University of West Bohemia

Software project of simulation-based fault injection for TTA

Petr Grillinger

April 2001

Abstract

Software Project of Simulation-based Fault Injection for TTA: The purpose of this document is to make the reader familiar with C-Sim based simulation of TTP/C protocol. In several chapters the reader will learn about the basics of time triggered architecture, about the modular structure of the simulated version of TTP/C protocol, about the problem of new application design and about the fault injection method for experimental validation of the simulated model. An example application running on the simulated cluster is included to demonstrate the basic features of the C-Sim based model of the TTP/C protocol.

Contents

Chapter 1

Introduction

1.1 Purpose of this document

The purpose of this document is to make the reader familiar with C-Sim based simulation of the TTP/C protocol. After reading this document any programmer experienced in C should be able to create a new simulation of TTP/C cluster using the libraries created for the FIT project. This document will also describe some of the tests performed on a simulated cluster, namely software based fault injection into a running controller. Results of such fault injection may be used to validate the model and also to find flaws in the protocol or cluster design. An example application developed for the simulated protocol is included to demonstrate the basic features of the C-Sim based TTP/C model.

This document is structured into the following chapters, each of them covering a different part of the model:

1. Introduction: C-Sim overview and TTA overview.
2. Modular Structure: 2D modular structure, description of modules.
3. Application Development: Host application, MEDL design, Visualization, Cluster composition.
4. Fault Injection: Software based fault injection, model validation.
5. Example application: Simulation of a cluster measuring sine wave signal. User manual.
6. Conclusion

In the appendices there are the sources of the sine wave application and a screen shot of the application interface.

1.2 Text Formatting Conventions

To make the understanding of this document easier several rules defining the font application are used. The following parts of text will be typeset in different fonts:

- Code fragments, variable and function names will by typeset in an unproportional font, e.g. `get_cluster_time()`.
- File names and module names will be printed in a sans serif font, e.g. sa1sinh.c.
- The first use of a new terminus that is explained in the abbreviations section will be emphasized by an italic font, e.g. *MEDL* is a table.

1.3 C-Sim Overview

C-Sim is a simulation tool written entirely in ANSI C. This means it is highly portable and at the same time very efficient. The tool is available in the form of a library or source codes at the University of West Bohemia. For detailed description of its interface as well as several application examples see [CSM99].

The terminology used in C-Sim based simulations is explained in a separate chapter at the end of the document, see section 6.

1.3.1 Library Functions

The following lines contain a list of all C-Sim functions and macros used in the TTP/C model and in this document. They are grouped by the provided functionality and briefly described.

Context switching can not occur inside a function called from the process, so all routines that perform context switching must be implemented as preprocessor macros. After the process is activated again, it continues from the point where it was interrupted.

- `passivate()`: Switches to the next scheduled process.
- `hold(dt)`: Switches to the next process and regains control in `dt` time units.

For *process type* definition (a structure containing all state parameters of a process) and *process program* definition (the function executed during the process life-time) C-Sim introduces two sets of preprocessor macros.

- `derived_process(ptype)`: Beginning of type definition. The new structure may be referred to as `struct ptype`.
- `end_derived(PTYPE)`: Ends the type definition. The new type is available also as `PTYPE`.
- `s_program(PTYPE, pprog)`: Beginning of process program definition. Defines a new function called `pprog`. Data of the current process is referenced with `p_my`.
- `s_end_program()`: Ends the process program definition. A semicolon after this macro causes a compilation error.

The following functions are used only once per simulation session to initialize the library and free resources.

- `inic_mem()`: Initializes the internal state of the library.
- `clear_mem()`: Frees all allocated resources within the C-Sim library.

To manipulate processes within the *calendar* (time ordered list of scheduled processes), the following functions are available.

- `reactivate_at(p, t)`: Inserts process `p` into the calendar, sets its schedule time to `t`.
- `reactivate_delay(p, dt)`: Inserts process `p` into the calendar at current time plus `dt`.
- `cancel(p)`: Removes process `p` from the calendar. The process must be scheduled, not passive or active.

Memory allocation for the process data structure and its initialization is handled by C-Sim. The same applies for other predefined C-Sim types too.

- `s_new_process(proc, prog)`: Creates a new instance of the process type `proc` and assigns the function `prog` as the process program function. Returns pointer to the new process.

At last there are two important functions for simulation control and state checking.

- `step()`: Executes one simulation step.
- `CSIM_TIME()`: Returns the current model time.

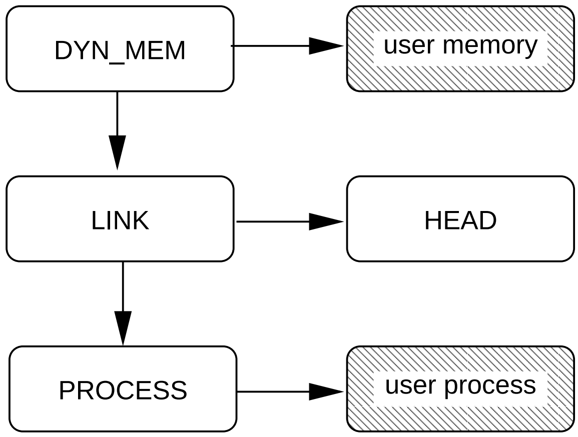

Figure 1.1: C-Sim type derivation

1.3.2 Memory Management

All C-Sim objects are derived from a single type. Reference to every C-Sim object is stored during the object's initialization within an internal list. During the simulation process, this list may be used for debugging purposes, after the simulation terminates, it is used to free all allocated memory. Automated memory disposal provides the ability to restart the simulation at any point without memory leaks. The predefined data types and some possible derivations are depicted in Figure 1.1. The derived types are displayed in gray boxes, the predefined types in the white boxes.

Memory allocation is handled by C-Sim, but the user may take care of it too, although any newly created entity must be first initialized with default values and registered by C-Sim.

Before using any C-Sim function, the library must be initialized with a call to `inic_mem()`.

1.3.3 Process Management

The library provides means for process creation and execution. The processes run interleaved at discrete points of time. To switch context the process must give up control. That is necessary because C-Sim does not use preemptive multitasking but instead is based on coroutines. Context switching is accomplished by a dedicated library call, either `passivate()` or `hold()`.

It is possible to change the scheduling of any process that is currently not active or terminated. To do this three different functions are provided, `reactivate_at()`, `reactivate_delay()`, and `cancel()`.

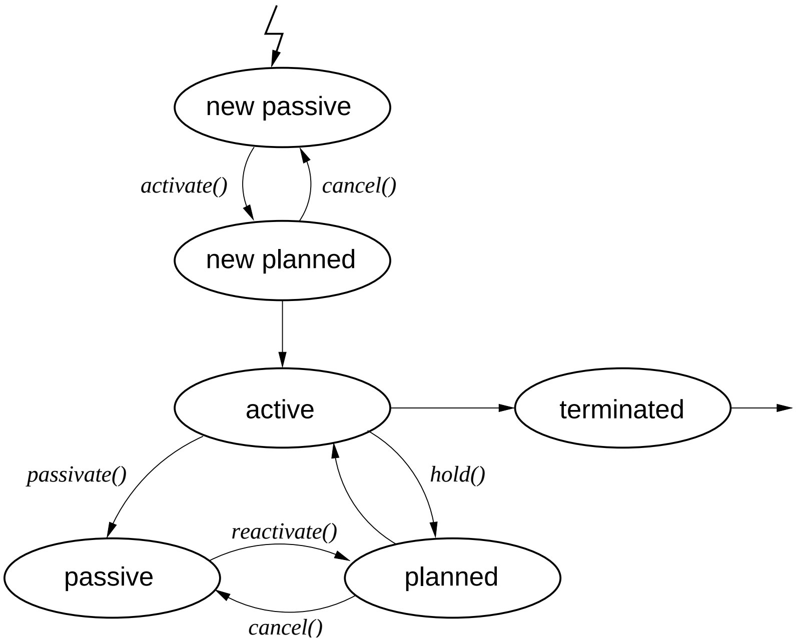

Figure 1.2: C-Sim process state diagram

A C-Sim process as a state machine is depicted in Figure 1.2. Each transition is described by the function that triggered it, or it has no description if the transition resulted from no particular service call.

1.3.4 Process Scheduler

C-Sim uses its own time domain, which we will later in this document refer to as the *model time*. This time does not progress continuously, but in discrete steps. If not said otherwise by *time* is always meant the model time of the simulation.

The model time is represented by a floating point number and may be at any time read with the function `CSIM_TIME()`. The model time has no connection to the real world time, except that it can progress only in the positive direction too.

C-Sim contains a simple process scheduler. We can look at it as a queue of processes, sorted by their execution time, in other words a *calendar*. The first process in the queue is the process that is currently executed. After the process execution stops, the process is removed from the calendar or scheduled to a new time.

The model time is updated each time the scheduler switches processes and is equal to the scheduled time of the current process. Several processes may be scheduled to the same time, but only a single process may be active at a time.

Process switching is implemented with the standard `longjmp` C facility. This solution is portable and very fast but has also one disadvantage, it corrupts the stack. This means, that no local variables, no function parameters and most important no return address is preserved after the process is switched. Due to this limitation no switching may be done inside a function called from the process and the process program must NOT return with the C `return` statement. A safe return from the process program (when the execution reaches the end) is guaranteed by C-Sim if the predefined process creation macros are used.

1.3.5 Process Type

The process type is a C structure that contains C-Sim `PROCESS` specific fields at the beginning and user defined fields at the end. Instances of such type can be overcast to the `PROCESS` type and used in any C-Sim functions.

The preferred way to derive a new process in C-Sim is displayed in the following example. The code defines a new process type and shows two different approaches to define a pointer to the process. The user data included in the new process are an identification number (`id`) and the parameter of a Poisson stream (`lambda`).

```
#include "kr_csim.h"

derived_process(hello_process)
    int id;
    float lambda;
end_derived(HELLO_PROCESS);

struct hello_process  *hello1;
HELLO_PROCESS         *hello2;
```

1.3.6 Process Program

The process program may be any function, that takes exactly one parameter named `p_my` — reference to a variable of the process type. The function must not return with the `return` statement.

C-Sim provides two macros: `s_program()` which creates the header of a process function including the opening brace, and `s_end_program()` which closes the function definition and implements clean termination when the process reaches end of its code.

An example code defining a simple process that prints the process number, the text “Hello World!”, and then waits for a random period of time, explains the basics of process writing. The code extends the example from the previous section.

```
#include <stdio.h>

s_program(HELLO_PROCESS, hello_program)
    for (;;) {
        printf("%d: Hello World!\n", p_my->id);
        hold(negexp(p_my->lambda));
    }
s_end_program
```

1.3.7 Process Creation

The creation of a new process requires two steps. In the first step we allocate memory for the process and assign a process program to the new instance. In the second step we must initialize all user defined attributes of the process, in our example the `id` and `lambda` parameters. Setting of the predefined name attribute is optional, but simplifies debugging of the simulation. It is handy to use a single function that creates and initializes all processes of one type.

The following example extends the code written in the previous section by a function for process creation.

```
HELLO_PROCESS *new_hello_process(int id, float lambda)
{
    HELLO_PROCESS *process;

    process = s_new_process(HELLO_PROCESS, hello_program);
    set_name(process, "hello process");
    process->id = id;
    process->lambda = lambda;

    return process;
}
```

1.3.8 Simulation Run

In the previous sections we have learned how to write a new process and how to initialize it. Now it is time to run the processes in a simple simulation.

The simulation can be run only from the main function of a program or thread. To execute one step of simulation the `step()` function is used.

In the following example we make use of all the previous examples. We create two instances of `HELLO_PROCESS` and schedule the new processes to activate immediately (in zero time). Then we will execute single simulation steps until the model time reaches the value of 1000. The `lambda` parameters of both processes are different so they will have different activation periods.

```
void main()
{
    inic_mem();           /* C-Sim initialization */

    /* Process creation */
    hello1 = new_hello_process(1, 0.01);
    hello2 = new_hello_process(2, 0.001);

    /* Process activation */
    activate_at((PROCESS *)hello1, 0.0);
    activate_at((PROCESS *)hello2, 0.0);

    /* Simulation loop: execution of single steps */
    while (CSIM_TIME() < 1000.0) {
        printf("%6.2f: ", CSIM_TIME());
        step();
    }

    clear_mem();    /* C-Sim memory disposal */
}
```

The output of our example looks like this:

```
  0.00: 1: Hello World!
  0.00: 2: Hello World!
  0.01: 1: Hello World!
 27.58: 1: Hello World!
 83.71: 1: Hello World!
583.89: 1: Hello World!
772.76: 1: Hello World!
774.41: 1: Hello World!
835.45: 1: Hello World!
868.76: 1: Hello World!
881.90: 1: Hello World!
983.28: 2: Hello World!
```

1.4 TTA Overview

In this section the most important features of *Time Triggered Architecture* (TTA) will be introduced together with the basic ideas of fault tolerant (FT) Real-Time (RT) systems. For sa detailed description of the TTP/C protocol see [TPS99], for a description of time triggered systems see [KOP97].

The TTP/C protocol provides the following properties, that are important for safety critical real-time applications:

- Synchronized global time
- Collision free medium access
- Predictable delivery time with minimal latency
- Node membership information
- Fail silence of nodes
- Various modes of operation
- Node replication: Shadow nodes
- Multiplexed nodes
- Replicated channels

1.4.1 TTP/C Node

A single TTP/C node consists of multiple parts as depicted in Figure 1.3. The host layer contains a microprocessor and RAM memory for execution of applications. The host can communicate with the controller part of the node only through the *Communication Network Interface* (CNI). The structure of CNI is strictly defined in the written specification, see [TPS99].

The controller part handles all message transmissions and ensures correctness in the time domain. The *Bus Guardian* (BG) unit can disable or enable write access to the replicated channels. It is independent and ensures fail silence on the bus if the controller fails. Temporal behavior of the node is described by *Message Descriptor List* (MEDL). The MEDL is stored in a ROM memory inside the controller, the host layer has no access to this memory.

A TTP/C node is considered the *Smallest Replaceable Unit* (SRU) in a TTP/C cluster. The mechanism for node reintegration allows to easily replace damaged SRUs without the need to re-test the whole cluster.

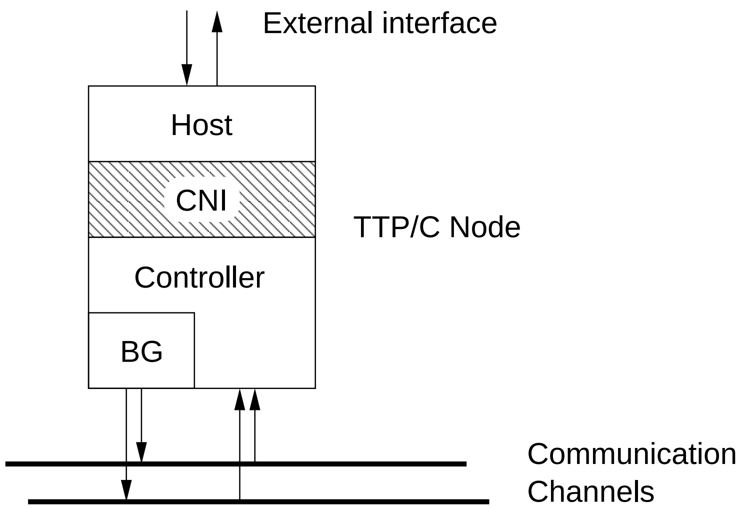

Figure 1.3: TTP/C Node

1.4.2 Time Units

Each node has its own local clock which measures the time in the units of *microticks*. Different nodes may have different microtick durations.

The microticks are used to measure fine time differences, as the granularity of the clock is very small. The node contains also a *macrotick* counter. Macroticks are derived from microticks, but the duration of one macrotick is the same in all nodes within a cluster. This unit is used for the global synchronized time.

1.4.3 TTP/C Cluster

A TTP/C cluster consists of several TTP/C nodes, the number of which can range from two to sixty four. The nodes are identical and can be easily replaced. Multiple nodes can form a fault tolerant unit (FTU). All SRUs are interconnected through two independent communication channels. An example of a small TTP/C cluster is depicted in Figure 1.4.

The state of the cluster (*C-State*) can be described by three values. In each TDMA round, the nodes check if their states agree. The C-State is stored in the CNI of each controller and contains the following information:

- Current mode number and MEDL position.
- Time of the next transmission in macroticks.
- Current membership vector.

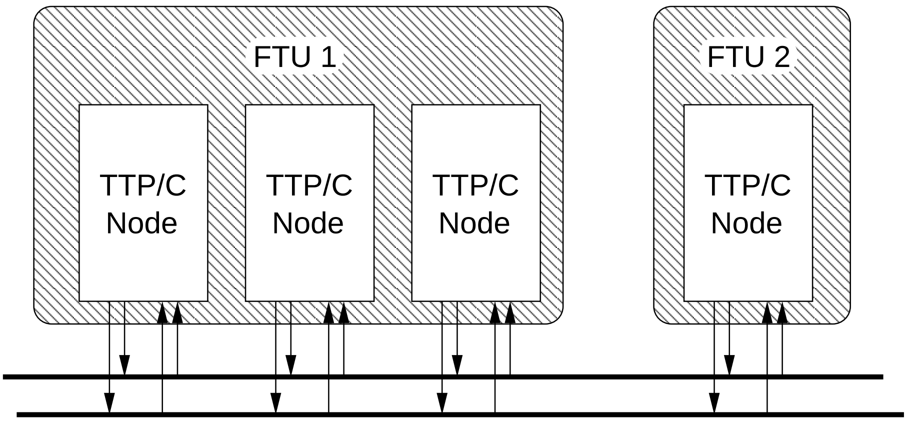

Figure 1.4: TTP/C Cluster

Each cluster must contain at least one node that is able to perform *cold start* of the cluster (startup when no SRU is sending frames). At least four nodes are required for a fault tolerant operation.

1.4.4 Node Host Layer

The core of the host layer of a controller is formed by a microprocessor and RAM memory. Internal memory of this subsystem is used to store application code and run-time data. Fault tolerance in the value domain can be achieved at host layer level, creating a FTU layer. This fault tolerance is provided by a dedicated operating system, *TTP OS*, but can be solved also at the application level. It ensures permanence of messages and if multiple replicated nodes are available, it provides fault tolerance in the value domain.

Access to the communication layer is possible only through the CNI interface. The application, the OS, and the MEDL table are transferred to the node during startup via a dedicated download mode.

1.4.5 Node CNI Interface

The CNI interface is represented by a *DPRAM* (Dual Port Random Access Memory) module. The memory contains at run time all values required for the communication between the host and communication layers. The layout of the fields is strictly specified to ensure compatibility, see [TPS99].

Basically there are three distinct areas in the CNI. They differ in their functionality:

- Status Area: Read only by the host, written by the controller.

- Control Area: Both controller and host can write in this area.
- Message Area: Both controller and host can write in this area.

The status area may not really be write protected, but any change made by the host should have no effect on the controller.

To precede data inconsistency errors, the protocol defines a *Non-Blocking Write Protocol* (NBW). This protocol ensures, that the host will not read or write values in the CNI when the controller is updating and that the host can detect an update of the CNI while it was reading.

1.4.6 Node Interrupt Line

To get asynchronous notification from the controller, a single interrupt line crossing the CNI is implemented. The host application or the TTP OS can specify a mask for the events they want to be notified of. There are three groups of interrupts:

- Time interrupts: Two 16-bit macrotick timers set by the host layer.
- User interrupts: Message slots specified at design time.
- State change interrupts: Change of the membership or operation mode.
- Error interrupts: Any fault condition, e.g. synchronization error.

Note: The first time interrupt is used by TTP OS for task scheduling.

1.4.7 Node Communication Controller

The communication controller handles several tasks that enable us to separate it into several functional layers:

Reconfiguration Layer

The reconfiguration layer handles two main tasks:

- Cluster cold start — when no nodes are sending.
- Reconfiguration into a running cluster — when any transmission on the bus is detected.

The cold start of a cluster is possible only if there is no traffic on any of the replicated channels for a specified period of time. Then a single initialization frame (*I-Frame*) containing the new C-State is transmitted.

The reconfiguration process is simple, as the state of the cluster is sent periodically on the bus in I-Frames. Normal frames (*N-Frames*) containing application data cannot be used for reconfiguration.

SRU Layer

The SRU layer handles three main tasks:

- SRU membership.
- Clock synchronization.
- Implicit acknowledgment.

The SRU membership information is stored in the membership vector in the CNI. A node is considered active and the corresponding bit in the vector is set if the node transmitted in its last slot at least one correct frame, otherwise its membership is cleared.

As all messages are sent at a priori known points in time and the nodes should be synchronized, the drift of local clock can be measured be the time difference between the expected and actual time of message arrival. This time difference is used to correct the local clock.

Implicit acknowledgment mechanism makes use of the C-State. It is contained in the I-Frames explicitly, and in the N-Frames in the CRC. If a node checks the correctness of a received frame it always compares its C-State with the C-State of the sender. If the C-State disagrees, the frame is not considered to be correct. As the membership vector is also a part of C-State, the sending node can determine from the next message how its own message was received.

Data Link/Physical Layer

The data link layer uses the *Time Division Multiple Access* (TDMA) strategy when sending on the bus. Every node in the cluster has a statically assigned time slot, where it can send, and no other node may send in the same slot. The messages are sent on two replicated channels. An example with two TDMA rounds, each with four slots assigned to nodes 'A', 'B', 'C', 'D' is depicted in Figure 1.5.

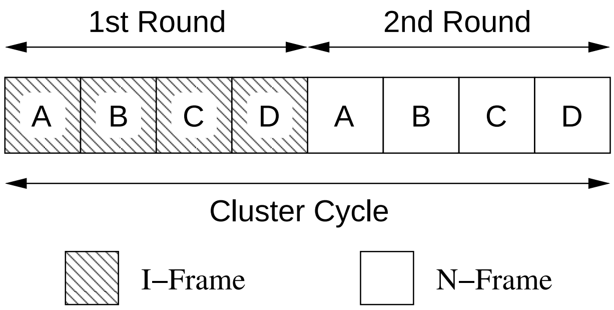

Figure 1.5: Cluster Cycle Example

The write access to the bus is restricted by the Bus Guardian by default. To enable it, the BG must be programed with bus access parameters and triggered before any write operation.

The physical layer provides bit encoding and bit synchronization. In the simulation this layer is not required.

1.5 Safety Critical Protocols

For safety critical protocols there are two possible variants, the *Event Triggered* (ET) and the *Time Triggered* (TT) approach. The description of both possible solutions and a comparison of their suitability for safety critical system will be presented in the next few sections.

1.5.1 Time Triggered Protocols

In a TT system, the messages are sent at predefined points in time. This enables to design a collision-free access schedule for message sending. By eliminating transmission collisions a TT protocol is able to deliver messages within a predictable time with minimal latency.

If we use the TDMA medium access strategy, we can guarantee a full transmission bandwidth for a single transmission, but the message length is restricted. The bus may be accessed only at specified points in time and the access pattern is typically periodical.

It is clear that a TT protocol is suitable for systems that rely on periodical control messages that must be delivered with minimal latency. Control messages are typically short and enable to design the access schedule (TDMA round) with small slots and higher frequency.

A TT protocol will be less efficient if the message sent are not periodical, but occur at random time. Also the transmission of large blocks of data is not well suited for a TT system, as the slot size must remain small to maintain a small latency time. The overhead of splitting a large message into smaller chunks would decrease the protocol efficiency significantly.

The efficiency of a TT protocol is typically higher than the efficiency of a ET protocol, as the messages does not need to contain any identifier or sender and receiver addresses. In TTP/C the sending schedule of each node is defined at design time and the receiver can decide the message type by the time of its reception.

A TT protocol is able to handle overload scenarios much better than current ET protocols as each message has its sending slot already allocated. In addition the TTP/C protocol allows mode changes to accommodate extreme external conditions.

1.5.2 Event Triggered Protocols

ET protocols, e.g. Ethernet or CAN, can send messages at any point of time. This brings the possibility of transmission collisions on the bus. There are two basic approaches how to handle collisions on the bus.

- *Collision Detection* (CD) — the sender listens to its own transmission and when it receives garbage data, the transmission is canceled. Then the protocol typically waits some time and tries to send the message again. The wait duration depends on the protocol, it can be a random value (The LON protocol), or a predefined one (Ethernet). Multiple transmission failures are solved by choosing different wait cycles. A typical example of CD strategy is the Ethernet protocol.
- *Collision Avoidance* (CA) — this strategy is used by the CAN protocol. Each message is equipped with unique priority identifier and only the message with the highest priority is transmitted. This is ensured by a hardware mechanism called *Bit Arbitration*. This mechanism requires that there is a dominant state on the bus (e.g. '0') which always overrides the other state. This condition limits the maximum transmission speed of the physical medium. The sender must check during the priority transmission if his priority won the bit arbitration.

For real time safety critical systems, protocols with CD strategy are not very useful as they introduce an unpredictable message transmission latency. The CA strategy is much more suitable as the latency can be predicted in some cases.

A CAN based network uses 11-bit arbitration field which denotes priority. If the system is well designed, i.e. all safety critical messages are correctly identified and have carefully assigned priorities, the delivery latency is very small. The messages are protected by a CRC code and so error detection is possible. Under normal circumstances, i.e. the load of the system is within the predicted bounds, the latency time remains low and the protocol efficiency is high.

The ET protocols experience problems when the external conditions move out of bounds and the system is overloaded with messages. Then even safety critical messages may be often delivered with a large unpredictable latency.

1.5.3 Combined Protocols

A protocol combining the ET properties with the TT approach is the Byteflight protocol. This protocol uses minislotting strategy when accessing the bus. The minislots and the messages that are sent are determined at design time. At run time empty slots are detected and assigned to the applications. Free bandwidth may be assigned to non-periodical application data. The protocol defines a SYNC signal for application synchronization. The allocation of minislots is handled by a bus-master unit, creating a dangerous single failure point in the protocol.

The messages are protected by a CRC code, but no acknowledgment is possible if not included at design time.

The Byteflight protocol provides sufficient efficiency by sending periodic messages while remaining flexible for sporadic message transmission. It is well suited for safety critical applications although it not so fault tolerant as the TTP/C protocol.

1.5.4 TTP/C versus Other Protocols

The TTP/C protocol is designed for hard real-time FT systems. We can compare its suitability for those system with other protocols designed for the same task, i.e. CAN and Byteflight. The results of such comparison are listed in Table 1.1. A more detailed survey can be found in [TPC00] or in [KOP97].

Summary

CAN is a flexible, inexpensive protocol that can run on most architectures. It is widely supported, but it is not suitable for FT system as it provides no means for verification of the provided fault coverage and in peak-load scenarios the protocol becomes unusable. The protocol does not provide many features that would increase the fault tolerance of a system in comparison to other CSMA strategies.

The Byteflight protocol offers a very flexible bus access strategy and adds the possibility of clock synchronization. For FT systems it is much more suitable than than the CAN protocol, although it is more expensive. Still there are a few setbacks that collide with the required fault tolerance. The protocol contains non-replicated parts so a single failure may cause a system failure. Also no acknowledgment mechanism is provided to detect node errors. The fault tolerance of a system based on Byteflight relies heavily on application design.

The TTP/C protocol provides the best fault coverage of the three protocols. It is not so flexible in bandwidth allocation, as all messages must be assigned statically at design time. A system based on TTP/C protocol may run on a wide range of physical layers making it a inexpensive solution. Every critical part of hardware is replicated and fault detection mechanisms are supported by hardware. Fault tolerant properties of a system can be tested at design time. Parts of the protocol have been formally verified. There are also some reasons why not to use TTP/C: The system is static (everything must be defined at design time) and as a relatively new protocol may still contain undetected design flaws.

Protocol property	CAN	Byteflight	TTP/C
Access strategy	CSMA/CA	Minislotting	TDMA
Collisions	no	no	no
Data rate	max 1Mbit/s	any	any
Network	bus	bus or start	bus or star
Synchronized time	no	yes	yes
Global time	no	no	yes
Eternal clock sync.	no	no	yes
Redundant channels	no	no	yes
Node Membership	no	no	yes
Acknowledgment	explicit	no	implicit
Composability	no	application dependent	yes
Efficiency	max 30%	max 54%	max 80%
Latency	increases with load	increases with load	constant
Message overhead	min 50%	min 40%	min 15%
CRC Check	yes	yes	yes
Bandwidth distribution	dynamic	partial	static
Bus access protection	no	partial	full
FT expansion	difficult	difficult	easy if predicted
Mode changes	no	2 modes	30 modes
Failure a node	unhandled	unhandled	fail-silence
Replicated nodes (TMR)	no	no	yes
Shadow nodes	no	no	yes
Local error detection	no	no	lifesign, BG, NBW

Table 1.1: Protocol Comparison

Chapter 2

Modular structure

2.1 Design Goals

The goal when writing the simulation of TTP/C protocol was not only to retrieve a working model but also to gain an exact C-code specification of the protocol. To separate functional specification and simulation code, the modular structure must be designed to contain only links to the lower levels of implementation.

In the first stage we will concentrate on developing an exact and easy to understand C specification. No simulation tools will be used and the specification will not be executable. An output of the first stage will be a stand-alone library with header files exporting the interface.

The second stage of development will be the writing of a C-Sim extension of the C specification. The simulation will connect the C-Sim library and the previously written functional specification. Result of the combination will be a library again and it will not be executable.

To produce an executable simulation will be the goal of stage three. A host application will be created and linked together with the simulation library. The result will be executable but to add any output capability to the model, we will have to choose some visualization tool. The output can be graphical, using some visual programming language, or console oriented. Logging to a file is also considered a valid output.

In the last stage we will proceed to the experimental validation of the model and black box testing of TTP/C Fault tolerant properties. The method of simulation based fault injection will be used.

2.2 General Rules

To maintain a project of a certain size some rules have to be set an obeyed. The sources must be well documented and consistently formatted so other people might be able to use and modify them. In the next few paragraphs the rules applied to this particular project are listed and explained. Any modifications to the project should follow these rules to preserve a clear structure and readability.

All modules contained within the project are written in C or C++, so the rules will not include any other languages and always refer to C or C++.

2.2.1 Naming Convention

In the sources we will distinguish the following constructs by their case:

- Type names: All letters are upper case, e.g. `PROCESS`.
- Variable names: All letters are lower case, e.g. `tmp`.
- Function names: All letters are lower case, the name ends with parentheses, e.g. `get_cni_co()`.

All exported identifiers must be descriptive and multiple words will be separated by underscores. Each exported identifier should be preceded by the name of the module that exports it to avoid name collisions. The identifier should not contain more than 30 characters.

Local identifiers should be shorter, less than 20 characters in general. For loop control variables single letter identifiers should be used.

Example: A global function defined within the `cni` module returning the value of "cluster time" should be called like this: `cni_get_cluster_time()`.

2.2.2 Code Formatting

Code formatting, also called pretty printing, is very important for the readability of the source codes. The formatting covers indenting of blocks, parentheses placement and white space usage. There are already some well defined coding styles, e.g. K&R style, GNU coding style, or Linux Kernel coding style. For the purposes of this project the coding style used in Linux kernel sources will be applied. The basic rules are:

- Indenting of blocks is one tabulator (not 8 spaces).

- Opening parenthesis of a function definition is on separate line.
- Opening parenthesis of a block command is on the same line.
- Closing parenthesis is on separate line.
- White space is administered as needed to increase readability.
- One line may contain only one statement.
- Between the function name and the left parenthesis is no space.
- Operators are separated by one space.
- Functions should not have side effects and do only one thing.
- Function code should be short, less than thirty lines optimally.
- Block commands should not be nested deeper than three levels.

To enforce this coding style the utility `indent` may be used with following parameters: `indent -kr -i8`. The `-kr` parameter stand for K&R style and the `-i8` parameter stands for eight characters wide indentation.

2.2.3 Comments

Comments are a substantial part of the source code. In the interface section they serve to describe the exported functionality, inside the function definition they explain difficult parts of the algorithm. The rules regarding commenting in this project are the following.

Every module must contain a header describing briefly what the module is for, what version it is and who made it. The header's structure was fixed for all modules in the project to this format:

```
/*
 * 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
 *
 * file name
 * module description
```

```
 *
 * version x.x - year-month-day
 */
```

In the header file every exported symbol must be commented. For variables and types the comment would describe purpose or contents, for functions also parameters, return values and side effects.

In function implementations difficult parts should also be documented.

2.2.4 Version management

To cover all possibilities we require three different versioning schemes. First is the version of the modeled protocol, here the version 0.1. Second is the version of the whole project, which is 1.0. Third are the versions of every file in the project. The file version is changed with every update since last archiving, so mostly this version can change once a day.

2.3 Modular Structure of the Model

There are two possible applications of the TTP/C model. The first is the exact C-written functional specification, the second is an executable C-Sim based model of a real TTP/C cluster. The simulated TTP/C version depends on the functional specification, but not vice versa. The pure C specification must be separable from this particular application and portable to other environments.

The functional and simulation layer will always be portable to other platforms, as they use only ANSI C. However we need also some visualization environment which will rarely be portable and so should be separated.

To achieve a clear structure we need to separate the modules into three layers. To ease the recognition of a module layer, the layer will be specified within the file name. Each layer modules will have different starting letters.

- Functional Layer: All module names begin with the letter '`f`'.
- Simulation Layer: All module names begin with the letter '`s`'.
- Visualization Layer: All module names begin with the letter '`v`'.

This structure reflects the portability of the modules. The first layer uses only ANSI C, the second requires C-Sim, and the third may be platform dependent.

These layers contain mixed modules providing general cluster functionality and providing application specific tasks, e.g. host applications and fault injector. We have to split the reusable general modules and the application dependent modules to be able to create a decent library. In order to do this we will need another module structure, different from the one described above.

In the new structure, there will be also three layers: one describing the protocol specification, one for any application modules, and one for modules used by the protocol or simulation, but outside the scope of the protocol. Again the modules will have file names with different starting letters.

- Library Layer: All modules begin with the letter 'l'.
- Protocol Layer: All module names begin with the letter 'p'.
- Application Layer: All module names begin with the letter 'a'.

It is clear, that the criteria for module separation here is the functionality provided by the modules.

Now we have two separate layering systems which we have to join together. The result of this join will be a two dimensional modular structure. An overview of this structure is depicted in Table 2.1. The table shows all possible combinations of the two layering systems and also suggested naming convention for the modules. All module names must start with two letters, one for each layering system. Moreover the third letter must be a number specifying the version of the simulated TTP/C protocol, in this case '1' for version 0.1.

Layer name	Libraries	Protocol	Application
Functional	fl1	fp1	fa1
Simulation	sl1	sp1	sa1
Visualization	vl1	vp1	va1

Table 2.1: 2D Modular structure and module naming

Because there are operating systems that do not support names longer than eight characters, the module names should not exceed this limit. That leaves five characters for module names and that should be enough. As the used language is C, the extensions of the modules will be '.c for the module implementation and '.h' for module interface. For visualization purposes the implementation part could also reside in '.cpp' files, as the C++ Builder does not properly link C code together with C++ code.

In the next section each layer will be described together with all modules contained within the layer. Possible additions to the layers will also be mentioned.

2.3.1 Functional Libraries (fl)

Modules in this layer contain no simulation specific tasks and are written in pure ANSI C. The purpose of these modules is not to deliver TTP/C protocol functionality, but to extend the standard C libraries for further use by higher layers.

fl1func provides preprocessor macros for definition of access functions to structure members. These macros are used in all higher layers to provide an unified access method to any fields exported by a module. It provides also functions extending the standard C library, e.g. CRC calculation.

fl1debug provides unified debugging support for the whole project. It exports logging functions and various preprocessor macros. For common application of the model it is not neccessary.

2.3.2 Functional Protocol Specification (fp)

This layer specifies the TTP/C protocol properties. It defines the data structures used within a cluster and also specifies the behavior of the cluster. No simulation code can be used and only ANSI C is permitted.

fp1busg specifies the data structure used to contain the internal state of the Bus Guardian unit and defines the unit's function. Actions over the unit can be performed only by calling exported functions.

fp1chan contains the specification of a single bus channel. Means for sending and receiving of frames are provided. No physical layer simulation is performed.

fp1cni defines a structure mirroring the CNI memory area. Access to particular fields in the structure is provided by access functions. The module must be initialized by a call to initialize function before its first use.

fp1const is designed to contain and export any global constants used in the TTP/C protocol specification. This module does not have any implementation part.

fp1cstat exports the definition of a structure containing the C-State and provides some actions over the C-State, e.g. comparing.

fp1ctrl is the definition of the the communication controller of a TTP/C node. The module exports a structure defining the internal structure of the controller and functions executing various tasks in the controller.

fp1frame defines two frame types, the I-frame and the N-frame. Also provides an union of these two frame types for easier manipulation and unite CRC checking.

fp1ldata is meant to contain all internal data used by the controller, but not specified in the TTP/C protocol specification [TPS99]. Usually such data are implementation specific and would change within another application of the protocol. Direct access to this structure is possible, no access functions are required.

fp1lname defines the logical name structure used by the MEDL and CNI. The module exports also access and compare functions for the defined structure.

fp1mdlrd exports functions for reading of MEDL tables from a text file. This is not exactly protocol specified operation, but the process requires knowledge of the TTP/C protocol and is not simulation or application specific. No data structures are exported, only the reading function. This module serves as replacement for the missing download mode.

fp1medl contains the definition of a structure containing the MEDL table of a controller. Besides this structure also the access functions to its fields are exported.

2.3.3 Functional Application Specification (fa)

This layer may in fact contain only a single module, as each application is different. The only common base is the TTP OS. As in other functional layers, only ANSI C is recommended and no simulation tool may be used.

fa1os provides the protocol specified data structures and functionality of TTP OS. In the current implementation this module is not present and the whole TTP OS is contained within the simulation layer. This may change in the future, but with no or very small impact on application developing.

2.3.4 Simulation Libraries (sl)

Modules in this layer extend the functionality of the C-Sim simulation environment. For maximum portability ANSI C is necessary also in the simulation layer. The simulation tool used will be C-Sim, no other libraries may be used.

sl1func extends the basic C-Sim process control functions. In future versions of C-Sim this module will probably become obsolete.

2.3.5 Simulated Protocol Specification (sp)

In this layer all of the modules from fp layer which provide temporal activity will be enhanced by C-Sim simulation control. For each of these modules a C-Sim process must be created. This layer may use only functional TTP/C specification, ANSI C and the C-Sim library.

sp1busg defines the process for execution of the Bus Guardian state machine. The new data structure used by the simulation is wrapped around the functional model. The module provides means for creation, initialization and automatic destruction of the process.

sp1chan adds transmission duration simulation to the functional model of a bus channel. The channel itself is represented as a C-Sim process and the module exports functions for its creation, initialization and automatic disposal.

sp1cni implements some of the access functions to the CNI in a different way, e.g. the time field is not stored, but instead is derived directly from the model time. This module does not contain any C-Sim process.

sp1ctrl defines the process executing the controller's state machine. The process type is wrapped around the functional specification and adds new simulation specific fields. The module contains functions for creation, initialization and automatic disposal of the C-Sim process.

2.3.6 Simulated Application Specification (sa)

In this layer there are all application specific modules which use the C-Sim tool. They may import the functional specification of the protocol, a prepared functional specification of the application or TTP OS and application specific modules. Only ANSI C and C-Sim should be used to ensure maximum portability.

An application typicaly consists of a cluster definition module and one or more task definition modules. The appliacation can be either standalone or built upon the base of TTP OS simulation.

sa1os contains the process simulating the TTP OS. The module uses the functional specification of the OS together with the C-Sim library and must be pure ANSI C. The description of TTP OS is not a part of this document.

2.3.7 Visualization Libraries (vl)

The libraries used for visualization can be also the standard C libraries, when only console output or file logging is used. Such a visualization should use only ANSI C to ensure complete portability even at visualization level.

Other visualization environments may be used also, but at cost of portability. In the current implementation of the simulation, the C++ Builder 5.0 is being used. Any other tool may be used instead.

No visualization libraries are included in this TTP/C simulation. The visualization tool used in the examples is commercial software and undergoes a limiting license agreement.

2.3.8 Visualized Protocol Specification (vp)

This layer could contain modules for monitoring of the TTP/C cluster simulation. Because of the visualization it would be most probably non-portable to other platforms. So far no modules for the protocol visualization have been prepared.

By developing a set of components ready to use in any simulation, new cluster configurations and output visualization could be greatly simplified and the development process speeded up.

2.3.9 Visualized Applications (va)

The visualization of the application running on a TTP/C cluster is an important final step in the model development. This layer would contain application specific modules, using the simulation enhanced protocol specification and prepared application simulations. For simulation flow control the C-Sim library must be used.

This layer contains only modules with an example application. It would be possible to create components for the visualization of the state of TTP OS as the OS module is included in most applications.

Chapter 3

Developing New Applications

3.1 Overview

In this chapter a brief recapitulation of the current state of the model implementation and also a comparison with the hardware implementation will be presented. The problem of new application design will be explained in detail. The description of an example application demonstrating the model possibilities can be found in chapter 5.

3.1.1 TTP/C Silicon Implementation

The host layer of a TTP/C node consists of a processing unit and two interfaces to the lower layers. The processing unit (in current implementation a Motorola processor) is able to execute only one process at a time. The interfaces to the lower layers are the CNI and a single interrupt line. The CNI consists of two different parts, the status area which can be read only by the host application, and the control area where writing is enabled.

An application consists of several tasks. Each task is executed in its own time slot. This slot has nothing in common with the TDMA round division and may not interfere with other tasks. Basically there are three types of tasks:

- One task that updates the host life-sign field.
- Optional background task that is executed when no other task is being executed.
- Any number of application specific tasks.

Management of the tasks is handled by a simple operating system called TTP OS. The OS uses a time interrupt from the controller to synchronize the execution of the tasks and implements a time-out for the maximum task execution duration to prevent host lockups.

The startup of the controller and transition into an application mode is also handled by the OS. The tasks are first executed after the controller is safely integrated into a running cluster and in an application mode.

The host application is written together with TTP OS into the host EPROM and the MEDLs are transferred to the node via a download mode cycle.

3.1.2 TTP/C C-Based Implementation

The C-Sim based simulation implements nearly all the functionalty of a real controller, and the interface to the host layer (the CNI and the interrupt line) is even exactly the same. This ensures that an application written for a silicon implementation will run on the simulated controller without any or only with a few modifications.

Some differences in the simulated controller are inevitable, as the protocol is very complex and not every hardware component can be easily modeled.

- Watchdog timer is not implemented and so controller lockups are possible. Such failures have to be checked from outside. As long as no fault injection into controller code is performed, this timer is not required.
- Download mode is not required and thus not implemented because the application is linked together with the controller simulation. As a replacement for the download mode reading of the MEDL tables from a text file was implemented.
- C-Sim code executes in zero model time, so no task can exceed its maximum execution duration. The same applies for CNI access duration, so the NBW protocol is not needed.

3.1.3 TTP/C Application Specification

Any application that runs on a TTP/C controller needs the following parts:

- Defined MEDL table.
- Interrupt handler.
- Life-sign update task.

- One or more application tasks.
- Startup routine.

Additionally there are some optional parts:

- External interface.
- Application initialization routine.

In the following sections the creation of all the necessary parts will be described with special care given to the design of MEDL tables and writing of application tasks. The startup routine will be described only briefly as a detailed description can be found in [JEZ01].

3.2 MEDL Table

When creating a new application every host must have its own MEDL. The contents of the MEDL describes the temporal behavior of the controller and defines messages that are sent. Some of the fields are required for the basic function of the controller within a cluster and some are optional. Only a small portion of the fields contained is used by the simulation.

The MEDL is read from a text file upon controller initialization and may not be modified while the controller is running. The format of the MEDL source files will be described later, in section 3.2.3. The text files can be edited manually or generated by a conversion utility from the binary files produced by TTP Plan. The conversion utility was developed at UWB and is available at the FIT web pages ([FIW]).

Every controller's MEDL must specify some minimal set of fields to be able to integrate into a cluster. Part of the MEDL contains the same values for each controller in the cluster (section 3.2.1), and the remaining values are specific to each SRU (section 3.2.2).

All of the fields described in this section must be set to a reasonable value, otherwise the controller may not operate correctly. To learn more details about the MEDL structure see [TPS99].

3.2.1 Common MEDL Fields

This paragraph describes all the fields of MEDL that must contain the same values for each controller within a cluster. Every field is required but may contain a null value.

- Controller configuration parameters
  - C-State Length: Number of bytes used by C-State.
  - TDMA Round Length: Duration of a single TDMA round in macroticks.
- Slot control entry (one for each slot in the cluster cycle)
  - SRU Slot Duration: Duration of the current slot in macroticks.
  - Bit in Membership vector: Bit position of the sending SRU.
  - EOT: Set if current slot is the last in the current transmission cycle.
  - Logical Sender Name: The logical name of the current sending SRU.
- Mode address entry (one for each slot and each mode)
  - IA: Set if the frame sent on channel A is an I-Frame.
  - IB: Set if the frame sent on channel B is an I-Frame.
  - DAFLA: Length of the application data sent over channel A in bytes minus one.
  - DAFLB: The same as DAFLA, but for channel B.

3.2.2 SRU Specific MEDL Fields

The following set of fields may contain different values for each controller.

- Role List
  - Logical Name: Logical name assigned to this specific role.
  - BG Parameters: BG parameter set used once the controller is synchronized.
- Controller configuration parameters
  - CF: Set if the current SRU can enter the cold start mode.
  - Maximum Cold Start Entry: Maximum number of attempts to send the initial I-Frame.
  - MMFC: Maximum number of successive membership failures.
  - Startup Timeout: The duration of all TDMA slots prior to this SRU in macroticks.
  - Free Running Macroticks: The minimal number of unchanged macroticks between two consequent clock synchronizations.

  - Microticks/Macrotick: Number of microtics per macrotick.

- BG Cold Start Parameters: BG parameter set used while in cold start phase.

- Implementation specific parameters

  - BG Synchronization Period: BG parameter set required for the startup of the controller.
  - BG Listen-Cold start Period: BG parameters required for transition from cold start to the listen state.

- Slot control entry (one for each slot in the cluster cycle)

  - RA: Set if the current SRU is allowed to reintegrate in the current slot.
  - SYF: If set, the frame sent in current slot is used for clock synchronization.
  - CS: If set, the clock synchronization is executed at the end of the current slot.
  - BA: If set, the BG Arm signal is raised in the controller.

- Mode address entry (one for each slot and each mode)

  - CNI Address (A): The word offset into the message area of CNI where the message sent over channel A is stored. Value of '`0x3FFF`' is reserved and denotes that the message is not stored.
  - CNI Address (B): The same as CNI Address (A) but for channel B.

There are many fields in the MEDL structure, that are not used by the current implementation of C-Sim model, mostly in the "Implementation Specific Parameters" section. Also there are some fields which are used only under certain circumstances, like multiplexing and shadowing. Description of all fields can be found in [TPS99], the list of implemented fields and implementation details in [JEZ01].

3.2.3 MEDL Text File Structure

The C-based TTP/C model does not implement a download mode, instead each node loads its MEDL table from a text file. The structure of this file was carefully designed to make it readable for any person and also to allow direct modifying of the MEDL files.

Each line in the MEDL source file can contain one of the following entries:

- A comment starting with '#'.
- A Blank line.
- Complete assignment — a value separated by white space from a complete field name.
- Default assignment — the value is missing and the one from the last complete assignment is used instead.
- Relative assignment — the name part starts with a dot, the first part of the name is then supplied from the last complete name specified.

Structure fields are separated by a dot, array elements can be selected in square brackets as in C.

An example of a file containing partial MEDL definition is displayed in the next lines:

```
# definition of three controller_cfg_params
1000    controller_cfg_params.tdma_round_length
100     .micro_to_macro
105     .ifg_time

# definition of slot control entry with index 0
# the last field will have a default value of zero
250     slot_control_entry[0].sru_slot_duration
0       slot_control_entry[0].membership_pos
        slot_control_entry[0].eor
```

3.2.4 MEDL Creation

The MEDL table contains parameters describing the temporal behavior of each node in the cluster. The tables are personalized for every node in the cluster and although some of the parameters must have the same values, no two MEDLs may be exactly the same.

The creation of all MEDL tables is a repetitive task as many of the parameters have the same value for all nodes and many can be derived from the cluster application specification. For those reasons a software tool is used to generate all MEDL tables from a common application specification. TTTech provides for the purpose of application planning a tool called *TTP Plan*. The output of this tool is a personalized MEDL table for each node in the cluster.

For the C-based specification the same tool can be used as a conversion utility was created at UWB, to convert binary MEDL files generated by

TTP Plan into text files used by the C-based specification. The problem of designing a new application using the TTP Plan tool is well covered in the user documentation supplied by TTTech and need not be discussed in this report. In the next section we will instead concentrate on the basic design principles and the methods used in simulated applications.

3.3 Host Application

3.3.1 CNI Access

CNI access is made available to the host through a pointer referencing the controller's CNI memory. This solution was chosen, because both parts of the SRU use the CNI and to make the access symmetric the CNI would have to be outside the controller and the host.

There are no restrictions that would disable writing into some areas of the memory like it is the case in the silicon implementation. However the parts of CNI which need write protection are shadowed within the SRU layer and any change to them is discarded. This solution ensures that no write operation into the protected area results into an unpredicted error condition.

The problem of data inconsistency in the CNI is solved with the means of NBW protocol. This protocol uses a four-bit counter, that is incremented before and after each memory update. The host in the hardware implementation cannot modify the memory if the counter contains an odd value, i.e. the controller is updating. In the simulation, the write operation is completed in zero time so the counter increments always by two. The host does not need to follow the NBW protocol in such cases. If the host however simulates write duration, the NBW counter must be used to check if the controller did not change the CNI during the operation.

3.3.2 Binding to the Controller

Binding to a controller is a necessary part of the host initialization. Typically only a reference to the CNI memory area is needed and the controller must registrate the host's interrupt function. A method of controller-host binding is shown in the sine wave example (Appendix A). A explanation of the binding process can be found in [JEZ01].

3.3.3 Startup Routine

The startup of the controller is quite a complex task and a detailed description can be found in [JEZ01]. Here only a short list of the steps needed to bring the controller on-line is presented.

1. Set Controller On (CO) field.
2. Wait for Controller Ready Flag.
3. Initialize application data and CNI message area.
4. Enable MI interrupt (mode change notification).
5. Wait for C-State valid flag and update the host lifesign field.
6. Schedule the host lifesign update task with the TI1 interrupt.

This routine is handled by TTP OS in the hardware cluster as well as in the simulation model, but it is also possible to create applications without the OS. Because the startup routine requires time progress, and that means the use of C-Sim time control, it must be directly in the process program and not in a separate function.

3.3.4 Interrupt handling

A single function serves as the interrupt handler for all asynchronous events. To allow the simulation of interrupt handling latency and to simulate execution duration, the handler must not provide application functionality as it is run from the controller process. Instead it should reschedule the host process and set an interrupt flag, so that the application can handle the interrupt.

To determine the cause of the interrupt the application must check the interrupt status field in the CNI. After handling the interrupt the status flag of the handled interrupt must be cleared. The steps taken when an interrupt arrives are the following.

1. The handler sets an 'interrupted' flag for the application and reschedules the application to activate within the given latency parameter.
2. The interrupt handler terminates.
3. The application wakes up and tests the 'interrupted' flag. When the flag is set, the application enters a new branch of execution.
4. The application checks all possible interrupt conditions (the interrupt status field in CNI) and branches the execution accordingly.
5. A routine handles the interrupt event and clears the interrupt status field.
6. The application's 'interrupted' flag is cleared.

3.3.5 Lifesign Update Task

For a real cluster this task is generated automatically by the *TTP Build* tool. For the simulation this task must be however created manually. The actions taken during the update of the lifesign are described by the TTP/C protocol and may be provided as a function from the 'fp' layer (not in current implementation). The function may also test the controller's lifesign, but so far no action was implemented in the case the controller fails, so the test can be skipped.

The update task is run just before the controller's own sending slot to ensure minimal host error detection delay. Scheduling of this task is always the same, it is run once per TDMA round just a few macroticks before the IFG preceding the sending slot of a controller. To ensure a synchronized activation of the task the Time Interrupt 1 (TI1) provided by the SRU layer is used. A simple function implementing the required functionality looks like this:

```
void task_update_lf(HOST *host)
{
    cni_set_host_lifesign(host->cni,
        ~cni_get_controller_lifesign(host->cni));
}
```

3.3.6 Application Tasks

Application functions that are scheduled periodically are called application tasks. They can perform any action as long as they ensure they will always return the control to the calling function. They cannot use any C-Sim time affecting services, because the C-Sim supports no process switching within sub-functions.

Each task is specified by the following parameters:

- Schedule time - A time value in macroticks relative to the begining of transmission cycle.
- Execution duration - Execution time of the task routine. The C-Sim tool executes code in zero time, so this value is not needed. It could be used for design checking performed by the model.
- Periodicity - Scheduling period in macroticks.
- Task routine - Function that performs the task's operation.

In the hardware implementation the tasks would be executed by the TTP OS. In the simulation however the tasks must be run directly from the C-Sim process simulating the host layer. The calling of the various task functions must be interlaced with `hold()` commands to simulate time progress.

A convenient way to synchronize the task execution with the controller is to use the second time interrupt provided by the SRU layer. This exactly what the TTP OS does.

3.3.7 Application Initialization

The application may specify one function to be executed only once, upon the initialization of the controller. This routine may be used to initialize any application specific resources. It is not a common task, because it is not scheduled again after its completion.

This function may not contain any C-Sim process switching and is run directly from the host application process program, the TTP OS or the application simulation if there is no OS.

3.3.8 External Interface

The external interface of an application can be any form of input or output. An application can use any devices in the computer, it can export any of its data also. Visualization always uses some kind of external interface to gather data from the simulation.

3.3.9 Lockup Avoidance

In the hardware implementation the TTP OS uses a timeout mechanism to prevent faulty application tasks from locking up the host processor. In the simulation there is no such mechanism, so other means of protection must be implemented. The application tasks itself must be safe, i.e. they must not contain any synchronization points or they must implement some timeout mechanism to avoid locking.

3.3.10 Cluster Initialization

The cluster initialization proceeds in several steps. The steps are heavily application dependent so only a general checkpoint list will be provided at this place:

1. Initialization of protocol specification.

2. External objects creation.
3. Creation of controllers and hosts.
4. Loading of MEDL tables into each controller.
5. Creation of two channels and binding of the channels to each controller.
6. Binding of the each controller to its host.
7. Binding of the external input and output to the hosts.
8. Initial scheduling of the hosts.
9. Initialization of the external objects.

It is generally a good idea to create a separate module for the cluster initialization as it uses only the simulation extension and ANSI C code. The cluster module can be made easily portable to any system. Adding of any visualization to a single module is also a much easier task.

3.4 Cluster Visualization

In this section there will be described the steps taken to visualize the simulation in the C++ Builder environment. First the modifications needed inside the C-Sim tool to compile with C++ Builder will be listed, then the multithreaded C++ extension of C-Sim will be explained. At last the user interface will be described briefly.

3.4.1 C-Sim Modifications

To compile with C++ Builder all linked files must be compiled either as C only or as C++, mixing of the two languages is not allowed. Because we will need to use the visual interface later, we have to compile everything as C++. C-Sim consists of several files with the '`.c`' extension which have to be renamed to '`.cpp`'.

All modifications will be made available in a new version of C-Sim. A correct version, although not official, is contained within the simulated TTP/C protocol.

The changes corrected some possible bugs and removed some warnings reported by the compiler. Moreover several exported symbols were renamed to gain C++ Builder compatibility. The following text will summarize only the changes to the exported part of the library, for the C-Sim documentation refers to the older official version.

Renamed exported symbols:

- ERROR → CSIM_ERROR
- BYTE → CSIM_BYTE
- WORD → CSIM_WORD
- BOOLEAN → CSIM_BOOLEAN
- TIME() → CSIM_TIME()

3.4.2 Multi-Threaded C-Sim Extension

The C-Sim tool is designed to run from the main program function, `main()` in a typical C program. In a visual application this is rarely possible, so we must find another solution. In a multi-threaded environment we can run the simulation in a designated thread to separate it from the user interface. In C++ Builder it is best to combine the thread and C-Sim into a new class to be used later in a number of simulations.

This class we will descend from the library class `TThread`. All vital C-Sim functions will be made available by publishing methods or properties. The simulation run will take place in the standard main thread method called `Execute()`.

The new class is called `TCSimThread` and exports the following properties and methods:

- `double slow_down` — Delay between two steps.
- `bool tracing` — Stop execution after each step.
- `double csim_time` — Current model time.
- `bool zero_stepping` — Stops after zero-time steps also.
- `void go()` — Runs the simulation without stepping.
- `void stop()` — Stops the simulation after the current step.
- `void trace()` — Executes one step of simulation.
- `void get_data()` — Retrieves the output of the simulation. The data must be prepared previously in a local buffer by the virtual method `export_data()`.

The following methods are defined as protected and should be redefined by the user in a descended class.

- `init()` — This method is run before the first step of simulation.

- `export_data()` — This method is called before each simulation step and should write any output data to a local buffer for retrieval by the `get_data()` method.

This class is provided by the module named csimthrd.

3.4.3 User Interface

To create the user interface C++ Builder environment will be used. It provides many graphical components that enable us to develop a visual user interface with only minimal effort. Unluckily the environment does not include a compiler for other system than MS Windows. For UNIX systems other environments must be used instead or the visualization has to be restricted to console output only.

Application development for a multi-threaded system requires careful manipulation of the data shared by multiple threads. Because the simulation runs in a separate thread, its output must be solved by means of inter-process communication, for example message passing or memory sharing. In this case the communication through shared memory will be used, as it is more effective than message passing. Message passing would make the distributed simulation run easier, but in this case, the simulation runs on a single machine.

To ensure data consistency in the shared memory area, a critical section has to be used.

Chapter 4

Simulation Based Fault Injection

In this chapter the basics of software simulated fault injection applied to a simulated TTP/C node will be presented. This method has been used in the sine wave example to test the model functionality and verify the expected TTP/C fault tolerant properties.

4.1 Fault Injection Targets

There are several possible targets for the simulated fault injection. Generally the simulation is able to perform a wider range of experiments than a real hardware would allow, e.g. modifications in ROM stored programming. The simulation based fault injection method is never destructive, thus lowering the costs of experiments significantly.

Several possible targets for simulated fault injection can be set and examined:

- CNI Status/Control/Message area (RAM)
- MEDL Table (ROM)
- Host application data (RAM)
- Controller registers (RAM)
- Controller code (ROM)
- Host application code (ROM)
- Interrupt signal line

- Channel services
- External inputs

From the options listed above only the fault injection into CNI memory area and to some extent into the MEDL also may be tested without complications (i.e. on the application level) on the hardware implementation. To receive comparable results the first experiments will concentrate on these areas. This approach allows to verify the model before proceeding to the experiments which cannot be easily replicated in hardware.

This document will describe only the fault injection into the CNI memory area of a chosen SRU.

4.1.1 Fault Injection into CNI

The CNI consists of three separate memory areas: control, status and message area. A fault in each of these areas will have a different effect on the SRU, namely different effect on the host layer and on the controller layer. The effects are summarized in Table 4.1.

CNI Area	Status Area	Control Area	Message Area
Affects host	yes	possibly	yes
Affects controller	no	yes	yes

Table 4.1: Effects of faults in the CNI

The effect of faults in the control area of CNI on the host depends on the application. Mostly the control fields are not used for status read-back, although there is one exception. The Controller On (CO) field is often used to test whether the controller is still online.

4.2 Fault Type Specification

There are multiple possibilities when it comes to fault injection. Basically the injection process can be described by four values:

- The mean frequency (periodicity) of faults.
- The size of the fault, i.e. single bit faults/multiple faults.
- The effect of the fault, i.e. the bit can be set to ‘1’, ‘0’ or a random value. An improbable fault effect is bit flip, but it is also the worst fault type, as it always changes the bit value.

- The exact position of fault injection. The position can be selected by a pseudo-random generator or specified by the user.

The first stage of experiments will use black box testing method, so the bit position of fault injection will be determined by random number generator. See section 4.3 for details on fault position finding.

In the verification process the worst fault type (i.e. the bit flip) will be mostly used. To simulate real conditions, other fault types must be also included.

The size of the faults will be described better in section 4.4. Most common will be the single bit faults.

The fault frequency must be variable to allow the simulation of different conditions, i.e. hard conditions with a high fault rate, or mild conditions with lower rate of faults. The periodical generation of faults is the topic of section 4.5.

4.3 Determining Fault Position

A common fault type is a random bit change in the memory. A typical black box approach would select one random bit from a memory range. To make the fault injection process repeatable with the same results a pseudo-random generator must be used to determine the bit position.

To make things easier we will assume that every bit in the memory has the same probability of failure. Then the bit position is a variable of an uniform probability distribution covering the whole memory area.

4.3.1 Positional Fault Mask

The implementation of CNI in the simulated TTP/C protocol does not copy the hardware implementation exactly. The individual fields are implemented as 32-bit integers and as each field defined by the protocol uses a maximum of 16 bits, there are gaps of various size between the fields. A black box method described in section 4.3 would inject faults in the CNI fields and in the unused space. The result would be an unpredictable model behavior which could lead to false assumptions, because there would be less failures than anticipated.

The solution to this problem is a positional fault mask that would tag all bits within a memory area that should be hit by fault injection. The easiest way to implement such mask is to make a copy of the fault injected structure and fill all desired fields bit-wise with `'1'`. This implementation requires a lot of memory but provides a very flexible fault selection method which could be used later in white box tests performed over the protocol.

An example of a fast algorithm for fault position computation can be found in section 4.6.

4.4 Fault Size Definition

The size of a single fault can range from a single bit to the whole memory area. Basically we have the following possibilities:

- Single bit.
- Single structure field, e.g. Cluster Time.
- Multiple bits — either fixed or random count.
- Memory area, e.g. whole Status area, whole MEDL, etc.

The most common effect of a faulty memory is the single bit or multiple bits fault size. The most common effect of a disastrous event as a lightning or electric short connection is an area-wide fault. For experiment purposes there is a limited number area-wide faults to test so we will concentrate on the fault size of a single bit first. Multiple bit faults can be simulated by increased fault rate.

4.5 Fault Rate Definition

The fault injection is a continuous process. We can define the severity of faults by specifying the mean fault frequency or the mean period between consecutive faults. To simulate the real environment better we cannot choose a constant period, but instead we will use a variable of the exponential probability distribution (i.e. we get a Poisson stream of faults). The parameter of this distribution, *lambda*, defines the mean fault frequency.

The fault injection is a periodic task and so is perfectly suited for a C-Sim process implementation. As C-Sim supports direct Poisson stream variable generation, the process will be very simple:

```
for (;;) {
    if (p_my->enabled && p_my->lambda > 0.0) {
        inject_fault(p_my);
        hold(negexp(p_my->lambda / 1000000.0));
    } else
```

```
        passivate();
}
```

The `enabled` property is used to disable and enable the fault injection without the need to change process scheduling. The `lambda` parameter denotes mean fault count per second, but as the model time runs in microseconds, it must be converted before use.

4.6 Fault Position Selection Algorithm

The algorithm for bit selection within a masked memory area is not simple if we want to get an effective solution. In the experimental validation of the model and TTP/C testing, the speed is more important than the memory consumption, so we may make use of additional memory structures to speed up the bit search.

4.6.1 Index Building

The algorithm applied in this project uses an index generated at mask specification time to effectively find a bit at chosen position. The index is an array of integers. The value of each element is defined by:

$$\texttt{index}[i] = \sum_{j=0}^{i} \texttt{bit\_count}(\texttt{mask}[j])$$

The `bit_count` function returns the number of tagged bits within one integer. The size of the index array will be the size of the mask divided by the size of integer, as the mask is read with integer granularity.

4.6.2 Integer Position Search

Now we have to find the integer position of the attacked bit within the selected memory area. If we use the index generated in previous section the task is quite simple. We have a sorted array of integers (the count of tagged bits within the mask) in the index, and we have the index of the attacked bit, so we can use the method of interval division to speed-up the search process.

```
while (left < right) {
    if (index[(left + right) / 2] >= bit_pos)
        right = (left + right) / 2;
    else
```

```
        left = (left + right) / 2 + 1;
}
```

This algorithm ensures, that on completion the `left` variable contains the integer position of the bit at position `bit_position`.

4.6.3 Bit Position Within Integer

Now we must find the selected bit position within the integer field found in the previous step. We have the integer position stored in `left`, and we have the bit index (counted from start of the memory area) in `bit_pos`. From these input values the following algorithm is able to compute a mask for the single bit specified by the bit selection mechanism. This mask is stored in the variable `bit_mask`.

```
if (left > 0)
    count = bit_pos - index[left - 1];
else
    count = bit_pos;

bit_mask = 1;
do {
    if (mask[left] & bit_mask)
        count--;
    if (count > 0)
        bit_mask <<= 1;
} while (count > 0);
```

4.6.4 Single Bit Access

Once we have found the integer position of the fault injection and also the bit-mask for this integer, we can perform the fault injection. The action taken in this step depends on the chosen fault type, so we will just show an example of bit flip type fault. This example demonstrates how to access the memory area to affect just the selected bit.

```
memory[left] ^= bit_mask;
```

4.7 CNI Fault Injection

As stated in the previous text, in this project mainly the faults injected into CNI memory will be used. This task is not application dependent and so should be designed as a reusable module. The only interface the fault injector needs is the address of the CNI area of an SRU.

The parameters describing the fault injector properties are:

- The mean frequency of faults (`lambda`).
- The fault type (bit flip, fixed zero, fixed one).
- `N` addresses to CNI.
- `N` fault masks.
- `N` fault injection enable fields.

The listed properties enable us to use one Fault injector to inject faults into multiple (`N`) CNI areas, i.e. into multiple SRUs. The injection into every single SRU can be disabled by clearing the `enable` field. The faults are generated with the specified frequency using a Poisson stream and always hit all enabled SRUs at the same bit position causing the same type of fault.

The fault injector is implemented as C-Sim process and exported by the module `sa1fault`. The injection algorithm is not bound to the CNI data structure and can be used to perform fault injection into other memory areas too (e.g. MEDL).

4.8 Channel Services Disruption

Another type of faults used to test the simulated TTP/C protocol is transmission disturbance. The fault injection is supported by a mechanism implemented directly in the channel model.

This type of faults has been used so far only for special case testing, mainly for the purposes of the development process. The injection process is controlled by the user and contains no possibility of periodical repetition. A single fault causes a single bit flip in the next transmitted message, so it is not possible to inject faults into IFG (when no frame is transmitted). The fault always affects both channels, as a single channel fault would be handled by the dual redundancy implemented in the protocol.

4.9 Experiment Organization

We follow two main goals when we applying the simulated fault injection to our model. First, we expect to find any bugs in the protocol simulation by testing all of the features supported by the protocol. Second, we may use the experiments to verify the expected fault tolerant properties of the protocol.

The first phase of the project—to detect any bugs in the implementation and remove them—has been to a great part already accomplished. In a project of this size it is impossible to certify, that a model is a true representation of the real world, but this is seldom necessary.

The second phase of experiments is targeted on the experimental validation of the proposed properties of TTP/C. As the whole protocol is very complex, we must separate the provided functionality into isolated parts and check one at a time. The process of experimental validation will have the following steps:

1. Protocol functionality selection.
2. Hypothesis: The protocol behaves as predicted.
3. Fault injection into CNI — multiple tests with different fault rates.
4. Experiment result evaluation: Was the hypothesis violated?

If the hypothesis was not violated (i.e. the protocol simulation behaves as predicted) we can formulate another hypothesis. If the experiment result was in contradiction with the original hypothesis, there are two possible reasons. Either the model is not correct, i.e. there is an implementation bug, or the protocol does not behave as predicted. Then we must reconsider if the assumptions made were correct.

If a flaw in the TTP/C protocol specification is detected, further experiments with slightly modified conditions would be required to specify the deviation more precisely and possibly to detect other flaws.

4.9.1 Tested Scenarios

During the protocol model implementation many experiments have been already executed. The hypotheses tested in the experiments and the results are listed in the following sections.

Hypothesis 1

The output of both FTUs must be the same if only one node in the TMR unit (first FTU) is exposed to fault injection.

The hypothesis was tested with different fault rates and the hypothesis was confirmed.

Hypothesis 2

The first FTU (the TMR unit) will fail to output a correct value occasionally if two of the SRUs are exposed to fault injection. The output failure probability will grow with higher fault injection rates. The output function will cease completely if the fault rate is +too high.

The hypothesis was confirmed.

Hypothesis 3

The node R should never fail if only node F is exposed to fault injection.

The hypothesis was confirmed.

Hypothesis 4

The node R should never fail if two nodes (F and G1) are exposed to fault injection.

The hypothesis was NOT confirmed. This suggests either an implementation fault, but so far none was detected, or a false assumption, or a protocol fault. The assumption is not based on any statement from the TTP/C specification, and so is possibly incorrect.

Hypothesis 5

Faults injected at a high rate into a node that is the only cold start node of the cluster prevent the cluster from starting up.

The hypothesis was confirmed, a fault rate of 100000 faults per second and higher resulted in the predicted model behavior.

Hypothesis 6

A single bus fault (both channels) cannot affect more than one node. If the node affected is the R node, the output of the second FTU is not correct, the output of the first FTU is always correct.

The hypothesis was confirmed.

Hypothesis 7

A double bus fault (two consecutive message faults on both channels) causes a failure of all nodes within the cluster. The cluster then starts up again using the cold start mode.

The hypothesis was confirmed.

Chapter 5

Sine Wave Application

5.1 Overview

Here an example of an application sending and evaluating a sine wave signal will be presented. All steps needed to create such application will be explained and demonstrated by code fragments. The development of the application will be based on the so-called empty application described in [JEZ01].

The sine wave application was the first proposed testing application for the TTP/C protocol. For our purpose the original design was modified in several parameters, but the main idea remained: a simple application with easy-to-verify output and limitless repeatability.

The source code listing of this example is available in Appendix A.

5.1.1 Application description

The application will run on a cluster consisting of four nodes as depicted in Figure 5.1. All nodes perform the same operation, the measurement of an external sine wave signal and the output of the measured signal value to an external register. The measured value will be transmitted on the bus so that the nodes may compare their measured values with each other.

Each of the FTUs provides one measured value. The second FTU is formed by a single node, R, and will output a correct result only if the node also operates correctly. In this example we will consider this node as a reference (or golden) node and we will never inject faults into this node. The first FTU comprises three nodes, forming a Triple Modular Redundancy (TMR) FTU. Such an FTU ensures correctness in the value domain even if one of the nodes fails. The result is determined by majority voting on all valid results from the nodes.

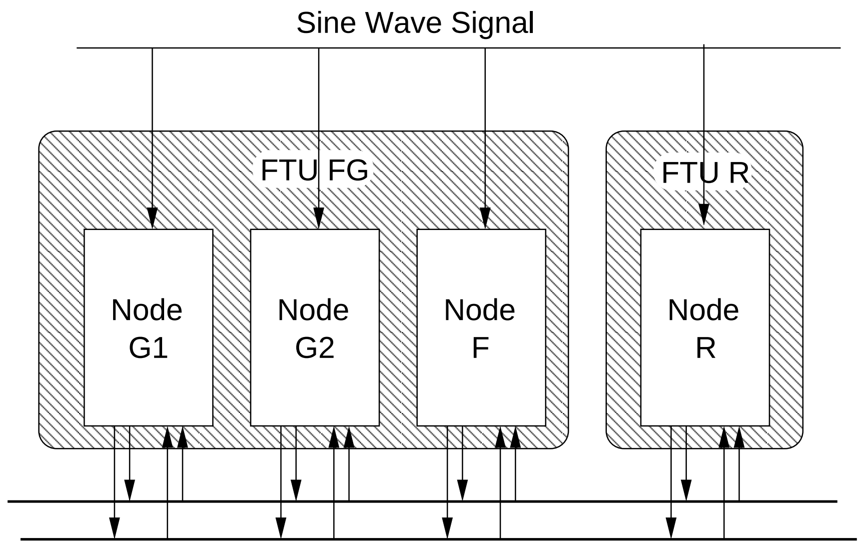

Figure 5.1: TTP/C Cluster for Sine Wave Application

The output of the first FTU will be an external register called *MFG*. The output of the second FTU (the node R) will be stored in a register called *MR*.

To verify the model validity and also the expected TTP/C fault tolerant properties, several checks must be included into the application. To test the model under hard conditions, we must simulate fault injection into some parts of the model. In this example fault injection into the CNI memory will be used to test the model.

5.2 Cluster Design

5.2.1 Application Messages

First we must decide what messages will be sent over the bus. As all of the nodes run the same application, this task is greatly simplified.

The only application message sent by the nodes will be the measured value of the sine wave signal. This value will be stored as a four-byte float number in the IEEE standard. The expected range of the signal is $< -1.0; 1.0 >$.

Each received message may be stored in the CNI message area. The offset within the area is specified in the MEDL table of the receiving controller. Each stored message contains a state information indicating if the message is correct (e.g. if it is equipped with a correct CRC).

5.2.2 Slot Allocation

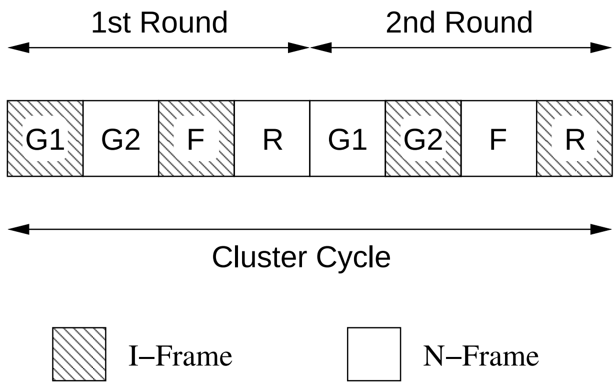

Figure 5.2: Sine Wave Application — Cluster Cycle

Our TTP/C cluster comprises four nodes that may be active at one time. This requires to use a TDMA round of at least four slots. We could design a larger round to allow future expansion of the cluster, but in this example this is not necessary. The round will contain only four slots, one for each node. The sending order of the nodes in the TDMA round is depicted in Figure 5.2.

The protocol requires that at least two nodes send an I-Frame once per cluster cycle. This ensures that failed nodes are able to reintegrate back into the cluster. To include I-Frames into our schedule, we must add a second TDMA round, so we get four additional sending slots. We can allocate some of the slots to N-Frames or use them all for I-Frames. In our case we do not need additional N-Frames, so we allocate four slots to I-Frames. The type of frames sent in each slot is also depicted in Figure 5.2.

5.2.3 Operation Modes

In the sine wave application only one mode is used, the Startup mode. This mode is designed to send only I-Frames during the startup of the cluster. Because the implementation did not support mode changes when this application was designed, this mode was modified to include also N-Frames. The index of the startup mode within the mode table is `'1'`.

5.2.4 Time Properties

The model requires several time parameters to be set. The default channel speed is set to 1Mbit/s and macrotick duration is by default set to one microsecond. The Inter-Frame Gap (IFG) between two consecutive frames may be set freely, but the current hardware implementation uses IFG value of 101 macroticks.

Now we must decide the length of each slot so that the controller has enough time to completely transmit each message. The longest message sent is in our case the I-Frame, containing six bytes of C-State, two byte CRC code and one byte header, making a total of nine bytes. The transmission of nine bytes takes approximately 72 microseconds, i.e. 72 macroticks. The longest message ever possible is nineteen bytes and the transmission would take 152 macroticks. Together with the IFG we get slots 253 macrotick long.

To make things easier, we will make all slots 250 macroticks long. Then a TDMA round is 1000 macroticks long, i.e. one millisecond in out example. A complete cluster cycle duration is thus two milliseconds. The chosen slot length enables a later extension of message length.

5.2.5 MEDL Tables

The cluster contains four nodes, so four MEDL tables must be prepared for the application execution. All four source MEDL files are listed in Appendix A.5. The names for the files were chosen to follow this pattern: the name must be "medlxx.txt", where xx is any number. The numbers chosen in the example are '00', '01', '02', '03'.

MEDL Setup

In the controller configuration parameters section the most interesting is the Cold-Start Flag (CF), that denotes if a particular controller is enabled to enter cold start mode. At least one node must have this parameter set.

The next part of the MEDL is necessary only for active nodes. It is the configuration of Bus Guardian (BG). There are four sets of BG parameters, each for a different state of the controller: two sets for the cold start state, one for the synchronization period, and one for normal mode of operation.

Implementation specific parameters section contains only BG settings for the simulated TTP/C controller.

For each slot within a cluster cycle the MEDL contains one slot control entry (*SCE*) which specifies the length of the slot, the sender name, and whether this slot should be used for clock correction. Besides the SCE entry, there are multiple mode address entries (*MAE*), one for each operation mode. Each MAE defines the length of transmitted data, frame type and address where to store the message data.

When deciding the offset of a message in the CNI, we must consider the message length and the state information. In our case the message is four bytes long and the state information is two bytes, together six bytes. If we get the message on both channels, then twelve bytes are required to store the received data.

The description of all fields in MEDL and its design principles can be found in [TPS99].

5.3 Host Application Design

5.3.1 Process Design

The C-Sim process representing a host application must contain all application specific parameters and links to other simulated objects, e.g. the CNI of the controller. The source file containing the module is named sa1sinh.

A simplified process definition for our example is defined here:

```
derived_process(swave_host)
    /* Links to external objects */
    CNI *cni;
    SINE_GENERATOR *sine_generator;
    float *output_reg, *reference_reg;

    unsigned index; /* distinction between different hosts */

    /* Interrupt specific state variables */
    double last_ui1;
    double last_ti1;
    unsigned interrupted;

    /* Counters for statistics gathering*/
    unsigned null_frame_counter;
    unsigned crc_error_counter;
    unsigned data_error_counter;
    unsigned cluster_fault_counter;
    unsigned mismatch_counter;
end_derived(SWAVE_HOST);
```

The links to external objects and the index of the host application must be set before executing the process. The rest of the attributes is initialized with default values during host initialization process.

5.3.2 Task Synchronization

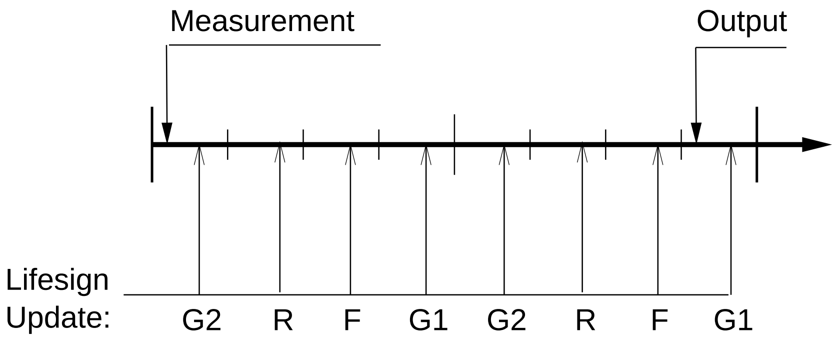

Figure 5.3: Application Task Synchronization

The application must measure the external sine wave signal and store it into the CNI message area to send it in the next sending slot. The value is measured during the first slot in each cluster cycle. To ensure that the value is measured synchronously in all nodes, the user interrupt (UI1) is used. The MEDL of each node has the Raise User interrupt 1 (RU1) bit of the first slot enabled.

The output of the measured value is also synchronized, this time in the last slot of the cluster cycle. Again the UI1 is used, but this time the enable bit RU1 is set in the last (eighth) sending slot. To decide what action shall be performed when the UI1 interrupt is raised, the application must check the current slot position. The measurement task provides the function `sw_task_measure()`, the output task is provided by the function called `sw_task_output()`.

The host application must update its lifesign before the sending slot. The update should take place just before the sending slot starts, so that the lifesign shows updated information. To update the lifesign in time, the application uses time interrupt 1 (TI1). The first update is performed 50 macroticks before the sending slot, the next updates repeat with the period of one TDMA round, i.e. 1000 macroticks. The lifesign update task is provided by the function `sw_task_update_lf()`.

For a better idea of the task synchronization see Figure 5.3.

5.3.3 Task Description

Lifesign Update Task

The task description can be found in section 3.3.5.

Sine Value Measure Task

This task is executed once per cluster cycle. It reads the sine wave value from an external register and stores the value into the CNI message area. The application must set the state of the CNI message slot where the value was stored to a `CRC_VALID` value, otherwise the controller will not send the message.

If both the controller and the host operate correctly, the message is sent in the next sending slot.

Sine Value Output Task

The sine wave value is sent to an output register at the end of every cluster cycle. There are two possible scenarios, depending on the FTU unit the node is part of.

Node 'R' is the sole node of an FTU and thus performs no voting mechanism to ensure value correctness. The value is always sent to the output.

Nodes 'G1', 'G2', 'F' are a part of an TMR FTU. This means that the output of the three nodes is always compared and only if at least two results match together are they sent to the output.

5.3.4 CNI Message Area

The messages sent over the TTP/C network are stored within the message area of CNI. Each message is stored in a separate message area entry. Each entry consist of a message state field and application data. The maximum size of application data is 16 bytes, but we use only 4 bytes. The size of the state field is two bytes.

We must specify where all messages received from other nodes will be stored and also where the message selected for sending is placed. The position in the CNI is written into MEDL as a word offset, considering the start of message area as zero offset.

Each node may store the messages at different addresses. But if all nodes store all messages, it is better to use the same address for one message within all controllers. The message entry offset for all messages is defined in Table 5.1.

–/–	Node G1	Node G2	Node F	Node R
Channel A	0	6	12	18
Channel B	3	9	15	21

Table 5.1: Offset of Message Entries within CNI

5.3.5 Lockup Avoidance

The sine wave application relies heavily on time and user interrupts. When the controller fails and stops to periodically raise one of these interrupts, the application must detect this and reset the controller. For this purpose the last occurance of time and user interrupt is recorded and compared to the current time. If the difference exceeds a predefined maximum value, the controller probably failed and is restarted.

5.3.6 Voting Mechanism

To ensure fault tolerance in the value domain, the TMR FTU uses a voting mechanism when computing the output value. The best idea of how the mechanism works gives a simple example:

```
if (!correct(G1))
    G1 = nonsense1;
if (!correct(G2))
    G2 = nonsense2;
if (!correct(F))
    F = nonsense3;

if (near_match(G1, G2) || near_match(G1, F))
    output(G1);
else if (near_match(G2, F))
    output(G2);
```

In this example the `nonsensex` value denotes any value outside the range of a sinus signal. The `near_match()` function returns true when the two parameters have a specified maximum difference.

5.3.7 Statistics Gathering

For the means of statistics gathering (e.g. number of faults injected, detected failures, etc.) the application must include counters for the specified cases. In this example, all four nodes perform statistical computation, and all perform the same one as they share a single application. The results of different nodes may differ so we will use only the results from the reference node, 'R'.

The gathered information will be:

- Number of node failures (restart count).
- Number of null frames received from node F.
- Number of CRC errors detected in messages from node F.
- Number of value domain errors of node F.
- Number of value domain errors of FG FTU (1st FTU).
- Number of injected faults.

The last statistics, the number of injected faults may not update the application, but the fault injector.

5.4 The Cluster Module

The modules containing the host application, the external sine wave generator, the fault injector, and the output registers are grouped in a separate module and linked together with the controller and channel modules. This enables to use the whole cluster as a separate runable module.

The module is application dependent and uses C-Sim simulation environment but no platform dependent modules, so it should be placed in the 'sa' layer. The name of the module is sa1swave.

The header file of the module exports only a structure providing all data entities of the simulation and an initialization function that creates all processes and sets their initial schedules.

```
typedef struct {
    SWAVE_HOST *host[4];
    CONTROLLER_MODEL *controller[4];
    CHANNEL_MODEL *channel[2];
    SINE_GENERATOR *generator;
    FAULT_INJECTOR *fault_injector;

    float mr, mfg;       /* Output registers */
} SINE_WAVE_APP;
```

```
/* Cluster initialization function. */
/* Returns 0 if successful. */
sine_wave_app_init(SINE_WAVE_APP *app);
```

5.4.1 Cluster Initialization

The cluster initialization proceeds in several steps. The steps are greatly application dependent so a checkpoint list specific to our example will be delivered in the next lines:

1. Initialization of protocol specification.
2. External process creation (Fault Injector, Sine Wave Generator).
3. Creation of four controllers and hosts.
4. Loading of MEDL tables into each controller.
5. Creation of two channels and binding of the channels to each controller.
6. Binding of each controller to its host.
7. Binding of the external input and output to the hosts.
8. Initial scheduling of the hosts and external processes.

5.5 Sine Wave Generator

The sine wave generator may be implemented as a separate process. Its action is strictly periodic and the process would be very simple, however it would consume system resources, depending on the granularity of the generated signal. If we forfeit the opportunity to generate an irregular signal we can compute the value of the signal from current model time.

A sine wave signal generated from the model time requires only one function returning the signal value and thus must not be implemented as a process and scheduled periodically. The function is provided by the module sa1swgen and looks like this:

```
float sine_generator_get_value()
{
    return sin(TIME() * frequency * 2 * PI) * amplitude;
}
```

The frequency parameter must be entered in the time units used by the model time, the amplitude is in our example equal to '1'.

5.6 User Interface

The user interface to our sine wave example is written in C++ Builder 5.0. To execute the simulation the modified C-Sim tool (section 3.4.1) and the multi-threaded C-Sim extension (3.4.2) are used.

5.6.1 Multi-threaded C-Sim Extension

To implement output, statistics and progress visualization we must create a new class descended from the predefined `TCSimThread` and redefine the virtual method `export_data()`. This method will after each simulation step copy the visualized state information into a local buffer. The new class is called `TSineWaveThread` and resides in the module named `valswthr`.

The output state buffer defines the following fields:

```
typedef struct {
    /* The following fields are implemented as round buffer */
    /* They contain snapshots for specified period of time */
    float round_sin_gen[ROUND_SIZE];  // sine generator value
    float round_sin_fg[ROUND_SIZE];   // FTU 1 output
    float round_sin_r[ROUND_SIZE];    // FTU 2 output
    PROTOCOL_STATE round_protocol_state[ROUND_SIZE][4];
    unsigned round_position;          // current round pos.
    double round_time;                // time of last snapshot

    /* the following field contain the current model state */
    double csim_time;                 // current model time
    float sin_gen;                    // sine generator output
    float sin_fg, sin_r;              // FTU output registers
    unsigned null_frame_count;        // null frames of F
    unsigned crc_error_count;         // CRC errors of F
    unsigned data_error_count;        // value errors of F
    unsigned cluster_fault_count;     // failures of R
    unsigned mismatch_count;          // FTU errors
    unsigned cni_fault_count          // CNI injected faults
    unsigned bus_fault_count;         // bus injected faults
    PROTOCOL_STATE protocol_state[4]; // state of controllers
    unsigned membership[4][4];        // memb. vectors
    unsigned null_frame[4][4];        // null.f. vectors
    float data[4][2][4];  // CNI data[node][channel][node_data]
    double time_dif[4];   // local clock difference
    int bg[4];            // BG state
} SNAPSHOT_BUFFER;
```

The first part of the buffer is formed by round buffer fields. These fields are used to display the history of the model. The size of the round determines the length of recorded history. The field `round_position` is the index to the last entry in the round buffers, the field `round_time` is the model time of the last state snapshot. The round buffer is updated with a periodicity of one slot, i.e. 250ms.

The initialization part of the simulation must be implemented by redefining the virtual method `init()` of our new class. The initialization is simple, as we can make use of the cluster module prepared in section 5.4:

```
void TSineWaveThread :: init()
{
    inic_mem();

    /* initialize the cluster */
    if (sine_wave_app_init())
        /* the cluster init. failed */
        error();
    else {
        /* initialization of the output buffer */
        memset(&buffer, 0, sizeof(SNAPSHOT_BUFFER));
        buffer.round_time = 0.0;
        /* load the buffer with current state of model */
        export_data();
    }
}
```

5.6.2 Visual Interface

Writing of the visual interface to the simulation is a task that is well covered in many publications and courses. The main program must include the module containing the modified simulation thread (`va1swthr`) and run the thread. The simulation will run autonomously and the user application can stop and resume the simulation flow at will. To receive results from the internal state buffer the user application must call the `get_data()` method.

The main file containing the sine wave example visual interface is named `va1main`. There are multiple other files used by the interface, they contain dialogs used for communication with the user.

5.7 User Manual

This section contains the basic information needed to compile and run the sine wave example application. A short description of the program will also be provided to supply some first user help.

The files distributed with this document are in the form of a zipped directory. The directory contains an executable file, multiple source files of the application, multiple source files of the simulated TTP/C protocol, and a subdirectory containing pictures used within the application.

5.7.1 Compilation and Linking

The sine wave project was written using the C++ Builer 5.0 development tool. The source files distributed with this document contain a project file that can be opened in the Builder environment using the open project command. The name of the project file is sw_vis.bpr. The project can be compiled from the Project menu by issuing the Build command. The result of the compilation (linking is done at the same time automatically) is an executable file labeled sw_vis.exe.

5.7.2 Running the Executable

The sine wave application sources and the Windows executable are a part of this project. To run the program, five files are needed, all of them residing in the same directory:

- sw_vis.exe: The program executable.
- medl00.txt: MEDL source file for node G1.
- medl01.txt: MEDL source file for node G2.
- medl02.txt: MEDL source file for node F.
- medl03.txt: MEDL source file for node R.

5.7.3 First User Help

The program starts up with a stopped simulation. To run the simulation the user must issue the Go command. The interface of the program contains several buttons. One of the buttons is labeled Help. New users should first click on this button and read the instructions that will be displayed in a new window.

The internal help of the application should cover all questions regarding the program user interface.

Chapter 6

Conclusion

During the work on this project a working simulation model of the TTP/C protocol was created. The model uses the modular structure described in this document and implements the fault injection method described in the previous chapters.

An example application was developed, that uses the C-based simulation libraries of the TTP/C protocol and allows user defined testing of the features offered by the TTP/C protocol. The sine wave example together with simulation based fault injection was used to verify a small set of protocol properties.

The tests performed on the designed TTP/C cluster clearly show the possibilities of time triggered protocols as TTP/C. The model application gives correct results even under massive fault injection.

Abbreviations and Terminology

C-Sim Terminology

Calendar The process scheduler uses a time ordered list of processes, a calendar.

Model Time A value internally stored within C-Sim. This time progresses only with process execution and does not depend on the real world time.

Process Program Any function with C-Sim specified header, that implements the flow of a process.

Process Type Any structure type derived from the predefined `PROCESS` type. Used for process state storage.

Type derivation The C language does not support inheritance, so another mechanism must be implemented. Derived structure is a structure, that contains all parent's fields at its beginning. Such structure can fully replace its parent as it contains all its fields.

TTA Terminology

BG Bus Guardian. A controller independent unit that controlls the bus access schedule to prevent faults in the time domain.

CO Controller-On is a field in the CNI memory. The host layer may turn the controller on and off with this field.

C-State All nodes within a cluster must agree about the current cluster state. The state is defined by several parameters globally called C-State.

Cluster cycle A periodically repeated sequence of TDMA rounds.

CNI Communication Network Interface is a well defined interface between the host layer of a node and the controller.

FT System Fault Tolerant system.

I-Frame Initialization Frame contains explicit C-State.

IFG Inter-frame Gap is required by physical controllers to separate messages and prepare the next message.

Macrotick Unit of synchronized cluster time.

MAE Mode Address Entry describes the length of a message and where the message should be stored.

MEDL Message Descriptor List defines the temporal properties of a TTP/C application.

Microtick Unit of local node time (not synchronized).

NBW Protocol Non-Blocking-Write Protocol is used to ensure data consistency in the CNI memory.

N-Frame Normal Frames are used for application data transmission.

RT System Real-Time system.

SCE Slot Control Entry defines the properties of a single slot within each TDMA round.

SRU Smallest Replaceable Unit. In a TTP/C Cluster on node failure the whole node is replaced.

TDMA Time Division Multiple Access is a bus access strategy that avoids collisions by statically assigned sending slots.

TDMA Round A periodically repeated fixed sequence of sending slots. Each TDMA round must have the same length, but different nodes can send different messages.

TMR Triple Modular Redundancy is used to ensure value domain fault tolerance in safety critical applications.

TTA Time Triggered Architecture.

TTP Time Triggered Protocol.

TTP/C TTP for class C real-time systems.

TTP/C Cluster An ensemble of TTP/C nodes connected through replicated channels.

TTP/C Node A hardware unit supporting TTP/C protocol, that can execute any user application.

TTP Build A software tool for host application development.

TTP OS A simple multitasking operating system designed for TTP/C nodes.

TTP Plan An offline planning tool designed for automated MEDL creation.

Other Abbreviations

Bit Arbitration Bus access strategy that ensures that the message with the highest priority can gain collision-free access. Used by the CAN protocol.

Byteflight A fault tolerant protocol that provides clock synchronization. Uses minislotting bus access strategy.

CAN A protocol based an CSMA/CA bus access strategy. Requires bit arbitration support from the physical medium.

CSMA/CD Carrier Sense Multiple Access with Collision Detection. A typical example is Ethernet.

CSMA/CA Carrier Sense Multiple Access with Collision Avoidance. A typical example is CAN.

DPRAM Dual Ported Random Access Memory. Ideal for interface, where two independent controllers may access the memory simultaneously.

FIT Project an EU project concerned with the validation of the FT properties of TTP/C.

UWB University of West Bohemia.

Bibliography

[TPS99] TTP/C Protocol – Specification of the Protocol, version 0.1 of 01-Feb-1999, TTTech Computertechnik GmbH

[CSM99] J. Hlavička, St. Racek, P. Herout (1999): C-Sim v. 4.1, Research Report DC-99-09, http://www.c-sim.zcu.cz,

[TPC00] Comparison CAN vs. Byteflight vs. TTP/C, Edition 3-Apr-2000, TTTech Computertechnik GmbH

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

[JEZ01] J. Ježek (2001): C-Based Reference Model of TTP/C Protocol, Diploma Thesis

[FIR00] 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

[FIW] FIT page at the University of West Boheamia, http://www.fit.zcu.cz

Appendix A

Sine Wave Application Listings

A.1 va1swthr Module

A.1.1 Header file

```
/*
 * 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
 *
 * va1swthr.h
 * C++ Builder C-Sim Thread for Sine Wave Application
 *
 * version 0.3 - 2001-03-17
 */

//---------------------------------------------------
#ifndef _VA1SWTHR_H
#define _VA1SWTHR_H
//---------------------------------------------------

#include "sa1swave.h"
#include "csimthrd.h"
```

```
#include "fp1cni.h"
#include "sa1fault.h"

#define ROUND_SIZE          640
#define SAMPLING_PERIOD     250

typedef struct {
  float round_sin_gen[ROUND_SIZE],
    round_sin_fg[ROUND_SIZE],
    round_sin_r[ROUND_SIZE];
  PROTOCOL_STATE round_protocol_state[ROUND_SIZE][4];
  unsigned round_position;
  double round_time;

  /* current data */
  double csim_time;
  float sin_gen, sin_fg, sin_r;
  unsigned null_frame_count,
    crc_error_count,
    data_error_count,
    cluster_fault_count;
  unsigned mismatch_count;
  unsigned cni_fault_count, bus_fault_count;
  PROTOCOL_STATE protocol_state[4];
  unsigned membership[4][4], null_frame[4][4];
  float data[4][2][4]; // [node][channel][node_data]

  double time_dif[4];
  int bg[4];

  unsigned cluster_cycle_counter;
} SNAPSHOT_BUFFER;

class TSineWaveThread : public TCSimThread {
private:
  SNAPSHOT_BUFFER buffer;
  unsigned f_channel_faults;
  double f_fault_lambda;
  FAULT_TYPES f_fault_type;
  unsigned f_fault_auto;
  unsigned f_fault_second;

protected:
  virtual void export_data();
```

```
  virtual void init();
public:
  SINE_WAVE_APP swave_app;  /* Process container */
  int vis_graph, vis_membnull,
    vis_state, vis_data, vis_results;

  TSineWaveThread()
    : TCSimThread(true) {};
  virtual void get_data(void *data);
  void inject_fault();

  __property unsigned channel_faults =
    {read = f_channel_faults, write = f_channel_faults};
  __property double fault_lambda =
    {read = f_fault_lambda, write = f_fault_lambda};
  __property FAULT_TYPES fault_type =
    {read = f_fault_type, write = f_fault_type};
  __property unsigned fault_auto =
    {read = f_fault_auto, write = f_fault_auto};
  __property unsigned fault_second =
    {read = f_fault_second, write = f_fault_second};

};

#endif
```

A.1.2 Module Implementation

```
/*
 * 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
 *
 * va1swthr.cpp
 * C++ Builder C-Sim Thread for Sine Wave Application
 *
 * version 0.4 - 2001-03-17
 */
```

```
//---------------------------------------------------------
#include <vcl.h>
#pragma hdrstop

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

#include "va1swthr.h"

#include "kr_csim.h"
#include "csimthrd.h"
#include "sa1swave.h"

//---------------------------------------------------------
#pragma package(smart_init)

void TSineWaveThread :: init()
{
  FAULT_INJECTOR *fi;

  inic_mem();

  if (sine_wave_app_init(&swave_app)) {
    MessageBox(0,
      "Probably one or more MEDL tables could
        not be read from file.\n"
      "Please check if the directory with the
        executable contains also four files 'medlxx.txt'\n\n"
      "The application will be now terminated.",
      "Sine Wave Application initialization error",
      MB_ICONERROR + MB_OK + MB_APPLMODAL);
    exit(-1);
    return;
  }

  memset(&buffer, 0, sizeof(SNAPSHOT_BUFFER));
  buffer.round_time = 0.0;

  vis_graph = 1;
  vis_membnull = 1;
  vis_state = 1;
  vis_data = 1;
  vis_results = 1;
```

```
  fi = swave_app.fault_injector;
  f_fault_auto = fi->enabled;
  f_fault_type = fi->fault_type;
  f_fault_lambda = fi->lambda;
  f_fault_second = fi->enabled2;

  export_data();

  Priority = tpLower;
}

void TSineWaveThread :: export_data()
{
  FAULT_INJECTOR *fi = swave_app.fault_injector;
  SWAVE_HOST *host_r = swave_app.host[3];
  LOCAL_DATA *ldata[4];
  CNI *cni[4];

  unsigned faults;
  int i, j;
  double d, e;
  float *pf;
  unsigned *pi;

  for (i = 0; i < 4; i++) {
    ldata[i] = &swave_app.controller[i]->ldata;
    cni[i] = &swave_app.controller[i]->cni;
  }

  faults = channel_faults;
  channel_faults = 0;
  if (faults > 0) {
    swave_app.channel[0]->faults_to_inject += faults;
    swave_app.channel[1]->faults_to_inject += faults;
  }

  if (fault_lambda != fi->lambda) {
    fi->lambda = fault_lambda;
    if (fault_auto)
      fault_injector_reschedule(fi, fault_lambda);
  }

  if (fault_auto != fi->enabled) {
```

```
    fi->enabled = fault_auto;
    fault_injector_reschedule(fi, fault_lambda);
  }

  if (fault_second != fi->enabled2) {
    fi->enabled2 = fault_second;
  }

  if (fault_type != fi->fault_type)
    fi->fault_type = fault_type;

  buffer.csim_time = csim_time;

  if (vis_data) {
    buffer.sin_gen = sine_generator_get_value(
        swave_app.generator);
      for (j = 0; j < 4; j++)
        for (i = 0; i < 4; i++) {
          pf = (float *)&cni_get_message_entry(
            cni[j], 6 * i)->data[0];
          pi = (unsigned *)pf;
          /* Exponent = 127 ---> NaN, decrease by "1" */
          if ((*pi & 0x7F800000) == 0x7F800000)
            *pi &= 0x7F000000;
          buffer.data[j][0][i] = *pf;

          pf = (float *)&cni_get_message_entry(
            cni[j], 6 * i + 3)->data[0];
          pi = (unsigned *)pf;
          /* Exponent = 127 ---> NaN, decrease by "1" */
          if ((*pi & 0x7F800000) == 0x7F800000)
            *pi &= 0x7F000000;
          buffer.data[j][1][i] = *pf;
      }
  }

  if (vis_results) {
    buffer.sin_fg = swave_app.mfg;
    buffer.sin_r = swave_app.mr;
    buffer.null_frame_count = host_r->null_frame_counter;
    buffer.crc_error_count = host_r->crc_error_counter;
    buffer.data_error_count = host_r->data_error_counter;
    buffer.cluster_fault_count =
      host_r->cluster_fault_counter;
```

```
    buffer.mismatch_count = host_r->mismatch_counter;
    buffer.cni_fault_count = fi->fault_count;
    buffer.bus_fault_count =
      swave_app.channel[0]->injected_faults;
  }

  if (vis_state) {
    for (j = 0; j < 4; j++) {
      i = cni_ps_get_protocol_state(
        &cni[j]->protocol_status);
      if (i > 12)
        i = 0;
      buffer.protocol_state[j] = (PROTOCOL_STATE)i;
    }

    d = ldata[3]->action_time;
    for (i = 0; i < 3; i++) {
      e = ldata[i]->action_time - d;
      /* The number 125(250) stands for
        half(whole) duration of the slot */
      if (e > 125)
        e -= 250;
      if (e < -125)
        e += 250;
      buffer.time_dif[i] = e * 100;
      /* e is in microseconds, 100 means 100 microticks
        per microsecond */
    }
    buffer.time_dif[3] = 0;

    for (i = 0; i < 4; i++)
      buffer.bg[i] =
        swave_app.controller[i]->bus_guardian->bg_state;
  }

  if (vis_membnull) {
    for (j = 0; j < 4; j++)
      for (i = 0; i < 4; i++) {
        buffer.membership[j][i] =
          c_state_get_membership(&cni[j]->c_state, i);
        buffer.null_frame[j][i] =
          cni_get_null_frame(cni[j], i);
      }
  }
```

```
  i = buffer.round_position;
  while (buffer.round_time <= csim_time) {
    if (vis_graph) {
      buffer.round_sin_gen[i] =
      sine_generator_get_value(swave_app.generator);
      buffer.round_sin_fg[i] = swave_app.mfg;
      buffer.round_sin_r[i] = swave_app.mr;
      for (j = 0; j < 4; j++)
        buffer.round_protocol_state[i][j] =
        (PROTOCOL_STATE)cni_ps_get_protocol_state(
        &cni[j]->protocol_status); }
    else {
      buffer.round_sin_gen[i] = 0;
      buffer.round_sin_fg[i] = 0;
      buffer.round_sin_r[i] = 0;
      memset(&buffer.round_protocol_state[i],
      0, sizeof(buffer.round_protocol_state[i]));
    }

    buffer.round_time += SAMPLING_PERIOD;
    i = (i + 1) % ROUND_SIZE;
  }
  buffer.round_position = i;

  buffer.cluster_cycle_counter =
    host_r->cluster_cycle_counter;
}

void TSineWaveThread :: get_data(void *data)
{
  EnterCriticalSection(&cs_export_data);
  memcpy(data, &buffer, sizeof(SNAPSHOT_BUFFER));
  LeaveCriticalSection(&cs_export_data);
}

void TSineWaveThread :: inject_fault()
{
  fault_injector_inject_fault(swave_app.fault_injector);
}
```

A.2 **sa1swgen** Module

A.2.1 Header file

```
/*
 * 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
 *
 * sa1swgen.h
 * SINE_GENERATOR c-sim proces
 *
 * version 0.4 - 2001-03-17
 */

#ifndef _SA1SWGEN_H
#define _SA1SWGEN_H

/***************************************************/

typedef struct {
    double amplitude;
        double frequency;
} SINE_GENERATOR;

/****************************************************/

/* Creates and initializes a new Sine Wave
    generator process.
   Returns pointer to the new process. */
SINE_GENERATOR *sine_generator_create();

/* Sets the generator's sine wave amplitude */
void sine_generator_set_amplitude(
  SINE_GENERATOR *sine_generator, double value);

/* Sets the generator's sine wave frequency
  [periods per second] */
void sine_generator_set_frequency(
```

```
  SINE_GENERATOR *sine_generator, double value);

/* Returns a value of the sine wave derived from
  C-Sim time */
float sine_generator_get_value(
  SINE_GENERATOR *sine_generator);

#endif
```

A.2.2 Module Implementation

```
/*
 * 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
 *
 * sa1swgen.c
 * SINE_GENERATOR c-sim proces
 *
 * version 0.4 - 2001-03-17
 */

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

#include "sa1swgen.h"

#include <math.h>
#include "kr_csim.h"

/*****************************************************/

#define PI 3.1415926536

/*****************************************************/

/* Global functions. */
```

```
/* Creates and initializes a new Sine Wave Generator
    process.
   Returns pointer to the new process. */
SINE_GENERATOR *sine_generator_create()
{
  SINE_GENERATOR *sine_generator =
    (SINE_GENERATOR *)malloc(sizeof(SINE_GENERATOR));

  sine_generator->amplitude = 1.0;
  sine_generator->frequency = 1.0;

  return sine_generator;
}

/* Sets the generator's sine wave amplitude */
void sine_generator_set_amplitude(
  SINE_GENERATOR *sine_generator, double value)
{
  sine_generator->amplitude = value;
}

/* Sets the generator's sine wave frequency
  [periods per second] */
void sine_generator_set_frequency(
  SINE_GENERATOR *sine_generator, double value)
{
  sine_generator->frequency = value;
}

/* Returns a value of the sine wave derived from C-Sim
  time */
float sine_generator_get_value(
  SINE_GENERATOR *sine_generator)
{
  return sin(CSIM_TIME()
    * sine_generator->frequency / 1000000 * 2 * PI) *
    sine_generator->amplitude;
}
```

A.3 sa1sinh Module

A.3.1 Header file

```
/*
 * 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
 *
 * sa1sinh.h
 * SIN_HOST c-sim proces
 *
 * version 0.5 - 2001-03-17
 */

#ifndef _SA1SINH_H
#define _SA1SINH_H

#include "kr_csim.h"
#include "fp1cni.h"
#include "sa1swgen.h"
#include "sp1ctrl.h"

/* SIN_HOST process data structure */
derived_process(swave_host)
  CNI *cni;
  SINE_GENERATOR *sine_generator;
  unsigned index;
  unsigned write_data_addr;
  unsigned read_data_addr[3];
  float *output_reg, *reference_reg;
  float last_value;
  unsigned max_iterations;
  unsigned meassured;

  double last_ui1;
  double last_ti1;
  unsigned interrupted;
  unsigned old_controller_lifesign;
```

```
  unsigned startup_complete;

  unsigned null_frame_counter;
  unsigned crc_error_counter;
  unsigned data_error_counter;
  unsigned cluster_fault_counter;
  unsigned mismatch_counter;

  unsigned cluster_cycle_counter;

end_derived(SWAVE_HOST);

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

/* Assigns the host to a controller.
  Stores controller's slot position and CNI address.
   Sets CNI Message Area write address and
    all read addresses.
   Registers host's interrupt function by the
   controller. */
void swave_host_attach_controller(
  SWAVE_HOST *swave_host, CONTROLLER *controller);

/* Assigns a Sine Wave Generator to the host. */
void swave_host_attach_generator(
  SWAVE_HOST *swave_host, SINE_GENERATOR *generator);

/* Assigns two external registers to the host.
  reg1 - output register for this FTU
  reg2 - output register of reference FTU */
void swave_host_attach_output(
  SWAVE_HOST *swave_host, float *reg1, float *reg2);

#endif
```

A.3.2 Module Implementation

```
/*
 * 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
 *
 * sa1sinh.c
 * SWAVE_HOST c-sim proces
 *
 * version 0.7 - 2001-04-10
 */

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

#include "sa1sinh.h"

#include <math.h>
#include "kr_csim.h"
#include "fp1cni.h"
#include "sp1cni.h"
#include "sa1swgen.h"
#include "sl1func.h"

/*********************************************

/* node F message entry offset */
#define NODE_F_MAE          12

/* maximum allowed difference between two float
  values to consider them equal */
#define EPSILON               0.001

#define host_hold(host, t) do { \
  print_debug((host, "Holding: %0.4f\n", t)); \
  hold(t); \
} while(0)

/*********************************************

static void controller_interrupt(void *ptr);
static int test_controller_lifesign(
  SWAVE_HOST *swave_host);
static int near_test(float a, float b);
```

```
/****************************************

/* Returns whether the two float values are
  nearly identical */
static int near_test(float a, float b)
{
  return (fabs(a - b) <= EPSILON);
}


/* Interrupt service function - called only
  by the asociated controller */
static void controller_interrupt(void *ptr)
{
#define swave_host ((SWAVE_HOST *)ptr)

  print_debug((swave_host,
    "Interrupt UI1 = %d, TI1 = %d, MI = %d ...",
    cni_is_get_ui1(&swave_host->cni->interrupt_status),
    cni_is_get_ti1(&swave_host->cni->interrupt_status),
    cni_is_get_mi(&swave_host->cni->interrupt_status)));

  swave_host->interrupted++;
  process_activate_delay((PROCESS *)swave_host, 0);

  print_debug((swave_host,"ok\n"));

#undef swave_host
}

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

  print_debug((swave_host,
    "Updating host lifesign..."));

  cni_set_host_lifesign(swave_host->cni,
    ~cni_get_controller_lifesign(swave_host->cni)
      & bit_mask(16));
```

```
  result = (cni_get_controller_lifesign(
    swave_host->cni) !=
    swave_host->old_controller_lifesign);
  swave_host->old_controller_lifesign =
    cni_get_controller_lifesign(swave_host->cni);

  print_debug((swave_host, "ok\n"));
  return result;
}

/***********************************************

float *get_float_ptr(
  SWAVE_HOST *host, unsigned addr, unsigned channel)
{
  return (float *)(&cni_get_message_entry(
    host->cni, addr + 3 * channel)->data[0]);
}

unsigned get_f_eif(SWAVE_HOST *host, unsigned channel)
{
  return cni_mae_get_eif(cni_get_message_entry(
    host->cni, NODE_F_MAE + 3 * channel));
}

float get_f_data(SWAVE_HOST *host, unsigned channel)
{
  float *pf;
  unsigned *pi;

  print_debug((host, "gd..."));
  pf = get_float_ptr(host, NODE_F_MAE, channel);
  pi = (unsigned *)pf;
  /* Exponent = 127 ---> NaN, decrease by "1" */
  if ((*pi & 0x7F800000) == 0x7F800000)
    *pi &= 0x7F000000;

  print_debug((host, "ok\n"));
  return *pf;
}

int eif_is_valid(
  SWAVE_HOST *host, unsigned index, unsigned channel)
{
```

```
  return cni_mae_get_eif(cni_get_message_entry(
    host->cni,
    host->read_data_addr[index] + 3 * channel)) ==
    EIF_CORRECT_FRAME;
}

void validate_eif(SWAVE_HOST *host)
{
  print_debug((host, "ve..."));
  cni_mae_set_eif(cni_get_message_entry(
    host->cni, host->write_data_addr + 0),
    EIF_CORRECT_FRAME);
  cni_mae_set_eif(cni_get_message_entry(
    host->cni, host->write_data_addr + 3),
    EIF_CORRECT_FRAME);
  print_debug((host, "ok\n"));
}

void set_data(SWAVE_HOST *host, float val)
{
  print_debug((host, "sd..."));
  *get_float_ptr(host, host->write_data_addr, 0) = val;
  *get_float_ptr(host, host->write_data_addr, 1) = val;
  print_debug((host, "ok\n"));
}

void init_data(SWAVE_HOST *host)
{
  /* Initialize CNI message area with different values */
  print_debug((host, "id..."));
  set_data(host, host->index * 0x222);
  *get_float_ptr(host, host->read_data_addr[0], 0) = 0xAAA;
  *get_float_ptr(host, host->read_data_addr[0], 1) = 0xAAA;
  *get_float_ptr(host, host->read_data_addr[1], 0) = 0x555;
  *get_float_ptr(host, host->read_data_addr[1], 1) = 0x555;
  *get_float_ptr(host, host->read_data_addr[2], 0) = 0x111;
  *get_float_ptr(host, host->read_data_addr[2], 1) = 0x111;
  print_debug((host, "ok\n"));
}

float get_data(SWAVE_HOST *host, unsigned index,
  unsigned channel)
{
  float *pf;
```

```
  unsigned *pi;

  print_debug((host, "gd..."));
  pf = get_float_ptr(host,
    host->read_data_addr[index], channel);
  pi = (unsigned *)pf;
  /* Exponent = 127 ---> NaN, decrease by "1" */
  if ((*pi & 0x7F800000) == 0x7F800000)
    *pi &= 0x7F000000;

  print_debug((host, "ok\n"));
  return *pf;
}

float get_valid_data(SWAVE_HOST *host,
  unsigned index, float bad)
{
  float x;

  print_debug((host, "gvd..."));
  if (eif_is_valid(host, index, 0)) {
    x = get_data(host, index, 0);
    if (eif_is_valid(host, index, 1)
      && x != get_data(host, index, 1))
      x = bad;
  }
  else {
    if (eif_is_valid(host, index, 1))
      x = get_data(host, index, 1);
    else
      x = bad;
  }

  print_debug((host, "ok\n"));
  return x;
}


/* SWAVE_HOST process - program code */
static s_program(SWAVE_HOST, swave_host_prog)
  for (;;) {
    int i;
    float a,b,c;
```

```
if (cni_get_co(p_my->cni) != CONTROLLER_ON) {
  /* controller is OFF: restart */
  print_debug((p_my, "Starting Controller\n"));
  if (p_my->startup_complete)
    p_my->cluster_fault_counter++;
  p_my->meassured = 0;
  cni_set_co(p_my->cni, 0);
  host_hold(p_my, 0);
  cni_set_co(p_my->cni, CONTROLLER_ON);
  host_hold(p_my, 0);
  if (cni_ps_get_controller_ready(
    &p_my->cni->protocol_status) != CONTROLLER_READY) {
    print_debug((p_my,
      "Error: Controller not ready, shutting down\n"));
    cni_set_co(p_my->cni, 0);
    host_hold(p_my, 1000);
    continue;
  }

  /* Controller is ready */
  init_data(p_my);
  validate_eif(p_my);

  print_debug((p_my,
    "Enabling MI (mode change) interrupt\n"));
  p_my->interrupted = 0;
  cni_ie_set_mi(&p_my->cni->interrupt_enable, 1);

  /* Update Host Life-Sign each round
    until C_State_Valid is set */
  p_my->old_controller_lifesign = 0xFFFF;
  host_hold(p_my, 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);
    host_hold(p_my, 1000);
  }

  if (cni_ps_get_c_state_valid(
    &p_my->cni->protocol_status)
```

```
          != C_STATE_VALID) {
        /* Controller did not reintegrate. */
        print_debug((p_my, "Too many wait
          iterations. Restarting!\n"));
        cni_set_co(p_my->cni, 0);
        host_hold(p_my, 1000);
        continue;
      }

      /* Controller is integrated
        and synchronized with cluster. */
      i = p_my->index - c_state_get_medl_position(
        &p_my->cni->c_state) % 4;
      if (i == 0) {
        test_controller_lifesign(p_my);
        /* My sending slot was now */
        i = 4;
      }
      else
        i = (4 + i) % 4;

      print_debug((p_my, "Enabling User
        and Time interrupts\n"));
      cni_ie_set_ui1(&p_my->cni->interrupt_enable, 1);
      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_ui1 = CSIM_TIME();
      p_my->last_ti1 = CSIM_TIME();
      p_my->startup_complete++;

      continue;
    }

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

      if (cni_is_get_mi(&p_my->cni->interrupt_status)) {
      /* Mode change interrupt */
        print_debug((p_my, "Handling Mode
          Change interrupt\n"));
        /* No action so far */
        cni_is_set_mi(&p_my->cni->interrupt_status, 0);
```

```
}

if (cni_is_get_ti1(&p_my->cni->interrupt_status)) {
  /* Time Interrupt 1: Update Host Lifesign */
  print_debug((p_my, "Handling Time interrupt 1\n"));
  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);
}

if (cni_is_get_ui1(&p_my->cni->interrupt_status)) {
  /* User Interrupt 1: Measuring or evaluating */
  print_debug((p_my, "Handling User interrupt 1\n"));
  p_my->last_ui1 = CSIM_TIME();
  if (c_state_get_medl_position(
    &p_my->cni->c_state) == 0) {
    /* First transmission slot: Measure sin value */
    p_my->meassured = 1;
    p_my->last_value = sine_generator_get_value(
      p_my->sine_generator);
    set_data(p_my, p_my->last_value);
    validate_eif(p_my);

    print_debug((p_my, "Measured value of Sinus:
      %0.6f\n", sine_generator_get_value(
        p_my->sine_generator)));

    if (!near_test(*p_my->output_reg,
      *p_my->reference_reg)) {
      p_my->mismatch_counter++;
      print_debug((p_my, "Mismatch counter
        = %u\n", p_my->mismatch_counter));
    }
  } else {
    /* Last transmission slot: Evaluate
      (vote) received data */
    p_my->cluster_cycle_counter++;
    if (p_my->meassured == 0)
      print_debug((p_my, "No meassured value,
        skipping voting!\n"));
    else {
```

```
        print_debug((p_my, "Voting\n"));
        p_my->meassured = 0;
        a = p_my->last_value;
        if (p_my->index < 3) {
          /* Nodes G1, G2, F */
          b = get_valid_data(p_my, 0, 0xAAA);
          c = get_valid_data(p_my, 1, 0x555);
            if (near_test(a, b) || near_test(a, c))
            *(p_my->output_reg) = a;
          else if (near_test(b, c) || near_test(b, a))
            *(p_my->output_reg) = b;
          else
            print_debug((p_my, "No matching sine
              values.\n"));
        } else {
          /* Node R */
          *(p_my->output_reg) = a;
        }

        print_debug((p_my, "Voted value of Sinus:
          %0.6f\n", *p_my->output_reg));
        a = *(p_my->output_reg);
        /* Test error conditions of node F */
        if (get_f_eif(p_my, 0) == EIF_NULL_FRAME
          || get_f_eif(p_my, 1) == EIF_NULL_FRAME)
          p_my->null_frame_counter++;
        else if (get_f_eif(p_my, 1) == EIF_CRC_ERROR
          || get_f_eif(p_my, 1) == EIF_CRC_ERROR)
          p_my->crc_error_counter++;
        else if (!near_test(get_f_data(p_my, 0), a)
          || !near_test(get_f_data(p_my, 1), a) )
          p_my->data_error_counter++;

        print_debug((p_my, "Counters: null_frame =
          %u, crc_error = %u, data_error = %u,
          cluster_fault = %u\n",
          p_my->null_frame_counter,
            p_my->crc_error_counter,
            p_my->data_error_counter,
            p_my->cluster_fault_counter));
      }
    }
    cni_is_set_ui1(&p_my->cni->interrupt_status, 0);
    print_debug((p_my, "End of User Interrupt 1\n"));
```

```
      } /* if (UI1) */

      p_my->interrupted = 0;
    } /* if (interrupted) */

    host_hold(p_my, 2000);

    if (CSIM_TIME() - p_my->last_ui1 > 4000.0
      || CSIM_TIME() - p_my->last_ti1 > 2000.0) {
      /* Interrupt timeout */
      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) {
        print_debug((p_my, "Timeout expired.
          Controller in LISTEN or COLD
          START state, restarting timeout\n"));
        p_my->last_ui1 = CSIM_TIME();
        p_my->last_ti1 = CSIM_TIME();
        test_controller_lifesign(p_my);
      } else {
        print_debug((p_my, "Error - Host timeout
          expired (no interrupt from controller)\n"));
        cni_set_co(p_my->cni, 0);
        host_hold(p_my, 1000);
      }
    }
  } /* for (;;) */
s_end_program

/**************************************************

/* Global functions. */

/* Creates and initializes a new SWAVE_HOST process.
   Returns pointer to the new process. */
SWAVE_HOST *swave_host_create(char *name)
{
  SWAVE_HOST *swave_host;

  swave_host = s_new_process(
    SWAVE_HOST, swave_host_prog);
  set_name(swave_host, name);
```

```
  swave_host->sine_generator = NULL;
  swave_host->output_reg = NULL;
  swave_host->reference_reg = NULL;

  swave_host->interrupted = 0;
  swave_host->meassured = 0;
  swave_host->null_frame_counter = 0;
  swave_host->crc_error_counter = 0;
  swave_host->data_error_counter = 0;
  swave_host->cluster_fault_counter = 0;
  swave_host->mismatch_counter = 0;
  swave_host->startup_complete = 0;

  swave_host->cluster_cycle_counter = 0;

  return swave_host;
}

/* Assigns the host to a controller. Stores
    controller's slot position and CNI address.
   Sets CNI Message Area write address and all
     read addresses.
   Registers host's interrupt function by the
     controller. */
void swave_host_attach_controller(
  SWAVE_HOST *swave_host, CONTROLLER *controller)
{
  swave_host->cni = &controller->cni;
  swave_host->index = logical_name_get_slot_position(
    &controller->medl.role_list.logical_name) - 1;

  controller_attach_host(controller,
    (void *)swave_host, controller_interrupt);

  swave_host->write_data_addr = swave_host->index * 6;

  switch (swave_host->index) {
  case 0:
    swave_host->read_data_addr[0] = 6;
    swave_host->read_data_addr[1] = 12;
    swave_host->read_data_addr[2] = 18;
    break;
  case 1:
```

```
    swave_host->read_data_addr[0] = 0;
    swave_host->read_data_addr[1] = 12;
    swave_host->read_data_addr[2] = 18;
    break;
  case 2:
    swave_host->read_data_addr[0] = 0;
    swave_host->read_data_addr[1] = 6;
    swave_host->read_data_addr[2] = 18;
    break;
  case 3:
    swave_host->read_data_addr[0] = 0;
    swave_host->read_data_addr[1] = 6;
    swave_host->read_data_addr[2] = 12;
    break;
  }
}

/* Assigns a Sine Wave Generator to the host. */
void swave_host_attach_generator(
  SWAVE_HOST *swave_host, SINE_GENERATOR *generator)
{
  swave_host->sine_generator = generator;
}

/* Assigns two external registers to the host.
  reg1 - output register for this FTU
  reg2 - output register of reference FTU */
void swave_host_attach_output(
  SWAVE_HOST *swave_host, float *reg1, float *reg2)
{
  swave_host->output_reg = reg1;
  swave_host->reference_reg = reg2;
}
```

A.4 **sa1swave** Module

A.4.1 Header file

```
/*
 * 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
 *
 * sa1swave.h
 * C++ Builder C-Sim Thread for Sine Wave Application
 *
 * version 0.1 - 2001-03-17
 */

#ifndef _SA1SWAVE_H
#define _SA1SWAVE_H

#include "fp1cni.h"
#include "sp1ctrl.h"
#include "sp1chan.h"
#include "sa1sinh.h"
#include "sa1fault.h"
#include "sa1swgen.h"

typedef struct {
  SWAVE_HOST *host[4];
  CONTROLLER_MODEL *controller[4];
  CHANNEL_MODEL *channel[2];
  SINE_GENERATOR *generator;
  FAULT_INJECTOR *fault_injector;

  float mr, mfg;    /* Output registers */
} SINE_WAVE_APP;

/* Creates and initializes all simulation processes.
   Returns nonzero code when somthing fails,
   zero otherwise. */
sine_wave_app_init(SINE_WAVE_APP *app);

#endif
```

A.4.2 Module Implementation

```
/*
 * 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
 *
 * sa1swave.cpp
 * Sine Wave application simulation
 *
 * version 0.2 - 2001-04-10
 */

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

#include "sa1swave.h"
#include "sp1ctrl.h"
#include "sp1chan.h"
#include "sa1swgen.h"
#include "sa1sinh.h"
#include "sa1fault.h"
#include "fp1mdlrd.h"
#include "sp1cni.h"
#include "sl1func.h"


/******************************************/

static double controller_drift[4] = {
  - 0.000002,
  + 0.000003,
  - 0.000001,
  + 0.000000
};

/*******************************************/

/* Creates and initializes all simulation processes.
   Returns nonzero code when somthing fails,
   zero otherwise. */
int sine_wave_app_init(SINE_WAVE_APP *app)
{
#define generator app->generator
```

```
#define channel app->channel
#define controller app->controller
#define host app->host
#define fault_injector app->fault_injector

  unsigned i;

  cni_module_initialize();

  /* Process creation. */
  generator = sine_generator_create();
  fault_injector = fault_injector_create(
    "Fault Injector");
  channel[0] = channel_create("Channel A");
  channel[1] = channel_create("Channel B");
  controller[0] = controller_create(
    "Controller G1");
  controller[1] = controller_create(
    "Controller G2");
  controller[2] = controller_create(
    "Controller F");
  controller[3] = controller_create(
    "Controller R");
  host[0] = swave_host_create("Host G1");
  host[1] = swave_host_create("Host G2");
  host[2] = swave_host_create("Host F");
  host[3] = swave_host_create("Host R");

  /* Sine Generator initialization. */
  sine_generator_set_amplitude(generator, 1);
  sine_generator_set_frequency(generator, 10);

  /* Channel initialization. */
  for (i = 0; i < 2; i++) {
    channel_set_bus_id((CHANNEL *)channel[i], i);
    channel_set_bg_error_callback((CHANNEL *)
      channel[i], controller_bg_error_callback);
  }

  /* Controller initialization. */
  for (i = 0; i < 4; i++) {
    controller_attach_channel((CONTROLLER *)
      controller[i], (CHANNEL *)channel[0], 0);
    controller_attach_channel((CONTROLLER *)
```

```
      controller[i], (CHANNEL *)channel[1], 1);
    controller_set_clock_drift((CONTROLLER *)
      controller[i], controller_drift[i]);
    if (medl_read_from_file(&((CONTROLLER *)
      controller[i])->medl, i))
      return 1;
  }

  /* Host initialization. */
  for (i = 0; i < 4; i++) {
    swave_host_attach_controller(host[i],
      (CONTROLLER *)controller[i]);
    swave_host_attach_generator(host[i], generator);
    swave_host_attach_output(host[i],
      (i < 3) ? &app->mfg : &app->mr,
      (i < 3) ? &app->mr : &app->mfg
    );
  }

  /* Fault Injector initialization. */
  fault_injector_attach_cni1(fault_injector,
    &controller[2]->cni);
  fault_injector_attach_cni2(fault_injector,
    &controller[0]->cni);

  /* Process activation. */
  for (i = 0; i < 4; i++)
    process_activate_delay((PROCESS *)host[i], 0);

  return 0;

#undef generator
#undef channel
#undef controller
#undef host
#undef fault_injector
}
```

A.5 MEDL Source Files

A.5.1 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 Sine Wave 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
0  mode_address_entry[1][0].cni_address_cha
0  mode_address_entry[1][0].dafla
0  mode_address_entry[1][0].cni_address_chb
0  mode_address_entry[1][0].daflb
1  mode_address_entry[1][0].ru1
0  mode_address_entry[1][0].ru2

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
0  mode_address_entry[1][1].ia
0  mode_address_entry[1][1].ib
0  mode_address_entry[1][1].eoc
6  mode_address_entry[1][1].cni_address_cha
```

```
3  mode_address_entry[1][1].dafla
9  mode_address_entry[1][1].cni_address_chb
3  mode_address_entry[1][1].daflb
0  mode_address_entry[1][1].ru1
0  mode_address_entry[1][1].ru2

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
0  mode_address_entry[1][2].cni_address_cha
0  mode_address_entry[1][2].dafla
0  mode_address_entry[1][2].cni_address_chb
0  mode_address_entry[1][2].daflb
0  mode_address_entry[1][2].ru1
0  mode_address_entry[1][2].ru2

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
0  mode_address_entry[1][3].ia
0  mode_address_entry[1][3].ib
0  mode_address_entry[1][3].eoc
18  mode_address_entry[1][3].cni_address_cha
3  mode_address_entry[1][3].dafla
21  mode_address_entry[1][3].cni_address_chb
3  mode_address_entry[1][3].daflb
0  mode_address_entry[1][3].ru1
1  mode_address_entry[1][3].ru2
```

```
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
0  mode_address_entry[1][4].ia
0  mode_address_entry[1][4].ib
0  mode_address_entry[1][4].eoc
0  mode_address_entry[1][4].cni_address_cha
3  mode_address_entry[1][4].dafla
3  mode_address_entry[1][4].cni_address_chb
3  mode_address_entry[1][4].daflb
0  mode_address_entry[1][4].ru1
0  mode_address_entry[1][4].ru2

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
0  mode_address_entry[1][5].cni_address_cha
0  mode_address_entry[1][5].dafla
0  mode_address_entry[1][5].cni_address_chb
0  mode_address_entry[1][5].daflb
0  mode_address_entry[1][5].ru1
0  mode_address_entry[1][5].ru2

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
0  mode_address_entry[1][6].ia
0  mode_address_entry[1][6].ib
0  mode_address_entry[1][6].eoc
12  mode_address_entry[1][6].cni_address_cha
3  mode_address_entry[1][6].dafla
15  mode_address_entry[1][6].cni_address_chb
3  mode_address_entry[1][6].daflb
0  mode_address_entry[1][6].ru1
0  mode_address_entry[1][6].ru2

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
0  mode_address_entry[1][7].cni_address_cha
0  mode_address_entry[1][7].dafla
0  mode_address_entry[1][7].cni_address_chb
0  mode_address_entry[1][7].daflb
1  mode_address_entry[1][7].ru1
1  mode_address_entry[1][7].ru2
```

A.5.2 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 Sine Wave 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
0  mode_address_entry[1][0].cni_address_cha
0  mode_address_entry[1][0].dafla
0  mode_address_entry[1][0].cni_address_chb
0  mode_address_entry[1][0].daflb
1  mode_address_entry[1][0].ru1
1  mode_address_entry[1][0].ru2

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
0  mode_address_entry[1][1].ia
0  mode_address_entry[1][1].ib
0  mode_address_entry[1][1].eoc
6  mode_address_entry[1][1].cni_address_cha
3  mode_address_entry[1][1].dafla
9  mode_address_entry[1][1].cni_address_chb
3  mode_address_entry[1][1].daflb
0  mode_address_entry[1][1].ru1
0  mode_address_entry[1][1].ru2

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
0  mode_address_entry[1][2].cni_address_cha
0  mode_address_entry[1][2].dafla
0  mode_address_entry[1][2].cni_address_chb
0  mode_address_entry[1][2].daflb
0  mode_address_entry[1][2].ru1
0  mode_address_entry[1][2].ru2

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
0  mode_address_entry[1][3].ia
0  mode_address_entry[1][3].ib
0  mode_address_entry[1][3].eoc
18  mode_address_entry[1][3].cni_address_cha
3  mode_address_entry[1][3].dafla
21  mode_address_entry[1][3].cni_address_chb
3  mode_address_entry[1][3].daflb
0  mode_address_entry[1][3].ru1
0  mode_address_entry[1][3].ru2

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
0  mode_address_entry[1][4].ia
0  mode_address_entry[1][4].ib
0  mode_address_entry[1][4].eoc
0  mode_address_entry[1][4].cni_address_cha
3  mode_address_entry[1][4].dafla
3  mode_address_entry[1][4].cni_address_chb
3  mode_address_entry[1][4].daflb
0  mode_address_entry[1][4].ru1
1  mode_address_entry[1][4].ru2

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
0  mode_address_entry[1][5].cni_address_cha
0  mode_address_entry[1][5].dafla
0  mode_address_entry[1][5].cni_address_chb
0  mode_address_entry[1][5].daflb
0  mode_address_entry[1][5].ru1
0  mode_address_entry[1][5].ru2

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
0  mode_address_entry[1][6].ia
0  mode_address_entry[1][6].ib
0  mode_address_entry[1][6].eoc
```

```
12  mode_address_entry[1][6].cni_address_cha
3  mode_address_entry[1][6].dafla
15  mode_address_entry[1][6].cni_address_chb
3  mode_address_entry[1][6].daflb
0  mode_address_entry[1][6].ru1
0  mode_address_entry[1][6].ru2

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
0  mode_address_entry[1][7].cni_address_cha
0  mode_address_entry[1][7].dafla
0  mode_address_entry[1][7].cni_address_chb
0  mode_address_entry[1][7].daflb
1  mode_address_entry[1][7].ru1
0  mode_address_entry[1][7].ru2
```

A.5.3 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 Sine Wave 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
0  mode_address_entry[1][0].cni_address_cha
0  mode_address_entry[1][0].dafla
0  mode_address_entry[1][0].cni_address_chb
0  mode_address_entry[1][0].daflb
1  mode_address_entry[1][0].ru1
0  mode_address_entry[1][0].ru2

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
0  mode_address_entry[1][1].ia
0  mode_address_entry[1][1].ib
0  mode_address_entry[1][1].eoc
6  mode_address_entry[1][1].cni_address_cha
3  mode_address_entry[1][1].dafla
9  mode_address_entry[1][1].cni_address_chb
3  mode_address_entry[1][1].daflb
0  mode_address_entry[1][1].ru1
1  mode_address_entry[1][1].ru2

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
```

```
0  mode_address_entry[1][2].cni_address_cha
0  mode_address_entry[1][2].dafla
0  mode_address_entry[1][2].cni_address_chb
0  mode_address_entry[1][2].daflb
0  mode_address_entry[1][2].ru1
0  mode_address_entry[1][2].ru2

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
0  mode_address_entry[1][3].ia
0  mode_address_entry[1][3].ib
0  mode_address_entry[1][3].eoc
18  mode_address_entry[1][3].cni_address_cha
3  mode_address_entry[1][3].dafla
21  mode_address_entry[1][3].cni_address_chb
3  mode_address_entry[1][3].daflb
0  mode_address_entry[1][3].ru1
0  mode_address_entry[1][3].ru2

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
0  mode_address_entry[1][4].ia
0  mode_address_entry[1][4].ib
0  mode_address_entry[1][4].eoc
0  mode_address_entry[1][4].cni_address_cha
3  mode_address_entry[1][4].dafla
3  mode_address_entry[1][4].cni_address_chb
3  mode_address_entry[1][4].daflb
```

```
0  mode_address_entry[1][4].ru1
0  mode_address_entry[1][4].ru2

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
0  mode_address_entry[1][5].cni_address_cha
0  mode_address_entry[1][5].dafla
0  mode_address_entry[1][5].cni_address_chb
0  mode_address_entry[1][5].daflb
0  mode_address_entry[1][5].ru1
1  mode_address_entry[1][5].ru2

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
0  mode_address_entry[1][6].ia
0  mode_address_entry[1][6].ib
0  mode_address_entry[1][6].eoc
12  mode_address_entry[1][6].cni_address_cha
3  mode_address_entry[1][6].dafla
15  mode_address_entry[1][6].cni_address_chb
3  mode_address_entry[1][6].daflb
0  mode_address_entry[1][6].ru1
0  mode_address_entry[1][6].ru2

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
0  mode_address_entry[1][7].cni_address_cha
0  mode_address_entry[1][7].dafla
0  mode_address_entry[1][7].cni_address_chb
0  mode_address_entry[1][7].daflb
1  mode_address_entry[1][7].ru1
0  mode_address_entry[1][7].ru2
```

A.5.4 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 Sine Wave 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
0  mode_address_entry[1][0].cni_address_cha
0  mode_address_entry[1][0].dafla
```

```
0  mode_address_entry[1][0].cni_address_chb
0  mode_address_entry[1][0].daflb
1  mode_address_entry[1][0].ru1
0  mode_address_entry[1][0].ru2

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
0  mode_address_entry[1][1].ia
0  mode_address_entry[1][1].ib
0  mode_address_entry[1][1].eoc
6  mode_address_entry[1][1].cni_address_cha
3  mode_address_entry[1][1].dafla
9  mode_address_entry[1][1].cni_address_chb
3  mode_address_entry[1][1].daflb
0  mode_address_entry[1][1].ru1
0  mode_address_entry[1][1].ru2

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
0  mode_address_entry[1][2].cni_address_cha
0  mode_address_entry[1][2].dafla
0  mode_address_entry[1][2].cni_address_chb
0  mode_address_entry[1][2].daflb
0  mode_address_entry[1][2].ru1
1  mode_address_entry[1][2].ru2

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
0  mode_address_entry[1][3].ia
0  mode_address_entry[1][3].ib
0  mode_address_entry[1][3].eoc
18  mode_address_entry[1][3].cni_address_cha
3  mode_address_entry[1][3].dafla
21  mode_address_entry[1][3].cni_address_chb
3  mode_address_entry[1][3].daflb
0  mode_address_entry[1][3].ru1
0  mode_address_entry[1][3].ru2

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
0  mode_address_entry[1][4].ia
0  mode_address_entry[1][4].ib
0  mode_address_entry[1][4].eoc
0  mode_address_entry[1][4].cni_address_cha
3  mode_address_entry[1][4].dafla
3  mode_address_entry[1][4].cni_address_chb
3  mode_address_entry[1][4].daflb
0  mode_address_entry[1][4].ru1
0  mode_address_entry[1][4].ru2

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
0  mode_address_entry[1][5].cni_address_cha
0  mode_address_entry[1][5].dafla
0  mode_address_entry[1][5].cni_address_chb
0  mode_address_entry[1][5].daflb
0  mode_address_entry[1][5].ru1
0  mode_address_entry[1][5].ru2

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
0  mode_address_entry[1][6].ia
0  mode_address_entry[1][6].ib
0  mode_address_entry[1][6].eoc
12  mode_address_entry[1][6].cni_address_cha
3  mode_address_entry[1][6].dafla
15  mode_address_entry[1][6].cni_address_chb
3  mode_address_entry[1][6].daflb
0  mode_address_entry[1][6].ru1
1  mode_address_entry[1][6].ru2

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
0  mode_address_entry[1][7].cni_address_cha
0  mode_address_entry[1][7].dafla
0  mode_address_entry[1][7].cni_address_chb
0  mode_address_entry[1][7].daflb
1  mode_address_entry[1][7].ru1
0  mode_address_entry[1][7].ru2
```

Appendix B

Sine Wave Application Screenshot

To provide a basic insight into the application interface a screenshot of the application running an experiment is included.

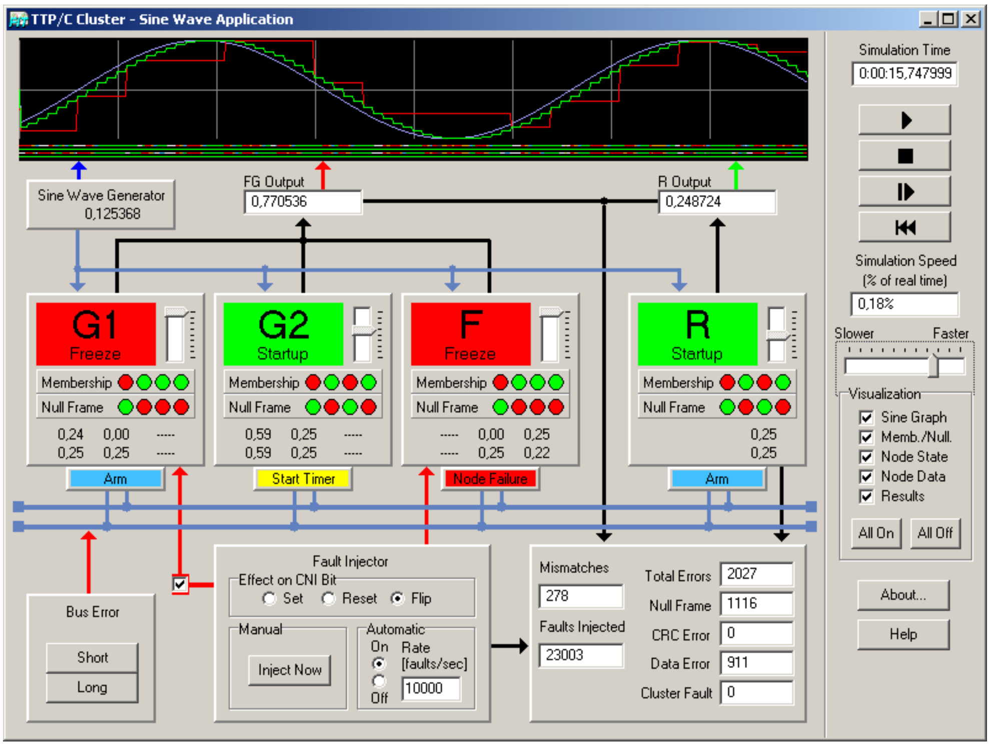

Figure B.1: Sine Wave Application Window