1 Introduction

This document contains listings of all source code files of the C-based reference model of TTP/C protocol, version 1.0. The files are presented in aplphabetical order.

The model was created at University of West Bohemia as part of an EU project Fault Injection for Time Triggered Architecture (IST-1999-10748).

2 Coding Style

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++.

2.1 Naming Convention

In the source codes we will distinguish the following constructs by their case:

- Type names and constants: 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()`.

All member fields of data structures that represent the TTP/C controller's CNI and MEDL memories (i.e. the CNI structure and the MEDL structure) have exactly the same name as they have in the written specification of the protocol (Specification version 0.1 of 01-Feb-1999), which was used as reference. Several exceptions to this rule exist mostly because the names of that fields are rather long and would produce unusably long variable names.

Acronyms were used instead of exact names in such cases. All of them are enlisted in table 1.

Acronym	Original Name
mfc	Membership Failure Counter
csct	Clock State Correction Term
mtd_cha	Measured Time Difference for Channel A
mtd_chb	Measured Time Difference for Channel B
app_id	Application Identification
app_version	Application Version Number
mmfc	Max Membership Failure Counter
membership_pos	Bit Position in Membership Vector

Table 1: List of Acronymised Field Names

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.

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.

2.4 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.

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. 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: 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.

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.

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 [**?**]. 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.

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. Because the current protocol specification contains only a vague description of the OS, this module is currently not implemented. All actions performed by the OS are left to the application. In the next version this module should be implemented.

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.

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.

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 and optionally a prepared functional specification of the application or TTP OS. Only ANSI C and C-Sim should be used to ensure maximum portability.

In the next version of TTP/C simulation, there may be only a single module in this layer, namely a simulation of TTP OS. Other application specific modules may be included only as an example. In the current implementation each application has a separate 'sa' module and the cluster definition resides in a module from the same layer.

sa1os contains the process simulating the TTP OS. The modules uses the functional specification of the OS together with the C-Sim library and must be pure ANSI C. So far there is no TTP OS implementation, the task of application management is left to the application developer.

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.

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.

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.