TTP/A Smart Transducer Programming
A Beginner's Guide

Version 0.6

W. Elmenreich, W. Haidinger, R. Kirner
T. Losert, R. Obermaisser, and C. Trödhandl[*]

Institut für Technische Informatik
Technische Universität Wien,Vienna, Austria

September 16, 2008

[*]Authors in alphabetical order

Contents

1 Introduction

1.1 Overview

TTP/A is the fieldbus protocol of the Time Triggered Architecture. It has been developed since 1998 by the Real-Time Systems Group at the Vienna University of Technology. Its purpose is the interconnection of transducer (sensor and actuator) nodes providing hard real-time guarantees for communication.

A *smart* transducer is the integration of an analog or digital sensor or actuator element and a local microcontroller that contains the interface circuitry, a processor, memory, and a network controller in a single unit. The smart sensor transforms the raw sensor signal to a standardized digital representation, checks and calibrates the signal, and transmits this digital signal via a standardized communication protocol to its users.

The design of the network interface for a smart transducer is of great importance. Transducers come in a great variety with different capabilities from different vendors. Thus, a smart transducer interface must be very generic to support all present and future types of transducers. However, it also must provide some standard functionalities to transmit data in a temporally deterministic manner and in a standard data format.

In response to a call for proposal for a new smart transducer standard by the Object Management Group (OMG) in December 2000, a smart transducer interface standard has been proposed that contains the TTP/A protocol. This smart transducer standard has been adopted by the OMG in January 2002.

Being thus standardized, TTP/A represents an interesting option for various sensor network applications. The smart transducer interface provides many features that are required by a fieldbus for automotive or automation industries.

1.2 Scope of this Document

It is the purpose of this document to give an introduction on development of applications on TTP/A smart transducer nodes.

The application is the part of the software that instruments the local sensors or actuators via the node's I/O ports. This software coexists with the TTP/A protocol software. The application exchanges data with the protocol software running on the same node. The protocol application performs the tasks which are required for communicating in the distributed fieldbus network according to the rules specified in the protocol definition.

Readers, who are interested in implementing the protocol software of a smart transducer are referred to [Trö02a].

1.3 Structure of this Document

We will first introduce general protocol features and programming techniques and then explain the particular implementation details for application programming on Atmel-TTP/A nodes with processors from the Atmel microcontroller series.

The remainder of this document is organized as follows:

The following section 2 gives an overview on the TTP/A fieldbus protocol. A TTP/A developer of a local node application will not have to know all features of the protocol in detail in order to develop an application. He mainly has to focus on the interface between the protocol and the local node application. This interface is called the interface file system (IFS), which is also described in this section.

Section 3 describes the programming model of state-of-the-art TTP/A nodes. This section will be essential for a developer.

Section 4 describes the usage of a tool for monitoring and debugging single TTP/A nodes. TTP/A enables the observation of the interface between protocol and local node application without a probe effect.

Section 5 describes the node hardware, plugs, cables, etc. This section will be helpful as a reference for existing nodes or for people who like to develop their own TTP/A node hardware.

Section 6 lists the steps for setting up a development environment for Windows or Linux. We mainly used freeware tools like the *Gnu compiler collection* and tools that have been programmed in-house.

Finally, section 7 summarizes this document.

2 The TTP/A Protocol

TTP/A is a time-triggered field-bus protocol used for the connection of low-cost smart transducer nodes. There is one active master controlling the network. A cluster can have one or more shadow masters which take over if the active master fails.

The master can also serve as gateway to an outer network running TTP/A on a different communication protocol such as TTP/C or TCP/IP or CORBA.

2.1 Capabilities of TTP/A

The TTP/A protocol was designed with the following objectives in mind [Kop01]:

Time-Triggered: TTP/A uses a TDMA (Time Division Multiple Access) scheme to achieve a predictable time behavior. The nodes communicate with each other in predefined slots, so that the origin of the messages are only determined by the time at which they are sent and thus minimizing protocol overhead.

The protocol also synchronizes the slaves' local clocks to the clock of the master. To achieve minimal jitter, it is recommended that the TTP/A protocol task runs with highest priority on a node.

Scalable hardware requirements: TTP/A can be implemented on a variety of nodes beginning on the low end with 8 bit micro-controllers with very limited resources up to 32 bit embedded single board systems which can serve as gateways to higher level protocols such as TTP/C and CORBA.

Small universal transducer interface: The interface of a smart transducer node must be understandable, data-efficient and predictable. The interfaces for smart transducers can be categorized into the following three types:

- The real-time service (RS) interface which provides the timely real-time services to the component during the operation of the system.
- The diagnostic and management (DM) interface which is used to set parameters and to retrieve information about the internals of a component, e.g. for the purpose of fault diagnosis. The DM interface is available during system operation without disturbing the real-time service. Normally the DM interface is not time-critical.
- The configuration and planning (CP) interface which is necessary to access configuration properties of a node. During the integration phase this interface is used to generate the "glue"between the nearly autonomous components. The CP interface is not time critical.

In TTP/A, transducer data is represented as standardized byte-oriented state messages via the RS interface. TTP/A delivers real-time sensor data in a predictable fashion and makes it possible to configure the attached nodes via the CP interface, e.g. set parameters for range selection, alarm limits, signal conditioning, calibration, ... without disturbing the real time communication. The Interface File System (IFS) makes it possible to share data between different TTP/A clusters in a transparent fashion, so that no distinction between data from local and remote nodes is made.

Sensor parameterization/plug and play capability: Nodes which are newly connected to a TTP/A cluster have to be configured before they can be used by the system. TTP/A serves these needs by means of the so called *master/slave rounds* which are periodically scheduled by the master concurrently to a real-time communication service. The bandwidth for each node in a TTP/A cluster is allocated at system design.

Master/slave protocol: In TTP/A there are two types of nodes: low cost slave nodes which handle the transmission of sensor/actuator data and more powerful master nodes which act as an interface between TTP/A and higher level protocols such as TTP/C and CORBA or serve as a monitoring node for a locally attached PC.

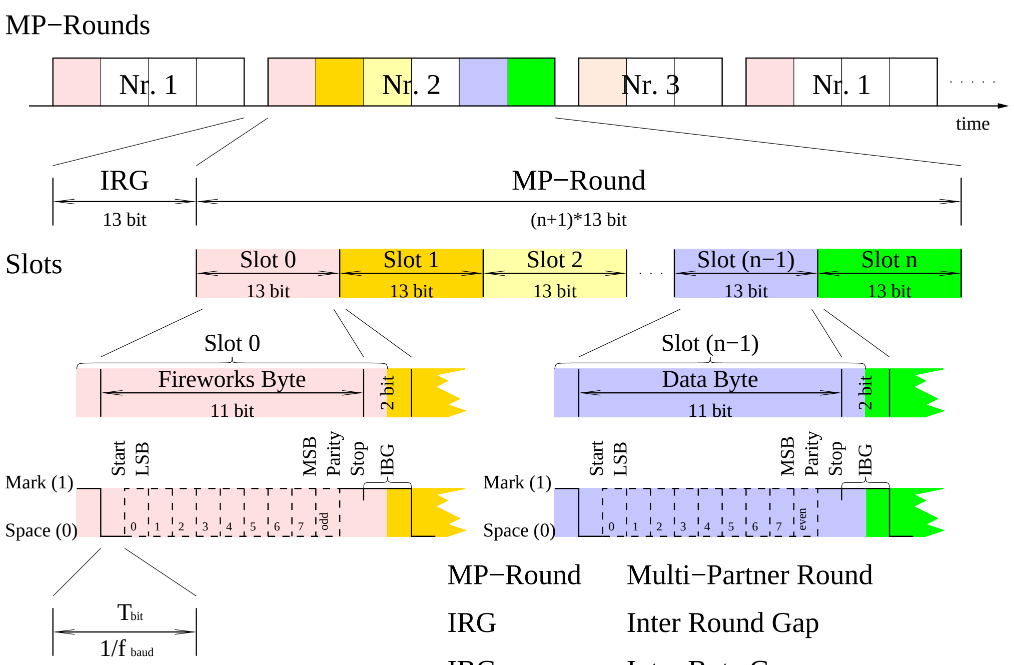
Multi-master protocol: A TTP/A cluster has one active master that controls the network by sending so-called *fireworks bytes* (FB) to start a TTP/A round. One ore more shadow masters may be present. A shadow master takes over control of the cluster in case the the active master has failed.

Flexible physical layers: TTP/A can be used on many different physical layers, from simple single wire interfaces (e.g. the ISO 9141 standard) up to high speed links based on fiber optics.

2.2 Principles of Operation

TTP/A is a time-triggered protocol used for the communication of one active master with or among smart transducer nodes within a cluster. This cluster is controlled by the master, which establishes a common time base among the nodes. In case of a master failure, a shadow master can take over control. Every node in this cluster has a unique alias, an 8 bit (1 byte) integer, which can be assigned to the node a priori or set at any time via the configuration interface.

The TTP/A communication is organized into rounds. Figure 1 shows a round sequence of four subsequent multi-partner (MP) rounds separated by inter round gaps (IRG). IRGs are slots where the TTP/A bus is inactive for at least 13 bit cells. A TTP/A round consists of one or more frames. A frame is a sequence of bytes transmitted by one node. A byte is transmitted in a slot consisting of 13 bit cells (one start-bit, eight data-bits, one parity, one stop-bits and a two bit cell wide inter byte gap (IBG)).



MP−Round	Multi−Partner Round
IRG	Inter Round Gap
IBG	Inter Byte Gap

Figure 1: A TTP/A Round

The rounds are independent from each other. Every round starts with a fireworks frame (FF) sent by the master. The arrival of the fireworks frame is a synchronization event for every node in the cluster and identifies the round. According to the specification of the selected round, the fireworks frame is followed by data frames (DF) of specified length from the specified nodes. Each such frame is described by an entry in the round descriptor list (RODL) in the file-system of the sender and the receiver(s).

Because the slot position at which each communication action takes place is defined a priori, no further communication for bus arbitration is necessary. Figure 2 shows the layout of a TTP/A round.

Beside single speed slots, with a single byte per slot, it is possible to define speedup slots with up to 32 bytes per slot (not supported on all devices), and multiplexed slots which are shared between different nodes.

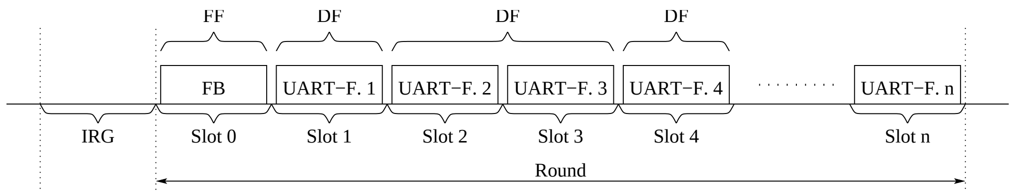


Figure 2: Structure of a TTP/A Round

2.2.1 Data Transmission

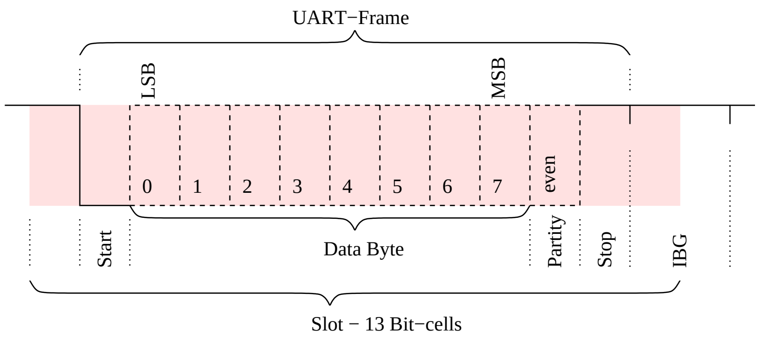


Figure 3: Data Byte

For the transmission of bytes on the TTP/A bus, a standard UART format has been chosen (see figure 3): One start bit, 8 data bits, one parity bit and one stop bit. The parity for data bytes has to be even, whereas for the fireworks byte the parity must be odd. The passive state on the bus is logical 1 (high). The start of a new byte is marked by the falling edge of the start bit. The stop bit (logical 1) is followed by the inter byte gap which is, in the current implementation of TTP/A, 2 bit cells long, for which the bus is also in passive state (logical 1). So the 11 bit cells long UART frame is embedded in a TTP/A timeslot of 13 bit cells times.

The length of the inter byte gap (IBG) depends on the chosen baud rate of the network and its physical size. In faster and more expansive TTP/A networks the length of the IBG might be up to 5 bit cells long.

A new round always is started by a *fireworks byte* (FB). The FB is transmitted with odd parity, in contrast to other data bytes which are sent with even parity. There are only 8 valid fireworks bytes (6 for multi partner rounds and 2 for master/slave rounds; see table 1). The fireworks bytes (protected by a parity bit) have Hamming distance of at least 4.

firework	Meaning	Description
0x78	RODL=0	Multi-Partner Round 0
0x49	MSD	Master/Slave Data Round
0xBA	RODL=2	Multi-Partner Round 2
0x8B	RODL=3	Multi-Partner Round 3
0x64	RODL=4	Multi-Partner Round 4
0x55	MSA	Master/Slave Address Round (startup sync.)
0xA6	RODL=6	Multi-Partner Round 6
0x97	RODL=7	Multi-Partner Round 7

Table 1: Firework codes

The Master/Slave Address (MSA) fireworks byte has been designed to generate a regular bit pattern, which can be used by slave nodes with an imprecise on-chip oscillator for startup synchronization (see figure 4).

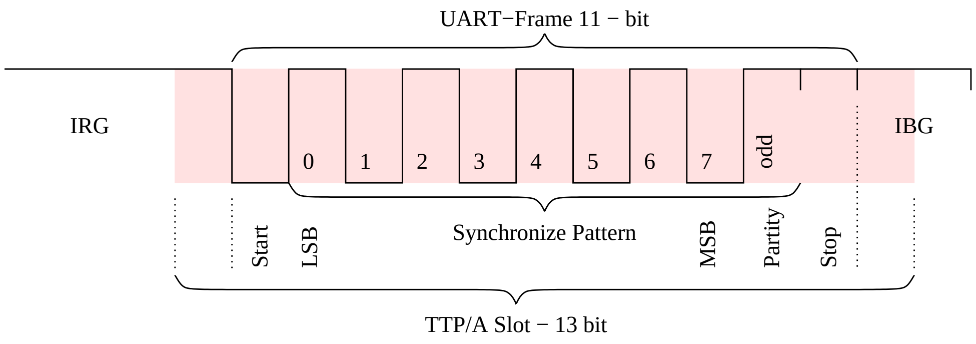


Figure 4: Synchronization Pattern

The three least significant bits (bit 0...2) of the FB denote the round name. The remaining bits (5 data and one parity bit) are used for error detection.

2.2.2 Properties of the Fireworks Byte

The generation of the firework codes had several requirements (see [Hai00]): First, the byte `0x55` must be a part of the code because this regular bit pattern is also used for initial synchronization of the slaves' UARTs. Hamming distance should be maximized and the resistance against burst errors should be optimal. The firework bytes are all sent with odd parity and the lower three bits of the code have to be equal to the round number.

Code Generation: Because `0x55` must be a member of the code, it was impossible to use a cyclic redundant code (CRC) with the requested properties. So the code was created by using an exhaustive search method.

Hamming Distance: The occurring Hamming distances are 4, 6 and 8. So the code will detect all errors of weight less than 4.

Parity: The code includes an odd parity bit. So it will also detect all errors with an odd weight above 4.

Burst Errors: Every possible burst error will be detected by the code

Bulk Errors (force the bus to low or high): It is impossible to get a valid FB by setting (or clearing) one or more adjacent bits. Therefore, it is impossible to corrupt a FB by applying a direct voltage impulse to the bus and get another valid FB.

Error Exposure: Due to the fact, that the byte is protected by a parity bit during transmission, we have an error exposure $\geq \frac{2^9-8}{2^9} \geq 98,43\%$

Error Pattern There exist only 7 error patterns that will not be exposed under certain circumstances. Four of them have weight 4 (`0x02D`, `0x11C`, `0x131`, and `0x1C2`), two error patterns have weight 6 (`0x0DE` and 0x0F3), and one has weight 8 (`0x1EF`).

2.2.3 Types of Rounds

Multi-partner Rounds: A multi-partner round is used to transmit messages over the bus from several nodes in predefined slots. Multi-partner rounds are scheduled periodically by the master. They are used to update real-time images, supporting the real-time view of the cluster and also periodically resynchronizing the slaves' clocks. It is possible to define 6 different multi-partner rounds per cluster.

Master-slave Rounds: The master of a TTP/A cluster can schedule master-slave rounds to read data from an IFS file record, to write data to an IFS file record, or to execute a selected IFS file record within the cluster.

The master-slave address (MSA) round specifies the node, the operation and the local address of the desired data within the addressed node. The master-slave data (MSD) round is used to transmit the data between master and slave.

Broadcast Rounds: A broadcast round is a special form of master-slave (MS) round were the name of the addressed node in the MSA round is set to `0x00`. All nodes except those with node alias `0xFF` are addressed by such a broadcast. Because more than one node is addressed in such a round, only write and execute operations are permitted for broadcast rounds. An example for such a broadcast is the *sleep* command, which puts all nodes in a TTP/A cluster into *sleep mode*.

2.3 The Interface File System

Every TTP/A node has its own local interface file system (IFS) which is the source and destination of the communication data. The IFS of a cluster consists of the local interface file systems of all nodes in it. The IFS also serves as an interface to the application. All relevant data of a node should be mapped into the IFS of this node.

The IFS of a single node can contain up to 64 files with a maximum of 256 four-byte records in each file. The layout of the IFS is statically defined and each file of the IFS can have a different length. The only file which has to be implemented on all nodes is the documentation file (file nr. `0x3D`) which is used to identify the node.

Bit No.	7	6	5	4	3	2	1	0
Byte 0	Persistence		MP-Perm.			MS-Perm.		
Byte 1	Length−1							
Byte 2	reserved				Storage Location			
Byte 3	reserved							

Persistence	0_b00: SRAM, not persistent, 0_b01: SRAM, persistent (exec. Header Rec.) 0_b10: ROM 0_b11. reserved
MP and MS Permission	3 bits for read - write - execute (like Unix permission)
Length−1	Number of records in file decreased by one; index of the last record; index start with 0 (header record)
reserved	These bits are reserved for further enhancements.
Storage Loc.	Implementation specific storage Location.

Figure 5: IFS Header Record

2.3.1 File Structure

A file in the IFS can be viewed as an array of records. However, particular files can have different lengths. Each record can store 4 bytes (32 bits) of data. For each node, up to 64 files can be defined. Every file must have a header record.

2.3.2 The Header Record

The first record of each file, which is the record with index `0x00`, is the header record (see figure 5). This record contains the information about the file. Bit 7 and bit 6 of byte 0 is set according to the persistence of the file. This field is set to 00_b if the contents of the file are located in RAM and are lost if the power of the node fails. The value 01_b specifies that the contents of the file can be written back to persistent memory (e.g., EEPROM) by executing the header record of the file. If this field is set to 10_b the file is located in ROM. The value 11_b is reserved for future extensions. The permission fields for multi-partner and master/slave access specify the allowed access mode for read, write, and execute access similar to Unix permissions. If the corresponding bit is set, the access is granted, otherwise access is denied. Byte 1 gives the length of the file (in records) without the header record. The storage location field in byte 2 is internally used by the TTP/A protocol to support different memory types for storing IFS files. Byte 3 of the header record is reserved.

Byte Nr.	0	1	2	3
Record 0	Header			
Record 1	Physical Name Hi			
Record 2	Physical Name Low			

Header	Header record of the documentation file
Physical Name Hi	Upper 32 bit of the Physical Name
Physical Name Low	Lower 32 bit of the Physical Name

Figure 6: The Documentation File

Code	Meaning
0x00	not synchronized, startup
0x01	passive (no send operation)
0x02	not baptized
0x03	passive lock (no send operation)
0x05	active
0x05	Error

Table 2: TTP/A protocol status

2.3.3 Special files

The Documentation File: Every TTP/A node must contain a documentation file (file nr. `0x3D`). This file is used to identify the node. The first and the second record after the header record contains the unique physical name of the node which is an 8 byte (64 bit) integer (see figure 6). This ID is stored in network order / big endian format (The MSB is stored in the lowest byte)

The documentation file is always set to read only. Additional information about this node can be added after the ID.

The Configuration File: The configuration file (file nr. 0x08; see figure 7) holds the current logical name of the node and is also used by the *baptizing* algorithm (see section 2.5) and with the *sleep* command.

Record `0x01` holds in byte 3 the current logical name and in byte 2 the new logical name used by the *baptize* algorithm. Byte `0x01` is the cluster name the node belongs to and byte `0x00` holds the current status of the node (see table 2).

The records `0x02` and `0x03` contain the ID compare value for the baptize algorithm. Record `0x04` holds the identifier for the currently executed multiplexed round (byte `0x00`), the fireworks-byte of the current round (byte `0x01`), the epoch counter (byte `0x02`), and the slot counter (byte `0x03`). Record `0x05` is used for the sleep command.

Byte Nr.	0	1	2	3
Record 0	Header			
Record 1	Status	Cluster N.	New LN	Curr. LN
Record 2	Cmp. Phys. Name Hi			
Record 3	Cmp. Phys Name Low			
Record 4	cmux	crnd	ectr	sctr
Record 5	Sleep			

Header	Header record of the configuration file
status	status of protocol (see table 2)
Cluster N.	Name of cluster to which the node belongs
New LN	New logical name used by the baptize algorithm
Curr. LN	The current logical name of the node
Cmp. Phys. Name Hi	Used for comparison by the baptize algorithm
Cmp. Phys. Name Low	Used for comparison by the baptize algorithm
cmux	multiplexed round identifier
crnd	current round
ectr	Epoch counter
sctr	slot counter
Sleep	Record used by the sleep command

Figure 7: The mandatory config File

The Membership File: It makes sense to implement this file on gateway or master nodes while ordinary slave nodes will not benefit from the information gathered in this file. The Membership file (file nr. `0x09`; see figure 8) contains two membership vectors of 256 bits (32 byte) each. The logical name of each node is interpreted as an index to the 256 bits in the membership vector. The first vector contains all slaves which have sent a live-sign during the last multi-partner round. The second membership vector contains all slaves which have responded to the most recent MS operation. To update the second membership vector the master may fill empty MS slots by issuing read operations to the slaves documentation file, which per definition must be present in every TTP/A node. The lowest bit of each of the membership vectors refers to the highest Logical Name (0xFF).

The Round Sequence (ROSE) file: Only a master will benefit from the implementation of the ROSE file. The ROSE file (file nr. `0x0A`) specifies the sequence in which rounds are scheduled by the master. In addition start time and duration of this sequence can be specified.

The ROSE file is divided into three sections. The first section is the status record (record nr. `0x01`; see figure 9). The second and the third section each contain a sequence of round names. At any given time, only one of the two sequences is active, the other is inactive. The first byte of the status record defines which section is active. The second and the third byte contain the start record of section two and three.

Byte Nr.	0	1	2	3
Record 0	Header			
Record 1	First Membership Vector			
Record 2				
Record 3				
Record 4				
Record 5				
Record 6				
Record 7				
Record 8				
Record 9	Second Membership Vector			
Record 10				
Record 11				
Record 12				
Record 13				
Record 14				
Record 15				
Record 16				

Header	Header record of the membership file
First Membership Vector	Membership vector of last sequence period
Second Membership Vector	Membership vector of master-slave responds

Figure 8: The Membership File

Byte Nr.	0	1	2	3
Record 0	Header			
Record 1	Active Sec.	Start Sec. 2	Start Sec. 3	

Header	Header record of the ROSE file
Active Sec.	0: section two is active; 1: section three is active
Start Sec. 2	start record of section two
Start Sec. 3	start record of section three

Figure 9: Status Record of the ROSE File

Byte Nr.	0	1	2	3
Record 2	Start			
Record 3	Time			
Record 4	Period			
Record 5	Time			
Record 6	Round Name	IRG length		
Record 7	Round Name	IRG length		
	...			

Start Time	Start time of the round sequence (in 64 bit GPS format described in [Obj01])
Period Time	Length of the round sequence
Round Name	Bits 0 ... 2 specify the name of the round, bit 7 is set if this is the last round in the sequence
IRG length	sets the length of the IRG following the specified round. Must be in the range from 1 to 15.

Figure 10: Section two of the ROSE File

Sections two and three have the following format (see figure 10): The first two records hold the start time of the sequence as an 64 bit (8 byte) integer value with a granularity of 2^{-24} s. The epoch starts at the epoch of GPS time (January 6, 1980) plus an offset of 2^{38} seconds. The next two records hold the period time of the whole sequence. All further records in the section define a scheduled round. In the first byte of such a record, the three LSB define the round. If the MSB is set, this entry is the last in this sequence. The second byte contains in the lower 4 bits the number of slots in the inter round gap. Valid entries are `0x01`, `0x02`, ..., `0x0F`.

The first entry of each sequence must be a MSA entry and every MSA - entry must have a complementary MSD entry.

2.3.4 Round Description List (RODL) files

The RODL file contains the information about the actions performed by a node for a particular round. The file names of the RODL files correspond to the round name.

A RODL entry (see figure 11) defines which type of operation is performed in up to 16 subsequent slots of a TTP/A round. The first byte of the entry defines the file, and in the case of a single speed frame, the offset within a record. For speedup frames, the offset value is used as speedup code (see table 3).

The second byte defines the record of the address. For read and write operations, these two bytes define the location from which the data is read from or to which it is written to. In case of an execute operation, the file name defines the task that corresponds with the given file name and the record and alignment fields are supplied as parameters to the called task.

Bit No.	7	6	5	4	3	2	1	0
Byte 0	File Name						Align/speedup	
Byte 1	Record Name							
Byte 2	OP – Code				Frame Length−1			
Byte 3	Mux		Slot Position					

File Name	Name of File (0-63)
R-align	Record Alignment or speedup depending on OP-Code(0-3)
Record Name	Record Name (0-255)
Frame Length−1	Length of Frame - 1
OP - Code	OP code of operation (see table)
Mux	Multiplex Code
Slot Position	Position of Slot in Round

Figure 11: RODL Entry

Speedup – Code	Multiplier
0_b00	4
0_b01	8
0_b10	16
0_b11	32

Table 3: Speedup code

The third byte holds the operation and length field of the frame. The operation code is explained in table 4. The length field gives the length in slots minus one. Table 4 shows that the meaning of

OP – Code	Description	Align/speedup	Mux
0_b0000	receive	align	–
0_b0001	send	align	–
0_b0010	receive with sync.	align	–
0_b0011	execute (Rec = task Param.)	–	–
0_b0100	receive with speedup	speedup	–
0_b0101	send with speedup	speedup	–
0_b0110	End-Of-Round (EOR)	–	–
0_b0111	execute and EOR	–	–
0_b1000	receive mux. (4 times)	align	mux
0_b1001	send mux. (4 times)	align	mux
0_b1010	receive mux. (8 times, MSB = 0)	align	mux
0_b1011	send mux. (8 times, MSB = 0)	align	mux
0_b1100	receive with speedup + mux. (4 times)	speedup	mux
0_b1101	send with speedup + mux. (4 times)	speedup	mux
0_b1110	receive mux. (8 times, MSB = 1)	align	mux
0_b1111	send mux. (8 times, MSB = 1)	align	mux

Table 4: RODL entry operation codes

the *align/speedup*, and the *mux* field depends on the operation code of the RODL entry. The *mux* field is only evaluated if a multiplexed (either 4 or 8 times) operation (*send* or *receive*) is performed. The *align/speedup* field is normally used as an alignment parameter for *receive* and *send* operations. It is used as speedup parameter (see table 3) if a speedup operation is performed.

The last byte holds the MUX code and the slot position in which the operation starts.

2.4 Master/Slave Rounds

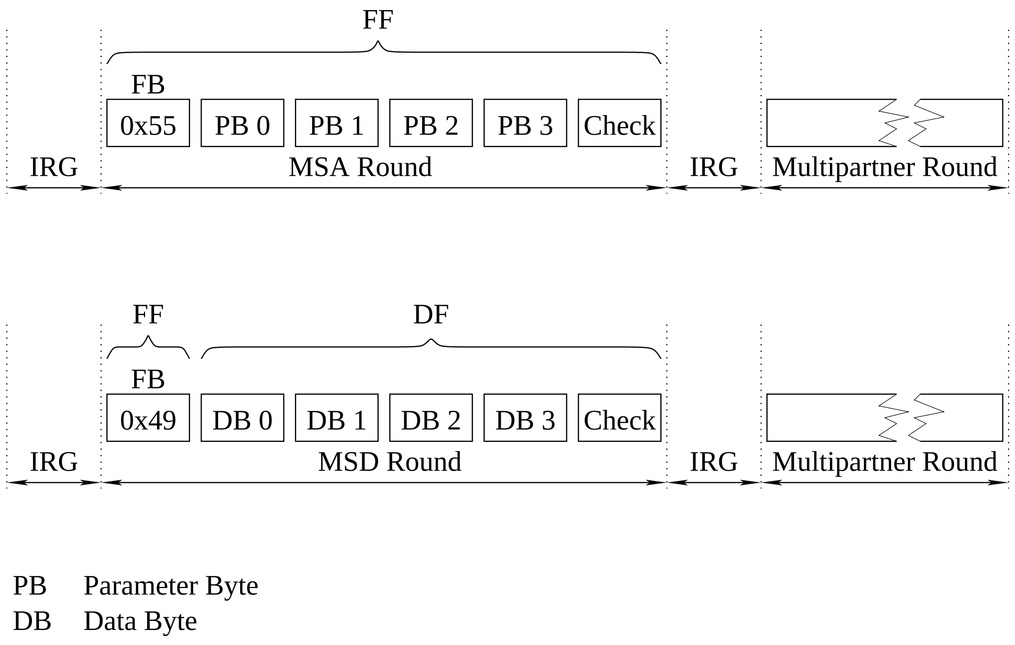


Figure 12: Master-slave round

The master-slave (MS) round is used by the master of a cluster to read data from an IFS file record, to write data to an IFS file record, or to execute a selected IFS file record within the cluster.

A master-slave round is divided into two separate phases (see figure 12). The first is the master-slave-address (MSA) round. During this phase, the master specifies (in a message to the slave node) which type of operation is intended (read, write, or execute) and the address of the selected file record. The format of the master-slave address round is shown in figure 13. The state of an issued master-slave-address round remains active until the arrival of a new master-slave-address round.

The MSA round consists of six bytes (see figure 12). PB0 contains an epoch counter which provides the current epoch of the cluster internal time base. PB1 is the name of the addressed node. PB2 specifies the operation type in bits 6 and 7 and the file which is addressed in the 6 low order bits. PB3 names the record, which is read from or written to. The parameter bytes are followed by a check byte. This check byte is calculated as an exclusive or conjunction over the parameter bytes

BitNr.	7	6	5	4	3	2	1	0
PB 0	Epoch							
PB 1	Logical Name							
PB 2	File Name						OP - Code	
PB 3	Record Name							
Check	Checkbyte							

Epoch	identifies the current epoch
Logical Name	logical name of the addressed node
OP - code	specifies the operation which is performed (read, write or execute)
File Name	name of the addressed file
Record Name	name of the addressed record
Checkbyte	Checksum of the frame

Figure 13: Parameters in a MSA Round

and the FB.

In a subsequent MSD round, the master sends the firework which indicates that it is either transmitting the record data or is waiting for the slave to transmit the requested record data, depending on the operation specified in the previous MSA round. The message in the data phase also consists of six bytes: The FB, four data bytes, and one check byte which is calculated the same way as in the MSA - round.

2.5 The Baptizing Algorithm

Until a logical name has been assigned to a node, it does not take part in the multi-partner rounds. The baptize algorithm [Elm02] is executed by the master to see which nodes are connected to the TTP/A bus and to assign each of them a logical name, which is unique in this TTP/A cluster.

This mechanism performs a binary search on all physical node names. A physical name is unique for every TTP/A node within the entire universe of TTP/A nodes. The identification of a new node takes 64 iterations. The master has to keep three 64 bit integer values, `lo`, `hi`, and the comparison identifier `ci`. The values of `lo` and `hi` are only used internally to calculate the new `ci` values.

The variables of `lo` and `hi` are initialized to the minimum and maximum value of the expected identifiers and move towards each other, until `lo` equals `hi`. In this case, the identifier of a node is found and the master assigns a new alias to this node. This is done by the *baptize* Operation.

The identifier comparison is done as follows:

- First, a lower limit for the node identifier is set by the master. This is done by performing a Master/Slave write operation to the records `0x02` and `0x03` of the mandatory config file

(0x08).

- Then the master requests an execute command on this records for all unbaptized nodes (all with logical name 0xFF). The action which is assigned with this execute operation is the comparison of the slave's own unique identifier to the comparison identifier.

- In the corresponding MSD round of this Master/Slave round, all unbaptized nodes whose own identifier is higher or equal than the comparison identifier write a data frame with content 0x00 in the first slot.

  If no node responds to this command, then the value of hi is set to the value of ci, otherwise lo is set to this value. If the value of ci and hi were equal and a node has responded in this round, then the identifier of the new node has been found.

After finding the unique identifier of a node, a new logical name must be assigned to this node. First, the new alias is written into byte 0x02 of record 0x01 of the mandatory config file by a Master/Slave write operation. Because this operation addresses all unbaptized nodes, in a second step, an execute operation is performed on this record. In this execute operation, only the node whose unique identifier equals the comparison identifier, copies this new logical name to its own logical name which is located in byte 0x03 of this record.

After this operation, the node takes part in multi-partner rounds and responds to Master/Slave rounds with this new logical name.

3 Programming Model

3.1 Software Architecture of the TTP/A Node

The software that controls a TTP/A node consists of the protocol code (implemented entirely in Assembler) and the application software which can be written in C, Atmel AVR Assembler, or both (see figure 14).

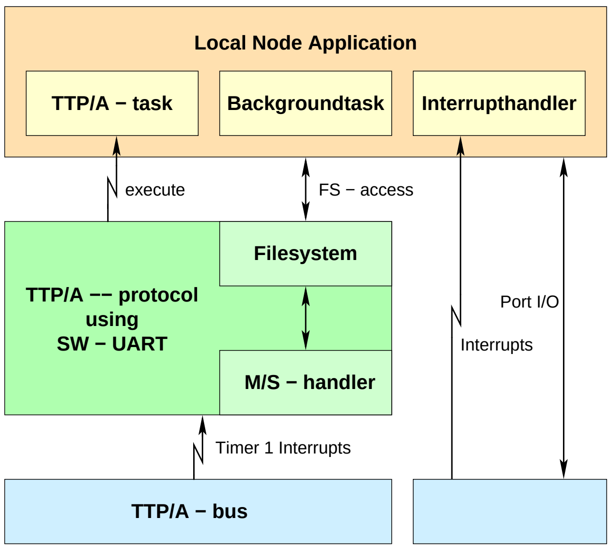


Figure 14: Software structure of a TTP/A - node

The application software itself consists of TTP/A - task functions which are executed synchronous in so called *execute* - slots of a RODL. For non time-critical tasks, background tasks can be used, which are scheduled in a simple round-robin schedule if the microcontroller is idle. Interrupt handlers also can be defined, providing a way to react on external events, timer matches, hardware UART and so on, as long as interrupts are re-enabled after entering the the interrupt handler in order to keep the node's responsiveness on protocol activities.

3.1.1 Task Structure

The task structure defines three priority levels: The protocol code with the highest priority, the time-triggered TTP/A tasks and background tasks which uses up the remaining idle time of the processor.

Protocol: The protocol uses a 16 bit timer and the corresponding *capture*, *match*, and *overflow* interrupts.

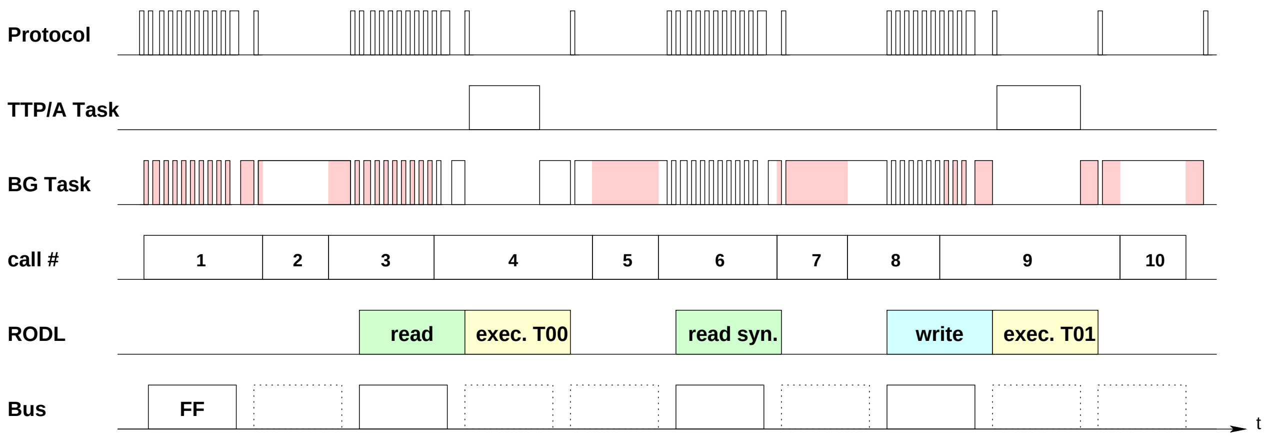


Figure 15: Execution of different tasks

TTP/A - Tasks: TTP/A - tasks (tasks which are scheduled in certain slots of the TTP/A round) and the corresponding TTP/A file are defined as shown in the following example:

```
struct myfile_struct {
        ifs_uint8_t b[4];
        ifs_int32_t l;
};

struct myfile_struct IFS_LOC(ifs_int_0) myfile;

IFS_ADDAPPLFILE(0x10, &myfile, mytask, IFS_FILELEN(struct myfile_struct), ifs_int_0, 077);

void mytask(ttpa_taskparam_t param)
{
        myfile.b[0]++;
}
```

This defines the TTP/A file `0x10` which contains an array of unsigned bytes (one record) and a signed 32 bit integer. Additional the corresponding TTP/A task is defined. Available data types are `ifs_int8_t` (signed int, 8 bit), `ifs_uint8_t` (unsigned int, 8 bit), `ifs_int16_t`, `ifs_uint16_t`, `ifs_int32_t`, `ifs_uint32_t`, and `ifs_float32_t`.

Background Tasks: If the processor is idle and no other task has to be executed (neither TTP/A protocol activity, nor active interrupt handler, nor TTP/A - task), background tasks will be scheduled by a simple, non preemptive round-robin scheduler (with priorities).

Figure 15 shows that the RODL contains commands related to instants of TTP/A slots. Protocol tasks for bus communication and task scheduling have the highest priority (interrupt level). Time-triggered (TT) tasks are second highest level and the background (BG) tasks have the lowest level. In the figure one can see how a background task is preempted by the TTP/A protocol code and two different TTP/A - tasks, which have higher priority. This leads to differences in the duration of execution of subsequent background task calls. The line with the background call numbers show, that the execution time of BG tasks is heavily affected by protocol and TTP/A tasks. So a BG task should not contain tasks with hard deadlines.

The following code shows an example of how to define a background task:

```
#include "schedule.h"

int mybgtask()
{
        static uint8_t cnt = 0;

        // increment counter
        cnt++;
// re-schedule task until cnt > 10
        return cnt > 10 ? 0 : 1;
}
```

Note: A background task may be preempted during its execution, thus the memory state could have changed due to the interrupting task (see figure 15).

Interrupt handlers: Beside the TTP/A task functions, handler for interrupts, which are not used by the protocol, can be defined. This can be used e.g. to define your own timer functions:

```
INTERRUPT(SIG_OVERFLOW0)
{
  /* blink every 0.5 second -> 7.3728MHz / 64 = 115.2kHz */
  /* switch lights on/off every 225 interrupts */
  if(blink_counter < BLINK_COUNT)
  {
    ++blink_counter;
  }
  else
  {
    /* code to blink LED */
  }
}
```

To guarantee that the TTP/A protocol code has the highest priority, only interrupt service routines (ISR) which enable the global interrupt flag after entering the function can be used. The macro `SIGNAL(<interrupt-vector>)` must not be used to define an interrupt handler for an application because it will block all interrupts (also the Timer 1 interrupts used by the TTP/A - protocol) and destroys real-time behavior. This has some drawbacks in cases were the interrupt condition is not automatically cleared by executing the interrupt vector (e.g. for the *UART receive complete* interrupt).

3.1.2 TTP/A Protocol Code

The TTP/A protocol code consists of the startup synchronization, the TTP/A state machine, the software (SW) UART, the interface file system (IFS), the code handling the master-slave rounds, and some additional functions.

TTP/A protocol state machine: The TTP/A protocol is implemented using a state machine (see figure 16).

The protocol starts in the state *UNSYNC*. On the master, it looks up the first ROSE entry and after sending the first MSA FB, the state is set to *ACTIVE*. On the slave, the first action is to wait for a IRG. After a (supposable) IRG has been found, the protocol tries to synchronize to the next sent byte. If the received byte is a MSA Fireworks byte, the state is changed to *PASSIVE*. In this state, the node verifies the configuration by listening to the transmitted frames on the bus and comparing these to its own RODL files. If a mismatch is detected, the state is set back to *UNSYNC*. Depending on the *Current Logical Name* of the node, the protocol state is set to *ACTIVE* on reception of a MSA FB, if the CLN is not equal `0xFF` (the node is already baptized), or set to *UNBAPT* if the CLN is set to `0xFF`. The node remains in *UNPAPT* state until it is baptized by the master. After baptizing of the node the state is set to *PASSIVE_LOCK*. This state is intended for unconfigured nodes without valid RODL files (the node only responds to MS requests). If the master has configured the node, the status is set to *ACTIVE* by the master.

Startup Synchronization: After a reset the protocol code calls the function `bus_sync()`. This function sets a timer to the length of an IRG and waits until this timer expires and proceeds to the function `bus_foundirg()`. If it has not expired until the next edge is detected on the TTP/A-bus then the synchronization procedure is started again. The input capture interrupt is set up to call

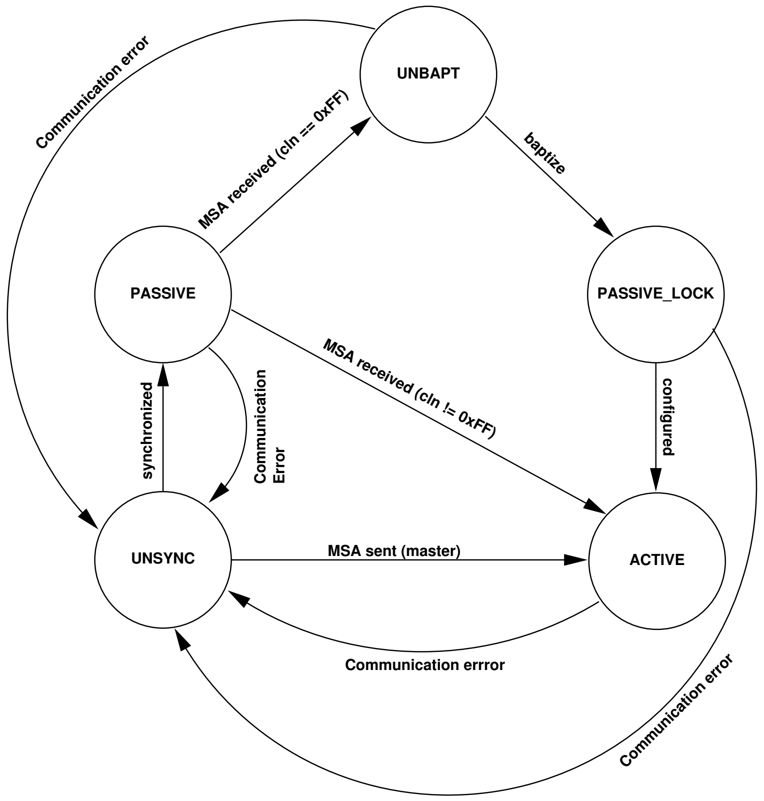


Figure 16: State diagram of the TTP/A - protocol

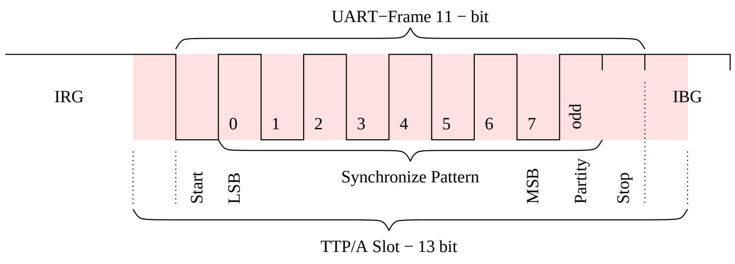


Figure 17: Synchronization Pattern

`bus_foundstart()` to detect the edge of the start-bit of a possible synchronization pattern (`0x55` sent with odd parity; see figure 17). The protocol then proceeds with measuring the time between falling and rising edges (function `bus_measuresync()`). It sets a timeout of 1.5 times the length of the start-bit after the latest edge of the synchronization pattern. Whenever a time between two edges exceeds this value a timer match interrupt will occur and the synchronization is restarted. This prevents the protocol code form staying in this state for too long and missing the next IRG, if the time between the edges was too long to represent a valid synchronization pattern. On each edge, the length of the current bit will be compared with its predecessor. If they differ fundamentally then the detected pattern is not a valid synchronization pattern and the process is started again.

After nine edges have been measured correctly, the duration of one bit cell and one slot are calculated and the protocol, because the synchronization pattern is the FB of the MSA round, begins to interpret the RODL file 5 which is the MSA RODL file.

Software UART:

3.1.3 Interface File System

The IFS is implemented using an array of pointers in the Flash ROM (one pointer for each file), pointing to a `struct` also located in Flash, containing the header record of the file and a pointer to the actual IFS file, which can be located in SRAM or Flash. The location is described by the Storage Location field in byte `0x02` of the header record (see figure 5).

3.1.4 Master-Slave Rounds

A master-slave round consists of two parts: The master-slave address (MSA) round and the master-slave data (MSD) round. The RODL files of the MSA and the MSD round are executed like in any other round with the following exceptions: On the end of the MSA round, the protocol code executes a special subroutine, which calculates the checksum for the received master-slave address. If the logical name (LN) in the address equals the name of the node, than the RODL entry of the MSD round is modified according to the requested operation (an IFS *read* operation is set on a master-slave *read* request, an IFS *write* operation is set on a master-slave *write* request) . If the requested operation was an IFS *read* operation, than the requested data is prepared in the MSD data file. Furthermore the check sum for the MSD data is calculated. If a master-slave *execute* was requested, than the appropriate task is executed.

After reception of the MSD FB (`0x49`) it is decided, whether this or another slave was addressed. If another slave was addressed, than five empty slots are scheduled. Otherwise the MSD RODL file is interpreted as usual. If the given operation was a master-slave *write*, than the check-byte is checked and if it is OK the record is transfered to the slave's IFS.

3.1.5 Additional Functions

Some optional code parts can be added to it, if required by the application.

Baptize: The functions, which are needed on the slave for the *plug and play* features of the TTP/A protocol to work properly are implemented as a separate task. This task carries out the required comparison operations, which are described in the *baptizing* algorithm (see section 2.5).

Monitoring: Monitoring of a node via RS232 is done by a background task which receives monitoring requests via the HW UART and carries out the requested operation. See also section 4 of this document.

3.2 RODL - File Format

The RODL files are used to describe the temporal order of send and receive operations on the bus and the execution of time-triggered tasks. From the eight possible RODL files, only six are available for the application. RODL file `0x05` and RODL file `0x01` are used for the MSA and MSD round.

For defining RODL entries, the following C macros should be used (defined in `ifs_rodl.h`:

`IFS_RE_RECV(re_sl_pos, re_addr, re_f_len)` receive from bus

`IFS_RE_SEND(re_sl_pos, re_addr, re_f_len)` send to bus

`IFS_RE_RECVSYNC(re_sl_pos, re_addr, re_f_len)` receive from bus with synchronization to the master

`IFS_RE_EXEC(re_sl_pos, re_addr)` execute task

`IFS_RE_RECV_SPDUP(re_sl_pos, re_addr, re_f_len, re_spdup)` receive from bus with speedup

`IFS_RE_SEND_SPDUP(re_sl_pos, re_addr, re_f_len, re_spdup)` send to bus with speedup

`IFS_RE_EOR(re_sl_pos)` end of round

`IFS_RE_EXEC_EOR(re_sl_pos, re_addr)` end of round with execution of task

`IFS_RE_RECV_MUX4(re_sl_pos, re_addr, re_f_len, re_mux)` receive from bus every fourth round

`IFS_RE_SEND_MUX4(re_sl_pos, re_addr, re_f_len, re_mux)` send to bus every fourth round

`IFS_RE_RECV_MUX8(re_sl_pos, re_addr, re_f_len, re_mux)` receive from bus every eighth round

`IFS_RE_SEND_MUX8(re_sl_pos, re_addr, re_f_len, re_mux)` send to bus every eighth round

`IFS_RE_RECV_SPDUP_MUX4(re_sl_pos, re_addr, re_f_len, re_spdup, re_mux)` receive from bus with speedup every fourth round

`IFS_RE_SEND_SPDUP_MUX4(re_sl_pos, re_addr, re_f_len, re_spdup, re_mux)` send to with speedup every fourth round

Beside these all purpose entries, there are two entries defined as shortcut to update the round specific mux counter:

`IFS_RE_MUX_MASTER(re_sl_pos)` master side

`IFS_RE_MUX_SLAVE(re_sl_pos)` slave side

These macros can be used to define RODL files as shown in the following example:

```
#include <stdio.h>
#include "ifs.h"
#include "ifs_types.h"
#include "ifs_rodl.h"
#include "ttpa_task.h"

/* define RODL file for round 0 */
IFS_RODLFILE(0x00, 4, ifs_int_eep)
{
        /* receive one byte in slot 1 (write to file 0x10, rec. 0x01,
           offs. 0x00 */
        IFS_RE_RECV(0x01, IFS_ADDR_I(0x10,0x01,0x00), 0),
        /* execute task 0x10 in slot */
        IFS_RE_EXEC_EOR(0x02, IFS_ADDR_I(0x10,0x01,0x00)),
        /* send two bytes in slot 0x03 */
        IFS_RE_SEND(0x03, IFS_ADDR_I(0x10,0x03,0x02), 0),
        /* end of round in slot 0x05 */
        IFS_RE_EOR(0x05)
};
```

3.3 Interface between Protocol and Application

3.3.1 Accessing the Interface File System

An IFS file is created by defining the corresponding C `struct` and using the `IFS_ADDAPPLFILE()` macro. Concurrently with the definition of the IFS file, the corresponding task has to be defined, as shown in the following example.

```
#include <stdio.h>
#include "ifs.h"
#include "ifs_types.h"
#include "ttpa_task.h"

struct myfile_struct {
        ifs_uint8_t foo1;
        ifs_uint8_t foo2;
        ifs_int16_t foo3;
        ifs_float32_t bar;
};

void mytask(ttpa_taskparam_t param);

void mytask(ttpa_taskparam_t param)
{
        /* do something */
}

IFS_ADDAPPLFILE(0x10, &myfile, mytask, IFS_FILELEN(struct myfile_struct),
                ifs_int_0, 077);
```

This example generates the IFS file `0x10` holding two unsigned, 8 bit integers named `foo1` and `foo2`, one signed 16 bit integer named `foo3`, and a float named `bar`.

Since the the byte-order of the node and that of the IFS differs, each access must use the macros defined in `ifs_types.h` With these macros, the previous defined file can be accessed by the application by using the `mystruct` structure:

```
void mytask(ttpa_taskparam_t param)
{
        int16_t x;
        float y;

        /* ... */

        /* read from IFS file */
        x = IFS_TO_INT16(myfile.foo3);

        /* ... */

        /* write to IFS file */
        myfile.bar = FLOAT32_TO_IFS(y);
}
```

For accessing other files than the own application file, the IFS functions defined in `ifs.h` can be used. Therefore, the file has to be opened first:

```
        ifs_fd_t fd;
        ifs_addr_t addr;
        uint8_t my_u8;
        int16_t my_i16;

        /* ... */
        addr.filen = 0x11; // filename
        addr.rec = 0x03; // record
        addr.align = 0x02; // alignment
        if(ifs_open(&fd, addr, IFS_REQ(IFS_ACCESS_APPL,IFS_OP_RW, 1) ==
           IFS_ERR_SUCC) {
                /* do file operations */
                /* read 16 bit int at offset 0 */
                my_i16 = ifs_rd_i16(&fd, 0);
                /* write 8 bit int to offset 2 */
                ifs_wd_u8(&fd, 2, my_u8);
                if(fd.error != IFS_ERR_SUCC) {
                        /* error handling */
                }
        } else {
                /* error handling */
        }

        /* ... */
```

These functions also handle the conversion of 16 and 32 bit integers from the big endian format which is used by the IFS to the native little endian format on Atmel AVR.

The IFS address is defined as a union of a 16 bit unsigned integer with a struct containing the filename `filen`, record number `rec`, and record alignment `align`:

```
typedef union {
        struct {
                unsigned align : 2;
                unsigned filen : 6;
                uint8_t rec;
        };
        uint16_t i;
} ifs_addr_t;
```

The usage of the 16 bit integer representation of the struct as parameter `addr` in the IFS function makes it possible to generate very efficient code for reading and writing to constant IFS addresses. Therefore the macro `IFS_ADDR_I(fn, rec, al)` has been defined which takes the addressing parameters and generates a 16 bit integer representation of the IFS address:

```
void some_func()
{

        /* open file */
        if(ifs_open(&fd, IFS_ADDR_(0x21, 3, 2)),
           IFS_REQ(IFS_ACCESS_APPL,IFS_OP_RW, 1) == IFS_ERR_SUCC) {
               /* do something */
        } else {
               /* error handling code */
        }
}
```

The error codes (see table 5), which are returned by the IFS functions, are the same as defined in the description of the Smart Transducer Interface OMG standard [Obj01]:

For local IFS access, only the codes *IFS_ERR_SUCC*, *IFS_ERR_NOFILE*, *IFS_ERR_NOREC*, *IFS_ERR_DAMMGD*, *IFS_ERR_RO*, and *IFS_ERR_NOTSUPP* [1] are of interest.

The following functions are available for accessing IFS files:

`ifs_err_t ifs_open(ifs_fd_t *fd, ifs_addr_t addr, ifs_request_t req)` open an IFS file

`ifs_err_t ifs_close(ifs_fd_t *fd)` close an IFS file

`uint8_t ifs_rd_u8(ifs_fd_t *fd, uint16_t offs)` read 8 bit unsigned integer from IFS

`int8_t ifs_rd_i8(ifs_fd_t *fd, uint16_t offs)` read 8 bit signed integer from IFS

`uint16_t ifs_rd_u16(ifs_fd_t *fd, uint16_t offs)` read 16 bit unsigned integer from IFS

[1] *IFS_ERR_NOTSUPP* is returned if a header record is addressed and the requested length goes beyond the boundary of this record. This is because in the current implementation, the header records of all files are located in a single array in ROM, but the data records of the corresponding file can be located in different location in SRAM or ROM.

No.	Name	Description
0	IFS_ERR_SUCC	no error
1	IFS_ERR_NOCLSTR	addressed cluster does not exist
2	IFS_ERR_NONODE	addressed node does not exist
3	IFS_ERR_NOFILE	addressed file does not exist
4	IFS_ERR_NOREC	addressed record does not exist or requested length would go beyond end of file
5	IFS_ERR_DAMMGD	file was damaged
6	IFS_ERR_NOTREADY	node was not able to fulfill the requested operation in the given time.
7	IFS_ERR_NOEXEC	Task could not be executed
8	IFS_ERR_RO	file or record is read only
9	IFS_ERR_NOMSG	no message from node
10	IFS_ERR_COMM	communication error
11	IFS_ERR_TIME	timing error
12	IFS_ERR_NOTSUPP	operation is not supported

Table 5: Error codes defined by Smart Transducer Interface OMG standard

`int16_t ifs_rd_i16(ifs_fd_t *fd, uint16_t offs)` read 16 bit signed integer from IFS

`uint32_t ifs_rd_u32(ifs_fd_t *fd, uint16_t offs)` read 32 bit unsigned integer from IFS

`int32_t ifs_rd_i32(ifs_fd_t *fd, uint16_t offs)` read 32 bit signed integer from IFS

`float ifs_rd_f(ifs_fd_t *fd, uint16_t offs)` read float from IFS

`ifs_err_t ifs_rd_blk(ifs_fd_t *fd, uint8_t *dest, uint16_t offs, uint16_t len)` copy memory block from IFS

`ifs_err_t ifs_wr_u8(ifs_fd_t *fd, uint16_t offs, uint8_t ui)` write 8 bit unsigned integer to IFS

`ifs_err_t ifs_wr_i8(ifs_fd_t *fd, uint16_t offs, int8_t ui)` write 8 bit signed integer to IFS

`ifs_err_t ifs_wr_u16(ifs_fd_t *fd, uint16_t offs, uint16_t ui)` write 16 bit unsigned integer to IFS

`ifs_err_t ifs_wr_i16(ifs_fd_t *fd, uint16_t offs, int16_t ui)` write 16 bit signed integer to IFS

`ifs_err_t ifs_wr_u32(ifs_fd_t *fd, uint16_t offs, uint32_t ui)` write 32 bit unsigned integer to IFS

`ifs_err_t ifs_wr_i32(ifs_fd_t *fd, uint16_t offs, int32_t ui)` write 16 bit signed integer to IFS

`ifs_err_t ifs_wr_f(ifs_fd_t *fd, uint16_t offs, float ui)` write float to IFS

`ifs_err_t ifs_wr_blk(ifs_fd_t *fd, uint8_t *src, uint16_t offs, uint16_t len)` copy memory block to IFS

3.3.2 Time-Triggered Tasks

The following code example shows how to implement a simple task that adds a configurable increment to a given integer.

```
/*
 * appl.h
 *      Example application for TTP/A (definitions)
 *
 */
#include "ttpa.h"
#include "ifs.h"
#include "ifs_types.h"

#define APPL_FN 0x30
#define APPL_SEC ifs_int_eep

struct applfile_struct {
        ifs_addr_t fileaddr; // IFS address for IO file
        ifs_int8_t inc;      // increment value
        ifs_uint8_t error;   // error flag (0: no error)
};

#define IO_FN 0x10
#define IO_SEC ifs_int_0

struct iofile_struct {
        ifs_int8_t value[8];
};

extern void mytask(ttpa_taskparam_t param);
```

```
/*
 * appl.c
 *      Example application for TTP/A
 *
 * The task reads the IFS address from the first entry in the application
 * file and opens the corresponding file. It then reads a 8 bit integer
 * from the opened file and adds the second entry in the first file.
 * The result is written back to the seconf file.
 *
 */
#include <stdio.h>
#include "ttpa.h"
#include "ifs.h"
#include "ifs_types.h"
#include "appl.h"

struct applfile_struct IFS_LOC(APPL_SEC) applfile = {
        IFS_ADDR_S(IO_FN, 0x01, 0x00), 1, 0};

struct iofile_struct IFS_LOC(IO_SEC) iofile;

IFS_ADDAPPLFILE(APPL_FN, &applfile,
        mytask, IFS_FILELEN(struct applfile_struct), APPL_SEC, 077);

IFS_ADDAPPLFILE(IO_FN, &iofile, NULL,
        IFS_FILELEN(struct iofile_struct), IO_SEC, 066);

void mytask(ttpa_taskparam_t param)
{
        ifs_fd_t io_fd;
        ifs_addr_t io_addr;
        int8_t data;

        io_addr = applfile.fileaddr;

        // use record parameter as offset into IO file for e.g., MS access
        if(param.rec < 2) {
                io_addr.rec += param.rec;
        }
        if(ifs_open(&io_fd, io_addr, IFS_REQ(IFS_ACCESS_APPL,IFS_OP_RW,1)) ==
           IFS_ERR_SUCC) {
                data = ifs_rd_i8(&io_fd, 0);
                if(io_fd.error == IFS_ERR_SUCC) {
                        data += IFS_TO_UINT8(applfile.inc);
                        if(ifs_wr_i8(&io_fd, 0, data) == IFS_ERR_SUCC) {
                                applfile.error = UINT8_TO_IFS(0);
                        } else {
                        applfile.error = UINT8_TO_IFS(1);
                        }
                } else {
                        applfile.error = UINT8_TO_IFS(1);
                }
        }
        ifs_close(&io_fd);
}
```

Note that as described in [Elm04], the configuration data of the service is separated from the I/O data. In this approach, a distributed embedded application will be first described *functionally*, i. e., by a set of interconnected real-time *services*. A service is described by its interfaces, its function, and properties like timing behavior or reliability requirements.

The interfaces of a service are divided into the following categories:

Service Providing Linking Interface (SPLIF): This interface provides the real-time service results to other services (cf. [Jon02]).

Service Requesting Linking Interface (SRLIF): A service that requires real-time input information requests these data via the SRLIF (cf. [Jon02]).

Diagnostic and Management (DM): This interface is used to set parameters and to retrieve information about intermediate and debugging data, e. g., for the purpose of fault diagnosis. Access of the DM interface does not change the (a priori specified) timing behavior of the service.

Configuration and Planning (CP): This interface is used during the integration phase to generate the "glue" between the nearly autonomous services (e. g., communication schedules). The CP interface is not time critical.

Local interfaces: The term local interfaces subsumes all kinds of devices, such as sensors, actuators, displays, and input devices, for which the service creates a unified access via the SPLIF or SRLIF services. For example, the service may instrument a physical sensor element by reading the measurement, calibrating the value, and exporting the measurement via its SPLIF.

Figure 18 depicts the interfaces of a service in a block diagram.

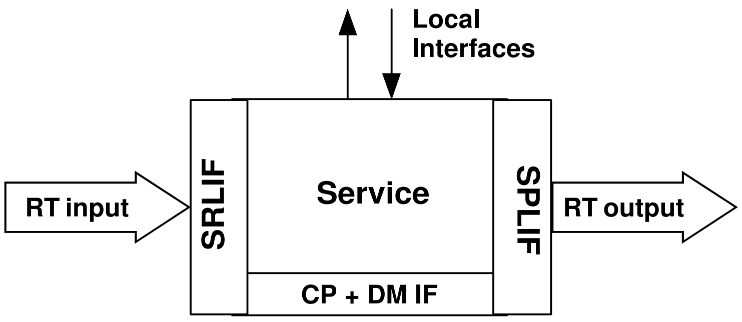


Figure 18: Interfaces of a Service

A particular component may comprise only a subset of the above described interface categories. Typically, a *smart sensor* service will implement a SPLIF, CP, DM, and a local interface to the physical sensor. An *actuator*, in contrast, will implement an SRLIF, CP, DM, and a local interface to the physical actuator.

Data flow over the SPLIF and SRLIF is performed using *ports*. A port is specified by a name, a description, and the structure of the data transmitted over the port (e. g., a 16bit measurement value from a sensor). The port structure consists of the data type of the expected input or, respectively, the produced output. The functional behavior of a service is implemented by a service *task*. The task of a service consumes the data at its SRLIF and produces an output at its SPLIF after termination.

The described port structures can be implemented with one or more I/O files. To interconnect different services on the same node, the application file of the TTP/A task contains the IFS address pointing to the structure in the I/O file (see figure 19).

The usage of only storing the IFS addresses of the real-time data instead of storing the data itself in the application file of the service has the advantage to provide a flexible way of configuration.

3.3.3 Initialization and Background tasks

Example definiton of an initialization task:

```
#include "schedule.h"
```

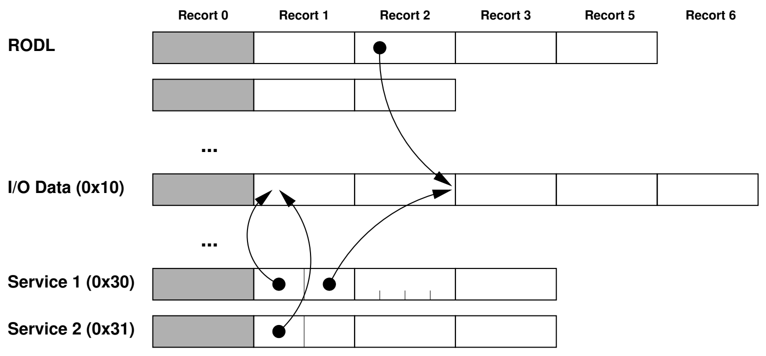


Figure 19: I/O file referenced by Services and RODL entry

```
int my_inittask(void)
{
        /* do initialization */

        return 0; /* do always return 0 */
}

ADD_INITTASK(my_inittask_h, my_inittask, 3, (1<<TTPA_STATE_UNSYNC));
```

The syntax for adding a initialization task is `ADD_INITTASK(`*taskhandle*`, `*task*`, `*order*`, `*states*` )`. The *taskhandle* parameter gives the name of a `sched_task_t` structure (located in Flash memory) where the given parameters are stored. The *task* parameter is a functionpointer to the given initialization task. *Order* gives the order in which initialization tasks are executed. The order is given as a decimal number ranging from `0` to `15`. `0` and `1` would be typically used to initialize/enable external SRAM. `2` is used to initialize UART and Transceiver. `4` is used by the protocol code to copy the containts of the EEPROM and Flash to SRAM. `8` is used to set up Timer for TTP/A.

States is a bit-vector of all states for which the initialization function is called (typically `(1<<TTPA_STATE_UNSYNC)`).

Note that an initialization task must always return `0`, otherwise it is called again.

Example definiton of a background task:

```
int my_bgtask(void)
{
        /* ... */

        if(ready) {
                return 0;
        } else {
                return 1;
        }
}

ADD_BGTASK(my_bgtask_h, my_bgtask, 1, (1<<TTPA_STATE_ACTIVE));
```

The syntax for adding a background task is similar but not identical to an initialization task: `ADD_BGTASK(`*taskhandle*`,` *task*`,` *priority*`,` *states* `)`. The parameter *priority* gives the priority of the task. The value `0` has the highest priority and is only used for initialization tasks. The maximum priority value is given by `SCHED_MAXPRIOR - 1` (currently `3`).

All other parameters are the same as for initialization tasks.

4 Compiling, Programming & Debugging

The development process for a TTP/A node usually contains the following steps:

1. Selecting the hardware of the node.
2. Configuring the protocol parameters for the employed hardware.
3. Writing the application code (e. g. the functions initialization tasks, the TTP/A tasks, the interrupt handlers and the background tasks).
4. Compiling the project and programming the code onto the microcontroller.
5. Testing and debugging of the application.

The following subsections will give hints on how to accomplish these tasks.

4.1 Configuring the Protocol Parameters

Application specific protocol parameters (e.g., bus speed, node type, transceiver, ...) can be configured in the `Makefile`:

```
# ...

PRJNAME = slave3
TARGET = 3
MCU = atmega8
ADD_NODECONF = -DLN_NODE=0x7f  -DSW_UART_TXUSEOC -DUSE_SCHEDULE
NODECONF = -DNODECONF_MEGA8 -DSW_UART -DSLAVE $(ADD_NODECONF)

# ...
```

Figure 20: Configuration parameters in the Makefile for a TTP/A node

Figure 20 shows how the most important properties of a TTP/A node that have to be fixed at compile time can be configured. By defining the variable `ADD_NODECONF` the following parameters can be set:

-DBAUDRATE=*value*: Set the baudrate to *value*

-DMASTER: configure node as master node

-DSLAVE: configure node as slave node

-DNODECONF_*xxx*: Set the node type

-DLN_NODE=*value*: Set the node's logical name (default: `0xff` – unbaptized)

4.2 Defining the IFS

Since the IFS of a node is static, the user has to a priori define size and layout of the node's IFS. The definition of a file has the following syntax:

```
struct structure name {
        file contents
};
struct structure name IFS_LOC(section) file name;
```

```
IFS_ADDAPPLFILE(number, &variable name, task name, IFS_FILELEN(struct structure name),
section, mode);
```

The parameter *number* gives the file name of the IFS file. IFS files for I/O data start at file number `0x10`, Configuration files for services as described in [Elm04] should be located in the range `0x30` to `0x3d`. The other parameters are the *stucture name* for the C `struct` that defines the stucture of the IFS file, the *variable name* of the file, the memory section in which the file is located (see 6)) and the access mode (*read – write – execute*) for multi-partner and master-slave access (as octal number).

Initialized from	Storage location			
	Internal SRAM	External SRAM	EEPROM	Flash
not initialized	.noinit	.ifs_ext	–	–
0 initialized	.bss	.ifs_ext_0	–	–
EEPROM	.ifs_int_eep	–	.eeprom	–
Flash	.data	.ifs_ext_flash	–	.ifs_flash

Table 6: Storage locations

On Atmel AVR the following memory sections are available for the IFS:

ifs_int: Internal SRAM memory, not initialized. The contents of this memory section are not cleared on reset.

ifs_int_0: Internal SRAM memory, initialized with 0 on startup.

ifs_int_eep: The data of the file is stored in the EEPROM of the microcontroller. At startup, the file contents are copied to the RAM. During runtime all IFS operations affect only the RAM part.

ifs_int_flash: Internal SRAM memory. The data of these files is stored in Flash ROM and transfered to internal SRAM on startup.

ifs_ext: External SRAM memory, not initialized.

ifs_ext_0: External SRAM memory, initialized with 0 on startup.

ifs_ext_eep: External SRAM memory, initialized from EEPROM (like ifs_int_eep)

ifs_ext_flash: External SRAM memory, initialized from Flash (like ifs_int_flash)

ifs_eep: EEPROM memory, read-only during runtime.

ifs_flash: The file will be mapped into the flash ROM of the microcontroller and will be read-only during runtime.

```
/* define RODL file for round 0 */
IFS_RODLFILE(0x00, 3, ifs_int_eep)
{
        /* receive one byte in slot 1 (write to file 0x10, rec. 0x01,
           offs. 0x00 */
        IFS_RE_RECV(0x01, IFS_ADDR_I(0x10,0x01,0x00), 0),
        /* execute task 0x10 in slot */
        IFS_RE_EXEC_EOR(0x02, IFS_ADDR_I(0x10,0x01,0x00)),
        /* send two bytes in slot 0x03 */
        IFS_RE_SEND(0x03, IFS_ADDR_I(0x10,0x03,0x02), 0),
        /* end of round in slot 0x05 */
        IFS_RE_EOR(0x05)
};
```

Figure 21: Example for a predefinition of RODL entries for round 0

A RODL file can be specified by using the macros defined in section 3.2. Each RODL entry contains the operation, an IFS address specifying file, record, and alignment/speedup, the slot position, the message length minus 1, and for multiplexed slots a mux identifier.

4.3 Compiling and Programming

We propose to use a program like `make` or `nmake` to automate the procedure of compiling and programming the node.

Figure 22 lists the lines of a sample makefile that have to be edited in order to make an own project. The makefile contains rules for compiling, linking and programming via the ZeusProg programmer.

The user merely has to configure all the application-specific opject files (in the `OBJS=` line), the extra libraries (if any), a name for his project and the type of microcontroller used. Using this makefiles, a project can easily be fresh compiled by calling the commands `make clean` and `make all`. The code will be downloaded into the node using the command `make install`.

4.4 Using AVR-Insight

Insight is a graphical font-end for the GNU GDB debugger. It can be used to debug Atmel AVR microcontrollers via the built-in remote debugging feature. This document describes how to setup *Insight* and the JTAG interface daemon *Avarice*. *Avarice* requires a serial port to communicate with the *JTAGIce* and runs on Linux, FreeBSD, MacOS X, Solaris, and Windows (with Cygwin installed).

4.4.1 Avarice

The *Avarice* program is used to translate the GDB commands received via a TCP/IP socket into Atmel-specific protocol that is used to communicate with the Atmel *JTAGIce*. The *JTAGIce* is attached to the computer through a serial connection. Beside using *Avarice* for debugging, it can also be used to erase, program, and to verify the memory contents of the target.

Avarice uses the following command line options:

`-h, --help`: Print help message

`-d, --debug`: Print debug information

`-D, --detach`: Detach once synced with JTAG ICE

`-C, --capture`: Capture running program.

`-f, --file <filename>`: Specify a file for use with the `--program` and `--verify` options. If `--file` is passed and neither `--program` nor `--verify` are given then `--program` is implied.

`-j, --jtag <devname>`: Serial port attached to JTAG box (default: `/dev/avrjtag`).

`-B, --jtag-bitrate <rate>`: Set the bitrate that the JTAG box communicates with the AVR target device. This must be less than 1/4 of the frequency of the target. Valid values are 1MHz, 500KHz, 250KHz or 125KHz. (default: 1MHz)

`-p, --program`: Erase and program target. Binary filename must be specified with `--file` option.

`-v, --verify`: Verify program in device against file specified with `--file` option.

`-e, --erase`: Erase target.

`-r, --read-fuses`: Read fuses bytes.

`-W, --write-fuses <eehhll>`: Write fuses bytes; `ee` is the extended fuse byte, `hh` is the high fuse byte and `ll` is the low fuse byte. The fuse byte data must be given in two digit hexadecimal format with zero padding if needed. All three bytes must be given correctly. NOTE: Currently, if the target device does not have an extended fuse byte (e.g. the atmega16), you should set `ee==ll` when writing the fuse bytes.

Care should be taken when setting the fuse bytes, since incorrect settings may render the controller unresponsive (e.g., disabling the JTAG interface).

`-L, --write-lockbits <ll>`: Write lock bits. The lock byte data must be given in two digit hexadecimal format with zero padding if needed.

`-P, --part<name>`: Target device name (e.g. atmega16)

To debug an Atmel AVR target with GDB, the Avarice program has to be started first:

```
avarice --program --file foo.elf  localhost:6423
```

This command can be easily integrated into the Makefile:

```
debug:  $(PRJNAME).elf
        avarice --program --file $< :6423
```

Note that the current implementation of Avarice does not support the GDB *load* command. Therefore, it is necessary to restart the Avarice program each time the `.elf` file is changed.

4.4.2 Debugging the Target

As already described, Insight is a graphical front-end for GNU GDB. It can built to use Atmel AVR as target and uses the remote debugging feature of GDB to communicate with Avarice.

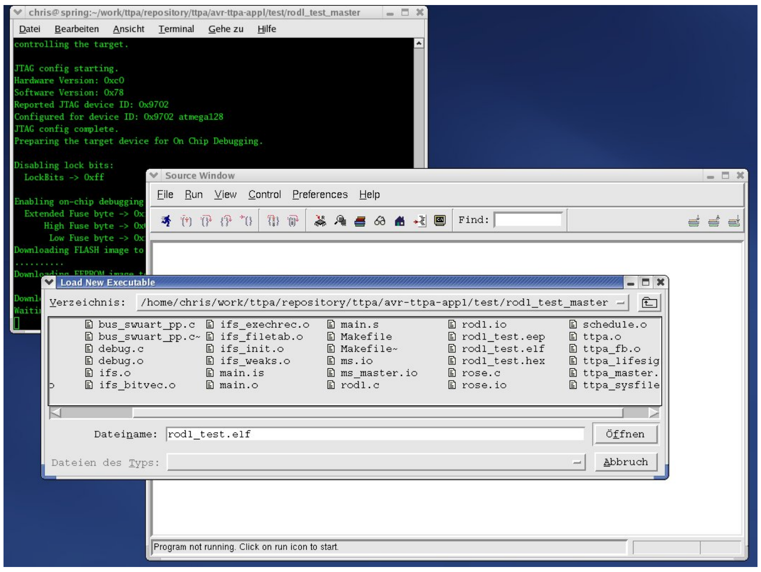
Connecting to the Target First of all, the JTAG daemon has to be started:

```
> avarice --program --file rodl_test.elf  localhost:6423
```

or:

```
> make debug
```

After that, `avr-insight` has to be started. Select *File→Open* and choose the `.elf` file to debug:


chris@spring:~/work/ttpa/repository/ttpa/avr-ttpa-appl/test/rodl_test_master
Datei Bearbeiten Ansicht Terminal Gehe zu Hilfe
Source Window
File Run View Control Preferences Help
Find:
Load New Executable
Öffnen
Abbruch

```
controlling the target.

JTAG config starting.
Hardware Version: 0xc0
Software Version: 0x78
Reported JTAG device ID: 0x9702
Configured for device ID: 0x9702 atmega128
JTAG config complete.
Preparing the target device for On Chip Debugging.

Disabling lock bits:
  LockBits -> 0xff

Enabling on-chip debugging
  Extended Fuse byte -> 0x
      High Fuse byte -> 0x
       Low Fuse byte -> 0x
Downloading FLASH image to
..........
Downloading EEPROM image t

Downl
Waiti
```

Verzeichnis: /home/chris/work/ttpa/repository/ttpa/avr-ttpa-appl/test/rodl_test_master

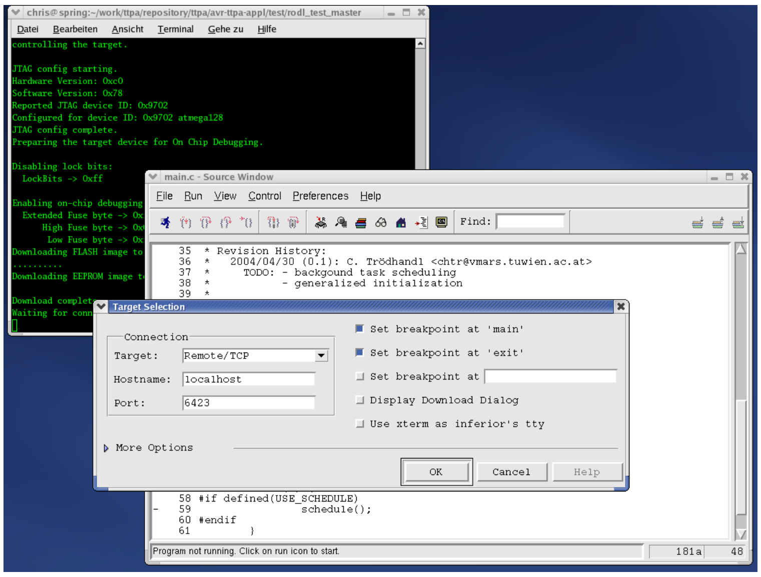
```
bus_swuart_pp.c   ifs_exechrec.o   main.s         rodl.io          schedule.o
bus_swuart_pp.c~  ifs_filetab.o    Makefile       rodl_test.eep    ttpa.o
debug.c           ifs_init.o       Makefile~      rodl_test.elf    ttpa_fb.o
debug.o           ifs_weaks.o      ms.io          rodl_test.hex    ttpa_lifesig
ifs.o             main.is          ms_master.io   rose.c           ttpa_master.
ifs_bitvec.o      main.o           rodl.c         rose.io          ttpa_sysfile
```

Dateiname: rodl_test.elf

Dateien des Typs:

Program not running. Click on run icon to start.

Before a connection to the target can be established, the target settings have to be configured. Select *Target Settings...* from the *File* menu, set *Target* to Remote/TCP, configure *hostname* and *Port* (e.g. localhost, 6423):


chris@spring:~/work/ttpa/repository/ttpa/avr-ttpa-appl/test/rodl_test_master
Datei Bearbeiten Ansicht Terminal Gehe zu Hilfe
main.c - Source Window
File Run View Control Preferences Help
Find:
Target Selection
More Options
OK
Cancel
Help

```
controlling the target.

JTAG config starting.
Hardware Version: 0xc0
Software Version: 0x78
Reported JTAG device ID: 0x9702
Configured for device ID: 0x9702 atmega128
JTAG config complete.
Preparing the target device for On Chip Debugging.

Disabling lock bits:
  LockBits -> 0xff

Enabling on-chip debugging
  Extended Fuse byte -> 0x
      High Fuse byte -> 0x
       Low Fuse byte -> 0x
Downloading FLASH image to
..........
Downloading EEPROM image t

Download complet
Waiting for conn
```

```
 * Revision History:
 *   2004/04/30 (0.1): C. Trödhandl <chtr@vmars.tuwien.ac.at>
 *     TODO: - backgound task scheduling
 *           - generalized initialization
 *
```

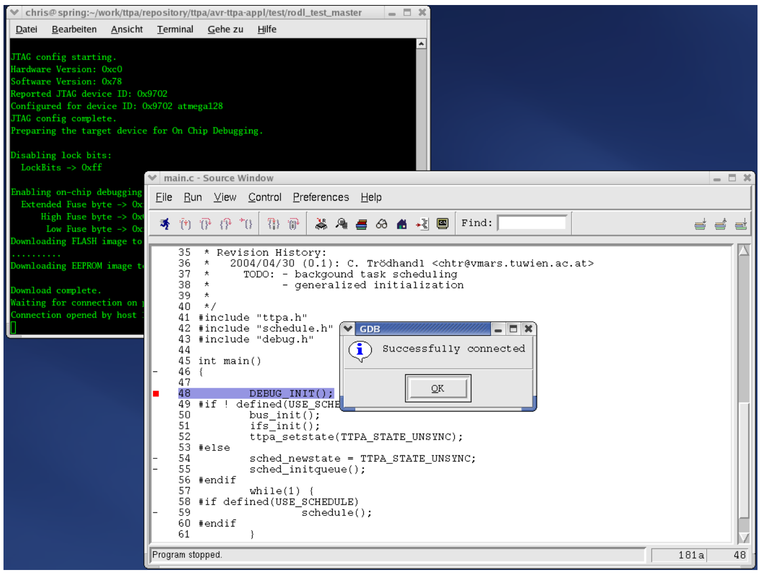
Connection
Target: Remote/TCP
Hostname: localhost
Port: 6423

[✓] Set breakpoint at 'main'
[✓] Set breakpoint at 'exit'
☐ Set breakpoint at
☐ Display Download Dialog
☐ Use xterm as inferior's tty

```
#if defined(USE_SCHEDULE)
                schedule();
#endif
        }
```

Program not running. Click on run icon to start. 181a 48

After selecting *Run→Connect to Target* the remote debugging connection is active.


chris@spring:~/work/ttpa/repository/ttpa/avr-ttpa-appl/test/rodl_test_master
Datei Bearbeiten Ansicht Terminal Gehe zu Hilfe
main.c - Source Window
File Run View Control Preferences Help
Find:
GDB
OK

```
JTAG config starting.
Hardware Version: 0xc0
Software Version: 0x78
Reported JTAG device ID: 0x9702
Configured for device ID: 0x9702 atmega128
JTAG config complete.
Preparing the target device for On Chip Debugging.

Disabling lock bits:
  LockBits -> 0xff

Enabling on-chip debugging
  Extended Fuse byte -> 0x
      High Fuse byte -> 0x
       Low Fuse byte -> 0x
Downloading FLASH image to
..........
Downloading EEPROM image t

Download complete.
Waiting for connection on
Connection opened by host
```

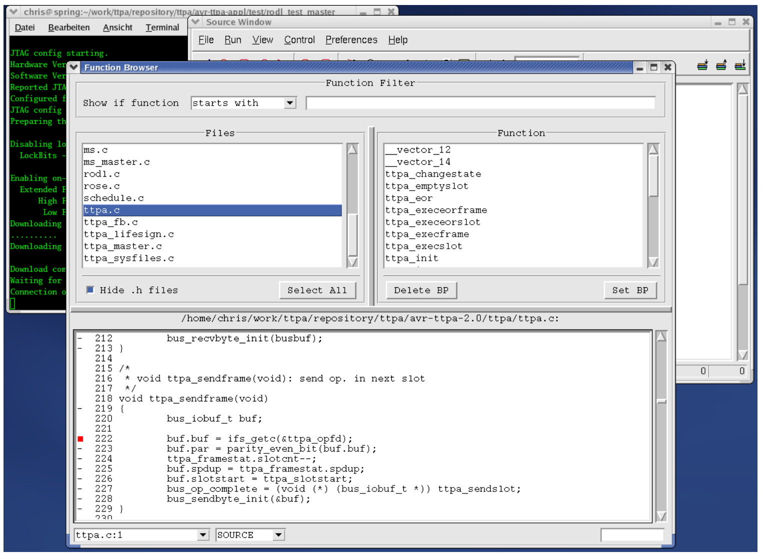
```
 * Revision History:
 *   2004/04/30 (0.1): C. Trödhandl <chtr@vmars.tuwien.ac.at>
 *     TODO: - backgound task scheduling
 *           - generalized initialization
 *
 */
#include "ttpa.h"
#include "schedule.h"
#include "debug.h"

int main()
{

        DEBUG_INIT();
#if ! defined(USE_SCHE
        bus_init();
        ifs_init();
        ttpa_setstate(TTPA_STATE_UNSYNC);
#else
        sched_newstate = TTPA_STATE_UNSYNC;
        sched_initqueue();
#endif
        while(1) {
#if defined(USE_SCHEDULE)
                schedule();
#endif
        }
```

Successfully connected

Program stopped. 181a 48

Starting and Debugging the Program After the connection to the target is established, the target CPU is stopped at the first instruction in the program memory (reset vector). By selecting *Continue* from the *Control* menu or pressing 'C', the execution of the program will continue to the first breakpoint (default: first instruction in `main()`).

Breakpoints can be set in the *Function Browser* window (select *Function Browser* from the *View* menu). In this window, functions can be selected from different source files and breakpoints can be set by clicking on the begin of a source line (breakpoints are shown as a red square).


chris@spring:~/work/ttpa/repository/ttpa/avr-ttpa-appl/test/rodl_test_master
Datei Bearbeiten Ansicht Terminal
Source Window
File Run View Control Preferences Help
Function Browser
Function Filter
Files
Function
Select All
Delete BP
Set BP

Show if function starts with

Files	Function
ms.c	__vector_12
ms_master.c	__vector_14
rodl.c	ttpa_changestate
rose.c	ttpa_emptyslot
schedule.c	ttpa_eor
ttpa.c	ttpa_execeorframe
ttpa_fb.c	ttpa_execeorslot
ttpa_lifesign.c	ttpa_execframe
ttpa_master.c	ttpa_execslot
ttpa_sysfiles.c	ttpa_init

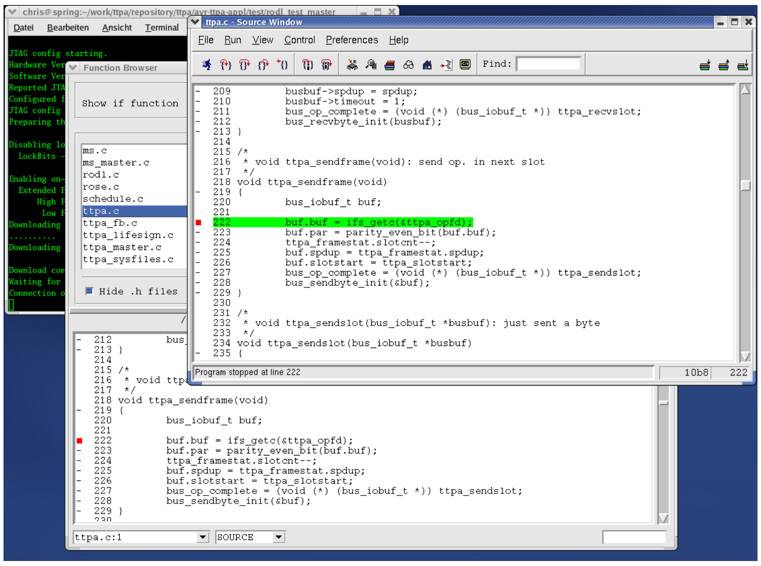
Hide .h files

/home/chris/work/ttpa/repository/ttpa/avr-ttpa-2.0/ttpa/ttpa.c:

```
        bus_recvbyte_init(busbuf);
}

/*
 * void ttpa_sendframe(void): send op. in next slot
 */
void ttpa_sendframe(void)
{
        bus_iobuf_t buf;

        buf.buf = ifs_getc(&ttpa_opfd);
        buf.par = parity_even_bit(buf.buf);
        ttpa_framestat.slotcnt--;
        buf.spdup = ttpa_framestat.spdup;
        buf.slotstart = ttpa_slotstart;
        bus_op_complete = (void (*) (bus_iobuf_t *)) ttpa_sendslot;
        bus_sendbyte_init(&buf);
}
```

ttpa.c:1 SOURCE

If the execution of the program stops at a breakpoint, the corresponding source line is highlighted.

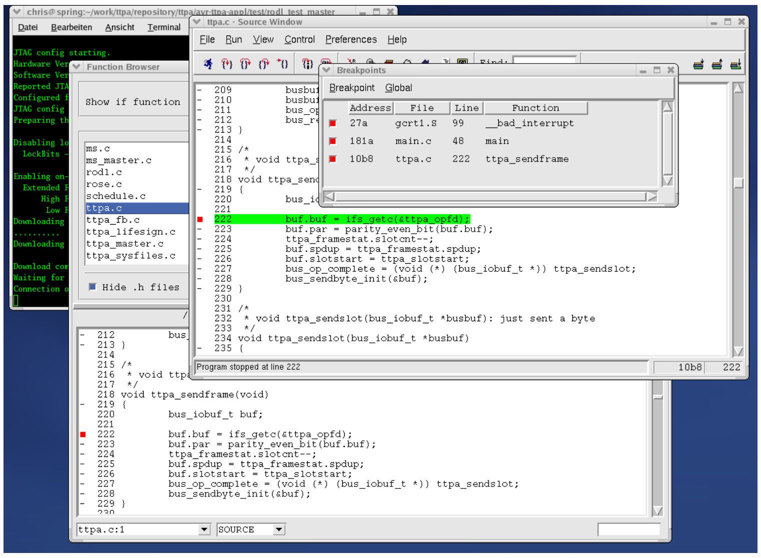


ttpa.c - Source Window

```
- 209         busbuf->spdup = spdup;
- 210         busbuf->timeout = 1;
- 211         bus_op_complete = (void (*) (bus_iobuf_t *)) ttpa_recvslot;
- 212         bus_recvbyte_init(busbuf);
- 213 }
  214
  215 /*
  216  * void ttpa_sendframe(void): send op. in next slot
  217  */
  218 void ttpa_sendframe(void)
- 219 {
  220         bus_iobuf_t buf;
  221
■ 222         buf.buf = ifs_getc(&ttpa_opfd);
- 223         buf.par = parity_even_bit(buf.buf);
- 224         ttpa_framestat.slotcnt--;
- 225         buf.spdup = ttpa_framestat.spdup;
- 226         buf.slotstart = ttpa_slotstart;
- 227         bus_op_complete = (void (*) (bus_iobuf_t *)) ttpa_sendslot;
- 228         bus_sendbyte_init(&buf);
- 229 }
  230
  231 /*
  232  * void ttpa_sendslot(bus_iobuf_t *busbuf): just sent a byte
  233  */
  234 void ttpa_sendslot(bus_iobuf_t *busbuf)
- 235 {
```

```
Program stopped at line 222                         10b8     222
```

The *Breakpoints* window (select *View*→*Breakpoints*) shows all defined breakpoints. Note that at a given time no more than four breakpoints should be active because of limitations of the JTAGIce. In the *Breakpoints* window, breakpoints can be activated, deactivated, set temporary or can be deleted.

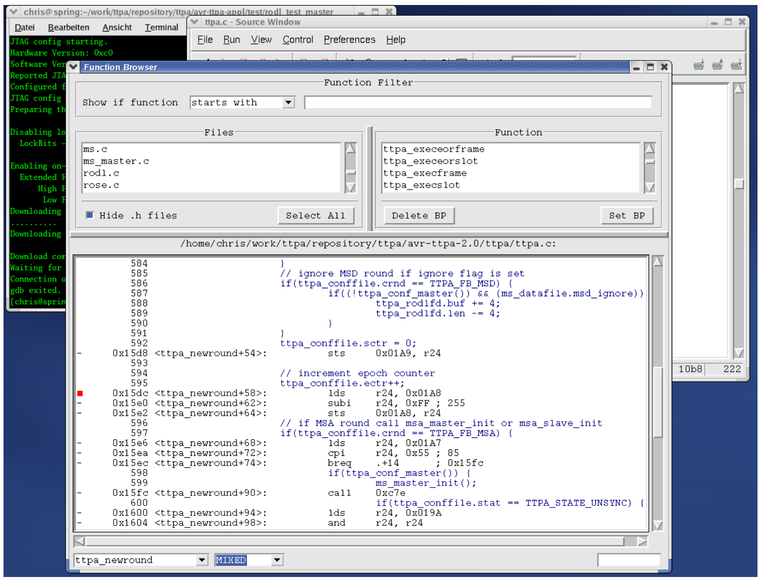


Breakpoints

	Address	File	Line	Function
🟥	27a	gcrt1.S	99	__bad_interrupt
🟥	181a	main.c	48	main
🟥	10b8	ttpa.c	222	ttpa_sendframe

For continuing the execution of the program either select *Step* (step into the next function), *Next* (step over the next function), *Finish* (continue to the end of the current function), and *Continue* (continue to the next breakpoint).

The program code can be viewed as source, assembler code, or in mixed mode:



```
ms.c
ms_master.c
rodl.c
rose.c
```

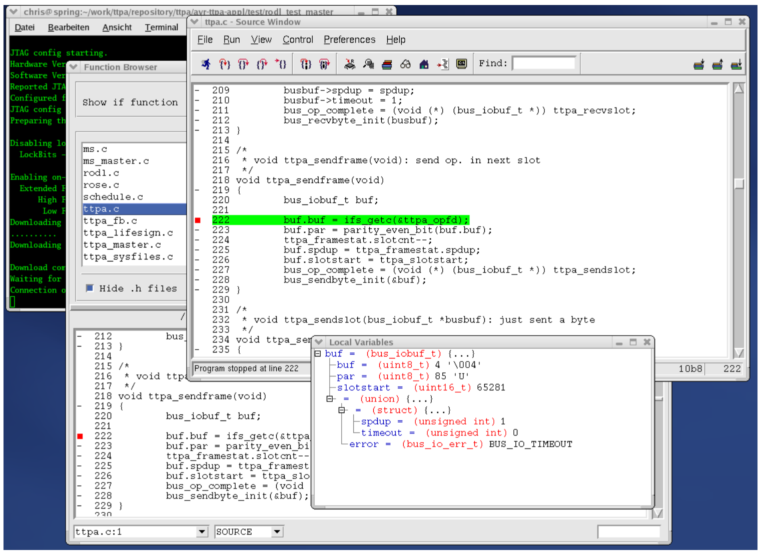
```
ttpa_execeorframe
ttpa_execeorslot
ttpa_execframe
ttpa_execslot
```

/home/chris/work/ttpa/repository/ttpa/avr-ttpa-2.0/ttpa/ttpa.c:

```
     584                                 }
     585                                 // ignore MSD round if ignore flag is set
     586                                 if(ttpa_conffile.crnd == TTPA_FB_MSD) {
     587                                         if((!ttpa_conf_master()) && (ms_datafile.msd_ignore))
     588                                                 ttpa_rodlfd.buf += 4;
     589                                                 ttpa_rodlfd.len -= 4;
     590                                         }
     591                                 }
     592                                 ttpa_conffile.sctr = 0;
-    0x15d8 <ttpa_newround+54>:              sts     0x01A9, r24
     593
     594                                 // increment epoch counter
     595                                 ttpa_conffile.ectr++;
■    0x15dc <ttpa_newround+58>:              lds     r24, 0x01A8
-    0x15e0 <ttpa_newround+62>:              subi    r24, 0xFF ; 255
-    0x15e2 <ttpa_newround+64>:              sts     0x01A8, r24
     596                                 // if MSA round call msa_master_init or msa_slave_init
     597                                 if(ttpa_conffile.crnd == TTPA_FB_MSA) {
-    0x15e6 <ttpa_newround+68>:              lds     r24, 0x01A7
-    0x15ea <ttpa_newround+72>:              cpi     r24, 0x55 ; 85
-    0x15ec <ttpa_newround+74>:              breq    .+14      ; 0x15fc
     598                                         if(ttpa_conf_master()) {
     599                                                 ms_master_init();
-    0x15fc <ttpa_newround+90>:              call    0xc7e
     600                                                 if(ttpa_conffile.stat == TTPA_STATE_UNSYNC) {
-    0x1600 <ttpa_newround+94>:              lds     r24, 0x019A
-    0x1604 <ttpa_newround+98>:              and     r24, r24
```

ttpa_newround MIXED

Variables and Memory Contents The contents of all local variables are available in the *Local Variables* window (select *View*→*Local Variables*). This window also supports expansion of `struct` and `union` members.



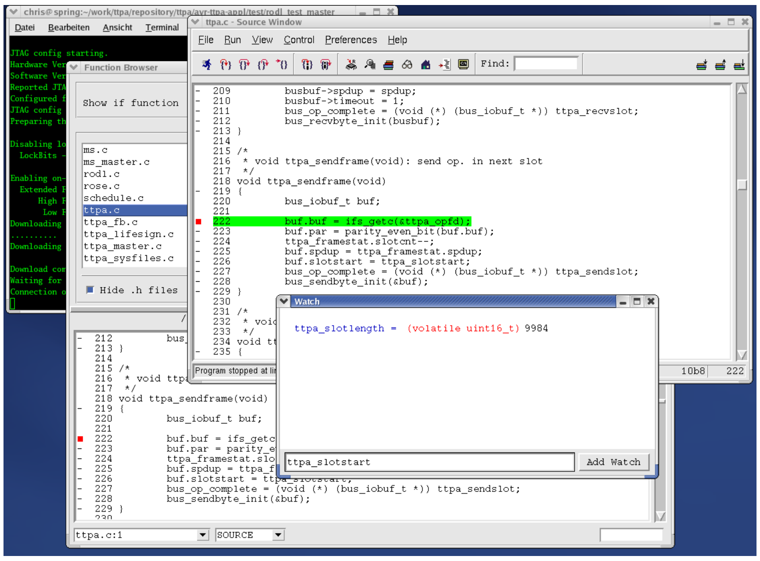
```
- 209         busbuf->spdup = spdup;
- 210         busbuf->timeout = 1;
- 211         bus_op_complete = (void (*) (bus_iobuf_t *)) ttpa_recvslot;
- 212         bus_recvbyte_init(busbuf);
- 213 }
  214
  215 /*
  216  * void ttpa_sendframe(void): send op. in next slot
  217  */
  218 void ttpa_sendframe(void)
- 219 {
  220         bus_iobuf_t buf;
  221
■ 222         buf.buf = ifs_getc(&ttpa_opfd);
- 223         buf.par = parity_even_bit(buf.buf);
- 224         ttpa_framestat.slotcnt--;
- 225         buf.spdup = ttpa_framestat.spdup;
- 226         buf.slotstart = ttpa_slotstart;
- 227         bus_op_complete = (void (*) (bus_iobuf_t *)) ttpa_sendslot;
- 228         bus_sendbyte_init(&buf);
- 229 }
  230
  231 /*
  232  * void ttpa_sendslot(bus_iobuf_t *busbuf): just sent a byte
  233  */
  234 void ttpa_sen
- 235 {
```

Program stopped at line 222   10b8   222

```
⊟ buf =  (bus_iobuf_t) {...}
  ├buf =  (uint8_t) 4 '\004'
  ├par =  (uint8_t) 85 'U'
  ├slotstart =  (uint16_t) 65281
  ⊟ =  (union) {...}
    ⊟ =  (struct) {...}
    │ ├spdup =  (unsigned int) 1
    │ └timeout =  (unsigned int) 0
    └error =  (bus_io_err_t) BUS_IO_TIMEOUT
```

ttpa.c:1 SOURCE

For displaying global variables, expressions can be defined with the *Watch* window (*View*→*Watch Expressions*).


chris@spring:~/work/ttpa/repository/ttpa/avr-ttpa-appl/test/rodl_test_master
Datei Bearbeiten Ansicht Terminal
Function Browser
Show if function
Hide .h files
ttpa.c - Source Window
File Run View Control Preferences Help
Find:
Watch
Add Watch

```
ms.c
ms_master.c
rodl.c
rose.c
schedule.c
ttpa.c
ttpa_fb.c
ttpa_lifesign.c
ttpa_master.c
ttpa_sysfiles.c
```

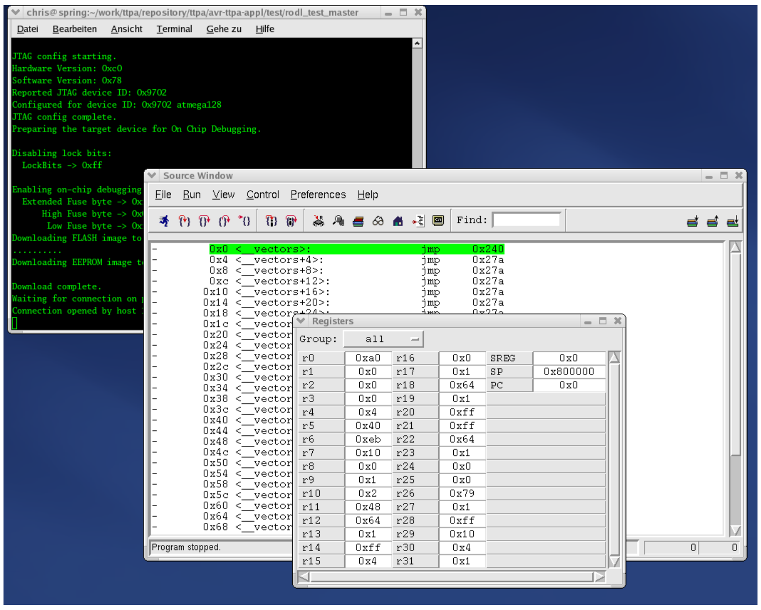
```
        busbuf->spdup = spdup;
        busbuf->timeout = 1;
        bus_op_complete = (void (*) (bus_iobuf_t *)) ttpa_recvslot;
        bus_recvbyte_init(busbuf);
}

/*
 * void ttpa_sendframe(void): send op. in next slot
 */
void ttpa_sendframe(void)
{
        bus_iobuf_t buf;

        buf.buf = ifs_getc(&ttpa_opfd);
        buf.par = parity_even_bit(buf.buf);
        ttpa_framestat.slotcnt--;
        buf.spdup = ttpa_framestat.spdup;
        buf.slotstart = ttpa_slotstart;
        bus_op_complete = (void (*) (bus_iobuf_t *)) ttpa_sendslot;
        bus_sendbyte_init(&buf);
}
```

```
ttpa_slotlength =  (volatile uint16_t) 9984
```

```
ttpa_slotstart
```

```
Program stopped at li        10b8    222
ttpa.c:1        SOURCE
```

The contents of the processor registers are displayed with the *Registers* window (*View*→*Registers*).


chris@spring:~/work/ttpa/repository/ttpa/avr-ttpa-appl/test/rodl_test_master
Datei Bearbeiten Ansicht Terminal Gehe zu Hilfe
Source Window
File Run View Control Preferences Help
Find:
Registers
Program stopped.

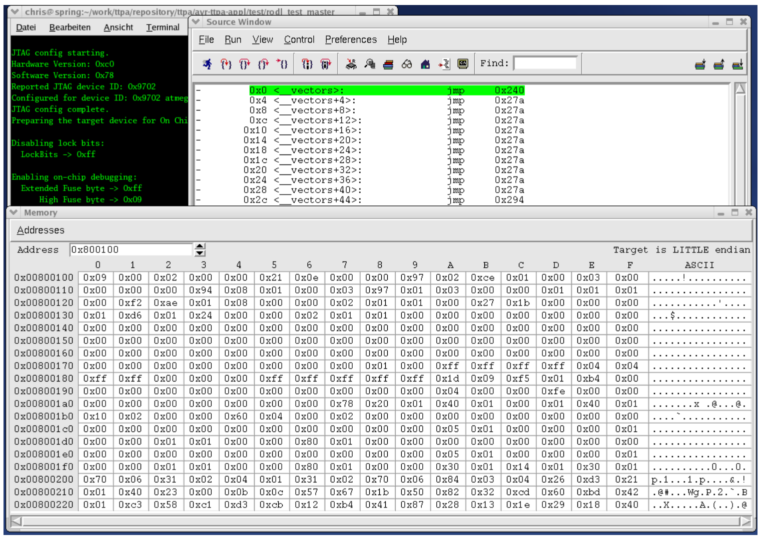
```
JTAG config starting.
Hardware Version: 0xc0
Software Version: 0x78
Reported JTAG device ID: 0x9702
Configured for device ID: 0x9702 atmega128
JTAG config complete.
Preparing the target device for On Chip Debugging.

Disabling lock bits:
  LockBits -> 0xff

Enabling on-chip debugging
  Extended Fuse byte -> 0x
      High Fuse byte -> 0x
       Low Fuse byte -> 0x
Downloading FLASH image to
..........
Downloading EEPROM image t

Download complete.
Waiting for connection on
Connection opened by host
```

```
 0x0 <__vectors>:        jmp   0x240
 0x4 <__vectors+4>:      jmp   0x27a
 0x8 <__vectors+8>:      jmp   0x27a
 0xc <__vectors+12>:     jmp   0x27a
0x10 <__vectors+16>:     jmp   0x27a
0x14 <__vectors+20>:     jmp   0x27a
0x18 <__vectors+24>:
0x1c <__vector
0x20 <__vector
0x24 <__vector
0x28 <__vector
0x2c <__vector
0x30 <__vector
0x34 <__vector
0x38 <__vector
0x3c <__vector
0x40 <__vector
0x44 <__vector
0x48 <__vector
0x4c <__vector
0x50 <__vector
0x54 <__vector
0x58 <__vector
0x5c <__vector
0x60 <__vector
0x64 <__vector
0x68 <__vector
```

Group: all

r0	0xa0	r16	0x0	SREG	0x0
r1	0x0	r17	0x1	SP	0x800000
r2	0x0	r18	0x64	PC	0x0
r3	0x0	r19	0x1		
r4	0x4	r20	0xff		
r5	0x40	r21	0xff		
r6	0xeb	r22	0x64		
r7	0x10	r23	0x1		
r8	0x0	r24	0x0		
r9	0x1	r25	0x0		
r10	0x2	r26	0x79		
r11	0x48	r27	0x1		
r12	0x64	r28	0xff		
r13	0x1	r29	0x10		
r14	0xff	r30	0x4		
r15	0x4	r31	0x1		

Memory contents are displayed displayed with the *Memory* window (*View*→*Memory*). Note that the GNU GCC toolchain uses a unified memory approach and therefore the debugger displays Flash and SRAM as a single, big block. Therefore, to evaluate a given SRAM location, an offset of `0x800000` has to be added to the SRAM address.


chris@spring:~/work/ttpa/repository/ttpa/avr-ttpa-appl/test/rodl_test_master
Datei Bearbeiten Ansicht Terminal
Source Window
File Run View Control Preferences Help
Find:
Memory
Addresses

```
JTAG config starting.
Hardware Version: 0xc0
Software Version: 0x78
Reported JTAG device ID: 0x9702
Configured for device ID: 0x9702 atmeg
JTAG config complete.
Preparing the target device for On Chi

Disabling lock bits:
  LockBits -> 0xff

Enabling on-chip debugging:
  Extended Fuse byte -> 0xff
      High Fuse byte -> 0x09
```

```
-        0x0 <__vectors>:          jmp     0x240
-        0x4 <__vectors+4>:        jmp     0x27a
-        0x8 <__vectors+8>:        jmp     0x27a
-        0xc <__vectors+12>:       jmp     0x27a
-       0x10 <__vectors+16>:       jmp     0x27a
-       0x14 <__vectors+20>:       jmp     0x27a
-       0x18 <__vectors+24>:       jmp     0x27a
-       0x1c <__vectors+28>:       jmp     0x27a
-       0x20 <__vectors+32>:       jmp     0x27a
-       0x24 <__vectors+36>:       jmp     0x27a
-       0x28 <__vectors+40>:       jmp     0x27a
-       0x2c <__vectors+44>:       jmp     0x294
```

```
Address  0x800100                                                                      Target is LITTLE endian
              0    1    2    3    4    5    6    7    8    9    A    B    C    D    E    F        ASCII
0x00800100 0x09 0x00 0x02 0x00 0x00 0x21 0x0e 0x00 0x00 0x97 0x02 0xce 0x01 0x00 0x03 0x00 .....!..........
0x00800110 0x00 0x00 0x00 0x94 0x08 0x01 0x00 0x03 0x97 0x01 0x03 0x00 0x00 0x01 0x01 0x01 ................
0x00800120 0x00 0xf2 0xae 0x01 0x08 0x00 0x00 0x02 0x01 0x01 0x00 0x27 0x1b 0x00 0x00 0x00 ...........'....
0x00800130 0x01 0xd6 0x01 0x24 0x00 0x00 0x02 0x01 0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00 ...$............
0x00800140 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 ................
0x00800150 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 ................
0x00800160 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 ................
0x00800170 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x01 0x00 0xff 0xff 0xff 0xff 0x04 0x04 ................
0x00800180 0xff 0xff 0x00 0x00 0x00 0xff 0xff 0xff 0xff 0xff 0x1d 0x09 0xf5 0x01 0xb4 0x00 ................
0x00800190 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x04 0x00 0x00 0xfe 0x00 0x00 ................
0x008001a0 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x78 0x20 0x01 0x40 0x01 0x00 0x01 0x40 0x01 .......x .@...@.
0x008001b0 0x10 0x02 0x00 0x00 0x60 0x04 0x00 0x02 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 ....`...........
0x008001c0 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x05 0x01 0x00 0x00 0x00 0x01 ................
0x008001d0 0x00 0x00 0x01 0x01 0x00 0x00 0x80 0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 ................
0x008001e0 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x05 0x01 0x00 0x00 0x00 0x01 ................
0x008001f0 0x00 0x00 0x01 0x01 0x00 0x00 0x80 0x01 0x00 0x00 0x30 0x01 0x14 0x01 0x30 0x01 ..........0...0.
0x00800200 0x70 0x06 0x31 0x02 0x04 0x01 0x31 0x02 0x70 0x06 0x84 0x03 0x04 0x26 0xd3 0x21 p.1...1.p....&.!
0x00800210 0x01 0x40 0x23 0x00 0x0b 0x0c 0x57 0x67 0x1b 0x50 0x82 0x32 0xcd 0x60 0xbd 0x42 .@#...Wg.P.2.`.B
0x00800220 0x01 0xc3 0x58 0xc1 0xd3 0xcb 0x12 0xb4 0x41 0x87 0x28 0x13 0x1e 0x29 0x18 0x40 ..X.....A.(..).@
```

4.5 Debugging and Monitoring TTP/A nodes

It is difficult to debug programs for embedded systems. Usually there is no keyboard or screen available for debugging. Due to the limited code size even a full-featured debugging software is hard to employ.

TTP/A nodes provide a very simple monitoring interface that supports access the local IFS of a particular node. Therefore a small monitoring routine is inserted that communicates via the hardware UART of the node (since the TTP/A-protocol uses a software UART this resource should be available) with a monitoring tool.

Thus, a PC or notebook can run a debugging tool with a graphical user interface for a number of nodes (limited mainly by the number of serial ports).

In order to enable debugging in a TTP/A-Node it is necessary to insert two routines into the file `main.c` of the node's project (see Figure 23). `init_monitoring()` initializes the resources used by the monitoring-task and `ttpa_monitoring_task()` is a task running alternately with the background-task of the node.

For using the monitoring interface, the node has to be connected via an RS232 cable to the serial port of a PC or notebook running the monitoring client. After starting the monitoring client (see Figure 24 it is necessary to adjust the settings for port and baud rate. The monitoring client tries then to enable a connection to the node (see status-line for diagnostic messages). If the connection cannot be established use the following check-list:

- Is the node connected to a running TTP/A node? The node does not execute the monitoring task if it cannot synchronize to a TTP/A master.
- Are the communication parameters in the monitoring program (baud rate and port number)

Debug signal	Pin
Timer match event	0
Timer capture event	1
Lost synchronization	2
RODL lookup	7

Table 7: Available debugging signals when using the DEBUG option for compiling.

set correctly?

- Are the communication parameters in the node (clock speed, baud-rate) set correctly?
- Are the `init_monitoring()` and `ttpa_monitoring_task();` included in the `main.c` file?

After establishing the connection the ID and series-/serial-number is figured out and the contents of the IFS of the connected node are displayed in an array of hexadecimal numbers. For greater convenience the RODL files (file 0x00-0x07) are displayed in the decoded way too. As long as the node is connected the monitoring client periodically polls for the contents of the node's IFS.

It is possible to change the view by selecting each single file of the node's IFS to be displayed or not. Since only the visible IFS parts are updated, selecting a subset of the node's IFS may speed up the update frequency.

It is also possible to change values in the node's IFS by selecting the desired record and writing new values into the edit-field. After pressing the button "Write Rec." the record is written into the node's IFS.

Be aware that changing vital values in the node's system files can cause the node to crash! In that case a power-off/on reset of the cluster is necessary.

4.6 Troubleshooting

In case a TTP/A node refuses to operate properly on the bus, some special code parts for troubleshooting should be used to debug this situation.

Therefore, the option `-DDEBUG_PORT=`*port* in `ADD_NODECONF=` line of the the Makefile (see Figure 20) must be set before compiling. A node that is compiled with the debug option, offers the following debugging signals on the I/O pins of the specified port as denoted in table 7.

The signals are only informative if evaluated with an oscilloscope together with the usual bus communication.

Timer match event: Helps in verifying the timing behavior of the software UART. Every time a bit is written to the bus or a bit is sampled, the timer match is signaled as a short impulse to low. All other times the timer match event line is held on high.

Timer capture event: Marks the expected start bit of a byte that is received. The timer capture is signaled as a short impulse to low. All other times the timer capture event line is held on high.

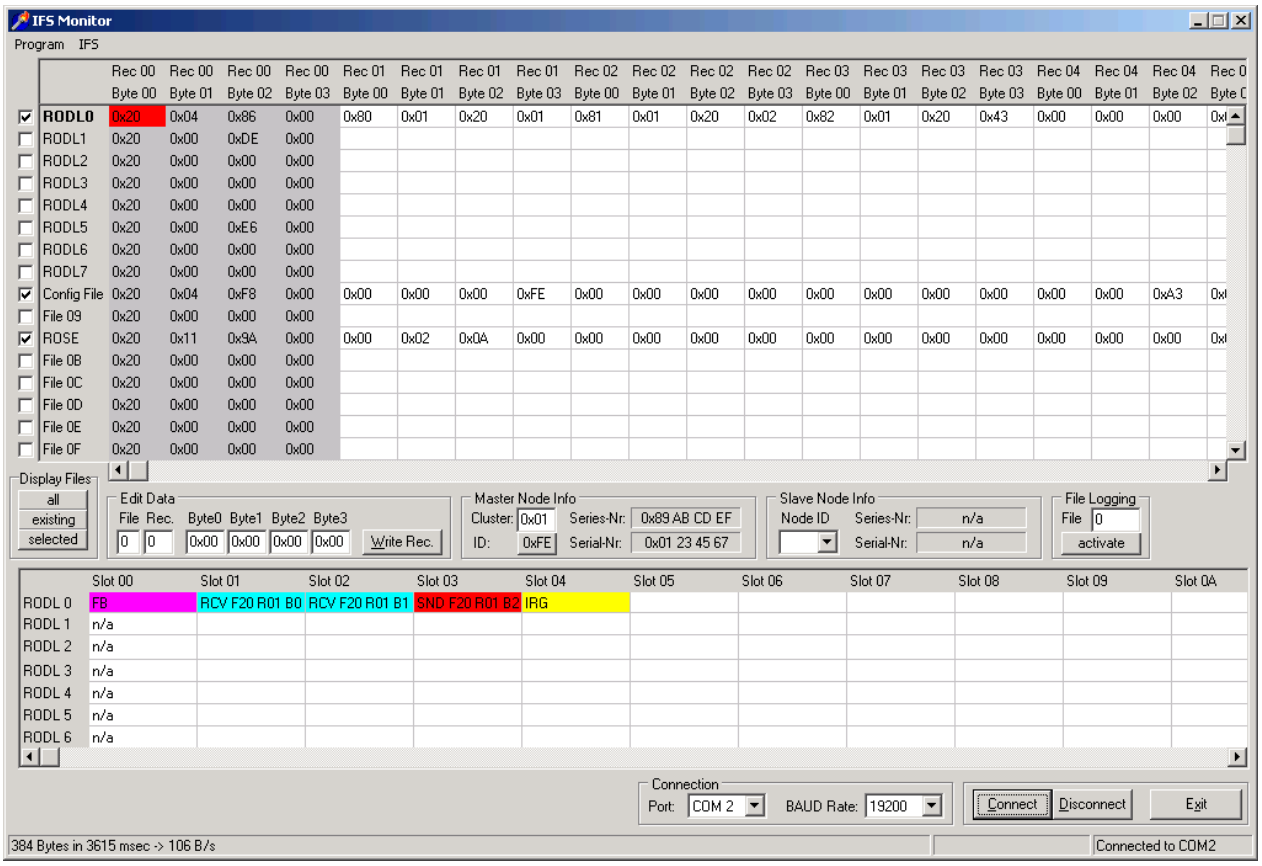
Lost synchronization: This signal is especially useful for detecting RODL misconfiguration. In this case a node will listen for a fireworks byte at the wrong time and looses synchronization. This event is signaled as a short impulse to low. All other times the lost synchronization signal is held on high. Usually the node will regain synchronization at the next real fireworks byte so that a synchronization loss does not necessarily result in a total communication failure of the node.

RODL lookup: Shows the duration, while the protocol program is accessing the filesystem or interpreting the RODL. The RODL lookup is signaled as a short impulse to low. All other times the RODL lookup event line is held on high.

```
# Makefile for slave 3

PRJNAME = slave3
TARGET = 3
MCU = atmega8
ADD_NODECONF = -DLN_NODE=0x7f  -DSW_UART_TXUSEOC -DUSE_SCHEDULE
NODECONF = -DNODECONF_MEGA8 -DSW_UART -DSLAVE $(ADD_NODECONF)


AS      = avr-as
LD      = avr-ld
CC      = avr-gcc
OBJCOPY= avr-objcopy
ZEUSPROG = ZeusProg
UISP    = uisp
PROGRAMMER= apa
TTPAROOT = ../../avr-ttpa-2.0-0.3
MONITORINGROOT = ../../monitoring
SCRIPTROOT = $(TTPAROOT)/scripts

IFSCONV = $(SCRIPTROOT)/conv_byteorder.pl


# add Include dirs here
IDIR    = -I./ -I$(TTPAROOT)/include/
LDDIR   = $(TTPAROOT)/ldscripts/

# select output format for *.hex and *.eep files
# srec: Motorola srec; ihex: Intel ihex
FORMAT = ihex
CFLAGS =  -ggdb -O2 $(IDIR) -mmcu=$(MCU) $(NODECONF)
ASFLAGS=  -ggdb $(IDIR) -mmcu=$(MCU) -Wa,-mmcu=$(MCU) $(NODECONF)
LDFLAGS= -Xlinker -T -Xlinker $(LDDIR)$(MCU).x -mmcu=$(MCU)
# all object files for the protocol code
OBJS= rodl.io appl.io
TTPAOBJS= bus_swuart.o bus_swuart_slave.o ifs.o ifs_weaks.o ifs_filetab.o \
ifs_init.o ifs_exechrec.o ms.io ms_slave.io ttpa.o ttpa_fb.o ttpa_slave.o \
ttpa_sysfiles.io ms_request.o main.o ifs_bitvec.o schedule.o ttpa_lifesign.o
HEADERS=
vpath %.h ./:$(TTPAROOT)/include/:$(MONITORINGROOT)/
vpath %.c ./:$(TTPAROOT)/ttpa/:$(MONITORINGROOT)/
vpath %.S ./:$(TTPAROOT)/ttpa/

all: $(TTPAOBJS) $(OBJS) $(PRJNAME).elf $(PRJNAME).eep $(PRJNAME).hex

%.s: %.c
        $(CC) $(CFLAGS) -S $< -o $@

%.is: %.s
        $(IFSCONV) <$< >$@

%.io: %.is
        $(CC) $(ASFLAGS) -x assembler -c $< -o $@

%.elf:  $(TTPAOBJS) $(OBJS)
```

```
main(void)
{
        /* Initialization of TTP/A Protocol   */
        /* set Input/Output ports, timer, ... */
        init_ttpa();
        /* Initialization of Monitoring       */
        /* set Baudrate, ...                  */
        init_monitoring();
        /* Initialization of User-Tasks       */
        init_user();
        while (1)
        {
                /* Background - Task */
                ttpa_bgtask();
                /* Monitoring- Task */
                ttpa_monitoring_task();
        }
}
```

Figure 23: Calling the monitoring server parts in the node's program



		Rec 00 Byte 00	Rec 00 Byte 01	Rec 00 Byte 02	Rec 00 Byte 03	Rec 01 Byte 00	Rec 01 Byte 01	Rec 01 Byte 02	Rec 01 Byte 03	Rec 02 Byte 00	Rec 02 Byte 01	Rec 02 Byte 02	Rec 02 Byte 03	Rec 03 Byte 00	Rec 03 Byte 01	Rec 03 Byte 02	Rec 03 Byte 03	Rec 04 Byte 00	Rec 04 Byte 01	Rec 04 Byte 02	Rec 0 Byte
✓	**RODL0**	0x20	0x04	0x86	0x00	0x80	0x01	0x20	0x01	0x81	0x01	0x20	0x02	0x82	0x01	0x20	0x43	0x00	0x00	0x00	0x
	RODL1	0x20	0x00	0xDE	0x00																
	RODL2	0x20	0x00	0x00	0x00																
	RODL3	0x20	0x00	0x00	0x00																
	RODL4	0x20	0x00	0x00	0x00																
	RODL5	0x20	0x00	0xE6	0x00																
	RODL6	0x20	0x00	0x00	0x00																
	RODL7	0x20	0x00	0x00	0x00																
✓	Config File	0x20	0x04	0xF8	0x00	0x00	0x00	0x00	0xFE	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0xA3	0x
	File 09	0x20	0x00	0x00	0x00																
✓	ROSE	0x20	0x11	0x9A	0x00	0x00	0x02	0x0A	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x
	File 0B	0x20	0x00	0x00	0x00																
	File 0C	0x20	0x00	0x00	0x00																
	File 0D	0x20	0x00	0x00	0x00																
	File 0E	0x20	0x00	0x00	0x00																
	File 0F	0x20	0x00	0x00	0x00																

Edit Data: File 0, Rec. 0, Byte0 0x00, Byte1 0x00, Byte2 0x00, Byte3 0x00 — Write Rec.

Master Node Info: Cluster: 0x01, ID: 0xFE, Series-Nr: 0x89 AB CD EF, Serial-Nr: 0x01 23 45 67

Slave Node Info: Node ID, Series-Nr: n/a, Serial-Nr: n/a

File Logging: File 0 — activate

	Slot 00	Slot 01	Slot 02	Slot 03	Slot 04	Slot 05	Slot 06	Slot 07	Slot 08	Slot 09	Slot 0A
RODL 0	FB	RCV F20 R01 B0	RCV F20 R01 B1	SND F20 R01 B2	IRG						
RODL 1	n/a										
RODL 2	n/a										
RODL 3	n/a										
RODL 4	n/a										
RODL 5	n/a										
RODL 6	n/a										

Connection: Port: COM 2, BAUD Rate: 19200

384 Bytes in 3615 msec -> 106 B/s

Connected to COM2

Figure 24: Screenshot of the Monitoring-Tool

5 Hardware Description

5.1 Programming Interface

The programming interface hardware consists of the programming cable and two connectors, one fitted at the 4433 slave and the other connected to the parallel port of the PC.

At a TTP/A node the programming connector is marked with a green pinhead. To identify the six pins of this pinhead a colored label is applied:

Color	Pin Atmel connector	Pin Printer port
green	1 (RESET)	16 (INIT)
magenta	2 (SCK)	1 (STROBE)
yellow	3 (GND)	21 (GND)
white	4 (MISO)	11 (BUSY)
brown	5 (MOSI)	2 (DATA)
red	6 (VCC)	not connected

Table 8: Programming Port Signals

The pins at the node and at the programming cable are marked in the way described above, so if the connector is placed in the right way the color codes on the pinhead match with the colors on the programming cable (see figure 26).

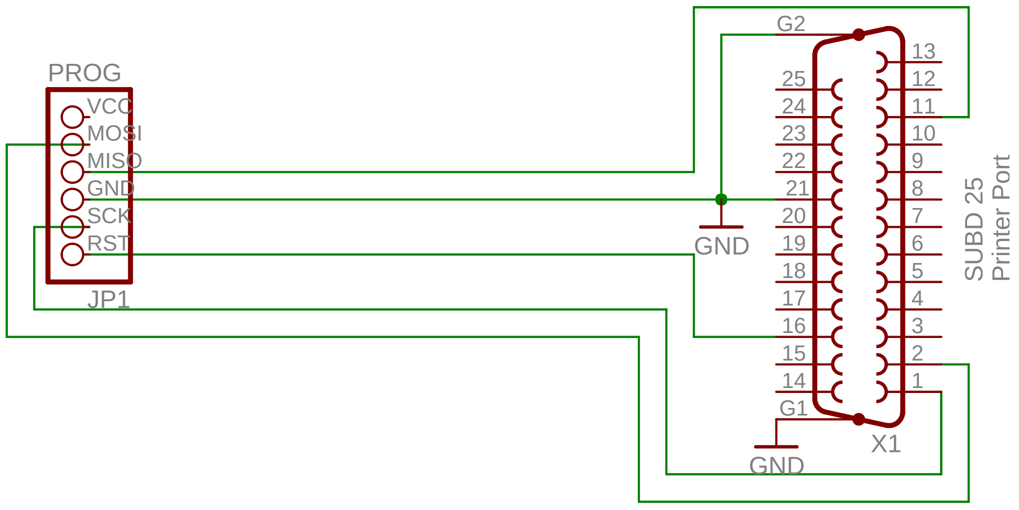


Figure 25: Programming Cable

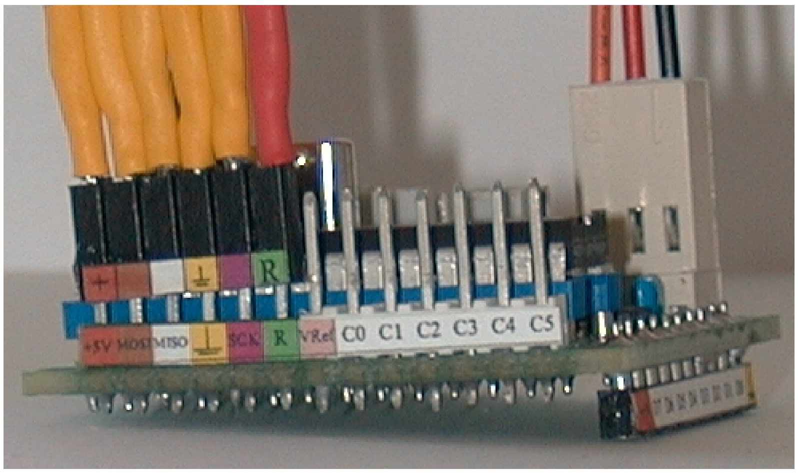


Figure 26: Applying the programming cable at the slave node

5.1.1 Applying the Programming Cable to the 4433 Slave Node

To modify the flash or EEPROM memory of a node the programming cable must be applied as depicted in figure 26. The other side of the programming cable is connected to a PC via the parallel port. Furthermore the node needs a suitable power supply, in our case the TTP/A bus itself.

5.2 Monitoring Interface

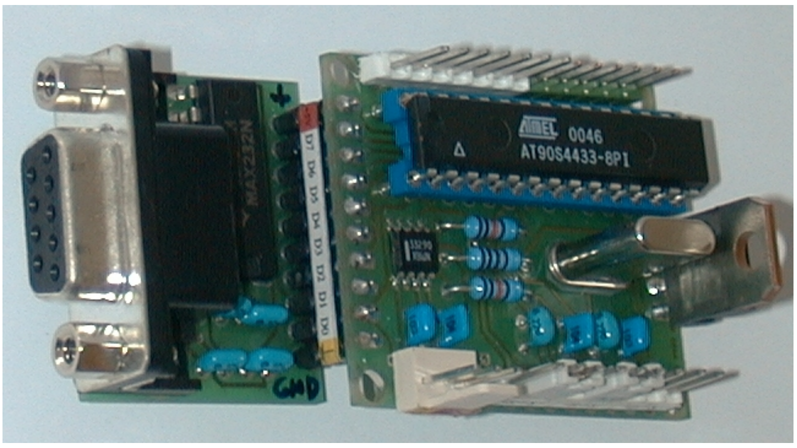


Figure 27: Attaching the monitoring interface to a TTP/A node

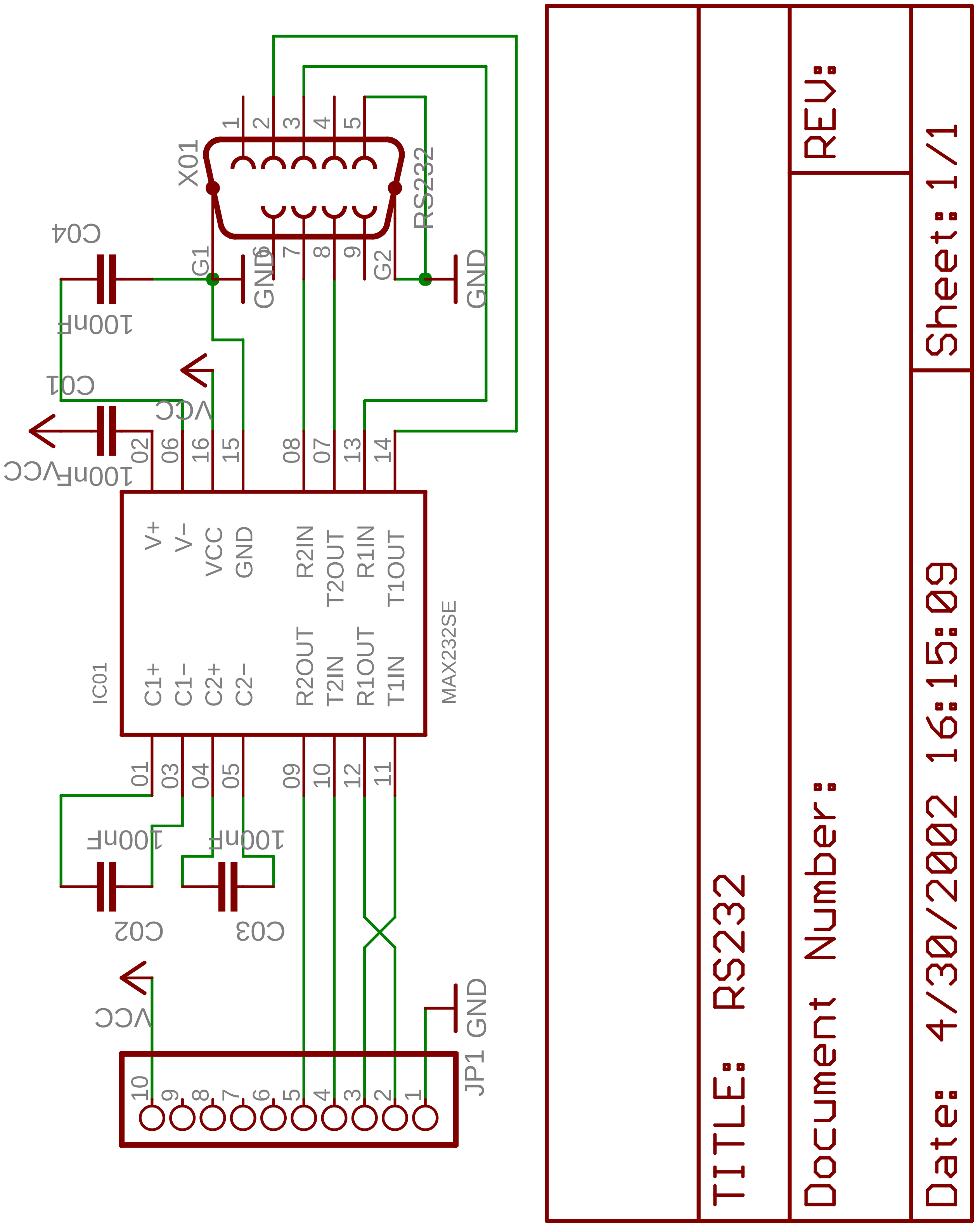


Figure 28: Schematic of the Monitoring Interface

5.3 Atmel 4433 Slave Node

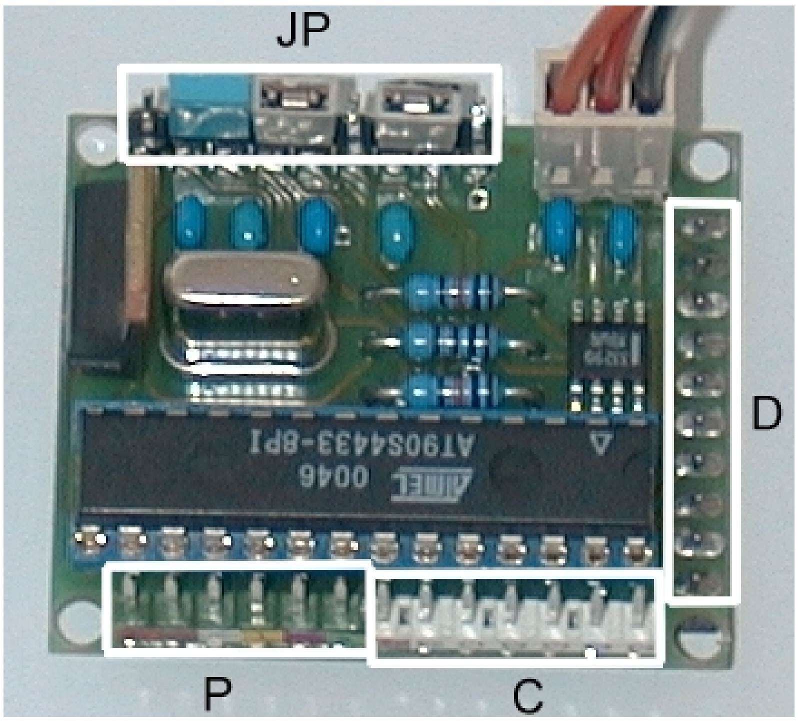


Figure 29: Top View

In figure 29 four parts of the 4433 slave node are emphasized and labeled.

- The pinhead used to apply the programming cable, labeled with the letter P.
- In case port C is used as an analog input port, a reference voltage is required. This single reference pin and the six pins of port C are labeled with the letter C.
- The eight pin port D is labeled with the letter D. This port is socketed and is placed on the bottom of the PCB.
- The pinhead labeled with JP is used to configure the 4433 slave node (see 5.3.1).

5.3.1 Setting the Jumpers of the 4433 Slave Node

While older versions of a 4433 slave node uses three jumpers for configuration (see below), newer board designs uses software configuration. Such a new designed board can be identifyed by the presence of a 74HC4053 IC and the absence of the jumper pinhead.

This software configurable 4433 slaves uses two pins of port B to configure port C (digital/analog) and to select the UART type used by the protocol software (HW-UART/SW-UART). The UART

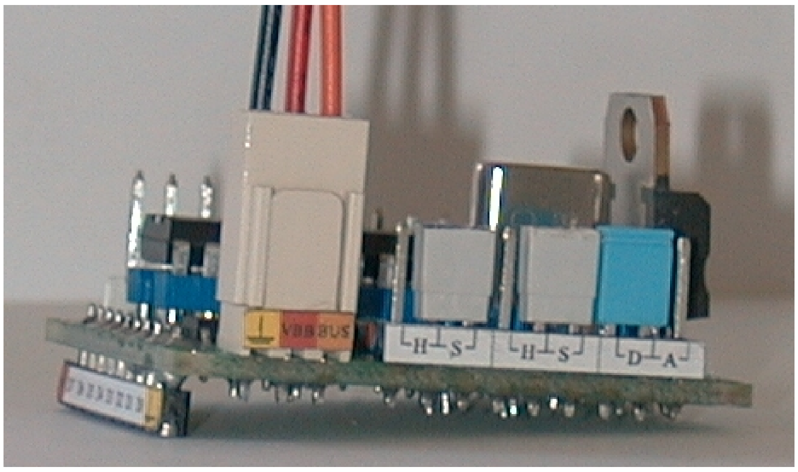


Figure 30: How to set the Jumper

type is selected via pin 2 of port B. A set bit 2 means the 4433 slave uses a SW-UART, a cleared bit 2 selects the HW-UART. Via pin 3 of port B the power supply of port C is selected. If bit 3 is set port C is configured for digital IO, while a cleared 3rd bit powers port C for analog measurements.

In figure 30 one can see the three jumpers used to configure the 4433 slave node. The two white jumper blocks are used to select the UART type (hardware[2] or software UART). In this picture both UARTs, transmitter and receiver, use a software UART (labeled with the letter S).

To use the hardware UART one must connect the TXD and RXD pin with the bus driver. Closing the pins labeled with H do so. The three pins beside the bus connector refer to the receive UART, while the connection of the transmit UART can be chosen via the middle three pins. Placing these jumpers right do not switch the protocol software from hardware UART to software UARD, this selects just the connection MCU – bus driver.

The three pins with the blue jumper block near the voltage regulator are used to select the supply of port C. In case port C is not used this jumper can be neglected. If port C is used as a digital in- or output the jumper block must connect the two pins labeled with letter D. For using port C as an analog input one must connect the two pins labeled with letter A.

In case the jumper block is removed, port C is disconnected from the power supply and the behavior of port C is undefined.

5.3.2 Schematic of the 4433 Slave Node

Figure 31 shows a schematic of the 4433 slave node.

[2]If the HW UART is chosen pin 0 and pin 1 of port D are used by the UART.

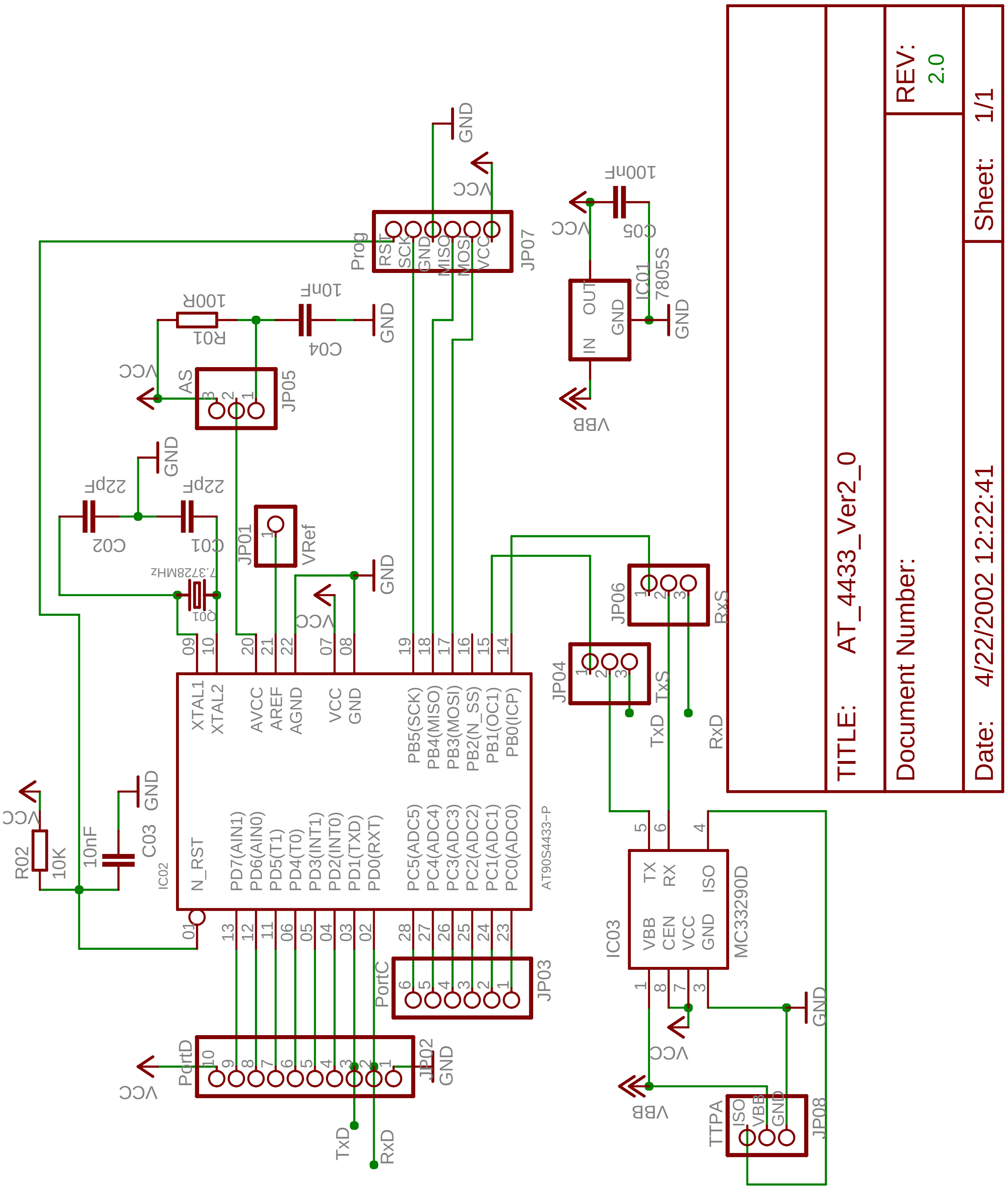


Figure 31: Schematic of the 4433 Slave Node

6 Software Description and Setup

Currently it is possible to develop, program, and debug smart transducer applications using tools either for Linux or Windows.

6.1 Linux Software

6.1.1 Compiler

Basically, we use the AVR-GCC compiler v3.0[3]. This is an open source cross-compiler that supports the Atmel AVR series. The compiler package is freely available for Windows and Linux systems. Since it is a widely-used tool, the software is well-tested and supported by the open source community.

Besides these arguments, open source software has the advantage of providing *a priori* information on the exact implementation of functions like interrupts or register usage in the compiled programs. While this information might be undocumented in commercial compiler products, with open source software it is possible to directly verify or modify the compiler's code.

In [Trö02b], Trödhandl describes in detail changes for a beta version of GCC v3.3 that improve the temporal behavior of the generated code. The main achievements are reduced interrupt latency when using interruptible interrupt handlers and a reduction of the required stack memory by implementing a detection of used registers. These features are necessary in order to achieve maximum transmission rates and minimum memory footprints for TTP/A implementations. However, for communication rates up to 50 kBit/sec, the TTP/A protocol implementation also works with the non-optimized version of the AVR-GCC compiler v3.0.

6.1.2 Programming Tools

A cross-compiler compiles the program source code and locally generates a file that contains the object code for the embedded system. In order to program the embedded microcontroller with this code, a programming tool that establishes communication to the chip is necessary. All Atmel AVR microcontrollers can be in-system-programmed with the same basic programming algorithms. Due to this standardization, there are many available commercial and open source systems that enable AVR programming. We have evaluated the following tools for Linux:

UISP: The Micro In-System Programmer for Atmel microcontrollers is a utility that is freely available for Windows and Linux systems in slightly different versions. It supports in-system programming, Atmel's prototype board/programmer, and an extremely low-cost parallel port

[3]Distribution from AVR Freaks (2001-07-01), http://www.avrfreaks.net/

programmer. However, the programming speed of the tool is tedious and the current version UISP 1.0b does not support the new ATmega128 chip that is used for making navigation decisions in the smart car. The host system must provide a free parallel port for attaching the programming cable.

ZEUS Programmer: When programming smart transducer networks, it is often necessary to program multiple nodes subsequently with different programs. Since the before mentioned programming systems only provide a single target interface, frequent switching of the programming cable from chip to chip is a common practise.

The ZEUS programmer system developed at our institute by Haidinger overcomes this problem by providing eight target interfaces that can be accessed via multiplexing. Thus, a system containing up to eight different AVR microcontrollers can be programmed in one go. The software is speed-optimized for the different controller types and provides the same speed as the commercial AVR STK500 system [Hai02]. The ZEUS programmer runs on any Linux or Windows computer system and needs a single free serial port.

6.2 Windows Software

6.2.1 Compiler

We use the same AVR-GCC Compiler (v3.0) as in the Linux system. After installation, the compiler environment can be started in an MSDOS window.

6.2.2 Programming Tools

We have evaluated the following tools for Windows:

UISP: There is also a windows version of the Micro In-System Programmer for Atmel microcontrollers available. UISP for Windows is a bit different from the Linux variant, but also freely available. If you want to use UISP for node programming, you need a computer running Windows 95 or Windows 98. Windows NT, 2000, or XP will *not* work (we have no experiences from Windows ME)!

AVR STK500: The STK500 starter kit is a commercial tool available via Atmel's distributors and provides slots for all 8-, 20-, 28-, and 40-pin AVR devices as well as an interface to an external target system. It is supported by Atmel's AVR studio, a tool with a graphical user interface. The STK 500 system runs on any Windows computer system and needs a free serial port on the development system, i. e., a PC.

ZEUS Programmer: The ZEUS Programmer works also with every Windows system, as long as a terminal program for accessing the serial port is available.

The cheapest solution is to use the UISP programmer and a programming cable for the parallel port. First you need to install the driver `TVicHW50`. The driver enables access to the parallel interface of AT/ATX computers. The parallel interface should be configured to EPP mode in the bios. After installing the driver you may install the Micro In System Programmer (UISP) for windows. We used version 1.1.2.0, it is available as a packet with a self-installer. We recommend to change the installation path to `C:\UISP` in order to be able to easily remember the path to the UISP tool.

UISP can be used in two modes, the interactive mode (started with `C:\UISP\UISP /term`) and the command line mode. It is recommended to start the tool in interactive mode after installation in order to verify that the programming tool is working. After correct installation of driver and UISP programmer and connection of the parallel cable to the node (the node has to be supplied with voltage, see section 5.1.1 for details), the UISP program will report the following:

```
C:\avrgcc>c:\uisp\uisp /term
Micro In-System Programmer Version: 1.1.2.0
Device AT90S4433 found.
FLASH: 4096 bytes
EEPROM: 256 bytes
Entering the AVR Terminal. ?-help, q-quit.
avr>
```

Figure 32: UISP connected successfully to a node

In case of misconfiguration, UISP will fail to connect to the node and reports a failure after some tries:

```
C:\avrgcc>c:\uisp\uisp /term
Micro In-System Programmer Version: 1.1.2.0
Exceeded 32 tries to find device.
Entering the AVR Terminal. ?-help, q-quit.
avr>
```

Figure 33: UISP failed after 32 tries

In that case check the following:

- Was the `TVicHW50` driver installed *before* the installation of UISP?
- Is the license of the `TVicHW50` driver still valid?
- Did you use the right programming cable and is it correctly connected to the parallel port and the Atmel Microcontroller?
- Was the parallel interface set to EPP in the computer's BIOS?

- Did you connect the node to a power supply (e. g. TTP/A bus)?

Good luck!

6.2.3 Local Monitoring Tool

The local monitoring tool is a program written in Delphi that connects to a node via an RS232 serial interface. The tool consist of a single executable file and needs no complex installation. However, it requires the library `vcl30.dpl`. The usage of the tool is described in section 4.

7 Summary

TTP/A is the low-cost member of TTP. It is a multi-master sensor bus system which meets the requirements of the TTA like the provision of a global time to the sensor nodes. To keep the complexity and thus cost of the smart-sensor nodes low, TTP/A is designed as a master/slave protocol. The master, which is controlling the bus and maintaining data exchange with system networks or host computers, performs the complex parts of the protocol. The sensor nodes execute the TTP/A slave protocol. TTP/A supports multiple sensor views: an operational view and a diagnostic view. Operational and maintenance activities are separated to reduce system complexity.

The interface file system (IFS) provides a common name space for data items that are exchanged among transducer nodes and the master node in a distributed real time system. It establishes a stable intermediate structure that is a solid base for many new services. The IFS is a fundamental part of the diagnostic and maintenance interface and is used to provide features like the *plug-and-play* capability. In comparison to other protocols, TTP/A has only low hardware requirements.

We presented a programming model for TTP/A featuring time-triggered TTP/A tasks, interrupt handlers and background tasks. Furthermore the user may specify initialization code, which is executed once at startup of the node.

We propose to use a tool like `make` for automating the development process. It is also recommended to add functions for the monitoring task, if the hardware UART is not used by a user application and the node still provides enough resources in RAM and ROM memory. The monitoring client provides access to the IFS, which is the interface between the protocol and the local node application.

7.1 Further Reading

A main idea of TTP/A is that the protocol is not bound to specific hardware. The hardware related parts of this document refer to nodes based on the AVR8 architecture. In principal, TTP/A can be run on any microcontroller with at least 64 bytes of RAM and 4K of FLASH. If you are interested in other hardware platforms and porting issues, the work from Stefan Krywult [Kry06] will provide helpful information.

We hope that the information provided in this document is useful for future developers working with TTP/A. If you have further questions regarding TTP/A please have a look at the resources and contacts provided at the TTP/A webpage (hosted by the Real-Time Systems Group at the Vienna University of Technology)[4].

[4]http://www.vmars.tuwien.ac.at/ttpa/

References

[Elm02] W. Elmenreich, W. Haidinger, P. Peti, and L. Schneider. New Node Integration for Master-Slave Fieldbus Networks. In *Proceedings of the 20th IASTED International Conference on Applied Informatics (AI 2002)*, pages 173–178, February 2002.

[Elm04] W. Elmenreich, S. Pitzek, and M. Schlager. Modeling Distributed Embedded Applications on an Interface File System. In *Proceedings of the 7th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 175–182, May 2004.

[Hai00] W. Haidinger and R. Huber. Generation and Analysis of the Codes for TTP/A Fireworks Bytes. Research Report 5/2000, Technische Universität Wien, Institut für Technische Informatik, Vienna, Austria, 2000.

[Hai02] W. Haidinger. A Tool for Programming AVR Microcontroller Networks – the ZEUS Programmer. Research Report 51/2002, Technische Universität Wien, Institut für Technische Informatik, Vienna, Austria, 2002.

[Jon02] C. Jones, M.-O. Killijian, H. Kopetz, E. Marsden, N. Moffat, D. Powell, B. Randell, A. Romanovsky, R. Stroud, and V. Issarny. Final Version of the DSoS Conceptual Model. *DSoS Project (IST-1999-11585) Deliverable CSDA1*, October 2002. Available as Research Report 54/2002 at http://www.vmars.tuwien.ac.at.

[Kop01] H. Kopetz, M. Holzmann, and W. Elmenreich. A Universal Smart Transducer Interface: TTP/A. *International Journal of Computer System Science & Engineering*, 16(2):71–77, March 2001.

[Kry06] S. Krywult. Real-Time Communication Systems for Small Autonomous Robots. Master's Thesis, Technische Universität Wien, Institut für Technische Informatik, Vienna, Austria, 2006.

[Obj01] Objective Interface Systems, TTTech Computertechnik, and VERTEL Corporation. Smart Transducers Interface. *OMG TC Document orbos/2001-06-03*, July 2001. Supported by Technische Universität Wien. Available at http://www.omg.org.

[Trö02a] C. Trödhandl. Architectural Requirements for TTP/A Nodes. Master's Thesis, Technische Universität Wien, Institut für Technische Informatik, Vienna, Austria, 2002.

[Trö02b] C. Trödhandl. Improving the Temporal Behavior of a Compiler for Embedded Real-Time Systems. Research Report 50/2002, Technische Universität Wien, Institut für Technische Informatik, Vienna, Austria, 2002.