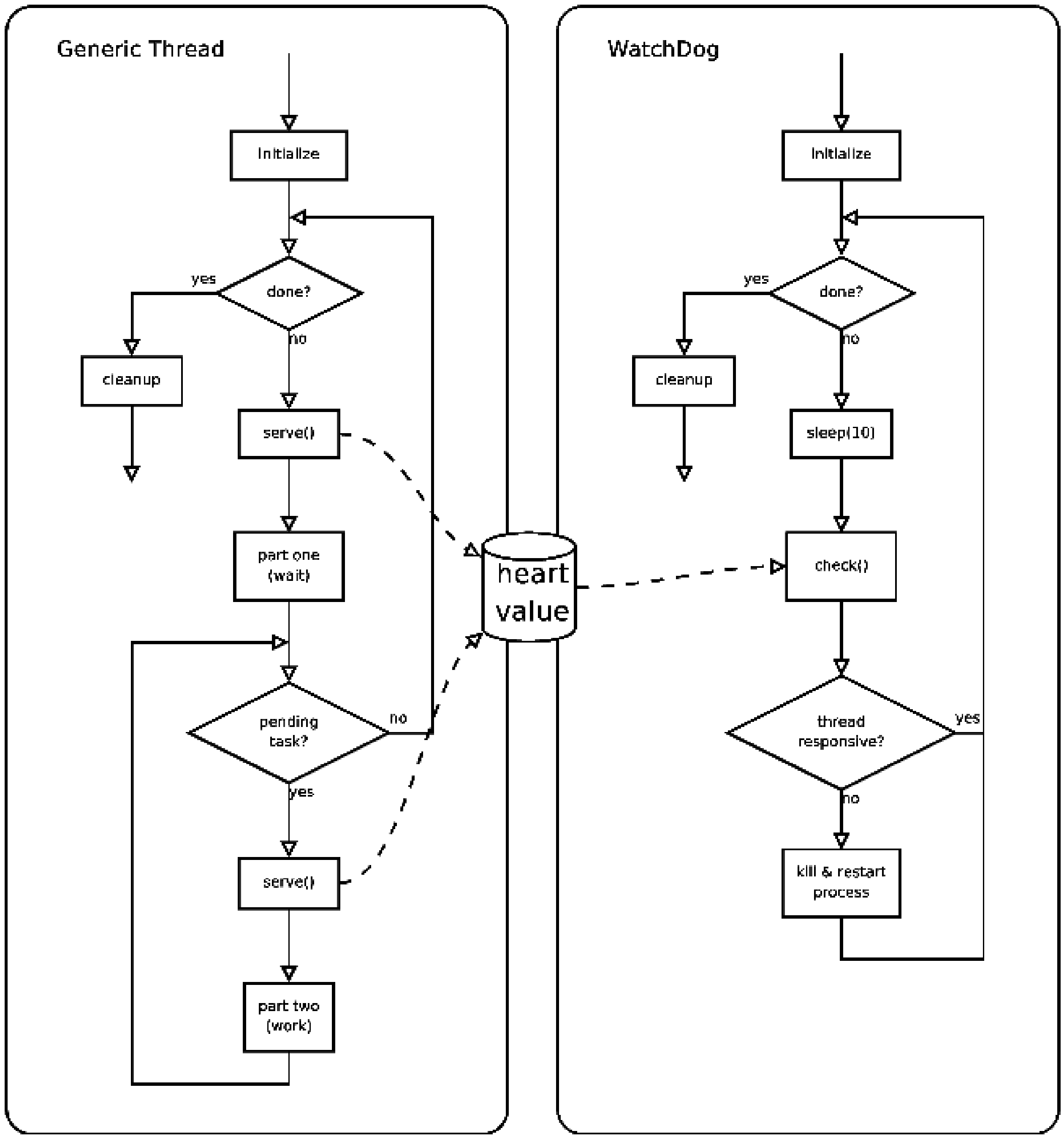

Figure 7: Flowchart of a generic thread and the Watchdog (simplified). The disk represents an instance of shared memory.

3.3.1.2 Actions upon detecting errors

When the Watchdog detects an error in a certain thread, it kills and restarts the owning process as follows:

1. Read the pidfile (see Section 3.1.3.1) of the process. If it doesn't exist, the process already terminated. Go to step 5.

2. Extract the process ID from the pidfile and send a signal 15 (SIGTERM) to the process. If sending fails, the process terminated without cleaning up. Go to step 5.
3. Wait ten seconds. If the process is still working correctly it will exit during this time.
4. Send a signal 9 (SIGKILL) to the process. It will exit immediately if it hasn't already.
5. Restart the process.

3.3.1.3 Supervision of Sailing Computer

The Watchdog process is also designated to supervise the function of the sailing computer and to inform the same that the communication platform is still in operating condition. The following concept however is only implemented on the communication platform but not on the sailing computer. The SSA team has been too busy working on the sailing algorithms to implement any kind of watchdog system.

The software system on the sailing PC builds up on an implementation of shared memory especially designed for robotic systems called DDX[18]. The Spring application collection that is built around DDX includes a HTTP server for access to the content of the shared memory. This allows us to fetch values via a GET request and write data to the DDX store using a POST request.

The approach to detect errors on both devices is the same as with the Software Watchdog. Only here the shared memory runs on the sailing computer and is accessed over the network. To prove it is still in a working condition, the Watchdog constantly updates a value in the DDX shared memory and also checks a value written by the sailing computer for periodical changes. When no changes are detected for two minutes the sailing PC is assumed to have crashed and restarted.

3.3.1.4 Hardware Watchdog

The hardware watchdog on the Gumstix platform is supported by the kernel module sa1100_wdt. It creates a character device called /dev/watchdog. The hardware watchdog is activated by opening the file and writing any data to it. After activation it has to be reset by writing to the same file every sixty seconds, otherwise it will perform a hard reset of the CPU, causing a reboot.

If it weren't for the hardware watchdog, the Watchdog process would constitute a very vulnerable point in our system of interprocess supervision: Should this one process fail, any error detected in the rest of the system would pass unnoticed and the communication platform could enter a unresponsive state without any possibility of recovery.

The Watchdog process serves the hardware watchdog in its main loop every ten seconds, even when restarting other processes (which will include sleeping for ten seconds). At the same time it will also blink the external LED, indicating normal behavior. If a process is found unresponsive and therefore restarted, the LED will blink twice, otherwise just once for one second.

3.3.1.5 Guarantees

Given the previous explanations about the software and hardware watchdogs, we can make the following statements:

- The Watchdog process is always running after system start, otherwise the platform will be reset.

- Every working thread has a heartbeat value, and it is executing the check() operation on its Software Watchdog at least once in a heartbeat. If that is not the case, the owning process is restarted by the Watchdog.

3.3.2 Power Manager

As the title of the thesis states, power awareness is a integral part of all work done in this project. Apart from the Watchdog, all processes were designed with power aware operation in mind. To start off, we now define the two terms *task* and *schedule* as we will be using them in this document.

Task:

> With respect to power management, we define a task as an operation that requires more energy than the base consumption of the Gumstix. As such, every task includes switching on an external device for a certain amount of time.
>
> The list of different tasks that can occur is fixed and known. Also we have measured the power consumption of every task and defined a minimal and maximal number of times it is executed per day. These properties are not important right now but will be discussed in the next chapter (see 4).
>
> Different processes will want to execute tasks in response to different causes. If a process wants to execute a task, we say the task is *generated*. If the process receives permission from the power management, it may proceed and *execute* the task. Some tasks are generated periodically while are caused by asynchronous external events. In the software system, each task is identified by an ID.

Schedule:

> A schedule is a data structure that assigns each task the number of times it may be executed in one day. It is implemented as a table using the task IDs as indices.

The sole purpose of the Power Manager process is to compute the schedule for the next 24 hours at the beginning of each day. It does so using an algorithm we will explain in detail in chapter 4. The output of that algorithm is then written to shared memory, as discussed in Section 3.2.1. From there, all other processes wishing to execute tasks can read the allowed total for the day and then decide whether to execute it or not. The processes always need to keep track themselves of the number of executions so far. For that decision either of two algorithms is used:

1. For a few tasks that do not occur periodically but rather asynchronously, the task is executed if the limit for the current day has not been reached yet. For example, if that task arrives four times in a specific day but is only allowed two executions, the two first instances will be executed, while in the second two instances it would be skipped.
2. For all other tasks that appear periodically, we want to spread out the actually executed instances evenly over one day. So if a task is generated every 30 minutes, but only granted six execution in a particular day, it will execute every four hours. This is achieved by comparing the ratio of executions so far to executions allowed in total to the ratio of 24 hours that has passed since the computation of the schedule. If the former is smaller, a new execution is allowed, otherwise it is not.

As we will see in chapter 4, the algorithm to compute the current schedule involves solving a MILP problem. We use a library called lp_solve which can solve mixed integer linear programs. lp_solve is available under the GPL and was initiated by Michel Berkelaar from Eindhoven University of Technology but then improved by many others [19]. The library can be used both in Matlab and from a C interface, which made it ideal for our purposes, as we could use the same routines for simulation as well as implementation.

The library was easily ported to the Gumstix as it only uses ANSI C. Because it didn't make use of the common GNU autotools but used custom makefiles, we didn't deem it worthwhile to port it as a shared library. Instead it was compiled as a static library and linked directly to the Power Manager executable.

3.3.3 Message Manager

The Message Manager executes all message- and communication-related tasks. On one hand side, it communicates with the Sailing computer over an IP link, on the other hand it manages the two communication devices Iridium and GSM, sending and receiving short messages and opening data connections as needed.

Naturally, these two aspects are implemented in two separate threads. The MessageServer is a simple TCP server that implements the TCP interface presented in Section 3.2.5. It parses the requests arriving over the internal network link and hands them over to the C++ interface implemented by the other component, the Message Handler. This unity handles all transmission requests, such as status messages, command messages and data transfers of log files. It switches power to the communication devices and handles retransmission. We are going to discuss these two parts in detail.

3.3.3.1 MessageServer

The MessageServer is a simple TCP server listening on port 2222. It uses a single thread which means any connection requests coming in while another connection has already been established will be queued until the server is done handling the last connection. The queue size for pending connections is 5 which will never be exceeded by the sailing PC in normal operation.

In accordance with the protocol specification the rough course of action in the TCP server is at follows: After constructing a socket, binding it to port 2222 and configuring it as a passive socket the thread goes to its main loop where it calls `accept()` on the socket. This function call blocks, until a new connection is established. When this has happened, the server proceeds to read from the socket until no more data is available, that is the client has sent a FIN packet. The byte vector read from the network is then fed to a handling routine which we will discuss later. It returns another byte array which is then written back to the socket and sent over the network.

A too simple implementation using `accept()`, `read()` and `write()` without further precautions however bears certain dangers: all of these functions can block for an indefinite time while waiting for data or connections. With `accept()` this is usually intentional, we however must serve the Software-Watchdog at least once every two minutes so we want to apply a timeout to that call. For the other two functions the situation is more serious: a crafted TCP connection could block such a overtly simple server up by slowly and continuously sending data. The only server thread would be stuck reading from that one client and new incoming connections could not be answered.

To overcome these dangers we firstly configure all sockets (the listening socket and the connection sockets) as non-blocking. This signifies that calls to the three mentioned functions will return immediately with an error code if no data is available. Secondly, we precede every one of these function calls with a call to `select()` (or `pselect()`) for which we can specify a timeout after which the call is guaranteed to return.

The timeout for the accept() call is the threads heartbeat as defined in Section 3.3.1.1. For the read() and write() functions we define an *accumulative* timeout of 30 seconds both. This signifies that from the start of the connection until all data is read no more than 30 seconds may expire and also sending all data over the TCP connection may not take longer than that. This guarantees that the message handling time will always be smaller than 60 seconds plus the time required to parse the data and handle the function calls.

Also we implemented a limit on the size of incoming messages. As soon as the data read from a single connection exceeds 2 kb the server drops the connection.

Handling of the received data is fairly simple: The first byte is inspected to find out which function the client is calling. The rest of the data is then converted to whatever data type that function expects as a parameter and the function implemented by the Message Handler is called. All precautions needed when communicating in between two threads are implemented on the side of the Message Handler. If the function called through the interface throws an exception or if the format of the message received over TCP is incorrect, the server sends a fail code and closes the connection.

3.3.3.2 Message Handler

The Message Handler is responsible for executing a fair share of the tasks as we defined them in the context of power management. For one, it will send the status messages received over the TCP interface back to the control center. Also, it will have to poll for command messages that were sent via short messaging and make them available to the sailing computer through the interface. Lastly, it opens a data connection and transmits log data from different sources and retrieves a weather forecast. The last point includes many tasks in the sense of power management, for the Message Handler however we summarize them to a big data transmission task so that only one connection is opened and then as many items as allowed are transmitted.

To avoid confusion with task as defined in the context of power management, we will now call the three jobs sending status messages, polling command messages and transmitting data *events*. Every event is identified by its type and the time it has to be executed. Before that time, the event is said to be *pending*, afterwards it is *active* until it is executed. Because the different events become active at different intervals, are generated by external sources or even have to be rescheduled because of transmission failures, we need an EventQueue to keep track of them. It includes operations to push an event in the queue, pop any event that is active and a query to find out the time until the next event becomes active. Internally the EventQueue keeps the events in a heap structure according to their due time.

Operations on the EventQueue are thread safe by locking the internal data structure with a mutex. This allows the MessageServer thread to directly push an event into the queue if it receives a status message from the sailing PC.

In its main loop, the Message Handler sleeps for the time it takes for the next event to become active or the heartbeat, whichever is smaller. It then feeds the watchdog and checks if any event is now active and needs to be handled.

We will now go through the three events and discuss how they are handled in detail:

Sending Status Messages

When the MessageServer receives a 'send status message' command from the sailing PC, it calls the appropriate function on the Message Handler, which pushes an event on its queue and stores the content of the message in a variable. Should a new message arrive before an old one has been sent, the old event is removed from the queue and the content of the old message is overwritten. In other words, there is always only one status message pending to be sent.

Upon pushing the new event in the queue, the Message Handler thread is also sent a signal so it wakes up should it be sleeping. It will then first check with power management, if message sending is allowed. If it is not, the message is dropped and not rescheduled. Otherwise the message will be sent using the current TransportHandlers `send_short_message()` method as discussed below. If transmission fails, the message will be rescheduled 180 seconds later for a maximum of three attempts.

Polling Command Messages

Events to poll status messages arrive periodically every hour. This is done by rescheduling the next event as soon as the previous one is handled.

As with status messages, the Power Manager is consulted first, to find out if execution is allowed. Then the poll_short_message() method is called on the current TransportHandler. If a message has been received, it is stored in a thread safe variable where the MessageServer can fetch it. Failures in message polling are ignored, as the next event will occur in one hour already.

Data Transmission

Data transmission events happen three times a day. Upon each execution, the Message Handler tries to send as much data as is allowed. If it succeeds, subsequent executions on the same day will have no more data to transmit and therefore simply return. This helps keeping down the overhead of opening a connection, which takes about 30 seconds.

The Message Handler first builds a list of all log files to be transmitted. That is, all files that are of interest and have not yet been sent and that the Power Manager allows sending. If this list is empty, another data transmission on the same day had been successful beforehand and nothing needs to be done. Then the Power Manager is queried if opening a data connection is allowed and if so, it is opened using the `open_ip_connection()` routine of the TransportHandler. Then each file in the list is sent over HTTP using the libcurl library and taken off the list if sending suceeds. Subsequently the IP connection is closed.

The data transmission event is also the place to poll a weather forecast from an internet source. As we haven't selected a specific provider yet the actual fetching and parsing has not been implemented. All the libraries for fetching data over HTP, FTP or email are in place however.

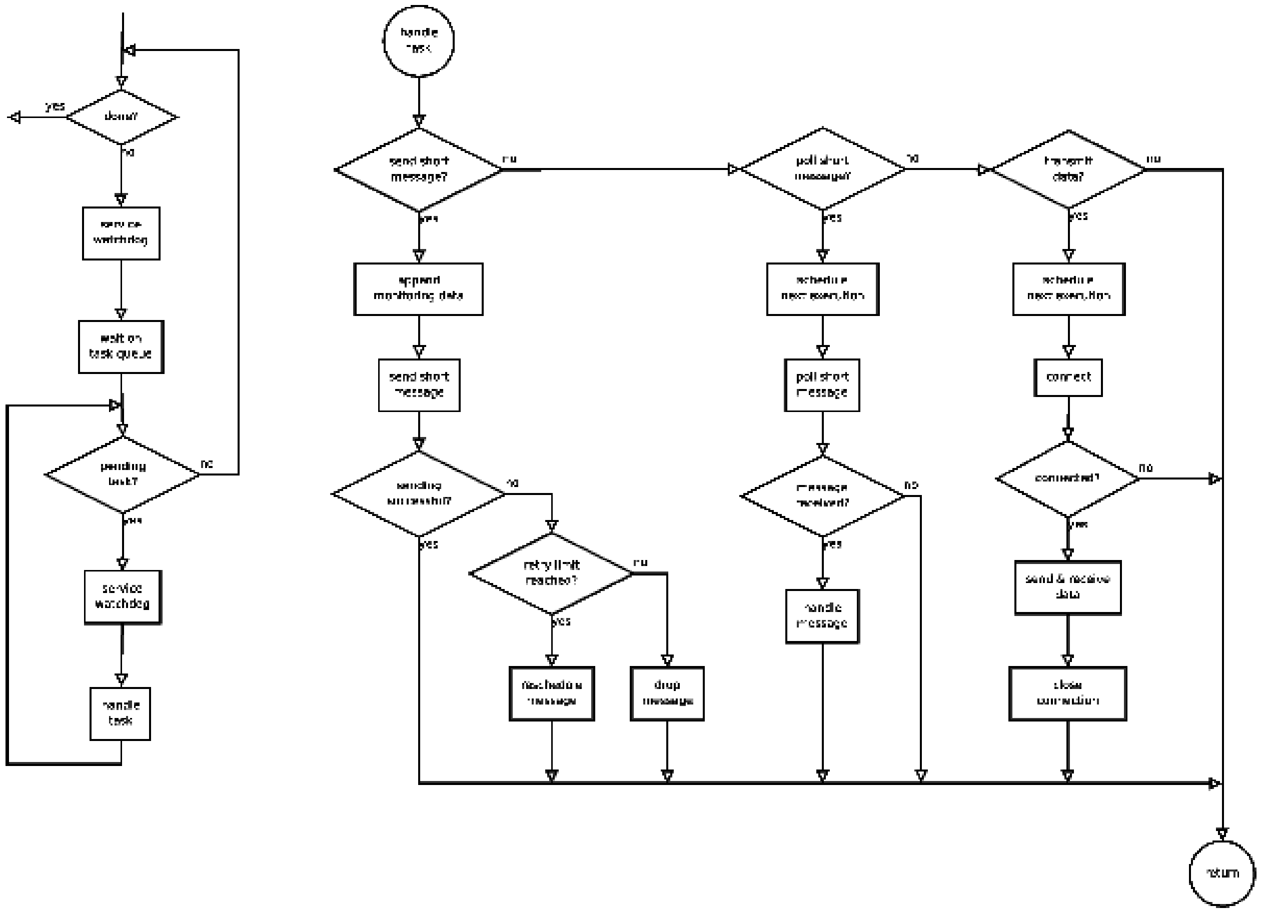

Figure 8: Flowchart of the Message Handler thread. Circles stand for method entry- and exit-points.

3.3.3.3 TransportHandler

To hide the implications of having different communication devices from the largest part of our software system, we specified a common interface to all those devices called TransportHandler we now first discuss the methods of the interface and then shortly list peculiarities of the different implementations.

Method name	**Return type**	**Description**
`open_ip_connection`	void	Open a data connection to the internet.
`is_connected`	boolean	Query if a data connection is currently open.
`close_ip_connection`	void	Close an open data connection.
`send_short_message(string)`	void	Send a short message without opening a data connection if the handler supports this.
`poll_short_message`	string	Check if a short message has been sent to this device. If no message is pending, return an empty string.

For Testing purposes we implemented a dummy handler called EthernetHandler which assumes that a connection is always available and sends short messages over HTTP. It is useful when testing the Message Manager on a standard PC.

For both the Iridium and the GSM modem we use the `pppd` program to open IP connections and they also identical parts of the AT command set for GSM Mobile Equipment [20][21] for sending and receiving SMS messages. We therefore implemented the common functionality in a TransportHandler called PPPHandler. Both the IridiumHandler and the GPRSHandler inherit from this implementation and adapt them to the specifics of their respective devices.

IP connections are established by starting the `pppd` daemon which connects to the modem, performs the dialing, the negotiation of the connection parameters and configures the network device to the address assigned over the PPP protocol. Upon both establishing and terminating a connection `pppd` runs a small helper program called MessageManagerNotifier which notifies the appropriate PPPHandler via shared memory of the current state of the connection.

To send and receive SMS messages, the PPPHandler directly connects to the modem over the serial port of the respective device. It then uses AT commands to send or query messages in PDU format. While the text format would have been much easier to implement, the Iridium modem only supports PDU messages. When communicating over the serial line we again implemented accumulative timeouts for both writing and reading which default to ten seconds. An exception is the actual send command, where we allow the modem 30 seconds to complete transmission and report success.

One peculiarity of the Iridium handler is that it runs an own thread for power management. This thread wakes up once every hour and asks the Power Manager if the modem may be turned on for the next 60 minutes. If so, the modem is turned on by default, otherwise it will only be switched on when needed. Leaving the modem powered up gives us the possibility to open a data connection from the control center by dialing the Iridium number from an analog modem. We then have shell access to the communication platform and could take precautions in a case of emergency.

3.3.4 DataLogger

To facilitate recording data from the different data sources we have connected to the platform, we introduced a generic interface called DataSource. It defines one function to read measurement values as a list of floating point numbers including a timestamp, as well as functions to turn the source on and off, if applicable. For all sources in our system we then wrote a class implementing this interface.

Some sources execute a task in the sense of power management when they perform a measurement, such as the GPS device. The function using the DataSource is responsible for checking with the Power Manager if performing this task is allowed. The same sources also require some time to power up their device and collect data before the first measurement can be taken. These porperties are all accessible through the DataSource interface.

Each DataSource has the following properties, accessible through getter methods:

Property name	**Data type**	**Description**
Name	String	Name of the data source
Setup time	Integer	Number of seconds the source needs to provide reliable data after being powered up.

Task ID	Integer	ID of the task the source executes when performing a measurement.

Each DataSource has the following methods:

Method name	**Return value**
Read current	A data structure containing the timestamp of the measurement and a list of floating point values with the measurement data.
Turn on	void
Turn off	void

Thanks to that interface, the DataLogger program can then run five identical threads called Data-Harvester, each operating on a different DataSource through the same interface. We will first describe the DataHarvester thread and then the five different data sources.

3.3.4.1 DataHarvester Thread

Each instance of the DataHarvester thread is started with one DataSource to operate on and an interval at which the source has to be read out. Upon startup it checks if the setup time of the source is larger than the polling interval. If this is the case, the source cannot be powered down in between measurements, so it is switched on immediately and then left running.

As a next step before going to its main loop the DataHarvester opens the logfile that is named according to the DataSource.

In its main loop, the thread first calculates how long it will have to wait until the next measurement is due. For this it simply takes the time passed since midnight, divides it by the polling interval and takes the remainder. This ensures that the period in between measurements will actually be kept over time and not drift as it may happen if we simply let it sleep for the interval time (or the interval minus the setup time). Also the sleep may be interrupted by a signal as we will see later on, and after handling the signal the thread will have to resume sleeping the remaining time.

While then sleeping for the previously calculated time, the thread my be woken up by a signal SIGUSR1 which we use for inter-thread communication. During normal operation this signal is blocked and only unblocked when sleeping. Such a signal could either represent a request to terminate the thread or to rotate the log file. Both of these requests are signaled by flags, which the thread checks after waking up.

If rotation of the log file is requested, the thread closes and re-opens the file and then goes back to sleep for the remaining amount of time.

Has the thread’s sleep not been interrupted by a signal, it is time to take a measurement, if the Power Manager allows it. Otherwise the main loop starts over. If required, the thread will now turn on the source and once again sleep for the setup time while the source is starting up. Then the measurement is taken and written to the log file, before the source is turned off again and the loop starts over.

3.3.4.2 Electrical Data

For the A/D converter on the BaseBoard there exists a kernel module called `ad77x8` that reads out all ten channels and imports them to the `proc` file system [22]. There the single channels are represented by files with the names `/proc/ad77x8/ain[1-10]` which return the voltage in milivolt. These are however not our final measurement values, but they have either been divided using voltage dividers for voltage measurements or converted from currents to voltages using shunt resistors and current sense amplifiers.

We have therefore for every channel a conversion factor which we multiply with the voltage reading to get the actual measurements in milivolts or miliamps.

Also we can calibrate each individual input. Therefore after the measurements are in the correct units, we add a calibration offset and once again multiply with a calibration factor.

To be included in frequent status messages, all electrical measurements are exported to a shared memory section where they can be read by the Message Manager.

3.3.4.3 Temperature and Humidity

For the temperature and humidity sensor we use no driver for the Gumstix platform exists. It uses a non-standard two-wire bus for communication which we implemented using two GPIO ports. The bus consists of one clock line, which is driven by the controller, i.e. the Gumstix and a bidirectional data line.

One measurement cycle starts with a special, *transmission start* sequence. It then requires sending one data word (8 bits) which specifies the command, such as ‘read temperature value’. After that the sensor needs a certain time to perform the measurement and when it is done the data can be read back on the data line (16 bits).

We implemented the protocol in a very simple kernel module called `sht1x`. It supports reading measurement values over the `proc` file system through the files `/proc/sht1x/humidity` and `/proc/sht1x/temperature`. Both return raw integer values that need to be converted to physical values using a polynomial of first or second degree as stated in the datasheet.

The module drives the bus clock at 50 kHz. During the high- and low-phases it uses `udelay()` which is a busy wait provided by the kernel. Synchronously to writing the clock the data output is also set or read, depending on the current bus direction. In between the writing of the command and the readout of the measurement, the module sleeps using `schedule_timeout()`, which allows scheduling to take over and execute other processes while we wait for the data. All in all, our module occupies the system for 320 microseconds in one stretch maximum when reading the measurement data.

The DataSource implementation for temperature and humidity measurement is very similar to the one discussed in Section 3.3.4.2. It will read the values from the `proc` file system and correct them using the constants in the data sheets. Also these measurements will be exported to a shared memory section where they can be read by the Message Manager.

3.3.4.4 GPS Positioning Data

The GPS module we use is assigned a device file with a name of the form `/dev/ttyACM[0-9]` by the driver when plugged in. Because the GSM modem is recognized as a device of the same category, it will receive a name of the same form and depending on which one was connected first, the GPS device can be `/dev/ttyACM0` or `/dev/ttyACM1`. To avoid opening the wrong device file, we wrote an `udev` rule [23], which creates a symlink called `/dev/ttyGPS` that points to whatever device file the GPS happened to be assigned.

The u-blox GPS module continuously sends standard NMEA messages which we can read directly through the device file. The GPS DataSource therefore opens this file and reads from it until it detects a GGA-message which contains time and location information. Then the content of the message is parsed, which is a comma separated list of alphanumerical values. When the location information is not present in the GPS module, either because there are not enough satelite in range or because the module has not been turned on for long enough, then the location fields in the message are left blank. This is recognized by our program which sets both longitude and latitude to 1000, which is an illegal value in either case and can be detected easily.

Should the module not be turned on when the readings are made or should it not have been recognized by the kernel for any reason, the measurement function throws an Exception. Exceptions are handled by the DataHarvester which will write them to the log file.

As with the two previous sources, the GPS measurements are exported to a shared memory section where they can be read by the Message Manager.

3.3.4.5 Pictures and Videos

The Logitech USB webcam we use is supported by the kernel module `uvcvideo` which makes it available through the v4l2 interface, the Linux standard for video devices. Thanks to this we can use standard tools to access the webcam. For both taking pictures and videos we run external programs that are already available and ported to the Gumstix.

To acquire still images from the camera, we use `uvccapture` which produces a jpeg image of 960 x 720 pixels. The image files are stored in the directory `/media/card/logs/pictures/` and named according to the date and time taken.

For recording video data we use ffmpeg. Currently it is set up to record videos of 5 seconds length with a resolution of 640 x 480 pixel. Video files are stored in `/media/card/logs/videos` and named according to the date and time taken.

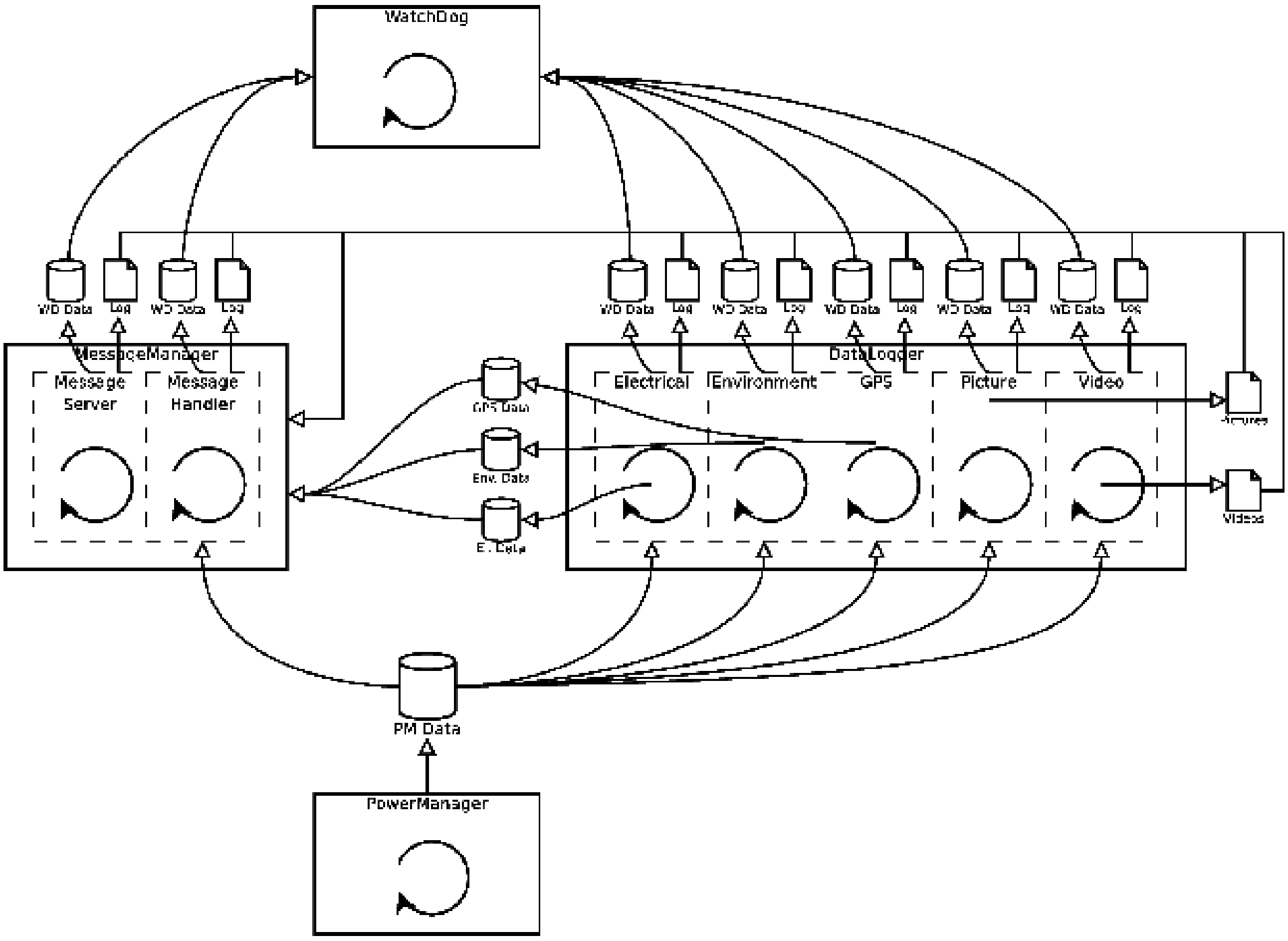

Figure 9: Diagram showing all processes as in Figure 3. Shown is all inter-process communication via shared memory (disc symbols) and files (file symbols).

4 Power Management

Up until now, power aware design was mentioned many times in this document and different parts of the communication platform explained so far are affected by it. In the hardware chapter we discussed several devices that can be switched on and off and thereby reduce consumption when unused. In the software section, we introduced the Power Manager process and how it can control the frequency of execution of tasks.

In this chapter we will formally state the problem setting that we are facing when doing power management on an autonomous sailboat. We will then show different possible solutions, i.e. algorithms, and present a metric to compare them. Finally the winning algorithm is discussed in detail and will fill the blank that we left in the last chapter about the inner workings of the Power Manager.

4.1 Problem Setting

The basic goal when doing power management for an autonomous sailing boat is to improve quality of service in the unpredictable environment of the open sea by using energy wisely and with foresight. What we just described in very vague terms shall be put into clear technical terms in this chapter. After presenting the assumptions we make about the complex environment we will show how we can adjust our energy consumption and what we mean by quality of service.

4.1.1 Assumptions

4.1.1.1 Battery

One of the basic building blocks of the power supply system is the battery. The SSA team decided to use a Lithium-Manganese battery with a capacity of 2400 Wh and a nominal voltage of 26 V. To put this number in relation with the consumption, running solely from the buttery the whole system of the boat could be powered for two and a half days. The charge and discharge curves (voltage against energy content) were measured and discussed in depth by Weber [4].

For our calculations we make two assumptions for the sake of simplification:

1. We assume ideal charge and discharge characteristics. That is, whatever the current state of charge of the battery every energy fed to it will be conserved 100% and can later be fully recovered by discharging. Also we assume that once the battery is fully loaded to its nominal capacity, it will destroy all additional energy that we try to charge it with.
2. We also assume that we are able to exactly determine the current energy content of the battery. The fact that we only measure the battery voltage and the big difference in the voltage/state-of-charge relationship between charging and discharging state make this seem like a very bold assumption that we will never be able to approach on the boat. Using the following approach however we can in fact determine the battery's energy content with useful accuracy: The voltage of the battery is measured early in the morning, before the solar panels generate any input. That way we can make sure the battery has been discharging for at least six hours and will now be very close to the ideal discharge curve. This gives us only one dependable measurement per day, but as we will see that is all we need for power management in our case.

4.1.1.2 Weather forecast

To make an informed decision in the area of power management we not only need to know the energy we have currently available in the battery, but also the power input we can expect in the future. Without such a prediction only rudimentary power management can be performed, because even the best algorithm can only react to past changes in energy input rather than anticipate future fluctuations. In other words, these systems have a dead time of one day.

One possibility suitable for an autonomous system is to compute a forecast using the input measured in the past few days. This requires knowledge about the temporal correlation of sunshine on subsequent days. Also, the quality of such predictions will never be competitive with calculations done by commercial weather services.

The variant we chose is to access a weather forecast provided by a commercial weather service. We have not yet evaluated and decided on one service provider and the data format and transfer protocol for the data transfer are not yet specified. On the software side however, we have all tools in place to implement a daily download of weather forecast.

Weather forecasts are often available for hours of sunshine and do not state watts per square meter of solar radiance, which we need to calculate the energy produced by the solar panels. We have however a heuristic to convert those two, which is explained in detail in Section 4.2.1.

In our calculations we therefore assume to have available a forecast on the power output of the solar panels for the next ten days.

4.1.2 Variability of Power Consumption

To have any impact on the system at all, power management needs some way to influence the current energy consumption. We split up the energy consuming devices in two parts: For one there is the communication platform with every device directly connected to it. Thanks to the efforts in integration, we have very fine grain control about which device is running and which is not.

The second part basically includes all devices on board responsible for sailing. These are mainly the sailing computer, all its sensors and actors. That part we simply call 'boat' as we have to treat it as one inseparable system.

4.1.2.1 Communication Platform Consumption

The Communication Platform has a static power consumption of 1W. This energy is consumed by the BaseBaord and the Gumstix computer which cannot be switched off. Any approach trying to reduce this base consumption would require additional hardware that can be programmed by the platform to shut it down and power it up again at a later time. As the platform is presented here, this is not possible and the system is assumed to be always running and consuming power.

In addition to that base consumption, which includes any power consumed by the software presented earlier, we identified a number of tasks which also consume power but that can be run less frequently or not at all should this be required by the power management. We can thus reduce power consumption by decreasing the number of executions of single tasks and thereby reducing the quality of service.

Already in Section Fehler: Referenz nicht gefunden we defined the notion of task and schedule. We will now repeat the definition with more focus on aspects relevant to the context of this chapter.

Task:

A task is an operation that requires more energy than the base consumption of the Gumstix. Every task has an upper and a lower bound on the number of times it can be executed per day. The power management is free to adjust the number of executions within these limits. Moreover, for every task its energy consumption is known and it has a unique priority indicating its relative importance. Tasks with lower priorities are more important.

Schedule:

A schedule assigns to every task in the system a number of times it is actually executed that complies with the limits for this task. Therefore we can calculate the power consumption of a task by adding the consumption of all tasks multiplied by the number of times they are to be executed.

We now present the list of all tasks that are generated by the different software processes running on the communication platform.

			Number of executions per day	
Task	**priority**	**cost per execution (J)**	**max**	**min**
take picture	7	9	2	0
shoot movie	10	100	1	0
measure GPS position	3	73	48	1
poll command message (3 min on)	1	282	24	6
send status message (3 min on)	2	282	2	1
leave modem on for 1 hour	11	5220	24	0
open data connection (3 min on)	4	282	10	0
receive weather forecast (100kb)	5	120	1	0
send home a picture (100kb)	9	120	1	0
send home a video (500kb)	12	500	1	0
send home current log files (700kb)	6	700	1	0
send home past log files (700kb)	8	700	10	0

Using the maximal and minimal number of executions and the energy consumption of each task, we can quickly calculate a maximal and minimal energy consumption of 147'494 and 2'047 Joule respectively. We also see a total of maximal 125 task executions per day and roughly 530 million possible combinations to construct a valid schedule. It is easy to see that using an appropriate combination of task, we could construct a schedule with almo1st any consumption in between the two extremes we calculated above.

We therefore split up the problem of finding an execution schedule for the communication platform in two parts:

- In a first step we assume the consumption to be variable continuously between the limits consumption of the maximal and minimal schedule plus the base consumption. The result of this first calculation is the amount of energy that the platform may use in one day.
- In a second step we construct a schedule that respects the budget calculated in the first part and executes as many tasks with high (i.e. small) priorities.

4.1.2.2 Boat Consumption

Much larger than the energy consumption of the communication platform is that of the rest of the boat. Because of unfortunate timing however, we were not able to obtain realistic measurements during the duration of this project. Once development and integration of the hardware was advanced enough to allow current measurement for all consumers on the boat, not one successful day of testing could be completed before the project was stalled. The numbers presented here are therefore estimations based on the consumption of the single parts and have to be refined and updated before the boat will start to the Microtransat competition.

Altogether, the SSA team states a system consumption of 40W. This is a mean value and the real consumption over time will not be continuous but spike during maneuvers such as jibing or tacking. An advanced model of consumption might take into account the number of maneuvers that is expected by the navigation software running on the sailing PC.

In our calculations however, the boat consumption is fixed at 40W and cannot be changed while the system is running. The only way to have any influence, is to shut down the sailing system completely, which has to be done cooperatively. Before the sailing algorithms can be stopped and the motors deactivated, the sail and the rudders have to be rotated into a safe position that will minimize drift.

Having this in mind, we agreed with the SSA team on the following approach to decrease power consumption of the boat if needed: We vary the percentage of time the boat is in powered, operating state, basically changing the duty cycle in a very coarse fashion. We may reduce the on-time of the boat in one hour steps over a period of four hours. This leaves us with a total of five steps where the boat is running 0%, 25%, 50%, 75% and 100% of the time respectively. With the duty cycle the power consumption will of course be reduced as well.

Figure 10 shows the boat state over one day with a duty cycle of 75%. This means that the boat is operational for three hours and then turned off for the next hour, repeated over the whole day.

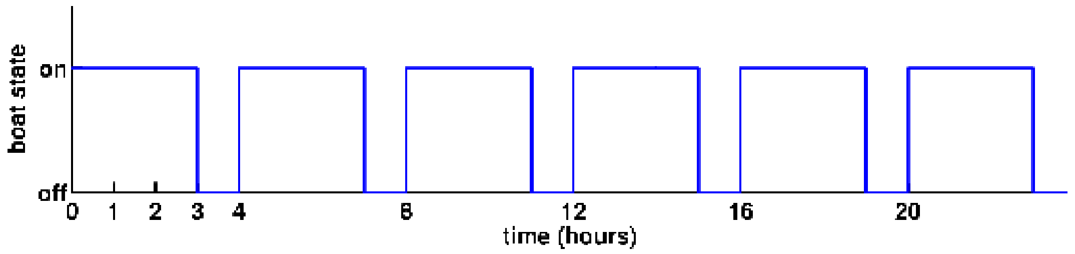

Figure 10: Example of boat state

4.1.3 Objective

To solve our problem of power management, we need an algorithm which can tell us how much energy we can use for both our main systems. Different approaches could be possible here:

- The algorithm could be run continuously, monitoring the energy situation and updating the communication platform schedule and boat power state on the fly.
- The algorithm could be run several times to reassess the energy situation and update the communication platform schedule and boat power state.
- The algorithm could be run once a day, assess the energy situation and issue a schedule for the communication platform and state the boat power state for the next 24 hours.

We decided on the the third option for the following reasons:

- During daytime there is no information about the current energy input. This is a restriction imposed by the placement of the current and voltage measurement. Only the battery voltage can be measured which will not allow reliable estimations on the battery content during the day.
- During early morning hours the conditions to estimate the battery's energy content are ideal as the voltage will have relaxed to the discharge curve [4].
- The forecasts for future energy input have a resolution of one day. Fluctuations during the day are not know in advance and thus there is no additional information that would justify running the algorithm more than once a day.

We are looking for an algorithm that, given

– the current energy content of the battery and

– the expected energy input of the solar panels for the next ten days,

considering all assumptions in Section 4.1.1, computes an energy budget

– for the communication platform in accordance with Section 4.1.2.1 and

– for the boat in accordance with Section 4.1.2.2.

There are, of course, many algorithms that fulfil these requirements, including a simplistic rule to just run everything with the least consumption possible. However, we are looking for a an algorithm that puts the boat in low power modes as little as possible. We now present a metric that translates this quality into a number and so lets us rate algorithms and look for the best.

4.1.4 Metric

4.1.4.1 Definitions

Different algorithms can only be compared if they run under equal preconditions. Because there is no way we can test different algorithms on the boat in identical conditions, and because there were hardly any testing days available during this project, we have to do the comparison by simulation. There we can present the algorithms with identical scenarios.

Scenario:

A scenario of a given length in days specifies for each day the energy input of the solar panels and the forecast for the next ten days for the same energy input.

In the simulation it is possible to feed the algorithm an exact forecast that is identical with the future input, or we can simulate a statistical error that adds uncertainty.

Given a scenario and an algorithm, we can now perform a simple simulation. The only variable it uses is the energy content of the battery:

1. Choose an initial battery content. This has less influence the longer the scenario is. Our assumption is that we start with a full battery that is 90% charged.
2. For every day in the scenario do:
    1. Run the algorithm with the current battery content and the forecast of the current day as inputs.
    2. Add the actual energy input of the current day to the battery content and subtract the calculated consumptions of communication platform and boat.
    3. If the state of the battery is over 100% set it back to 100%. If it is lower than 0% issue an error.
    4. Go to step 1.

In addition, the simulation will of course keep track of the history of the energy budgets and the battery state, so they can be evaluated later.

Our simulations showed that the discerning factor between different algorithms is the planning of the boat energy budget. Because the minimal consumption of the communication platform (88.4 kJ/day) is about ten times smaller than the minimal consumption of the boat (864 kJ/day) it is mainly the boat that needs to save power when input is low and the communication platform can always run on a high budget.

We therefore base our metric on the history of the boat energy budget in a simulation run *S*.

$$\begin{aligned} &b_i(S): \text{"energy budget step of the boat on the day } i \text{ in the simulation } S\text{"} \\ &d(S): \text{duration of simulation } S \text{ in days} \\ &b_i(S) \in \{0..s\} \quad \forall i \in \{0..d(S)\} \end{aligned} \tag{1}$$

Budget steps range from 0 to s, where step 0 means the boat is shut off completely and step s means the boat is running continuously. From Section 4.1.2.2 we know that s is 4.

We now define the following histogram function:

$$h_i(S) := |\{b_j(S) | b_j(S) = i\}| \tag{2}$$

That is, $h_i(S)$ equals the number of times the boat was running on budget step i during simulation *S*. The metric is now defined as the following weighted sum of the histogram function:

$$m(S) := \sum_{i=0}^{s} h_i(S) \cdot d(S)^{s-i} \tag{3}$$

We use this metric to compare simulation runs of the same length. Because $d(S)$ is raised to the power of s-i, the metric grows faster than linearly with the duration of *S* and comparisons of simulation runs of different lengths are therefore meaningless.

4.1.4.2 Properties

We will now present four properties of the metric that relate differences between simulation runs to the difference between their respective metrics. As mentioned before, whenever two simulation runs are compared, they are assumed to be of identical duration.

The first two properties are presented in words first:

1. If one of two simulation runs has a lower minimal boat budget, its metric is higher.
2. If two simulation runs have the the same minimal boat budget, the one with more days in the minimal budget has a higher metric.

That is, a better simulation run has a lower metric. We will now proof these properties, using two simulation runs S_1 and S_2 of identical duration d. The first property presumes that one simulation run has a lower minimal boat budget than the simulation run it is compared to:

$$\begin{gathered} m_1 = min(\{b_i(S_1)\}) \\ m_2 = min(\{b_i(S_2)\}) \\ m_1 < m_2 \end{gathered} \tag{4}$$

Here we use the notation $\{b_i(S_1)\}$ as a short form for $\{b_i(S_1) \mid i \in \{0..s\}\}$ and we remember that $h_i(S_1) = 0$ for all $i < m_1$. We now calculate a lower bound for $m(S_1)$ by neglecting every part of the sum except for the one with the lowest index $i = m$:

$$\begin{aligned} m(S_1) = \sum_{i=0}^{s} h_i(S_1) \cdot d(S_1)^{s-i} = \sum_{i=m_1}^{s} h_i(S_1) \cdot d^{s-i} \quad &> \quad h_{m_1}(S_1) \cdot d^{s-m_1} \\ &> \quad d^{s-m_1} \end{aligned} \tag{5}$$

Similarly, we also find an upper bound for $m(S_2)$. The sum has its maximal value if $h_m(S_2)$ equals d. If it is lower than d, a higher budget step will be used at least one day, which has a lower factor d^{s-i} with $i > m_2$.

$$\begin{aligned} m(S_2) = \sum_{i=0}^{s} h_i(S_2) \cdot d(S_2)^{s-i} = \sum_{i=m_2}^{s} h_i(S_2) \cdot d^{s-i} \quad &\leq \quad d \cdot d^{s-m_2} \\ &\leq \quad d^{s-(m_2-1)} \\ &\leq \quad d^{s-m_1} \qquad \textit{(using } m_1 \leq m_2 - 1\textit{)} \end{aligned} \tag{6}$$

Now we can put $m(S_1)$ and $m(S_2)$ in relation:

$$m(S_1) > d^{s-m_1} \geq m(S_2) \tag{7}$$

Which leads us to the mathematical formulation of the first property:

$$min(\{b_i(S_1)\}) < min(\{b_i(S_2)\}) \Rightarrow m(S_1) > m(S_2) \tag{8}$$

We now proof the second property using two different simulation runs S_1 and S_2 with the following premise:

$$\begin{gathered} m = min(\{b_i(S_1)\}) = min(\{b_i(S_2)\}) \\ h_m(S_1) > h_m(S_2) \end{gathered} \tag{9}$$

Similar to the proof of the first property, we find a lower bound for $m(S_1)$ and an upper bound for $m(S_2)$:

$$m(S_1) = \sum_{i=m}^{s} h_i(S_1) \cdot d^{s-i} > h_m(S_1) \cdot d^{s-m} \tag{10}$$

$$\begin{aligned} m(S_2) = \sum_{i=m}^{s} h_i(S_2) \cdot d^{s-i} &\le h_m(S_2) \cdot d^{s-m} + d \cdot d^{s-m-1} && \textit{(using } h_{m+1}(S_2) \le d \textit{)} \\ &\le \left(h_m(S_2)+1\right) \cdot d^{s-m} \\ &\le h_m(S_1) \cdot d^{s-m} && \textit{(using } h_m(S_1) \ge h_m(S_2)+1 \textit{)} \end{aligned} \tag{11}$$

Putting together equations (10) and (11) we get

$$m(S_1) > h_m(S_1) \cdot d^{s-m} \ge m(S_2) \quad , \tag{12}$$

proving the second property.

$$\left.\begin{array}{c} m = min(\{b_i(S_1)\}) = min(\{b_i(S_2)\}) \\ h_m(S_1) > h_m(S_2) \end{array}\right\} \Rightarrow m(S_1) > m(S_2) \tag{13}$$

As a generalization of the two properties that have just been proved, we add a third one in words as well as in a mathematical formulation:

3. If simulation run S_1 has a bigger metric than S_2, there exists a budget step for which the histogram value of S_1 is larger than the one of S_2 while the histogram entries for lower steps are identical.

$$m(S_1) > m(S_2) \Leftrightarrow \exists\, n \in \{0..s\}: \begin{array}{l} h_i(S_1) = h_i(S_2)\ \forall\ i<n \\ h_n(S_1) > h_n(S_2) \end{array} \tag{14}$$

For the proof we first state that n is the lowest budget step for which the histograms of the two simulation runs differ:

$$n = min(\{ i \mid h_i(S_1) \neq h_i(S_2)\}) \tag{15}$$

Now we split up the sum that constitutes the metric in two parts: one where the index runs from zero to below n, and the rest. The first part, which we call k, is identical for both simulation runs, while the second one differs.

$$k = \sum_{i=0}^{n-1} h_i(S_1) \cdot d(S_1)^{s-i} = \sum_{i=0}^{n-1} h_i(S_2) \cdot d(S_2)^{s-i} \tag{16}$$

$$\begin{aligned} m(S_1) &= k + \sum_{i=n}^{s} h_i(S_1) \cdot d(S_1)^{s-i} \\ m(S_2) &= k + \sum_{i=n}^{s} h_i(S_2) \cdot d(S_2)^{s-i} \end{aligned} \tag{17}$$

We first proof one direction of the property, that is

$$m(S_1) > m(S_2) \Leftarrow \exists\, n \in \{0..s\}: \begin{array}{l} h_i(S_1) = h_i(S_2)\ \forall\ i<n \\ h_n(S_1) > h_n(S_2) \end{array} \quad . \tag{18}$$

We do this using the same technique as for the second property by finding a lower bound for $m(S_1)$ and an upper bound for $m(S_2)$.

$$m(S_1) \;=\; k+\sum_{i=n}^{s} h_i(S_1)\cdot d(S_1)^{s-i} \;>\; k+h_n(S_1)\cdot d^{s-n} \tag{19}$$

$$\begin{aligned} m(S_2) \;=\; k+\sum_{i=n}^{s} h_i(S_2)\cdot d(S_2)^{s-i} \;&\le\; k+h_n(S_2)\cdot d^{s-n}+d\cdot d^{s-n-1} \\ &<\; k+\left(h_n(S_2)+1\right)\cdot d^{s-n} \\ &<\; k+h_n(S_1)\cdot d^{s-n} \end{aligned} \tag{20}$$

Which finishes the proof for the first direction:

$$m(S_1) \;>\; k+h_n(S_1)\cdot d^{s-n} \;>\; m(S_2) \tag{21}$$

To proof the second direction, we need to find an *n* that fulfills the right hand side of the property, given the left hand side is true. We already stated that value for *n* in (15) which obviously satisfies

$$h_i(S_1)=h_i(S_2)\ \forall\ i<n \tag{22}$$

and also

$$h_n(S_1)\neq h_n(S_2)\quad . \tag{23}$$

By contradiction, we can now that show that $h_n(S_1)$ has to be bigger than $h_n(S_2)$: Assuming the opposite is the case ($h_n(S_1)$ is smaller than $h_n(S_2)$), the right hand side would imply $m(S_1)$ to be smaller than $m(S_2)$, which contradicts the left hand side of (14). Thus we can conclude that the third property holds in both directions.

Lastly we state a fourth property for simulation runs with identical metrics.

4. If two algorithms have the same metric their histograms are identical.

$$m(S_1)=m(S_2)\ \Leftrightarrow\ h_i(S_1)=h_i(S_2)\ \forall\ i\in\{0..s\} \tag{24}$$

This equation is a direct implication of (14). For the sake of clarity we will repeat the equation with S_1 and S_2 in both positions:

$$m(S_1)>m(S_2)\ \Leftrightarrow\ \exists\ n\in\{0..s\}\colon\ \begin{array}{l} h_i(S_1)=h_i(S_2)\ \forall\ i<n \\ h_n(S_1)>h_n(S_2) \end{array} \tag{14}$$

$$m(S_2)>m(S_1)\ \Leftrightarrow\ \exists\ n\in\{0..s\}\colon\ \begin{array}{l} h_i(S_2)=h_i(S1)\ \forall\ i<n \\ h_n(S_2)>h_n(S_1) \end{array} \tag{25}$$

We can now conclude, that if S_1 and S_2 fulfill neither (14) nor (25) on the left hand side, then they must fulfill neither of the statements on the right hand side as well. On the left this implicates that the metrics are identical, and on the right that the histograms are as well.

4.1.4.3 Conclusion

As we have seen, simulation runs where the boat spends more days in a lower budget step have a higher metric, therefore *the better an algorithm performs in a certain scenario the lower is its metric*.

We can now formulate the problem setting for the next sub-chapters: We are looking for an algorithm as defined in Section 4.1.3 that has the lowest metric in any scenario. If such an algorithm doesn't exist (that is if one or another algorithm performs better depending on the scenario) we are looking for the algorithm that performs best in scenarios typical for the course the boat will take in the Microtransat challenge.

4.2 Method

To find the best algorithm as defined in the last chapter, we formulated several candidates. Most were based on MILPs, but also some heuristic ones were tried. We will present the most promising ones in this chapter.

All algorithms were compared in simulations carried out in Matlab. As input for the simulations we used historical data from representative sites and generated data.

4.2.1 Simulation Data

For the absolute values of energy input from the solar panels we rely on the Bachelor thesis of Jürg Weber who devised the power supply of the sailing boat [4]. As a part of his work Weber conducted detailed simulations to find the power input of the solar panels. The simulations were performed for positions near Ireland and near the Equator and considered factors such as the angle of sunshine, the angle of the solar panels, the albedo factor of the water and the efficiency of the solar panels. What his calculations do not include however is the influence of clouds and generally bad weather.

We use the numerical results obtained by Weber for perfect conditions to calibrate our data.

4.2.1.1 Historical Data

For the relevant sites near Ireland and near the equator records of solar radiation power for passed years are not available. What is available on the other hand are records of hours of sunshine per day. We used historical data of that form available from Weather Online [24] that covers the same time period the boat would be sailing during the Microtransat from last years.

We then used the solar energy input per day as calculated by Weber for the respective location and assumed this to be correct for days with maximal number of sunshine hours (10). For all other days the energy input was scaled down linearly with the hours of sunshine. Consequentially for days with no sunshine the energy input was zero.

This way we constructed realistic scenarios with data from both Ireland and Senegal. The first is where the boat will start and the second one is at the same latitude as the second part of the course.

4.2.1.2 Statistical Data

To have more and different data available than the one generated from historical records we specified a very simple statistical model for solar input power on the ocean. Radiation power is assumed to be normally distributed with that fixed mean over one simulation run and a standard deviation of 1.5 times the mean.

We used mean values between 0 and 6.8 MJ per day which covers both usual inputs in Ireland (2 MJ/day) and Senegal (6 MJ). Generated data is particularly useful for to run long simulations over more than 1000 days where the randomness of the input data is evened out.

Next to the actual energy input a scenario also includes the forecast the algorithm is presented with each day. We tried two approaches for generating forecast values: In one case the forecast was completely exact, basically a copy of the actual input that is going to come. In another case we added uncertainty to the forecast by giving it a normal distribution with a mean value of the actual input and a standard deviation of 0.68 MJ for the first day that rises linearly to 6.8 MJ for the tenth and last day.

4.2.2 Algorithms tested

When we started developing our first approaches to power management, it became clear quickly that if we wanted to go beyond simple heuristics the planning algorithm needs to solve an optimization problem. Integer Linear Programming (ILP) presented itself as a convenient solution because all mathematical relations we use are linear and also ILP libraries are readily available and easy to port to the Gumstix platform.

We will now present a very simple, heuristic algorithm and then a number of ILP based algorithms.

4.2.2.1 Heuristic Algorithm

As a first approach we implemented a simple heuristics. As it turned out, its performance was superior to some vastly more complex MILP algorithms but could not compete with the best.

1. Build the sum of the current battery content and the input predicted for the next ten days and divide by ten. This is the energy budget for today.
2. If the previously calculated budget is bigger than the battery content and the prediction for todays input, shrink it to that value.
3. If the budget is smaller than the minimal consumption of the communication platform, set both energy budgets to zero.
4. If the budget is within both limits of the communication platform consumption, set the latter to the former and the boat budget to zero.
5. Otherwise set the budget for the communication platform to its maximum and from whatever is left of the budget fill as many steps of the boat budget.

4.2.2.2 Mixed Integer Linear Program based Algorithms

Next we present the different MILP algorithms that we developed. The names are meant to be descriptive but are mainly used to keep them apart in comparisons. The first four algorithms are all slight variations on the same theme and therefore share most of their equations while the last two use different approaches.

Vanilla

The first and most generic MILP program we developed is a good starting point to introduce all the variables and the main concept that all MILP based algorithms share, even if it does not perform very well in comparison.

The concept of all six algorithms in this section is to simulate the whole period for which a forecast is available. That is, we calculate feasible energy budgets for the next ten days that respect the physical limits, i.e. battery capacity. We can then formulate optimization criteria based on the budget of the complete period and not just the current day.

To formulate the MILP, we need to introduce several variables:

$$\begin{aligned}
&b_i\text{: boat budget step for day } i\\
&b_s\text{: boat budget step size}\\
&c_i\text{: communication platform budget for day } i\\
&I_i\text{: predicted energy input at day } i\\
&C_1\text{: initial charge of battery}\\
&n\text{: duration of forecast period in days}
\end{aligned} \tag{26}$$

As we will see, the variables b_i and c_i don't provide enough degrees of freedom to simulate one period. The physical limits mentioned before dictate that the battery can not be charged over 100%. In situations where the energy input is larger than what we can maximally use and than the battery can absorb the following MILP will fail if it weren't for the following variable:

$$w_i\text{: energy wasted at day } i \tag{27}$$

This variable will hold all energy that could not be fit into the battery on really sunny days.

We now state the limits of the variables to be determined by the MILP.

$$\begin{aligned}
&b_i \in \mathbb{Z},\ c_i \in \mathbb{R}\\
&\left.\begin{aligned} 0 \le b_i, c_i\\ b_i \le 4\\ c_i \le c_{max}\end{aligned}\right\} \ \forall i \in \{0..n\}
\end{aligned} \tag{28}$$

$$\begin{aligned}
&w_i \in \mathbb{R}\\
&0 \le w_i \ \ \forall i \in \{0..n\}
\end{aligned} \tag{29}$$

We omit a lower limit for the communication platform budget because otherwise the problem might not have a solution. If the resulting budget should be below the minimum, we will set it to zero after the MILP has been solved.

We can now present the restriction that makes sure the upper and lower limits of the battery capacity are respected. The sum calculates the battery's energy content after each day and restricts it to the allowed range. If it weren't for the parameter w_i, this restriction couldn't be satisfied if I_i is too large.

$$0 \ < \ C_1 + \sum_{i=1}^{k} \left(I_i - b_i \cdot b_s - c_i - w_i\right) \ < \ C_{max} \ \ \forall k \in \{1..n\} \tag{30}$$

All that's missing now is the function we want to optimize:

$$f = \sum_{i=1}^{n} \left(b_i \cdot b_s + c_i\right) \tag{31}$$

To summarize, the complete ILP:

Vanilla:

Maximize the function (31) while respecting the restrictions (28), (29) and (30).

End Limit

One concern with the Vanilla algorithm especially with short forecast periods was the fact that it would try to maximize the energy budgets and leave the battery completely empty at the end. We therefore added another restriction, which ensured that the battery would be left with some charge at the end of the forecast period.

$$C_2 : \text{desired battery charge at the end of forecast period} \tag{32}$$

$$C_1 + \sum_{i=1}^{n} \left(I_i - b_i \cdot b_s - c_i - w_i \right) \ \geq \ C_2 \tag{33}$$

All other conditions are identical.

EndLimit:

Maximize the function (31) while respecting the restrictions (28), (29), (30) and (33).

Smooth

Another concern with the Vanilla problem is that it creates b_i successions that look very ridged, i.e. the budget varies wildly from one day to the next while we would prefer smooth transitions from one budget step to the next. This variant thus adds another restriction that enforces that the boat budget may not change by more than one step per day.

$$-1 < b_{(i+1)} - b_i < 1 \quad \forall i \in \{1..n-1\} \tag{34}$$

Again, the rest of the conditions are identical with Vanilla:

Smooth:

Maximize the function (31) while respecting the restrictions (28), (29), (30) and (34).

Minimum

This variant also tries to avoid wildly varying budgets and, in particular, unnecessarily low budgets. It doesn't try to maximize the overall consumption like the variants before did but rather the minimum budget. This is the first approach which actually tries to imitate the metric function in its optimization goal.

To calculate the minimum budget, we have to introduce a new variable:

$$v < b_i \cdot b_s + c_i \, \forall i \in \{1..n\} \tag{35}$$

Also, the optimization function has changed:

$$f = v \tag{36}$$

Minimum:

Maximize the function (36) while respecting the restrictions (28), (29), (30) and (35).

Minimum+Today

The last algorithm does try to maximize the minimum budget over the duration of the forecast period. It does not, however, try to improve the budget on days where it could be higher than the minimum. Lots of energy could go unused if a forecast period requires the boat to shut off completely for one day. Then the energy budget for the other days is not optimized because it has no influence on the objective value.

As a middle way between the Vanilla and the Minimum approach we try to maximize the sum of the budget of the first day and of the minimal budget over the whole period:

$$f = v + b_1 + c_1 \tag{37}$$

Minimum:

Maximize the function (37) while respecting the restrictions (28), (29), (30) and (35).

Mean

A very similar effect as with the previous effect as with variant can be achieved by using less variables. Instead of varying the budgets for each day of the simulation period and then maximize their minimum, we can simply set the budget of all days equal and just use one variable.

$$\begin{gathered} b: \text{boat budget step for all days} \\ c: \text{communication platform budget for all days} \end{gathered} \tag{38}$$

The limits of the budgets stay the same:

$$\begin{gathered} b \in \mathbb{Z}, c \in \mathbb{R} \\ 0 \le b, c \\ b \le 4 \\ c \le c_{max} \end{gathered} \tag{39}$$

The main restriction also looks very similar:

$$0 \ < \ C_1 + \sum_{i=1}^{k} \left(I_i - b \cdot b_s - c - w_i \right) \ < \ C_{max} \quad \forall k \in \{1..n\} \tag{40}$$

Again, we maximize the sum of the budgets:

$$f = b \cdot b_s + c \tag{41}$$

Mean:

Maximize the function (41) while respecting the restrictions (39), (29) and (40).

Cost

Lastly, we present an algorithm that uses a very different approach in its optimization function. It actually uses a cost function completely analogous to the metric presented in Section 4.1.4. To implement a basically nonlinear cost function, we need to adapt the parametrization. Instead of one variable that has the value of the boat budget step for one day, we now have indicator variables for every single step. Exactly one of those indicator variables equals one and those of the remaining steps are zero. The variable carrying the value one indicates the step at which the boat is running.

$$b_{i,j}: \text{indicator if boat budget is at step } j \text{ on day } i \tag{42}$$

$$\begin{gathered} b_{i,j} \in \mathbb{Z} \\ 0 \le b_{i,j} \le 1 \\ \sum_{j=0}^{s} b_{i,j} = 1 \quad \forall i = 1..n \end{gathered} \tag{43}$$

Naturally, the constraints have to be adapted to the indicator variables:

$$0 \ < \ C_1 + \sum_{i=1}^{k} (I_i - \sum_{j=0}^{s} (b_{i,j} \cdot b_s \cdot j) - c_i - w_i) \ < \ C_{max} \quad \forall k = 1..n \tag{44}$$

As stated before, the optimization function resembles the metric used to rate algorithms. It includes the sum of the communication platform budgets minus

$$f = \sum_{i=1}^{n} (c_i - \sum_{j=0}^{s} (b_{i,j} \cdot n^{s-j})) \tag{45}$$

Cost:

Maximize the function (45) while respecting the restrictions (28), (29), (43) and (44).

4.3 Results

4.3.1 Historical Data

Historical data proofed not very helpful to set apart different algorithms. For one the duration of the scenarios is to short to use randomized forecasts. The effect of the randomized distorts the results because they can not be evened out over a big number of days.

Then the solar input in Senegal proved to be big enough to let the boat run on the maximal energy budget each day. Thus all algorithms performed identically well.

For the sake of completeness we show the results of a simulation run with data from Ireland and an exact forecast. The metric is plotted on a logarithmic axis because it is an exponentially weighted sum. We see that the algorithms heuristic, minimum+today, mean and cost perform equally well. The length of the simulation is simply not long enough to bring out any differences in between those four.

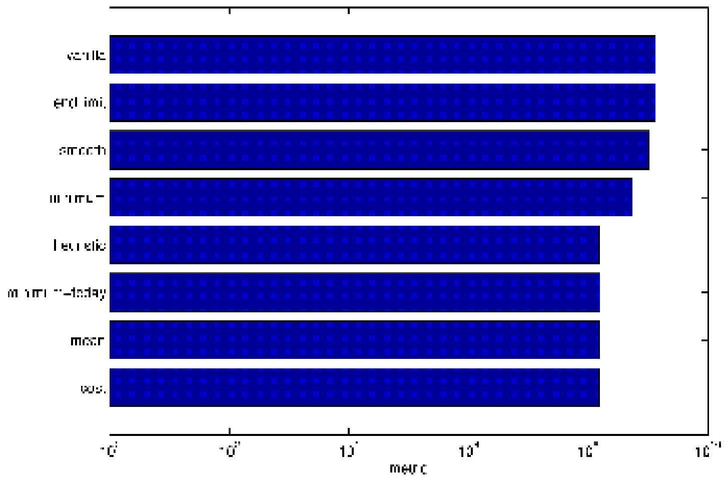

Figure 11: Algorithm comparison with historical data from Ireland using an exact forecast.

4.3.2 Statistical Data

Generated data was is much more helpful in bringing out differences between the different algorithms. We used both scenarios with exact and randomized forecast and varied the mean input power. Next we show plots of simulation runs over 10'000 days where the mean input power was varied in ten steps from 0.68 MJ/day to 6.8 MJ/day.

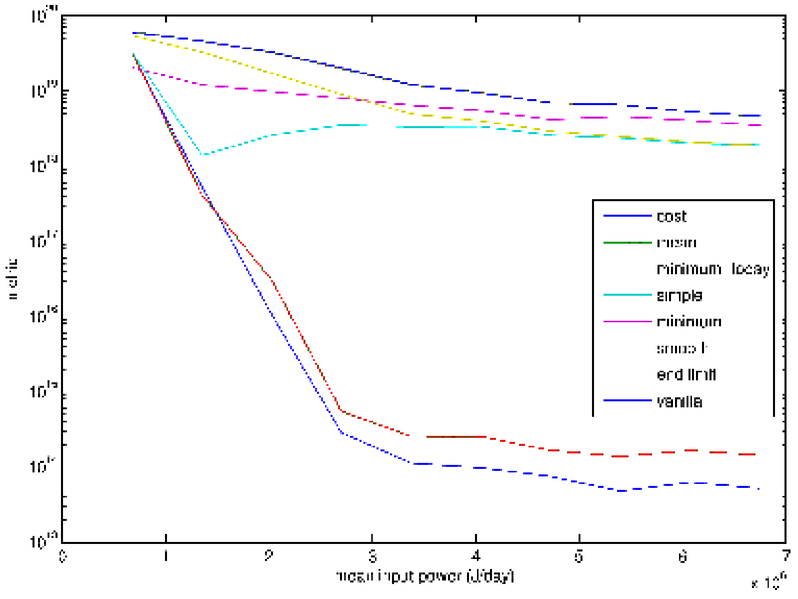

Figure 12: Simulation runs over 10'000 days with exact weather forecast, varying input power

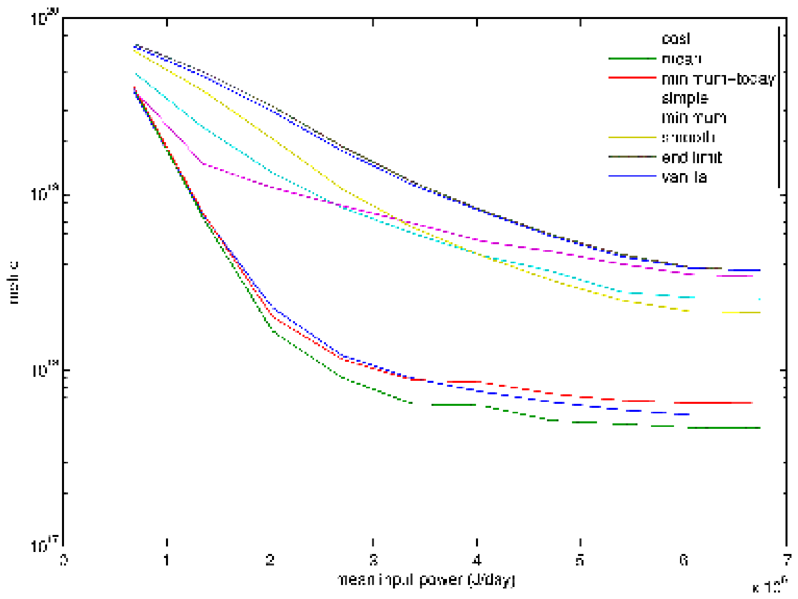

Figure 13: Simulation runs over 10'000 days with randomized weather forecast, varying input power. Mean over 4 runs.

4.4 Discussion

In both simulations with exact and with randomized forecast we can divide the algorithm in the same two groups according to their performance. The three best performing algorithms are clearly cost, minimum+today and mean (not easy to spot in the first plot). All three of them have an objective function which approximates the metric by which the algorithms are graded. This obviously lets the outperform the vanilla algorithm and its variants. A big flaw in the objective to maximize the sum of all budget over all days is that it values solutions identically which have drastically different metrics. For example if the vanilla algorithm is presented with two solutions for b1 and b2 where the first is $b_1 = b_2 = 2$ and the second is $b_1 = 0$, $b_2 = 4$ both will have the same effect on the objective function and which one is chosen is more or less random. For the metric on the other hand, the first variant is far better than the second.

We can now have a closer look at the differences between the algorithms in the better performing group. In the simulation run where the algorithms were fed a weather forecast that was 100% correct the cost algorithm outperformed the other members of the group except at very low energy input. When the predictions are randomized however, the mean algorithm takes an equally clear leading position.

The reason once again lies in the objective functions: The cost function is the only algorithm that tries to maximize a function very similar to the metric. If the predictions are correct it can therefore take optimal, concise decisions on the energy consumption. If however the information about future energy input are imprecise the cost algorithm's precise calculations are in vain. In such a situation more conservative algorithms like mean shine. What it looses when predictions are accurate because it doesn't optimize to the last bit it wins when there's uncertainty. The more conservative results help the algorithm not to spend too much energy based on wrong information and then paying for it with an empty battery.

Seeing as real-world weather forecasts will newer be perfectly accurate we prefer the mean algorithm for our application. It outperforms every other algorithm in the presence of insecurity in the forecast and also with a perfect forecast it is very competitive, only being dominated by an algorithm with almost seven times as many variables. Where the Gumstix platform computationally less powerful the much smaller complexity would be another argument in favor of the mean algorithm.

5 Conclusion

We will now summarize the accomplishments of the work presented in this thesis. Also we will point out some possibilities of future work in the scope of this project.

The task was to build a “Power Aware Communication Platform for an Autonomous Sailboat” from hardware to software. After deciding on the building blocks of the hardware, which was heavily influenced by previous project the platform was assembled in one plastic box and interconnected with the remaining system on the Avalon sailing boat. In particular it disposes of two means of wireless communication, one of which is a Iridium satellite modem.

All devices belonging to the communication platform and also the rest of the boat were reviewed and made power-switchable if possible and meaningful. Power consumption measurements were performed on all parts under our control and the energy consumption of certain tasks was identified.

A self-monitoring software system was designed and implemented to meet the diverse functional requirements. Communication service is provided to the sailing computer over a TCP interface. It can now e.g. send status messages without knowing over what data link they are sent and if transmission has to be retried because of failures. A set of data sources periodically collects measurements which are sent back to a control center together with diagnostic messages of the whole boat.

Every part of the software system respects limits set by the power management that has very fine-grain control about the energy consumption of the communication platform. Also the consumption of the remaining system, namely the boat can be adjusted, but only collaboratively and in five coarse steps.

To maximize the minimal level of service the different systems on board can offer a power management concept was devised. It assumes that a forecast on the energy production of the solar panel is available for the next ten days. Using this and a few other abstractions we defined a simulation method and a metric with which we could assess the performance of different algorithms in different scenarios.

Subsequently several algorithms were implemented, simulated and compared. In scenarios with uncertainty in the weather forecast as in the real world, one simple MILP based algorithm stood out and was thus selected and implemented for the power management on the boat.

5.1 Future work

At the end of this project possibilities for future work present themselves in different areas. Some are simple feature requests that could not yet be implemented like integrating the camera on the boat or selecting a suitable service for weather forecasts and implementing the required communication.

But there are also possibilities to further improve the work done in the more theoretical part of power management. For one the simulation could be made more realistic by using a model of a real battery with charge and discharge losses. Also the time axis could be handled more finely by choosing smaller steps than one day for the simulation.

For generating simulation data more complex models can be investigated to have more realistic scenarios. Especially model used for the uncertainty in weather forecasts could have an influence on algorithm performance. Maybe it is possible to use historical data together with actual forecasts from that time.

6 Appendix

6.1 References

[1] Colin Sauzé and Mark Neale: *Design Considerations for Sailing Robots Performing Long Term Autonomous Oceanography*. In Proceedings of The International Robotic Sailing Conference, Pages 21-29, May 2008

[2] Gion-Andri Büsser, *Design and Implementation of a Navigation Algorithm for an Autonomous Sailboat*, Swiss Federal Institute of Technology (ETH) Zürich, 2009

[3] Hendrik Erckens, *Design of a Trajectory Following Controller for an Autonomous Sailboat*, Swiss Federal Institute of Technology (ETH) Zürich, 2009

[4] Jürg Weber, *Design of a Power Supply for an Autonomous Sailboat*, Swiss Federal Institute of Technology (ETH) Zürich, 2009

[5] Lian Giger, *Auslegung und Konstruktion eines Riggs für ein Autonomes Segelboot*, Swiss Federal Institute of Technology (ETH) Zürich, 2009

[6] Patrick Moser, *Auslegung und Konstruktion eines Kiels für ein Autonomes Segelboot*, Swiss Federal Institute of Technology (ETH) Zürich, 2009

[7] Stefan Wismer, *Design of a Communication System for an Autonomous Sailboat*, Swiss Federal Institute of Technology (ETH) Zürich, 2009

[8] Students Sail Autonomously, http://www.ssa.ethz.ch/, 22.10.2009

[9] J. Beutel, S. Gruber, A. Hasler, R. Lim, A. Meier, C. Plessl, I. Talzi, L. Thiele, C. Tschudin, M. Woehrle and M. Yuecel: *Operating a Sensor Network at 3500 m Above Sea Level*. In Proc. 8th ACM/IEEE Int'l Conf. on Information Processing in Sensor Networks (IPSN/SPOTS '09), pages 405-406, April 2009

[10] PermaSense, http://permasense.ch/, 22.10.2009

[11] TinyNode, http://tinynode.com/, 22.10.2009

[12] *9522B L-Band Transceiver Product Information Guide* V1.6. Iridium Satellite LCC, 2008

[13] *MC75 AT Command Set* 04.001. Siemens AG, 2007

[14] *LEA-5 u-blox 5 Modules for GPS and GALILEO* A1. u-blox AG, 2008

[15] *SHT1X / SHT7X Humidity & Temperature Sensor* v2.04. Sensirion AG, 2005

[16] Wiegers, Karl E., *Software Requirements, 2nd Edition*, Microsoft Press, 2003

[17] Use reentrant functions for safer signal handling, http://www.ibm.com/developerworks/linux/library/l-reent.html, 22.10.2009

[18] Spring, an open source collection of DDX applications and libraries, http://wiki.csiro.au/confluence/display/ASLABOSS/Spring, 22.10.2009

[19] libcurl - the multiprotocol file transfer library, http://curl.haxx.se/libcurl/, 22.10.2009

[20] *ISU AT Command Reference Version 2.22* . , 2009

[21] *AT command set for GSM Mobile Equipment Version 7.4.0* . , 1998

[22] Access the Linux kernel using the /proc filesystem, http://www.ibm.com/developerworks/library/l-proc.html, 22.10,2009

[23] udev, http://www.kernel.org/pub/linux/utils/kernel/hotplug/udev.html, 22.10.2009

[24] Weather Online UK, http://www.weatheronline.co.uk/, 24.08.2009

[25] GCC, the GNU Compiler Collection, http://gcc.gnu.org/, 15.11.2009

[26] The GNU configure and build system, http://airs.com/ian/configure/, 17.11.2009

[27] VMime: C++ Mail Library, http://www.vmime.org/, 15.11.2009

[28] Setting up a build environment, http://www.gumstix.net/Setup-and-Programming/view/Getting-started/Setting-up-a-build-environment/111.html, 22.10.2009

[29] Replacing the file system image, http://www.gumstix.net/Setup-and-Programming/view/Getting-started/Replacing-the-file-system-image/111.html, 22.10.2009

6.2 Software Documentation

We list here general information about the software as well as several HOWTOS for common task when working with the software from this project.

6.2.1 Log Files

Every process writes diagnostic messages to a log file which is by default located in `/media/card/logs/`, i.e. on the SD flash drive. Every line in the log file is preceded by a human readable timestamp of the form “2009-10-19 00:38:52”. Then the name of the thread issuing the log message is printed followed by the message itself.

Log files are rotated daily by `lorgotate`. This means they are renamed by appending the current date and then moved to the directory `/media/card/logs/`. The process writing the log file is then sent a signal SIGHUP in response to which it closes and reopens the file. Subsequently the old log file is compressed using gzip. This reduces the file size drastically, as the content is usually highly repetitive which will save transmission costs. In the end we have a directory with a history of daily log files where the DataManager will find them and pick the important ones to send back home.

6.2.2 How to Connect to the Communication Platform from a Regular PC

The only interface to connect to the communication platform is the serial line normally connected to the sailing computer. You can either directly use the serial line with a program such as `minicom` or `kermit`. If you have to interact with the bootloader, this is actually the only way. Alternatively you can set up your computer for SLIP and use the serial line as a network connection just like the communication platform does in regular operation. For both alternatives you will need the files in `/src/scripts/cpc/`.

For a serial connection there exists a `kermit` configuration file called `kermit-setup-gumstix` in the aforementioned location. Edit the 'set line' directive to match your serial port. Then start `kermit` like this:

```
$ kermit kermit-setup-gumstix
```

This should configure your serial port and connect to the Gumstix. When you restart the communication platform you will be greeted by the u-boot boot-loader. Interrupt it and change the default runlevel to something other than 2. In runlevel 2 no shell is started on the serial line but rather it is configured as network device and your `kermit` connection will be useless. In any other runlevel you will be greeted by a login prompt.

To set up the serial link as network connection, you will need the `slattach` program (included in the debian package `net-tools`). Edit the file `serial` to match your serial device. Then run it as follows:

```
$ sudo ./serial start
```

This will create a new network interface called sl0 with an IP of 192.168.33.3. You can now connect to the communication platform using ssh:

```
$ ssh root@192.168.33.2
```

6.2.3 How to Compile the Software

To compile and run the software on a PC system we need the GNU C compiler [25], the GNU build system [26], and two software libraries: `libcurl` [19] and `libvmime`[27]. On Debian compatible systems, the following packages need to be installed:

- `build-essential`
- `gcc`
- `g++`
- `libvmime-dev`
- `libcurl4-gnutls-dev`
- `automake`

The source code is available as a zip file on the DVD in `/src/platform.zip`. Copy and uncompress the archive to disk and open a command line in that directory. As with every project using the GNU build system, the compilation is done with the well known command sequence:

```
$ ./configure
$ make
```

The configuration script has one useful argument: `--disable-gumstix`. It will redefine a compile-time constant that disables features that are only available on the Gumstix, such as GPIO ports or the hardware watchdog. This is useful when testing the software on a standard PC.

To compile the software for use on the Gumstix, you need to have a cross-compilation environment set up. Follow the instructions on [28] to set up and compile the tool-chain yourself. Be warned, as this usually takes several hours even on modern hardware. Then copy the user overlay i.e. the directory `/src/user.collection` to your installation directory. It contains the Permasense overlay and the software from this project. To build different parts of the software use the following commands:

```
1 $ bitbake commer
2 $ bitbake sht1x-kmodule
3 $ bitbake gumstix-commer-image
```

Command 1 will compile the software package `commer` containing all system daemons. The second command compiles the kernel module for the SHT1X temperature sensor. Both produce an output in form of an ipkg package suitable for installing on an OpenEmbedded system. The `commer` package is located at `build/tmp/deploy/glibc/ipk/armv5te/commer_1.0.0-r1_armv5te.ipk` while the kernel module is at `build/tmp/deploy/glibc/ipk/gumstix-permafrozer-verdex/sht1x-kmodule_1.0.0-r1_gumstix-permafrozer-verdex.ipk`. Lastly, the third command builds a complete system and kernel image. Both are located in the directory `build/tmp/deploy/glibc/images/gumstix-permafrozer-verdex/`.

6.2.4 How to Deploy the Software

The ipkg files generated when compiling the software to use on the Gumstix have to be transferred to the communication platform. This can either be done by copying them on the flash card using a card reader and then transferring the card to the Gumstix. Alternatively it can be copied over the SLIP link using scp.

Once the packages are on the Gumstix, install them with the following command.

```
$ ipkg install commer_1.0.0-r1_armv5te.ipk
```

To install an completely new image to the Gumstix flash memory, connect to the Gumstix using the serial line and follow the instructions in [29].

Alternatively you can copy the file system image (extension jffs2) and kernel image (extension bin) to the flash card. On the Gumstix make sure the symbolic links `rootfs` and `uImage` point to the file system and the kernel image respectively. Then change to the directory `/media/card/mtd-utils` and execute:

```
$ flash.rootfs
$ flash.kernel
```

This Method is usually much faster than copying the files over the slow serial link.

6.2.5 Software Directory Structure

We will list here all directories used by our software system on the communication platform. Wherever possible, we used standard directories, for example to store the configuration files.

Directory	Content
`/usr/bin/`	All executables: Message Manager, DataLogger, WatchDog, Power Manager and MessageManagerNotifier.
`/etc/commer/`	All configuration files.
`/media/card/logs/`	All *current* log files.
`/media/card/logs/old`	Rotated log files that have not yet been sent.
`/media/card/logs/sent`	Old log files that have been sent back to the control center.
`/media/card/logs/pictures`	Pictures taken from the webcam.
`/media/card/logs/videos`	Videos taken from the webcam.

6.2.6 How to Configure the Software

All Programs developed in this project have a number of configuration items. A configuration item is basically a variable that can be assigned a value. All configuration items have a default value, that is compiled into the program and is overwritten by values in the configuration files and on the command line, in that order. Moreover, each item has a data type. We use variables of type string, integer and boolean.

Configuration files are subdivided into sections, which feature options for different parts of the respective program. Every section is started with a header of the form [Message Manager], and all options in between that header and the next one belong to that section.

We will now list all configuration options for every program, including a short description and the default value.

Program:		**Message Manager**		
Configuration File:		/etc/commer/mm.conf		
Name	**Section**	**Type**	**Default value**	**Description**
`log directory`	main	string	`/media/card/logs/`	Directory for the main program log named "mm.log". This default is only true on the Gumstix. When Gumstix features are disabled, it is `/var/log/`
`fork`	main	bool	`0`	Fork to background upon startup. This is usually done using the command line parameter.
`use ethernet`	main	bool	`1`	Use the ethernet TransportHandler
`use gprs`	main	bool	`0`	Use GPRS TransportHandler
`use iridium`	main	bool	`0`	Use iridium TransportHandler
`status message retry limit`	Message Manager	int	`3`	How many times to retry sending a status message should it fail.
`status message retry period`	Message Manager	int	`180`	How long to wait in between sending attempts of status messages, in seconds
`command message period`	Message Manager	int	`3600`	How often to schedule a request to poll a command message in seconds.
`mailserver outgoing`	Message Manager	string		Mail server for outgoing emails
`mailserver incoming`	Message Manager	string		Mail server for incoming emails
`mailserver login`	Message Manager	string		Login name for both mail servers
`mailserver password`	Message Manager	string		Password for both mail servers
`mail expeditor`	Message Manager	string		Sender address of emails messages
`mail recipient`	Message Manager	string		Recipient of email messages

`port`	MessageServer	int	`2222`	TCP port to listen on for incoming connections
`read timeout`	MessageServer	int	`30`	Accumulative read timeout for incoming data in seconds. If the client takes longer than this to send data, the connection is dropped.
`write timeout`	MessageServer	int	`30`	Accumulative write timeout while sending data in seconds. If the client takes longer than this to receive data, the connection is dropped.
`max incoming msgsize`	MessageServer	int	`2000`	Maximal size of an incoming message in bytes. If a client tries to send more data, the connection is dropped.

Program:		**DataLogger**		
Configuration File:		/etc/commer/dl.conf		
Name	**Section**	**Type**	**Default value**	**Description**
`log directory`	main	string	`/media/card/logs/`	Directory for the main program log named "dl.log". This default is only true on the Gumstix. When Gumstix features are disabled, it is `/var/log/`
`fork`	main	bool	`0`	Fork to background upon startup. This is usually done using the command line parameter.
`Electrical`	main	bool	`1`	Measure electrical data
`GPS`	main	bool	`1`	Record GPS data
`Environment`	main	bool	`1`	Record environmental data (temperature and humidity)
`Picture`	main	bool	`0`	Take pictures in regular intervals
`Video`	main	bool	`0`	Record a 10 second video sequence regularely
`Time`	main	bool	`0`	Use the dummy TimeSource. Useful for thesting if no other source is available
`device file basename`	Electrical	string	`/proc/ad77x8/`	The basename of the device files for the analog inputs. To get the

			`ain`	individual channels, a number in the range [1-10] is appended.
`device file temperature`	Environment	string	`/proc/sht1x/temperature`	The device file for temperature. It is expected to read a raw value from the SHT1X sensor.
`device file humidity`	Environment	string	`/proc/sht1x/humidity`	The device file for humidity. It is expected to read a raw value from the SHT1X sensor.
`device file`	GPS	string	`/dev/ttyGPS`	The GPS device file. The default device is a symlink to /dev/ttyACM? created by udev.
`GPIO port`	GPS	int	`66`	The GPIO port used to switch the GPS device. The port is brought high to turn the device on and switched to high impedance to turn it off.
`device file`	Camera	string	`/dev/video0`	The v4l device file to read the pictures and videos from
`output directory`	Camera	string	`/media/card/logs/pictures`	Directory to store the pictures and videos in
`GPIO port`	Camera	int	`73`	GPIO port to switch the camera. The port is brought high to turn the device on and switched to high impedance to turn it off.

Program:		**Power Manager**		
Configuration File:		/etc/commer/pm.conf		
Name	**Section**	**Type**	**Default value**	**Description**
`log directory`	main	string	`/media/card/logs/`	Directory for the main program log named "pm.log". This default is only true on the Gumstix. When Gumstix features are disabled, it is `/var/log/`
`fork`	main	bool	`0`	Fork to background upon startup. This is usually done using the command line parameter.

Program:	**WatchDog**
Configuration File:	/etc/commer/wd.conf

Name	**Section**	**Type**	**Default value**	**Description**
`log directory`	main	string	`/media/card/logs/`	Directory for the main program log named "wd.log". This default is only true on the Gumstix. When Gumstix features are disabled, it is `/var/log/`
`fork`	main	bool	`0`	Fork to background upon startup. This is usually done using the command line parameter.
`watchdog_device`	main	string	`/dev/watchdog`	Gumstix watchdog device.

6.2.7 Connector Pin-Outs

6.2.7.1 BaseBoard P5

Connector between the BaseBoard and the small circuit board carrying the drivers for the serial line to the Iridium modem.

	3.3V	1	2	3.3V	
Iridium Power	GPIO71	3	4	IR_RXD	RX
Main Power	GPIO70	5	6	IR_TXD	TX
	GND	7	8	GND	

6.2.7.2 Case Power Input

3-Pin Souriau socket in the box for power input to the BaseBoard.

Wire color	Function	Souriau 3-Pin
Yellow / green	GND	⊥
(blue)	unused	1
Brown	VCC (Battery)	2

6.2.7.3 Case Serial and Power output

4-Pin Souriau socket in the box, internally connects to the switched power output and the Mini-DIN serial connector. Sub-D 9-Pin numbers are given as reference.

Function	Souriau 4-Pin	Sub-D 9-Pin	Mini-DIN
TX	A	3	3
RX	B	2	5

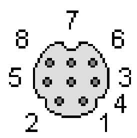

Figure 14: Mini-DIN

GND	C	5	4
+13V switched	D		

6.2.7.4 Case Main Power Switch

3-Pin Souriau socket in the box for in- and output to the main switch.

Wire color	Function	Souriau 3-Pin
Yellow / green	GND	⊥
blue	VCC (Battery)	1
Brown	Vout	2

6.2.7.5 USB connectors

Pin numbers are the same for standard USB plugs as shown below and for the BaseBoard connectors P14, P15 and P16.

Wire color	Function	Pin
red	VCC	1
white	D-	2
green	D+	3
black	GND	4

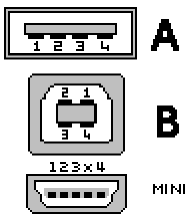

Figure 15: USB female

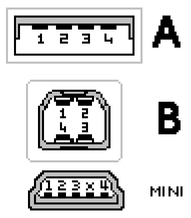

Figure 16: USB male

6.2.7.6 Iridium connector

The IDC connector is located on the circuit board plugged into P15 of the BaseBoard. The Souriau 19-Pin socket is in the Box wall.

Function	Souriau 19-Pin	IDC 26-Pin		Souriau 19-Pin	Function
Power Switch	L	1	2	NC	
	NC	3	4	N	
GND	P	5	6	R	PWR
PWR	S	7	8	T	GND
	U	9	10	V	
	NC	11	12	NC	DTR
RI	NC	13	14	K	
RTS	J	15	16	H	DSR
TX	G	17	18	F	CTS
DCD	M	19	20	NC	SGND
	NC	21	22	E	
	D	23	24	C	SGND
RX	B	25	26	A	

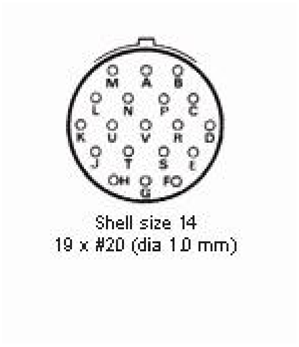

Figure 17: Souriau 19-Pin

6.3 Microtransat Rules

6.3.1 Safety Rules

1. Safety should take priority over winning.
2. Competitors may not attempt to inhibit other competitors by intentionally colliding with or obstructing their boat or by interfering with radio and electronic equipment.
3. All radio equipment must comply with appropriate International regulations.
4. Each boat must be equipped with a navigation light which is turned one during the hours of darkness. It should be visible from all directions and from a distance of at least 2 miles. The light maybe a single white light or a tri-colour red/white/green light.
5. Boats must take appropriate precautions to avoid collisions. This might include the use of radar reflectors, brightly coloured panels, warning labels/flags or AIS transponders and avoiding known shipping lanes. Each team must decide the exact precautions they wish to take.
6. The boat owner is liable for any damage caused to their boat or by their boat. The organisers take no responsibility for any damage caused.
7. Boats must remain outside any defined exclusion zones.
8. The organisers will only arrange permission for the boats to operate in the waters of the country of departure. If permission is not obtainable then the launch will take place in international waters. Competitors are responsible for arranging permission for their boat to enter

the waters of their destination country and other countries along the way. Competitors are recommended to remain in international waters where possible.

6.3.2 Tracking of boats and transmission on data

1. Each competitor will be required to provide their boat's position to the organisers via a web or email interface every 24 hours. Competitors are free to decide how this information is obtained and transmitted. A map showing each boat's position will be provided on this website. Any boat which fails to transmit for more than 10 consecutive days will be disqualified.
2. In adition to transmiting position data, each boat should keep a record onboard of its position at least once every 24 hours. A copy of this must be presented to the jury upon completion.
3. Competitors may transmit status information such as battery state from their boats.
4. During the race competitors may not transmit any information to their boats, including new waypoints, weather information or software updates. Any competitor which does will be disqualified. However if a competitor wishes to implement such features for use in an emergency or after the race, then they may do so on the understanding that their use during the race will result in disqualification. The jury may request to examine satellite phone bills, log files or computer code if they suspect data has been sent to the boat.

6.3.3 Criteria for entry

Every boat entered must fulfill the following criteria:

1. No source of propulsion other than wind.
2. The sailboat must be fully autonomous, no operator control is allowed.
3. The sailboat must be energetically autonomous, carrying on board any required batteries and electricity generating equipment.
4. The length of the boat must not exceed four metres.

6.3.4 The competition

1. The aim of the competition is simple, to sail an autonomous sailing boat between Europe and the Caribbean in the fastest possible time.
2. The competition will start over the 7 days. Competitors may launch at any time during this time.
3. The start point is planned for Ballydavid, County Kerry, Ireland, 52.22 degrees North, 10.4 degrees West.
4. The finishing line is the line of longitude between 10.00 degrees North, 60.00 degrees West and 25.00 degrees North 60.00 degrees West. Before departing, each team must choose a target area of 50km diameter along this line. A boat will only be considered to have finished the race when it reaches this 50km target, even if it has already crossed the finish line.

6.3.5 Judging Criteria

1. How quickly the boat crosses the Atlantic between the designated start point and the team's target end point.
2. A handicap will be calculated by the jury based on the boat's hull length using the following formula: Time Corrected = Time * square root(4 meters)/square root(length in meters)
3. In the event of no boat reaching the finishing line, no winner will be declared.
4. The result will be given by the jury within one week of the last boat arriving or giving up. During this time each competitor will submit a complete log of positions (minimum of 1 every 24 hours) along with any contest or comment to the jury and to all other teams.

6.4 Acronyms

AIS	Automatic Identification System, a nautical tracking system used for exchanging position, course, and speed between ships
EPOS	Maxon servo controllers
GPIO	General Purpose Input Output
GPRS	General Packet Radio Service
GSM	Global System for Mobile communications
ILP	Integer Linear Programming
IMU	Inertial Measurement Unit
RS-232	Recommended Standard 232, a standard for serial binary data signals
SBD	Short Burst Data
SLIP	Serial Line Internet Protocol
SMS	Short Message Service
SSA	Students Sail Autonomously
STL	Standard Template Library
UBS	Universal Serial Bus
WLAN	Wireless Local Area Network