Implementation Details on the TTP/A Slave Protocol

M. Holzmann
W. Elmenreich

Institut für Technische Informatik
TU Vienna, Austria
mike@vmars.tuwien.ac.at
wilfried@vmars.tuwien.ac.at

July 27, 1999

Abstract

This paper describes the experiences made with the implementation of the time-triggered protocol/Class A (TTP/A)on a dedicated 8 bit MCU. TTP/A is a multimaster UART-based sensor network that provides temporally predictable communication to sensor and actuator nodes in a distributed real-time system. It provides multiple views of a sensor node, such as a periodic real-time service view and an aperiodic diagnostic view. Since the protocol provides start-up synchronization of the sensor nodes by the master, sensor nodes can be built up with any low-cost microcontroller with an on-chip resonator, a timer, and a digital input/output port. At the Institut für Technische Informatik, the TTP/A master and the TTP/A slave part of the protocol have been implemented for evaluation purposes. This paper describes implementation details and experiences with the TTP/A Slave part of the protocol on an ATMEL AVR8 microcontroller with an 8 bit timer/counter.

Keywords: real-time systems, time-triggered, sensor bus, UART.

Introduction

The TTP/A slave protocol was implemented on an Atmel AVR8 MCU. The Tiny11, the cheapest model of the AVR8 series, was choosen. In huge quantities it costs only about 0.8 USD and thus a node built up with this MCU will cost about 2 USD. Besides the MCU only a bus driver (MC33290 ISO K-line driver) and a voltage regulator is needed to built up a node. Picture 1 shows a prototype with AVR8 MCU and the Motorola bus driver.

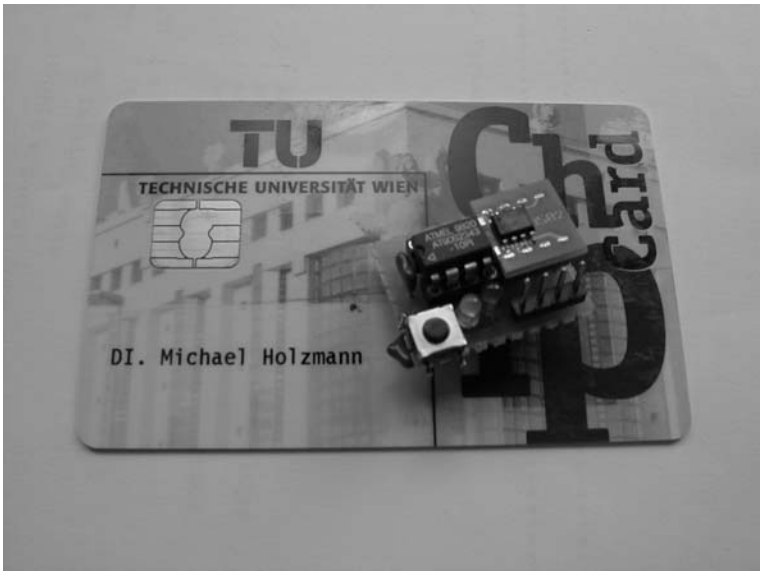


Figure 1: TTP/A Slave Node Prototype

TTP/A Protocol

The TTP/A protocol [Kop95] is a low-cost multi-master sensor network for connecting sensors and actuators (transducers) to real time systems. In TTP/A, there are two different types of nodes. The complex *master* node, that controls the operation of the protocol, and simple transducer nodes, the *slaves*, that react to the activities of the master. TTP/A supports temporal predictability, but does not provide internal fault-tolerance.

Transducer Nodes

Any properly programmed 8-bit or 16-bit microcontroller that supports external interrupt capability and a timer can be a transducer node in a TTP/A system. In the context of TTP/A, a transducer node is a slave node that must execute the basic TTP/A slave protocol.

Master Node

A master node requires a dedicated protocol processor, e.g., a Motorola 68332 processor, timers, non-volatile memory for the storage of the protocol code and the application specific protocol data structures (the message descriptor list MEDL), a dual-ported memory for communication to the host processor, and a dedicated bi-directional UART channel for every supported TTP/A bus.

Principles of Operation

The basic TTP/A protocol is based on one-byte state messages. Most of these messages are data messages, while only one special message, the fireworks message, is a control message. Every protocol event either occurs at a predefined point of time (e.g., sending a message) or in a predefined time window (e.g., receipt of a message).

Basic TTP/A Protocol

In TTP/A, all communication activities are organized as rounds (Figure 2). A round consists of a sequence of slots where each node is assigned several slots for communication. This assignment is specified a priori in a round-descriptor list (RODL). A slot is subdivided into the transmission phase (TP) and the inter frame gap (IFG). During the TP a data frame is transmitted, while during the IFG the protocol is executed.

A slot is 16 bit cells long, where the TP occupies 11 bit cells (8 data bits, framing, and parity) and the IFG occupies 5 bit cells. A round starts with a special control message, the fireworks frame (FF), which is transmitted by the active master. The fireworks frame serves two purposes:

- it is the global synchronization event for the start of a new round, and
- it contains the name of the active RODL for this round.

After the fireworks frame the active master transmits a second control frame, the parameter frame (PF). It contains the parameter byte which extends the information given in the fireworks byte. The parameter byte is followed by a sequence of data bytes from the individual nodes as specified in the active RODL. A round terminates when the end of the active RODL is reached. Each round is independent of previous rounds. To be able to differentiate between a fireworks byte and a data byte, the fireworks byte has characteristic features in the value domain and in the time domain:

- the fireworks byte has an even parity while all data bytes have odd parity
- the intermessage gap between the fireworks byte and the first data byte is significantly longer than the intermessage gap between the succeeding data bytes.

These characteristic features make it possible for all nodes to recognize a new fireworks byte, even if some faults have disturbed the communication during the previous round. The characteristic features of the fireworks byte simplify the reintegration of repaired nodes – a repaired node monitors the network until a correct fireworks byte is detected. Because each node is assigned its sending/receiving slots a priori by the definition of the RODL, it is not necessary to carry the identifier of a message as part of the message. All eight data bits of a message are true data bits. This improves the data efficiency of the protocol, particularly for short single byte messages that are typical for field bus applications.

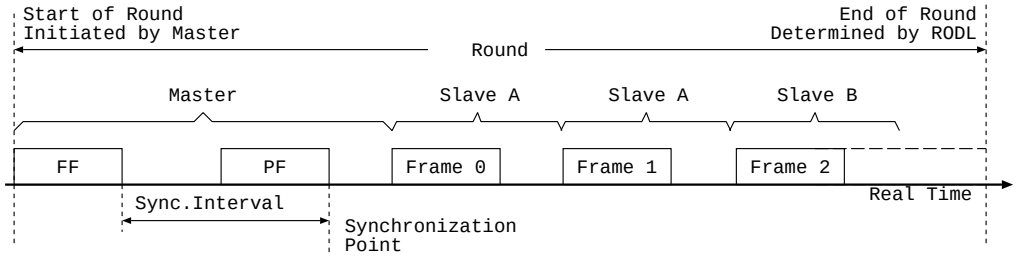


Figure 2: Structure of a TTP/A round

Startup Synchronization

To support slave nodes with imprecise, but cheap resonators like on-circuit RC-oszillators, TTP/A provides a *startup synchronization* mechanism. The master broadcasts a square wave pattern on the bus (see figure 3), where the temporal distance between two falling edges is exactly eight bitcells. By measuring this intervall, the slaves can adjust the length of one bitcell to their local time and setup their UARTs with this value. The next falling edge of the startup synchronization wave form is interpreted as start bit of a frame and the slave reads the information on the bus with the UART. If 01111110_2 was read, startup synchronization is successfully completed and the slave awaits a valid fireworks frame to begin with normal operation. Otherwise it restarts startup synchronization with measuring the eight-bitcell interval. *Startup synchronization* is the initial protocol state of a slave if it is powered up or restarts due to failures. The startup synchronization pattern can be broadcasted by the master between two normal rounds to enable integration of new or re-synchronizing slaves.

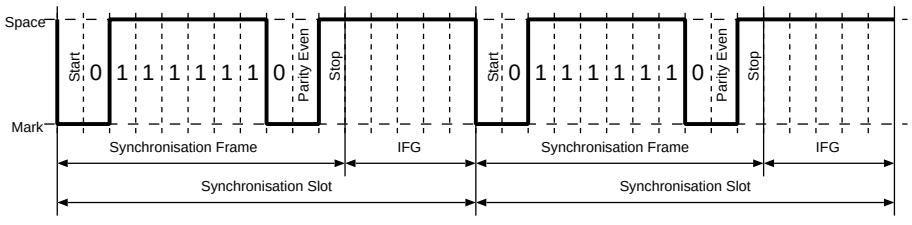


Figure 3: Startup Synchronization Wave Form produced by the TTP/A Master

Protocol State Machine

The protocol was implemented as a state machine (figure 4) with a set of states for startup synchronization and synchronized protocol operation from round to round. The transitions between the states are caused either by timed events (when the timer/counter reaches a particular value), by external events (when the bus signal goes from high to low e.g. at the start bit of a transmission) or by program control.

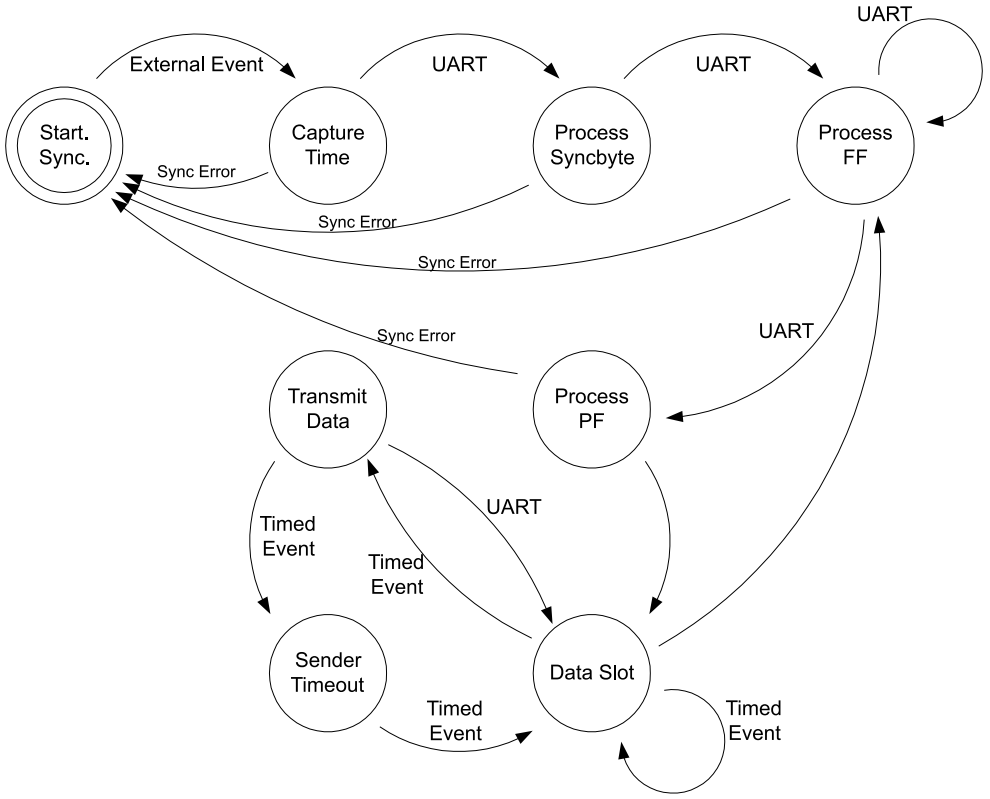


Figure 4: Protocol State Machine

When the slave is initialized, its protocol state is *Start Synchronization*. In this state the slave is listening on the bus for a synchronization pattern from the master. On an external event indicating a transition on the bus, a timer is startet for measuring. The protocol state swaps to *Capture Time*.
In the *Capture Time* state the protocol reacts to the next external event and reads the timer value. When the measured time is not within the possible tolerance, the slave restarts at *Start Synchronization*, otherwise all timing intervals are calculated based on the measured value and the next bus activity triggers a UART reception. The state changes to *Process Syncbyte*. The synchronization byte always has a value of 01111110_2 with even parity. When the received byte is not equal to the expected value, the slave restarts at *Start Synchronization*. This may happen several times until the slave catches a true synchronization pattern.

When the synchronization pattern was read correctly, the slave is synchronized and awaits a fireworks transmission from the master (*Process FF*). Depending on the received value, either there comes another synchronization pattern, which is ignored, or the slave restarts at *Start Synchronization* if the parity was not correct or the fireworks byte is accepted and the successional state is the *Process PF*.

After processing the parameter frame the state changes to *Data Slot*. When there is a data transmission in the next slot, the following state is *Transmit Data*. If the current slot is the last one in the round descriptor list, the protocol awaits a new fireworks frame (*Process FF*), otherwise the state stays the same. The *Transmit Data* state changes to *Sender Timeout* if the sending node fails to start the transmission within a given time window. When the sending or receiving transmission was successful, the next state is *Data Slot* again for handling the next slot.

Implementation

Protocol Timing

Due to the fact, that all slot-timing dependent timeouts have to be recalculated at least one time per round and, thus, dynamically stored, it was a goal to reduce the number of required timeouts for protocol execution to a minimum. One time each slot the protocol code is executed. The protocol determines whether a data transmission has to be performed in the following slot. If this is the case it sets a timeout to the begin of the UART action (ToIFG). Else it sets a timeout of a slot's length (To16 - length 16 bitcells) to invoke protocol execution in the consecutive slot.

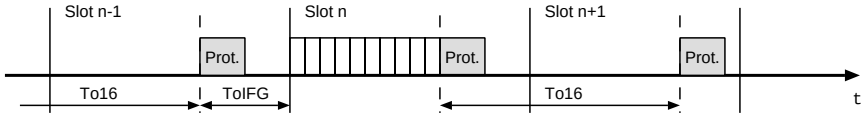


Figure 5: Protocol Timing during Send Slot

Figure 5 shows the protocol timing in the case of sending a data byte in slot n. In slot n-1 the protocol sets a timeout to the begin of slot n. If the timeout expires the UART function is called. The UART sets 10 consecutive timeouts of one bitcell length each to transmit framing, parity and data bits. After sending the stop bit the protocol is immediately executed. Due to no transmission activities in slot n+1, the protocol sets the timeout $T016$ to be invoked in the consecutive slot.

Figure 6 depicts how receive of data happens. The protocol sets a timeout to 1/2 of the length of the receive window before the begin of the next slot if it gets aware that data has to be received next slot (see figure 6, slot $n-1$). When this timeout expires the node starts waiting for an incoming start bit. If the start bit is not detected within a given time window (receive window), the

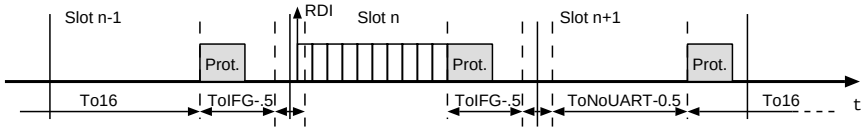


Figure 6: Protocol Timing during Receive Slot

receive action is cancelled and protocol execution is continued at the same time within the slot as usual, this is 10 bitcells after slotbegin, i. e. the length of a UART transmission (see figure 6, slot $n + 1$).

Timer Routines

The 8 bit timer unit of the MCU had to be extended to 16 bit resolution by adequate overflow handling and timer reset routines. The timer unit has also to generate the timeouts for the bitcells when the software UART is transmitting data. While the software UART is active, the local time must not be lost. Due to timers of the most cheap MCUs support only interrupt generation on overflow but no match functionality, the cycles spent in execution between raise of a timer interrupt and the reload of the timer have also to be considered.

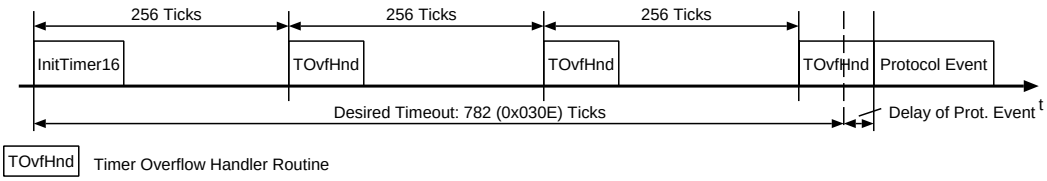


Figure 7: Concurrency between Timer Interrupt and Protocol Event

A 16 bit timeout is divided into multiple 8 bit timeouts and a rest. If this rest is less than the time the timer overflow routine needs to reset the timer, the timeout is delayed. To avoid concurrence between the timer interrupt handler and protocol events, it is necessary to split 16 bit timeouts into parts greater than the time the timer overflow handler needs for execution.

In figure 7 it is shown how a protocol event is delayed by the execution of the timer interrupt handler.

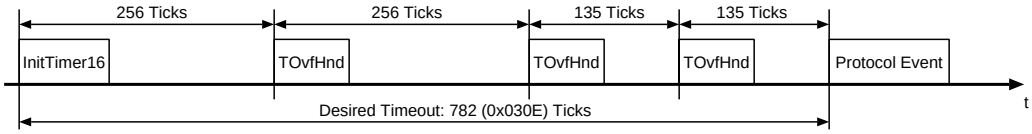


Figure 8: Correct Execution of a 16 Bit Timeout

The correct execution of the timeout by splitting the last full 256 tick part of the timeout and the rest of the 16 bit timeout into two equal parts is depicted in figure 8.

The Software UART

The UART is implemented in software for two reasons:

1. The recalibration of timeouts due to the instability of the internal RC oszillator needs that the length of a bitcell of the UART is adjustable in timer tick granularity.

2. For economic reasons: The smallest MCUs usually do not provide hardware-UARTs.

The length of a UART bitcell is calculated from the measured *synchronization interval* by division. Thus the value is a real number of local MCU timer ticks. When cutting the value to an integer number of timer ticks, the error is multiplied when several consecutive timeouts (UART bitcells) are set. While data transmission takes part the only timer of the MCU is needed for the UART so a loss of ticks during UART operation is not tolerable. Otherwise on one hand the UART would get imprecise in the last bitcells on the other hand the protocol would lose the correct timing.

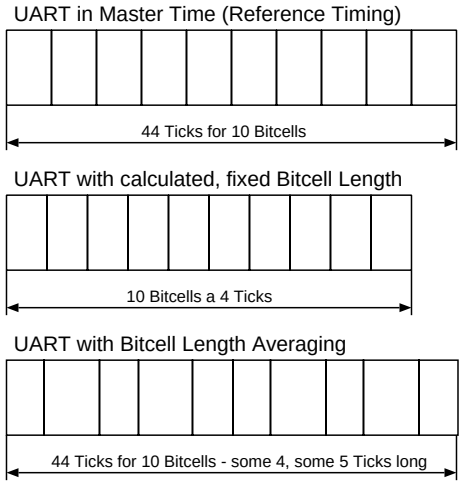


Figure 9: The UART Bitcell Averaging Algorithm

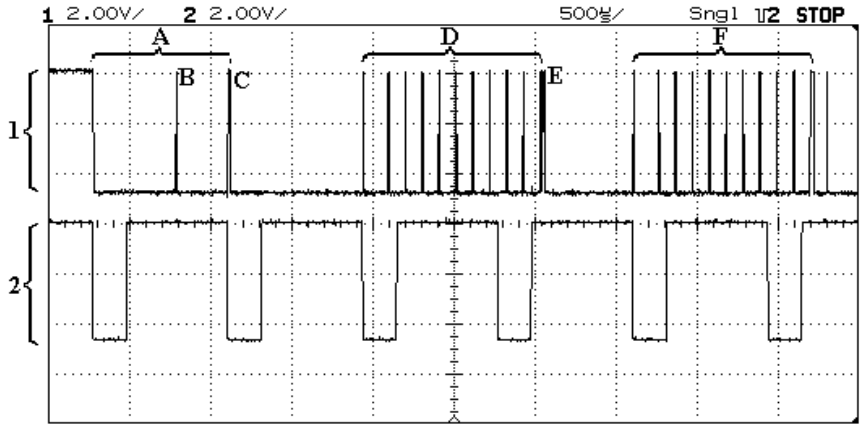
Thus the UART maintains a fraction part counter to enlarge some bitcells to achieve an averagely correct bitlength and a correct UART length within one tick as shown in the example in figure 9.

Performance Evaluation

Protocol part	Frequency	# Instructions executed
UART Send 1 Byte	at slots with a Send activity for the node	133
UART Receive 1 Byte	Fireworks slot, Parameter slot and each slot with a Receive activity for the node	273
Protocol normal slot processing	for each slot of the round	18
Protocol transmit data processing	each slot with send/receive activity	18+9
Protocol Fireworks and Parameter frame processing	at beginning of round	67
Timer Overflow Handler	averagely all 256 Timerticks	9
Interrupt dispatcher	every protocol event	12

Table 1: Instruction effort for different protocol activities

Table 1 lines up the number of assembler instructions needed for different program activities. The maximum utilization is given during UART operations, when i.e. 273 instructions have to be executed during 10 UART bits. Assuming a data transfer rate of 10 kB and an average execution time of 1 microsecond per instruction, the workload of the MCU is up to 27.3 %. Increasing the Baud rate worsens the proportion. During all other protocol operations the workload is below the utilization of a UART receiving communication. (See also Fig. 10 for a comparison of program execution time to protocol events and times.)



1 *The upper signal shows the protocol activities: a high value indicates protocol operation, a low value idle time for application execution*
2 *The signal in the lower half of the captured screen is a periodic synchronization pattern from the master*
A *Measuring the reference time*
B *Execution of the Timer Overflow Handler because timeout is greater than 256 ticks*
C *Calculation of all needed timeouts based on the measured reference time*
D *UART mode - trying to read the synchronization pattern*
E *Sync pattern was read correctly, node operates properly, now waiting for beginning of new round (indicated by a Fireworks)*
F *Reading the Fireworks*

Figure 10: Evaluation of the Protocol Timing at startup

References

[Kop95] H. Kopetz. TTP/A - A Time-Triggered Protocol for Body Electronics Using Standard UARTS. In *International Congress and Exposition Detroit, Michigan*, Commonwealth Drive, Warrendale, Febr.-March 1995. The Engineering Society For Advancing Mobility Land Sea Air and Space, SAE International.