Making TTP/A Portable[1]

Stefan Krywult

Version 0.3, 2007

Abstract

TTP/A is a lightweight time-triggered communication protocol providing a real-time communication service for wired sensor networks. A main idea of TTP/A is that the protocol is not bound to specific hardware.

This report analyzes the TTP/A implementation for AVR 8-bit microcontrollers and introduces mechanisms to ease the porting of this implementation to other microcontroller families.

As a testbed, the communication architecture of a small robot with several different dedicated microcontrollers is chosen. As an example the porting of TTP/A to an 32 bit microcontroller with an ARM7 core is exercised.

[1]This document is a result of the TTCAR project supported by the Austrian FWF project TTCAR under contract No. P18060-N04.

Contents

Chapter 1

Introduction

In many applications embedded systems take over even security and safety relevant tasks. Small integrated computer systems have been developed for controlling their environment (e. g., drive/fly by wire). The logical consequence is the development of completely autonomous systems. They explore their environment and cope with their tasks without a human user's action. Autonomous systems need to be able to collect relevant data of their environment with sensors, to make decisions based on that data and to influence the environment using actuators.

1.1 General Issue and Background

With the increasing computing power the complexity of embedded systems grows. Handling this complexity is a difficult issue that can be managed using distributed systems i. e., the system is subdivided in smaller parts that act jointly. The success and the quality of the collaboration highly depends on the communication system.

The communication system that connects the parts of an autonomous distributed system has to guarantee that the required pieces of information are delivered to the various subsystems with an almost constant delay and early enough, so that the autonomous system can react on changes of its environment in time i. e., real-time communication.

The case study and target platform of this work is the Tinyphoon research platform, a small robot for playing robot soccer in the Mirosot league. Because of its modular design a high performance real-time protocol is needed, which permits a communication suitable for such an autonomous system in a highly dynamic environment.

1.2 Problem Statements and Methodology

This work shows the applicability of a time-triggered approach as a solution to the problem of real-time communication in small real-time systems with special communication requirements.

The limited resources, the need of fast real-time communication and the variety of the involved platforms leads to high requirements:

- real-time: guaranteed transmission before a specified deadline
- performance: communication speed and efficiency
- integration: connection of the subsystems to the communication system
- availability: available in software and/or in hardware as chip or as intellectual properties
- portability: implementation for all platforms are available or source code can be ported easily
- resource-saving: low usage of Flash memory and Random Access Memory (RAM), small geometrical footprint and low power consumption
- licence: cost and conditions of licencing should allow the use in non-mass products with total system costs of less than Euro 2000.-

An existing implementation of th open real-time protocol Time-Triggered Protocol – Class A (TTP/A) that has a small footprint and is designed to be implemented on off-the-shelf microcontrollers, is modified to make the source code portable. TTP/A is then used on a platform of the Tinyphoon project. This case study is expected to reveal, which features are missing, which features are not used at all and which issues have to be solved when making a real-time communication system portable to many different hardware platforms.

The outcome of this work is relevant for the fast growing domain of high performance distributed real-time systems with low resource requirements e. g., in the automotive industry and small autonomous systems.

1.3 Outline

This report gives an overview of the target platform and the communication requirements of its subsystems (chapter **??**). Then the applicability of protocols, for small automotive distributed systems is analyzed (chapter **??**). Finally, a design of

a platform independent and enhanced version of the TTP/A protocol is proposed as a solution for the Tinyphoon robot and similar systems (chapter **??**).

The conclusion sums up the contributions of this work and gives an outlook on future developments (chapter **??**).

Chapter 2

Tinyphoon Communication Architecture

The Tinyphoon[NM05] is a small autonomous mobile robot in the shape of a cube with a side length of about seven centimeters. It has been developed at the Center of Excellence for Autonomous Systems of the Vienna University of Technology by a team led by Gregor Novak and Stefan Mahlknecht as a research platform for small autonomous robots. New concepts can be developed and approved in the challenging field of robot soccer. Its design is modular, various modules can be stacked one over another (see figure 2).

These modules are interconnected with a bus and are carried by a mechanical system that has been designed for playing robot soccer. This mechatronic part is about three centimeters high and is equipped with two strong DC motors. Moreover, the rechargeable battery for powering the Tinyphoon is stored here.

Three electronic modules have been designed up to now: a Vision Unit, a Decision Making Unit and a Motion Unit controlling the locomotion. Additional modules are planned to host extra sensors or provide computing power. The current architecture of the Tinyphoon robot is shown in figure 2.

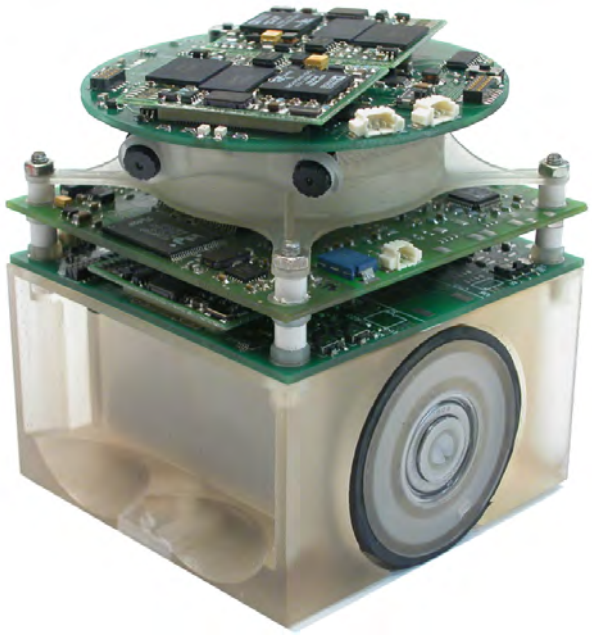

Figure 2.1: The Tinyphoon Robot

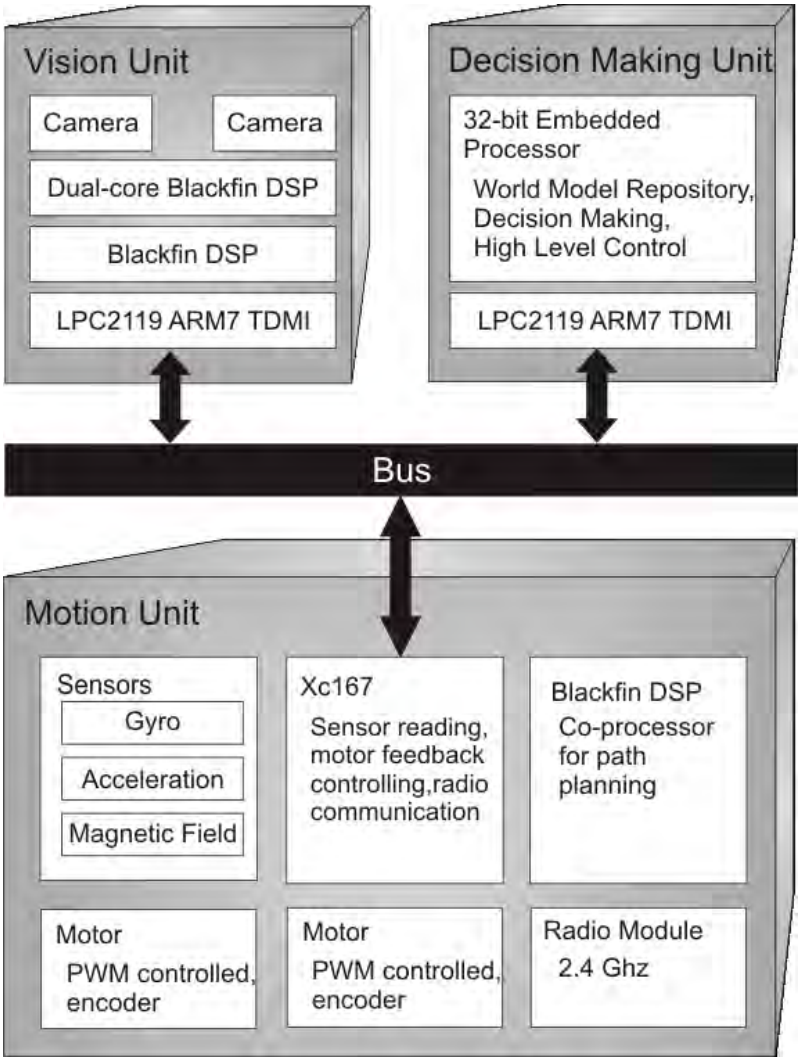

Figure 2.2: System Architecture of the Tinyphoon Robot

2.1 Subsystems

In order to play soccer a robot has to collect data about its environment, to make strategic decisions upon this information and to act accordingly. Each of these capabilities is a complex task. To handle this complexity modularity has been one of the most important principles for the design of the Tinyphoon. There is a separate and exchangeable module for each capability, namely the Vision Unit module for performing image recognition and visual self localization, the Decision Unit module for making strategic decisions and the Motion Unit module for path planning and for controlling the motors of the wheels. The hardware of the modules differs heavily.

2.1.1 Motion Unit

In figure 2.1.1 the Motion Unit of the Tinyphoon is shown. The main task of this unit is the feedback control of the motors. A XC 167 processor[Sie06](marked in the figure with letter *a*) from Infineon[1] is used for this task because of its outstanding peripherals. It also processes the signals from the encoders (512 impulses per rotation) of every motor, monitors the supply voltage from the battery pack and reads the sensors mounted on the Motion Unit: an analog gyro sensor with built-in thermometer (letter *b*), an analog magnetic field sensor (letter *c*) and two two-axis acceleration sensors with Pules Width Modulation (PWM) output (letter *d*).

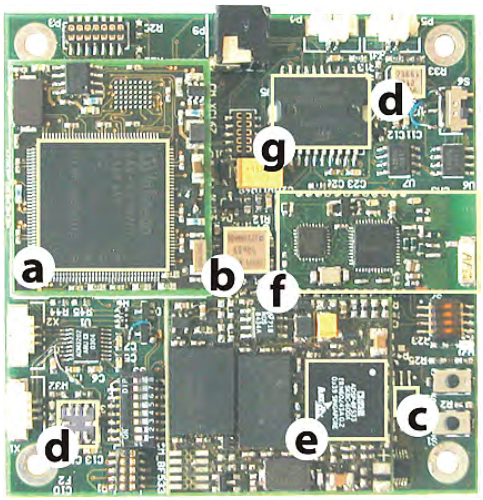

Figure 2.3: The Motion Unit

A Blackfin 531 Digital Signal Processor (DSP) (letter *e*) acts as coprocessor. It delivers the computing power needed for path planning [SJ05][SJN05]. In earlier versions of the Tinyphoon the DSP was also used for running a ball detection

[1] `http://www.infineon.com`

algorithm[MON05]. It is connected with the XC 167 via a fast SPI connection.

Moreover, the Motion Unit is equipped with a radio module (letter f) that allows exchanging information with other Tinyphoon robots and receiving commands from a personal computer.

The motor driver (letter g) controls the supply voltage of the motors accordingly to the PWM signals from the XC 176.

The communication interface to other modules will be implemented on the XC 167, there is no dedicated communication controller.

XC 167CS The XC 167CS is a 16-bit automotive microcontroller with 5 pipeline stages and runs at 40 Mhz. It has multiple register banks allowing fast context switches. It has several powerful peripherals that support a kind of Direct Memory Access (DMA) mode. Some of the main features of the XC 167CS are listed below:

- 6 kbyteon-chip Static Random Access Memory (SRAM), 128 kbyteon-chip Flash Program Memory, 512 kbyteexternal Random Access Memory (RAM)
- One cycle 16bit multiplication and one cycle Multiply and Accumulate (MAC) instruction
- 16 channel Analog Digital Converter (ADC) (10-bit or 8-bit, conversion Time down to 2.55μs)
- 16 programmable Interrupt priorities for 77 sources
- Five multifunctional timers/counters (with the functionality of a two channel decoder)
- Two 16-channel capture/compare units and flexible PWM signal generation
- Two synchronous/asynchronous serial communication channels (USART), two high-speed synchronous serial communication channels
- Dual Controller Area Network (CAN) Interface
- I^2C bus interface

Blackfin coprocessor The Blackfin processor is connected to the XC 167 via a Serial Peripheral Interface (SPI) interface. It runs at 600 MHz and works as coprocessor. The Blackfin processor from Analog Devices [2] is a mixture of a microprocessor and a DSP. Therefore, it is perfectly suited for number crunching applications like path planning.

[2] www.analogdevices.com

Sensors A gyro sensor detects the rotation of the robot and two orthogonally mounted acceleration sensors sense the change of velocity in each direction. The magnetic field of the earth is also measured, but the results of this sensor are disturbed because of the strong magnetic fields of the electro motors. 2 magnetic encoders on the wheels with 512 pulses per rotation

2.1.2 Vision Unit

The Vision Unit subsystem is shown in figure 2.1.2. The Vision Unit enables the robot to get an overview about its environment. Two CMOS cameras with a resolution of 640 x 480 take pictures simultaneously. Each of them is connected to a separate core of a dual-core Blackfin 561[Dev06] DSP from Analog Devices[3] (marked with letter h). The two cores perform edge and color blob detection. The gathered data is then sent to a single-core Blackfin 537 DSP (letter i) via SPI[4], where the two-dimensional edges are combined and three-dimensional lines are calculated. Based on the blobs and the lines object recognition can be performed. More details about the vision system can be found [BAS+06].

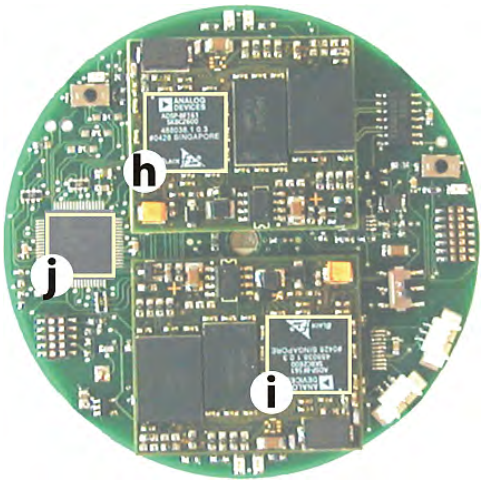

Figure 2.4: The TinyVision subsystem

The complete subsystem is mounted on a head that can be rotated by a stepper motor. This motor is controlled by a LPC 2119 ARM7-TDMI processor[Phi06] (letter j) that also functions as a communication interface to the other subsystems. It is connected to the single-core DSP via SPI.

[3] `http://www.analog.com`
[4] Serial Peripheral Interface

2.1.3 Decision Making Unit

An embedded 32-bit processor provides the necessary computation power for decision making. A built-in floating point unit supports algorithms based on fuzzy logic or neuronal networks and various filter algorithms. Linux or Windows CE can be used as operating system on this platform. The processors of the MCP family with Power architecture from Freescale[5] are currently evaluated regarding its use on the Decision Making Unit. S fuzzy logic framework for the Decision Unit has already been implemented [Web06].

Two tasks are assigned to the Decision Making Unit:

1. It merges the data received from the sensors on the Motion Unit, from the vision unit and from other robots to create a world model.

2. Decisions are made based upon the interpretation of this world model and a predefined strategy.

Integrating a communication system with high real-time requirements as part of such a complex software system is only possible using special operating systems (e.g. Real-Time Linux). Therefore, we decided to integrate a LPC 2119 ARM7-TDMI processor for handling the communication.

2.2 Communication Requirements

This section analyzes the data to be exchanged and additional attributes of the communication system.

2.2.1 Data Provided/Needed by the Subunits

Tables 2.1, 2.2, and 2.3 show the data that needs to be communicated by each module with a description of the data, its length, the transmission direction (input or output) and the update interval.

Table 2.1 lists the data that is sent and received by the Vision Unit. The Vision Unit has to be configured with the description of the objects that have to be recognized. The more different types of objects are to be searched for in the image the slower runs the detection algorithm. A maximum of eight object descriptions can be processed at a time. But these descriptions can be changed for every new image. A list of the recognized objects is then transmitted by the Vision Unit. Moreover, the to-be and the current position of the rotatable head and and the result of the visual self-localization has to be communicated.

The communication requirements of the Motion Unit are shown in table 2.2. It receives a command specifying the action to be performed with the necessary

[5] `http://www.freescale.com`

Description	Len	Dir	Interval
object descriptions	$8 \cdot 32$	I	20ms
detected objects	$16 \cdot 8$	O	20ms
to-be head position	2	I	20ms
last head position	2	O	20ms
self-localization	6	O	20ms

Table 2.1: Input/Output Data of the Vision Unit

parameters. The Motion Unit provides the data collected from the sensors as well as commands and data from other robots that is received via the wireless link.

Description	Len	Dir	Interval
command	12	I	20ms
position delta	6	O	20ms
gyro sensor	2	O	20ms
acceleration sensor	2*2*2	O	20ms
magnetic field	2	O	20ms
temperature sensor	1	O	1000ms
supply voltage	1	O	1000ms
remote command	32	O	20ms
remote response	32	I	20ms
remote robot data	3*64	O	20ms

Table 2.2: Input/Output Data of the Motion Unit

The communication requirements of the Decision Unit are shown in table 2.3. The results of the object recognition of the Vision Unit, data received via the serial link and the information from the sensors on the Motion Unit are combined with the knowledge about the environment (e.g. the football ground, the soccer game) to a more reliable world model.

2.2.2 Real-Time Requirements

The main sensor of the Tinyphoon is the camera system. The maximum update frequency of the Vision Unit is 50 Hz. This frequency is also used for scheduling communication cycles. A trigger for the communication may be received via the wireless link to synchronize the robots. This is important, because information about recognized objects is exchanged among the robots and this is much more useful if the robots rate exactly the same situation.

The maximum speed of the Tinyphoon robot is 4 m/s. Therefore, an object may

Description	Len	Dir	Interval
current position	6	O	20ms
teammates	$3 \cdot 6$	O	20ms
opponents	$4 \cdot 6$	O	20ms
ball	4	O	20ms
goals	$2 \cdot 6$	O	20ms
boards	$4 \cdot 6$	O	20ms

Table 2.3: Input/Output Data of the Decision Unit

move up to $2 \cdot 8$ cm from one image to another (e.g. two Tinyphoon robots driving with full speed in opposite directions). State estimation can be used to calculate the likely position of objects in the near future. However, this requires that the point in time when the picture is taken is known exactly.

If an error of ≤ 1 mm is considered as negligible, the overall jitter (including the jitter of the communication) has to be smaller than 125 μs.

2.2.3 Fault Tolerance / Dependability Requirements

Every functional unit exists only once on the Tinyphoon. Therefore, there is no fault tolerance on level of functional units. This is not a serious problem because a fail safe state exists, the robot can stop all motors at any time.

The communication system is partly integrated on functional units (e.g. the XC 167 microcontroller on the Motion Unit) and can not be replicated to provide redundancy.

Moreover, all subsystems depend on each other. The Tinyphoon can not continue operation when a subsystem fails. However, the failure of a subsystem or of the communication system must be detectable for every other unit to change to a safe state.

The length of the communication wires amounts only in a few centimeters and using the physical layer with differential signal transmission protects the data on the bus against electromagnetic disturbance (e.g. from the motors).

The small size of the Tinyphoon robot guarantees that no human and no machine could be harmed in case of a failure. Hence, features related to fault tolerance need not be considered when a communication protocol for the Tinyphoon robot is selected.

2.2.4 Data Throughput

Table 2.4 shows the number of bytes that need to be transferred for every subsystem.

770 bytes need to be transferred every 20ms.

Subsystem	**Length**	**Interval**
Vision Unit	394	20ms
Motion Unit	288	20ms
Decision Unit	88	20ms
Total	**770**	20ms

Table 2.4: Tinyphoon Communication Requirements

$$\frac{770\ byte}{20\ ms} = 15400\ byte/s \tag{2.1}$$

TTP/A needs thirteen bits (one bit inter-byte gap, one start bit, eight data bits, one parity bit, one stop bit and again one bit inter-byte gap) to encode one byte on the bus.

$$15400\ byte/s \equiv 204100\ bit/s \tag{2.2}$$

Considering the protocol overhead and future expansions, a communication speed of at least 0.25 Mbit is required.

2.2.5 Maintainability

Maintainability is one of the main issues for the communication system of the Tinyphoon robot. New subsystems will be created and the old ones are modified continuously. Hence, the communication system has to be adaptable to new hardware platforms and new fields of application. The same code has to be reused on many platforms to confine the effort of changing the protocol.

2.2.6 Debugging and Monitoring

As the Tinyphoon is a research project many people work on various topics around the robot. Hence, debugging facilities are one of the crucial features the communication system is required to provide. On the one hand the communication itself has to be debugged on the other hand the interfaces and the functioning of the subsystems have to be tested using the communication system's debugging facilities.

For software development a Joint Test Action Group (JTAG) interface[IEE90] exists for every microcontroller or processor. However, JTAG debugging interfaces are not compatible for devices from different vendors. Thus, a mechanism is needed to access the interface of every subsystem.

In the future we also plan to simulate parts of the Tinyphoon in software for testing purposes. Hence, a fast real-time communication between the simulation host (typically a standard personal computer) and the respective Tinyphoon hardware will be required.

2.2.7 Cost

The latest technologies are used to build the Tinyphoon, which is very cost-intensive. However, the cost for particular development tools turned out to be much more expensive than the actual hardware. Therefore, the availability of free or low priced development tools and documentation is vitally important for future extensions.

2.2.8 Implementation Effort

Currently various platforms are used on the Tinyphoon. Therefore, the communication system has to be integrated on each of this platforms. A small implementation effort allows the fast adaption of the communication system to new hardware platforms.

2.2.9 Comprehensibility of Interfaces

On large microcontrollers or processors a communication layer is introduced. So only a few developer get in touch with the interface of the communication system directly. Application programmers can use an Application Programming Interface (API) to access the functionality of the communication system. As the Tinyphoon is a very heterogeneous system a comprehensible interface makes it easier to port this API to other platforms.

Chapter 3

Selection of a Protocol

This chapter analyses existing implementation of the communication on the Tinyphoon robot.

3.1 Current Communication

Currently, two modules for the Tinyphoon are available: the Motion Unit and the Vision Unit module. The two modules originally communicated using the Controller Area Network (CAN) protocol. The communication was performed by the integrated CAN controllers of the XC 167 on the Motion Unit and of the LPC 2119 on the Vision Unit. The LPC 2119 has been added to the schematics of the Vision Unit to perform the rotation of the head and all communication related tasks, whereas the XC 167 has to perform many other, partly time critical tasks: reading the sensors and the encoders, generating the PWM signals for the motors, sending and receiving data through the wireless link and communicate with its co-processor.

The hardware of the modules has been designed in a way that allows connecting a CAN transceiver either to the I/O pins of the integrated CAN controller or to the I/O pins of the Universal Asynchronous Receiver and Transmitter (UART) of the LPC 2119 and the XC 176, respectively. This guarantees flexibility and allows the software implementation of a communication protocol. Thus, a communication protocol that can be implemented in software and that uses the serial ports can be integrated on the Tinyphoon without having to modify the hardware.

For the FIRA World Championship in Summer 2006 a stripped-down version of the Tinyphoon has been created. It consists only of a motion and a vision unit. These two modules communicate with a very simple protocol, based on standard UART frames. There is no mechanism for bus arbitration required because the connection is bidirectional. However, the communication system turned out to be one of the most error-prone subsystems. Problems with the interpretation of the semantic and

the structure could be solved by rendering the specification of the protocol more precisely. But the reception of messages asynchronously to the program execution caused unexpected problems that were hard to track down and solutions were quite complex. Moreover, this approach is not applicable for systems with more than two communication partners.

3.2 Hardware Versus Software Implementation

Small systems like the Tinyphoon put strong restrictions on the design of the Printed Circuit Board (PCB). An external communication controller needs additional space on the PCB. It may not be possible on all systems to provide this space. However, on the Vision Unit the LPC 2119 microcontroller is only used for the generation of a PWM signal for rotating the head and for the communication. The PWM signal could also be generated on one of the Blackfin DSP. The MF4300 Flexray controller has about the same size as the LPC 2119. Thus, the communication controller could be used instead of the microcontroller.

On the Motion Unit the XC 167 performs many tasks and can not be replaced by a communication controller. Because of the high number of sensors and the necessary analog circuitry there is not enough space for an additional communication controller without a complete redesign of the Motion Unit.

Software implementations of communication systems require additional calculation time and peripherals. On the LPC 2119 of the Vision Unit enough resources are available. The XC 167 on the Motion Unit is also equipped with enough peripherals and plenty of calculation time is available, because calculation intensive tasks are delegated to the coprocessor. However, the XC 167 already has to perform some time-critical tasks. These have to be combined with the requirements of the communication system. One possibility is that the scheduler of the communication also triggers all the other time critical tasks.

3.3 TTP/A

Time-Triggered Protocol (TTP)/A is very efficient (see table **??**) and the source code of an open source implementation in C[1] exists, which can be used for free, even for commercial applications. TTP/A offers clock synchronization and a comprehensible application interface, the Interface File system (IFS). The global time allows the the generation of time stamps that are valid in the whole cluster.

Hence, TTP/A fulfills all major requirements for the communication on the Tinyphoon. However, there are some issues that require a closer investigation:

[1] available at `http://www.vmars.tuwien.ac.at/ttpa`

- The performance has to be boosted. The transmission is byte-oriented with a synchronization event between every two bytes, limiting the performance and the efficiency.
- In the current implementation, the error handling of messages has to be done by the application, increasing the complexity of the application software.
- The existing implementation is designed for eight bit microcontroller from Atmel[2]. The effort for porting the code to other platforms (16- and 32-bit platforms) has to be evaluated.
- The C implementation of TTP/A uses a software UART, but only pins with hardware UART functionality are connected to the bus transceivers on the modules of the Tinyphoon.

3.4 Results

The discussion in this chapter has shown that the lack of space on the PCB does not allow the use of an external communication controller.

TTP/A is more efficient than comparable protocols (e. g., CAN or Local Interconnect Network (LIN)). It features a global time and the byte-wise transmission of the payload allows the exchange of messages up to 62 bytes without any fragmentation.

Due to these reasons we decided to evaluate TTP/A on the hardware of the Tinyphoon robot.

[2] `http://www.atmel.com`

Chapter 4

Time-Triggered Communication System

This chapter describes the techniques used to make the existing implementation of TTP/A portable and to adapt it for the Tinyphoon platform. Finally, the result of the implementation are evaluated and discussed.

4.1 Existing TTP/A implementation

The existing implementation of TTP/A was written by Christian Trödhandl[1]. In contrast to the first version that is realized in assembler and therefore is extremely hardware dependent, the current version is written in C. The TTP/A is targeted to the the AVR platform from the Atmel[2].

4.1.1 Source Code

The source was developed for the AVR enabled version of the GNU C Compiler (GCC). It uses some features of the GCC that are not compliant with the American National Standards Institute (ANSI) C standard[ANS89].

The source code is organized three subdirectories:

/src This directory contains the actual source code of the TTP/A protocol.

/include In this directory the header files are store. It has to be added to the application that wants to use the TTP/A protocol.

/ldscripts The linker scripts for the special memory sections of the protocol code are stored here.

[1] available at `http://www.vmars.tuwien.ac.at/ttpa`

[2] `http://www.atmel.com`

The code is divided in six logical parts:

Main File The main file `main.c` implements the entry point and initializes the protocol and the user application.

Scheduler In `schedule.c` a small scheduler is implemented that supports the prioritization of tasks. The scheduler can be deactivated.

Protocol Core The core of the protocol is implemented in `ttpa.c`. The files `ttpa_*.c` contain some core related functions.

MS round The handling of Master Slave Address (MSA) and Master Slave Data (MSD) rounds is implemented in the files `ms.c` and `ms_*.c`. The implementations of Master slave (MS) rounds for master and slave differ strongly. Thus, role specific parts are implemented in separate files.

IFS Functions for managing and accessing the IFS are provided in the file `ifs.c`. Helper functions are implemented in `ifs_*.c`. Two assembler files are used for the definition of the file look-up table (`ifs_tab.S`) and weak symbols for the files (`ifs_weak.S`).

Bus The bus related functions are implemented in the files `bus_*.c`. The original version of TTP/A provides a software UART only. Additional code is needed for accessing the bus in the slave implementation (`bus_*_slave.c`).

4.1.2 Architecture

Figure 4.1 shows a simplified Unified Modelling Language (UML) activity chart of the TTP/A slave implementation.

After the initialization the slave waits for a Master Slave Address (MSA) round. It uses the periodical pattern of the MSA fireworks byte for synchronization (`bus_sync`). When the bus operation succeeds the received fireworks byte is handled (`ttpa_recvfb`). After bus synchronization the only possible firework byte is 0x55, the identifier of a MSA round. The new communication round is started (`ttpa_newround`) by selecting the Round Definition List (RODL) that is assigned to the round. Then the first frame is marked as current frame (`ttpa_next_frame`) and the scheduled action is read from the RODL (`ttpa_next_rodlentry`). The code execution is then continued depending on the current schedule entry:

Bus Receive Sync One or more bytes have to be received from the bus. The start of the transmission is used to resynchronize the node. The reception is prepared in the function `ttpa_recvsyncframe` by configuring the bus subsystem for the next receive operation (`bus_receivebyte_init`). The completion of the bus operation is signaled by the bus subsystem by calling the function, which the pointer `bus_op_complete` points at. It has been set to

`ttpa_recvsyncslot`. In this function the clock synchronization is performed and then the handler for received bytes, `ttpa_recvslot`, is called. If there are more bytes that have to be received, the normal receive operation (see next paragraph) is performed. Otherwise, a new frame is started.

Bus Receive This operation is the same as the one described above, but no clock synchronization is performed.

Bus Send Passive This operation is performed in the send slots of the node when the node tries to integrate itself in the cluster. It is used at start-up or after an error. The node tries to receive something in its send slots to ensure that no other node uses these slots for transmissions. Thus, collisions due to wrong configurations can be avoided. First `ttpa_sendpasvframe` configures the bus subsystem (`bus_recvbyte_init`) for receiving and waits for the completion of this operation. Then `ttpa_sendpasvslot` is called. If the receive operation was not successful, the slot is empty and the node can use it for transmissions. If there are more slots to be tested, this operation is repeated, otherwise the handling of the next frame is started.

Bus Send Once the node has verified, that its send slot is free, it uses this operation to transmit data in the slot. In `ttpa_sendbyte_init` the bus subsystem is configured for a transmission (`bus_send_byte_init`). After the completion of the bus operation, `ttpa_sendslot` is called. If more bytes have to be transmitted, the send operation is repeated. Otherwise, otherwise the handling of the next frame is started.

Execute This operation executes a task that has either been defined by the application or by the protocol (e. g., for handling Master slave (MS) rounds). In the function `ttpa_execframe` a timer is configured to generate a time-out event at the start of the time slot. After the time-out the function `ttpa_execslot` is called. It triggers the preparation of the next frame and then performs the configured task.

Execute at the End of the Round Also the Inter Round Gap (IRG) can be used for executing tasks. This operation is almost the same as the normal execute operation but it signals the end of a communication round instead of preparing the next frame.

End of the Round If the communication round is finished and no task execution is scheduled for the IRG, tis operation is performed. In `ttpa_eor` a timer is configured to trigger the start of a new round at the appropriate point of time.

After the processing of the frame either a new frame or a new communication round is started. In the first case `ttpa_next_frame` is called and the next scheduled action is read from the RODL (`ttpa_next_rodlentry`). In the second case the slave

node executes the function `ttpa_eor_slave` that configures the bus subsystem for the reception of a new fireworks byte. As soon as the bus operation is finished, the fireworks byte is interpreted (`ttpa_fbrecv`) and the according communication round is started (`ttpa_newround`).

4.1.3 Bus Subsystem

The read and write operations on the bus are performed through the bus subsystem. This design decision makes it easy to integrate new implementations of the bus access. Listing 4.1 shows the interface of the bus subsystem that is defined in `bus.h`.

```
int bus_init(void);
void bus_sendbyte_init(bus_iobuf_t *param);
void bus_recvbyte_init(bus_iobuf_t *param);
void bus_sync(void);
void (* volatile bus_op_complete) (bus_iobuf_t *);
```

Listing 4.1: Interface of the Bus Subsystem

The interface defines function for initializing the bus subsystem and for initiating a transmission or a reception of a byte. On slave nodes an additional function for synchronizing with the bus has to be implemented. A pointer `bus_op_complete` has to be set to a function that is called after the bus operation has been finished. The configuration and the result of a bus operation is stored in the structure `bus_iobuf_t`. Listing 4.2 shows the fields of this structure.

```
typedef struct bus_iobuf_t {
  uint8_t buf;
  parity_t par;
  uint16_t slotstart;
  union status_t {
    struct fields_t {
      unsigned spdup   : 3;
      unsigned timeout : 1;
    } fields;
    bus_io_err_t error;
  } status;
} bus_iobuf_t;
```

Listing 4.2: Structure for Bus Operations

The field `uint8_t buf` stores the byte that has to be transmitted or that has been received. The field `parity_t par` is an enumeration type that specifies the parity. The number of the microtick when a byte has to be transmitted or has been received is stored in `uint16_t slot_start`. The union `status` stores a speed-up factor and a timeout flag, when the structure is used for the configuration of a bus

operation, or an error field when the structure is used for returning the result of an operation.

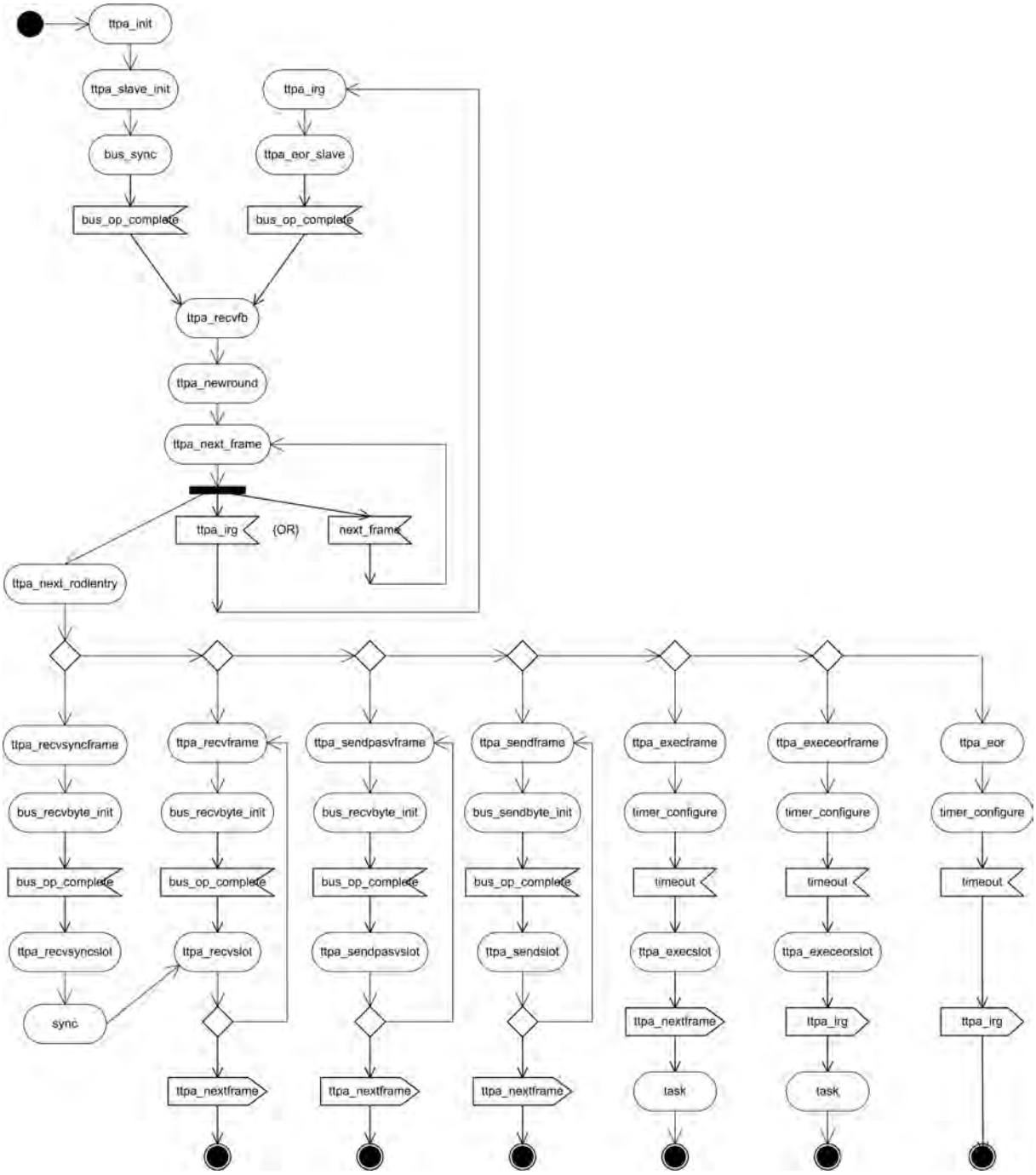

Figure 4.1: UML Activity Chart of the TTP/A Implementation

4.2 Hardware UART Implementation

The TTP/A implementation originally uses a UART that is implemented in software. This ensures a very precise synchronization because an input capture logic is used to detect the start of a transmission with the precision of one internal micro tick. However, the software implementation needs one interrupt for every bit. This leads to a high interrupt load even at small bit rates and influences coexisting applications on the same microcontroller. Thus, the bus interface has also been implemented using a hardware UART. Due to the schematics of the Tinyphoon's modules only this implementation can be used for the communication.

The source code shown in the next sections 4.2.1 and 4.2.2 has been designed to work with the Hardware Abstraction Layer (HAL) proposed in section 4.3.2. All hardware specific parts are replaced by the call of a macro.

4.2.1 Receiving

The code for receiving a byte from the bus is divided in four functions and a interrupt service routine.

The first function is part of the bus interface and is called by the TTP/A core code to initiate the reception of a byte in the next time slot. The function `bus_recvbyte_init` has a parameter of the type `bus_iobuf_t *` (see listing 4.3). It stores the time-out of this parameter in its own data structure, sets the global pointer for the timer callback to the setup function and configures the timer to raise an interrupt on the start of the next slot. `HAL_HW_UART_RECV_SU_CORR` can be used to correct a possible delay of the hardware UART. Then, the function `bus_transcvr_recv` switches a possible transceiver to receive mode, if needed. The UART receive interrupt is still disabled.

```
void bus_recvbyte_init(bus_iobuf_t *param)
{
  bus_hwuart_buf.timeout =
       param->status.fields.timeout;

  ttpa_sig_oc = (void (*)()) bus_recvbyte_setup;

  hal_set_timer_compare_value(param->slotstart +
       HAL_HW_UART_RECV_SU_CORR + ttpa_bitlength);
  hal_delete_compare_match_interrupt();
  hal_enable_compare_match_interrupt();

  bus_transcvr_recv();

  hal_uart_rx_disable();
```

```
}
```

Listing 4.3: HW UART Receive Initialization

On the start of the time slot `bus_recvbyte_setup` is called (see listing 4.4). The expected transmission starts one bit time after the call of this function. First the reception of bytes is enabled and the output compare logic is configured to generate a match event 12.5 bit times after the start of the slot. Following, the call back that is called, when a byte is received by the UART, is set to `bus_recvbyte`.

The bus subsystem has to implement two modes of byte reception, one with time-out and another without a time-out. In the case that a time-out is used, the pointer for the callback of the compare match interrupt is set to `bus_recvbyte_to` and the interrupt is enabled. Otherwise, the interrupt is disabled.

At the end of the function pending UART receive interrupts are cleared and the interrupt is enabled.

```
void bus_recvbyte_setup(void)
  hal_uart_rx_enable();
  hal_inc_timer_compare_value(12 * ttpa_bitlength +
                              ttpa_bitlength>>1);
  bus_uart_recv = bus_recvbyte;

  if(bus_hwuart_buf.timeout) {
    ttpa_sig_oc = (void (*)()) bus_recvbyte_to;
    hal_delete_compare_match_interrupt();
    hal_enable_compare_match_interrupt();
  } else {
    hal_disable_compare_match_interrupt();
  }
  hal_delete_uart_receive_interrupt();
  hal_enable_uart_receive_interrupt();
}
```

Listing 4.4: HW UART Receive Setup

If a byte has been received successfully, then `bus_recvbyte` is called to handle it (see listing 4.5). The UART receive interrupt is disabled and a structure `bus_iobuf_t` is prepared to be passed to the `bus_op_complete` callback. In this structure the time of the start of the transmission, a possible error during the transmission, the received byte and its parity are stored. Afterwards, the reception of bytes is disabled. This has to be done after the received byte and its parity have been read, because this information may be lost, when the UART is disabled. Finally, the `bus_op_complete` callback is called.

```
void bus_recvbyte(void)
```

```
{
  bus_iobuf_t buf;
  hal_disable_uart_receive_interrupt();
  buf.slotstart = recv_time - (12*ttpa_bitlength);
  if (hal_uart_check_error()) {
    buf.status.error = BUS_IO_FE;
  }
  else {
    buf.status.error = BUS_IO_OK;
  }
  buf.par = hal_uart_get_parity();
  buf.buf = hal_uart_get_value();

  hal_uart_rx_disable();

  (*bus_op_complete)(&buf);
}
```

Listing 4.5: HW UART Handle Received Byte

In case that a time-out occurs, the function `bus_recvbyte_to` (see listing 4.6) is called by the compare match interrupt service routine. Again a `bus_iobuf_t` is prepared. The error field is set accordingly to signal a time-out. Then both, the compare match and the UART receive interrupts are disabled and the callback `bus_op_complete` is called.

```
void bus_recvbyte_to(void)
{
  bus_iobuf_t buf;

  buf.status.error = BUS_IO_TIMEOUT;

  hal_delete_compare_match_interrupt();
  hal_enable_compare_match_interrupt();
  hal_disable_uart_receive_interrupt();

  (*bus_op_complete)(&buf);
}
```

Listing 4.6: HW UART Receive Time-Out

The last piece of the hardware UART implementation for the reception of bytes is the interrupt service routine for the UART receive interrupt (see listing 4.7). It stores the time of the reception of the byte and adds a correction value to consider the

different delays of different hardware UART. Then the callback, which the pointer `bus_op_complete` points at is called.

```
hal_uart_receive_interrupt()
{
  recv_time = hal_get_timer();
  recv_time += HAL_HW_UART_RECV_CORR;
  (*bus_uart_recv)();
}
```

Listing 4.7: HW UART Receive Interrupt

4.2.2 Sending

The implementation for sending bytes is realized in two functions. The first one is called by the TTP/A core to configure the transmission of a byte for the next time slot. The second is used internally as callback for the timer, when the transmission starts.

The function `bus_sendbyte_init` (see listing 4.8) is part of the bus interface. It initiates the transmission of a byte at the beginning of the next time slot. The byte to be transmitted, its parity and the start of the next time slot is passed to the function in a `bus_iobuf_t` structure.

The parameters for the transmission are copied to local data structures and the callback for the compare match interrupt service routine is set to `bus_sendbyte`. Then, the compare value is set accordingly, so that the compare match interrupt is raised exactly when the transmission has to be started. Possibly pending compare match interrupts are deleted and the interrupt is enabled. Finally, a possible transceiver is switched to transmit mode.

```
void bus_sendbyte_init(bus_iobuf_t *param)
{
  bus_eff_bitlen = ttpa_bitlength >>
                   param->status.fields.spdup;

  bus_hwuart_buf.buf = param->buf;
  bus_hwuart_buf.par = param->par;

  ttpa_sig_oc = (void (*)()) bus_sendbyte;

  hal_set_timer_compare_value(param->slotstart +
                              HAL_HW_UART_SEND_CORR);

  hal_delete_compare_match_interrupt();
```

```
  hal_enable_compare_match_interrupt();

  hal_transceiver_send();
}
```

Listing 4.8: HW UART Initialize Transmission

The function `bus_sendbyte` (see listing 4.9) is called when the transmission has to be started. The parity mode of the hardware UART is configured and the data byte is transmitted. Then a `bus_iobuf_t` structure is prepared. The start of the next slot is set to 13 bit times after the start of the last slot. The error field is set accordingly to signal a successful transmission. Finally, the `bus_op_complete` callback is called and the structure is passed as parameter.

```
void bus_sendbyte(void)
{
    bus_iobuf_t buf;

    hal_uart_set_parity(bus_hwuart_buf.par);
    hal_uart_send_byte(bus_hwuart_buf.buf);

    buf.slotstart = hal_get_timer_compare_value() +
                    bus_eff_bitlen * 13;
    buf.status.error = BUS_IO_OK;

    (*bus_op_complete)(&buf);
}
```

Listing 4.9: HW UART Perform Transmission

4.3 Portable TTP/A Implementation

Based on the description of the TTP/A implementation above, an architecture and modifications of the source code are proposed in this section, to make the main part of the C implementation independent from the target platform. The adoptions are minimized and pooled in two files, one for the processor architecture and one for the concrete hardware of the node (including processor model and I/O configuration).

4.3.1 Compiler Independence

Many hardware platforms are only supported by a single compiler and there is no universal compiler. Therefore, a portable source code has to be written in a way, that as many compilers as possible are able to compile it correctly. These compilers are often very specialized, limited and/or proprietary. However, every

modern compiler is at least compatible to the C standard of ANSI. Source code that does not use any other C constructs than defined by this standard is understood by most of the compilers. Thus, the first step of creating a portable version of TTP/A is replacing constructs that are not compliant with the ANSI C standard. Especially constructs for the definition and initialization of complex data types (e. g., anonymous structures, initialization of selected fields of a structure, ...) that are supported by the GCC are not part of the ANSI C standard.

4.3.2 Hardware Abstraction Layer

To make the source code independent from the hardware the implementation of a HAL is proposed. The HAL defines a standardized interface to the hardware at a very low level of abstraction. The level has to be that low, because portability has not been considered consequently during design phase of the implementation. One the one hand there is an interface to the bus subsystem that allows a high level of abstraction. It would allow driver like implementation of the bus subsystem. On the other hand the timer used by TTP/A is read, written and configured in several protocol states. Moreover, in the current source code the timer hardware is accessed directly, there is no abstract software interface.

Figure 4.2 shows the proposed architecture for a portable version of a TTP/A implementation.

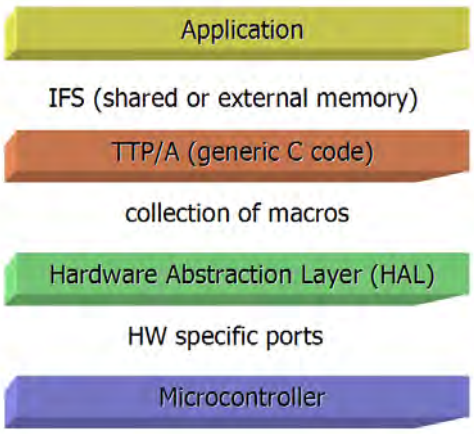

Figure 4.2: Layers of the Portable TTP/A implementation

The HAL is implemented using macros. The advantage of macros is that it is guaranteed, that no overhead is added to the code. Almost all compilers understand macros. Another possibility is the use of inline functions. A compiler can check the syntax of a call of an inline function in contrast to the call of a macro. However, inline functions are not part of the ANSI C standard. Moreover, if a compiler supports inline functions, it is up to the compiler to decide how the call of an inline

function is realized and whether an overhead is added or not.

The HAL is divided in three parts: timer related functionality, UART-related functionality and other functionality that does not fit in one of the two other groups. Because of the low level of abstraction many macros have to be defined. However, tests on the LPC platform have shown, that most of the macros can be kept simple and implemented with one line of code.

Timer

The macros of the HAL that are related to the timer are listed and described in table 4.1. If the implementation of the software UART is used, another seven macros listed in table 4.2 are necessary.

Macro	**Description**
`hal_configure_timer()`	Configures timer for use with TTP/A
`hal_set_timer(value)`	Sets TTP/A timer/counter to `value`
`hal_get_timer()`	Sets TTP/A timer/counter to `value`
`hal_set_timer_compare_value(value)`	Sets TTP/A timer/counter compare register to `value`
`hal_inc_timer_compare_value(value)`	Sets TTP/A timer/counter compare register to `value`
`hal_get_timer_compare_value()`	Sets TTP/A timer/counter compare register to `value`
`hal_delete_compare_match_interrupt()`	Deletes eventually pending timer overflow interrupt
`hal_enable_compare_match_interrupt()`	Enables timer overflow interrupt
`hal_disable_compare_match_interrupt()`	Disables timer overflow interrupt
`hal_delete_overflow_interrupt()`	Deletes eventually pending timer overflow interrupt
`hal_enable_overflow_interrupt()`	Enables timer overflow interrupt
`hal_disable_overflow_interrupt()`	Disables timer overflow interrupt
`hal_delete_input_capture_interrupt()`	Deletes eventually pending timer overflow interrupt

Table 4.1: Timer HAL Macros

Additionally, for every used timer interrupt an according macro has to be defined that is used as entry point for the interrupt service routine (`hal_overflow_interrupt()`, `hal_compare_match_interrupt()`, and `hal_input capture_interrupt()`. An-

Macro	Description
`hal_enable_input_capture_interrupt()`	Enables timer input capture interrupt
`hal_disable_input_capture_interrupt()`	Disables timer input capture interrupt
`hal_set_input_capture_falling_edge()`	Configures input capture mechanism to trigger on a falling edge
`hal_set_input_capture_raising_edge()`	Configures input capture mechanism to trigger on a raising edge
`hal_set_input_capture_toggle_edge()`	Configures input capture mechanism to trigger on any edge
`hal_get_input_capture_value()`	Reads value from input capture register
`hal_set_pin_on_compare_match(value)`	Configures compare match logic to set pin state to `value` on next match event

Table 4.2: Additional Timer HAL Macros for the Software UART

other macro (`hal_timer_t`) is used to define the data type of the timer (e. g., `uint16_t` for a 16 bit wide timer register).

UART

The macros of the HAL that are related to the timer are listed and described in table 4.3.

Other Issues

Some more macros are defined in the HAL for accessing to Flash memory and Electrically Erasable Programmable Read-Only Memory (EEPROM), for enabling and disabling interrupts, for configuring the node, for reading the I/O pin for the software UART and for controlling a possible, external transceiver. These macros are described in table 4.4, 4.5, 4.6, and 4.7.

The macros for accessing the EEPROM may be defined as constant numbers if on a particular platform no EEPROM is available. The parts of the code for accessing the EEPROM is never executed, if no files of the IFS are configured to be stored in the EEPROM.

The macros related to the transceiver may be empty if no transceivers are used that need to be switched between transmit and receive mode.

Macro	Description
`hal_uart_configure()`	Configures the UART
`hal_uart_set_parity(value)`	Sets parity mode to even or to odd
`hal_uart_rx_disable()`	Disables reception of bytes (often also the received data is cleared)
`hal_uart_rx_enable()`	Enables reception of bytes
`hal_uart_send_byte(value)`	Transmits the byte `value`
`hal_uart_get_value()`	Returns last byte received
`hal_uart_check_error()`	Checks whether an error occurred during reception of the last byte
`hal_uart_get_parity()`	Returns parity of the last byte received
`hal_uart_receive_interrupt()`	Header of UART receive interrupt
`hal_delete_uart_receive_interrupt()`	Deletes possibly pending UART receive interrupt
`hal_enable_uart_receive_interrupt()`	Enables UART receive interrupt
`hal_disable_uart_receive_interrupt()`	Disables UART receive interrupt

Table 4.3: UART HAL Macros

4.3.3 Linker Script

A special linker script is used for generating special sections for the IFS and a table of tasks that is executed during the initialization. The linker script strongly depends on the used platform and of the linker. Thus it has to be changed accordingly. Unfortunately, at the time of writing there is no documentation about the layout of these sections available.

The according linker file for a target platform have to be created using the linker file of the available implementation as example. For many linker graphical tools exist, which allow a convenient adaptation of the linker files. In many cases linker files have to be created for new projects anyway.

Macro	Description
`hal_progmem_read_byte()`	Reads a byte from the program memory (usually flash ROM)
`hal_progmem_read_word()`	Reads a word from the program memory (usually flash ROM)
`hal_eeprom_is_ready()`	Returns true if the EEPROM is ready for a read or write operation
`hal_eeprom_read_byte()`	Reads a byte from the EEPROM

Table 4.4: HAL Macros for Memory Access

Macro	Description
`hal_enable_interrupts()`	Globally enables interrupts
`hal_disable_interrupts()`	Globally disables interrupts

Table 4.5: HAL Macros for En-/Disabling Interrupts

Macro	Description
`hal_configure_swuart_io()`	Configures the I/O pins used by the software UART
`hal_configure_node_swuart()`	Performs node specific configuration that is related to the software UART
`hal_configure_hwuart()`	Performs general configuration that is related to the hardware UART
`hal_configure_node_hwuart()`	Performs node specific configuration that is related to the hardware UART
`hal_sw_uart_rxpin_is_high()`	Tests whether the receive pin of the software UART is at a high level

Table 4.6: HAL Macros for the Node and I/O Configuration

Macro	Description
`hal_init_transceiver()`	Initializes the transceiver
`hal_transceiver_recv()`	Switches the transceiver to receive mode
`hal_transceiver_send()`	Switches the transceiver to transmit mode

Table 4.7: HAL Macros for Controlling an External Transceiver

Chapter 5

Summary and Conclusion

This chapter summarizes the outcome and the contribution of this work and gives an outlook on future work in the covered domain.

5.1 Evaluation

To evaluate the proposed HAL architecture on the one hand and the applicability of TTP/A on the current Tinyphoon hardware TTP/A has been ported to the LPC 2119 from Philips Semiconductors[1]. The experiences gained during porting have been used to improve the HAL architecture and to suggest improvements of the TTP/A protocol that make it even more suitable for the use on small autonomous robots.

5.1.1 TTP/A on the LPC 2119

The LPC 2119[Phi06] microcontroller bases upon the ARM7 core. It has been selected to be used on the Tinyphoon because of its powerful architecture despite its small footprint. It features the standard peripherals of a typical microcontroller (UART, timers with input capture and compare match logic, I/O ports, I^2C and CAN bus, ...).

A test cluster consisting of a master and a slave node has been build. The WinARM version of the ARM GCC 4.0.2 has been used for the development. The original TTP/A implementation also has been written for a version of the GCC. Thus, all compiler extensions that are not supported by the ANSI C standard have been disabled to emulate a compiler that understands ANSI C constructs only.

During the implementation of the HAL some problems had to be solved:

- The AVR platform supports at most 16-bit wide timers whereas the timer registers of the LPC 2119 are 32 bit wide. A special data type has been

[1] now NXP, `http://www.nxp.com`

defined that has the same bit width as the timer registers. Whenever the value of a timer is stored in a variable, this data type has to be used.

- The timers of the LPC 2119 can not raise an interrupt when the timer register overflows. Thus, this behavior has to be emulated using another compare match unit.
- The interrupt concept of the LPC 2119 is much more complex than that of the AVR microcontrollers. The macros have been designed in a way to support both of them. However, on platforms with a complex interrupt logic some additional steps might be necessary till the interrupt service routine is called correctly (e. g., entries in linker and/or start-up files, modifications of interrupt vector table, ...
- The timings might vary from one platform to another (e. g., the UART receive interrupt is raised earlier or later after the reception of a byte). To solve this problem correction values are used in the critical parts of the source code.
- The adaption of the linker file turned out to be a very cumbersome job.
- The implementation of the hardware UART has not improved the maximum performance as expected. The reason for this behavior is, that the action for the next frame is prepared in the inter frame gap after the previous bus operation has been finished. The implementation of the software UART stops the reception of a byte after the raising edge of the stop bit has been detected. In contrast the hardware UART samples mid of the stop bit and then raises the receive interrupt. Therefore, the preparation time for the next slot is even shorter and the maximum baud rate is smaller than that of the implementation using a software UART.

5.1.2 Suggested Improvements of TTP/A

The suggestions are based on the issues that came up during porting and testing TTP/A on the hardware of the Tinyphoon. The problems are grouped in three categories: enhancing performance, improving the handling of large chunks of data and avoiding the use of linker scripts.

Enhancing Performance

A hardware UART has been implemented in order to improve the performance of TTP/A while the load of the microcontroller is reduced. However, tests have shown, that on the AVR platform the hardware UART configuration (up to 16000 bit/s) does not even reach the performance of the version using the software UART (up to 19200 bit/s).

The different way of stop bit detection in combination with the way of preparing the next byte frame causes this problem. The hardware UART generates a receive interrupt after the stop bit has been sampled in the middle of the bit, whereas the software UART stops receiving as soon the raising edge of the stop bit has been detected. The preparation of the next frame slot is started as soon as the reception or transmission of the last slot has been completed. Thus, the preparation time is shorter when the hardware UART is used.

To solve this problem the preparation of next frame slot can be started while the preceding frame is still being received or transmitted. In this moment the communication is completely handled by the hardware UART. Therefore, the computing power of the Central Processing Unit (CPU) can be used for the preparation.

The frames of TTP/A are 13 bits long. but the maximum length of a transmission with a hardware UART is twelve bits (one start bit, eight data bits, one parity bit and two stop bits). Thus, a continuous transmission using a standard hardware UART is impossible. This restriction impedes the exploitation of the powerful features of more enhanced hardware UART that allow the transmission of multiple bytes without the interaction of the CPU.

Moreover, the transmission of every byte has to be triggered by a timer. One the one hand this increases the load on the CPU(e. g., timer configuration, interrupt handling) on the other hand not only the start of a transmission but the complete transmission is closely related to the timing, which makes a separate implementation of the UART and the timer subsystems hardly possible. Hence, the code lacks portability.

Handling of Large Data Packets

TTP/A has been designed as communication protocol for smart transducers. The protocol is adequate for exchanging small amounts of data. It causes only a little overhead but only supports frames with the length of one byte. Though many frames can be scheduled consecutively to avoid fragmentation of the payload, TTP/A provides no means to check the integrity of the complete payload. Thus, the application has to ensure, that all bytes of the payload have been communicated successfully. A checksum mechanism as provided by LIN would reduce the effort for the application design. Additionally, the encapsulation of an integrity check in the protocol ensures the compatibility thereof among all TTP/A nodes.

Linker Scripts

Linker scripts are highly hardware and linker dependent. The process of porting TTP/A also includes the complex adaption of the linker scripts. This effort only can be reduced by avoiding linker scripts. The current version of the TTP/A im-

plementation uses them heavily for providing a comfortable programming interface while keeping memory requirements low.

In the future configuration tools will be available that can configure the IFS and the RODL of TTP/A nodes. Hence, there programming interface will not be used by the programmer directly and can be designed in a way that no linker scripts are needed.

5.2 Contribution of This Work

The communication requirements of the subsystems of the Tinyphoon robot have been analyzed. Each subsystem and the data it needs or provides has been described. This analysis of the communication on the Tinyphoon has shown, that real-time features (e. g., predictable communication, small jitter, global time) improve the interaction of the subsystems and allow new approaches of application design. Fault tolerance is not an issue on small robots but a high data throughput and the maintainability of the communication system are vitally important.

TTP/A combines the basic real-time features with a small implementation footprint. Its standardized debugging and configuration interface allows the observation of the communication and the platform-independent tuning of the configuration of each subsystem. TTP/A is implemented in software and coexists with other applications, which saves valuable space on the PCB. Moreover, the source code of TTP/A is available for free. Because of these features TTP/A has been evaluated on the Tinyphoon robot.

Based on an analysis of the mode of operation of the current TTP/A implementation an architecture for making TTP/A portable is proposed. A HAL has been designed and implemented on the original platform of the TTP/A implementation. A case study on the hardware of the Tinyphoon robot has proved the capability of this concept and that TTP/A can be used for the communication on small autonomous systems.

Based on the evaluation of this case study a proposal for further improvements of TTP/A have been derived. Methods have been proposed to make TTP/A easier to port to new hardware platforms and to make it even more suitable for the use on small autonomous systems like the Tinyphoon robot.

5.3 Outlook to Future Work

Currently a new protocol based on TTP/A is being developed. Its performance will be boosted by supporting the application of the advanced features of modern, powerful UART hardware units. The protection of data frames with a checksum

will ease the handling of large chunks of data. This protocol will suit the needs of small autonomous systems even better than TTP/A.

The communication architecture of the Tinyphoon robot has to be revised in the future. The current implementations of the subsystems are designed according to the event-triggered paradigm. With the introduction of a global time base the complete system can be synchronized. Hence, all measurements on the robot can be triggered at the same point in time, which makes the fusion of the result considerably easier. Therefore, we plan to integrate real-time communication on the Tinyphoon robot and to use a time-triggered software design.

In the future the wireless communication system of the Tinyphoon robot can be used to synchronize measurements within a group of robots. This simplifies the exchange and the interpretation of data about their environment. Thus, a distributed world model of all robots can be established.

The separation of the communication protocol from the application reduces the effort of maintaining the source code. The communication system can be added to the software of new subsystems easily.

Bibliography

[ANS89] ANSI – American National Standard Institute. Ansi standard x3.159-1989, December 1989.

[BAS+06] M. Bader, M. Albero, R. Sablatnig, J. E. Simó, G. Benet, G. Novak, and F. Blanes. "embedded real-time ball detection unit for the yabiro biped robot". In *"Fourth Workshop on Intelligent Solutions in Embedded Systems (WISES'06)"*, Vienna, Austria, 2006.

[Dev06] Analog Devices. *Datasheet Blackfin 561*, 2006.

[IEE90] IEEE – Institute of Electrical and Electronics Engineers. Ieee standard 1149.1: Standard test access port and boundary-scan architecture, 1990.

[MON05] S. Mahlknecht, R. Oberhammer, and G. Novak. "real-time image recognition system for tiny autonomous mobile robots". *Real-Time Systems*, 29, 2005.

[NM05] G. Novak and S. Mahlknecht. Tinyphoon - a tiny autonomous mobile robot. Technical report, Institute for Computer Technology, Vienna Universitys of Technology, Vienna, Austria, 2005.

[Phi06] Philips. *Datasheet LPC2119*, 2006.

[Sie06] Siemens. *Datasheet XC167*, 2006.

[SJ05] M. Seyr and S. Jakubek. "mobile robot predictive trajectory tracking". In *Proceedings of the 2005 ICINCO*, ICINCO, 2005.

[SJN05] M. Seyr, S. Jakubek, and G. Novak. Neural network predictive trajectory tracking of an autonomous two-wheeled mobile robot. In *Proceedings of the 16th IFAC World Congress*, Elsevier, Prag, 2005.

[Web06] Daniel Weber. Decision making in the robot soccer domain. Master's thesis, Vienna University of Technology, 2006.