University of West Bohemia

C-BASED REFERENCE MODEL OF TTP/C PROTOCOL

Jan Ježek

April 2001

Abstract

C-Based Reference Model of TTP/C Protocol
This document brings a survey of the TTP/C (Time-Triggered Protocol for SAE Class C Applications) properties and compares them to other available similar protocols, namely CAN (Controller Area Network) and Byteflight. Second part of this document brings a description and programming manual for a functional specification of the TTP/C protocol and its incorporation into the C-Sim simulation environment. The last part deals with an example implementation of the simulation and presents a simple application.

Contents

Chapter 1

Introduction

At first this diploma thesis brings a brief description of the Time-Triggered Protocol (TTP/C) and compares it to other available competitive protocols namely the Controller Area Network (CAN) and the Byteflight. Different aspects of the protocols are compared and then given arranged in tables.

The second and longest part of this thesis was to create a C-based model of the TTP/C protocol and to present a description and a programming manual for it. The structure and the behavior of the model of the TTP/C protocol is specified here. The C-based functional specification was created using a natural language specification of the protocol [SPEC]. The description was written to make it possible for anyone to use the specification and to allow him to alter it or extend it to a for example newer protocol version.

The third part offers a way how to incorporate the functional specification into a simulation environment, namely the C-Sim. An example of the simulation was created and is also presented here. The example is very simple and was chosen so to make it as easy as possible to understand and to give the reader the basic knowledge to create a more complex simulations.

Chapter 2

Time-Triggered Protocol

2.1 Overview

The TTP/C protocol provides standard specification of message interchange between different electronic nodes. The TTP/C protocol is time-triggered. That means that each node has an unique time slot in which it is allowed to transmit data. The time slot is perfectly known to the other nodes in the whole system and that makes bus collisions all but impossible even at peak loads.

The protocol is designed to provide the following services:

- Ability to quickly detect and handle single fault in any node with zero impact to the other nodes in the system.
- Instant membership knowledge.
- Internal support for replicated shadow nodes.
- Implicit acknowledgement and thus no necessity for any extra acknowledgement messages.
- Support for mode changes.
- Both internal and external clock synchronization algorithms. The internal clock synchronization is performed without any increase in message size or number.
- High data transfer efficiency because of generally small protocol overhead.
- An unique clique avoidance mechanism to recover from Byzantine like faults.

2.2 TTP/C Network Structure

The TTP/C network consists of a set of interconnected nodes. A TTP/C node is often denoted as SRU (Small Replaceable Unit). On or more such SRUs can be combined to form a FTU (Fault-Tolerant Unit). A FTU delivers its service even if one (or more, if designed that way) of its SRUs fail.

The TTP/C protocol does not require or rely on any particular physical layer. It can run on top of a bus architecture as well as a star architecture and still it keeps all of its benefits. It uses two independent channels to handle faults that occur during message transmission and even permanent faults at the connection media (e.g. a broken cable).

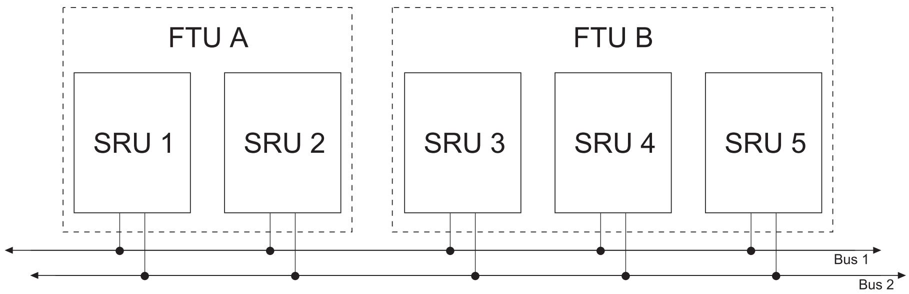

Figure 2.1: Example of a Bus Architecture

An independent set of nodes connected together on the basis of TTP/C protocol is called a cluster. The cluster executes user applications.

2.3 TTP/C Node Structure

A structure of the TTP/C node is depicted in figure 2.3. Each single SRU consists of two major parts: the TTP/C communication controller and the host computer. The communication controller runs the protocol and the host computer runs the application built on top of the TTP/C protocol.

Between the host computer and the communication controller there is a special memory – the CNI (Communication Network Interface), that serves as a place, through which the two subsystems interchange data. The communication controller uses it to store its state and received messages and the host computer puts there the data to be sent and may alter several protocol parameters through it. In hardware the CNI is implemented as a DPRAM (Dual-Ported Random Access Memory).

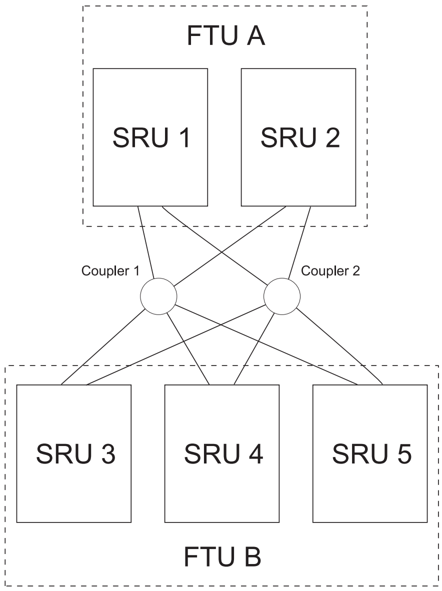

Figure 2.2: Example of a Star Architecture

There is an additional single signal line from the communication controller to the host computer that is used as an interrupt signal. The host computer can select which interrupts from a set it wants the controller to be issuing. It may select all, some or even none of the interrupts.

The communication controller comprises the protocol processor, the MEDL (Message Descriptor List) and the bus guardian. The protocol processor is a hardware controller that runs the TTP/C protocol microcode, the MEDL is an EEPROM memory which contains the control data for the node and the bus guardian protects the communication channels from timing failures of the controller.

2.4 The Protocol Operation

The communication channels are accessed on the basis of TDMA (Time-Division Multiple Access) scheme. All SRUs intended to send are assigned a time slot in a sequence called TDMA round. Each of those SRUs can send and must send exactly one message on each channel in one TDMA round. For each application the slot pattern in a TDMA round is determined at design time and remains the same all the time. Only the type and length of messages may differ.

The sequence of all TDMA rounds with different messages forms a cluster

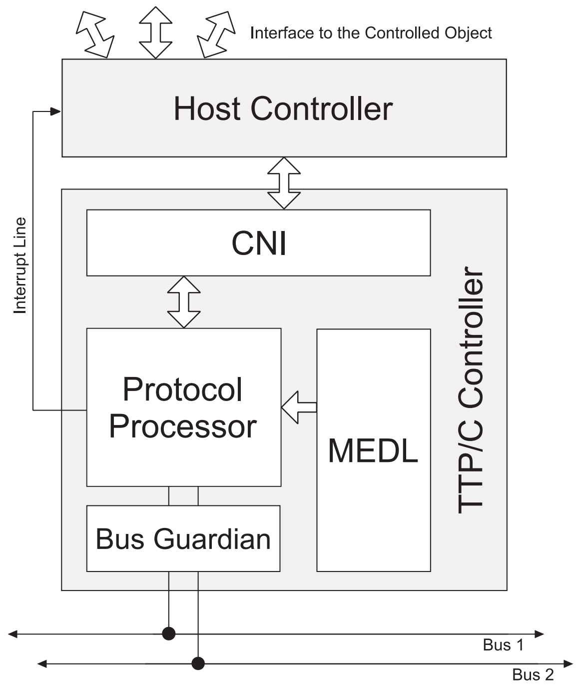

Figure 2.3: A TTP/C Node

cycle. A sequence of 2, 4, 8 or 16 cluster cycles of the same length forms a transmission cycle. After the transmission cycle is finished a new one is then started immediately and everything goes all over again.

In each node the communication controller operates independently of the host computer. It requires no control signals, all the data required resides in the MEDL. The MEDL content is unique for each node in the network and provides the following information:

- For each slot in the transmission cycle it contains information on what message from which node has to be sent.
- The CNI field where to put the received data or from where to get the data to be sent. The number of these entries corresponds to the length of the transmission cycle and thus to the total number of slots.
- Various additional data needed by the protocol such as node-specific identification numbers and predefined constants.

During normal operation all the nodes in the system are synchronized and agree on the current position in the MEDL. Therefore the actions carried out in all the nodes as specified in the MEDL perfectly correspond to each other (One node transmits a message, the others receive etc.).

All the timing in the TTP/C nodes is based on so called macroticks. A macrotick has a given cluster-wide duration (e.g. 1 microsecond). All time constants are given in macroticks, for example a SRU slot may be 250 macroticks long, TDMA round length may be 1000 macroticks etc. Besides macroticks, as can be estimated, microticks also exist. However a microtick duration is not a cluster-wide constant. A macrotick may consist of different number of microticks in different controllers. Even a non-integral number of microticks may form an macrotick, this is not forbidden by the protocol.

2.5 Protocol Layers

The protocol is organized in a set of layers as depicted in figure 2.4. This is just a conceptual model of logical structuring, not an implementation model. Below is a only a brief description of each of the layers. A more detailed description can be found in [SPEC].

2.5.1 Data-Link Layer

The data-link layer is responsible for the exchange of frames between the SRUs.

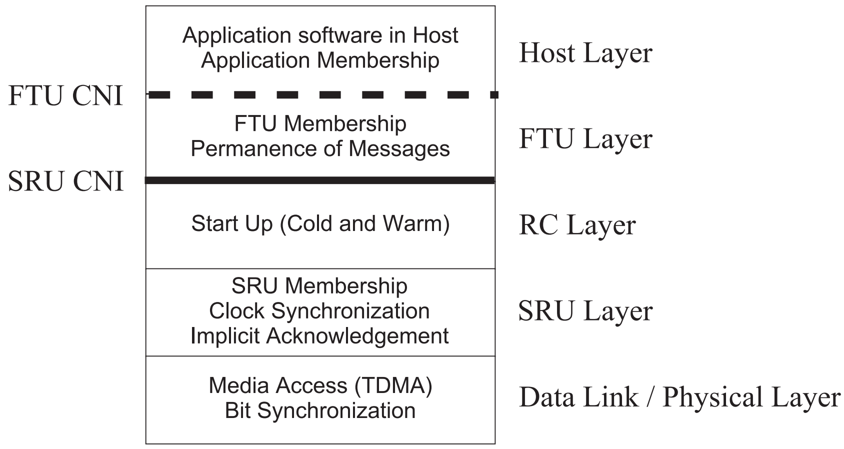

Figure 2.4: TTP/C Protocol Layers

2.5.2 SRU Layer

The SRU layer controls the functions of a SRU when it is synchronized with the rest of the cluster, namely:

- Store the received data into the CNI according to the MEDL table.
- Maintain membership service.
- Implicit message acknowledgement.
- Internal clock synchronization.
- Consistent mode change service.

2.5.3 Reconfiguration Layer (RC Layer)

The reconfiguration layer (RC layer) controls the mechanisms for the startup of a TTP/C cluster. It is responsible for the reintegration of SRUs that have lost their membership.

2.5.4 FTU Layer

The FTU layer manages the replicated SRUs within a FTU. It ensures that a failed SRU is instantly replicated by another from within its FTU. It performs a redundancy management. A FTU layer must ensure that the data can only be visible to higher layers after it has become permanent.

Several different approaches to achieve this exist:

- Two fail-silent SRUs. This FTU configuration will provide its service as long as at least one of the SRUs operates correctly.
- Three SRUs formed to a TMR (Triple Modular Redundancy). This FTU configuration tolerates a failure of any one of the SRUs.
- Four SRUs work in TMR mode even if one of them fails or is temporarily removed for maintenance.
- Single SRU can form a FTU if redundancy is maintained at a higher layer (e.g. four wheels of a car in speed measurement).
- A distributed FTUs can also be formed. If the system that consists of subsystems S_a, S_b and S_c is distributed to SRUs A, B and C in a way that SRU A executes subsystems S_a and S_b, SRU B executes subsystems S_b and S_c and SRU C executed subsystems S_a and S_c, a fail silent failure of any of this SRUs can be tolerated.

2.5.5 Host Layer

The host layer executes the application. It provides the data to be sent over the TTP/C network. It is also responsible for continuous updating of the host lifesign (see 4.10.1).

The host layer is not an integral part of the protocol.

2.6 Protocol Constraints

The TTP/C system is a distributed system and it will work properly only if a minimum of four SRUs is operational within a cluster.

If less than four SRUs are present, the cluster will operate in a degraded mode. The clock synchronization algorithm will no longer be Byzantine resilient. A cluster with two SRUs will only operate properly if no faults occur and a single SRU cannot operate at all. Of course that would be nonsensical anyway.

Chapter 3

The Competitors

3.1 Controller Area Network Protocol

Controller Area Network (CAN) protocol has the following properties:

- Prioritization of messages.
- Guarantee of latency times.
- Configuration flexibility.
- Multicast reception with time synchronization.
- System wide data consistency.
- Multimaster.
- Error detection and error signalling.
- Automatic retransmission of corrupted messages as soon as the bus is idle again.
- Distinction between temporary errors and permanent failures of nodes and autonomous switching off of defect nodes.

The CAN protocol is structured into several layers:

- The physical layer defines how signals are actually transmitted. The physical layer is not nailed down in the specification to allow implementations to be optimized for their application.

- The MAC (Medium Access Control) sublayer represents the kernel of the CAN protocol. It manages the exchange of messages between physical and LLC layers. The MAC layer is responsible for message framing, arbitration, acknowledgement, error detection and signalling, and fault handling.
- The LLC sublayer is concerned with message filtering, overload notification and recovery management.

On the contrary to TTP/C the CAN protocol does not use time slots when accessing the bus. Instead it uses CSMA/CA (Carrier Sense Multiple Access with Collision Avoidance) with so called bitwise arbitration. If 2 or more units start transmitting messages at the same time, the bus access conflict is resolved and the mechanism does not cost any extra time. During arbitration every transmitter compares the level of the bit transmitted with the level that is monitored on the bus. If the transmitted bit level is greater or equal to the monitored one, the unit will continue to send, otherwise it will stop and wait until the bus frees again.

More on the CAN protocol can be found in the protocol specification [CASP].

3.2 Byteflight Protocol

The Byteflight protocol combines the advantages of synchronous and asynchronous bus systems:

- Access to the bus is not controlled by the master but is completely time controlled by the protocol controller implemented in hardware.
- Guaranteed latency for a certain number of top-priority telegrams even at maximum bus load, so the highest-priority message cannot block the bus.
- At lower bus loads, low-priority messages can be transmitted using asynchronous access methods.
- Transmission of an alarm state (safing sensor function) via the same physical medium and detection of this state by two independent components.

The Byteflight protocol uses an optical bus. The individual nodes are then connected to one another via an active star coupler. When using alternative electrical connection, a bus architecture may be used. This is not recomended though.

One node in the system is a special bus master, that outputs synchronisation pulses. Each node can be configured as a bus master by means of the software. However, there must be only one bus master in the bus system. All the other connected nodes (bus slaves) use the sync pulse for internal synchronisation. If the bus master breaks down, one of the other nodes is configured as the master automatically.

3.3 The Comparison

3.3.1 Basic Protocol Properties

The TTP/C protocol uses TDMA media access strategy. The mechanism to resolve concurrency between nodes is determined at design time and no collisions occur at runtime. It can have up to 64 participants and can achieve up to 2 Mbit/s data rate with various physical media.

The CAN protocol uses CSMA/CA media access strategy. It resolves concurrency between nodes by using bit arbitration. It has typically (but is not limited to) up to 20 nodes and speed of approximately 1 Mbit/s on cheap media. High speeds are difficult to achieve at longer distances though.

The Byteflight protocol uses application-driven minislotting and the concurrency is resolved at design time. It can have up to 22 participants and achieve speeds at up to 10Mbit/s.

	TTP/C	**CAN**	**Byteflight**
Media access	TDMA	CSMA/CA	Minislotting
Collision resolving	at design time	arbitration	at design time
Network size	up to 64	up to 20	up to 22
Data rate	2 Mbit/s	1 Mbit/s	10Mbit/s

Table 3.1: Basic Protocol Properties

3.3.2 Features

The TTP/C protocol has (depending on controller) ca. 2 kbytes of message buffers. The messages are transmitted and received on two redundant channels according to a schedule and are acknowledged implicitly. The message sequence repeats each cluster cycle in a fully predictable fashion. The clock in each node is synchronized by a distributed algorithm. Each node has instant knowledge on the state of other nodes by using membership bit vector.

The CAN protocol currently has ca. 120 bytes of message buffers. Messages are transmitted on request but possibly delayed by collision and must be acknowledged explicitly. There are no redundant channels, no global timebase and no membership service.

The Byteflight protocol has ca. 240 bytes of message buffers. There are synchronous and asynchronous messages with IDs, no acknowledgement but frequent retransmission. The clocks are synchronized by sync pulse from bus master. There are no redundant channels and no membership service.

	TTP/C	**CAN**	**Byteflight**
Message buffers	2 KB	120 B	240 B
Global timebase	yes	no	sync pulses
Redundant channels	2 replicated	none	none
Membership service	yes	none	none
Acknowledgement	implicit	explicit	none

Table 3.2: Features

3.3.3 Performance

The TTP/C protocol reaches data efficiency of around 60-80%. Messages are 1-16 bytes long, have exactly 23 bits overhead and hamming distance of 6. There is no additional overhead.

The CAN protocol reaches up to 30% efficiency which falls with load. Messages are 1-8 bytes long plus at least 62 bits overhead and have hamming distance of 6. Bus load must be kept low to avoid trashing.

The Byteflight protocol efficiency is about 50%. Messages are 1-12 bytes long with large overhead of 40 bits plus 2 bits per data byte. The hamming distance is 6 as well. Additional overhead is caused by propagation delays and sync pulses.

	TTP/C	**CAN**	**Byteflight**
Data efficiency	60-80%	30%	50%
Message data size	1-16 B	1-8 B	1-12 B
Message overhead	23 b	> 62 b	> 40 b
Hamming distance	6	6	6

Table 3.3: Performance

3.3.4 Error detection and Fault-Tolerance

The TTP/C protocol uses two redundant channels with guaranteed consistency and synchronous timebase. Nodes have fail-silent timing and fail-consistent data transmission. The protocol internally supports redundant node replication and broken nodes are replicated instantly. Cliques are removed within a bounded time interval. The bus guardian eliminates the babbling idiot failure. Each node has numerous local error detection mechanisms.

Both CAN and Byteflight protocols do not support redundant channels, have fail-uncontrolled nodes with no support for redundant replication, no protection against babbling idiots and no local error detection mechanisms.

	TTP/C	**CAN**	**Byteflight**
Channel redundancy	supported	not supported	not supported
Node error detection	supported	not supported	not supported
Node replication	supported	not supported	not supported
Babbling idiot	avoided	possible	possible

Table 3.4: Error detection and Fault-Tolerance

3.3.5 Pros and Cons

The TTP/C protocol has formally verified mechanisms for high-level fault tolerance, strong error detection at node level, can run on several physical layers and has guaranteed latency for all messages in the system. On the other hand unplanned extensions require update of all nodes.

The CAN protocol is highly flexible, inexpensive and easy to use. It is currently widely available as integrated module on several CPUs. However it does not offer reliable mechanisms to build fault-tolerant or even safety-critical systems and it has a very limited capability for higher data rates.

The Byteflight protocol has high precision of central master clock synchronization and distributes the bandwidth flexibly between messages. The cons are existing single points of failure, no acknowledgement mechanism and heavily-application-relied fault tolerance strategies.

Chapter 4

TTP/C Functional Description

4.1 Structure of the Description

The TTP/C functional description (and the model) has been created on the basis of the document Specification of the TTP/C Protocol, specification version 0.1 of 1st February 1999. It is written in the C language in its ANSI-C norm and thus easily portable. It has been successfully tested on Windows and on several unix machines. It will not only work in less-than-32-bit environments.

The functional description is divided in several parts:

- The CNI data structure.
- The Interrupts.
- The MEDL data structure.
- The Data link layer, that deals with frames and channels behavior.
- The SRU layer, that explains functions that represent the TTP/C controller behavior.
- The RC layer, that describes the TTP/C controller's state diagram.
- The bus guardian.
- The controller's local data - these are the data that are not explicitly defined by the protocol specification but resulted as needed from the implementation.

Each of these parts will be described in detail in following chapters.

4.2 Common Characteristics

4.2.1 Object oriented programming

Because the model is created using the tool called C-Sim (see below) which is written entirely in C and by the fact that C is the most widespread language it had been decided that the model would also be written in C.

The C does not offer any means of object oriented programming, however in the following description the terms public field, private field etc. will be used occasionally. Such terms have the same meaning as they would have in object oriented programming. E.g. by private field an ordinary C structure field is meant, only it is treated as private in the program.

4.2.2 Constants

Important protocol constants are all placed in one file called `fp1const.h`, which is included as needed by other files. Bellow is a list of the constants and their meanings:

`MEDL_READ_FILE_NAME` – A template for file names that the MEDL tables will be read from.

`CRC_POLYNOMIAL_1, CRC_POLYNOMIAL_2` – Numbers representing the two CRC generator polynomials as defined in the specification.

`MAX_LENGTH_FOR_CRC_POLY_1` – The maximum length for a frame to use CRC generator polynomial 1. If the frame is longer than this value, polynomial 2 will be used.

`CRC_STARTING_VALUE` – The starting value of the CRC register. This is usually 0xFFFF.

`MAX_APPLICATION_DATA_SIZE` – The maximum number of bytes for the application data in N-frames. For the current protocol version this is 16.

`SYNC_STACK_SIZE` – Size of the stack for the disctributed internal clock synchronization algorithm (see below).

`CNI_MAE_NO_ADDRESS` – If the CNI storage address of an incomming message is equal to this value, the message will be discarded.

`C_STATE_VALID` – This value should be in the respective CNI field to consider the controller's C–state as valid.

`CONTROLLER_READY` – This value should be in the respective CNI field to consider the controller as ready.

`MAX_MODE_COUNT` – The maximum number of application modes, for current protocol version this is 30.

`FIRST_APP_MODE_NUMBER` – The number of the first application mode, i.e. the lowest number for a mode to consider it application mode.

`LAST_APP_MODE_NUMBER` – The number of the last application mode, i.e. the highest number for a mode to consider it application mode.

`COLD_START_MODE_NUMBER` – The mode number that is reserved for cold start. The cold start I–frame will indicate this mode number.

`STARTUP_MODE_NUMBER` – The mode number of the startup mode.

`DOWNLOAD_MODE_NUMBER` – The mode number of the virtual download mode. If a frame indicating download mode is received during startup the controller transits to download state. Though this is not implemented in the model.

`MAX_TRANSMISISON_CYCLE_LENGTH` – The maximum number of slots in the transmission cycle.

`MAX_SRU_COUNT` – The maximum number of active SRUs in the cluster (and thus the maximum length of membership vector etc.). Currently this is 64.

`BUS_GUARDIAN_FREQUENCY` – The clock frequency of the bus guardian in megahertzs.

`BUS_GUARDIAN_DIVISOR` – The divisor for the bus guardian parameters in the MEDL.

`MACROTICK_DURATION` – The duration of one macrotick given in microseconds.

`CLOCK_PRECISION` – The [Pi] parameter. This affects several protocol features, for details see the specification [SPEC].

`CHANNEL_SPEED` – The rate at which the channels transmit bits of data given in bits per second. This value is currently set to 1 Mbit/s.

4.2.3 Structure Field Access Scheme

Many C structures used in the code use a common field access scheme. The fields are not accessed directly, but by using special access functions that makes it possible to see C structures more encapsulated. Much like in object oriented programming.

Assume a structure is defined as:

```
typedef struct {
  ...
  unsigned cluster_time;
  ...
} CNI;
```

Than exactly these three access functions exist:

```
unsigned cni_get_cluster_time(CNI *cni);
```

Returns value of the field `cluster_time` of the structure pointed to by the `cni` parameter.

```
unsigned cni_set_cluster_time(CNI *cni, unsigned new_value);
```

Sets the field `cluster_time` of the structure pointed to by the `cni` parameter to the value of the `new_value` parameter. Returns the newly assigned value.

```
unsigned cni_increment_cluster_time(CNI *cni, signed how_much);
```

Increments the field `cluster_time` of the structure pointed to by the `cni` parameter with the value of the `how_much` parameter. If the `how_much` parameter is negative, the field will be decremented. Returns the new incremented value.

Since writing these functions separately for all fields of all existing structures would be very time-consuming and would produce huge source files, a set of special C preprocessor macros has been defined taking in advantage that the vast majority of structure fields are of the unsigned type. These macros serve in part for declaration of functions prototypes and in part for the actual definition of their bodies.

The macro for declaration of function that returns a particular structure field is defined as follows (for the precise definition see the source code):

```
#define get_function_prototype\
  (struct_name, func_name, field_name)
```

The struct_name parameter denotes the name of the type of the structure. The func_name parameter denotes the prefix for the function name (usually name of the type in lower case). The field_name parameter meaning is obvious.

The macros for functions for setting and incrementing fields are defined in a similar way. So to create prototypes for functions from the above example the following macro calls can be used:

```
get_function_prototype(CNI, cni, cluster_time);
set_function_prototype(CNI, cni, cluster_time);
increment_function_prototype(CNI, cni, cluster_time);
```

Defining function bodies are just as easy. A very similar macros exist defined as:

```
#define get_function_body\
  (struct_name, func_name, field_name, bit_size)
```

The meaning of the first three parameters remains the same. The fourth bit_size parameter specifies how large (in bits) that particular field should be. The macro-created functions then ensure that the field will never contain a value that exceeds its size. This allows to manage a large number of fields of different sizes in a convenient and portable way.

To create the bodies for the access functions from the example the following macro calls can be used (assuming the cluster_time field is 16 bits wide):

```
get_function_body(CNI, cni, cluster_time, 16)
set_function_body(CNI, cni, cluster_time, 16)
increment_function_body(CNI, cni, cluster_time, 16)
```

Notice that there are no semicolons at the end of the macro calls. The created functions will then be:

```
unsigned cni_get_cluster_time(CNI *cni)
{
return (cni->cluster_time & 65535);
}

unsigned cni_set_cluster_time(CNI *cni, unsigned new_value)
{
return (cni->cluster_time & new_value & 65535);
}
```

```
unsigned cni_increment_cluster_time(CNI *cni, int how_much)
{
return cni_set_cluster_time(cni->cluster_time + how_much);
}
```

To make declaration and definition of access functions even easier two additional macros exist:

```
#define declare_access_functions\
  (struct_name, func_name, field_name)
```

This macro just calls the three macros for access function prototypes passing them its arguments and thus creates prototypes for all the three types of access functions.

```
#define implement_access_functions\
  (struct_name, func_name, field_name, bit_size)
```

This macro is similar, only this time for function bodies. It creates bodies for all the three types of access functions.

So to fully cover the `cluster_time field` from the above example with access functions only two macro calls have to be provided besides the structure declaration. In the header file:

```
declare_access_functions(CNI, cni, cluster_time);
```

and in the respective C file:

```
implement_access_functions(CNI, cni, cluster_time, 16)
```

As can be seen, this saves a lot of typing.

As was mentioned above, these macros will work only with fields of the unsigned type. Therefore macros for signed fields are defined in a similar way:

```
#define declare_sig_access_functions\
  (struct_name, func_name, field_name)
#define implement_sig_access_functions\
  (struct_name, func_name, field_name, bit_size)
```

Finally macros also exist for indexed (array) fields, e.g. the membership vector field, which is defined in the `C_STATE` structure as:

```
unsigned membership[64];
```

can be covered with access functions by using the following macros:

```
#define declare_index_access_functions\
  (struct_name, func_name, field_name)
#define implement_index_access_functions\
  (struct_name, func_name, field_name, bit_size)
```

Each of the access functions will then have an additional parameter that represent the index, e.g.:

```
unsigned c_state_get_membership(C_STATE* c_state, int index);
```

4.2.4 CRC Calculation

The CRC code calculation is implemented in the `fl1func` module. This module contains the following function:

```
unsigned get_crc(
  unsigned *crc,
  unsigned polynomial,
  unsigned data,
  unsigned bits);
```

The function has following parameters:

`crc` – Pointer to the so-far calculated crc value. When starting a calculation the variable should be filled with the value of the `CRC_STARTING_VALUE` constant.

`polynomial` – The desired generator polynomial in binary form. Two constants (defined by the protocol specification [SPEC]) are defined for this parameter: `CRC_POLYNOMIAL_1` and `CRC_POLYNOMIAL_2`.

`data` – 1-32 bits of input data.

`unsigned bits` – The count of valid bits passed by the data parameter.

The function chains the data bit to the CRC and stores the new CRC value to the variable pointed to by the crc parameter. Also this new value is the function return value.

Currently the CRC calculation is used in the model only by the data link layer for validation of frames.

4.3 Communication Network Interface

This chapter describes the CNI (Communication Network Interface), its purpose in the protocol and how it is implemented in the C language model.

The CNI memory is divided in three major memory areas:

CNI Status Area Used to display the current state of the TTP/C controller to the host processor. Fields in this area are periodically updated by the controller and read by the host. This area is read-only for the host, i.e. in the model the host should never write in this area. Though this cannot be forbidden in the C language of course.

CNI Control Area Through this area the host processor can alter certain protocol parameters such as require a mode change or perform external clock synchronization. This area can be both read and written by the host.

CNI Message Area Contains all TTP/C messages that are sent or received by the SRU. The exact structure of this area is differs in different applications. Each message data field has an associated status field containing information whether the reception of the message was successful or not.

4.3.1 Implementation

The CNI is declared as a structure, which is declared in the file `fp1cni.h`:

```
typedef struct {
  /* ... the fields */
} CNI;
```

In the model the CNI is implemented in two levels:

The CNI The exact CNI structure as defined by the specification declared as a public part of the controller. It contains all of the above three areas.

The Local Status Holds a private copy of the real CNI status area for the purposes of the controller. This ensures correct behavior of the model when injecting faults into the real CNI.

The CNI implementation resides in one main module called `fp1cni` and in two submodules, namely `fp1cstat` which holds the C-state structure and `fp1lname` which holds the structure of SRU logical name.

In the text bellow for each CNI field its meaning in the real TTP/C controller and in the model and its declaration will be described. The stated declarations however must be always understood as member fields of the CNI structure, so for example:

```
unsigned ccf;
```

should be understood as:

```
typedef struct {
unsigned ccf;
... /* some other other fields */
} CNI;
```

4.3.2 CNI Update Rate

The CNI status area is updated once in every slot. This is achieved by copying the local status shadow CNI copy (described above) to the real CNI. The update is performed during the controller's post-receive phase (see *odkaz*). The status area is some times also updated beyond this schedule, this includes mostly updates before asynchronous interrupts etc. Exception to this rule is the field `cluster_time`, which is handled in a very special way (see 5.3.5).

The CNI control area is rarely being written to by the controller, but if so, the writing is performed immediately.

The CNI message area is updated during the post-receive phase every time a message is received or should be.

4.3.3 Concurrency Control Field

```
unsigned ccf;
```

The CCF (Concurrency Control Field) serves in the real TTP/C controller as a register for the NBW (Non-Blocking Write) protocol which assures that when writting data a collision between the controller and the host never remains undetected. This is never used in the model because it is discreet and thus the collision can never happen.

4.3.4 C-State

```
typedef struct {
unsigned time;
```

```
  /* C-state MEDL field */
  unsigned dmc;
unsigned mode;
unsigned medl_position;

unsigned membership[MAX_SRU_COUNT];
} C_STATE;

C_STATE c_state;
```

The C-state contains an essential part of the node state data. It consists of the current time, current position in the MEDL, the current mode of operation, any pending mode change and the current membership.

The `time` field contains the synchronized time in macroticks of the next action time (i.e. the start of the next transmission) and remains the same for the duration of a complette SRU slot. During the post-receive phase (see 4.7.3) this value is incremented by the duration of the current slot.

When nonzero, the `dmc` field contains the pointer to the MEDL mode control block entry (see 4.5.8) for the current mode. At the end of the transmission cycle, all the nodes change the current mode of operation to the one set in the respective position of mode control block.

The `mode` field simply contains the number of the current mode of operation. This field is used as a column index to the MEDL mode address entry table (see 4.5.10).

The `medl_position` field contains the current position in the MEDL, i.e. the number of the current slot in a cluster sysle. This field is used as a row index to the MEDL mode address entry table.

The `membership` is declared as an array and provides a constant view of activity of all SRUs in the cluster. One field (only a single bit is used) in the array is reserved for each SRU. The length of the membership vector depends on the `MAX_SRU_COUNT` constand which is currently defined to the value of 64.

4.3.5 Null-Frame Vector

```
unsigned null_frame[MAX_SRU_COUNT];
```

The null-frame vector is of the same size as the membership vector. Each field represents whether the corresponding SRU did transfer (a bus activity was detected on any channel) in its last own sending slot or not. All the bits in the null-frame vector are set to active at the power-up of the node.

4.3.6 Frame Diagnosis Field

```
typedef struct {
  FRAME_HEADER header_cha;
  unsigned eif_cha;
  unsigned mtd_cha;

  FRAME_HEADER header_chb;
  unsigned eif_chb;
  unsigned mtd_chb;
} CNI_FD;

CNI_FD frame_diagnosis;
```

The frame diagnosis field contains diagnostic information on the two frames received (or even transmitted) in the last SRU slot. There are three types of subfields for each channel (for channel A the name ends with ``_cha'', for channel B with ``_chb''):

Error Indication Field – `eif`

4-bit field, contains information on the error condition of the reception of the received frame. It recognizes null-frames, invalid frames, CRC errors and correct frames and whether there was noise. In the table 4.1 the values, that are possible in the model, are shown.

Error Condition	No Noise	Noise
Null-Frame	0000	n/a
Invalid Frame	n/a	0011
CRC Error	0100	n/a
Correct Frame	1000	n/a

Table 4.1: Error Codes

Header

4-bit field, contains a copy of the received frame header. The structure of the field is identical to the frame header and is described in section 4.6.1. This field is only valid when correct frame has been received.

Meassured Time Difference – `mtd`

8-bit field, contains the time difference between expected and actual arrival time of the frame. It is expressed in microticks. This field is only valid when correct frame has been received.

4.3.7 Protocol Status Field

```
typedef struct {
  unsigned accept_counter;
  unsigned failed_counter;
  unsigned invalid_counter;
  unsigned controller_ready;
  unsigned mfc;             /* membership failure count */
  unsigned i_frame_switch;
  unsigned csct;            /* clock state correction term */
  unsigned c_state_valid;
  unsigned protocol_state;
  unsigned bl;              /* bus guardian lifesign */

  /* error flags */
  unsigned be;  /* bus guardian error */
  unsigned ae;  /* acknowledgement error */
  unsigned se;  /* synchronization error */
  unsigned me;  /* membership error */
  unsigned mc;  /* MEDL CRC error */
  unsigned cb;  /* comunication system blackout */
  unsigned dc;  /* download comleted */
  unsigned so;  /* slot occupied error */
  unsigned mv;  /* mode violation error */
  unsigned ne;  /* NBW protocol error */
} CNI_PS;

CNI_PS protocol_status;
```

The protocol status field displays internal status information about the execution if the TTP/C protocol. All of its subfields are updated in the controller's post-receive phase (4.7.3).

Accept Counter

Contains the number of SRUs (excluding the SRU that owns this counter), that transmitted at least one correct frame (i.e. at least on one channel the transmission was successful). This counter is reset to zero at the pre-send phase of the controller (see 4.7.2).

Failed Counter

Contains the number of SRUs (excluding the SRU that owns this counter), that did not transmit a correct frame, but at least one frame of the two was valid (i.e. correctly received with wrong CRC). This counter is reset to zero at the pre-send phase of the controller (see 4.7.2).

Invalid Counter

Contains the number of SRUs (excluding the SRU that owns this counter), that did not transmit a correct frame or valid frame, but bus activity has been observed (e.g. a noise). This counter is reset to zero at the pre-send phase of the controller (see 4.7.2). All the three counters (accept, failed and invalid) are handled in a special way by the implicit acknowledgement algorithm (see 4.7.3).

Membership Failure Counter – `mfc`

The number of successive membership failures of the node.

I-Frame Switch

The current value of the I-frame switch. This controls the transmission of I-frames during the cold start period (see 4.8.5).

Clock State Correction Term – `csct`

This is the last computed value of the clock correction term, the calculated relative difference of the local clock to the clocks of the other nodes. It is shown in microticks.

C-State Valid

This field indicates whether the C-state field of the CNI contains a valid value, in other words whether the controller is synchronized with the rest of the cluster. Hereby a value of `C_STATE_VALID` indicates that the C-state is valid, whereas any other value indicates invalid C-state. This is used by the host in the startup of the cluster to know whether its controller is already synchronized or not.

Controller Ready

This field equals to CONTROLLER_READY once the controller's initialization has been finished, i.e. in the model all the memory has been allocated and initialized and the controller is ready to be turned on by the host.

Protocol State

Shows the current protocol state according to the table 4.2.

State Name	Predefined Constant	Value
Init	PS_INIT	1
Listen	PS_LISTEN	2
Cold Start	PS_COLDSTART	3
Startup	PS_STARTUP	4
Application	PS_APPLICATION	5
Ready	PS_READY	6
Passive	PS_PASSIVE	7
Await	PS_AWAIT	8
Selftest	PS_SELFTEST	9
Freeze	PS_FREEZE	10
Download	PS_DOWNLOAD	12

Table 4.2: Protocol States

Bus Guardian Lifesign – bl

This field indicates whether the bus guardian is currently open, i.e. whether access to the bus is possible. This is not used in the model of the protocol.

Protocol Errors

These flags are cleared in the init state and are set by the controller during its operation.

BE Bus Guardian Error. The bus guardian has detected an error.

AE Acknowledgement Error. The controller is in disagreement with the rest of the cluster.

SE Synchronization Error. The controller is desynchronized.

ME Membership Error. The controller has lost its membership many times (exceeded the maximum possible value set in the MEDL).

MC MEDL CRC Error. Not implemented in the model.

DC Download Completed. Not implemented in the model.

CB Communication System Blackout. No bus activity has been seen during the last TDMA round.

SO Slot Occupied. The controller's sending slot has been already occupied by another controller.

MV Mode Violation Error. The requested mode change is not permitted.

NE NBW Protocol Error. Not implemented in the model.

4.3.8 Personality Field

```
typedef struct {
  unsigned physical_sru_name;
  unsigned controller_version;
  unsigned app_id;
  unsigned app_version;
  LOGICAL_NAME logical_name;
} CNI_PERS;

CNI_PERS personality;
```

The personality field contains the SRU identification and version number of the controller. There are five of the fields but only the `logical_name` is currently used by the model.

Logical Name

```
typedef struct {
  unsigned ps;
  unsigned slot_position;
  unsigned multiplex_id;
} LOGICAL_NAME;
```

This is the name of the SRU that determines the role of the SRU.

`ps` – If this 1-bit field is set, the role of the SRU is intended to be passive, i.e. without a sending slot.

`slot_position` – This number corresponds to the position of the controller's sending slot within a TDMA round.

`multiplex_id` – Used to distinguish between multiplexed nodes that share one sending slot (determined by the `slot_position`) in a TDMA round. If the node is not intended to be multiplexed, this field is set to zero.

4.3.9 Controller Lifesign Field

```
unsigned controller_lifesign;
```

This field is used by the controller to provide periodic lifesign information to the host (see 4.10.1).

4.3.10 Assertion Code Field

This is not used by the protocol model at all. For description on this field see the protocol specification [SPEC].

4.3.11 Cluster Time Field

```
unsigned cluster_time;
```

This field contains the current global sinchronized time in units of macrotick. This field is handled in a very special way by the process oriented model of the protocol to ensure that it is always up to date (see 5.3.5).

4.3.12 Interrupt Status Field

```
typedef struct {
  unsigned be; /* BIST error */
  unsigned pe; /* protocol error */
  unsigned he; /* host error */
  unsigned ml; /* membership loss */
  unsigned mc; /* membership changed interrupt */
  unsigned mi; /* mode changed interrupt */
  unsigned ui1; /* user defined interrupt 1 */
  unsigned ui2; /* user defined interrupt 2 */
  unsigned ti1; /* time interrupt 1 */
  unsigned ti2; /* time interrupt 2 */
}  CNI_IS;

CNI_IS interrupt_status;
```

From the interrupt status field the host can gather information on the cause of the interrupt. The information is encoded in a vector of bits where each bit represents one interrupt condition. The flags are set by the controller upon an interrupt condition and are cleared by the host when it reads them. The read operation is thus also consideread as interrupt acknowledgement.

The host may select which interrupts it wants to be raised by setting the respective bit in the interrupt enable vector.

The following flags denote errors detected by the controller:

BE BIST (Built-In SelfTest) Error. Not implemented in the model.

PE Protocol Error. An error has been detected by the controller in the protocol control. The exact cause of this error Can be determined by reading the values in the protocol status field after this interrupt has been raised.

HE Host Error. This means that the controller has detected an error caused by the host, e.g. the host had not updated its lifesign properly.

ML The controller has lost its membership in the cluster.

The following flags denote interrupts during normal controller operation:

MC If this flag is set, the interrupt was trigerred by a change in the actual membership vector.

MI This flag is set if a mode change triggered the interrupt.

UI1 This flag is set if the interrupt was triggered due to a set user interrupt 1 bit in the slot control entry for the current slot.

UI2 This flag is set if the interrupt was triggered due to a set user interrupt 2 bit in the slot control entry for the current slot.

TI1 A proggrammable timer 1 triggered the interrupt if this flag is set.

TI2 A proggrammable timer 2 triggered the interrupt if this flag is set.

4.3.13 Built-In Selftest (BIST) Status Field

This field shows the reason for a built-in selftest failure. This is not implemented in the model.

4.3.14 Host Lifesign Field

```
unsigned hist_lifesign;
```

This field is the first of the control fields. It is used by the host to provide periodic lifesign information to the controller. A detailed description of the lifesign mechanism is described below (see 4.10.1).

4.3.15 External Rate Correction Term

```
unsigned external_rate_correction;
```

The external rate correction term provides the application with a possibility to alter the clock. Using this field, external clock synchronization algorithm can be performed. When the controller calculates its clock state correction term, this field is added to it.

4.3.16 Controller On Fild

```
unsigned co;
```

This field is used to turn the controller on and off. During normal operation this field must be set to a value of `0xFFFF`. Upon error detected by the controller, this field is reset and the controller transits to the freeze state. When the host detects an error it may also reset this field to shut the controller down.

The host is responsible for turning the controller on. So host is the one who must set this field to `0xFFFF` at startup.

4.3.17 Controller Await Field

```
unsigned ca;
```

This field forces the controller to transit to the await state. It is used by the download mechanism, which is not currently implemented in the model.

4.3.18 Mode Change Request Field

```
unsigned mode_change_request[2];
```

This is a 3-bit vector used by the host to initiate the mode change. The controller reads this field when assembling frames for transmission. If the mode change is permitted (i.e. it corresponds to the permission information written in the MEDL) the controller transmits the mode change request with the next frame. The point of time when a mode change is executed depends on its type (immediate or deferred).

The layout of these bits is identical to the one in the frame header.

4.3.19 Timer Fields

```
unsigned timer1;
unsigned timer2;
```

The timer mechanism allows the host to choose a specific point in time at which it wants and interrupt to be raised. For this purpose the host should write in the value of the desired global time. Whenever the global time (the `cluster_time` field in the CNI) reaches this value, the interrupt is raised. Two independent timers exist.

In the real world this mechanism is used by the special operating system run on the host called TTPos to schedule the tasks. The host tasks in the model can of course also be implemented this way.

4.3.20 Interrupt Enable Field

```
CNI_IS interrupt_enable;
```

This field has the same structure as the interrupt status field (see 4.3.12) and is used to select, which interrupts the host wants to be raised and which not. A masked-out (i.e. flag value of zero) interrupt will never be raised even if all other its conditions are fulfilled.

4.3.21 Built-In Selftest (BIST) Enable Field

Used to enable particular BIST mechanisms. This is not implemented in the model.

4.3.22 Timer Overflow Counter Field

```
unsigned time_overflow_counter
```

This field is incremented by one each time the global time (the `cluster_time` field in the CNI) overflows, i.e. its value changes from `0xFFFF` back to `0x0000`.

Upon initialization of the controller this field is set to `0x0000`. It can be preset by the host to any desired value at any time as needed by the application.

4.3.23 The Message Area

```
char cni_msg_base[CNI_MSG_AREA_SIZE];
```

This is the last of the three main CNI memory areas. It is declared as an array of characters but the exact layout is determined by the SRU's MEDL. The area is used for both host and the controller to store the data of the messages to be sent as well as the data of the received messages.

Entries are placed in this area according to MEDL table. Each entry consists of the message status field and its associated message data field. The structure of a message area entry is the following:

```
typedef struct {
  /* message status field */
  char not_used  /* unused 8 bits */
  char ccf:4;    /* concurrency control field */
  char eif:4;    /* error indication field */
  /* message data field */
  char data[MAX_APPLICATION_DATA_SIZE];
} CNI_MAE;
```

This structure is the only precisely defined data structure in the model. All the other structures do not follow (for speed and simplicity reasons) exactly the same physical structure as they have in the real hardware. This case is different and the reason is to allow the model to use the same MEDL tables as would an application use on the real cluster. In the MEDL tables there are the addresses to the CNI message area where the message data should be stored or from where it should be taken. If our structure was different from the one in hardware, the MEDL tables would have to be altered for the model in order to have the message data fit in the area.

The `ccf` field is used by the NBW (Non-Blocking Write) protocol. It is not implemented by the model and thus not used.

The `eif` field contains diagnostic information on the received message. The possible values are shown in the table 4.1. The information in this field is not only used to pass information from the controller to the host, but also the other way. If the host wants the respective data to be sent, it must assure that the eif field is set to "Correct Frame". Otherwise a host error is raised.

The data field is declared as array of characters with the size of 16 (or the constant `MAX_APPLICATION_DATA_SIZE`). This is the current maximum length of the data in the message. But the length of this array is not always equal to this value, it is just declared that way. Various entries in the CNI message area may have various data size depending on the application and ranging from a single byte to (currently) 16 bytes.

4.4 Interrupts

This chapter describes how the interrupts are implemented in the C language model. The interrupt is the only means of communication between the host and the controller that passes the CNI. It is raised by the controller and received by the host.

There are several conditions that may trigger an interrupt:

- A timer can be set to trigger the interrupt.
- A mode change can be signaled.
- Two independent user interrupts can be raised.
- An interrupt can be raised when a host error is detected.
- A change in the SRU membership can trigger an interrupt.
- A loss in controller's membership can raise an interrupt.
- An internal error in the TTP/C protocol may also raise an interrupt.
- Controller's Built-in Selftest may raise an interrupt (not implemented).

4.4.1 Model Interrupt Mechanism

In the model the interrupts are implemented by using a special callback function. At the program initialization by calling the following function:

```
typedef void (*PTR_CALLBACK)(void *);

void controller_attach_host(
  CONTROLLER *controller,
  void *host,
  PTR_CALLBACK interrupt_func);
```

The host gives the address of its interrupt handler to the controller (in the `interrupt_func` parameter). This handler function must be of the following prototype:

```
void name_of_the_function(void *controller);
```

When the controller raises an interrupt, it first sets the appropriate flag in the interrupt status field in the CNI and then calls this callback function.

As a parameter to this function the controller passes the pointer to itself, so in an application model several hosts may use the same callback function. Then the decision which controller raised the interrupt is done by looking at this parameter. For an example see 6.

4.4.2 Interrupt Types

Concerning the actual time, when the interrupt is triggered, three kinds of interrupts are recognized:

Asynchronous Interrupts These interrupts are raised independently on time and are not even restricted to the macrotick bound. They are mainly errors like host error and protocol errors.

Macrotick Synchronous Interrupts These interrupts always coincide with a full macrotick of a global time.

Transmission Phase Synchronous Interrupts These interrupts are raised only at the action time of the controller. These interrupts are mainly protocol-related like membership change interrupt, mode change interrupt, etc.

4.4.3 Asynchronous Interrupts

Host Error If the controller detects a state that, fulfills the host error criteria, it enters the passive state. The host must then continue updating the host lifesign to force the controller to reintegrate back to the cluster.

Protocol Error This interrupt is triggered, when the controller detects an internal error in the protocol control, e.g. it has lost its membership or synchronization error occurred. After raising this interrupt the controller transits to the freeze state.

BIST Error This is not used nor implemented in the model.

4.4.4 Macrotick Synchronous Interrupts

Timer Interrupt This interrupt is triggered upon a request from the host. The host writes a value of global time, at which it wants the interrupt to be raised, into the timer register. As soon as the global time reaches this value the interrupt is raised. There are two such independent timers.

4.4.5 Transmission Phase Synchronous Interrupts

Membership Loss This interrupt is triggered when the controller has lost its membership in the cluster.

User-Defined Interrupts Every slot in the time schedule may have assigned up to two user-defined interrupts. When the cluster reaches a slot that have at least one of the user interrupts enabled, the respective interrupt is raised.

Membership Change If any bit in the SRU membership vector changes, this interrupt is raised by the controller.

Mode Change This interrupt is raised at the point of time, where a mode change is performed.

4.5 Message Descriptor List

This chapter deals with the MEDL (Message Descriptor List), how it is used by the controller and its implementation in the C language model.

In the real world, the MEDL is implemented as flash ROM memory and thus it is read-only for the controller during normal operation. The MEDL table contains control information for the controller. There are all the information needed on the correct points of time for message reception and transmission, on the message lengths and structures and where the messages should be stored in the CNI message area. Furthermore the MEDL contains all the controller's configuration parameters. A rough structure of the MEDL memory can be seen in figure 4.1.

Whole MEDL memory is protected by CRC codes. Although room for these codes is present in the model data structure, they are never used. The MEDL is assumed by the model to be always correct and therefore cannot be subject to fault injection testing in current version.

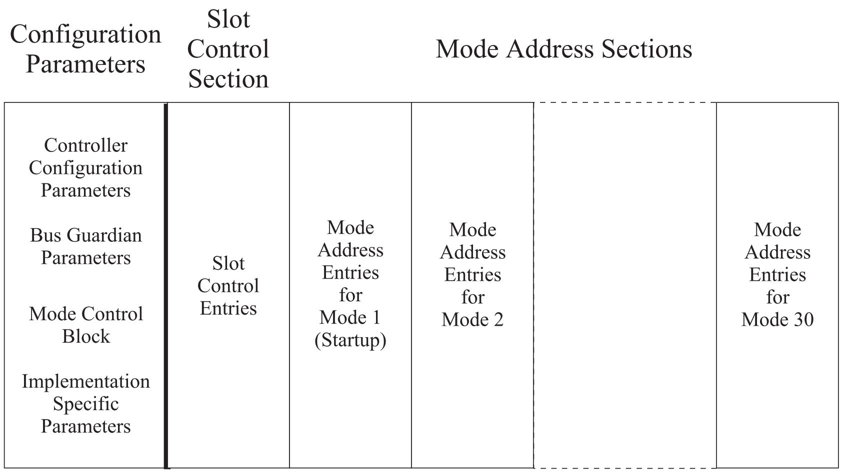

Figure 4.1: Structure of the MEDL

4.5.1 Implementation

The MEDL is implemented in the model as a C structure, which is declared in the file fp1medl.h:

```
typedef struct {
  /* ... the fields */
} MEDL;
```

All the fields in the structure use the same access scheme described in 4.2.3.

In the text bellow for each MEDL field its meaning in the real TTP/C controller and in the model and its declaration will be described. The stated declarations however must be always understood as member fields of the MEDL structure, so for example:

```
unsigned revision;
```

should be understood as:

```
typedef struct {
  unsigned revision;
  /* ... some other other fields */
} MEDL;
```

4.5.2 MEDL files

For the purpose of the model a simple file format for the storage of MEDLs was created. The file is text file and its format is trivial:

- Any line that begins with the character "`#`" is considered a comment.
- Empty lines are ignored.
- Any other line is considered a single medl parameter.

Medl parameters are given in the following form:

```
value name
```

The value is the number the parameter should be equal to and the name is the C language name of the parameter. For example a line like this:

```
749 bus_guardian_cold_start.bg_slot_offset
```

It says that the `slot_offset` parameter in the bus guardian cold start parameters is equal to 749 bus guardian macroticks.

The MEDL files can be read by the following function defined in the fp1mdlrd module:

```
int medl_read_from_file(MEDL *medl, unsigned index);
```

The function assumes that the memory for the MEDL structure has already been allocated. The name of the file is constructed from the second parameter of the function in the form `medlxx.txt`, where `xx` is double figure representation of the parameter, e.g. if `index` is 1, the file `medl01.txt` will be read.

4.5.3 Revision Number

```
unsigned revision;
```

This 16-bit field represents the MEDL revision number. In the real hardware this number is used to ensure that the MEDL is compatible with the actual controller. However in the model this is not needed and thus not used.

The revision number consist of major revision number and minor revision number. The model declares all the fields that conforms to the major revision number 0 and minor revision number 4.

4.5.4 TTP/C Controller Parameters

```
typedef struct {
  unsigned physical_sru_name;
  unsigned controller_version;
  unsigned app_id;
  unsigned app_version;
  unsigned cf;
  unsigned c_state_length;
  unsigned tdma_round_length;
  unsigned startup_timeout;
  unsigned free_running_macroticks;
  unsigned mmfc; /* max. membership failure count */
  double micro_to_macro;
  unsigned max_cold_start_entry;
  unsigned crc; /* not used */
} MEDL_CCP;

MEDL_CCP controller_cfg_params;
```

SRU Name – `physical_sru_name`

Physical name of the SRU. Not used by the model.

Controller Version Number – `controller_version`

Version number of the controller design. Not used by the model.

Application Identification – `ttapp_id`

Unique identification of the application. Not used by the model.

Application Version Number – `app_version`

Application version. Not used by the model.

Cold Start Flag – `cf`

If this 1-bit wide flag is set the SRU is allowed to enter the cold start state of the protocol, i.e. it is allowed to initiate the cluster communication by sending the first I-frame.

Normally all the nodes in the cluster should have this flag set. Only nodes that are specified for a multiplexed role are not allowed to enter cold start.

Free-Running Macroticks – `free_running_macroticks`

The clock synchronization algorithm works on the basis of prolonging or shortening of macroticks by single microticks. This field contains the total number of macroticks that have to remain unchanged before the clock synchronization algorithm changes the next macrotick length.

Maximum Cold Start Entry – `max_cold_start_entry`

If this field is nonzero, it specifies the maximum number of cold start state entries, i.e. the maximum number of attempts to send an initial I-frame. If this field is zero it is ignored.

Maximum Membership Failure Count – `mmfc`

If this field is nonzero, it specifies the maximum number of successive membership losses before the controller transits to the freeze state. If this field is zero it is ignored.

TDMA Round Length – `tdma_round_length`

This field contains the total length of one TDMA round in units of macrotick. Because the layout of the TDMA round never changes in one application, this is an application-wide constant.

Startup Timeout – `startup_timeout`

This field holds the length of the startup timeout in macroticks. The startup timeout is used by the reconfiguration protocol layer and is equal to the total duration of the slots prior to the controller's own in the TDMA round.

C-State Length – `c_state_length`

This field contains the length of the C-state in words. The C-state can by from 3 to 6 words long. This applies the length of the membership vector which is present in the C-state.

Microtick to Macrotick – `micro_to_macro`

This field contains the number of microticks in a macrotick. Note that this need not to be integer number. In the hardware MEDL version this field is divided in two subfields (the integral and the fraction part), because the

TTP/C controller uses fixed point arithmetic. In the model it is ordinary floating-point double.

4.5.5 Bus Guardian Parameters

The bus guardian is an independent device which protects the bus from being occupied by a faulty constantly transmitting node (a "babbling idiot"). It periodically enables and disables access to the bus according to a programmable parameters and so-called arm signal from the controller. For details see section 4.9.

There are in sum five complete sets of the bus guardian parameters in the MEDL, each of them is used in a different case. They are all represented by a common data structure type:

```
typedef struct {
  unsigned bg_slot_offset;
  unsigned bg_slot_duration;
  unsigned bg_remaining_round;
  unsigned crc; /* not used */
} MEDL_BGP;
```

`bg_slot_offset` – The interval between the arm signal and the time when the bus access should be enabled.

`bg_slot_duration` – The interval for how long the bus access remains enabled.

`bg_remaining_round` – The rest of the time, the bus access is disabled during this interval. After it the bus guardian is ready for another arm signal.

All of the three above parameters are represented in units of $\frac{1}{16}$ of the clock frequency of the bus guardian. This may seem strange but the reason is simple. In the current hardware the bus guardian operates at 16MHz while the controller's macrotick rate is 1MHz. Knowing this it is clear that the units of the bus guardian parameters corresponds to the macroticks of the controller.

Bus Guardian Cold Start Parameters

```
MEDL_BGP bus_guardian_cold_start;
```

These are the time values for the bus guardian used when the controller enters the cold start state. For correct setting consult the protocol specification [SPEC].

Bus Guardian Download Parameters

```
MEDL_BGP bus_guardian_download;
```

These are the time values for the bus guardian used during the download procedure. Not used by the model.

4.5.6 Role List

```
typedef struct {
  LOGICAL_NAME logical_name;
  MEDL_BGP bus_guardian;
  unsigned crc; /* not used */
} MEDL_RL;

MEDL_RL role_list;
```

The role list is meant to allow a node to play several roles in the cluster, e.g. to create a back-up node that can replace other two nodes which need not to be exactly the same. However so far (as of version 0.1 of the protocol) only one role for one node is possible and that is the reason why the role list is not declared as an array but as a single field instead.

A role contains the SRU's logical name and a set of bus guardian parameters. For description of the `LOGICAL_NAME` structure see section 4.3.8. The bus guardian parameters are the ones used during the normal controller operation, i.e. when it is synchronized and sends frames according to the MEDL table. For correct setting consult the protocol specification [SPEC].

4.5.7 Implementation Specific Parameters

```
typedef struct {
  unsigned entries;
  unsigned rom_speed;
  unsigned i_frame_time;
  unsigned ifg_time;
  unsigned listen_delay;
  unsigned watchdog_slot_length;
  unsigned listen_timeout_extension;
  unsigned listen_timeout_restart_time;
  unsigned oversampling_rate;
  unsigned sync_interval;
  unsigned crc; /* not used */
```

```
  MEDL_BGP synchronization_period_params;
  MEDL_BGP listen_cold_start_period_params;
} MEDL_ISP;

MEDL_ISP implementation_params;
```

The implementation specific parameters structure contains a couple of hardware-related constants that are mostly not used by the model and two additional sets of bus guardian parameters.

IFG Time – `ifg_time`

This is the only field from the implementation specific configuration parameters (excluding the bus guardian parameters) that is used by the model. It contains the duration of the IFG (Interframe Gap), the time interval, where most of the controller's code is executed (the post-receive and pre-send phase). The value is in macroticks and is used in the model for correct slot timing.

Bus Guardian Synchronization Period Parameters

These are the time values for the bus guardian used when the controller is about to synchronize into the cluster. They must be set so to allow the controller to transmit almost immediately after the arm signal. For correct setting consult the protocol specification [SPEC].

Bus Guardian Listen-Cold Start Period Parameters

These are the time values for the bus guardian used when a cold-start-allowed controller enters the listen protocol state and begins its listen timeout. For correct setting consult the protocol specification [SPEC].

4.5.8 Mode Control Block

```
typedef struct {
  unsigned deferred_mode[3];
  unsigned crc; /* not used */
} MEDL_MCE;

MEDL_MCE mode_control_entry[MAX_MODE_COUNT];
```

To understand the mode control block, please be aware of the fact, that from each operational mode a deferred mode change can be performed only to at most another three modes (one at a time of course). This limitation is due to the lack of space in the frame header, which is used to distribute the mode change request. The numbers of these three modes must be predefined during the application design process and cannot be changed later at runtime.

The user defines a kind of state machine with predefined transitions from mode to mode and the MEDL mode control block then contains the transition table for it. For each possible mode of operation it contains three another mode numbers.

4.5.9 Slot Control Entries

```
typedef struct {
  unsigned sru_slot_duration;
  unsigned mo;
  unsigned membership_pos;
  unsigned medl_pos;
  unsigned mm;
  unsigned eor;
  unsigned ra;
  unsigned eot;
  unsigned cs;
  unsigned syf;
  unsigned ba;
  unsigned delay_correction;
  LOGICAL_NAME logical_name;
} MEDL_SCE;

MEDL_SCE slot_control_entry[MAX_TRANSMISSION_CYCLE_LENGTH];
```

For each slot in the cluster cycle there is one slot control entry. The slot control entry defines various parameters for the respective slot, which are invariant no matter what operational mode is currently active.

SRU Slot Duration – `sru_slot_duration`

This field contains the duration of the slot in macroticks. Note that this value is a cluster-wide constant independent on the current operational mode.

Multiplexed SRU with Own Membership – `mo`

This flag is set in case the slot belongs to a multiplexed node, which is assigned its own bit in the membership vector.

Bit Position in Membership Vector – `membership_pos`

This field contains the bit position of the sender in the membership vector.

MEDL Position – `medl_pos`

This field contains current MEDL position, which is actually same as the number of this slot control entry (i.e. same as the index to the array). This field is probably used in the hardware to simplify the microcode, it is useless for the model.

Multiplexed SRU with Multiplexed Membership – `mm`

This flag is set in case the slot belongs to a multiplexed node, which shares the bit in the membership vector with the others.

End Of Round – `eor`

This flag is set if this is the last slot in the TDMA round. This field is probably used in the hardware to simplify the microcode, it is useless for the model.

End Of Transmission Cycle – `eot`

This flag is set if this is the last slot in the transmission cycle, i.e. it is the last slot control entry in the MEDL. Unlike the eor field this one is used in the model, since it is the only way to determine the end of the transmission cycle and thus the MEDL position should be reset to zero.

Clock Synchronization – `cs`

If this flag is set, the clock synchronization algorithm is executed at the post receive phase in the slot (see 4.7.3).

Synchronization Frame – `syf`

If this flag is set the frames that arrive in the slot are used by the internal clock synchronization algorithm to correct local clock. If this flag is not set the frames are ignored by the internal clock synchronization algorithm. This is useful when having a cheap node with a pure inaccurate clock to prevent it from affecting the clocks of other nodes.

Bus Guardian Arm – `ba`

This flag is set when the controller should send the arm signal to the bus guardian. The arm signal is then set at the action time in the slot (i.e. at the point of time where the transmission starts).

Delay Correction – `delay_correction`

This field contains the expected propagation delay of the frames that are sent in the slot. It helps the internal clock synchronization algorithm to operate more accurately. This is not currently implemented in the model.

Logical Sender Name – `logical_name`

This is the identification of the sender. Note that the sender is permanently defined for each slot in the transmission cycle and cannot be changed ad runtime.

4.5.10 Mode Address Entries

```
typedef struct {
  unsigned ru1;
  unsigned ia;
  unsigned cni_address_cha;
  unsigned ru2;
  unsigned ib;
  unsigned cni_address_chb;

  /* mode change permission */
  unsigned mcp_i[3];
  unsigned mcp_d[3];
  unsigned mcp_c;

  unsigned eoc;
```

```
  unsigned dafla;
  unsigned daflb;
  unsigned crc; /* not used */
} MEDL_MAE;

MEDL_MAE mode_address_entry
  [MAX_MODE_COUNT+1]
  [MAX_TRANSMISSION_CYCLE_LENGTH];
```

For each slot in the cluster cycle and each operation mode there is one mode address entry. The mode address entry defines various parameters for the respective slot in the respective mode, which differ in various operation modes.

Raise User Interrupt – `ru1, ru2`

If one of these flags is set, the controller raises the appropriate user interrupt.

I-Frame on Channel – `ia, ib`

These flags indicate for both channels (A and B), whether I-frame or N-frame should be sent (or received) in the slot. A value of 1 means I-frame, a value of 0 means N-frame. Currently the hardware controller does not support sending different types of frames on different channels at the same time, it is not the case of the model.

CNI Address on Channel – `cni_address_cha, cni_address_chb`

These 14-bit fields represent for each channel (A and B) the word address offset to the CNI message area for the data of the N-Frame sent in the slot. I-Frames never contain any application data and therefore these fields are ignored for I-frames.

Immediate Mode Change Permission – `mcp_i[3]`

Each entry in this array field contains a flag that tells whether the sender in the slot is allowed to request the respective immediate mode change or not.

Deferred Mode Change Permission – `mcp_d[3]`

Each entry in this array field contains a flag that tells whether the sender in the slot is allowed to request the respective deferred mode change or not.

Mode Change Cancel Permission – `mcp_c`

This flag tells whether the sender in the slot is allowed to cancel a pending deferred mode change.

End Of Cluster Cycle – `eoc`

This flag is set if this is the last slot in the cluster cycle. This field is probably used in the hardware to simplify the microcode, it is useless for the model.

Data Field Length on Channel – `dafla, daflb`

These fields represent the length in bytes of the data of the N-frame on the respective channel. The values are less 1, i.e. value of 0 means data size of 1 while value of 15 means data size of 16.

4.6 Data Link Layer

The data link layer is implemented by two modules:

- `fp1frame` – Contains all the data structures and functions required for frames and their management.
- `fp1chan` – Contains the implementation of a channels, through which frames can be sent.

4.6.1 Frames

All data structures concerning frames are placed in the module `fp1frame`.

Frame Header

```
typedef struct {
  unsigned i_n_frame;
  unsigned mode_change[3];
} FRAME_HEADER;
```

The `i_n_frame` field identifies whether the frame is I-frame (value of 1) or N-frame (value of 0). The `mode_change` array has 1-bit wide entries and informs about a panding mode change or initiates a new one or cancels it. It is interpreted as shown in the table 4.3.

Following functions are associated with the `FRAME_HEADER` structure:

Bit 0	Bit 1	Bit 2	Mode Change
0	0	0	none
0	1	0	deferred 1
0	0	1	deferred 2
0	1	1	deferred 3
1	0	0	clear pending
1	1	0	immediate 1
1	0	1	immediate 2
1	1	1	immediate 3

Table 4.3: Meaning of Mode Bits in Frame Header

```
void frame_header_chain_crc(
  FRAME_HEADER *frame_header,
  unsigned *crc,
  unsigned polynomial);
```

Chains the frame header to the CRC calculation.

```
unsigned frame_header_get_dmc(FRAME_HEADER *frame_header);
```

Returns number of required deferred mode change, i.e. translates the mode change header bits according to the table 4.3 to the deferred mode change number. If no deferred mode change is requested, the return value is 0.

```
void frame_header_set_dmc(
  FRAME_HEADER *frame_header,
  unsigned mode);
```

Sets the mode change header bits to request the deferred mode change given by the mode parameter.

```
unsigned frame_header_get_imc(FRAME_HEADER *frame_header);
```

Returns number of required immediate mode change, i.e. translates the mode change header bits according to the table 4.3 to the immediate mode change number. If no immediate mode change is requested, the return value is 0.

```
void frame_header_set_imc(
  FRAME_HEADER *frame_header,
  unsigned mode);
```

Sets the mode change header bits to request the immediate mode change given by the mode parameter.

I-Frame

```
typedef struct {
  FRAME_HEADER header;
  C_STATE c_state;
  unsigned crc;
  unsigned c_state_length; /* model-specific field */
} I_FRAME;
```

I-frames contain the header, the C-state and of course the CRC code. The `C_STATE` structure is explained in section 4.3.4. Because it must be known to the model what is the length of the C-state in the frame, the `c_state_length` field is also declared although not physically present in the frame.

Following functions are associated with the `I_FRAME` structure:

```
unsigned i_frame_get_length(I_FRAME *i_frame);
```

Returns physical I-frame length in bits. That value is then used to determine the duration of the its transmission.

```
unsigned i_frame_calculate_crc(I_FRAME *frame);
```

Calculates the CRC code for the I-frame.

N-Frame

```
typedef struct {
  FRAME_HEADER header;
  unsigned char application_data[MAX_APPLICATION_DATA_SIZE];
  unsigned crc;
  unsigned application_data_length; /* model-specific field */
} N_FRAME;
```

I-frames contain the header, the application data and the CRC code. Because it must be known to the model what is the length of the application data stored in the frame, the `application_data_length` field is also declared although not physically present in the frame.

Following functions are associated with the `N_FRAME` structure:

```
unsigned n_frame_get_length(N_FRAME *n_frame);
```

Returns physical N-frame length in bits. That value is then used to determine the duration of the its transmission.

```
unsigned n_frame_calculate_crc(
  N_FRAME *frame,
  C_STATE *c_state,
  unsigned c_state_length);
```

Calculates the CRC code for the N-frame. The CRC code for N-frames is never calculated only over the actual frame but always with an appended C-state. This mechanism is used for the implicit acknowledgement algorithm (see 4.7.3). This function has therefore two additional parameters denoting the C-state to be appended and its length.

Common Frame Storage

```
typedef union {
  N_FRAME n;
  I_FRAME i;
} FRAME_UNION;

typedef struct {
  FRAME_TYPES type;
  FRAME_STATES state;
  double time_of_delivery;
  unsigned noise;
  FRAME_UNION frame;
} FRAME;
```

The FRAME structure is intended to be a common storage type for either I-frames or N-frames. The `FRAME_UNION` structure works for that purpose. This structure is being passed to the channels to transmit them. The fields are:

`type` – Denotes the type of the frame, can be equal either to `FRAME_TYPE_I` or `FRAME_TYPE_N`. It may seem that this is redundant due to the fact that the same information is also present in the frame header, but the frame header can be subject to a fault injection testing and if it gets corrupted, this field insures that the model still knows, what the type of the frame originally was.

`state` – State of the frame. The values for this field and their meaning are shown in the table 4.4.

`frame` – The actual frame, either N-frame or I-frame according to the type parameter.

Value	Meaning
`FRAME_STATE_NULL`	No frame is present in the structure.
`FRAME_STATE_RECEIVING`	Something is in the structure but it is not a complete frame. The type field contains a correct value but the frame field not.
`FRAME_STATE_COMPLETE`	A complete frame is in the structure.

Table 4.4: Frame States

`noise` – Denotes whether there was noise during the transmission.

`time_of_delivery` – This is the time, when the frame transmission started. It has no meaning for the functional protocol specification, only it is used by the C-Sim model. It will be described later.

Following functions are associated with the FRAME structure:

```
unsigned frame_get_length(FRAME *frame);
```

Returns physical frame size in bits. That value is then used to determine the duration of the its transmission.

```
unsigned frame_is_valid_i_frame(FRAME *frame);
```

Returns 1, if frame is complete and it is an I-frame with valid CRC.

```
unsigned frame_is_valid_n_frame(
  FRAME *frame,
  C_STATE *c_state,
  unsigned c_state_length);
```

Returns 1, if frame is complete and it is an N-frame with valid CRC.

```
unsigned frame_is_valid(
  FRAME *frame,
  C_STATE *c_state,
  unsigned c_state_length);
```

Returns 1, if frame is complete, has valid CRC and its C-state corresponds to the given one.

4.6.2 Channel

Channels are implemented by the module `fp1chan`. The data structure for the channel is declared as follows:

```
typedef void (*PTR_INT_CALLBACK)(void *, unsigned);
typedef void (*PTR_CALLBACK)(void *);

typedef struct channel {
  unsigned bus_id;
  int active;
  FRAME frame;

  void *sender;

  unsigned receiver_count;
  void *receiver[MAX_SRU_COUNT];

  PTR_CALLBACK send_callback;
  PTR_INT_CALLBACK receive_callback[MAX_SRU_COUNT];
  PTR_CALLBACK bg_error_callback;

  FUNC_CHANNEL_SEND_FRAME channel_send_frame;
} CHANNEL;
```

The fields and their meaning:

- `bus_id` – The ID of the channel, i.e. 0 or 1.
- `active` – This field is nonzero if the channel is active, i.e. if a frame is just being transmitted.
- `frame` – The frame that is being transmitted throught the channel.
- `sender` – Pointer to the controller that sent the frame.
- `receiver_count, receiver[]` – The list of controllers that have registered themselves as receivers, i.e. frame would be passed to all of them once the transmission is complete.
- `send_callback, receive_callback[], bg_error_callback` – These are pointers to callback functions. They are called upon an event, i.e. `send_callback` is called once the frame is completely transmitted, `receive_callback` is called when a frame is received and the third one, `bg_error_callback`, is called when the sender's bus guardian refused access to the bus.

`channel_send_frame` – Pointer to a send function. This function is called by a controller that wants to send a frame.

Other function associated with channels:

```
int channel_receive(
  CHANNEL *channel,
  void *controller,
  PTR_INT_CALLBACK callback);
```

Using this function, controllers register themselves as listeners on the channel. When nonzero, the parameter callback points to a function that will be called once a frame is received.

```
void channel_cancel_receive(
  CHANNEL *channel,
  void *controller);
```

Using this function, controllers inform the channel that they are no longer listening for frames.

```
void channel_set_bg_error_callback(
  CHANNEL *channel,
  PTR_CALLBACK callback);
```

This function assigns a callback to the bus guardian error event.

4.7 SRU Layer

This chapter describes the model implementation of protocol actions, that are being done periodically, when the SRU (often reffered here as controller) is synchronized with the rest of the cluster.

4.7.1 Implementation

The controller is implemented by the `fp1ctrl` module. The data structure for the controller is declared as follows:

```
typedef struct controller {
  CNI *lcni;
  CNI cni;
  LOCAL_STATUS lstatus;
```

```
  MEDL medl;
  LOCAL_DATA ldata;

  BUS_GUARDIAN *bus_guardian;
  CHANNEL *channel[2];

  FUNC_CONTROLLER_UPDATE_CLUSTER_TIME update_cluster_time;
  PTR_INT_CALLBACK receive_callback;
  PTR_CALLBACK send_callback;
} CONTROLLER;
```

The fields and their meaning:

`cni` – The controller's communication network interface (see 4.3).

`lstatus` – Internal private copy of the status area of the CNI.

`lcni` – This field always points to the lstatus.

`medl` – The controller's message descriptor list.

`ldata` – The controller's local data. These are the data, that are not explicitly defined by the TTP/C protocol specification, but they showed themselves necessary for the model implementation. These data fields can be compared to the hardware chip's internal registers used by the microcode.

`bus_guardian` – Pointer to the bus guardian (see 4.9).

`channel[2]` – Two pointers to the channels the controller is attached to (see 4.6.2).

`update_cluster_time` – Pointer to function that can perform some actions when the cluster time changes. It is called automatically by the code. This function is used heavily by the C-Sim model.

`receive_callback` – Callback function called by the channel when frame is received.

`send_callback` – Callback function called by the channel when frame transmission is completed.

In the model all controller's actions during the slot are being done at one or two single points of time depending on whether the controller should or should not send in the slot. These two points of time include:

Pre-Send Phase All the actions that have to be performed before the action time of the slot in which the controller is the sender.

Post-Receive Phase All the actions that have to be performed in every slot after the transmission phase.

4.7.2 Pre-Send Phase

The pre-send phase is done for the controller by calling the following function passing to it the pointer to the controller:

```
unsigned psp(CONTROLLER *controller);
```

The function returns nonzero value if the controller should send frames in current slot and zero if not.

Controller's Own Slot

At first the psp function checks whether the slot part of the controller's current logical name, which is present in the cni.personality field, is equal to the current logical sender name, which is present in the current MEDL slot control entry. This check is performed using the function:

```
unsigned is_my_slot(CONTROLLER *controller);
```

Another useful function used from inside of `is_my_slot` is the function

```
MEDL_SCE *get_current_sce(CONTROLLER *controller);
```

It returns pointer to the current MEDL slot control entry. It does this by looking at the C-state `medl_position` field.

If the check succeeds, three additional tasks are performed. Firstly the check whether the value of the accept counter is greater or equal to the sum of the value of the failed counter and the value of the invalid counter. If not, the controller transits to freeze state and the `psp()` function returns zero.

Secondly the check is whether the sum of all the three counters is greater than zero, i.e. there has been an activity in the TDMA round other than the controller's own. If not, the controller transits to the freeze state.

Thirdly if both of the above checks succeeded, all the three counters are reset and the psp function returns zero.

Current Sender

The `psp()` function then checks whether the current logical name of the controller is equal to the current logical sender name at the current MEDL slot control entry. If not, the function returns zero, i.e. the controller is not the sender in current slot.

Otherwise it tests the host lifesign field to know whether the host is alive. It is done with the call to the function:

```
unsigned test_host_lifesign(CONTROLLER *controller);
```

According to the correct lifesign update mechanism, the controller's lifesign is updated as a secondary effect of the function (see 4.10.1). If the host is not alive, the controller transits to passive state and the `psp()` function returns zero.

Then if the controller already is in passive state, it transits to ready state due to the fact that the host is alive.

Now if the controller is in ready state, it may only send frames when reconfiguration is allowed. The check whether it is allowed is performed by the function:

```
unsigned is_reconfiguration_allowed(CONTROLLER *controller);
```

Following conditions must be fulfilled:

- The RA bit in the MEDL current slot control entry must be set.
- The controller's own bit in the null frame vector must be set.
- The controller's own bit in the membership vector must be zero. If it is set and the two above conditions are fulfilled, slot occupied error is raised and the controller transits to passive state.

Now if the controller is in ready state, it transits to startup state or application state according to the current operational mode in the C-state.

Finally the frames to be sent are prepared in a special buffer in the controller's local data (`ldata.frame[]`) using a call to the function:

```
void controller_prepare_frame(CONTROLLER *controller);
```

Then the return value of the `psp()` function is one.

4.7.3 Post-Receive Phase

The post-receive phase is done for the controller by calling the following function passing to it the pointer to the controller:

```
void prp(CONTROLLER *controller);
```

Initial Actions

At first the `prp()` function cancels any pending listening or any unfinished frame reception. This is needed, because the post-receive phase is always done at the beginning of the IFG (Inter-Frame Gap) and thus after the end of the transmission phase. Therefore all transmissions and receptions must be already completed and any unfinished frames are erroneous anyway.

Next is the check whether the controller resides in the freeze protocol state. If this is the case, the function exits with no further actions since it is nonsensical to do anything.

Implicit Acknowledgement

The implicit acknowledgement mechanism is based on the fact that the controller's C-state is present, either physically or virtually, in all frames. Each node can therefore compare its own C-state with the one of the current sender. Since the C-state holds the membership vector and therefore a complete membership information, the node has an instant knowledge of the current sender's view of the cluster membership. The algorithm is then run through three steps (Please note that although the algorithm is fully implemented in the model, its description here is little simplified, for detailed description consult the TTP/C protocol specification **??**):

- When the node (node A) is the current sender, it assumes that the frame it sent was correct.
- In the next slot A compares its own C-state with the one received in the slot (from node B). If they are consistent, A's frame is positively acknowledged and the algorithm is finished. If they are inconsistent, another check is performed. A compares the received C-state with its own one modified in the way that B's membership bit is reset. If these two are consistent, A assumes, that the frame it just received was faulty, and clears B's membership bit in its own C-state, i.e. it does not yet admit that it failed, instead it presumes that B failed. If both of the checks fail, they are repeated in the next slot.

- The frame (from node C) in the next slot arbitrates. A compares its C-state with the one of node C. Note that now A has B's membership bit reset. If they are consistent A knows that its frame is positively acknowledged and that B had failed and the algorithm is finished. If they are inconsistent, in the next check A compares its C-state with its own membership bit reset B's membership set with C's C-state. If they are consistent, A's frame is negatively acknowledged, it admits that it failed and lost its membership. If the second check fails as well, a so-called double failure occured and A clears both B's and C's membership bit in its C-state.

In the model the implicit acknowledgement uses a variable in controller's local data (`ldata.iack_state`) to store the state of the algorithm. Table 4.5 shows possible values for this variable.

State	**Meaning**
IACK_OK	The node's frame has been positively acknowledged and no further checks are needed.
IACK_SEND	Then node have just send a frame.
IACK_CHECK1	The node is about to check it's successor's C-state.
IACK_CHECK2	The node's successor claimed that the node had failed. Another frame needs to be chacked.
IACK_ERROR	The node's frame has been negatively acknowledged.
IACK_DOUBLE_FAILURE	A double failure occured.

Table 4.5: Implicit Acknowledgement Algorithm States

The algorithm is performed by the function with an easy-to-remember name:

```
unsigned perform_iack(CONTROLLER *controller);
```

This function at first calls another one named `check_iack()`, which determines the state, which the algorithm should be switched to. It does this by looking at the current state and by performing the checks described above. The `check_iack` function is called for each channel, i.e. twice.

After this the `perform_iack()` function does specific actions according to the executed state transition (figure 4.2).

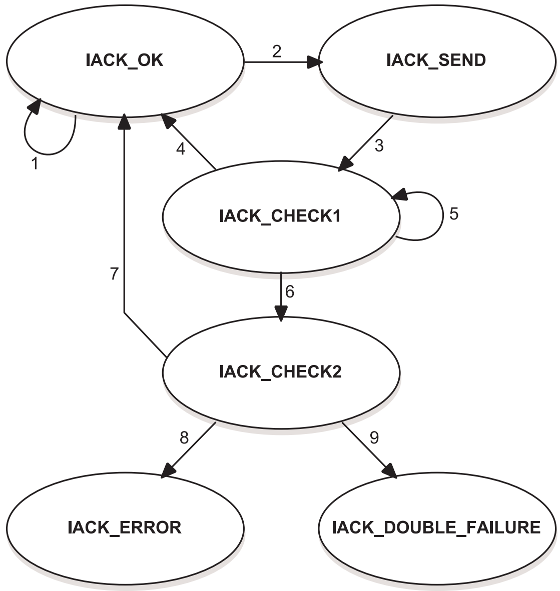

Figure 4.2: Implicit Acknowledgement Diagram

Legend to the figure 4.2:

1. No further actions.

2. No further actions.

3. No further actions.

4. Accept counter is incremented by 1.

5. If bus activity has been observed, the sender's membership bit is cleared and failed counter is incremented by 1.

6. No further actions.

7. Both accept and failed counter are incremented by 1 and the successor's membership bit is cleared.

8. Accept counter is incremented by 2. The node's membership bit is cleared and the membership bits of both first and second successor are set. The controller transits to ready protocol state.

9. Failed counter is incremented by 1. If bus activity has been observed it is incremented by another 1. The membership bits of both first and second successor are cleared. The controller transits to ready protocol state.

Frame Recognition

After calling the `perform_iack` function, the `prp()` function check the frames on both channels to determine the state of the received frames to fill in the error indication fields. There are several states that can be recognized:

- `EIF_NULL_FRAME` – No bus activity has been observed on the channel.
- `EIF_INVALID_FRAME` – Bus activity has been observed but no frame is recognized.
- `EIF_CRC_ERROR` – Frame has been received but has failed the CRC check.
- `EIF_CORRECT_FRAME` – Frame has been received and passed the CRC check. In case of N-frame, the application data bytes are copied to the appropriate place in the CNI message area.

Clock Synchronization

Having the frames recognized the `prp()` function then performs the clock synchronization algorithm by calling the function:

```
void perform_clock_synchronization(CONTROLLER *controller);
```

For each slot, that has the syf bit in the MEDL mode address entry set, the time differences between the expected and actual frame arrival are measured on both channels. The average of the differences is calculated in microticks and stored in a special stack. This stack is present in the controller's local data (`ldata.sync_stack`) and can hold up to four such values. Upon controller's initialization all values this stack are reset to zero.

When the cs bit in the MEDL mode address entry for the current slot is set, the clock is about to be resynchronized. This is done by reading the four values from the synchronization stack. From these four values the greatest and the lowest one are discarded. The mean value of the remaining two values incremented by external clock correction term is the resulting clock state correction term. It is the value in microticks, that is considered as the difference between the controller's local clock and the rest of the cluster. It is stored in the appropriate entry of the protocol status field of the CNI.

If the calculated clock state correction term is not within a bound interval (which is `CLOCK_PRECISION/2`, see constants 4.2.2), a synchronization error is raised and the controller transits to the freeze protocol state.

Finalization

At the end the `prp()` function shifts to the next slot by a call to the function:

```
void set_next_slot(CONTROLLER *controller);
```

This function is responsible for the C-state update. It increments the time and MEDL position fields (and wraps around if neccessary).

The last action is raising user interrupts if desired in the next slot (one of the `ui1` or `ui2` bits in the MEDL mode address entry is set). The interrupts are not raised immediately, they are just planned.

4.8 Reconfiguration Layer – RC Layer

The reconfiguration layer contains the mechanisms related to the startup of a cluster, the dynamic allocation of SRU slots ant the reset of the controller.

4.8.1 Timeouts

Several correctly set up timeouts are mandatory for correct controller and cluster operation. Wrongly configured timeouts may render the controller inoperable or may even prevent the whole cluster from starting up.

Startup Timeout This timeout is specific to each controller in the cluster. Its value is equal to the duration of all TDMA slots prior to the controller's one. Value of this timeout is present in the MEDL in macroticks.

Cold Start Timeout Value of this timeout is equal to the sum of the startup timeout and the duration of a single TDMA round. The cold start timeout represents the time the controller will wait for an answer after sending the first cold start I-frame. It is different for each controller to prevent collisions on the bus.

Listen Timeout Value of this timeout is equal to the sum of the startup timeout and the duration of two TDMA rounds. The listen timeout represents the time the controller will listening on the bus before starting the cold start. It is different for each controller to prevent collisions on the bus. This choice of the listen timeout also ensures that the longest cold start timeout is always shorter than the shortest listen timeout.

4.8.2 Controller States

During the startup of the cluster and also in case of error like membership loss the controller transits through several protocol states. These are namely:

Init Initialization of the controller, memory cleanup, etc.

Listen In this state the controller is not yet synchronized with the rest of the cluster and waits for I-frames, which would allow it to synchronize.

Cold Start In cold start state the controller becomes sychronized. It does this by setting cluster time to zero and sending the first I-frame. Other controllers (that should be in the listen state) will then accept it.

Startup In startup the controller is already synchronized and waits for demand to switch to an application mode.

Application This state is very similar to the startup, only application is used for modes 2 to 30, i.e. all except the startup mode.

Ready In ready the controller is synchronized but does not have yet acquired cluster membership. Therefore it does not send.

Passive In this state the controller is synchronized, but its host is faulty and does not update its lifesign. No frames are sent.

Freeze The controller is shut off in this state and waits for external (host) intervention.

On the figure 4.3 you can see the graph with the possible transitions between the states. The functional implementation of these states is described in detail below.

Other states exist in the protocol specification, though they are not implemented in the model since it does not require them. These are namely Await, Selftest and Download. For details please consult the specification [SPEC].

4.8.3 Init State

The controller enters the init state as soon as it is switched on via the CO field. It is implemented by the function:

```
void controller_init_state(CONTROLLER *controller);
```

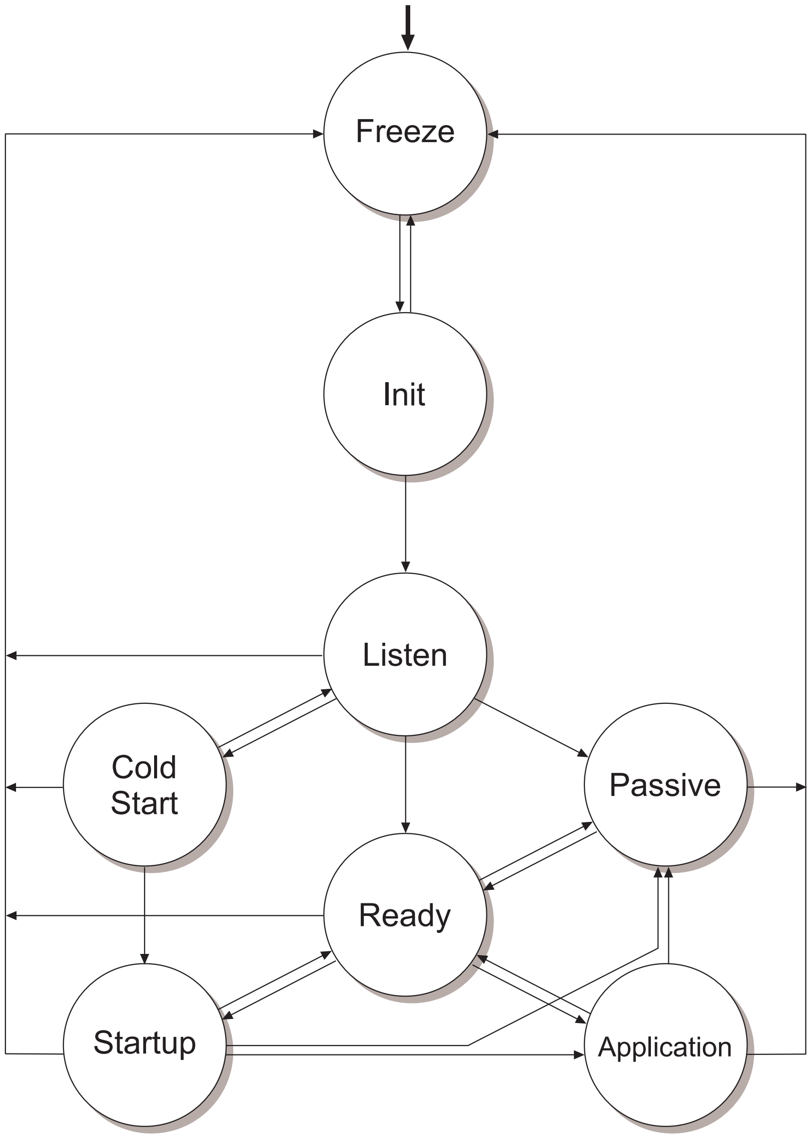

Figure 4.3: State Transition Graph of the TTP/C Controller

At first the CNI memory is cleared, except that the CO field is omitted. The local copy (`lcni`) of the CNI status area is cleared as well. Then the personality field is filled in with the values read from MEDL.

The MEDL validity check is performed by a call to a special function `medl_validity_check()`. However this function does nothing in current version and just returns that the MEDL is okay. Then the function determines the controller's membership bit position by searching the MEDL slot control entries for matching logical name. Then the function sets the controller ready field in the CNI to inform the host that the initialization is complete.

Finally the function swiches to the listen state. All state transitions are done by a call to the function:

```
void set_state(CONTROLLER *controller, unsigned state);
```

4.8.4 Listen State

The initial part of the listen state is handled by the function:

```
int controller_prepare_listen_state(CONTROLLER *controller);
```

At the beginning the controller checks whether the CO field is set, otherwise it transits to the freeze state. The same applies to the CA field and the bist enable field where the controller transits to await or to selftest state. In case of a state transition a nonzero value is returned.

Bus guardian is then programmed to its cold start parameters and the arm signal is set, but only if the controller is allowed to enter the cold start state, i.e. if the CS field in the MEDL controller configuration parameters is set and the I-frame switch in the CNI protocol status is less or equal to the maximum stated in the max cold start entry. This is determined by the function:

```
unsigned is_cold_start_allowed(CONTROLLER *controller);
```

After the above process is finished the controller must wait for any incoming I-frames, for a listen timeout period, if it is allowed to enter the cold start state, and for an infinite time, if it is not.

Once an I-frame is received (or the listen timeout period is expired in case of cold-start-allowed controller), the execution goes to the function:

```
void controller_finish_listen_state(CONTROLLER *controller);
```

At first the CO field is checked again and the freeze state is entered if necessary. If no bus activity has been observed, the cold start state is entered.

The controller checks whether there has been bus activity and a valid I-frame has been received. It is determined by the function:

```
unsigned get_valid_i_frame(CONTROLLER *controller);
```

It returns the channel number (0 or 1), where the valid I-frame has been received, or `0xFFFF` in case of no valid I-frame.

Synchronization on I-Frame

The synchronization on a received I-frame is done by the function:

```
void synchronize_on_i_frame(
  CONTROLLER *controller,
  unsigned channel);
```

This function is quite lengthy. It first sets the new bus guardian parameters to allow the controller immediate transmission, there is also a flag holding the info, that the controller can transmit immediately) in the controller's local data (`immediate_send_permission`). Then the accept counter is incremented.

The pivotal task in the function is to copy the C-state that is present in the received I-frame to the controller's own C-state and to set the cluster time correctly. To contain reasonable value, the cluster time must be set to the time when the I-frame transmission started. Luckily, this is just the value present in the C-state time field. There are some more fields concerning the cluster time in the controller's local data, more on that in section 5.3.5.

In the end the controller goes to the next slot via a call to the function `set_next_slot()`, enters the appropriate protocol state (ready or passive) and raises a mode change and membership change interrupt.

4.8.5 Cold Start State

The cold start state execution is divided in three functions. The first of them is the following:

```
int controller_prepare_cold_start_state(
  CONTROLLER *controller)
```

This function does the initial actions connected with the cold start. It sets up the new C-state (with time equal to zero), then prepares the I-frame on the appropriate determined by the value of the I-frame switch (which is then incremented). Finally it sends the frame. The return value is nonzero, if the function failed, i.e. the CO field was cleared etc. The functions involved in these actions are:

```
void setup_cold_start_c_state(
  CONTROLLER *controller);

void setup_cold_start_i_frame(
  CONTROLLER *controller,
  unsigned channel);

int send_frame(
  CONTROLLER *controller,
  unsigned channel);
```

When transmitting, the frames must always be prepared in the controller's local data (`frame[]`). The `send_frame` function passes the prepared frame to the appropriated channel and returns nonzero value, if the action was successfull.

The next function concerning the cold start state is:

```
void controller_run_cold_start_state(
  CONTROLLER *controller,
  unsigned macroticks);
```

It is responsible for correct updating of the controller's C-state during the cold start timeout period when the controller waits for an answer to its previously sent cold start I-frame. The parameter macroticks tells the function how many macroticks elapsed and the function updates the C-state appropriately.

The function that handles the rest of the cold start actions is:

```
int controller_finish_cold_start_state(CONTROLLER *controller);
```

Its work is simple and is quite similar to the one of the listen state function `controller_finish_listen_state()`. If bus activity has been observed, but no valid I-frame has been received, the controller transits back to the listen state. A correct I-frame in this case is however not only the one that is correctly received, but also the one, whose C-state correctly match the controller's own. It is determined by function:

```
unsigned cold_start_get_valid_i_frame(CONTROLLER *controller);
```

In case of no bus activity during the cold start timeout the cold start state actions should be repeated.

4.8.6 Synchronized States

The synchronized states are ready, passive, startup and application. The model handles them all the same with only minor differencies. Please, see the explanation of the pre-send (4.7.2) and post-receive (4.7.3) phases.

4.9 Bus Guardian

The bus guardian is an autonomous subsystem of the TTP/C controller that protects the communication channels from temporal transmission failures. It uses an independent oscillator for its clock to prevent a common failure mode. The idea is that when the controller breaks up and transmits randomly, the bus guardian would allow bus access only during the controller's own slot and thus prevent otherwise possible bus collisions.

4.9.1 Principle of operation

The controller periodically sets the arm signal of the bus guardian. As shown on figure 4.4, the arm signal starts a bus access cycle, which consists of three periods.

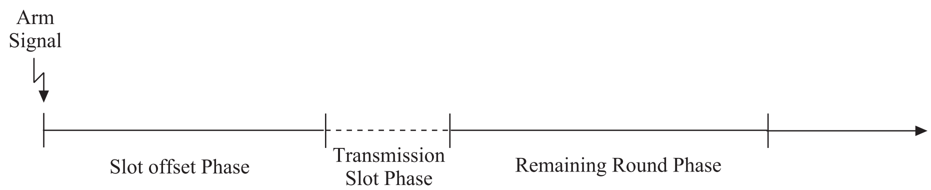

Figure 4.4: Bus Access Cycle

Slot offset phase The time duration from the arm signal to the controller's own sending slot. The bus access is disabled during this phase.

Transmission slot phase The time duration of the transmission of the longest controller's message. The bus access is enabled during this phase. During this phase the controller is responsible for clearing the arm signal. If it fails to do it, it is considered an error.

Remaining round phase The time duration to the next expected arm signal. The bus access is disabled during this phase.

Only after all of these phases are complete a new arm signal can be set. If the controller sets the arm signal too early, it is considered a failure.

4.9.2 Implementation

The bus guardian is implemented by the module fp1busg. The data structure for the bus guardian is declared as follows:

```
typedef struct bus_guardian {
  void *controller;

  unsigned bg_state;
  /* configuration registers */
  unsigned tso;
  unsigned tsd;
  unsigned trr;
  /* control registers */
  unsigned am;
  unsigned ht;
  unsigned rc;
  unsigned cmod;
  unsigned saved_cmod;
  /* status registers */
  unsigned error_code;
  unsigned state_code;
  unsigned lf;
  unsigned er;
  /* control functions (methods) */
  FUNC_BUS_GUARDIAN_ARM_SIGNAL arm_signal
} BUS_GUARDIAN;
```

The fields and their meaning:

`bg_state` – The numerical code of the bus guardian's state. The states are defined by the `BUS_GUARDIAN_STATES` enumeration.

`tso` – The slot offset register.

`tsd` – The slot duration register.

`trr` – The remaining round register.

`am, ht, rc, cmod` – The arm, halt, reconfiguration and controller mode flags.

`error_code, state_code` – The error code register and the state code register, which holds the state number, at which the error occured.

`lf` – Lifesign field. It is set when the bus access is enabled.

`er` – Error flag.

`arm_signal` – Pointer to a function that is used by the controller to pass the arm signal to the bus guardian.

4.9.3 Programming

The three mandatory timeout parameters are needed for correct operation of the bus guardian can be programmed easily with the following function:

```
void bus_guardian_set_params(
  BUS_GUARDIAN *bus_guardian,
  MEDL_BGP *params);
```

The function does reset the bus guardian as well so it is ready for the arm signal. Unlike in hardware in the model the arm signal can be set immediatelly after the programming.

The arm signal is set with the function pointed to by the `arm_signal` field of the `BUS_GUARDIAN` structure. It is done this way to make modifications to the bus guardian possible and to simplify the integration with the C-Sim toolkit. The function has the following prototype:

```
void arm_signal_func(
  BUS_GUARDIAN *bus_guardian,
  unsigned value);
```

The parameter value is either 1 to set the arm signal or 0 to clear the arm signal.

4.9.4 Bus Guardian States

The state diagram of the bus guardian is depicted in figure 4.5.

Below each of this states is described along with the function that implements it.

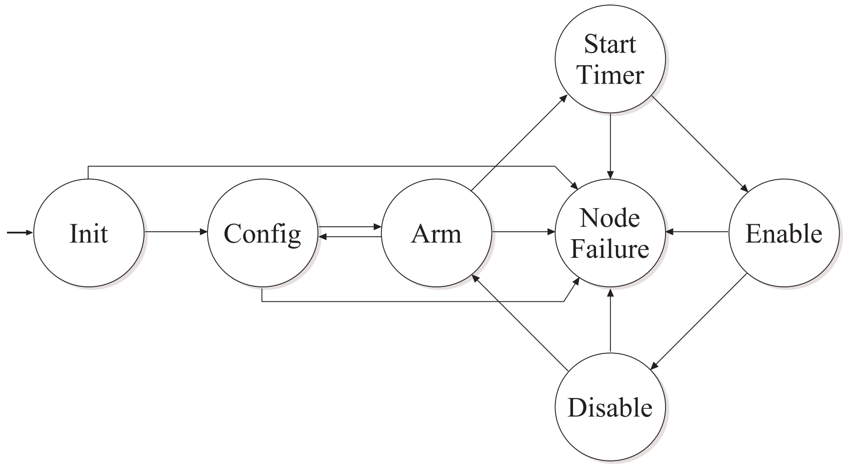

Figure 4.5: State Diagram of the Bus Guardian

Init State

```
void bus_guardian_init_state(BUS_GUARDIAN *bus_guardian);
```

I this state all of the bus guardian's internal variables are initialized to their standard values. This is done by the function `bus_guardian_initialize`. Then it transits to the config state immediately.

Config State

```
void bus_guardian_configure_state(BUS_GUARDIAN *bus_guardian);
```

The bus guardian waits in this state until the rc flag is reset, i.e. the timeout parameters are programmed.

Arm State

```
void bus_guardian_arm_state(BUS_GUARDIAN *bus_guardian)
```

This is the state, in which the bus guardian waits for the arm signal. Actually it is the only state, in which the arm signal can be set. When the arm signal is set, the execution transits to the start timer state.

Start Timer State

```
void bus_guardian_start_timer_state(
  BUS_GUARDIAN *bus_guardian);
```

This function should be called after the slot offset timeout expires. It sets the lf flag and transits the execution to the enable state.

Enable State

```
void bus_guardian_enable_state(BUS_GUARDIAN *bus_guardian);
```

This function should be called after the slot duration timeout expires. It clears the lf flag and transits the execution to the disable state. If the arm signal is not reset, the execution transits to the node failure state.

Disable State

```
void bus_guardian_disable_state(BUS_GUARDIAN *bus_guardian);
```

This function should be called after the remaining round timeout expires. It just transits the execution back to the arm state.

Node Failure State

This state uses no function, nothing happens in this state. It just indicates that an error occured.

4.10 Services from Higher Layers

After power up, the host must turn on the CO field. This starts the controller. Then it must wait for the controller ready field to be set. Finaly besides running the application software it must start updating the lifesign field periodically.

4.10.1 Updating the Lifesign Fields

As long as the host is active it has to periodically update the lifesign field. The pseudocode bellow illustrates the algorithm, with which the controller test the host lifesign and also updates its own. This code is performed by the `test_host_lifesign()` function.

```
controller_lifesign() {
  if (invert(cni.host_lifesign) == cni.controller_lifesign)
    host_status = ALIVE;
  else
    host_status = DEAD;
  cni.host_lifesign = 0;
  cni.controller_lifesign = clear_MSB(c_state.time);
}
```

The host must update its lifesign using the following pseudocode.

```
host_lifesign() {
  cni.host_lifesign = invert(cni.controller_lifesign);
  if (old_controller_lifesign != cni.controller_lifesign)
    controller_status = ALIVE;
  else
    controller_status = DEAD;
  old_controller_lifesign = cni.controller_lifesign;
}
```

For the operation to be the most reliable the controller should update its lifesign during the pre-send phase (4.7.2) and the host should update its lifesign as close as possible before the controller checks it.

As soon as the host fails to update its lifesign the controller falls to the passive protocol state and stops transmitting messages. Therefore it will lose its membership in the cluster.

4.10.2 External Clock Synchronization

External clock synchronization works on the application level and thus is in the responsibility of the host. For this purpose the external rate correction field is provided in the CNI. It must be noted that the controller never clears the field by itself so the host is also responsible for clearing of the field once the value has been applied.

4.10.3 Error Handling

In order to insure continous operation the host must contain capabilities to handle certain error states. Basically two types of errors exist in the model:

Host Errors These errors are not fatal to the protocol operation and thus the controller does not need to transit to the freeze state. The host may respond either by ignoring the error or by reseting the controller.

Protocol Errors These errors are fatal and the controller therefore transits to the freeze state. The host may obtain diagnostic information from the CNI protocol status fields. Then it may restart the controller by turning on the CO field.

Chapter 5

TTP/C Process-Oriented Simulation

5.1 The C-Sim Tool

This chapter describes the C-Sim tool briefly. For more detailed description please consult the C-Sim documantation [CSIM].

5.1.1 Overview

C-Sim is a programming tool designed to provide a way for easy creation of simulations of discrete processes. It is an extension of C language obtained by including SIMULA-like macros and functions. C-Sim provides the user with a set of special C functions macros. The typical application area of C-Sim is functional validation of distributed, parallel and fault-tolerant systems and programs.

The major advantages of C-Sim over other simulation tools include:

- Simplicity – The set of functions and macros C-Sim offers is quite small, easy to learn and once you understand the principle quite self-explanatory.
- Speed – The simulation is written entirely in C and therefore once compiled it is obviously very fast.
- Portability – By using the C language in its ANSI-C norm the model is compilable and runable on almost any machine running Windows, any unix-like system and possibly others.

- Code Reusability – Again the C language. The parts of the model the user writes can be then easily reused in the real application or possibly in other models etc.

5.1.2 The Principle

The idea of C-Sim was taken from the programming language SIMULA and the library provides SIMULA-like resources from the system classes SIMSET and SIMULATION.

Like SIMULA, the C-Sim needs the description of behavior of the simulated system in the form of processes. To model the objects of the real world with program objects in the process form is straightforward and natural. The C language, unlike SIMULA, has no tools for object oriented programming but some of the basic principles were implemented using predefined macro commands.

For example to declare a new process derived from the standard C-Sim PROCESS structure, the following code is used:

```
derived_process(my_process)
  /* user-defined attributes, e.g.: */
  struct my_process *p_myself;
end_derived(MY_PROCESS);
```

The above code expands after passing through the C preprocessor to the following:

```
typedef struct my_process {
  ... /* standard PROCESS attributes */
  struct my_process *p_myself;
} MY_PROCESS;
```

As you can see a pointer to `MY_PROCESS` would be somewhat compatible with a pointer to `PROCESS`. This works exactly the same as you would expect from an object oriented language like C++.

5.1.3 Pseudo-Parallel Processes

The used method of discrete simulation is based on the use of pseudo-parallel processes. The programs of the processes are constructed as reentrant hence one program can be used by several processes (every process has its own local data record). The process programs are also called cooperating routines.

The processes are running in so-called model time, which in contrast to the real time, can be arbitrarily slowed down or accelerated. The life of processes in model time has a discrete character, i.e active phases of the process have a zero duration in model time and between these phases there are phases of inactivity with non-zero duration in model time.

It is obvious that switching between several running process programs is needed. The approach, which was used during the implementation of the library, is based on the functions `setjmp()` and `longjmp()`, which are part of the standard C-language library. These functions allow storing the process context into the predefined structure and later return to the stored process status.

5.1.4 Hello World in C-Sim

In this chapter a simple program will be created using the C-Sim toolkit. The program consists of two running processes of the same kind, which print characters on the screen at selected model time intervals. As the two characters of the two processes interleave, they result in a popular "Hello World" message. A complete source code of the example can be found in appendix A.

At the beginning of each program, that uses C-Sim we must include the header file among others:

```
#include "kr_csim.h"
```

Then we declare a data structure of our character-printing process. The process will need to know the characters it is supposed to print, the model time interval between them and also the number of already printed characters:

```
derived_process(hello_printer)
  char *text;
  double interval;
  unsigned count;
end_derived(HELLO_PRINTER);
```

Then of course the process needs a program. So we create a simple process program that holds for the selected model time interval and then prints the character in a loop:

```
s_program(HELLO_PRINTER, prog_hello_printer)
  for (my.count = 0; my.count < strlen(my.text); my.count++) {
    hold(my.interval);
    putchar(my.text[my.count]);
  }
s_end_program
```

hold() is a C-Sim macro that suspends the process execution for a given model time interval. Meanwhile other processes may run.

It is wise to have a creator function for each process type in an application. For our simple example it may look like this:

```
HELLO_PRINTER *new_hello_printer(char *f_text, double f_interval)
{
  HELLO_PRINTER *hello_printer;

  hello_printer = s_new_process(
    HELLO_PRINTER, prog_hello_printer);
  hello_printer->chars_to_print = f_chars_to_print;
  hello_printer->interval = f_interval;

  return hello_printer;
}
```

Now we have everything prepared to write the main function for our application. Before doing anything that concerns processes, the C-Sim memory management must be initialized:

```
  inic_mem();
```

After the memory is initialized we may step to the creation of processes. Assuming that we have two variables h1 and h2 declared as pointers to HELLO_PRINTER, we may do the following:

```
  h1 = new_hello_printer("Hlo ol!", 2.0);
  h2 = new_hello_printer("el,Wrd\n", 2.0);
```

We have just created two hello_printer processes. Each of them is given six characters to be printed and time interval of two seconds (of the model time). The character strings may seem senseless, but at closer look they are just a splitted “Hello, World!” message.

Having the processes created is not sufficient to run them, they must be activated first. We will activate both of the processes, but one of them just one second (of the model time) later then the other. Note that the h1 and h2 variables must be type-casted to pointers to PROCESS, this is needed due to C language limitations:

```
  activate_at((PROCESS *)h1, 0.0);
  activate_at((PROCESS *)h2, 1.0);
```

Now we can run our simple simulation by making steps. The `step()` function makes a single model step. It returns nonzero value in case of an error. Since when there is no remaining process, it is considered an error, we may simply make steps until both of our processes are finished with an empty-body loop:

```
while(step());
```

After all the steps are finished, we finalize the C-Sim memory management and our program is complete.

```
clear_mem();
```

The `clear_mem()` function finalizes the memory management and is also responsible for freeing all created processes.

5.2 Structure of the TTP/C Simulation

The C modules, that implement the simulation, are all named in the form `sp1xxxx`. More on that can be found in [GRI01].

Every TTP/C Cluster C-Sim based model uses at least five types of C-Sim processes, these are namely:

- Channel Process
- Controller Process
- Timer Process
- Bus Guardian Process
- Host Process

These processes are interconnected as depicted in figure 5.1, which shows a model of a two-node TTP/C cluster. Each arrow in the figure represents a C pointer. The pointer from host to the controller is optional since only pointer to the controller's CNI memory is required for the host.

5.2.1 Channel Process

There are two channel processes in the simulation. Each of the processes simulates one of the two communication channels of the TTP/C network. These processes encapsulate the data link layer of the TTP/C protocol.

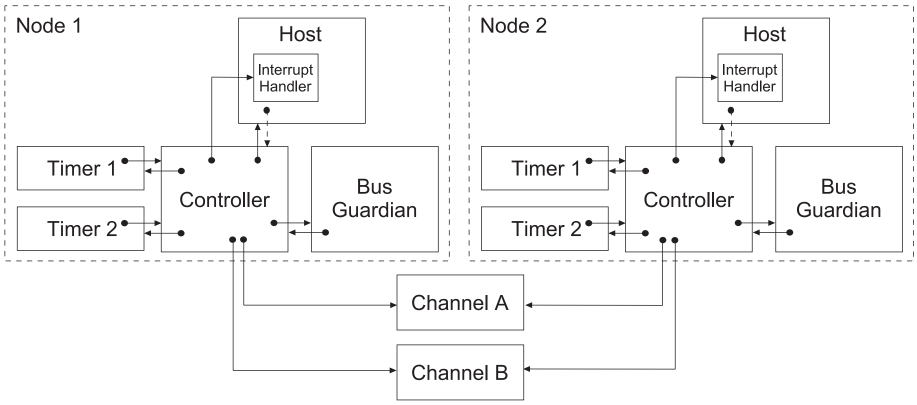

Figure 5.1: Processes Interconnection

Each controller knows the channels it is attached to via two C language pointers. The controllers periodically call send and receive functions from the channel module. The channel records receive requests from controllers in an array and when a frame is sent, it passes the frame to all of the recorded controllers.

5.2.2 Controller Process

For each controller in the simulated TTP/C cluster there is one instance of the C-Sim controller process. That process encapsulates the SRU and the reconfiguration layers of the TTP/C protocol.

Each controller has pointers to its bus guardian, to its host and to both channels it is attached to.

5.2.3 Timer Process

Each controller has two attached timer processes, which are used for the timer interrupts. When a timer interrupt has to be raised, a timer process is planned on the appropriate point in time, where it then raises the interrupt.

5.2.4 Bus Guardian Process

An instance of the bus guardian process is created for each controller in the simulated TTP/C cluster. It has a pointer to its controller.

When sending a frame, the channel asks the bus guardian of the sender, whether it has enabled access to the bus.

5.2.5 Host Process

The host process is currently not a part of the TTP/C model. It is the user's responsibility to create one. Such process is then responsible for all the services that the application layer must provide (see 4.10).

The host may have a pointer to the controller, although a clearer way would be to have only a pointer to the controller's CNI. Upon creation the host must also provide the controller with an interrupt handler routine to allow the controller to raise interrupts.

5.3 TTP/C Controller Submodel

5.3.1 Implementation

As an extension to the `fp1ctrl` module the TTP/C controller C-Sim process model is implemented by the sp1ctr module. The controller model data structure is declared as an extension to the `CONTROLLER` structure of the `fp1ctrl` module:

```
typedef struct controller_model {
  ... /* The same fields as in CONTROLLER structure */
  CONTROLLER_PROCESS *process;
  int wake_on_co;
  int wake_on_ca;
  int wake_on_receive;
  double saved_csim_time;
  TIMER_PROCESS *timer[2];
} CONTROLLER_MODEL;
```

`process` – This is reference to the derived C-Sim process for the controller, it is declared the following way:

```
derived_process(controller_process)
  CONTROLLER *protocol;
  struct controller_model *model;
end_derived(CONTROLLER_PROCESS);
```

The `controller_model` field is reference back to the process owner, i.e. the `CONTROLLER_MODEL` structure.

`wake_on_co` – If this flag is set, the controller process would be woken up when the CO field of the CNI changes.

`wake_on_ca` – If this flag is set, the controller process would be woken up when the CA field of the CNI changes.

`wake_on_receive` – If this flag is set, the controller process would be woken up by the channel when a frame is received.

`saved_csim_time` – This is an artificial variable used for temporary storage of C-Sim model time.

`timer[2]` – This is the reference to the two timer processes used for the timer interrupts. The timer processes are just very simple. They are just planed once a timer interrupt is enabled and when activated, they raise the interrupt by calling the host interrupt handler.

This aproach of making the `CONTROLLER_MODEL` data structure compatible with the `CONTROLLER` data structure makes it possible to use it with the functions from the fp1ctrl module, i.e. the TTP/C controller functional specification. However due to the C language limitation it must always be typecasted.

5.3.2 Simulating the CNI

Certain fields of the CNI must have an immediate effect on the C-Sim processes of the TTP/C controller simulation. Therefore the functional specification of the `fp1cni` module needs an extension. This extension is placed in the module `sp1cni`. The get and set functions for such fields often call other functions from other modules of the simulation. The fields are namely:

`timer1, timer2` – Setting this fields signifies that the timer processes that handle the timer interrupts need to be replanned. Then the function `controller_timer_changed()` of the `sp1ctrl` module is called.

`ti1, ti2` of the interrupt enable field – The set function for these fields affect the timer processes as well.

`co` – Setting this field often has the meaning of reseting the controller, so the controller process should be immediately woken up. The set function for this field calls the `controller_co_changed()` function of the `sp1ctrl` module.

`ca` – The set function for this field calls the `controller_ca_changed()` function of the `sp1ctrl` module. It should transit the controller to the await protocol state. It is not implemented, but still the model responds to it.

`cluster_time` – Unlike the other above fields, for this one only the get function exist in the sp1cni module. That is to say an up-to-date time value should be always returned. Therefore the get function first calls the `update_cluster_time()` function, which is pointed to by a field in the `CONTROLLER` structure. See section 5.3.5.

5.3.3 Process

The C-Sim process of the TTP/C controller is present in the `sp1ctrl` module and is named `prog_controller`. It consists mainly of the switch-case block driven by the controller's protocol state. The individual branches of the switch block mostly calls the functions of the `fp1ctrl` module, which were described in the reconfiguration layer (see 4.8). To handle the time flow, C-Sim commands (like `hold()` or `passivate()` etc.) are inserted between these function calls.

For most functions, that services the individual protocol states of the RC layer, have an extension implemented called `model_xxx`, where `xxx` is the name of the original function. These extensions mostly call their equivalents and do some additional stuff concerning the C-Sim process. For example the function:

```
int model_prepare_listen_state(CONTROLLER_MODEL *controller);
```

It just does the same as `controller_prepare_listen_state()` and then sets the `wake_on_co` and `wake_on_receive` flags to ensure that during the oncoming listen timeout the process would be woken up immediatelly after the CO field changes or a frame is received.

5.3.4 Synchronized Actions

The fundamental part of the process function is executed when the controller is synchronized, i.e. in states startup, application, passive or ready. It is divided in two major parts, the first one is executed, when the controller is the sender in the current slot, otherwise the second is executed.

The controller first computes the point in model time of the next controller's own action time and stores it in a variable in the controller's local data (`ldata.action_time`). Then when it is the current sender, which is determinded by calling the `psp()` function (see 4.7.2]), it waits for the calculated action time. Then it raises any pending synchronized interrupts like user interrupts etc.

Now when the controller has an immediate send permission, i.e. it has just synchronized and now reacts on an I-frame received in the listen state, it sets the bus guardian arm signal now. Then it sends frames on both channels ba calling the `send_frame()` function twice. The frames are already previously prepared in the controller's local data by the `psp()` function. After that the controller waits for the end of the transmission phase, programs the bus guardian with normal (synchronized) parameters.

If the controller is not the current sender, it waits for the action time as well. Then it raises any pendindg synchronized interrupts and if the current slot has the BA flag set in the MEDL mode address entry, it sets the bus guardian arm signal. Then it waits for the end of the transmission phase.

Then, regardless of if the controller is the sender or not, the `prp()` function is called to perform the post-receive phase (see 4.7.3). Finally the controller registers itself as an receiver by calling the `start_receive()` function and updates the CNI.

5.3.5 Cluster Time Flow

For converting model time to macroticks or microticks the following functions exist:

```
double controller_micro_to_csim(
  CONTROLLER_MODEL *controller, double x);

double controller_csim_to_micro(
  CONTROLLER_MODEL *controller, double x);

double controller_macro_to_csim(
  CONTROLLER_MODEL *controller, double x);

double controller_csim_to_macro(
  CONTROLLER_MODEL *controller, double x);
```

They convert a duration in microticks or macroticks to the model time and vice versa counting in the drift of the controller's clock.

Every time the controller or the host process need to know the current cluster time, they call the `cni_get_cluster_time()` function of the CNI, which then calls back the `update_cluster_time()` function of the controller. This function is responsible for keeping the cluster time up to date, it refreshes the value according to the passed model time since the last call to it. Since this all is done every time the cluster time field is accessed, it makes it seem like the field always contain an up-to-date value.

Besides the cluster time fields there are several other variables that helps to keep the exact time to make this mechanism working. These additional variables are placed in the controller's local data and they are namely:

`double clock_drift` – This is a non-dimansional number that represents the clock drift of the controller. Each controller may have different clock drift. If this value is greater than zero, the controller's clock is faster than ideal, if this value is smaller than zero, the controller's clock is slower than ideal.

`double clock_correction_time` – This variable holds the next point in model time, where the clock correction may be applied. It is calculated using the free running macroticks field of the MEDL, i.e. the minimum number of macroticks in a row that must remain unchanged. The variable may point to the past, saying that a clock correction may be applied now if desired, or to the future, which means that the clock correction cannot be applied yet.

`double last_clock_access` – This holds the exact model time of the last call to the `update_cluster_time()` function. It is then used to determine the passed model time since the last call.

`double last_mt_fract` – Since the CNI cluster time field only stores the integral part of the time in macroticks, the fractional part must be stored somewhere else. This variable holds the fractional part of elapsed macroticks at the time of `last_clock_access`, i.e. the last call to the `update_cluster_time()` function.

Now let's have a closer look on the `controller_update_cluster_time()` function. This function is responsible for all the time flow in the controller. It is placed in the `sp1ctrl` module and is the field `update_cluster_time` of the `CONTROLLER` structure points to it.

The first task is to calculate the number of microticks, that should be added to or substracted from the passed time, i.e. calculate the clock correction, that should be applied now.

```
dt = CSIM_TIME() - ldata->clock_correction_time;
```

Now `dt` is the elapsed time since the last point in time, when a clock correction could be applied. Note that `dt` could also be negative, which happend when the `clock_correction_time` variable points to the future.

```
c = get_clock_correction(controller_model, dt);
```

The variable `c` now contains the number of microticks that should be added (or substracted in case of negative value) to the computed elapsed total. That is the amout clock correction that will be appliad now, so the clock state correction term must be decremented by this value.

```
cni_ps_increment_csct(
  &controller->lcni->protocol_status, -c);
```

The closest point in time, when a next clock correction would be possible, can now be shifted by c times the free running macroticks.

```
ldata->clock_correction_time +=
  abs(c) * controller_macro_to_csim(controller_model,
    medl_ccp_get_free_running_macroticks(medl_ccp));
```

Now we can proceed to calculating of the elapsed macroticks. We must add the fractional part of the cluster time (`last_mt_fract`) as well as the previously determined amount of clock correction microticks.

```
x = CSIM_TIME()
  - ldata->last_clock_access + ldata->last_mt_fract;
x = controller_csim_to_macro(controller_model, x);
x += (double)c / medl_ccp_get_micro_to_macro(medl_ccp);
```

Now `x` is the number of macroticks that elapsed since the last call to the `update_cluster_time()` funciton, counting the clock drift and clock correction `c`.

Here the `last_clock_access` and the `last_mt_fract` can be updated.

```
ldata->last_clock_access = CSIM_TIME();
ldata->last_mt_fract =
  controller_macro_to_csim(controller, fract(x));
```

Since `x` contains the total amount of macroticks that elapsed since the last call, the cluster time field can now be incremented by the integral part of `x`. We must increment cluster time field of both the external CNI as well as the local status copy of it.

```
n = (unsigned)x;
controller->lcni->cluster_time += n;
controller->cni.cluster_time =
  controller->lcni->cluster_time;
```

5.3.6 Usage

A new controller can be created by a call to the following function:

```
CONTROLLER_MODEL *controller_create(char *name);
```

The `name` parameter is just any character string. It is not mandatory, but still useful to fill in, since the name can be then be very helpful during debugging.

The function does all the work needed for the creation of the controller, it allocates the memory and creates the C-Sim process for it. It also automatically attaches a bus guardian to the controller and the two timer processes.

5.4 Bus Guardian Submodel

5.4.1 Implementation

As an extension to the fp1busg module the bus guardian C-Sim process model is implemented by the sp1busg module. The bus guardian model data structure is declared as an extension to the `BUS_GUARDIAN` structure of the `fp1busg` module:

```
typedef struct bus_guardian_model {
  ... /* The same fields as in BUS_GUARDIAN structure */
  BUS_GUARDIAN_PROCESS *process;
  double clock_drift;
} BUS_GUARDIAN_MODEL;
```

`process` – This is reference to the derived C-Sim process for the bus guardian, it is declared the following way:

```
derived_process(bus_guardian_process)
  BUS_GUARDIAN *protocol;
  struct bus_guardian_model *model;
end_derived(BUS_GUARDIAN_PROCESS);
```

The `bus_guardian_model` field is reference back to the process owner, i.e. the `BUS_GUARDIAN_MODEL` structure.

`clock_drift` – This field holds the clock drift of the bus guardian. If this value is greater than zero, the clock of the bus guardian is faster than ideal, if it is smaller than zero, the clock is slower than ideal. The clock drift can be different to the one of the controller.

5.4.2 Process

The C-Sim process of the TTP/C controller is present in the sp1busg module and is named `prog_bus_guardian`. It is basicly jast onde switch-case block driven by the state of the bus guardian. Each branch of the switch block represents one state and calls an appropriate function from the fp1busg module. These functions were described in section 4.9. To handle the time flow, C-Sim commands (like `hold()` or `passivate()` etc.) are inserted between these function calls.

5.4.3 Usage

A new bus guardian can be created by a call to the following function:

```
BUS_GUARDIAN_MODEL *bus_guardian_create(char *name);
```

The name parameter is just any character string and is not mandatory, though it may be helpful during debugging. The function does all the work needed for the creation of the bus guardian, it allocates the memory and creates the C-Sim process for it.

For the purpose to attach the bus guardian to a specific controller the function `bus_guardian_attach_controller()` exist. However the user of the TTP/C model library hardly creates a bus guardian alone, since it is created automatically when creating a controller.

5.5 Communication Channel Submodel

As an extension to the `fp1chan` module the channel C-Sim process model is implemented by the `sp1chan` module. The channel model data structure is declared as an extension to the `CHANNEL` structure of the `fp1chan` module:

```
typedef struct channel_model {
  ... /* The same fields as in CHANNEL structure */
  CHANNEL_PROCESS *process;
  unsigned faults_to_inject;
  unsigned injected_faults;
} CHANNEL_MODEL;
```

`process` – This is reference to the derived C-Sim process for the channel, it is declared the following way:

```
derived_process(channel_process)
  CHANNEL *protocol;
  struct channel_model *model;
end_derived(CHANNEL_PROCESS);
```

`faults_to_inject` – A simple fault injection is implemented in the model of the channel. It does a frame corruption by a single bit flip. This variable holds the number of frames that should be corrupted that way.

`injected_faults` – This is the total number of corrupted frames by the fault injection. The purpose of this field is only to gather statistical information.

5.5.1 Process

The process of the channel is placed in the sp1busg module and its name is `channel_prog`. It is really very simple. It is all the time passive and wakes up only when a frame transmission is complete. Then it calls the `channel_send_complete()` function which is responsible for noticing the receivers.

The major work of the channel is done by the function:

```
int channel_send_frame(
  CHANNEL *channel,
  void *controller,
  PTR_CALLBACK);
```

This function copies the frame from the controller to the channel's local variable. Then it calculates how long does it take to transmit the frame at the given transmission speed. Knowing th duration it plans the channel process to wake up after that time.

At the moment of sending a frame all the receiver controllers must already have been registered in the channel as receivers. Should any receiver register itself too lete, it will be given a corrupted frame.

The third parameter of the `channel_send_frame()` function is a pointer to a callback function, that will be called once the frame is transmitted. It can be used by the sender, if it wants to be notified of it.

5.5.2 Usage

The channel can be easily created by the function:

```
CHANNEL_MODEL *channel_create(char *name)
```

The name parameter is just any character string and is not mandatory, though it may be helpful during debugging. The function does all the work needed for the creation of the channel, it allocates the memory and creates the C-Sim process for it.

To attach a controller to the channel the user must additionally do a call to the following function:

```
controller_attach_channel(
  CONTROLLER *controller,
  CHANNEL *channel,
  int number);
```

It attaches the controller given by the first parameter to the channel given by the second parameter. A controller an be attached to two channels simultaneously, therefore the third parameter denotes the number of the channel, it can be either 0 or 1.

5.6 Model Limitations

5.6.1 Physical layer of TTP/C protocol

There is no physical layer (i.e. bit Encoding/Decoding) implemented in the model. Transmission of a frame is started by a sender at a `t1` point of model-time, a flag of channel activity is set on, but all the frame is copied to the receivers model time point `t1+transmit_time`. This simplification still enables to corrupt the frame "during the transmission" including selective corruption for various receivers.

5.6.2 Discretized computation

Normally the hardware controller spends the interframe gap duration executing its microcode. In the model however this activity is discretized and thus limited to a single point in time.

It is quite hard to predict the impact it may have on the results of the model run.

5.6.3 Code Cohesion

The functional specification of the protocol does not implement any time durations at all. Everything concerning time is implemented in the C-Sim part of the model. This however forces that also some parts that would logically belong to the functional specification must be implemented in the C-Sim part. Therefore some portions of the code maight not seem so clear.

To solve this problem only the library needs not to be divided in the two parts, i.e. the functional specification and the C-Sim model. Since one part of this work was to create a functional specification of the TTP/C protocol using pure C without any special libraries including C-Sim, this is afield.

5.6.4 Multithreading Needed for Visualization

This issue is a general problem of C-Sim. The C-Sim process switching uses a way that corrupts the program stack. This would cause problems with any windowing environment, either Win32 or GTK+ or any other. Therefore if visualisation is needed, the simulation must run in a separate thread, which has a separate program stack.

Since all the modern operating systems allow multithreading, this is not a major problem after all.

Chapter 6

TTP/C Cluster Communication Activity Model

In this chapter a simple TTP/C cluster simulation will be created. The used application will be empty, that is the hosts will do no actions but udating the host lifesigns. Such application were used during the model implementation for debugging purposes. We call the application "Empty Application".

The Empty application has the following properties:

- four TTP/C nodes
- only I-frames are sent
- one operational mode
- two identical TDMA rounds in the transmission cycl
- all slots 250 macroticks long
- TDMA round 1000 macroticks long
- macrotick one microsecond long
- no fault injection

To create an application with the TTP/C library, we must create the following:

- a module that implements the hosts

- a module that creates and interconnects all the components
- `main()` function
- text files with the MEDL tables

In the text bellow only portions of the code are presented, for a complete source listing please see the appendix B.

6.1 Empty Host

At first we need a C-Sim process for the host. Let's name the host module `sa1ehost`, the host can be declared for example the following way:

```
derived_process(empty_host)
  CNI *cni;
  unsigned index;
  unsigned max_iterations;
  double last_ti1;
  /* ...possibly other fields */
end_derived(EMPTY_HOST);
```

`cni` – This is the pointer to the attached controller's CNI memory

`index` – Here we will store the number of the host, i.e. its position in the TDMA round.

`max_iterations` – This is the maximum number of TDMA rounds the host will wait for the controller to reintegrate in the cluster.

`last_ti1` – Tha last point int model time, when the timer interrupt 1 was raised.

There are some other fields declared that are not so iportant for our purposes and are not decribed here, please see the complete source code for details.

Then we should have a function that would create a new host for us, i.e. create the C-Sim process, allocate memory and initialize the variables:

```
EMPTY_HOST *empty_host_create(char *name);
```

Another function that would be useful is a function, that would attach the host to a controller. The function must for example initialize the interrupt handler:

```
void empty_host_attach_controller(
  EMPTY_HOST *empty_host,
  CONTROLLER *controller);
```

We will also need a function that will do for us the lifesign updating and testing (it is done simultaneously, see the section 4.10.1):

```
int test_controller_lifesign(EMPTY_HOST *empty_host);
```

A very important function that must be created is the interrupt handler for the host. The controller will call the handler every time it raises an interrupt. In our simple Empty Host, we will manage with a funcion that only sets a flag and wakes up the host process:

```
static void controller_interrupt(void *ptr)
{
  ((EMPTY_HOST *)ptr)->interrupted++;
  process_activate_delay((PROCESS *)ptr, 0);
}
```

The last and most challenging task when creating a host is to implement its C-Sim process program. The process must be able to correctly periodically update the host lifesign field, to start the controller when it is shut down and also recover from possible errors. For the source code of the program, please see the appendix B.2.

Our host process program is divided in two branches. The first one is executed when the controller is shut down (the CO field is reset) and the second is executed upon an interrupt (by testing the interrupted flag).

The first branch is responsible for a correct startup of the controller and for the reintegration. It first turns the controller on by setting the CO field and waits until the controller declare itself ready (controller ready CNI field is set).

Once the controller is ready, the host starts updating its lifesign periodically until the controller is sychronized. The controller is sychronized once the C-state valid CNI field is set. To prevent lockups the host waits only for the duration of 10 TDMA rounds for the controller to synchronize. If it fails to synchronize within this boudary, the host restarts it again.

Once the controller is synchronized, the host enables the timer interrupt 1, which will be used for the lifesign update task.

The second branch of the host process is responsible for interrupt handling. Since we have only one type of inerrupt, the timer interrupt 1, the task is quite easy. We just update the host lifesign and plan the next interrupt.

When the host process is not woken up for a long time (say 2 TDMA rounds) it assumes, that an error occured, since the controller has not raised any interrupt. It therefore restarts the controller.

6.2 Application Module

It is a wise thing to have all the code that handles all the creation of the application in one module. We name that module `sa1eapp`.

At first we declare a data structure, that would encapsulate all the components needed in the application:

```
typedef struct empty_app {
  EMPTY_HOST *host[4];
  CONTROLLER_MODEL *controller[4];
  CHANNEL_MODEL *channel[2];
} EMPTY_APP;
```

The next step is a function that would create the application:

```
EMPTY_APP *empty_app_init();
```

It must first allocate memory for the application data structure. Then it must initislize the TTP/C model and C-Sim libraries:

```
app = (EMPTY_APP *)malloc(sizeof(EMPTY_APP));;
inic_mem();
cni_module_initialize();
```

After this it may proceed to the creation of the simulation components: two channels, four controllers and four Empty hosts. The bus guardians need not to be created since the controller create them automatically.

Then it must perform the initialization of the components and interconnect them. Please see the source code in appendix B.4 to learn the procedure, it is well commented.

The last thing the function needs to do is to activate the four host processes. The host processes are the only ones that need to be activated, the rest of the processes are activated automatically.

Another function would be useful, a one that would finalize the application and release the memory it holds. Such function is very easy due to the used C-Sim memory management:

```
void empty_app_done(EMPTY_APP *app)
{
  clear_mem();
  free(app);
}
```

6.3 Main Function

Every C program needs to have a `main()` function. In our Empty application simulation program this function is surprisingly easy:

At first create and initialize the Empty application:

```
EMPTY_APP *app;
if ((app = empty_app_init()) == NULL)
  return 1;
```

Then let it run for say one tenth of a second of model time:

```
while(CSIM_TIME() < 100000 && step())
  ;
```

And finally release the memory and exit.

```
empty_app_done(app);
return 0;
```

6.4 The MEDL Tables

The MEDL tables are present in four files:

```
medl00.txt
medl01.txt
medl02.txt
medl03.txt
```

The complete content of these files can be seen in appendices B.5 through B.8. Here only the gross overview will be presented.

All the four nodes are given a complete set of bus guardian parameters. For the cold start, for the time it is sychronized, for the reintegration and for the listen state. These parameters are calculated using the knowledge of the node's slot number. For details please see the protocol specification [SPEC].

Each node is given an unique slot position in the role list. The PS bit and the miltiplexed ID fields of the role list are left zero. According to a slot position the nodes are given a specific startup timeout. All the nodes are allowed to enter the cold start state. The TDMA round length is set to 1000 macroticks, the IFG is 105 macroticks long and one macrotick consists of 100 microticks in all nodes.

A slot control entry and a mode address entry for each slot are filled in the MEDLs. Each slot has a duration of 250 macroticks. The fields ligical name, membership pos, EOR and EOT in the slot control entry are set appropriately to correspond the slot position. All of the slots have the SYF bit set, i.e. all frames are used for the clock synchronization algorithm, and slots 3 and 7 (counting from zero) have the CS bit set, which makes them the clock synchronization points. In the mode address entries the IA and IB bits are set in all slots, i.e. only I-frames are transmitted.

6.5 Output

The output of the Empty application is formed by the debugging messages of the TTP/C model. There are tons of these debugging messages and they would consume another diploma thesis to cover all of them with a description.

However they are quite self-explanatory for someone with a knowledge of the TTP/C protocol. All of them are in the form:

```
time: event
```

The `time` is the model time and `event` is a description what happened at that point in model time.

An example output is in appendix B.9, which covers the first three miliseconds of model time of the Empty Application.

Chapter 7

Conclusion

The goal of this work was to create a strict functional (C-based) specification of the TTP/C protocol and a runable simulation of the TTP/C cluster based on C-Sim.

The functional specification has reached a point, when it is quite complete altough several portions maight need some more additional work. Namely the code that handles mode changes is not yet complete and is untested.

The runable simulation was created on top of the functional specification with the idea to test the functional specification for validity. Since the simulated TTP/C cluster is successfully running and the results gathered from the run seem reasonable, it is assumed that the functional specification is valid, at least to some point.

One thing that was not taken in consideration so far is the TTP/C operating system called TTP/os. The TTP/os runs at the host controllers in the TTP/C nodes. The presented host simulation uses only simplified form of task activation. In case of a more complicated application it maight be wise to implement a simulation of the TTP/os first and build the application on top of it.

Another impulse for the creation of the model was to make it possible to test the TTP/C protocol itself for validity. Possible ways are to use various methods of fault injection, namely random bit flips in the CNI or in the MEDL or random corruption of frames on the channels etc. A very interesting results maight be observed, when testing the clique-avoidance mechanism of the protocol be corrupting frames only for part of the cluster. There are numbers of further possibilities, some of them are covered in [GRI01].

Abbreviations and Acronyms

CA Field Controller Await Field

CO Field Controller On Field

C-State Controller State

CNI Communication Network Interface

CRC Cyclic Redundancy Check

DPRAM Dual-Ported Random Access Memory

FTU Fault Tolerant Unit

FTU CNI FTU Communication Network Interface

I-Frame Initialization Frame

IFG Interframe Gap

MEDL Message Descriptor List

N-Frame Normal Frame

NBW Non-Blocking Write Protocol

RC Layer Reconfiguration Layer

TDMA Time Division Multiple Access

TTA Time-Triggered Architecture

TTP/C Time-Triggered Protocol for SAE Class C Applications

TTP/os Operating System for Time-Triggered Protocol

Bibliography

[SPEC] TTP/C Protocol – Specification of the Protocol, version 0.1 of 01-Feb-1999, TTTech Computertechnik GmbH

[CSIM] J. Hlavička, St. Racek, P. Herout (1999): C-Sim v. 4.1, Research Report DC-99-09, http://www.c-sim.zcu.cz,

[BYSP] Byteflight protocol specification, version 0.5 of 29-Oct-1999, BMW A.G., http://www.byteflight.com

[CASP] CAN Specification, version 2.0 B, CAN in Automation, http://www.can-cia.com

[COMP] Comparison CAN vs. Byteflight vs. TTP/C, Edition 3-Apr-2000, TTTech Computertechnik GmbH

[SEMI] TTP/C Seminar/Workshop 6 May 2000, Gothenburg, Sweden, TTTech Computertechnik GmbH

[KOP97] H. Kopetz (1997): Real-Time Systems, Design Principles for Distributed Embedded Applications, Kluwer Academic Publishers, USA

[GRI01] P. Grillinger (2001): Software Project of Simulation-based Fault Injection for TTA, Diploma Thesis

[FITR] J. Hlavička, St. Racek, P.Herout (2000): C-Sim Based Model of TTP/C Protocol/Controller, FIT Project Report

[SRC] P. Grillinger, J. Ježek (2001): C-Sim Based Model of TTP/C Protocol Source Code

Appendix A

C-Sim Example Source

A.1 File hello.c

```
/* C-SIM Example - Hello, World!

   Written by: Jan Jezek
   University of West Bohemia, 2001

   Part of the diploma thesis:
   C-based Reference Model of TTP/C Controller
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "kr_csim.h"

/* The data structure of the C-Sim hello_printer
   process. */
derived_process(hello_printer)
  /* Characters to be printed on the screen. */
  char *chars_to_print;
  /* Time interval between printed characters. */
  double interval;
  /* Number of characters already printed. */
  unsigned count;
end_derived(HELLO_PRINTER);

/* The program of the C-Sim hello_printer process. */
s_program(HELLO_PRINTER, prog_hello_printer)
```

```
  /* For each character... */
  for (my.count = 0;
    my.count < strlen(my.chars_to_print);
    my.count++) {
    /* ...hold for the desired time interval... */
    hold(my.interval);
    /* ...and then print the character on the screen. */
    putchar(my.chars_to_print[my.count]);
  }
s_end_program

/* The function to create a new C-Sim hello_printer
   process. */
HELLO_PRINTER *new_hello_printer(
  char *f_chars_to_print, double f_interval)
{
  HELLO_PRINTER *hello_printer;

  /* Create a new C-Sim process of the type
     HELLO_PRINTER with the program prog_hello_printer/ */
  hello_printer = s_new_process(
    HELLO_PRINTER, prog_hello_printer);
  /* Set the process parameters to the desired ones. */
  hello_printer->chars_to_print = f_chars_to_print;
  hello_printer->interval = f_interval;

  return hello_printer;
}

int main()
{
  HELLO_PRINTER *h1, *h2;

  /* At first initialize C-Sim memory management. */
  inic_mem();

  /* Then create two hello printer processes. Let's
     give the processes interleaved characters of
     the message and use a 2 seconds long interval for
     both of them. */
  h1 = new_hello_printer("Hlo ol!", 2.0);
  h2 = new_hello_printer("el,Wrd\n", 2.0);

  /* Now activate the two processes, one at time 0
```

```
     and one a second later. */
  activate_at((PROCESS *)h1, 0.0);
  activate_at((PROCESS *)h2, 1.0);

  /* Do model time steps until there are no processes. */
  while (step())
    ;

  /* Finally clear the C-Sim memory and exit. */
  clear_mem();
  return 0;
}
```

Appendix B

Empty Application Source

B.1 File sa1ehost.h

```
/*
 * This software is part of an EU project:
 *  FIT - Fault Injection for TTA
 *  IST-1999-10748
 *
 * TTP/C - protocol v.0.1
 *
 * Design (c) TTTech & TU Vienna
 * Implementation (c) UWB Pilsen & CTU Prague
 *
 * sa1ehost.h
 * EMPTY_HOST C-Sim proces
 *
 * version 0.5 - 2001-03-17
 */

#ifndef _SA1EHOST_H
#define _SA1EHOST_H

#include "kr_csim.h"
#include "fp1cni.h"
#include "sp1ctrl.h"

/* EMPTY_HOST process data structure */
derived_process(empty_host)
CNI *cni;
```

```
unsigned index;
unsigned max_iterations;

double last_ti1;
unsigned interrupted;
unsigned old_controller_lifesign;
unsigned startup_complete;
end_derived(EMPTY_HOST);

/* Creates and initializes a new EMPTY_HOST process.
   Returns pointer to the new process. */
EMPTY_HOST *empty_host_create(char *name);

/* Assigns the host to a controller. Stores controller's
   slot position and CNI address. Registers host's
   interrupt function by the controller. */
void empty_host_attach_controller(
  EMPTY_HOST *empty_host, CONTROLLER *controller);

#endif
```

B.2 File sa1ehost.c

```
/*
 * This software is part of an EU project:
 *  FIT - Fault Injection for TTA
 *  IST-1999-10748
 *
 * TTP/C - protocol v.0.1
 *
 * Design (c) TTTech & TU Vienna
 * Implementation (c) UWB Pilsen & CTU Prague
 *
 * sa1ehost.c
 * EMPTY_HOST C-Sim proces
 *
 * version 0.7 - 2001-04-10
 */

#include "debug.h"
#include <assert.h>

#include "sa1ehost.h"
```

```
#include "kr_csim.h"
#include "fp1cni.h"
#include "sp1cni.h"
#include "sl1func.h"

/* Interrupt handler function - called only by the
   asociated controller */
static void controller_interrupt(void *ptr)
{
  ((EMPTY_HOST *)ptr)->interrupted++;
  process_activate_delay((PROCESS *)ptr, 0);
}

/* Test controllers lifesign - returns 1 on success,
   update host lifesign as well */
static int test_controller_lifesign(EMPTY_HOST *empty_host)
{
  int result;

  cni_set_host_lifesign(
    empty_host->cni,
    ~cni_get_controller_lifesign(empty_host->cni)
      & bit_mask(16));
  result = (cni_get_controller_lifesign(empty_host->cni)
    != empty_host->old_controller_lifesign);
  empty_host->old_controller_lifesign =
    cni_get_controller_lifesign(empty_host->cni);

  return result;
}

/* EMPTY_HOST process - program code */
static s_program(EMPTY_HOST, empty_host_prog)
  for (;;) {
    int i;

    if (cni_get_co(p_my->cni) != CONTROLLER_ON) {
      /* controller is OFF. */

      /* Restart controller */
      cni_set_co(p_my->cni, 0);
      hold(0);
      cni_set_co(p_my->cni, CONTROLLER_ON);
```

```
hold(0);

/* Wait for controller to become ready. */
if (cni_ps_get_controller_ready(
  &p_my->cni->protocol_status) != CONTROLLER_READY) {
  cni_set_co(p_my->cni, 0);
  hold(1000);
  continue;
}

/* Enable Interrupts. */
cni_ie_set_mi(&p_my->cni->interrupt_enable, 1);
p_my->interrupted = 0;

/* Update Host Life-Sign each TDMA round until
   C_State_Valid is set. */
p_my->old_controller_lifesign = 0xFFFF;
hold(0);
p_my->max_iterations = 10;
while (p_my->max_iterations-- > 0
  && cni_get_co(p_my->cni) == CONTROLLER_ON
  && cni_ps_get_c_state_valid(&p_my->cni->protocol_status)
    != C_STATE_VALID) {
  test_controller_lifesign(p_my);
  hold(1000);
}

/* If the C-State is still not valid after 10 TDMA rouds,
   restart. */
if (cni_ps_get_c_state_valid(
  &p_my->cni->protocol_status) != C_STATE_VALID) {
  cni_set_co(p_my->cni, 0);
  hold(1000);
  continue;
}

/* Controller is synchronized, enable the timer
   interrupt correctly. */
i = p_my->index - c_state_get_medl_position(
  &p_my->cni->c_state) % 4;
if (i == 0) {
  /* My sending slot was now */
  test_controller_lifesign(p_my);
  i = 4;
```

```
    }
    else
      i = (4 + i) % 4;
    cni_set_timer1(p_my->cni, c_state_get_time(
      &p_my->cni->c_state) + 250 * i - 150);
    cni_ie_set_ti1(p_my->cni, 1);
    p_my->last_ti1 = CSIM_TIME();
    p_my->startup_complete++;
    continue;
  }

  if (p_my->interrupted) {
    /* Controller is synchronized with the cluster. */
    /* Controller issued an interrupt */

    if (cni_is_get_ti1(&p_my->cni->interrupt_status)) {
      /* Time Interrupt 1: Update Host Lifesign */
      p_my->last_ti1 = CSIM_TIME();
      test_controller_lifesign(p_my);
      cni_set_external_rate_correction(p_my->cni, 0);
      cni_increment_timer1(p_my->cni, 1000);
      cni_is_set_ti1(&p_my->cni->interrupt_status, 0);
    }

    p_my->interrupted = 0;
  }

  /* Timeout for an interrupt to come 2 TDMA rounds */
  hold(2000);

  if (CSIM_TIME() - p_my->last_ti1 > 2000.0) {
    if (cni_ps_get_protocol_state(
      &p_my->cni->protocol_status) == PS_LISTEN ||
        cni_ps_get_protocol_state(&p_my->cni->protocol_status)
          == PS_COLD_START) {
      /* The timeout expired, but the controller is in listen
         or coldstart, that is ok. */
      p_my->last_ti1 = CSIM_TIME();
      test_controller_lifesign(p_my);
    } else {
      /* Timeout expired, assume an error and restart. */
      cni_set_co(p_my->cni, 0);
      hold(1000);
    }
```

```
    }
  } /* for (;;) */
s_end_program

/* Creates and initializes a new EMPTY_HOST process.
   Returns pointer to the new process. */
EMPTY_HOST *empty_host_create(char *name)
{
  EMPTY_HOST *empty_host;

  empty_host = s_new_process(EMPTY_HOST, empty_host_prog);
  set_name(empty_host, name);

  empty_host->interrupted = 0;
  empty_host->startup_complete = 0;

  return empty_host;
}

/* Assigns the host to a controller. Stores controller's
   slot position and CNI address. Registers host's interrupt
   function by the controller. */
void empty_host_attach_controller(
  EMPTY_HOST *empty_host,
  CONTROLLER *controller)
{
  empty_host->cni = &controller->cni;
  empty_host->index = logical_name_get_slot_position(
    &controller->medl.role_list.logical_name) - 1;
  controller_attach_host(controller, (void *)empty_host, controller_interrupt);
}
```

B.3 File sa1eapp.h

```
/*
 * This software is part of an EU project:
 *  FIT - Fault Injection for TTA
 *  IST-1999-10748
 *
 * TTP/C - protocol v.0.1
 *
 * Design (c) TTTech & TU Vienna
 * Implementation (c) UWB Pilsen & CTU Prague
```

```
 *
 * sa1eapp.h
 * Empty Application
 *
 * version 0.1 - 2001-03-17
 */

#ifndef _SA1EAPP_H
#define _SA1EAPP_H

#include "sa1ehost.h"

/* Data structure that holds all Empty application
   components. */
typedef struct empty_app {
  EMPTY_HOST *host[4];
  CONTROLLER_MODEL *controller[4];
  CHANNEL_MODEL *channel[2];
} EMPTY_APP;

/* Creates and initializes all simulation processes for
   the empty application. */
EMPTY_APP *empty_app_init();

/* Deinitializes empty application and releases
   allocated memory. */
void empty_app_done(EMPTY_APP *app);

#endif
```

B.4 File sa1eapp.c

```
/*
 * This software is part of an EU project:
 *  FIT - Fault Injection for TTA
 *  IST-1999-10748
 *
 * TTP/C - protocol v.0.1
 *
 * Design (c) TTTech & TU Vienna
 * Implementation (c) UWB Pilsen & CTU Prague
 *
 * sa1eapp.c
```

```
 * Empty Application
 *
 * version 0.2 - 2001-04-10
 */

#include "debug.h"
#include <assert.h>

#include "sp1cni.h"
#include "fp1mdlrd.h"
#include "sa1eapp.h"
#include "sa1ehost.h"
#include "sl1func.h"


/* Creates and initializes all simulation processes for
   the empty application. */
EMPTY_APP *empty_app_init()
{
  EMPTY_APP *app;
  unsigned i;

  /* The clock drift values for the four controllers. */
  static double controller_drift[4] = {
    - 0.000002,
    + 0.000003,
    - 0.000001,
    + 0.000000
  };

  app = (EMPTY_APP *)malloc(sizeof(EMPTY_APP));;

  /* Initialize the ttpc library. */
  inic_mem();
  cni_module_initialize();

  /* Now create the simulated modules:
     two channels
     four controllers
     four hosts */
  app->channel[0] = channel_create("Channel A");
  app->channel[1] = channel_create("Channel B");
  app->controller[0] = controller_create("Controller 1");
  app->controller[1] = controller_create("Controller 2");
```

```
  app->controller[2] = controller_create("Controller 3");
  app->controller[3] = controller_create("Controller 4");
  app->host[0] = empty_host_create("Host 1");
  app->host[1] = empty_host_create("Host 2");
  app->host[2] = empty_host_create("Host 3");
  app->host[3] = empty_host_create("Host 4");

  /* Initialize the channels, give them a bus ID. */
  channel_set_bus_id((CHANNEL *)app->channel[0], 0);
  channel_set_bus_id((CHANNEL *)app->channel[1], 1);

  /* Initialize the four controllers. */
  for (i = 0; i < 4; i++) {
    /* Attach them to the channels. */
    controller_attach_channel(
      (CONTROLLER *)app->controller[i],
      (CHANNEL *)app->channel[0], 0);
    controller_attach_channel(
      (CONTROLLER *)app->controller[i],
      (CHANNEL *)app->channel[1], 1);
    /* Give them a clock drift. */
    controller_set_clock_drift(
      (CONTROLLER *)app->controller[i],
       controller_drift[i]);
    /* And read the MEDL tables from files. */
    if (medl_read_from_file(
      &((CONTROLLER *)app->controller[i])->medl, i))
      return NULL;
  }

  /* Attach the controllers to hosts. */
  for (i = 0; i < 4; i++)
    empty_host_attach_controller(
      app->host[i], (CONTROLLER *)app->controller[i]);

  /* And activate the C-Sim processes of the hosts. */
  for (i = 0; i < 4; i++)
    process_activate_delay((PROCESS *)app->host[i], 0);

  return app;
}

/* Deinitializes empty application and releases allocated
   memory. */
```

```
void empty_app_done(EMPTY_APP *app)
{
  clear_mem();
  free(app);
}
```

B.5 File medl00.txt

```
#
# This software is part of an EU project:
#     FIT - Fault Injection for TTA
#     IST-1999-10748
#
# TTP/C - protocol v.0.1
#
# Design (c) TTTech & TU Vienna
# Implementation (c) UWB Pilsen & CTU Prague
#
# medl00.txt
# MEDL source file for Empty Application
# Controller 1
#
# version 0.1 - 2001-03-13
#

749   bus_guardian_cold_start.bg_slot_offset
145   bus_guardian_cold_start.bg_slot_duration
65535 bus_guardian_cold_start.bg_remaining_round

749   role_list.bus_guardian.bg_slot_offset
145   role_list.bus_guardian.bg_slot_duration
105   role_list.bus_guardian.bg_remaining_round

0     implementation_params
        .synchronization_period_params.bg_slot_offset
251   implementation_params
        .synchronization_period_params.bg_slot_duration
65535 implementation_params
        .synchronization_period_params.bg_remaining_round

1998  implementation_params
        .listen_cold_start_period_params.bg_slot_offset
148   implementation_params
```

```
        .listen_cold_start_period_params.bg_slot_duration
65535 implementation_params
        .listen_cold_start_period_params.bg_remaining_round


1     role_list.logical_name.slot_position
0     role_list.logical_name.ps
0     role_list.logical_name.multiplex_id

1     controller_cfg_params.cf
0     controller_cfg_params.startup_timeout
3     controller_cfg_params.c_state_length
1000  controller_cfg_params.tdma_round_length
100   controller_cfg_params.micro_to_macro
105   implementation_params.ifg_time

250   slot_control_entry[0].sru_slot_duration
0     slot_control_entry[0].membership_pos
0     slot_control_entry[0].eor
1     slot_control_entry[0].ra
0     slot_control_entry[0].eot
0     slot_control_entry[0].logical_name.ps
1     slot_control_entry[0].logical_name.slot_position
0     slot_control_entry[0].logical_name.multiplex_id
1     slot_control_entry[0].syf
1     mode_address_entry[1][0].ia
1     mode_address_entry[1][0].ib
0     mode_address_entry[1][0].eoc

250   slot_control_entry[1].sru_slot_duration
1     slot_control_entry[1].membership_pos
0     slot_control_entry[1].eor
1     slot_control_entry[1].ra
0     slot_control_entry[1].eot
0     slot_control_entry[1].logical_name.ps
2     slot_control_entry[1].logical_name.slot_position
0     slot_control_entry[1].logical_name.multiplex_id
1     slot_control_entry[1].syf
1     slot_control_entry[1].ba
1     mode_address_entry[1][1].ia
1     mode_address_entry[1][1].ib
0     mode_address_entry[1][1].eoc

250   slot_control_entry[2].sru_slot_duration
```

```
2      slot_control_entry[2].membership_pos
0      slot_control_entry[2].eor
1      slot_control_entry[2].ra
0      slot_control_entry[2].eot
0      slot_control_entry[2].logical_name.ps
3      slot_control_entry[2].logical_name.slot_position
0      slot_control_entry[2].logical_name.multiplex_id
1      slot_control_entry[2].syf
1      mode_address_entry[1][2].ia
1      mode_address_entry[1][2].ib
0      mode_address_entry[1][2].eoc

250    slot_control_entry[3].sru_slot_duration
3      slot_control_entry[3].membership_pos
1      slot_control_entry[3].eor
1      slot_control_entry[3].ra
0      slot_control_entry[3].eot
0      slot_control_entry[3].logical_name.ps
4      slot_control_entry[3].logical_name.slot_position
0      slot_control_entry[3].logical_name.multiplex_id
1      slot_control_entry[3].syf
1      slot_control_entry[3].cs
1      mode_address_entry[1][3].ia
1      mode_address_entry[1][3].ib
0      mode_address_entry[1][3].eoc

250    slot_control_entry[4].sru_slot_duration
0      slot_control_entry[4].membership_pos
0      slot_control_entry[4].eor
1      slot_control_entry[4].ra
0      slot_control_entry[4].eot
0      slot_control_entry[4].logical_name.ps
1      slot_control_entry[4].logical_name.slot_position
0      slot_control_entry[4].logical_name.multiplex_id
1      slot_control_entry[4].syf
1      mode_address_entry[1][4].ia
1      mode_address_entry[1][4].ib
0      mode_address_entry[1][4].eoc

250    slot_control_entry[5].sru_slot_duration
1      slot_control_entry[5].membership_pos
0      slot_control_entry[5].eor
1      slot_control_entry[5].ra
0      slot_control_entry[5].eot
```

```
0     slot_control_entry[5].logical_name.ps
2     slot_control_entry[5].logical_name.slot_position
0     slot_control_entry[5].logical_name.multiplex_id
1     slot_control_entry[5].syf
1     slot_control_entry[5].ba
1     mode_address_entry[1][5].ia
1     mode_address_entry[1][5].ib
0     mode_address_entry[1][5].eoc

250   slot_control_entry[6].sru_slot_duration
2     slot_control_entry[6].membership_pos
0     slot_control_entry[6].eor
1     slot_control_entry[6].ra
0     slot_control_entry[6].eot
0     slot_control_entry[6].logical_name.ps
3     slot_control_entry[6].logical_name.slot_position
0     slot_control_entry[6].logical_name.multiplex_id
1     slot_control_entry[6].syf
1     mode_address_entry[1][6].ia
1     mode_address_entry[1][6].ib
0     mode_address_entry[1][6].eoc

250   slot_control_entry[7].sru_slot_duration
3     slot_control_entry[7].membership_pos
1     slot_control_entry[7].eor
1     slot_control_entry[7].ra
1     slot_control_entry[7].eot
0     slot_control_entry[7].logical_name.ps
4     slot_control_entry[7].logical_name.slot_position
0     slot_control_entry[7].logical_name.multiplex_id
1     slot_control_entry[7].syf
1     slot_control_entry[7].cs
1     mode_address_entry[1][7].ia
1     mode_address_entry[1][7].ib
1     mode_address_entry[1][7].eoc
```

B.6 File medl01.txt

```
#
# This software is part of an EU project:
#     FIT - Fault Injection for TTA
#     IST-1999-10748
#
```

```
# TTP/C - protocol v.0.1
#
# Design (c) TTTech & TU Vienna
# Implementation (c) UWB Pilsen & CTU Prague
#
# medl01.txt
# MEDL source file for Empty Application
# Controller 2
#
# version 0.1 - 2001-03-13
#

999   bus_guardian_cold_start.bg_slot_offset
145   bus_guardian_cold_start.bg_slot_duration
65535 bus_guardian_cold_start.bg_remaining_round

749   role_list.bus_guardian.bg_slot_offset
145   role_list.bus_guardian.bg_slot_duration
105   role_list.bus_guardian.bg_remaining_round

0     implementation_params
        .synchronization_period_params.bg_slot_offset
251   implementation_params
        .synchronization_period_params.bg_slot_duration
65535 implementation_params
        .synchronization_period_params.bg_remaining_round

2248  implementation_params
        .listen_cold_start_period_params.bg_slot_offset
148   implementation_params
        .listen_cold_start_period_params.bg_slot_duration
65535 implementation_params
        .listen_cold_start_period_params.bg_remaining_round

2     role_list.logical_name.slot_position
0     role_list.logical_name.ps
0     role_list.logical_name.multiplex_id

1     controller_cfg_params.cf
250   controller_cfg_params.startup_timeout
3     controller_cfg_params.c_state_length
1000  controller_cfg_params.tdma_round_length
100   controller_cfg_params.micro_to_macro
105   implementation_params.ifg_time
```

```
250   slot_control_entry[0].sru_slot_duration
0     slot_control_entry[0].membership_pos
0     slot_control_entry[0].eor
1     slot_control_entry[0].ra
0     slot_control_entry[0].eot
0     slot_control_entry[0].logical_name.ps
1     slot_control_entry[0].logical_name.slot_position
0     slot_control_entry[0].logical_name.multiplex_id
1     slot_control_entry[0].syf
1     mode_address_entry[1][0].ia
1     mode_address_entry[1][0].ib
0     mode_address_entry[1][0].eoc

250   slot_control_entry[1].sru_slot_duration
1     slot_control_entry[1].membership_pos
0     slot_control_entry[1].eor
1     slot_control_entry[1].ra
0     slot_control_entry[1].eot
0     slot_control_entry[1].logical_name.ps
2     slot_control_entry[1].logical_name.slot_position
0     slot_control_entry[1].logical_name.multiplex_id
1     slot_control_entry[1].syf
1     mode_address_entry[1][1].ia
1     mode_address_entry[1][1].ib
0     mode_address_entry[1][1].eoc

250   slot_control_entry[2].sru_slot_duration
2     slot_control_entry[2].membership_pos
0     slot_control_entry[2].eor
1     slot_control_entry[2].ra
0     slot_control_entry[2].eot
0     slot_control_entry[2].logical_name.ps
3     slot_control_entry[2].logical_name.slot_position
0     slot_control_entry[2].logical_name.multiplex_id
1     slot_control_entry[2].syf
1     slot_control_entry[2].ba
1     mode_address_entry[1][2].ia
1     mode_address_entry[1][2].ib
0     mode_address_entry[1][2].eoc

250   slot_control_entry[3].sru_slot_duration
3     slot_control_entry[3].membership_pos
1     slot_control_entry[3].eor
```

```
1     slot_control_entry[3].ra
0     slot_control_entry[3].eot
0     slot_control_entry[3].logical_name.ps
4     slot_control_entry[3].logical_name.slot_position
0     slot_control_entry[3].logical_name.multiplex_id
1     slot_control_entry[3].syf
1     slot_control_entry[3].cs
1     mode_address_entry[1][3].ia
1     mode_address_entry[1][3].ib
0     mode_address_entry[1][3].eoc

250   slot_control_entry[4].sru_slot_duration
0     slot_control_entry[4].membership_pos
0     slot_control_entry[4].eor
1     slot_control_entry[4].ra
0     slot_control_entry[4].eot
0     slot_control_entry[4].logical_name.ps
1     slot_control_entry[4].logical_name.slot_position
0     slot_control_entry[4].logical_name.multiplex_id
1     slot_control_entry[4].syf
1     mode_address_entry[1][4].ia
1     mode_address_entry[1][4].ib
0     mode_address_entry[1][4].eoc

250   slot_control_entry[5].sru_slot_duration
1     slot_control_entry[5].membership_pos
0     slot_control_entry[5].eor
1     slot_control_entry[5].ra
0     slot_control_entry[5].eot
0     slot_control_entry[5].logical_name.ps
2     slot_control_entry[5].logical_name.slot_position
0     slot_control_entry[5].logical_name.multiplex_id
1     slot_control_entry[5].syf
1     mode_address_entry[1][5].ia
1     mode_address_entry[1][5].ib
0     mode_address_entry[1][5].eoc

250   slot_control_entry[6].sru_slot_duration
2     slot_control_entry[6].membership_pos
0     slot_control_entry[6].eor
1     slot_control_entry[6].ra
0     slot_control_entry[6].eot
0     slot_control_entry[6].logical_name.ps
3     slot_control_entry[6].logical_name.slot_position
```

```
0     slot_control_entry[6].logical_name.multiplex_id
1     slot_control_entry[6].syf
1     slot_control_entry[6].ba
1     mode_address_entry[1][6].ia
1     mode_address_entry[1][6].ib
0     mode_address_entry[1][6].eoc

250   slot_control_entry[7].sru_slot_duration
3     slot_control_entry[7].membership_pos
1     slot_control_entry[7].eor
1     slot_control_entry[7].ra
1     slot_control_entry[7].eot
0     slot_control_entry[7].logical_name.ps
4     slot_control_entry[7].logical_name.slot_position
0     slot_control_entry[7].logical_name.multiplex_id
1     slot_control_entry[7].syf
1     slot_control_entry[7].cs
1     mode_address_entry[1][7].ia
1     mode_address_entry[1][7].ib
1     mode_address_entry[1][7].eoc
```

B.7 File medl02.txt

```
#
# This software is part of an EU project:
#     FIT - Fault Injection for TTA
#     IST-1999-10748
#
# TTP/C - protocol v.0.1
#
# Design (c) TTTech & TU Vienna
# Implementation (c) UWB Pilsen & CTU Prague
#
# medl02.txt
# MEDL source file for Empty Application
# Controller 3
#
# version 0.1 - 2001-03-13
#

1249  bus_guardian_cold_start.bg_slot_offset
145   bus_guardian_cold_start.bg_slot_duration
65535 bus_guardian_cold_start.bg_remaining_round
```

```
749   role_list.bus_guardian.bg_slot_offset
145   role_list.bus_guardian.bg_slot_duration
105   role_list.bus_guardian.bg_remaining_round

0     implementation_params
        .synchronization_period_params.bg_slot_offset
251   implementation_params
        .synchronization_period_params.bg_slot_duration
65535 implementation_params
        .synchronization_period_params.bg_remaining_round

2498  implementation_params
        .listen_cold_start_period_params.bg_slot_offset
148   implementation_params
        .listen_cold_start_period_params.bg_slot_duration
65535 implementation_params
        .listen_cold_start_period_params.bg_remaining_round

3     role_list.logical_name.slot_position
0     role_list.logical_name.ps
0     role_list.logical_name.multiplex_id

1     controller_cfg_params.cf
500   controller_cfg_params.startup_timeout
3     controller_cfg_params.c_state_length
1000  controller_cfg_params.tdma_round_length
100   controller_cfg_params.micro_to_macro
105   implementation_params.ifg_time

250   slot_control_entry[0].sru_slot_duration
0     slot_control_entry[0].membership_pos
0     slot_control_entry[0].eor
1     slot_control_entry[0].ra
0     slot_control_entry[0].eot
0     slot_control_entry[0].logical_name.ps
1     slot_control_entry[0].logical_name.slot_position
0     slot_control_entry[0].logical_name.multiplex_id
1     slot_control_entry[0].syf
1     mode_address_entry[1][0].ia
1     mode_address_entry[1][0].ib
0     mode_address_entry[1][0].eoc

250   slot_control_entry[1].sru_slot_duration
```

```
1      slot_control_entry[1].membership_pos
0      slot_control_entry[1].eor
1      slot_control_entry[1].ra
0      slot_control_entry[1].eot
0      slot_control_entry[1].logical_name.ps
2      slot_control_entry[1].logical_name.slot_position
0      slot_control_entry[1].logical_name.multiplex_id
1      slot_control_entry[1].syf
1      mode_address_entry[1][1].ia
1      mode_address_entry[1][1].ib
0      mode_address_entry[1][1].eoc

250    slot_control_entry[2].sru_slot_duration
2      slot_control_entry[2].membership_pos
0      slot_control_entry[2].eor
1      slot_control_entry[2].ra
0      slot_control_entry[2].eot
0      slot_control_entry[2].logical_name.ps
3      slot_control_entry[2].logical_name.slot_position
0      slot_control_entry[2].logical_name.multiplex_id
1      slot_control_entry[2].syf
1      mode_address_entry[1][2].ia
1      mode_address_entry[1][2].ib
0      mode_address_entry[1][2].eoc

250    slot_control_entry[3].sru_slot_duration
3      slot_control_entry[3].membership_pos
1      slot_control_entry[3].eor
1      slot_control_entry[3].ra
0      slot_control_entry[3].eot
0      slot_control_entry[3].logical_name.ps
4      slot_control_entry[3].logical_name.slot_position
0      slot_control_entry[3].logical_name.multiplex_id
1      slot_control_entry[3].syf
1      slot_control_entry[3].cs
1      slot_control_entry[3].ba
1      mode_address_entry[1][3].ia
1      mode_address_entry[1][3].ib
0      mode_address_entry[1][3].eoc

250    slot_control_entry[4].sru_slot_duration
0      slot_control_entry[4].membership_pos
0      slot_control_entry[4].eor
1      slot_control_entry[4].ra
```

```
0     slot_control_entry[4].eot
0     slot_control_entry[4].logical_name.ps
1     slot_control_entry[4].logical_name.slot_position
0     slot_control_entry[4].logical_name.multiplex_id
1     slot_control_entry[4].syf
1     mode_address_entry[1][4].ia
1     mode_address_entry[1][4].ib
0     mode_address_entry[1][4].eoc

250   slot_control_entry[5].sru_slot_duration
1     slot_control_entry[5].membership_pos
0     slot_control_entry[5].eor
1     slot_control_entry[5].ra
0     slot_control_entry[5].eot
0     slot_control_entry[5].logical_name.ps
2     slot_control_entry[5].logical_name.slot_position
0     slot_control_entry[5].logical_name.multiplex_id
1     slot_control_entry[5].syf
1     mode_address_entry[1][5].ia
1     mode_address_entry[1][5].ib
0     mode_address_entry[1][5].eoc

250   slot_control_entry[6].sru_slot_duration
2     slot_control_entry[6].membership_pos
0     slot_control_entry[6].eor
1     slot_control_entry[6].ra
0     slot_control_entry[6].eot
0     slot_control_entry[6].logical_name.ps
3     slot_control_entry[6].logical_name.slot_position
0     slot_control_entry[6].logical_name.multiplex_id
1     slot_control_entry[6].syf
1     mode_address_entry[1][6].ia
1     mode_address_entry[1][6].ib
0     mode_address_entry[1][6].eoc

250   slot_control_entry[7].sru_slot_duration
3     slot_control_entry[7].membership_pos
1     slot_control_entry[7].eor
1     slot_control_entry[7].ra
1     slot_control_entry[7].eot
0     slot_control_entry[7].logical_name.ps
4     slot_control_entry[7].logical_name.slot_position
0     slot_control_entry[7].logical_name.multiplex_id
1     slot_control_entry[7].syf
```

```
1     slot_control_entry[7].cs
1     slot_control_entry[7].ba
1     mode_address_entry[1][7].ia
1     mode_address_entry[1][7].ib
1     mode_address_entry[1][7].eoc
```

B.8 File medl03.txt

```
#
# This software is part of an EU project:
#     FIT - Fault Injection for TTA
#     IST-1999-10748
#
# TTP/C - protocol v.0.1
#
# Design (c) TTTech & TU Vienna
# Implementation (c) UWB Pilsen & CTU Prague
#
# medl03.txt
# MEDL source file for Empty Application
# Controller 4
#
# version 0.1 - 2001-03-13
#

1449  bus_guardian_cold_start.bg_slot_offset
145   bus_guardian_cold_start.bg_slot_duration
65535 bus_guardian_cold_start.bg_remaining_round

749   role_list.bus_guardian.bg_slot_offset
145   role_list.bus_guardian.bg_slot_duration
105   role_list.bus_guardian.bg_remaining_round

0     implementation_params
        .synchronization_period_params.bg_slot_offset
251   implementation_params
        .synchronization_period_params.bg_slot_duration
65535 implementation_params
        .synchronization_period_params.bg_remaining_round

2748  implementation_params
        .listen_cold_start_period_params.bg_slot_offset
148   implementation_params
```

```
        .listen_cold_start_period_params.bg_slot_duration
65535 implementation_params
        .listen_cold_start_period_params.bg_remaining_round

4     role_list.logical_name.slot_position
0     role_list.logical_name.ps
0     role_list.logical_name.multiplex_id

1     controller_cfg_params.cf
750   controller_cfg_params.startup_timeout
3     controller_cfg_params.c_state_length
1000  controller_cfg_params.tdma_round_length
100   controller_cfg_params.micro_to_macro
105   implementation_params.ifg_time

250   slot_control_entry[0].sru_slot_duration
0     slot_control_entry[0].membership_pos
0     slot_control_entry[0].eor
1     slot_control_entry[0].ra
0     slot_control_entry[0].eot
0     slot_control_entry[0].logical_name.ps
1     slot_control_entry[0].logical_name.slot_position
0     slot_control_entry[0].logical_name.multiplex_id
1     slot_control_entry[0].syf
1     slot_control_entry[0].ba
1     mode_address_entry[1][0].ia
1     mode_address_entry[1][0].ib
0     mode_address_entry[1][0].eoc

250   slot_control_entry[1].sru_slot_duration
1     slot_control_entry[1].membership_pos
0     slot_control_entry[1].eor
1     slot_control_entry[1].ra
0     slot_control_entry[1].eot
0     slot_control_entry[1].logical_name.ps
2     slot_control_entry[1].logical_name.slot_position
0     slot_control_entry[1].logical_name.multiplex_id
1     slot_control_entry[1].syf
1     mode_address_entry[1][1].ia
1     mode_address_entry[1][1].ib
0     mode_address_entry[1][1].eoc

250   slot_control_entry[2].sru_slot_duration
2     slot_control_entry[2].membership_pos
```

```
0      slot_control_entry[2].eor
1      slot_control_entry[2].ra
0      slot_control_entry[2].eot
0      slot_control_entry[2].logical_name.ps
3      slot_control_entry[2].logical_name.slot_position
0      slot_control_entry[2].logical_name.multiplex_id
1      slot_control_entry[2].syf
1      mode_address_entry[1][2].ia
1      mode_address_entry[1][2].ib
0      mode_address_entry[1][2].eoc

250    slot_control_entry[3].sru_slot_duration
3      slot_control_entry[3].membership_pos
1      slot_control_entry[3].eor
1      slot_control_entry[3].ra
0      slot_control_entry[3].eot
0      slot_control_entry[3].logical_name.ps
4      slot_control_entry[3].logical_name.slot_position
0      slot_control_entry[3].logical_name.multiplex_id
1      slot_control_entry[3].syf
1      slot_control_entry[3].cs
1      mode_address_entry[1][3].ia
1      mode_address_entry[1][3].ib
0      mode_address_entry[1][3].eoc

250    slot_control_entry[4].sru_slot_duration
0      slot_control_entry[4].membership_pos
0      slot_control_entry[4].eor
1      slot_control_entry[4].ra
0      slot_control_entry[4].eot
0      slot_control_entry[4].logical_name.ps
1      slot_control_entry[4].logical_name.slot_position
0      slot_control_entry[4].logical_name.multiplex_id
1      slot_control_entry[4].syf
1      slot_control_entry[4].ba
1      mode_address_entry[1][4].ia
1      mode_address_entry[1][4].ib
0      mode_address_entry[1][4].eoc

250    slot_control_entry[5].sru_slot_duration
1      slot_control_entry[5].membership_pos
0      slot_control_entry[5].eor
1      slot_control_entry[5].ra
0      slot_control_entry[5].eot
```

```
0      slot_control_entry[5].logical_name.ps
2      slot_control_entry[5].logical_name.slot_position
0      slot_control_entry[5].logical_name.multiplex_id
1      slot_control_entry[5].syf
1      mode_address_entry[1][5].ia
1      mode_address_entry[1][5].ib
0      mode_address_entry[1][5].eoc

250    slot_control_entry[6].sru_slot_duration
2      slot_control_entry[6].membership_pos
0      slot_control_entry[6].eor
1      slot_control_entry[6].ra
0      slot_control_entry[6].eot
0      slot_control_entry[6].logical_name.ps
3      slot_control_entry[6].logical_name.slot_position
0      slot_control_entry[6].logical_name.multiplex_id
1      slot_control_entry[6].syf
1      mode_address_entry[1][6].ia
1      mode_address_entry[1][6].ib
0      mode_address_entry[1][6].eoc

250    slot_control_entry[7].sru_slot_duration
3      slot_control_entry[7].membership_pos
1      slot_control_entry[7].eor
1      slot_control_entry[7].ra
1      slot_control_entry[7].eot
0      slot_control_entry[7].logical_name.ps
4      slot_control_entry[7].logical_name.slot_position
0      slot_control_entry[7].logical_name.multiplex_id
1      slot_control_entry[7].syf
1      slot_control_entry[7].cs
1      mode_address_entry[1][7].ia
1      mode_address_entry[1][7].ib
1      mode_address_entry[1][7].eoc
```

B.9 Output

```
    0.0000: Controller 1: FREEZE      MV=[0000] NFV=[0000] A=0 F=0
I=0, TI1=0, Timer1=0
    0.0000: Controller 1: INIT        MV=[0000] NFV=[0000] A=0 F=0
I=0, TI1=0, Timer1=0
    0.0000: Controller 1: LISTEN      MV=[0000] NFV=[1111] A=0 F=0
I=0, TI1=0, Timer1=0
```

```
    0.0000: BG - Controller 1: Arm. Open scheduled at  1998.0000.
    0.0000: Controller 1: Starting listen timeout.
    0.0000: Controller 2: FREEZE      MV=[0000] NFV=[0000] A=0 F=0
I=0, TI1=0, Timer1=0
    0.0000: Controller 2: INIT        MV=[0000] NFV=[0000] A=0 F=0
I=0, TI1=0, Timer1=0
    0.0000: Controller 2: LISTEN      MV=[0000] NFV=[1111] A=0 F=0
I=0, TI1=0, Timer1=0
    0.0000: BG - Controller 2: Arm. Open scheduled at  2248.0000.
    0.0000: Controller 2: Starting listen timeout.
    0.0000: Controller 3: FREEZE      MV=[0000] NFV=[0000] A=0 F=0
I=0, TI1=0, Timer1=0
    0.0000: Controller 3: INIT        MV=[0000] NFV=[0000] A=0 F=0
I=0, TI1=0, Timer1=0
    0.0000: Controller 3: LISTEN      MV=[0000] NFV=[1111] A=0 F=0
I=0, TI1=0, Timer1=0
    0.0000: BG - Controller 3: Arm. Open scheduled at  2498.0000.
    0.0000: Controller 3: Starting listen timeout.
    0.0000: Controller 4: FREEZE      MV=[0000] NFV=[0000] A=0 F=0
I=0, TI1=0, Timer1=0
    0.0000: Controller 4: INIT        MV=[0000] NFV=[0000] A=0 F=0
I=0, TI1=0, Timer1=0
    0.0000: Controller 4: LISTEN      MV=[0000] NFV=[1111] A=0 F=0
I=0, TI1=0, Timer1=0
    0.0000: BG - Controller 4: Arm. Open scheduled at  2748.0000.
    0.0000: Controller 4: Starting listen timeout.
    0.0000: BG - Controller 1: Holding: 1998.0000
    0.0000: BG - Controller 2: Holding: 2248.0000
    0.0000: BG - Controller 3: Holding: 2498.0000
    0.0000: BG - Controller 4: Holding: 2748.0000
 1998.0000: BG - Controller 1: Bus access enabled.
 1998.0000: BG - Controller 1: Holding: 148.0000
 2000.0040: Controller 1: Updating controller lifesign
 2000.0040: Controller 1: COLDSTART   MV=[0000] NFV=[1111] A=0 F=0
I=0, TI1=0, Timer1=0
 2000.0040: Channel A: Sending I-frame (duration 76.29, arival at
2076.30)
 2000.0040: Controller 1: Starting cold start timeout.
 2076.2979: Controller 2: Got valid I-Frame on channel 0.
 2076.2979: Controller 2: READY       MV=[1000] NFV=[0111] A=1 F=0
I=0, TI1=0, Timer1=0
 2076.2979: Controller 2: CState-Time = 2250, Cluster-Time = 2076,
Next Action Time = 2250.0033
 2076.2979: Controller 2: External rate correction = 0
```

```
 2076.2979: Controller 2: Updating controller lifesign
 2076.2979: Controller 2: Reconfiguration Enabled
 2076.2979: Controller 3: Got valid I-Frame on channel 0.
 2076.2979: Controller 3: READY       MV=[1000] NFV=[0111] A=1 F=0
I=0, TI1=0, Timer1=0
 2076.2979: Controller 3: CState-Time = 2250, Cluster-Time = 2076,
Next Action Time = 2250.0043
 2076.2979: Controller 3: External rate correction = 0
 2076.2979: Controller 4: Got valid I-Frame on channel 0.
 2076.2979: Controller 4: READY       MV=[1000] NFV=[0111] A=1 F=0
I=0, TI1=0, Timer1=0
 2076.2979: Controller 4: CState-Time = 2250, Cluster-Time = 2076,
Next Action Time = 2250.0040
 2076.2979: Controller 4: External rate correction = 0
 2146.0000: BG - Controller 1: Bus access disabled.
 2146.0000: BG - Controller 1: Holding: 65535.0000
 2249.0045: BG - Controller 1: Arm. Open scheduled at  2998.0045.
 2249.0045: BG - Controller 1: Holding: 749.0000
 2250.0033: BG - Controller 2: Arm. Open scheduled at  2250.0033.
 2250.0033: Controller 2: Hold 0
 2250.0033: BG - Controller 2: Bus access enabled.
 2250.0033: BG - Controller 2: Holding: 251.0000
 2250.0033: Channel A: Sending I-frame (duration 76.29, arival at
2326.30)
 2250.0033: Channel B: Sending I-frame (duration 76.29, arival at
2326.30)
 2326.2972: Controller 1: STARTUP     MV=[1100] NFV=[1111] A=0 F=0
I=0, TI1=0, Timer1=0
 2326.2972: Controller 1: CState-Time = 2250, Cluster-Time = 2326,
Next Action Time = 67786.1356
 2326.2972: Controller 1: External rate correction = 0
 2350.0044: Controller 3: Timer 1: Timer activated
 2395.0028: Controller 2: Performing IAck: SEND       -> CHECK1
 2395.0028: Controller 2: SYF: AT = 2250.00
 2395.0028: Controller 2: SYF on channel A: MTD = 0
 2395.0028: Controller 2: SYF on channel B: MTD = 0
 2395.0028: Controller 2: CSA Stack: [0 0 0 0]
 2395.0028: Controller 2: STARTUP     MV=[1100] NFV=[0011] A=0 F=0
I=0, TI1=1, Timer1=3100
 2395.0028: Controller 2: CState-Time = 2500, Cluster-Time = 2394,
Next Action Time = 2500.0025
 2395.0028: Controller 2: External rate correction = 0
 2395.0040: Controller 4: Performing IAck: OK         -> OK
 2395.0040: Controller 4: SYF: AT = 2250.00
```

```
 2395.0040: Controller 4: SYF on channel A: MTD = 0
 2395.0040: Controller 4: SYF on channel B: MTD = 0
 2395.0040: Controller 4: CSA Stack: [0 0 0 0]
 2395.0040: Controller 4: READY       MV=[1100] NFV=[0011] A=2 F=0
I=0, TI1=1, Timer1=2600
 2395.0040: Controller 4: CState-Time = 2500, Cluster-Time = 2394,
Next Action Time = 2500.0040
 2395.0040: Controller 4: External rate correction = 0
 2395.0044: Controller 3: Performing IAck: OK          -> OK
 2395.0044: Controller 3: SYF: AT = 2250.00
 2395.0044: Controller 3: SYF on channel A: MTD = 0
 2395.0044: Controller 3: SYF on channel B: MTD = 0
 2395.0044: Controller 3: CSA Stack: [0 0 0 0]
 2395.0044: Controller 3: READY       MV=[1100] NFV=[0011] A=2 F=0
I=0, TI1=1, Timer1=3350
 2395.0044: Controller 3: CState-Time = 2500, Cluster-Time = 2394,
Next Action Time = 2500.0045
 2395.0044: Controller 3: External rate correction = 0
 2395.0044: Controller 3: Updating controller lifesign
 2395.0044: Controller 3: Reconfiguration Enabled
 2500.0025: BG - Controller 2: Arm. Open scheduled at  3249.0025.
 2500.0025: BG - Controller 2: Holding: 749.0000
 2500.0045: BG - Controller 3: Arm. Open scheduled at  2500.0045.
 2500.0045: Controller 3: Hold 0
 2500.0045: BG - Controller 3: Bus access enabled.
 2500.0045: BG - Controller 3: Holding: 251.0000
 2500.0045: Channel A: Sending I-frame (duration 76.29, arival at
2576.30)
 2500.0045: Channel B: Sending I-frame (duration 76.29, arival at
2576.30)
 2600.0040: Controller 4: Timer 1: Timer activated
 2645.0021: Controller 2: Performing IAck: CHECK1      -> OK
 2645.0021: Controller 2: SYF: AT = 2500.00
 2645.0021: Controller 2: SYF on channel A: MTD = 0
 2645.0021: Controller 2: SYF on channel B: MTD = 0
 2645.0021: Controller 2: CSA Stack: [0 0 0 0]
 2645.0021: Controller 2: STARTUP     MV=[1110] NFV=[0001] A=1 F=0
I=0, TI1=1, Timer1=3100
 2645.0021: Controller 2: CState-Time = 2750, Cluster-Time = 2644,
Next Action Time = 2750.0018
 2645.0021: Controller 2: External rate correction = 0
 2645.0040: Controller 4: Performing IAck: OK          -> OK
 2645.0040: Controller 4: SYF: AT = 2500.00
 2645.0040: Controller 4: SYF on channel A: MTD = 0
```

```
 2645.0040: Controller 4: SYF on channel B: MTD = 0
 2645.0040: Controller 4: CSA Stack: [0 0 0 0]
 2645.0040: Controller 4: READY      MV=[1110] NFV=[0001] A=3 F=0
I=0, TI1=1, Timer1=3600
 2645.0040: Controller 4: CState-Time = 2750, Cluster-Time = 2644,
Next Action Time = 2750.0040
 2645.0040: Controller 4: External rate correction = 0
 2645.0040: Controller 4: Updating controller lifesign
 2645.0040: Controller 4: Reconfiguration Enabled
 2645.0046: Controller 3: Performing IAck: SEND       -> CHECK1
 2645.0046: Controller 3: SYF: AT = 2500.00
 2645.0046: Controller 3: SYF on channel A: MTD = 0
 2645.0046: Controller 3: SYF on channel B: MTD = 0
 2645.0046: Controller 3: CSA Stack: [0 0 0 0]
 2645.0046: Controller 3: STARTUP    MV=[1110] NFV=[0001] A=0 F=0
I=0, TI1=1, Timer1=3350
 2645.0046: Controller 3: CState-Time = 2750, Cluster-Time = 2645,
Next Action Time = 2750.0048
 2645.0046: Controller 3: External rate correction = 0
 2750.0040: BG - Controller 4: Arm. Open scheduled at  2750.0040.
 2750.0040: Controller 4: Hold 0
 2750.0040: BG - Controller 4: Bus access enabled.
 2750.0040: BG - Controller 4: Holding: 251.0000
 2750.0040: Channel A: Sending I-frame (duration 76.29, arival at
2826.30)
 2750.0040: Channel B: Sending I-frame (duration 76.29, arival at
2826.30)
 2750.0048: BG - Controller 3: Arm. Open scheduled at  3499.0048.
 2750.0048: BG - Controller 3: Holding: 749.0000
 2895.0013: Controller 2: Performing IAck: OK         -> OK
 2895.0013: Controller 2: SYF: AT = 2750.00
 2895.0013: Controller 2: SYF on channel A: MTD = 0
 2895.0013: Controller 2: SYF on channel B: MTD = 0
 2895.0013: Controller 2: CSA Stack: [0 0 0 0]
 2895.0013: Controller 2: STARTUP    MV=[1111] NFV=[0000] A=2 F=0
I=0, TI1=1, Timer1=3100
 2895.0013: Controller 2: CState-Time = 3000, Cluster-Time = 2894,
Next Action Time = 3000.0010
 2895.0013: Controller 2: External rate correction = 0
 2895.0040: Controller 4: Performing IAck: SEND       -> CHECK1
 2895.0040: Controller 4: SYF: AT = 2750.00
 2895.0040: Controller 4: SYF on channel A: MTD = 0
 2895.0040: Controller 4: SYF on channel B: MTD = 0
 2895.0040: Controller 4: CSA Stack: [0 0 0 0]
```

```
 2895.0040: Controller 4: STARTUP     MV=[1111] NFV=[0000] A=0 F=0
I=0, TI1=1, Timer1=3600
 2895.0040: Controller 4: CState-Time = 3000, Cluster-Time = 2894,
Next Action Time = 3000.0040
 2895.0040: Controller 4: External rate correction = 0
 2895.0049: Controller 3: Performing IAck: CHECK1      -> OK
 2895.0049: Controller 3: SYF: AT = 2750.00
 2895.0049: Controller 3: SYF on channel A: MTD = 0
 2895.0049: Controller 3: SYF on channel B: MTD = 0
 2895.0049: Controller 3: CSA Stack: [0 0 0 0]
 2895.0049: Controller 3: STARTUP     MV=[1111] NFV=[0000] A=1 F=0
I=0, TI1=1, Timer1=3350
 2895.0049: Controller 3: CState-Time = 3000, Cluster-Time = 2894,
Next Action Time = 3000.0050
 2895.0049: Controller 3: External rate correction = 0
 2998.0045: BG - Controller 1: Bus access enabled.
 2998.0045: BG - Controller 1: Holding: 145.0000
 3000.0040: BG - Controller 4: Arm. Open scheduled at  3749.0040.
 3000.0040: BG - Controller 4: Holding: 749.0000
 3100.0007: Controller 2: Timer 1: Timer activated
 3143.0045: BG - Controller 1: Node Failure.
 3145.0006: Controller 2: Performing IAck: OK          -> OK
 3145.0006: Controller 2: SYF: AT = 3000.00
 3145.0006: Controller 2: CSA Stack: [0 0 0 0]
 3145.0006: Controller 2: STARTUP     MV=[0111] NFV=[1000] A=2 F=0
I=0, TI1=1, Timer1=4100
 3145.0006: Controller 2: CState-Time = 3250, Cluster-Time = 3145,
Next Action Time = 3250.0003
 3145.0006: Controller 2: External rate correction = 0
 3145.0006: Controller 2: Updating controller lifesign
 3145.0040: Controller 4: Performing IAck: CHECK1      -> CHECK1
 3145.0040: Controller 4: SYF: AT = 3000.00
 3145.0040: Controller 4: CSA Stack: [0 0 0 0]
 3145.0040: Controller 4: STARTUP     MV=[0111] NFV=[1000] A=0 F=0
I=0, TI1=1, Timer1=3600
 3145.0040: Controller 4: CState-Time = 3250, Cluster-Time = 3144,
Next Action Time = 3250.0040
 3145.0040: Controller 4: External rate correction = 0
 3145.0051: Controller 3: Performing IAck: OK          -> OK
 3145.0051: Controller 3: SYF: AT = 3000.01
 3145.0051: Controller 3: CSA Stack: [0 0 0 0]
 3145.0051: Controller 3: STARTUP     MV=[0111] NFV=[1000] A=1 F=0
I=0, TI1=1, Timer1=3350
 3145.0051: Controller 3: CState-Time = 3250, Cluster-Time = 3145,
```

```
Next Action Time = 3250.0053
 3145.0051: Controller 3: External rate correction = 0
 3249.0025: BG - Controller 2: Bus access enabled.
 3249.0025: BG - Controller 2: Holding: 145.0000
 3250.0003: Channel A: Sending I-frame (duration 76.29, arival at
3326.29)
 3250.0003: Channel B: Sending I-frame (duration 76.29, arival at
3326.29)
```