Table of Contents

4. Source Code Listings 10
 4.1. csimthrd.cpp 10
 4.2. csimthrd.h 12
 4.3. debug.cpp 13
 4.4. debug.h 15
 4.5. fl1func.cpp 16
 4.6. fl1func.h 17
 4.7. fp1busg.cpp 22
 4.8. fp1busg.h 26
 4.9. fp1cni.cpp 28
 4.10. fp1cni.h 32
 4.11. fp1const.h 37
 4.12. fp1cstat.cpp 39
 4.13. fp1cstat.h 40
 4.14. fp1ctrl.cpp 41
 4.15. fp1ctrl.h 70
 4.16. fp1frame.cpp 74
 4.17. fp1frame.h 77
 4.18. fp1chan.cpp 80
 4.19. fp1chan.h 83
 4.20. fp1ldata.cpp 85
 4.21. fp1ldata.h 86
 4.22. fp1lname.cpp 87
 4.23. fp1lname.h 88
 4.24. fp1mdlrd.cpp 89
 4.25. fp1mdlrd.h 102
 4.26. fp1medl.cpp 103
 4.27. fp1medl.h 105
 4.28. sa1fault.cpp 109
 4.29. sa1fault.h 114
 4.30. sl1func.cpp 115
 4.31. sl1func.h 117
 4.32. sp1busg.cpp 118
 4.33. sp1busg.h 121
 4.34. sp1cni.cpp 122
 4.35. sp1cni.h 124
 4.36. sp1ctrl.cpp 124
 4.37. sp1ctrl.h 136
 4.38. sp1chan.cpp 137
 4.39. sp1chan.h 140

4. Source Code Listings

4.1. csimthrd.cpp

```
/*
* 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
*
* csimthrd.cpp
* Win32 C-Sim thread
*
* version 0.4 - 2001-03-13
*/

#pragma hdrstop

#include <stdio.h>
#include <SysUtils.hpp>
#include "CSimThrd.h"
#include "kr_csim.h"

#pragma package(smart_init)

//-----------------------------------------------------------------------
__fastcall TCSimThread :: TCSimThread(bool create_suspended)
  : TThread(create_suspended)
{
  InitializeCriticalSection(&cs_export_data);
  f_slow_down = 0.0;
  remaining_slow_down = 0.0;
  f_tracing = false;
  f_zero_stepping = false;
  init();
}
//-----------------------------------------------------------------------
__fastcall TCSimThread :: ~TCSimThread()
{
  Terminate();
  Resume();
  WaitFor();

  clear_mem();
  DeleteCriticalSection(&cs_export_data);
}
//-----------------------------------------------------------------------
void TCSimThread :: go()
{
  tracing = false;
}
//-----------------------------------------------------------------------
void TCSimThread :: stop()
{
  tracing = true;
while (!Suspended)
    Sleep(1);
}
//---------------------------------------------------------------------------
void TCSimThread :: trace(bool wait)
{
  tracing = true;
  Resume();
while (wait && !Suspended)
    Sleep(1);
}
//---------------------------------------------------------------------------
void TCSimThread :: set_tracing(bool value)
{
  f_tracing = value;
if (!f_tracing) {
    Resume();
  }
}
//---------------------------------------------------------------------------
bool TCSimThread :: get_tracing()
{
bool result;

  result = f_tracing;
return result;
}
//---------------------------------------------------------------------------
double TCSimThread :: get_time()
{
return CSIM_TIME();
}
//---------------------------------------------------------------------------
void __fastcall TCSimThread :: Execute()
{
  CSIM_ERROR er;
double old_csim_time;
int steps;
unsigned wait_dur;

while (!Terminated) {
    while (!Terminated) {

      wait_dur = (remaining_slow_down += slow_down);

      if (wait_dur > 0) {
        remaining_slow_down -= wait_dur;
        Sleep(wait_dur);
      }

      old_csim_time = csim_time;
      steps = 1000;
      do {
        if (!step()) {
           error(&er);
         MessageBox(0, error_message[er.error_code], "CSim Error",
                MB_OK + MB_APPLMODAL + MB_ICONERROR);
```

```
          break;
        }
      EnterCriticalSection(&cs_export_data);
      export_data();
      LeaveCriticalSection(&cs_export_data);
      } while (!Terminated && !zero_stepping && (csim_time ==
old_csim_time) && (--steps > 0));

      if (!Terminated && tracing) {
        Suspend();
      }
    }
    if (!Terminated)
      Suspend();
  }
}
```

4.2. csimthrd.h

```
/*
* This software is part of an EU project:
*     FIT - Fault Injection for TTA
*     IST-1999-10748
*
* TTP/C - protocol v.0.1
*
* Design (c) TTTech & TU Vienna
* Implementation (c) UWB Pilsen & CTU Prague
*
* csimthrd.h
* Win32 C-Sim thread
*
* version 0.4 - 2001-03-13
*/

#ifndef _CSIMTHRD_H
#define _CSIMTHRD_H

#include <Classes.hpp>
//---------------------------------------------------------------------------
class TCSimThread : public TThread
{ private:
double f_slow_down, remaining_slow_down;
bool f_tracing;
bool was_suspended;
bool f_zero_stepping;

bool get_tracing();
void set_tracing(bool value);

double get_time();
bool has_finished();
protected:
  TRTLCriticalSection cs_export_data;

void __fastcall Execute();
virtual void init() = 0;
    // Model initialization - must redeclare
virtual void export_data()
    {};
public:
__property double slow_down = {read = f_slow_down, write = f_slow_down};
    // Wait between steps
__property bool tracing = {read = get_tracing, write = set_tracing};
    // Stop after each step?
__property double csim_time = {read = get_time};
    // Current simulation time
__property bool zero_stepping = {read = f_zero_stepping, write =
f_zero_stepping};
    // Is simulation complete?

__fastcall TCSimThread(bool create_suspended = true);
__fastcall ~TCSimThread();
void go();
    // Run simulation until FinishTime is reached (Disable Tracing)
void stop();
    // Stop simulation run (Enable Tracing)
void trace(bool wait = true);
    // Perform one step (process), by default wait for completetion

virtual void get_data(void *data)
    {};
};
//---------------------------------------------------------------------------

#endif
```

4.3. debug.cpp

```
/*
* 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
*
* debug.cpp
* Debuging output
*
* version 0.1 - 2001-03-13
*/

#include "debug.h"

#ifndef NDEBUG

#include "kr_csim.h"
#include "fp1ldata.h"
#include "sp1ctrl.h"

#define DEBUG_START               0 * 1000000.0
#define TIME_PRINTF_PRECISION     "10.4"

/* Returns the name of the IACK state. */
```

```
char *get_iack_state_name(unsigned iack)
{
switch (iack) {
    case IACK_OK:             return "OK         ";
    case IACK_SEND:           return "SEND       ";
    case IACK_CHECK1:         return "CHECK1     ";
    case IACK_CHECK2:         return "CHECK2     ";
    case IACK_ERROR:          return "ERROR      ";
    case IACK_DOUBLE_FAILURE: return "DBL_FAILURE";
  }
return "Unknown!";
}

/* Returns RC layer state name */

char *get_protocol_state_name(unsigned state_number)
{
switch (state_number) {
    case PS_INIT:        return "INIT       ";
    case PS_LISTEN:      return "LISTEN     ";
    case PS_COLD_START:  return "COLDSTART  ";
    case PS_STARTUP:     return "STARTUP    ";
    case PS_APPLICATION: return "APPLICATION";
    case PS_READY:       return "READY      ";
    case PS_PASSIVE:     return "PASSIVE    ";
    case PS_AWAIT:       return "AWAIT      ";
    case PS_SELFTEST:    return "SELFTEST   ";
    case PS_FREEZE:      return "FREEZE     ";
    case PS_DOWNLOAD:    return "DOWNLOAD   ";
  }
return "Unknown!";
}

void debug_printf(void *p, char *format, ...)
{
  va_list args;

if (CSIM_TIME() >= DEBUG_START) {
    #ifdef __CONSOLE__
      printf("%" TIME_PRINTF_PRECISION "f: %s: ", CSIM_TIME(), ((PROCESS
*)p)->name);
      va_start(args, format);
      vprintf(format, args);
      va_end(args);
    #else
      FILE *f;

      f = fopen("ttpc.out", "at");
      fprintf(f, "%" TIME_PRINTF_PRECISION "f: %s: ", CSIM_TIME(),
((PROCESS *)p)->name);
      va_start(args, format);
      vfprintf(f, format, args);
      va_end(args);
      fclose(f);
    #endif
  }
}
void debug_printf_no_time(char *format, ...)
```

```
{
  va_list args;

if (CSIM_TIME() >= DEBUG_START) {
    #ifdef __CONSOLE__
      va_start(args, format);
      vprintf(format, args);
      va_end(args);
    #else
      FILE *f;

      f = fopen("ttpc.out", "at");
      va_start(args, format);
      vfprintf(f, format, args);
      va_end(args);
      fclose(f);
    #endif
  }
}

#endif
```

4.4. debug.h

```
/*
* This software is part of an EU project:
*     FIT - Fault Injection for TTA
*     IST-1999-10748
*
* TTP/C - protocol v.0.1
*
* Design (c) TTTech & TU Vienna
* Implementation (c) UWB Pilsen & CTU Prague
*
* debug.h
* Debuging output
*
* version 0.1 - 2001-03-13
*/


#ifndef _DEBUG_H
#define _DEBUG_H

/*
*/
#define NDEBUG

#ifdef __BORLANDC__
  #pragma warn -sig
  #pragma warn -rch
  #pragma warn -eff
#endif


#ifdef assert
  #error assert.h must not be included before debug.h
#endif
```

```
/* TODO: remove this stuff: */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>


#ifdef NDEBUG
  /* Debugging OFF */
  #define print_debug(x)
  #define print_debug_no_time(x)

#else
  /* Debugging ON */
  #include <stdarg.h>
  #include "kr_csim.h"

  #ifdef unix
    #define __CONSOLE__
  #endif

void debug_printf(void *p, char *format, ...);
void debug_printf_no_time(char *format, ...);

  /* Returns the name of the IACK state. */
char *get_iack_state_name(unsigned iack);
  /* Returns RC layer state name */
char *get_protocol_state_name(unsigned state_number);


  /* Macros for printing debug messages,
     double parentheses must be used, eg. print_debug(("i = %u\n", i)); */
  #define print_debug(x) debug_printf x
  #define print_debug_no_time(x) debug_printf_no_time x

#endif


#endif
```

4.5. fl1func.cpp

```
/*
* 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
*
* fl1func.cpp
* Some useful global functions and macro definitions
*
* version 0.7 - 2001-03-13
*/
```

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

#include "fl1func.h"

/* get_crc does continuous calcullation of crc code. */

unsigned get_crc(unsigned *crc, unsigned polynomial, unsigned data,
unsigned bits)
{
while (bits > 0) {
    *crc = ((*crc << 1) & bit_mask(16)) ^ ((*crc >> 15) ^ (data & 1) ?
polynomial & bit_mask(16) : 0);
    bits--;
    data >>= 1;
   }

return *crc;
}
```

4.6. fl1func.h

```
/*
* This software is part of an EU project:
*     FIT - Fault Injection for TTA
*     IST-1999-10748
*
* TTP/C - protocol v.0.1
*
* Design (c) TTTech & TU Vienna
* Implementation (c) UWB Pilsen & CTU Prague
*
* fl1func.h
* Some useful global functions and macro definitions
*
* version 0.8 - 2001-03-17
*/

#ifndef _FL1FUNC_H
#define _FL1FUNC_H

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

/* Callback functions - named by their formal parameters */
typedef void (*PTR_INT_CALLBACK)(void *, unsigned); typedef void
(*PTR_CALLBACK)(void *);


/* Bit_mask(n) creates a binary mask with a given number of ones,
  i.e. bit_mask(4) expands to 15 (the lowest four bits are set). */
#define bit_mask(n) ((1 << (n)) - 1)

/* fract(x) returns fractional part of a number. */
#define fract(x) ((x) - floor(x))

/* Macro to create a prototype of function for getting simple structure
fields.
```

```
  i.e. get_function_prototype(FRAME, frame, type) expands to:

   unsigned frame_get_type(FRAME *_structure);
*/
#define get_function_prototype(struct_name, function_name,
field_name)       \
unsigned function_name##_get_##field_name(struct_name *_structure)

#define get_sig_function_prototype(struct_name, function_name,
field_name)       \
signed function_name##_get_##field_name(struct_name *_structure)


/* Macro to create a body of function for getting simple structure fields.
  i.e. get_function_body(FRAME, frame, type, 3) expands to:

   unsigned frame_get_type(FRAME *_structure)
   {
     return _structure->type & bit_mask(3);
   }
*/
#define get_function_body(struct_name, function_name, field_name,
bit_size)  \
   get_function_prototype(struct_name, function_name, field_name)       \
   {                                                                     \
     return _structure->field_name & bit_mask(bit_size);          \
   }

#define get_sig_function_body(struct_name, function_name, field_name,
bit_size)  \
   get_sig_function_prototype(struct_name, function_name, field_name)
\
   {                                                                     \
     signed x = _structure->field_name; \
     if ((x & (1 << (bit_size - 1))) == 0) \
       return x; \
     else \
       return x | ~bit_mask(bit_size); \
   }


/* Macro to create a prototype of function for getting vectored structure
fields.
  i.e. get_index_function_prototype(FRAME, frame, type) expands to:

   unsigned frame_get_type(FRAME *_structure, unsigned _index);
*/
#define get_index_function_prototype(struct_name, function_name,
field_name)                 \
unsigned function_name##_get_##field_name(struct_name *_structure, unsigned
_index)


/* Macro to create a body of function for getting vectored structure
fields.
  i.e. get_index_function_body(FRAME, frame, type, 3) expands to:

   unsigned frame_get_type(FRAME *_structure, unsigned _index);
   {
    return _structure->type[_index] & bit_mask(3);
  }
*/
#define get_index_function_body(struct_name, function_name, field_name,
bit_size)  \
  get_index_function_prototype(struct_name, function_name,
field_name)       \

{
\
    return _structure->field_name[_index] & bit_mask(bit_size);        \
  }



/* Macro to create a prototype of function for setting simple structure
fields.
  i.e. set_function_prototype(FRAME, frame, type) expands to:

  void frame_set_type(FRAME *_structure, unsigned _value);
*/
#define set_function_prototype(struct_name, function_name,
field_name)                    \
unsigned function_name##_set_##field_name(struct_name *_structure, unsigned
_value)

#define set_sig_function_prototype(struct_name, function_name,
field_name)                    \
signed function_name##_set_##field_name(struct_name *_structure, signed
_value)


/* Macro to create a body of function for setting simple structure fields.
  i.e. set_function_body(FRAME, frame, type, 3) expands to:

  void frame_set_type(FRAME *_structure, unsigned _value);
  {
    _structure->type = _value & bit_mask(3);
  }
*/
#define set_function_body(struct_name, function_name, field_name,
bit_size)  \
  set_function_prototype(struct_name, function_name, field_name)       \
  {                                                                     \
    assert(_value <= bit_mask(bit_size));                       \
    return _structure->field_name = _value & bit_mask(bit_size); \
  }

#define set_sig_function_body(struct_name, function_name, field_name,
bit_size)  \
  set_sig_function_prototype(struct_name, function_name, field_name)
\
  {                                                                     \
/*        assert(abs(_value) <= bit_mask(bit_size - 1));    */   \
    _structure->field_name = _value & bit_mask(bit_size); \
    return _value; \
  }
```

```
/* Macro to create a prototype of function for setting vectored structure
fields.
  i.e. set_index_function_prototype(FRAME, frame, type) expands to:

   void frame_set_type(FRAME *_structure, unsigned _index, unsigned _value);
*/
#define set_index_function_prototype(struct_name, function_name,
field_name)                                \
unsigned function_name##_set_##field_name(struct_name *_structure, unsigned
_index, unsigned _value)



/* Macro to create a body of function for setting vectored structure
fields.
  i.e. set_index_function_body(FRAME, frame, type, 3) expands to:

   void frame_set_type(FRAME *_structure, unsigned _index, unsigned _value);
   {
     _structure->type[_index] = _value & bit_mask(3);
   }
*/
#define set_index_function_body(struct_name, function_name, field_name,
bit_size)          \
   set_index_function_prototype(struct_name, function_name,
field_name)                \

{
       \
    assert(_value <= bit_mask(bit_size));                       \
    return _structure->field_name[_index] = _value &
bit_mask(bit_size);       \
   }



/* Macro to create a prototype of function for incrementing simple
structure fields.
  i.e. increment_function_prototype(FRAME, frame, type) expands to:

   void frame_increment_type(FRAME *_structure, unsigned _value);
*/
#define increment_function_prototype(struct_name, function_name,
field_name)                        \
unsigned function_name##_increment_##field_name(struct_name *_structure,
int _value)

#define increment_sig_function_prototype(struct_name, function_name,
field_name)                        \
signed function_name##_increment_##field_name(struct_name *_structure, int
_value)

/* Macro to create a body of function for incrementing simple structure
fields.
  i.e. increment_function_body(FRAME, frame, type, 3) expands to:

   void frame_increment_type(FRAME *_structure, unsigned _value);
   {
    unsigned _u = (frame_get_type(_structure) + _value) & bit_mask(3);
    frame_set_type(_structure, _u);
    return _u;
  }
*/
#define increment_function_body(struct_name, function_name, field_name,
bit_size)                           \
  increment_function_prototype(struct_name, function_name,
field_name)                                 \

{
                         \
    unsigned _u = (function_name##_get_##field_name(_structure) + _value) &
bit_mask(bit_size); \
    function_name##_set_##field_name(_structure,
_u);                                        \
    return
_u;
          \
  }

#define increment_sig_function_body(struct_name, function_name, field_name,
bit_size)                           \
  increment_sig_function_prototype(struct_name, function_name,
field_name)                                 \

{
                         \
    signed _u = function_name##_get_##field_name(_structure) + _value; \
    function_name##_set_##field_name(_structure,
_u);                                        \
    return
_u;
          \
  }


/* Macro to create prototypes for both getting and setting of simple
structure fields.
  see get_function_prototype() and set_function_prototype() macros */
#define declare_access_functions(struct_name, function_name, field_name)
\
  get_function_prototype(struct_name, function_name, field_name);     \
  set_function_prototype(struct_name, function_name, field_name);     \
  increment_function_prototype(struct_name, function_name, field_name)

#define declare_sig_access_functions(struct_name, function_name,
field_name)    \
  get_sig_function_prototype(struct_name, function_name, field_name);     \
  set_sig_function_prototype(struct_name, function_name, field_name);     \
  increment_sig_function_prototype(struct_name, function_name, field_name)



/* Macro to create bodies for both getting and setting of simple structure
fields.
  see get_function_body() and set_function_body() macros */
#define implement_access_functions(struct_name, function_name, field_name,
bit_size)  \
  get_function_body(struct_name, function_name, field_name,
bit_size)           \
  set_function_body(struct_name, function_name, field_name,
bit_size)           \
  increment_function_body(struct_name, function_name, field_name, bit_size)

#define implement_sig_access_functions(struct_name, function_name,
field_name, bit_size)  \
  get_sig_function_body(struct_name, function_name, field_name,
bit_size)           \
  set_sig_function_body(struct_name, function_name, field_name,
bit_size)           \
  increment_sig_function_body(struct_name, function_name, field_name,
bit_size)


/* Macro to create prototypes for both getting and setting of vectored
structure fields.
  see get_index_function_prototype() and set_index_function_prototype()
macros */
#define declare_index_access_functions(struct_name, function_name,
field_name) \
  get_index_function_prototype(struct_name, function_name, field_name);  \
  set_index_function_prototype(struct_name, function_name, field_name)



/* Macro to create bodies for both getting and setting of simple structure
fields.
  see get_index_function_body() and set_index_function_body() macros */
#define implement_index_access_functions(struct_name, function_name,
field_name, bit_size) \
  get_index_function_body(struct_name, function_name, field_name,
bit_size)          \
  set_index_function_body(struct_name, function_name, field_name, bit_size)


/* get_crc does continuous calcullation of crc code. */
unsigned get_crc(unsigned *crc, unsigned polynomial, unsigned data,
unsigned bits);



#endif /* #ifndef _FL1FUNC_H */
```

4.7. fp1busg.cpp

```
/*
* 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
*
* fp1busg.c
* Functions and definitions concerning the bus guardian
*
* version 0.6 - 2001-03-13
*/
```

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

#include "fp1busg.h"

#include "fl1func.h"
#include "fp1ctrl.h"

/* BUS_GUARDIAN field access functions. */
implement_access_functions(BUS_GUARDIAN, bus_guardian, tso, 16)
implement_access_functions(BUS_GUARDIAN, bus_guardian, tsd, 16)
implement_access_functions(BUS_GUARDIAN, bus_guardian, trr, 16)
implement_access_functions(BUS_GUARDIAN, bus_guardian, cmod, 2)
implement_access_functions(BUS_GUARDIAN, bus_guardian, error_code, 3)
implement_access_functions(BUS_GUARDIAN, bus_guardian, state_code, 4)
implement_access_functions(BUS_GUARDIAN, bus_guardian, lf, 1)
implement_access_functions(BUS_GUARDIAN, bus_guardian, er, 1)

/* Returns ht flag. */
get_function_body(BUS_GUARDIAN, bus_guardian, ht, 1)

/* set BUS_GUARDIAN halt bit, transit into failure state */

unsigned bus_guardian_set_ht(BUS_GUARDIAN *bus_guardian, unsigned value)
{
  value &= bit_mask(1);
  bus_guardian->ht = value;
if (value != 0)
    bus_guardian_raise_error(bus_guardian, BGEC_HALT);
return value;
}

/* Returns rc flag. */
get_function_body(BUS_GUARDIAN, bus_guardian, rc, 1)

/* set BUS_GUARDIAN reconfiguration bit, check for errors */

unsigned bus_guardian_set_rc(BUS_GUARDIAN *bus_guardian, unsigned value)
{
if (bus_guardian->rc == value)
    return bus_guardian->rc;
  bus_guardian->rc = value & bit_mask(1);

if (bus_guardian->rc == 1) {
    bus_guardian_set_lf(bus_guardian, 0);
  }
else
    bus_guardian->bg_state = BGS_ARM;

return bus_guardian->rc;
}

/* Returns am flag. */
get_function_body(BUS_GUARDIAN, bus_guardian, am, 1)

/* Sets am flag. */

unsigned bus_guardian_set_am(BUS_GUARDIAN *bus_guardian, unsigned value)
{
```

```
if (value == bus_guardian->am)
    return value;
  bus_guardian->am = value & bit_mask(1);

if (bus_guardian->am == 1) {
    if (bus_guardian_get_cmod(bus_guardian) == 0)
      bus_guardian_raise_error(bus_guardian,
BGEC_ARM_DURING_TRANSMISSION_DISABLED);
    else if (bus_guardian->bg_state != BGS_ARM)
      bus_guardian_raise_error(bus_guardian,
BGEC_ARM_DURING_BUS_ACCESS_CYCLE);
  }
else
    if (bus_guardian->bg_state != BGS_ENABLE)
      bus_guardian_raise_error(bus_guardian, BGEC_UNKNOWN_ERROR);
return bus_guardian->am;
}

/* switches bus guardian into node failure state and sets error code. */
void bus_guardian_raise_error(BUS_GUARDIAN *bus_guardian, unsigned
error_code)
{
  bus_guardian_set_state_code(bus_guardian, bus_guardian->bg_state);
  bus_guardian_set_error_code(bus_guardian, error_code);
  bus_guardian->am = 0;
  bus_guardian->lf = 0;
  bus_guardian->er = 1;
  bus_guardian->bg_state = BGS_NODE_FAILURE;
  controller_bg_error_callback(bus_guardian->controller);
}

/* Set the initial power-on values */

void bus_guardian_initialize(BUS_GUARDIAN *bus_guardian)
{
  bus_guardian->bg_state = BGS_INITIALIZE;
  bus_guardian->state_code = BGS_INITIALIZE;
  bus_guardian->tso = 0;
  bus_guardian->tsd = 0;
  bus_guardian->trr = 0;
  bus_guardian->am = 0;
  bus_guardian->ht = 0;
  bus_guardian->rc = 1;
  bus_guardian->cmod = 0;
  bus_guardian->lf = 0;
  bus_guardian->er = 0;
  bus_guardian->error_code = BGEC_NO_ERROR;
}

/* returns 1 if specified channel is enabled */

int bus_guardian_is_enabled(BUS_GUARDIAN *bus_guardian, unsigned channel)
{
return (bus_guardian_get_lf(bus_guardian) &&
    (((unsigned)1 << channel) & bus_guardian->saved_cmod) == ((unsigned)1
<< channel));

/*    if (bus_guardian_get_lf(bus_guardian) == 0)
```

```
    bus_guardian_raise_error(bus_guardian, BGEC_BUS_ACCESS_VIOLATION);
  else if ((channel_to_bit(channel) & bus_guardian->saved_cmod) !=
channel_to_bit(channel))
    bus_guardian_raise_error(bus_guardian, BGEC_BUS_ACCESS_VIOLATION);

  return bus_guardian_get_er(bus_guardian) != 0;*/
}

/* Reset and setup new bus guardian parameters. */

void bus_guardian_set_params(BUS_GUARDIAN *bus_guardian, MEDL_BGP *params)
{
  /* FIXME: check BG errors */
  bus_guardian_initialize(bus_guardian);
  bus_guardian_set_rc(bus_guardian, 1);
  bus_guardian_set_tso(bus_guardian, medl_bgp_get_bg_slot_offset(params));
  bus_guardian_set_tsd(bus_guardian,
medl_bgp_get_bg_slot_duration(params));
  bus_guardian_set_trr(bus_guardian,
medl_bgp_get_bg_remaining_round(params));
  bus_guardian_set_cmod(bus_guardian, 3);
  bus_guardian_set_rc(bus_guardian, 0);
}

/* Perform actions for init state. */

void bus_guardian_init_state(BUS_GUARDIAN *bus_guardian)
{
  /* Set power-on values. */
  bus_guardian_initialize(bus_guardian);
  /* Transit to configure state. */
  bus_guardian->bg_state = BGS_CONFIGURE;
}

/* Perform actions for configure state. */

void bus_guardian_configure_state(BUS_GUARDIAN *bus_guardian)
{
  /* Wait until the rc bit is reset. */
if (bus_guardian_get_rc(bus_guardian) == 0)
    bus_guardian->bg_state = BGS_ARM;
}

/* Perform actions for arm state. */

void bus_guardian_arm_state(BUS_GUARDIAN *bus_guardian)
{
if (bus_guardian_get_rc(bus_guardian) == 1) {
    bus_guardian->bg_state = BGS_CONFIGURE;
  } else if (bus_guardian_get_am(bus_guardian) == 1) {
    bus_guardian->bg_state = BGS_START_TIMER;
    bus_guardian->saved_cmod = bus_guardian_get_cmod(bus_guardian);
  }
  /* Wait for the arm signal. */
}

/* Perform actions for start_timer state. */
```

```
void bus_guardian_start_timer_state(BUS_GUARDIAN *bus_guardian)
{
  /* Wait the slot-offset-duration time and transit to enable state. */
  bus_guardian->bg_state = BGS_ENABLE;
  bus_guardian_set_lf(bus_guardian, 1);
}

/* Perform actions for enable state. */

void bus_guardian_enable_state(BUS_GUARDIAN *bus_guardian)
{
  /* Wait the slot-duration time and transit to disable state. */
  bus_guardian->bg_state = BGS_DISABLE;
  bus_guardian_set_lf(bus_guardian, 0);
if (bus_guardian_get_am(bus_guardian) != 0)
   bus_guardian_raise_error(bus_guardian, BGEC_UNKNOWN_ERROR);
}

/* Perform actions for disable state. */

void bus_guardian_disable_state(BUS_GUARDIAN *bus_guardian)
{
  /* Wait the remaining-round-duration time and transit back to arm state.
*/
  bus_guardian->bg_state = BGS_ARM;
}
```

4.8. fp1busg.h

```
/*
* This software is part of an EU project:
*     FIT - Fault Injection for TTA
*     IST-1999-10748
*
* TTP/C - protocol v.0.1
*
* Design (c) TTTech & TU Vienna
* Implementation (c) UWB Pilsen & CTU Prague
*
* sp1busg.h
* Functions and definitions concerning the bus guardian
*
* version 0.6 - 2001-03-13
*/

#ifndef _FP1BUSG_H
#define _FP1BUSG_H

#include "debug.h"

#include "fl1func.h"
#include "fp1medl.h"

/* Bus guardian tick length in microseconds */
#define BUS_GUARDIAN_CLK_PER_SEC    (1000000.0 * BUS_GUARDIAN_FREQUENCY /
BUS_GUARDIAN_DIVISOR)
struct bus_guardian;

/* Bus Guardian state codes */
```

```
typedef enum {
  BGS_INITIALIZE = 0,
  BGS_CONFIGURE,
  BGS_ARM,
  BGS_START_TIMER,
  BGS_ENABLE,
  BGS_DISABLE,
  BGS_NODE_FAILURE
} BUS_GUARDIAN_STATES;


/* Bus Guardian error codes */

typedef enum {
  BGEC_NO_ERROR = 0,
  BGEC_HALT = 1,
  BGEC_BUS_ACCESS_VIOLATION = 2,
  BGEC_ARM_DURING_BUS_ACCESS_CYCLE = 3,
  BGEC_ARM_DURING_TRANSMISSION_DISABLED = 5,
  BGEC_UNKNOWN_ERROR
} BUS_GUARDIAN_ERROR_CODES;
typedef void (*FUNC_BUS_GUARDIAN_ARM_SIGNAL)(struct bus_guardian
*bus_guardian, unsigned value);

/* Bus guardian protocol-defined data fields */
#define d_bus_guardian                                     \
void *controller;       /* The attached controller. */ \
                                                         \
unsigned bg_state;                                     \
  /* configuration registers */            \
unsigned tso;           /* slot offset duration */   \
unsigned tsd;           /* slot duration */      \
unsigned trr;           /* remaining round duration */ \
  /* control registers */               \
unsigned am;            /* arm signal */       \
unsigned ht;            /* halt */          \
unsigned rc;            /* reconfiguration */     \
unsigned cmod;          /* controller mode */     \
unsigned saved_cmod;                                   \
  /* status registers */               \
unsigned error_code;                                   \
unsigned state_code;                                   \
unsigned lf;            /* lifesign */        \
unsigned er;            /* error */         \
  /* control functions (methods) */          \
  FUNC_BUS_GUARDIAN_ARM_SIGNAL arm_signal

/* Bus guardian data structure */

typedef struct bus_guardian {
  d_bus_guardian;
} BUS_GUARDIAN;

/* BUS_GUARDIAN field access functions - set and get */
declare_access_functions(BUS_GUARDIAN, bus_guardian, tso);
declare_access_functions(BUS_GUARDIAN, bus_guardian, tsd);
declare_access_functions(BUS_GUARDIAN, bus_guardian, trr);
declare_access_functions(BUS_GUARDIAN, bus_guardian, cmod);
declare_access_functions(BUS_GUARDIAN, bus_guardian, error_code);
declare_access_functions(BUS_GUARDIAN, bus_guardian, state_code);
declare_access_functions(BUS_GUARDIAN, bus_guardian, lf);
declare_access_functions(BUS_GUARDIAN, bus_guardian, er);

get_function_prototype(BUS_GUARDIAN, bus_guardian, am);
get_function_prototype(BUS_GUARDIAN, bus_guardian, ht);
get_function_prototype(BUS_GUARDIAN, bus_guardian, rc);
set_function_prototype(BUS_GUARDIAN, bus_guardian, am);
set_function_prototype(BUS_GUARDIAN, bus_guardian, ht);
set_function_prototype(BUS_GUARDIAN, bus_guardian, rc);

/* switches bus guardian into node failure state and sets error code. */
void bus_guardian_raise_error(BUS_GUARDIAN *bus_guardian, unsigned
error_code);

/* returns 1 if specified channel is enabled */

int bus_guardian_is_enabled(BUS_GUARDIAN *bus_guardian, unsigned channel);

/* Set the initial power-on values */

void bus_guardian_initialize(BUS_GUARDIAN *bus_guardian);

/* Reset and setup new bus guardian parameters. */

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

/* Perform actions for init state. */

void bus_guardian_init_state(BUS_GUARDIAN *bus_guardian);

/* Perform actions for configure state. */

void bus_guardian_configure_state(BUS_GUARDIAN *bus_guardian);

/* Perform actions for arm state. */

void bus_guardian_arm_state(BUS_GUARDIAN *bus_guardian);

/* Perform actions for start_timer state. */

void bus_guardian_start_timer_state(BUS_GUARDIAN *bus_guardian);

/* Perform actions for enable state. */

void bus_guardian_enable_state(BUS_GUARDIAN *bus_guardian);

/* Perform actions for disable state. */

void bus_guardian_disable_state(BUS_GUARDIAN *bus_guardian);

#endif /* #ifndef _FP1BUSG_H */
```

4.9. fp1cni.cpp

```
/*
* 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
*
* fp1cni.c
* Functions and definitions concerning the CNI structure
*
* version 0.11 - 2001-03-13
*/

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

#include "fp1cni.h"

#include <string.h>
#include "fl1func.h"

/* CNI_PS (protocol status) field access functions - get and set */
implement_access_functions(CNI_PS, cni_ps, accept_counter, 8)
implement_access_functions(CNI_PS, cni_ps, failed_counter, 8)
implement_access_functions(CNI_PS, cni_ps, invalid_counter, 8)
implement_access_functions(CNI_PS, cni_ps, controller_ready, 8)
implement_access_functions(CNI_PS, cni_ps, mfc, 8)
implement_access_functions(CNI_PS, cni_ps, i_frame_switch, 8)
  /* FIXME: csct get function is bad: test last_clock_access!!! */
implement_sig_access_functions(CNI_PS, cni_ps, csct, 8)
implement_access_functions(CNI_PS, cni_ps, c_state_valid, 8)
implement_access_functions(CNI_PS, cni_ps, protocol_state, 4)
implement_access_functions(CNI_PS, cni_ps, bl, 1)
implement_access_functions(CNI_PS, cni_ps, be, 1)
implement_access_functions(CNI_PS, cni_ps, ae, 1)
implement_access_functions(CNI_PS, cni_ps, se, 1)
implement_access_functions(CNI_PS, cni_ps, me, 1)
implement_access_functions(CNI_PS, cni_ps, mc, 1)
implement_access_functions(CNI_PS, cni_ps, cb, 1)
implement_access_functions(CNI_PS, cni_ps, dc, 1)
implement_access_functions(CNI_PS, cni_ps, so, 1)
implement_access_functions(CNI_PS, cni_ps, mv, 1)
implement_access_functions(CNI_PS, cni_ps, ne, 1)

/* CNI_FD (frame diagnosis) field access functions - get and set */
implement_access_functions(CNI_FD, cni_fd, eif_cha, 4)
implement_access_functions(CNI_FD, cni_fd, eif_chb, 4)
implement_sig_access_functions(CNI_FD, cni_fd, mtd_cha, 8)
implement_sig_access_functions(CNI_FD, cni_fd, mtd_chb, 8)

/* CNI_PERS (personality) field access functions - get and set */
implement_access_functions(CNI_PERS, cni_pers, physical_sru_name, 8)
implement_access_functions(CNI_PERS, cni_pers, controller_version, 8)
implement_access_functions(CNI_PERS, cni_pers, app_id, 8)
implement_access_functions(CNI_PERS, cni_pers, app_version, 8)

/* CNI_IS (interrupt status) field access functions - get and set */
implement_access_functions(CNI_IS, cni_is, be, 1);
implement_access_functions(CNI_IS, cni_is, pe, 1);
implement_access_functions(CNI_IS, cni_is, he, 1);
implement_access_functions(CNI_IS, cni_is, ml, 1);
implement_access_functions(CNI_IS, cni_is, mc, 1);
implement_access_functions(CNI_IS, cni_is, mi, 1);
implement_access_functions(CNI_IS, cni_is, ui1, 1);
implement_access_functions(CNI_IS, cni_is, ui2, 1);
implement_access_functions(CNI_IS, cni_is, ti1, 1);
implement_access_functions(CNI_IS, cni_is, ti2, 1);

/* CNI_IE (interrupt enable) field access functions - get and set */
implement_access_functions(CNI_IE, cni_ie, be, 1);
implement_access_functions(CNI_IE, cni_ie, pe, 1);
implement_access_functions(CNI_IE, cni_ie, he, 1);
implement_access_functions(CNI_IE, cni_ie, ml, 1);
implement_access_functions(CNI_IE, cni_ie, mc, 1);
implement_access_functions(CNI_IE, cni_ie, mi, 1);
implement_access_functions(CNI_IE, cni_ie, ui1, 1);
implement_access_functions(CNI_IE, cni_ie, ui2, 1);

/* Enable Timer interrupt */
get_function_body(CNI_IE, cni_ie, ti1, 1);
get_function_body(CNI_IE, cni_ie, ti2, 1);
/*increment_function_body(CNI_IE, cni_ie, ti1, 1);
increment_function_body(CNI_IE, cni_ie, ti2, 1);*/
FUNC_CNI_IE_SET_TI cni_ie_set_ti1;
FUNC_CNI_IE_SET_TI cni_ie_set_ti2;


/* CNI_BS (BIST status) field access functions - get and set */
implement_access_functions(CNI_BS, cni_bs, sc, 1);
implement_access_functions(CNI_BS, cni_bs, av, 1);
implement_access_functions(CNI_BS, cni_bs, br, 1);
implement_access_functions(CNI_BS, cni_bs, rm, 1);
implement_access_functions(CNI_BS, cni_bs, rf, 1);
implement_access_functions(CNI_BS, cni_bs, wd, 1);

/* CNI_BE (bist enable) field access functions - get and set */
implement_access_functions(CNI_BE, cni_be, rm, 1);
implement_access_functions(CNI_BE, cni_be, rf, 1);

/* CNI_MAE (message area entry) field access functions - get and set */
implement_access_functions(CNI_MAE, cni_mae, ccf, 4);
implement_access_functions(CNI_MAE, cni_mae, eif, 4);
implement_index_access_functions(CNI_MAE, cni_mae, data, 8);

/* Copy message data from the frame to the message area entry. */

void cni_mae_extract_frame_data(CNI_MAE* mae, FRAME *frame, unsigned
data_length)
{
unsigned i;

for (i = 0; i < data_length; i++)
    mae->data[i] = frame->frame.n.application_data[i];
}

/* CNI (comunication network interface) field access functions - get and
set */
implement_access_functions(CNI, cni, ccf, 16)
implement_index_access_functions(CNI, cni, null_frame, 1)
implement_access_functions(CNI, cni, controller_lifesign, 16)
implement_access_functions(CNI, cni, host_lifesign, 16)
implement_sig_access_functions(CNI, cni, external_rate_correction, 16)
implement_index_access_functions(CNI, cni, mode_change_request, 1)
implement_access_functions(CNI, cni, time_overflow_counter, 16)
```

```
/* Timer access functions */
get_function_body(CNI, cni, timer1, 16)
get_function_body(CNI, cni, timer2, 16)
increment_function_body(CNI, cni, timer1, 16)
increment_function_body(CNI, cni, timer2, 16)
FUNC_CNI_SET_TIMER cni_set_timer1;
FUNC_CNI_SET_TIMER cni_set_timer2;

/* The "set" and "increment" functions for cluster_time are the normal
ones. */
set_function_body(CNI, cni, cluster_time, 16)
increment_function_body(CNI, cni, cluster_time, 16)
FUNC_CNI_GET_CLUSTER_TIME cni_get_cluster_time;

/* The "get" function for co is the normal one. */
get_function_body(CNI, cni, co, 16)
FUNC_CNI_SET_CO cni_set_co;

/* The "get" function for ca is the normal one. */
get_function_body(CNI, cni, ca, 16)
FUNC_CNI_SET_CA cni_set_ca;


/* Other functions */

/* This function provides easy access to the cni message area by the word
offset. */
CNI_MAE *cni_get_message_entry(CNI *cni, unsigned offset)
{
return (CNI_MAE *)( (char *)(&cni->cni_msg_base[0]) + 2 * offset);
}


/* Erase all fields in CNI except the CO field */

void cni_clear_mem(CNI *cni)
{
  memset(&cni->ccf, 0, sizeof(cni->ccf));
  memset(&cni->c_state, 0, sizeof(cni->c_state));
  memset(&cni->protocol_status, 0, sizeof(cni->protocol_status));
  memset(&cni->frame_diagnosis, 0, sizeof(cni->frame_diagnosis));
  memset(&cni->personality, 0, sizeof(cni->personality));
  memset(&cni->controller_lifesign, 0, sizeof(cni->controller_lifesign));
  memset(&cni->assertion_code, 0, sizeof(cni->assertion_code));
  memset(&cni->cluster_time, 0, sizeof(cni->cluster_time));
  memset(&cni->interrupt_status, 0, sizeof(cni->interrupt_status));
  memset(&cni->bist_status, 0, sizeof(cni->bist_status));
  memset(&cni->host_lifesign, 0, sizeof(cni->host_lifesign));
  memset(&cni->external_rate_correction, 0, sizeof(cni-
>external_rate_correction));
  memset(&cni->mode_change_request[3], 0, sizeof(cni-
>mode_change_request));
  memset(&cni->ca, 0, sizeof(cni->ca));
  memset(&cni->bist_enable, 0, sizeof(cni->bist_enable));
  memset(&cni->timer1, 0, sizeof(cni->timer1));
  memset(&cni->timer2, 0, sizeof(cni->timer2));
  memset(&cni->interrupt_enable, 0, sizeof(cni->interrupt_enable));
  memset(&cni->time_overflow_counter, 0, sizeof(cni-
>time_overflow_counter));
  memset(&cni->cni_msg_base, 0, sizeof(cni->cni_msg_base));
}

/* Erase all fields in CNI */

void cni_clear_status_fields(CNI *cni)
{
  memset(&cni->ccf, 0, sizeof(cni->ccf));
  memset(&cni->c_state, 0, sizeof(cni->c_state));
  memset(&cni->protocol_status, 0, sizeof(cni->protocol_status));
  memset(&cni->frame_diagnosis, 0, sizeof(cni->frame_diagnosis));
  memset(&cni->personality, 0, sizeof(cni->personality));
  memset(&cni->controller_lifesign, 0, sizeof(cni->controller_lifesign));
  memset(&cni->assertion_code, 0, sizeof(cni->assertion_code));
  memset(&cni->cluster_time, 0, sizeof(cni->cluster_time));
  memset(&cni->interrupt_status, 0, sizeof(cni->interrupt_status));
  memset(&cni->bist_status, 0, sizeof(cni->bist_status));
}
```

4.10. fp1cni.h

```
/*
* This software is part of an EU project:
*     FIT - Fault Injection for TTA
*     IST-1999-10748
*
* TTP/C - protocol v.0.1
*
* Design (c) TTTech & TU Vienna
* Implementation (c) UWB Pilsen & CTU Prague
*
* fp1cni.h
* Functions and definitions concerning the CNI structure
*
* version 0.10 - 2001-03-13
*/

#ifndef _FP1CNI_H
#define _FP1CNI_H

#include "fl1func.h"
#include "fp1const.h"
#include "fp1cstat.h"
#include "fp1lname.h"
#include "fp1frame.h"

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

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

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

/* CNI_PS field access functions - get and set */
declare_access_functions(CNI_PS, cni_ps, accept_counter);
declare_access_functions(CNI_PS, cni_ps, failed_counter);
declare_access_functions(CNI_PS, cni_ps, invalid_counter);
declare_access_functions(CNI_PS, cni_ps, controller_ready);
declare_access_functions(CNI_PS, cni_ps, mfc);
declare_access_functions(CNI_PS, cni_ps, i_frame_switch);
declare_sig_access_functions(CNI_PS, cni_ps, csct);
declare_access_functions(CNI_PS, cni_ps, c_state_valid);
declare_access_functions(CNI_PS, cni_ps, protocol_state);
declare_access_functions(CNI_PS, cni_ps, bl);
declare_access_functions(CNI_PS, cni_ps, be);
declare_access_functions(CNI_PS, cni_ps, ae);
declare_access_functions(CNI_PS, cni_ps, se);
declare_access_functions(CNI_PS, cni_ps, me);
declare_access_functions(CNI_PS, cni_ps, mc);
declare_access_functions(CNI_PS, cni_ps, cb);
declare_access_functions(CNI_PS, cni_ps, dc);
declare_access_functions(CNI_PS, cni_ps, so);
declare_access_functions(CNI_PS, cni_ps, mv);
declare_access_functions(CNI_PS, cni_ps, ne);

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

/* Possible values for error indication fields */
typedef enum {
  EIF_NULL_FRAME         = 0x00,
  EIF_INVALID_FRAME     = 0x02,
  EIF_CRC_ERROR         = 0x04,
  EIF_CORRECT_FRAME     = 0x08,
  EIF_NOISE        = 0x01
} CNI_EIF_VALUES;

/* CNI - frame diagnosis field structure */
typedef struct {
  FRAME_HEADER header_cha;
unsigned eif_cha;        /* error indication flag for channel A */
  FRAME_HEADER header_chb;
unsigned eif_chb;        /* error indication flag for channel B */
unsigned mtd_cha;                /* measured time difference cha */
unsigned mtd_chb;                /* measured time difference chb */
} CNI_FD;

/* CNI_FD field access functions - get and set */
declare_access_functions(CNI_FD, cni_fd, eif_cha);
declare_access_functions(CNI_FD, cni_fd, eif_chb);
declare_sig_access_functions(CNI_FD, cni_fd, mtd_cha);
declare_sig_access_functions(CNI_FD, cni_fd, mtd_chb);

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

/* CNI - personality field structure */
typedef struct {
unsigned physical_sru_name;
unsigned controller_version;
unsigned app_id;
unsigned app_version;
  LOGICAL_NAME logical_name;
} CNI_PERS;

/* CNI_PERS field access functions - get and set */
declare_access_functions(CNI_PERS, cni_pers, physical_sru_name);
declare_access_functions(CNI_PERS, cni_pers, controller_version);
declare_access_functions(CNI_PERS, cni_pers, app_id);
declare_access_functions(CNI_PERS, cni_pers, app_version);

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

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

/* CNI_IS field access functions - get and set */
declare_access_functions(CNI_IS, cni_is, be);
declare_access_functions(CNI_IS, cni_is, pe);
declare_access_functions(CNI_IS, cni_is, he);
declare_access_functions(CNI_IS, cni_is, ml);
declare_access_functions(CNI_IS, cni_is, mc);
declare_access_functions(CNI_IS, cni_is, mi);
declare_access_functions(CNI_IS, cni_is, ui1);
declare_access_functions(CNI_IS, cni_is, ui2);
declare_access_functions(CNI_IS, cni_is, ti1);
declare_access_functions(CNI_IS, cni_is, ti2);
typedef CNI_IS CNI_IE;

/* CNI_IE field access functions - get and set */
declare_access_functions(CNI_IE, cni_ie, be);
declare_access_functions(CNI_IE, cni_ie, pe);
declare_access_functions(CNI_IE, cni_ie, he);
declare_access_functions(CNI_IE, cni_ie, ml);
declare_access_functions(CNI_IE, cni_ie, mc);
declare_access_functions(CNI_IE, cni_ie, mi);
declare_access_functions(CNI_IE, cni_ie, ui1);
declare_access_functions(CNI_IE, cni_ie, ui2);

/* Timer interrupt enable */
get_function_prototype(CNI_IE, cni_ie, ti1);
get_function_prototype(CNI_IE, cni_ie, ti2);
/*increment_function_prototype(CNI_IE, cni_ie, ti1);
```

```
increment_function_prototype(CNI_IE, cni_ie, ti2);*/

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

/* CNI - BIST status field structure */
typedef struct {
unsigned sc;        /* selftest complete */
unsigned av;        /* assertion violation */
unsigned br;        /* invalid BIST request */
unsigned rm;        /* RAM error */
unsigned rf;        /* register file error */
unsigned wd;        /* watchdog error */
}  CNI_BS;

/* CNI_BS field access functions - get and set */
declare_access_functions(CNI_BS, cni_bs, sc);
declare_access_functions(CNI_BS, cni_bs, av);
declare_access_functions(CNI_BS, cni_bs, br);
declare_access_functions(CNI_BS, cni_bs, rm);
declare_access_functions(CNI_BS, cni_bs, rf);
declare_access_functions(CNI_BS, cni_bs, wd);

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

/* CNI - BIST enable field structure */
typedef struct {
unsigned rm;        /* RAM error */
unsigned rf;        /* register file error */
} CNI_BE;

/* CNI_BE field access functions - get and set */
declare_access_functions(CNI_BE, cni_be, rm);
declare_access_functions(CNI_BE, cni_be, rf);

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

/* This constant represents the maximum reasonable size of the cni message
area. */
#define CNI_MSG_AREA_SIZE (sizeof(CNI_MAE) * 2 * 512)

/* CNI - Message area entry */
typedef struct {
  /* message status field */
char not_used;
char ccf:4;        /* concurrency control field */
char eif:4;        /* error indication field */
  /* message data field */
char data[16];
} CNI_MAE;

/* CNI_MAE field access functions - get and set */
declare_access_functions(CNI_MAE, cni_mae, ccf);
declare_access_functions(CNI_MAE, cni_mae, eif);
declare_index_access_functions(CNI_MAE, cni_mae, data);

/* Copy message data from the frame to the message area entry. */
void cni_mae_extract_frame_data(CNI_MAE* mae, FRAME *frame, unsigned
data_length);


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

```
/* "On" values for CA and CO fields in the CNI */
#define CONTROLLER_ON        0xFFFF
#define CONTROLLER_AWAIT    0xFFFF

#define cni_status_area_fields \
unsigned ccf; \
  C_STATE c_state; \
unsigned null_frame[MAX_SRU_COUNT]; \
  CNI_PS protocol_status; \
  CNI_FD frame_diagnosis; \
  CNI_PERS personality; \
unsigned controller_lifesign; \
  CNI_BS assertion_code; \
unsigned cluster_time; \
  CNI_IS interrupt_status; \
  CNI_BS bist_status

#define cni_control_area_fields \
unsigned host_lifesign; \
unsigned external_rate_correction; \
unsigned mode_change_request[3]; \
unsigned ca; /* controller await */ \
  CNI_BE bist_enable; \
unsigned timer1; \
unsigned timer2; \
  CNI_IE interrupt_enable; \
unsigned co; /* controller on */ \
unsigned time_overflow_counter

/* CNI - comunication network interface structure */
typedef struct {
  /* CONTROLLER process that owns this CNI - not physicaly present:
     the (void *) type is used, because otherwise we would have
     to include controller.h and that would confuse the compiler
     function that use this pointer must typecast to (CONTROLLER *) */
void *controller;

  /* CNI - status area */
  cni_status_area_fields;

  /* CNI - control area */
  cni_control_area_fields;

  /* CNI - message area */
char cni_msg_base[CNI_MSG_AREA_SIZE];

} CNI;

/* LOCAL_STATUS - the controller's internal copy of the host-read-only part
of CNI */
typedef struct {
void *controller;
  cni_status_area_fields;
} LOCAL_STATUS;

/* CNI field access functions - get and set */
declare_access_functions(CNI, cni, ccf);
declare_index_access_functions(CNI, cni, null_frame);
```

```
declare_access_functions(CNI, cni, controller_lifesign);
declare_access_functions(CNI, cni, host_lifesign);
declare_sig_access_functions(CNI, cni, external_rate_correction);
declare_index_access_functions(CNI, cni, mode_change_request);
declare_access_functions(CNI, cni, time_overflow_counter);

/* Timer interrupt enable */
typedef unsigned (*FUNC_CNI_IE_SET_TI)(CNI *cni, unsigned value); extern
FUNC_CNI_IE_SET_TI cni_ie_set_ti1; extern FUNC_CNI_IE_SET_TI
cni_ie_set_ti2;

/* Increment and set timer- functions */
get_function_prototype(CNI, cni, timer1);
get_function_prototype(CNI, cni, timer2);
increment_function_prototype(CNI, cni, timer1);
increment_function_prototype(CNI, cni, timer2); typedef unsigned
(*FUNC_CNI_SET_TIMER)(CNI *cni, unsigned value); extern FUNC_CNI_SET_TIMER
cni_set_timer1; extern FUNC_CNI_SET_TIMER cni_set_timer2;


/* The "set" and "increment" functions for cluster_time are the normal
ones. */
set_function_prototype(CNI, cni, cluster_time);
increment_function_prototype(CNI, cni, cluster_time); typedef unsigned
(*FUNC_CNI_GET_CLUSTER_TIME)(CNI *cni); extern FUNC_CNI_GET_CLUSTER_TIME
cni_get_cluster_time;

/* The "get" function for co is the normal one. */
get_function_prototype(CNI, cni, co); typedef unsigned
(*FUNC_CNI_SET_CO)(CNI *cni, unsigned value); extern FUNC_CNI_SET_CO
cni_set_co;

/* The "get" function for ca is the normal one. */
get_function_prototype(CNI, cni, ca); typedef unsigned
(*FUNC_CNI_SET_CA)(CNI *cni, unsigned value); extern FUNC_CNI_SET_CA
cni_set_ca;

/* This function provides easy access to the cni message area in case you
have the offset. */
CNI_MAE *cni_get_message_entry(CNI *cni, unsigned offset);

/* Erase all fields in CNI except the CO field, the CO field is set to
0xFFFF */
void cni_clear_mem(CNI *cni);

/* Erase all status fields in CNI */
void cni_clear_status_fields(CNI *cni);


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

#endif /* #ifndev _FP1CNI_H */
```

4.11. fp1const.h

```
/*
* This software is part of an EU project:
*     FIT - Fault Injection for TTA
*     IST-1999-10748
*
* TTP/C - protocol v.0.1
```

```
*
* Design (c) TTTech & TU Vienna
* Implementation (c) UWB Pilsen & CTU Prague
*
* fp1const.h
* Protocol-defined global constants
*
* version 0.5 - 2001-03-18
*/

#ifndef _FP1CONST_H
#define _FP1CONST_H

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

/* Source MEDL file name definition, %d will be substituted by an index */
#define MEDL_READ_FILE_NAME            "medl%02d.txt"

/* CRC generator polynomials (given in octal) */
#define CRC_POLYNOMIAL_1                0233023
#define CRC_POLYNOMIAL_2            0214461

/* Maximum frame length to use CRC polynomial 1.
  If it is greater than this, polynimial 2 will be used. */
#define MAX_LENGTH_FOR_CRC_POLY_1       151

/* CRC calculation starting value */
#define CRC_STARTING_VALUE              0xFFFF

/* The maximum number of bytes sent in N-Frames. */
#define MAX_APPLICATION_DATA_SIZE       16

/* Size of the stack used by the clock synchronization algorithm. */
#define SYNC_STACK_SIZE                 4

/* TTP/C protocol states */

typedef enum {
  PS_INIT        = 1,
  PS_LISTEN      = 2,
  PS_COLD_START  = 3,
  PS_STARTUP     = 4,
  PS_APPLICATION = 5,
  PS_READY       = 6,
  PS_PASSIVE     = 7,
  PS_AWAIT       = 8,
  PS_SELFTEST    = 9,
  PS_FREEZE      = 10,
  PS_DOWNLOAD    = 12
} PROTOCOL_STATE;

/* Denotes that a message should not be stored in the CNI */
#define CNI_MAE_NO_ADDRESS              0x3FFF

/* Denotes valid C-state in the cni.c_state_valid field. */
#define C_STATE_VALID                   0x0A

/* Denotes valid controller ready value in the cni.controller_ready field.
*/
```

```
#define CONTROLLER_READY                0x0A

/* maximum mode count in MEDL */
#define MAX_MODE_COUNT                  30

/* First application mode number */
#define FIRST_APP_MODE_NUMBER           2

/* Last application mode number */
#define LAST_APP_MODE_NUMBER            30

/* cold start mode number - no real MEDL mode entry */
#define COLD_START_MODE_NUMBER          0

/* startup mode number */
#define STARTUP_MODE_NUMBER             1

/* download mode number - no real MEDL mode entry */
#define DOWNLOAD_MODE_NUMBER            31

/* maximum slot count in one transmission cycle */
#define MAX_TRANSMISSION_CYCLE_LENGTH  512

/* maximum number of SRUs in cluster */
#define MAX_SRU_COUNT                   64

/* Bus guardian clock speed in megahertz. */
#define BUS_GUARDIAN_FREQUENCY          16

/* bus guardian frquency divisor for values in MEDL */
#define BUS_GUARDIAN_DIVISOR            16

/* Duration of 1 macrotick in microseconds */
#define MACROTICK_DURATION              1.0

/* Maximum clock difference */
#define CLOCK_PRECISION                 0.9 * MACROTICK_DURATION

/* Channel transmission speed [bit/s] */
#define CHANNEL_SPEED                   (1 * 1024 * 1024)

/* The host scans medl for sending slots of each controller.
  This is maximum number of stored slots.
  Each one ocupies 4 * MAX_SRU_COUNT * MAX_MODE_COUNT bytes. */
#define MAX_HOST_SLOT_COUNT 4


#endif /* #ifdef _FP1CONST_H */
```

4.12. fp1cstat.cpp

```
/*
* 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
*
* fp1cstat.c
* Functions and definitions concerning the C_STATE structure
*
* version 0.6 - 2001-03-13
*/

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

#include "fp1cstat.h"

#include "fl1func.h"
#include "fp1const.h"

/* Field access functions - set and get */
implement_access_functions(C_STATE, c_state, time, 16)
implement_access_functions(C_STATE, c_state, medl_position, 9)
implement_access_functions(C_STATE, c_state, mode, 5)
implement_access_functions(C_STATE, c_state, dmc, 2)
implement_index_access_functions(C_STATE, c_state, membership, 1)

/* calculate CRC of the C_STATE vector - no inicialization */
void c_state_chain_crc(C_STATE *c_state, unsigned *crc, unsigned
polynomial, unsigned length)
{
int i;

  get_crc(crc, polynomial, c_state_get_time(c_state), 16);
  get_crc(crc, polynomial, c_state_get_medl_position(c_state), 9);
  get_crc(crc, polynomial, c_state_get_mode(c_state), 5);
  get_crc(crc, polynomial, c_state_get_dmc(c_state), 2);
for (i = 0; i < (((int)length - 2) * 16); i++)
    get_crc(crc, polynomial, c_state_get_membership(c_state, i), 1);
}

/* compares two C_STATE structures, returns 1 if equal */
unsigned c_state_cmp(C_STATE *cs1, C_STATE *cs2)
{
int i = 0;
int result =
    (c_state_get_time(cs1) == c_state_get_time(cs2))
    && (c_state_get_medl_position(cs1) == c_state_get_medl_position(cs2))
    && (c_state_get_mode(cs1) == c_state_get_mode(cs2))
    && (c_state_get_dmc(cs1) == c_state_get_dmc(cs2));
while (result && i < MAX_SRU_COUNT) {
    result = result && (c_state_get_membership(cs1, i) ==
c_state_get_membership(cs2, i));
    i++;
  }
return result;
}
```

4.13. fp1cstat.h

```
/*
* This software is part of an EU project:
*     FIT - Fault Injection for TTA
*     IST-1999-10748
```

```
*
* TTP/C - protocol v.0.1
*
* Design (c) TTTech & TU Vienna
* Implementation (c) UWB Pilsen & CTU Prague
*
* fp1cstat.h
* Functions and definitions concerning the C_STATE structure
*
* version 0.7 - 2001-03-16
*/

#ifndef _FP1CSTAT_H
#define _FP1CSTAT_H

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

#include "fl1func.h"
#include "fp1const.h"


/* C-state field */
typedef struct {
unsigned time;
  /* C-state medl */
unsigned medl_position;
unsigned mode;
unsigned dmc;        /* deffered mode change */
  /* Membership vector */
unsigned membership[MAX_SRU_COUNT];
} C_STATE;


/* C_STATE field access functions - get and set */
declare_access_functions(C_STATE, c_state, time);
declare_access_functions(C_STATE, c_state, medl_position);
declare_access_functions(C_STATE, c_state, mode);
declare_access_functions(C_STATE, c_state, dmc);
declare_index_access_functions(C_STATE, c_state, membership);

/* calculate CRC of the C_STATE vector - no inicialization */
void c_state_chain_crc(C_STATE *c_state, unsigned *crc, unsigned
polynomial, unsigned length);

/* compares two C_STATE structures, returns 1 if equal */
unsigned c_state_cmp(C_STATE *cs1, C_STATE *cs2);

#endif /* #ifndef _FP1CSTAT_H */
```

4.14. fp1ctrl.cpp

```
/*
* 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
```

```
*
* sp1ctrl.c
* Functions and definitions concerning the controller c-sim proces
*
* version 0.7 - 2001-05-31
*/

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

#include "fp1ctrl.h"

#include <string.h>
#include <math.h>
#include "fp1const.h"
#include "fp1cni.h"
#include "fp1medl.h"
#include "fp1frame.h"
#include "sp1ctrl.h"

/***********************************************************************
***/
/* Prototypes of functions in this module: */

/* switch the current state of the controller */

void set_state(CONTROLLER *controller, unsigned state);

/* Copies the local CNI status area into the public CNI, updates cluster
time */

void update_cni(CONTROLLER *controller);

/* Updates MEDL position and Cluster time in C-State */

void set_next_slot(CONTROLLER *controller);

/* returns pointer to the controller's current MEDL Slot Control Entry */
MEDL_SCE *get_current_sce(CONTROLLER *controller);

/* returns pointer to the controller's current Mode Address Entry (in
MEDL)*/
MEDL_MAE *get_current_mae(CONTROLLER *controller);

/* Returns whether the current mode is any application mode (mode 2..30) */
unsigned in_application_mode(CONTROLLER *controller);

/* Returns whether the current mode is startup mode (mode 1) */

unsigned in_startup_mode(CONTROLLER *controller);

/* Return true if the node is multiplexed */

unsigned is_multiplexed(CONTROLLER *controller);

/* Check whether current slot position is the controller's sending slot */
unsigned is_my_slot(CONTROLLER *controller);

/* Check whether the controller is sender in current slot */
```

```
unsigned is_current_sender(CONTROLLER *controller);

/* Returns whether bus activity was observed since the last "receive" call
*/

unsigned bus_activity(CONTROLLER *controller);

/* Returns whether cold start is allowed (depends on CF flag and I-Frame
switch counter */

unsigned is_cold_start_allowed(CONTROLLER *controller);

/* Returns whether the controller is allowed to reconfigure in the cluster
  i.e. RA bit is set and there was silence on the bus last round */
unsigned is_reconfiguration_allowed(CONTROLLER *controller);

/* Raise host interrupt, clear local CNI interrupt_status field */

void raise_interrupt(CONTROLLER *controller);

/* Raise TP synchronous interrupt if there is any */

void raise_tp_sync_interrupt(CONTROLLER *controller);

/* Check if MI interrupt is enabled and raise it. */

static void raise_mode_changed_interrupt(CONTROLLER *controller);

/* Check if MC interrupt is enabled and raise it. */

static void raise_membership_changed_interrupt(CONTROLLER *controller);

/* Returns the number of any channel that transmitted a correct I-frame.
  If no correct I-frame was received then 0xFFFF is returned */

unsigned get_valid_i_frame(CONTROLLER *controller);

/* Returns tru if the host lifesign is uptodate and clears the host
lifesign */

unsigned test_host_lifesign(CONTROLLER *controller);

/* Sends specified frame to the appropiate channel.
  Returns 0 if succesfull, nozero otherwise */

int send_frame(CONTROLLER *controller, unsigned channel);

/* Starts receiving on both channels.
  Return 0 if successfull, otherwise nonzero stops receiving. */

int start_receive(CONTROLLER *controller);

/* If receiving then stops receiving on both channels */

void stop_receive(CONTROLLER *controller);

/* Returns Listen Timeout length in macroticks */
```

```
unsigned get_listen_timeout(CONTROLLER *controller);

/* Returns the duration of cold start timeout in macroticks */

unsigned get_cold_start_timeout(CONTROLLER *controller);

/* Returns the MEDL position of controller's first sending slot */

unsigned get_first_sending_slot(CONTROLLER *controller);

/* Initializes C-State in the Cold Start state */

void setup_cold_start_c_state(CONTROLLER *controller);

/* Sets specified frame to be a Cold Start I-Frame */

void setup_cold_start_i_frame(CONTROLLER *controller, unsigned channel);

/* Compares current C-State with the one in received I-Frame (must be
valid):
  - Time and Medl position must match, both SRUs must be in membership
vector
  - Mode field in the I-Frame must contain STARTUP mode
  Returns 0xFFFF if no I-Frame is valid, channel number otherwise */
unsigned cold_start_get_valid_i_frame(CONTROLLER *controller);

/* Sets new BG params if there was no bus activity.
  Returns nonzero code if Bus activity was detected (break) */

int cold_start_set_arm(CONTROLLER *controller);

/* If BG is still in synchonisation mode then set normal mode parameters */
void set_normal_bg_params(CONTROLLER *controller);

/* Sets BG Arm signal if current slot has set the BA field in MEDL */

void set_bg_arm(CONTROLLER *controller);

/* Checks MEDL whether to send an I-frame or N-frame and prepares the frame
in Local Data.
  Returns 0 if the controller is not sending. */

void controller_prepare_frame(CONTROLLER *controller);

/* Prepares an I-frame for the specified channel in the local data */

void prepare_i_frame(CONTROLLER *controller, unsigned channel);

/* Prepares a N-frame for the specified channel in the local data, copies
data from CNI */

void prepare_n_frame(CONTROLLER *controller, unsigned channel);

/* Controller initialization (INIT state):
   - clears CNI, sets Null Frame vect, Personality
   - checks MEDL
   - transits into LISTEN state */

void controller_init_state(CONTROLLER *controller);
```

```
/* Initializes the LISTEN state of the controller.
   - checks controller's state
   - sets BG params for cold start nodes
   - sets wake flags
  Returns zero if all conditions are OK, nonzero otherwise (state
transition follows). */

int controller_prepare_listen_state(CONTROLLER *controller);

/* Finishes the LISTEN state of the controller.
   - stops receiving
   - checks bus for I-Frames
   - synchronizes on valid I-Frame
   - enter Cold Start if enabled and bus is silent */

void controller_finish_listen_state(CONTROLLER *controller);

/* Progress ot the cold start state: computes MEDL position.
  Updates C-State of the controller - MEDL position and Time fields */

void controller_run_cold_start_state(CONTROLLER *controller, unsigned
macroticks);

/* Initializes controller's Cold Start state (1st part):
   - checks controller's state
   - initializes C-State
   - sends cold start frame, increments I-Frame switch
  Returns zero if everything is OK, otherwise nonzero code (state
transition) */

int controller_prepare_cold_start_state(CONTROLLER *controller);

/* Progress ot the cold start state: computes MEDL position.
  Updates C-State of the controller - MEDL position and Time fields */

void controller_run_cold_start_state(CONTROLLER *controller, unsigned
macroticks);

/* Finishes the COLD_START state of the controller.
   - stops receive, clears wake flags
   - updates time fields and MEDL position
   - checks bus for I-Frames - synchronizes on valid I-Frame
  Returns zero if valid I-Frame was received - synchronize, nonzero
otherwise */

int controller_finish_cold_start_state(CONTROLLER *controller);

/* Controller's Pre Send Phase.
  When a frame should be sent (controller's sending slot), then:
   - Checks frame counters (clique avoidance, blackout) and clears them
   - Prepares frames to be sent in controller's local data
   - Returns nonzero code
  Otherwise zero is returned.
  Function may switch state. */

unsigned psp(CONTROLLER *controller);

/* Perform Implicit acknowledgement algorithm check on specified channel.
  Sets state to the new IACK state but does not change controller's local
```

```
data.
  If the new IACK state could not be determined, then *valid is cleared. */
void check_iack(CONTROLLER *controller, unsigned channel, IACK_STATES
*state, unsigned *valid);

/* Performs implicit acknoledgement algorithm on received frame.
  Sets IACK state in controller's local_data and updates membership vector.
  May switch controller's state.
  Returns zero if the IACK algorithm decision is final, 1 otherwise. */
unsigned perform_iack(CONTROLLER *controller);

/* performs clock-synchronization algorithm if enabled in MEDL */

void perform_clock_synchronization(CONTROLLER *controller);

/* Controller's Post-Receive-Phase.
  Stops current receive operation if stil running.
  Checks received frames, runs Implicit ack. algorithm.
  Updates membership and null frame vectors.
  Data read from N-Frames are stored into CNI. */

void prp(CONTROLLER *controller);

/* Initializes a Controller data structure.
  Returns a pointer to the initialized structure. */
CONTROLLER *controller_init(CONTROLLER *controller);

/* Stores hosts address and interrupt function. */

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

/* Attaches the controller to the specified channel. */

void controller_attach_channel(CONTROLLER *controller, CHANNEL *channel,
unsigned index);

/* Sets the controllers local clock drift in units of [sec/sec] */

void controller_set_clock_drift(CONTROLLER *controller, double drift);

/* Called by the channel after controller's frame transmission is
completed. */

void controller_send_callback(void *controller);

/* Called by the channel after a frame is received. */

void controller_receive_callback(void *controller, unsigned channel);


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

/* Check if MI interrupt is enabled and raise it. */

static void raise_mode_changed_interrupt(CONTROLLER *controller)
{
if (cni_ie_get_mi(&controller->cni.interrupt_enable) == 1) {
    cni_is_set_mi(&controller->lcni->interrupt_status, 1);
    raise_interrupt(controller);
  }
}

/* Check if MC interrupt is enabled and raise it. */

static void raise_membership_changed_interrupt(CONTROLLER *controller)
{
if (cni_ie_get_mc(&controller->cni.interrupt_enable) == 1) {
    cni_is_set_mc(&controller->lcni->interrupt_status, 1);
    raise_interrupt(controller);
  }
}


/* switch the current state of the controller */

void set_state(CONTROLLER *controller, unsigned state)
{
  cni_ps_set_protocol_state(&controller->lcni->protocol_status, state);
if (state == PS_FREEZE)
    cni_set_co(&controller->cni, 0);
}

/* Copies the local CNI status area into the public CNI, updates cluster
time */

void update_cni(CONTROLLER *controller)
{
if (controller->cni.cluster_time != controller->lcni->cluster_time)
    controller->update_cluster_time(controller);
  memcpy(&controller->cni, controller->lcni, sizeof(LOCAL_STATUS));
}

/* Updates MEDL position and Cluster time in C-State */

void set_next_slot(CONTROLLER *controller)
{
  /* Increment C-State time */
  c_state_set_time(&controller->lcni->c_state, bit_mask(16) & (
    c_state_get_time(&controller->lcni->c_state) +
medl_sce_get_sru_slot_duration(get_current_sce(controller))));

  /* Increment medl_position and test End Of Transmission flag: */
if (medl_sce_get_eot(get_current_sce(controller)) == 0)
    c_state_increment_medl_position(&controller->lcni->c_state, 1);
else
    c_state_set_medl_position(&controller->lcni->c_state, 0);

  /* FIXME: raise interrupt (maybe slot change) */
}

/* returns pointer to the controller's current MEDL Slot Control Entry */
MEDL_SCE *get_current_sce(CONTROLLER *controller)
{
return &controller-
>medl.slot_control_entry[c_state_get_medl_position(&controller->lcni-
>c_state)];
}
```

```
/* returns pointer to the controller's current Mode Address Entry (in
MEDL)*/
MEDL_MAE *get_current_mae(CONTROLLER *controller)
{
return &controller->medl.mode_address_entry
    [c_state_get_mode(&controller->lcni->c_state)]
    [c_state_get_medl_position(&controller->lcni->c_state)];
}

/* Returns whether the current mode is any application mode (mode 2..30) */
unsigned in_application_mode(CONTROLLER *controller)
{
return (c_state_get_mode(&controller->lcni->c_state) >=
FIRST_APP_MODE_NUMBER
    && c_state_get_mode(&controller->lcni->c_state) <=
LAST_APP_MODE_NUMBER);
}

/* Returns whether the current mode is startup mode (mode 1) */

unsigned in_startup_mode(CONTROLLER *controller)
{
return (c_state_get_mode(&controller->lcni->c_state) ==
STARTUP_MODE_NUMBER);
}

/* Return true if the node is multiplexed */

unsigned is_multiplexed(CONTROLLER *controller)
{
return (logical_name_get_multiplex_id(&controller-
>cni.personality.logical_name) > 0);
}

/* Check whether current slot position is the controller's sending slot */
unsigned is_my_slot(CONTROLLER *controller)
{
return (logical_name_get_slot_position(&controller->lcni-
>personality.logical_name)
    == logical_name_get_slot_position(&get_current_sce(controller)-
>logical_name));
}

/* Check whether the controller is sender in current slot */

unsigned is_current_sender(CONTROLLER *controller)
{
return logical_name_cmp(&controller->lcni->personality.logical_name,
&get_current_sce(controller)->logical_name);
}

/* Returns whether bus activity was observed since the last "receive" call
*/

unsigned bus_activity(CONTROLLER *controller)
{
return controller->ldata.frame[0].state != FRAME_STATE_NULL
    || controller->ldata.frame[1].state != FRAME_STATE_NULL;
}
```

```
/* Returns whether cold start is allowed (depends on CF flag and I-Frame
switch counter */

unsigned is_cold_start_allowed(CONTROLLER *controller)
{
return (medl_ccp_get_cf(&controller->medl.controller_cfg_params) == 1)
    && ((cni_ps_get_i_frame_switch(&controller->lcni->protocol_status)
      < medl_ccp_get_max_cold_start_entry(&controller-
>medl.controller_cfg_params))
      || medl_ccp_get_max_cold_start_entry(&controller-
>medl.controller_cfg_params) == 0);
}

/* Returns whether the controller is allowed to reconfigure in the cluster
  i.e. RA bit is set and there was silence on the bus last round */
unsigned is_reconfiguration_allowed(CONTROLLER *controller)
{
if (medl_sce_get_ra(get_current_sce(controller)) != 1)
    return 0;
if (cni_get_null_frame(controller->lcni,
medl_sce_get_membership_pos(get_current_sce(controller))) != 1)
    return 0;
if (c_state_get_membership(&controller->cni.c_state,
medl_sce_get_membership_pos(get_current_sce(controller))) == 1) {
    print_debug((((CONTROLLER_MODEL *)controller)->process,
"Reconfiguration disabled: Slot ocupied error\n"));
    cni_ps_set_so(&controller->cni.protocol_status, 1);
    cni_is_set_he(&controller->cni.interrupt_status, 1);
    if (cni_ie_get_he(&controller->cni.interrupt_enable) == 1)
      raise_interrupt(controller);
    set_state(controller, PS_PASSIVE);
    return 0;
  }
  print_debug((((CONTROLLER_MODEL *)controller)->process, "Reconfiguration
Enabled\n"));
return 1;
}

/* Raise host interrupt, clear local CNI interrupt_status field */

void raise_interrupt(CONTROLLER *controller)
{
  /* FIXME: We may need to test interrupt_enable field here */
if (controller->ldata.host_interrupt_callback != NULL) {
    memcpy(&controller->cni.interrupt_status, &controller->lcni-
>interrupt_status, sizeof(CNI_IS));
    memset(&controller->lcni->interrupt_status, 0, sizeof(CNI_IS));
    controller->ldata.host_interrupt_callback(controller->ldata.host);
  }
}

/* Raise TP synchronous interrupt if there is any */

void raise_tp_sync_interrupt(CONTROLLER *controller)
{
if (controller->ldata.tp_sync_interrupt) {
    raise_interrupt(controller);
    controller->ldata.tp_sync_interrupt = 0;
  }
}


/* Returns the number of any channel that transmitted a correct I-frame.
  If no correct I-frame was received then 0xFFFF is returned */

unsigned get_valid_i_frame(CONTROLLER *controller)
{
if (frame_is_valid_i_frame(&controller->ldata.frame[0]))
    return 0;
if (frame_is_valid_i_frame(&controller->ldata.frame[1]))
    return 1;

return 0xFFFF;
}

/* Returns tru if the host lifesign is uptodate and clears the host
lifesign */

unsigned test_host_lifesign(CONTROLLER *controller)
{
unsigned result;

  print_debug((((CONTROLLER_MODEL *)controller)->process, "Updating
controller lifesign\n"));
  result = ((~cni_get_host_lifesign(&controller->cni) & bit_mask(16)) ==
cni_get_controller_lifesign(controller->lcni));
  cni_set_host_lifesign(&controller->cni, 0);
  cni_set_controller_lifesign(controller->lcni,
c_state_get_time(&controller->lcni->c_state) & bit_mask(15));

if (!result)
    print_debug((((CONTROLLER_MODEL *)controller)->process, "Host not
alive\n"));

return result;
}

/* Synchronizes the controller from the data in an I-Frame received on
specified channel.
  Sets C-State and Valid Flag, changes controller's state.*/

static void synchronize_on_i_frame(CONTROLLER *controller, unsigned
channel)
{
  CNI *lcni = controller->lcni;
  LOCAL_DATA *ldata = &controller->ldata;
  I_FRAME *i_frame = &ldata->frame[channel].frame.i;

  /* Set Synchronization Params for BG */
  bus_guardian_set_params((BUS_GUARDIAN *)controller->bus_guardian,
    &controller->medl.implementation_params.synchronization_period_params);
  ldata->immediate_send_permission = 1;

  cni_ps_increment_accept_counter(&lcni->protocol_status, 1);

if (c_state_get_mode(&i_frame->c_state) == COLD_START_MODE_NUMBER &&
is_multiplexed(controller))
```

```
    return;         /* multiplexed node: discard frame */

  /* Copy C-State */
  lcni->c_state = i_frame->c_state;
if (c_state_get_mode(&i_frame->c_state) == COLD_START_MODE_NUMBER)
    /* Cold Start I-Frame: Modify Mode Number */
    c_state_set_mode(&lcni->c_state, STARTUP_MODE_NUMBER);

  cni_ps_set_c_state_valid(&lcni->protocol_status, C_STATE_VALID);
  cni_set_null_frame(lcni,
medl_sce_get_membership_pos(get_current_sce(controller)), 0);

  /* Set Cluster Time */
  cni_set_cluster_time(lcni, c_state_get_time(&lcni->c_state));
  ldata->last_clock_access = ldata->frame[channel].time_of_delivery;
  ldata->last_mt_fract = 0;
  cni_ps_set_csct(&lcni->protocol_status, 0);

/*    if (frame_header_get_imc(&ldata->frame[channel].frame.i.header) != 0)
{*/
    /* TODO: interrupt mode change */
    /* TODO: Why is this not done in Cold start Allowed nodes? */
/*        c_state_set_mode(&lcni->c_state, frame_header_get_imc(&ldata-
>frame[channel].frame.i.header));
  }*/

  update_cni(controller);
  set_next_slot(controller);

if (in_application_mode(controller)) {
    /* TODO: The App. mode is skipped if Cold Start is allowed. Why? */
    test_host_lifesign(controller);
    start_receive(controller);
    set_state(controller, PS_PASSIVE);
  }
else if (in_startup_mode(controller)) {
    start_receive(controller);
    set_state(controller, PS_READY);
  }
else if (test_host_lifesign(controller)) {
    start_receive(controller);
    set_state(controller, PS_READY);
  }
else {
    start_receive(controller);
    set_state(controller, PS_PASSIVE);
  }

  /* Interrupts */
  raise_mode_changed_interrupt(controller);
  raise_membership_changed_interrupt(controller);
}

/* Sends specified frame to the appropiate channel.
 Returns 0 if succesfull, nozero otherwise */

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

```
int result;

  /* FIXME: Warning: Is it OK to set memberhip here? */
  c_state_set_membership(&controller->lcni->c_state,
get_current_sce(controller)->membership_pos, 1);

  controller->ldata.mtd_a = 0;
  controller->ldata.mtd_b = 0;

  result = controller->channel[channel]->channel_send_frame(
    controller->channel[channel], (void *)controller, controller-
>send_callback);

if (result != 0)
    controller->bus_guardian->arm_signal(controller->bus_guardian, 0);

return result;
}

/* Starts receiving on both channels.
  Return 0 if successfull, otherwise nonzero stops receiving. */

int start_receive(CONTROLLER *controller)
{
int result;

  result = channel_receive((CHANNEL *)controller->channel[0], (void
*)controller, controller->receive_callback)
    | channel_receive((CHANNEL *)controller->channel[1], (void
*)controller, controller->receive_callback);

if (result)
    stop_receive(controller);

return result;
}

/* If receiving then stops receiving on both channels */

void stop_receive(CONTROLLER *controller)
{
  channel_cancel_receive((CHANNEL *)controller->channel[0], (void
*)controller);
  channel_cancel_receive((CHANNEL *)controller->channel[1], (void
*)controller);
}

/* Returns Listen Timeout length in macroticks */

unsigned get_listen_timeout(CONTROLLER *controller)
{
return medl_ccp_get_startup_timeout(&controller-
>medl.controller_cfg_params)
    + 2 * medl_ccp_get_tdma_round_length(&controller-
>medl.controller_cfg_params);
}

/* Returns the duration of cold start timeout in macroticks */
```

```
unsigned get_cold_start_timeout(CONTROLLER *controller)
{
return medl_ccp_get_tdma_round_length(&controller-
>medl.controller_cfg_params) +
      medl_ccp_get_startup_timeout(&controller-
>medl.controller_cfg_params);
}

/* Returns the MEDL position of controller's first sending slot */

unsigned get_first_sending_slot(CONTROLLER *controller)
{
unsigned i = 0;

while ((i < MAX_TRANSMISSION_CYCLE_LENGTH) &&
    (logical_name_cmp(&controller->medl.slot_control_entry[i].logical_name,
&controller->medl.role_list.logical_name) == 0))
    i++;
  assert(i < MAX_TRANSMISSION_CYCLE_LENGTH); /* Bad MEDL design */
return i;
}

/* Initializes C-State in the Cold Start state */

void setup_cold_start_c_state(CONTROLLER *controller)
{
  c_state_set_mode(&controller->lcni->c_state, COLD_START_MODE_NUMBER);
  c_state_set_medl_position(&controller->lcni->c_state,
get_first_sending_slot(controller));

  /* set Membership vector */
  memset(&controller->lcni->c_state.membership, 0, sizeof(controller->lcni-
>c_state.membership));
  c_state_set_membership(&controller->lcni->c_state, controller-
>ldata.membership_pos, 1);

  c_state_set_dmc(&controller->lcni->c_state, 0);
  c_state_set_time(&controller->lcni->c_state,
cni_get_cluster_time(controller->lcni));
}

/* Sets specified frame to be a Cold Start I-Frame */

void setup_cold_start_i_frame(CONTROLLER *controller, unsigned channel)
{
  /* set header */
  controller->ldata.frame[channel].type = FRAME_TYPE_I;
  controller->ldata.frame[channel].state = FRAME_STATE_NULL;
  frame_header_set_i_n_frame(&controller-
>ldata.frame[channel].frame.i.header, 1);
  frame_header_set_dmc(&controller->ldata.frame[channel].frame.i.header,
0);
  /* set C-State */
  controller->ldata.frame[channel].frame.i.c_state = controller->lcni-
>c_state;
  controller->ldata.frame[channel].frame.i.c_state_length =
medl_ccp_get_c_state_length(&controller->medl.controller_cfg_params);
  /* set CRC */
  i_frame_set_crc(&controller->ldata.frame[channel].frame.i,
i_frame_calculate_crc(&controller->ldata.frame[channel].frame.i));
}

/* Compares current C-State with the one in received I-Frame (must be
valid):
  - Time and Medl position must match, both SRUs must be in membership
vector
  - Mode field in the I-Frame must contain STARTUP mode
  Returns 0xFFFF if no I-Frame is valid, channel number otherwise */
unsigned cold_start_get_valid_i_frame(CONTROLLER *controller)
{
unsigned mbp1, mbp2;
unsigned medl_position;
unsigned c;
  I_FRAME *i_frame;

if ((c = get_valid_i_frame(controller)) > 1)
    return 0xFFFF;
  i_frame = &controller->ldata.frame[c].frame.i;

  mbp1 = controller->ldata.membership_pos;
  mbp2 = medl_sce_get_membership_pos(get_current_sce(controller));
  medl_position = c_state_get_medl_position(&i_frame->c_state);

if ( (c_state_get_mode(&i_frame->c_state) == STARTUP_MODE_NUMBER)
    && (c_state_get_time(&i_frame->c_state) ==
c_state_get_time(&controller->lcni->c_state))
    && (medl_position == c_state_get_medl_position(&controller->lcni-
>c_state))
    && (c_state_get_membership(&i_frame->c_state, mbp1) == 1)
    && (c_state_get_membership(&i_frame->c_state, mbp2) == 1) )
    return c;
else
    return 0xFFFF;
}

/* Sets new BG params if there was no bus activity.
  Returns nonzero code if Bus activity was detected (break) */

int cold_start_set_arm(CONTROLLER *controller)
{
if (bus_activity(controller)) {
    update_cni(controller);
    return 1;
  }

  bus_guardian_set_params((BUS_GUARDIAN *)controller->bus_guardian,
&controller->medl.bus_guardian_cold_start);
  controller->bus_guardian->arm_signal(controller->bus_guardian, 1);

  update_cni(controller);
return 0;
}

/* If BG is still in synchonisation mode then set normal mode parameters */
void set_normal_bg_params(CONTROLLER *controller)
{
if (controller->ldata.immediate_send_permission) {
    bus_guardian_set_params((BUS_GUARDIAN *)controller->bus_guardian,
&controller->medl.role_list.bus_guardian);
    controller->ldata.immediate_send_permission = 0;
   }
}

/* Sets BG Arm signal if current slot has set the BA field in MEDL */

void set_bg_arm(CONTROLLER *controller)
{
if (medl_sce_get_ba(get_current_sce(controller))) {
    set_normal_bg_params(controller);
    controller->bus_guardian->arm_signal(controller->bus_guardian, 1);
   }
}

/* Checks MEDL whether to send an I-frame or N-frame and prepares the frame
in Local Data.
  Returns 0 if the controller is not sending. */

void controller_prepare_frame(CONTROLLER *controller)
{
unsigned membership_pos = controller->ldata.membership_pos;
  /* Store current Membership Vector status */
unsigned membership = c_state_get_membership(&controller->lcni->c_state,
membership_pos);

  c_state_set_membership(&controller->lcni->c_state, membership_pos, 1);

if (medl_mae_get_ia(get_current_mae(controller)) == 1)
    prepare_i_frame(controller, 0);
else
    prepare_n_frame(controller, 0);
  controller->ldata.frame[0].state = FRAME_STATE_COMPLETE;

if (medl_mae_get_ib(get_current_mae(controller)) == 1)
    prepare_i_frame(controller, 1);
else
    prepare_n_frame(controller, 1);
  controller->ldata.frame[1].state = FRAME_STATE_COMPLETE;

  /* Restore origin Membership Vector status */
  c_state_set_membership(&controller->lcni->c_state, membership_pos,
membership);
}

/* Prepares an I-frame for the specified channel in the local data */

void prepare_i_frame(CONTROLLER *controller, unsigned channel)
{
  I_FRAME *frame = &controller->ldata.frame[channel].frame.i;
unsigned i;

  controller->ldata.frame[channel].type = FRAME_TYPE_I;
  frame_header_set_i_n_frame(&frame->header, 1);
for (i = 0; i < 3; i++)
    frame_header_set_mode_change(&frame->header, i,
cni_get_mode_change_request(&controller->cni, i));

  frame->c_state = controller->lcni->c_state;
  frame->c_state_length = medl_ccp_get_c_state_length(&controller-
>medl.controller_cfg_params);
  i_frame_set_crc(frame, i_frame_calculate_crc(frame));
}

/* Prepares a N-frame for the specified channel in the local data, copies
data from CNI */

void prepare_n_frame(CONTROLLER *controller, unsigned channel)
{
  N_FRAME *frame = &controller->ldata.frame[channel].frame.n;
  CNI_MAE *cni_mae;
unsigned data_len;
unsigned i;

if (channel == 0) {
    cni_mae = cni_get_message_entry(&controller->cni,
medl_mae_get_cni_address_cha(get_current_mae(controller)));
    data_len = medl_mae_get_dafla(get_current_mae(controller)) + 1;
  }
else {
    cni_mae = cni_get_message_entry(&controller->cni,
medl_mae_get_cni_address_chb(get_current_mae(controller)));
    data_len = medl_mae_get_daflb(get_current_mae(controller)) + 1;
  }

if (cni_mae_get_eif(cni_mae) != EIF_CORRECT_FRAME) {
    /* FIXME: raise interrupt (host error) */
    cni_is_set_he(&controller->lcni->interrupt_status, 1);
    set_state(controller, PS_PASSIVE);
  }

  controller->ldata.frame[channel].type = FRAME_TYPE_N;
  frame_header_set_i_n_frame(&frame->header, 0);
for (i = 0; i < 3; i++)
    frame_header_set_mode_change(&frame->header, i,
cni_get_mode_change_request(&controller->cni, i));

  frame->application_data_length = data_len;
for (i = 0; i < data_len; i++)
    n_frame_set_application_data(frame, i, cni_mae_get_data(cni_mae, i));

  n_frame_set_crc(frame, n_frame_calculate_crc(frame, &controller->lcni-
>c_state,
    medl_ccp_get_c_state_length(&controller->medl.controller_cfg_params)));
}

/* Controller initialization (INIT state):
  - clears CNI, sets Null Frame vect, Personality
  - checks MEDL
  - transits into LISTEN state */

void controller_init_state(CONTROLLER *controller)
{
unsigned i;

  /* clear DPRAM */
  cni_clear_mem(&controller->cni);
  cni_clear_status_fields(controller->lcni);
for (i = 0; i < MAX_SRU_COUNT; i++)
    controller->lcni->null_frame[i] = 1;
  local_data_initialize(&controller->ldata);

  /* Set Personality Field */
  controller->lcni->personality.logical_name = controller-
>medl.role_list.logical_name;
if (medl_validity_check(&controller->medl) == 0) {
    /* Error: Invalid MEDL */
    cni_ps_set_mc(&controller->lcni->protocol_status, 1);
    set_state(controller, PS_FREEZE);
    return;
  }

  /* store controller's membership position */
  controller->ldata.membership_pos = medl_sce_get_membership_pos(
    &controller-
>medl.slot_control_entry[get_first_sending_slot(controller)]);

  cni_ps_set_controller_ready(&controller->lcni->protocol_status,
CONTROLLER_READY);
  set_state(controller, PS_LISTEN);

  bus_guardian_initialize(controller->bus_guardian);
}

/* Initializes the LISTEN state of the controller.
  - checks controller's state
  - sets BG params for cold start nodes
 Returns zero if all conditions are OK, nonzero otherwise (state
transition follows). */

int controller_prepare_listen_state(CONTROLLER *controller)
{
  /* test CO field: enter FREEZE state */
if (cni_get_co(&controller->cni) != CONTROLLER_ON) {
    set_state(controller, PS_FREEZE);
    return 1;
  }
  /* test CA field: enter AWAIT state */
if (cni_get_ca(&controller->cni) == CONTROLLER_AWAIT) {
    set_state(controller, PS_AWAIT);
    return 2;
  }
  /* test BIST enable field: enter SELFTEST state */
if (cni_be_get_rm(&controller->cni.bist_enable) != 0 ||
cni_be_get_rf(&controller->cni.bist_enable) != 0) {
    set_state(controller, PS_SELFTEST);
    return 3;
  }

  /* Test whether this is a cold-start-allowed node. */
if (is_cold_start_allowed(controller)) {
    /* Set BG Cold Start params */
    bus_guardian_set_params(controller->bus_guardian, &controller-
>medl.implementation_params.listen_cold_start_period_params);
    controller->bus_guardian->arm_signal(controller->bus_guardian, 1);
  }

return 0;
}

/* Finishes the LISTEN state of the controller.
  - stops receiving
  - checks bus for I-Frames
  - synchronizes on valid I-Frame
  - enter Cold Start if enabled and bus is silent */

void controller_finish_listen_state(CONTROLLER *controller)
{
unsigned channel;

  stop_receive(controller);

if (cni_get_co(&controller->cni) != CONTROLLER_ON) {
    set_state(controller, PS_FREEZE);
    return;
  }

if (bus_activity(controller)) {
    /* bus active: check for a valid I-frame on both channels */
    channel = get_valid_i_frame(controller);
    if (channel == 0 || channel == 1) {
      print_debug((((CONTROLLER_MODEL *)controller)->process, "Got valid I-
Frame on channel %d.\n", channel));
      synchronize_on_i_frame(controller, channel);
      return;        /* Synchronized (if not multiplexed) */
    }
  } else
    if (is_cold_start_allowed(controller) &&
test_host_lifesign(controller)) {
      set_state(controller, PS_COLD_START);
      return;        /* Enter Cold Start mode */
    }

  print_debug((((CONTROLLER_MODEL *)controller)->process, "No valid I-Frame
Received\n"));
}

/* Initializes controller's Cold Start state (1st part):
  - checks controller's state
  - initializes C-State, stores CSim time
  - sends cold start frame, increments I-Frame switch
  Returns zero if everything is OK, otherwise nonzero code (state
transition) */

int controller_prepare_cold_start_state(CONTROLLER *controller)
{
  CNI *lcni = controller->lcni;
unsigned channel;

  /* test CO field: enter FREEZE state */
if (cni_get_co(&controller->cni) != CONTROLLER_ON) {
    set_state(controller, PS_FREEZE);
    return 1;
  }
  /* Determine which channel to use - LSB in I-frame switch. */
  channel = cni_ps_get_i_frame_switch(&lcni->protocol_status) & 1;
  /* Prepare C-State. */
  setup_cold_start_c_state(controller);
  setup_cold_start_i_frame(controller, channel);

  /* Send I-Frame on appropriate channel. */
  send_frame(controller, channel);
  cni_ps_increment_i_frame_switch(&lcni->protocol_status, 1);

  /* Wait for next slot and setup BG */
  update_cni(controller);

  print_debug((((CONTROLLER_MODEL *)controller)->process, "Starting cold
start timeout.\n"));
return 0;
}

/* Progress in the cold start state during coldstart timeout.
  Updates C-State of the controller - MEDL position and Time fields */

void controller_run_cold_start_state(CONTROLLER *controller, unsigned
macroticks)
{
unsigned i;
unsigned duration;

  i = c_state_get_medl_position(&controller->lcni->c_state);
while (macroticks >= (duration =
medl_sce_get_sru_slot_duration(&controller->medl.slot_control_entry[i]))) {
    set_next_slot(controller);
    macroticks -= duration;
    /* Advance through medl slot control entries */
    if (medl_sce_get_eot(&controller->medl.slot_control_entry[i]) == 1)
      i += 1;
    else
      i = 0;
  }
}

/* Finishes the COLD_START state of the controller.
  - stops receive, clears wake flags
  - updates time fields and MEDL position
  - checks bus for I-Frames - synchronizes on valid I-Frame
 Returns zero if valid I-Frame was received - synchronize, nonzero
otherwise */

int controller_finish_cold_start_state(CONTROLLER *controller)
{
unsigned channel;
  CNI *lcni = controller->lcni;

  stop_receive(controller);

if (cni_get_co(&controller->cni) != CONTROLLER_ON) {
    set_state(controller, PS_FREEZE);
    return 1;
  }

if (bus_activity(controller)) {
    channel = cold_start_get_valid_i_frame(controller);
    if (channel == 0 || channel == 1) {
      /* received correct I-Frame */
      cni_ps_set_i_frame_switch(&lcni->protocol_status, 0);
      lcni->c_state = controller->ldata.frame[channel].frame.i.c_state;
      set_next_slot(controller);
      set_state(controller, PS_STARTUP);
      /* Interrupts */
      raise_mode_changed_interrupt(controller);
      raise_membership_changed_interrupt(controller);
      return 0;    /* synchronized */
    }
    else {
      /* no valid I-Frame received */
      print_debug((((CONTROLLER_MODEL *)controller)->process, "No valid I-
Frame received\n"));
      set_state(controller, PS_LISTEN); /* back to listen */
    }
  }
else {
    /* no bus activity detected */
    print_debug((((CONTROLLER_MODEL *)controller)->process, "No bus
activity detected\n"));
    if (!is_cold_start_allowed(controller))
      set_state(controller, PS_LISTEN);
  }
return 2;
}

/* Controller's Pre Send Phase.
  When a frame should be sent (controller's sending slot), then:
   - Checks frame counters (clique avoidance, blackout) and clears them
   - Prepares frames to be sent in controller's local data
   - Returns nonzero code
  Otherwise zero is returned.
  Function may switch state. */

unsigned psp(CONTROLLER *controller)
{
  CNI *lcni = controller->lcni;

if (is_my_slot(controller)) {
    /* current slot is my sending slot in any TDMA round (may not be
current)*/
    if ((cni_ps_get_failed_counter(&lcni->protocol_status) +
cni_ps_get_invalid_counter(&lcni->protocol_status))
      >= cni_ps_get_accept_counter(&lcni->protocol_status)) {
      /* acknowledgement error */
      cni_ps_set_ae(&lcni->protocol_status, 1);
      set_state(controller, PS_FREEZE);
      return 0;
    }
    if ((cni_ps_get_failed_counter(&lcni->protocol_status) +
cni_ps_get_invalid_counter(&lcni->protocol_status)
      + cni_ps_get_accept_counter(&lcni->protocol_status)) == 0) {
```

```
      /* communication system blackout */
      cni_ps_set_cb(&lcni->protocol_status, 1);
      set_state(controller, PS_FREEZE);
      return 0;
    }
    /* reset frame counters */
    cni_ps_set_failed_counter(&lcni->protocol_status, 0);
    cni_ps_set_invalid_counter(&lcni->protocol_status, 0);
    cni_ps_set_accept_counter(&lcni->protocol_status, 0);
  }

if (is_current_sender(controller)) {
    /* current slot is definitely my sending slot */
    if (!test_host_lifesign(controller)) {
      set_state(controller, PS_PASSIVE);
      return 0;
    } else
      if (cni_ps_get_protocol_state(&lcni->protocol_status) == PS_PASSIVE)
        set_state(controller, PS_READY);

    if (cni_ps_get_protocol_state(&lcni->protocol_status) == PS_READY &&
!is_reconfiguration_allowed(controller))
      return 0;

    controller_prepare_frame(controller);
    controller->ldata.iack_state = IACK_SEND;

    /* FIXME: maybe also for PASSIVE state */
    if (cni_ps_get_protocol_state(&lcni->protocol_status) == PS_READY) {
      if (in_startup_mode(controller))
        set_state(controller, PS_STARTUP);
      else
        set_state(controller, PS_APPLICATION);
    }

    return 1;
  }
return 0;
}

/* Perform Implicit acknowledgement algorithm check on specified channel.
  Sets state to the new IACK state but does not change controller's local
data.
  If the new IACK state could not be determined, then *valid is cleared. */
void check_iack(CONTROLLER *controller, unsigned channel, IACK_STATES
*state, unsigned *valid)
{
  C_STATE c_state;
  LOCAL_DATA *ldata = &controller->ldata;

  *valid = 1;
switch (ldata->iack_state) {
    /* Assumming that I am node A, node B is my successor and node C is B's
successor. */
    case IACK_SEND:
      /* Send phase */
      *state = IACK_CHECK1;
      break;
    case IACK_OK:
      /* Nothing to check. */
      *state = IACK_OK;
      break;
    case IACK_CHECK1:
      /* Did node B say than I was OK? */
      if (frame_is_valid(&ldata->frame[channel], &controller->lcni-
>c_state,
        medl_ccp_get_c_state_length(&controller-
>medl.controller_cfg_params))) {
        *state = IACK_OK;
        break;
      }
      /* Did node B say that I was not OK? */
      c_state = controller->lcni->c_state;
      c_state_set_membership(&c_state, ldata->membership_pos, 0);
      if (frame_is_valid(&ldata->frame[channel], &c_state,
        medl_ccp_get_c_state_length(&controller-
>medl.controller_cfg_params))) {
        *state = IACK_CHECK2;
        break;
      }
      /* Neither is the case, we must wait for another node B (successor).
*/
      *state = IACK_CHECK1;
      *valid = 0;
      break;
    case IACK_CHECK2:
      /* Did node C say that I was OK and B had failed? */
      c_state = controller->lcni->c_state;
      c_state_set_membership(&c_state, ldata->successor, 0);
      if (frame_is_valid(&ldata->frame[channel], &c_state,
        medl_ccp_get_c_state_length(&controller-
>medl.controller_cfg_params))) {
        *state = IACK_OK;
        break;
      }
      /* Did node C say that I had failed and B was OK? */
      c_state = controller->lcni->c_state;
      c_state_set_membership(&c_state, ldata->membership_pos, 0);
      c_state_set_membership(&c_state, ldata->successor, 1);
      if (frame_is_valid(&ldata->frame[channel], &c_state,
        medl_ccp_get_c_state_length(&controller-
>medl.controller_cfg_params))) {
        *state = IACK_ERROR;
        break;
      }
      /* Neither is the case, double failure occured. */
      *state = IACK_DOUBLE_FAILURE;
      *valid = 0;
      break;
    case IACK_ERROR:
    case IACK_DOUBLE_FAILURE:
      break;
  }
}
```

```
/* Performs implicit acknoledgement algorithm on received frame.
  Sets IACK state in controller's local_data and updates membership vector.
  May switch controller's state.
  Returns zero if the IACK algorithm decision is final, 1 otherwise. */
unsigned perform_iack(CONTROLLER *controller)
{
  IACK_STATES iack_a, iack_b;
unsigned valid_a, valid_b;
  CNI *lcni = controller->lcni;
  LOCAL_DATA *ldata = &controller->ldata;

  print_debug((((CONTROLLER_MODEL *)controller)->process, "Performing IAck:
"));

if (((cni_ps_get_protocol_state(&lcni->protocol_status)) == PS_READY)
      || ((cni_ps_get_protocol_state(&lcni->protocol_status)) ==
PS_PASSIVE)
      || ((cni_ps_get_protocol_state(&lcni->protocol_status)) ==
PS_COLD_START))    {
    /* In READY, PASSIVE, COLD START states no checking is done */
    iack_a = IACK_OK;
  }
else {
    /* get IACK results for both channels */
    check_iack(controller, 0, &iack_a, &valid_a);
    check_iack(controller, 1, &iack_b, &valid_b);

    /* if one result is invalid, then replace it by the other */
    if (!valid_a && valid_b)
      iack_a = iack_b;
    else
      if (!valid_b && valid_a)
        iack_b = iack_a;

    if (iack_a != iack_b) {
      /* iack results differ -> transit into FREEZE */
      cni_bs_set_av(&lcni->bist_status, 1);
      set_state(controller, PS_FREEZE);
      /* TODO: interrupt (BIST error) */
      return 1;
    }
  }

  print_debug_no_time(("%s -> %s\n", get_iack_state_name(ldata-
>iack_state), get_iack_state_name(iack_a)));

  /* switch according to previous IACK state */
  /* assume:    A is sender waiting for acknoldgement,
      B is working successor of A
      C is direct succesor of B */
switch (ldata->iack_state) {
    case IACK_OK:
      return 0;
    case IACK_SEND:
      break;
    case IACK_CHECK1:
      if (iack_a == IACK_OK)
```

```
        /* B sends correct ack. */
        cni_ps_increment_accept_counter(&lcni->protocol_status, 1);
      if (iack_a == IACK_CHECK1 && bus_activity(controller)) {
        /* B failed */
        cni_ps_increment_failed_counter(&lcni->protocol_status, 1);
        c_state_set_membership(&lcni->c_state,
medl_sce_get_membership_pos(get_current_sce(controller)), 0);
      }
      if (iack_a == IACK_CHECK2) {
        /* B sent negative ack. */
        ldata->successor =
medl_sce_get_membership_pos(get_current_sce(controller));
      }
      break;
    case IACK_CHECK2:
      if (iack_a == IACK_ERROR) {
        /* B and C agree that A failed */
        cni_ps_increment_accept_counter(&lcni->protocol_status, 2);
        c_state_set_membership(&lcni->c_state, ldata->membership_pos, 0);
        c_state_set_membership(&lcni->c_state, ldata->successor, 1);
        c_state_set_membership(&lcni->c_state,
medl_sce_get_membership_pos(get_current_sce(controller)), 1);
        cni_ps_increment_mfc(&lcni->protocol_status, 1);
        /* check number of successive membership failures */
        if (cni_ps_get_mfc(&lcni->protocol_status) >=
medl_ccp_get_mmfc(&controller->medl.controller_cfg_params)
            && medl_ccp_get_mmfc(&controller->medl.controller_cfg_params) >
0)
          set_state(controller, PS_FREEZE);
        else
          set_state(controller, PS_READY);
        /* TODO: interrupt (membership lost) */
      }
      if (iack_a == IACK_DOUBLE_FAILURE) {
        /* B says A is wrong, C failed */
        if (bus_activity(controller))
          cni_ps_increment_failed_counter(&lcni->protocol_status, 2);
        else
          cni_ps_increment_failed_counter(&lcni->protocol_status, 1);
        c_state_set_membership(&lcni->c_state, ldata->successor, 0);
        c_state_set_membership(&lcni->c_state,
medl_sce_get_membership_pos(get_current_sce(controller)), 0);
        set_state(controller, PS_READY);
      }
      if (iack_a == IACK_OK) {
        /* C says B failed, A is OK */
        cni_ps_increment_accept_counter(&lcni->protocol_status, 1);
        cni_ps_increment_failed_counter(&lcni->protocol_status, 1);
        c_state_set_membership(&lcni->c_state, ldata->successor, 0);
      }
      break;
    case IACK_ERROR:
    case IACK_DOUBLE_FAILURE:
      break;
  }

  ldata->iack_state = iack_a;
```

```
return 1;
}

/* performs clock-synchronization algorithm if enabled in MEDL */

void perform_clock_synchronization(CONTROLLER *controller)
{
  CNI_PS *protocol_status = &controller->lcni->protocol_status;
  CNI_FD *fd = &controller->lcni->frame_diagnosis;
  LOCAL_DATA *ldata = &controller->ldata;
signed dif_a, dif_b, sum = 0;
signed i = 0, min, max;

if (medl_sce_get_syf(get_current_sce(controller)) == 1) {
    print_debug((((CONTROLLER_MODEL *)controller)->process, "SYF: AT =
%0.2f\n", ldata->action_time));

    if (cni_fd_get_eif_cha(fd) == EIF_CORRECT_FRAME) {
      dif_a = controller->ldata.mtd_a;
      cni_fd_set_mtd_cha(fd, dif_a);
      sum += dif_a;
      print_debug((((CONTROLLER_MODEL *)controller)->process, "SYF on
channel A: MTD = %d\n", dif_a));
      i++;
    }
    if (cni_fd_get_eif_chb(fd) == EIF_CORRECT_FRAME) {
      dif_b = controller->ldata.mtd_b;
      cni_fd_set_mtd_chb(fd, dif_b);
      sum += dif_b;
      print_debug((((CONTROLLER_MODEL *)controller)->process, "SYF on
channel B: MTD = %d\n", dif_b));
      i++;
    }

    if (i > 0)
      local_data_push_sync_stack(ldata, sum / i);

    print_debug((((CONTROLLER_MODEL *)controller)->process, "CSA Stack: [%i
%i %i %i]\n",
      ldata->sync_stack[0],
      ldata->sync_stack[1],
      ldata->sync_stack[2],
      ldata->sync_stack[3]
    ));
  }
if (medl_sce_get_cs(get_current_sce(controller)) == 1) {
    /* compute clock state correction term */
    min = 0; max = 0;
    sum = ldata->sync_stack[0];
    for (i = 1; i < 4; i++) {
      if (ldata->sync_stack[min] > ldata->sync_stack[i])
        min = i;
      else if (ldata->sync_stack[max] < ldata->sync_stack[i])
        max = i;
      sum += ldata->sync_stack[i];
    }
    /* throw maximum and minum difference away */
    sum = sum - ldata->sync_stack[min] - ldata->sync_stack[max];
    sum = - sum / 2 + cni_get_external_rate_correction(&controller->cni);

    if (abs(sum) / medl_ccp_get_micro_to_macro(&controller-
>medl.controller_cfg_params) > CLOCK_PRECISION / 2.0) {
      /* FIXME: raise interrupt */
      print_debug((((CONTROLLER_MODEL *)controller)->process, "Clock
synchronization error\n"));
      cni_ps_set_se(protocol_status, 1);
      set_state(controller, PS_FREEZE);
    }

    cni_ps_set_csct(protocol_status, sum);
    ldata->clock_correction_time = CSIM_TIME();
  }
}

/* Controller's Post-Receive-Phase.
  Stops current receive operation if stil running.
  Checks received frames, runs Implicit ack. algorithm.
  Updates membership and null frame vectors.
  Data read from N-Frames are stored into CNI. */

void prp(CONTROLLER *controller)
{
unsigned membership_pos =
medl_sce_get_membership_pos(get_current_sce(controller)); /* memb.pos. of
actual sender */
  CNI_MAE dummy_mae;
  CNI_MAE *cni_mae_a, *cni_mae_b; /* CNI message area entry addresses for
both channels */
  CNI_EIF_VALUES eif_a, eif_b;
unsigned iack_in_progress;

  /* Cancel pending receive. */
  stop_receive(controller);

if (cni_ps_get_protocol_state(&controller->lcni->protocol_status) ==
PS_FREEZE)
    return;

  /* We assume that the current sender is valid. */
  c_state_set_membership(&controller->lcni->c_state, membership_pos, 1);

  iack_in_progress = perform_iack(controller);

  /* Setup CNI message area addresses */
if (get_current_mae(controller)->cni_address_cha == CNI_MAE_NO_ADDRESS)
    cni_mae_a = &dummy_mae;
else
    cni_mae_a = cni_get_message_entry(&controller->cni,
medl_mae_get_cni_address_cha(get_current_mae(controller)));
if (get_current_mae(controller)->cni_address_chb == CNI_MAE_NO_ADDRESS)
    cni_mae_b = &dummy_mae;
else
    cni_mae_b = cni_get_message_entry(&controller->cni,
medl_mae_get_cni_address_chb(get_current_mae(controller)));
```

```
  /* Check frame on channel A */
if (controller->ldata.frame[0].state == FRAME_STATE_NULL) {
    eif_a = EIF_NULL_FRAME;
  } else {
    /* Activity on channel A */
    if (controller->ldata.frame[0].state != FRAME_STATE_COMPLETE) {
      eif_a = EIF_INVALID_FRAME;
    } else {
      /* Valid frame on channel A */
      if (frame_is_valid(&controller->ldata.frame[0], &controller->lcni-
>c_state,
          medl_ccp_get_c_state_length(&controller-
>medl.controller_cfg_params))) {
        eif_a = EIF_CORRECT_FRAME;
        /* If N-Frame on channel A then copy data into CNI */
        if (medl_mae_get_ia(get_current_mae(controller)) == 0)
          cni_mae_extract_frame_data(cni_mae_a, &controller->ldata.frame[0],
            medl_mae_get_dafla(get_current_mae(controller)) + 1);
      } else {
        /* CRC error in frame on channel A */
        eif_a = EIF_CRC_ERROR;
      }
    }
  }
  /* Check noise on channel A */
if (controller->ldata.frame[0].noise != 0)
    eif_a = (CNI_EIF_VALUES)(eif_a | EIF_NOISE);
  /* set message status field and frame diagnosis field for channel A */
  cni_fd_set_eif_cha(&controller->lcni->frame_diagnosis, eif_a);
if (medl_mae_get_ia(get_current_mae(controller)) == 0) {
    cni_mae_set_eif(cni_mae_a, eif_a);
    cni_mae_increment_ccf(cni_mae_a, 2);
  }

  /* Check frame on channel B */
if (controller->ldata.frame[1].state == FRAME_STATE_NULL) {
    eif_b = EIF_NULL_FRAME;
  } else {
    /* Activity on channel B */
    if (controller->ldata.frame[1].state != FRAME_STATE_COMPLETE) {
      eif_b = EIF_INVALID_FRAME;
    } else {
      /* Valid frame on channel B */
      if (frame_is_valid(&controller->ldata.frame[1], &controller->lcni-
>c_state,
          medl_ccp_get_c_state_length(&controller-
>medl.controller_cfg_params))) {
        eif_b = EIF_CORRECT_FRAME;
        /* if N-Frame on channel B then copy data into CNI */
        if (medl_mae_get_ib(get_current_mae(controller)) == 0)
          cni_mae_extract_frame_data(cni_mae_b, &controller->ldata.frame[1],
            medl_mae_get_daflb(get_current_mae(controller)) + 1);
      } else {
        eif_b = EIF_CRC_ERROR;
      }
    }
  }
```

```
 /* Check noise on channel B */
if (controller->ldata.frame[1].noise != 0)
  eif_b = (CNI_EIF_VALUES)( eif_b | EIF_NOISE);
 /* set message status field and frame diagnosis field for channel B */
 cni_fd_set_eif_chb(&controller->lcni->frame_diagnosis, eif_b);
if (medl_mae_get_ib(get_current_mae(controller)) == 0) {
  cni_mae_set_eif(cni_mae_b, eif_b);
  cni_mae_increment_ccf(cni_mae_b, 2);
 }

 /* update membership and null-frame vector */
 cni_set_null_frame(controller->lcni, membership_pos, 0);
if (((eif_a | eif_b) & EIF_CORRECT_FRAME) == 0) {
  /* No correct frame received -> sending SRU loses it's membership */
  c_state_set_membership(&controller->lcni->c_state, membership_pos, 0);
  if (((eif_a | eif_b) & EIF_CRC_ERROR) == 0) {
   if (((eif_a | eif_b) & EIF_INVALID_FRAME) == 0) {
    /* Null frame on both channels */
    cni_set_null_frame(controller->lcni, membership_pos, 1);
   } else {
    /* No valid frame received */
    if (!iack_in_progress)
     cni_ps_increment_invalid_counter(&controller->lcni-
>protocol_status, 1);
   }
  }
  else {
   if (!iack_in_progress)
    cni_ps_increment_failed_counter(&controller->lcni->protocol_status,
1);
  }
 }
else {
  if (!iack_in_progress)
   cni_ps_increment_accept_counter(&controller->lcni->protocol_status,
1);
 }

 /* time synchronization algorithm */
 perform_clock_synchronization(controller);

 /* Update medl position and cluster time in C-State */
 set_next_slot(controller);

if (cni_ie_get_ui1(&controller->cni.interrupt_enable) &&
medl_mae_get_ru1(get_current_mae(controller))) {
  cni_is_set_ui1(&controller->lcni->interrupt_status, 1);
  controller->ldata.tp_sync_interrupt = 1;
 }
if (cni_ie_get_ui2(&controller->cni.interrupt_enable) &&
medl_mae_get_ru2(get_current_mae(controller))) {
  cni_is_set_ui2(&controller->lcni->interrupt_status, 1);
  controller->ldata.tp_sync_interrupt = 1;
 }

 /* TODO: mode change in Frame header */
 /* TODO: interrupt membership change */
}
```

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

/* Global functions. */

/* Initializes a Controller data structure.
  Returns a pointer to the initialized structure. */
CONTROLLER *controller_init(CONTROLLER *controller)
{
  memset(&controller->cni, 0, sizeof(CNI));
  memset(&controller->medl, 0, sizeof(MEDL));
  controller->lcni = (CNI *)&controller->lstatus;
  memset(controller->lcni, 0, sizeof(LOCAL_STATUS));

  controller->send_callback = controller_send_callback;
  controller->receive_callback = controller_receive_callback;

  controller->cni.controller = (void *)controller;
  controller->lcni->controller = (void *)controller;

  controller->channel[0] = NULL;
  controller->channel[1] = NULL;
  controller->ldata.host = NULL;
  controller->ldata.host_interrupt_callback = NULL;

  set_state(controller, PS_FREEZE);
  update_cni(controller);

return controller;
}

/* Stores hosts address and interrupt function. */

void controller_attach_host(CONTROLLER *controller, void *host,
PTR_CALLBACK interrupt_func)
{
  controller->ldata.host = host;
  controller->ldata.host_interrupt_callback = interrupt_func;
}

/* Attaches the controller to the specified channel. */

void controller_attach_channel(CONTROLLER *controller, CHANNEL *channel,
unsigned index)
{
  controller->channel[index] = channel;
}

/* Sets the controllers local clock drift in units of [sec/sec] */

void controller_set_clock_drift(CONTROLLER *controller, double drift)
{
  controller->ldata.clock_drift = drift;
}

/* Called by the channel after controller's frame transmission is
completed. */
```

```
void controller_send_callback(void *controller)
{
#define ctrl ((CONTROLLER *)controller)
  ctrl->bus_guardian->arm_signal(ctrl->bus_guardian, 0);
#undef ctrl
}

/* Called by the channel after a frame is received. */

void controller_receive_callback(void *controller, unsigned channel)
{
#define ctrl ((CONTROLLER *)controller)
  LOCAL_DATA *ldata = &ctrl->ldata;
  FRAME *frame = &ldata->frame[channel];

  *frame = ctrl->channel[channel]->frame;

if (cni_ps_get_c_state_valid(&ctrl->lcni->protocol_status) == C_STATE_VALID
      &&  fabs(frame->time_of_delivery - ldata->action_time) >
CLOCK_PRECISION)
    frame->state = FRAME_STATE_NULL;
#undef ctrl
}

/* Called from BG when the controller violates bus access cycle.
  Sets error flag and freezes the controller. */

void controller_bg_error_callback(void *controller)
{
#define ctrl ((CONTROLLER *)controller)
  /* FIXME: raise interrupt */
  cni_ps_set_be(&ctrl->cni.protocol_status, 1);
/*    cni_set_co(ctrl->lcni, 0);*/
  set_state(ctrl, PS_FREEZE);
  update_cni(ctrl);
#undef ctrl
}


/* Raises specified timer interrupt and sets its status flag */

void controller_raise_timer_interrupt(CONTROLLER *controller, unsigned
timer)
{
if (timer == 0 && cni_ie_get_ti1(&controller->cni.interrupt_enable) == 1)
    cni_is_set_ti1(&controller->lcni->interrupt_status, 1);
if (timer == 1 && cni_ie_get_ti2(&controller->cni.interrupt_enable) == 1)
    cni_is_set_ti2(&controller->lcni->interrupt_status, 1);

  raise_interrupt(controller);
}
```

4.15. fp1ctrl.h

```
/*
* This software is part of an EU project:
*     FIT - Fault Injection for TTA
*     IST-1999-10748
```

```
*
* TTP/C - protocol v.0.1
*
* Design (c) TTTech & TU Vienna
* Implementation (c) UWB Pilsen & CTU Prague
*
* fp1ctrl.h
* Functions and definitions concerning the controller
*
* version 0.6 - 2001-03-13
*/

#ifndef _FP1CTRL_H
#define _FP1CTRL_H

#include "debug.h"

#include "fl1func.h"
#include "fp1cni.h"
#include "fp1medl.h"
#include "fp1ldata.h"
#include "fp1chan.h"
#include "fp1busg.h"
struct controller;
typedef void (*FUNC_CONTROLLER_UPDATE_CLUSTER_TIME)(struct controller
*controller);

#define d_controller \
  CNI *ecni;                      /* (public) pointer to cni */          \
  CNI *lcni;                      /* (private) pointer to lstatus */
\
  CNI cni;                        /* (public) the cni */             \
  LOCAL_STATUS lstatus;           /* (private) copy of the status area of
the cni */ \
  MEDL medl;                      /* (private) the medl */            \
  LOCAL_DATA ldata;               /* (private) local_data */            \
  \
  BUS_GUARDIAN *bus_guardian;     /* pointer to controller's bus guardian
*/    \
  CHANNEL *channel[2];            /* pointer to the (2 instances of) bus
*/    \
  \
  FUNC_CONTROLLER_UPDATE_CLUSTER_TIME update_cluster_time; \
  PTR_INT_CALLBACK receive_callback; \
  PTR_CALLBACK send_callback;


/* Controller data structure */
typedef struct controller {
  d_controller
} CONTROLLER;


/* switch the current state of the controller */
void set_state(CONTROLLER *controller, unsigned state);

/* Copies the local CNI status area into the public CNI, updates cluster
time */
void update_cni(CONTROLLER *controller);
```

```
/* returns pointer to the controller's current MEDL Slot Control Entry */
MEDL_SCE *get_current_sce(CONTROLLER *controller);

/* Check whether the controller is sender in current slot */
unsigned is_current_sender(CONTROLLER *controller);

/* Returns whether cold start is allowed (depends on CF flag and I-Frame
switch counter */
unsigned is_cold_start_allowed(CONTROLLER *controller);

/* Raise TP synchronous interrupt if there is any */
void raise_tp_sync_interrupt(CONTROLLER *controller);

/* Sends specified frame to the appropiate channel.
  Returns 0 if succesfull, nozero otherwise */
int send_frame(CONTROLLER *controller, unsigned channel);

/* Starts receiving on both channels.
  Return 0 if successfull, otherwise nonzero stops receiving. */
int start_receive(CONTROLLER *controller);

/* Returns Listen Timeout length in macroticks */
unsigned get_listen_timeout(CONTROLLER *controller);

/* Returns the duration of cold start timeout in macroticks */
unsigned get_cold_start_timeout(CONTROLLER *controller);

/* Sets new BG params if there was no bus activity.
  Returns nonzero code if Bus activity was detected (break) */
int cold_start_set_arm(CONTROLLER *controller);

/* If BG is still in synchonisation mode then set normal mode parameters */
void set_normal_bg_params(CONTROLLER *controller);

/* Sets BG Arm signal if current slot has set the BA field in MEDL */
void set_bg_arm(CONTROLLER *controller);

/* Controller initialization (INIT state):
   - clears CNI, sets Null Frame vect, Personality
   - checks MEDL
   - transits into LISTEN state */
void controller_init_state(CONTROLLER *controller);

/* Initializes the LISTEN state of the controller.
   - checks controller's state
   - sets BG params for cold start nodes
   - sets wake flags
  Returns zero if all conditions are OK, nonzero otherwise (state
transition follows). */
int controller_prepare_listen_state(CONTROLLER *controller);

/* Finishes the LISTEN state of the controller.
   - stops receiving
   - checks bus for I-Frames
   - synchronizes on valid I-Frame
   - enter Cold Start if enabled and bus is silent */
void controller_finish_listen_state(CONTROLLER *controller);

/* Progress ot the cold start state: computes MEDL position.
  Updates C-State of the controller - MEDL position and Time fields */
```

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

/* Initializes controller's Cold Start state (1st part):
   - checks controller's state
   - initializes C-State
   - sends cold start frame, increments I-Frame switch
  Returns zero if everything is OK, otherwise nonzero code (state
transition) */
int controller_prepare_cold_start_state(CONTROLLER *controller);

/* Progress ot the cold start state: computes MEDL position.
  Updates C-State of the controller - MEDL position and Time fields */
void controller_run_cold_start_state(CONTROLLER *controller, unsigned
macroticks);

/* Finishes the COLD_START state of the controller.
   - stops receive, clears wake flags
   - updates time fields and MEDL position
   - checks bus for I-Frames - synchronizes on valid I-Frame
  Returns zero if valid I-Frame was received - synchronize, nonzero
otherwise */
int controller_finish_cold_start_state(CONTROLLER *controller);

/* Controller's Pre Send Phase.
  When a frame should be sent (controller's sending slot), then:
   - Checks frame counters (clique avoidance, blackout) and clears them
   - Prepares frames to be sent in controller's local data
   - Returns nonzero code
  Otherwise zero is returned.
  Function may switch state. */
unsigned psp(CONTROLLER *controller);

/* Perform Implicit acknowledgement algorithm check on specified channel.
  Sets state to the new IACK state but does not change controller's local
data.
  If the new IACK state could not be determined, then *valid is cleared. */
//void check_iack(CONTROLLER *controller, unsigned channel, IACK_STATES
*state, unsigned *valid);

/* Performs implicit acknoledgement algorithm on received frame.
  Sets IACK state in controller's local_data and updates membership vector.
  May switch controller's state.
  Returns zero if the IACK algorithm decision is final, 1 otherwise. */
//unsigned perform_iack(CONTROLLER *controller);

/* performs clock-synchronization algorithm if enabled in MEDL */
//void perform_clock_synchronization(CONTROLLER *controller);

/* Controller's Post-Receive-Phase.
  Stops current receive operation if stil running.
  Checks received frames, runs Implicit ack. algorithm.
  Updates membership and null frame vectors.
  Data read from N-Frames are stored into CNI. */
void prp(CONTROLLER *controller);

/* Initializes a Controller data structure.
  Returns a pointer to the initialized structure. */
CONTROLLER *controller_init(CONTROLLER *controller);
```

```
/* Stores hosts address and interrupt function. */
void controller_attach_host(CONTROLLER *controller, void *host,
PTR_CALLBACK interrupt_func);

/* Attaches the controller to the specified channel. */
void controller_attach_channel(CONTROLLER *controller, CHANNEL *channel,
unsigned index);

/* Sets the controllers local clock drift in units of [sec/sec] */
void controller_set_clock_drift(CONTROLLER *controller, double drift);

/* Called by the channel after controller's frame transmission is
completed. */
void controller_send_callback(void *controller);

/* Called by the channel after a frame is received. */
void controller_receive_callback(void *controller, unsigned channel);

/* Called from BG when the controller violates bus access cycle.
  Sets error flag and freezes the controller. */
void controller_bg_error_callback(void *controller);

/* Raises specified timer interrupt and sets its status flag */
void controller_raise_timer_interrupt(CONTROLLER *controller, unsigned
timer);


#endif /* #ifndef _FP1CTRL_H */
```

4.16. fp1frame.cpp

```
/*
* 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
*
* fp1frame.c
* Functions and definitions concerning frames
*
* version 0.7 - 2001-03-13
*/

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

#include "fp1frame.h"

#include "fp1const.h"
#include "fl1func.h"
#include "fp1cstat.h"

/* FRAME_HEADER field access functions - get and set */
implement_access_functions(FRAME_HEADER, frame_header, i_n_frame, 1)
implement_index_access_functions(FRAME_HEADER, frame_header, mode_change,
1)
```

```
/* calculate crc of FRAME_HEADER - without inicialization */
void frame_header_chain_crc(FRAME_HEADER *frame_header, unsigned *crc,
unsigned polynomial)
{
  get_crc(crc, polynomial, frame_header_get_i_n_frame(frame_header), 1);
  get_crc(crc, polynomial, frame_header_get_mode_change(frame_header, 0),
1);
  get_crc(crc, polynomial, frame_header_get_mode_change(frame_header, 1),
1);
  get_crc(crc, polynomial, frame_header_get_mode_change(frame_header, 2),
1);
  get_crc(crc, polynomial, 0, 4);
}

/* return mode number if deffered mode change is requested, 0 otherwise */
unsigned frame_header_get_dmc(FRAME_HEADER *frame_header)
{
if (frame_header_get_mode_change(frame_header, 0) == 0)
    return (frame_header_get_mode_change(frame_header, 2) << 1)
      +frame_header_get_mode_change(frame_header, 1);
else
    return 0;
}

/* set deffered mode number */
void frame_header_set_dmc(FRAME_HEADER *frame_header, unsigned mode)
{
  frame_header_set_mode_change(frame_header, 0, 0);
  frame_header_set_mode_change(frame_header, 1, mode & 1);
  frame_header_set_mode_change(frame_header, 2, mode & 2);
}


/* return mode number if immediate mode change is requested, 0 otherwise */
unsigned frame_header_get_imc(FRAME_HEADER *frame_header)
{
if (frame_header_get_mode_change(frame_header, 0) == 1)
    return (frame_header_get_mode_change(frame_header, 2) << 1)
      + frame_header_get_mode_change(frame_header, 1);
else
    return 0;
}

/* set immediate mode number */
void frame_header_set_imc(FRAME_HEADER *frame_header, unsigned mode)
{
  frame_header_set_mode_change(frame_header, 0, 1);
  frame_header_set_mode_change(frame_header, 1, mode & 1);
  frame_header_set_mode_change(frame_header, 2, mode & 2);
}


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

/* I_FRAME field access functions - get and set */
implement_access_functions(I_FRAME, i_frame, crc, 16)

/* returns complete CRC code from an I-frame */
```

```
unsigned i_frame_calculate_crc(I_FRAME *frame)
{
unsigned crc_poly, crc;

  crc_poly = ((8 + 16 * frame->c_state_length) < MAX_LENGTH_FOR_CRC_POLY_1)
    ? CRC_POLYNOMIAL_1 : CRC_POLYNOMIAL_2;
  crc = CRC_STARTING_VALUE;

  frame_header_chain_crc(&frame->header, &crc, crc_poly);
  c_state_chain_crc(&frame->c_state, &crc, crc_poly, frame-
>c_state_length);
  get_crc(&crc, crc_poly, 0, 16);

return crc;
}


/* returns physical I-frame size in bits */
unsigned i_frame_get_length(I_FRAME *i_frame)
{
return 8 + 16 * i_frame->c_state_length + 16;
}

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

/* N_FRAME field access functions - get and set */
implement_index_access_functions(N_FRAME, n_frame, application_data, 8)
implement_access_functions(N_FRAME, n_frame, crc, 16)

/* Returns CRC code from a N-frame concatenatedd with C-state */
unsigned n_frame_calculate_crc(N_FRAME *frame, C_STATE *c_state, unsigned
c_state_length)
{
unsigned crc_poly, crc;
unsigned i;

  /* Determine which polynom to use. */
  crc_poly = ((8 + 16 * c_state_length + frame->application_data_length *
8) < MAX_LENGTH_FOR_CRC_POLY_1)
    ? CRC_POLYNOMIAL_1 : CRC_POLYNOMIAL_2;
  crc = CRC_STARTING_VALUE;

  frame_header_chain_crc(&frame->header, &crc, crc_poly);
for (i = 0; i < frame->application_data_length; i++)
    get_crc(&crc, crc_poly, n_frame_get_application_data(frame, i), 8);
  c_state_chain_crc(c_state, &crc, crc_poly, c_state_length);
  get_crc(&crc, crc_poly, 0, 16);

return crc;
}

/* returns physical N-frame size in bits */
unsigned n_frame_get_length(N_FRAME *n_frame)
{
return 8 + 8 * n_frame->application_data_length + 16;
}

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

```
/* returns physical frame size in bits */
unsigned frame_get_length(FRAME *frame)
{
if (frame->type == FRAME_TYPE_N)
    return n_frame_get_length(&frame->frame.n);
else if (frame->type == FRAME_TYPE_I)
    return i_frame_get_length(&frame->frame.i);
else
    return 0;
}

/* returns 1 if frame is complete, it is an I-frame with valid CRC */
unsigned frame_is_valid_i_frame(FRAME *frame)
{
if (frame->type != FRAME_TYPE_I || frame->state != FRAME_STATE_COMPLETE)
    return 0;
else
    return i_frame_get_crc(&frame->frame.i) ==
i_frame_calculate_crc(&frame->frame.i);
}

/* returns 1 if frame is complete, it is a N-frame with valid CRC */
unsigned frame_is_valid_n_frame(FRAME *frame, C_STATE *c_state, unsigned
c_state_length)
{
if (frame->type != FRAME_TYPE_N || frame->state != FRAME_STATE_COMPLETE)
    return 0;
else
    return n_frame_get_crc(&frame->frame.n) ==
n_frame_calculate_crc(&frame->frame.n, c_state, c_state_length);
}

/* returns 1 if frame is complete, has valid CRC and its c_state
corresponds to the given one. */
unsigned frame_is_valid(FRAME *frame, C_STATE *c_state, unsigned
c_state_length)
{
switch (frame->type) {
    case FRAME_TYPE_I:
      return frame_is_valid_i_frame(frame) && c_state_cmp(&frame-
>frame.i.c_state, c_state);
    case FRAME_TYPE_N:
      return frame_is_valid_n_frame(frame, c_state, c_state_length);
   }
return 0;
}
```

4.17. fp1frame.h

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

```
* fp1frame.h
* Functions and definitions concerning frames
*
* version 0.8 - 2001-03-17
*/


#ifndef _FP1FRAME_H
#define _FP1FRAME_H

#include "fp1const.h"
#include "fp1cstat.h"
#include "fl1func.h"

/* frame header structure */
typedef struct {
unsigned i_n_frame;
unsigned mode_change[3];
} FRAME_HEADER;

/* FRAME_HEADER field access functions - get and set */
declare_access_functions(FRAME_HEADER, frame_header, i_n_frame);
declare_index_access_functions(FRAME_HEADER, frame_header, mode_change);

/* calculate crc of FRAME_HEADER - without inicialization */
void frame_header_chain_crc(FRAME_HEADER *frame_header, unsigned *crc,
unsigned polynomial);

/* return mode number if deffered mode change is requested, 0 otherwise */
unsigned frame_header_get_dmc(FRAME_HEADER *frame_header);

/* set deffered mode number */
void frame_header_set_dmc(FRAME_HEADER *frame_header, unsigned mode);

/* return mode number if immediate mode change is requested, 0 otherwise */
unsigned frame_header_get_imc(FRAME_HEADER *frame_header);

/* set immediate mode number */
void frame_header_set_imc(FRAME_HEADER *frame_header, unsigned mode);

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

/* I-frame structure */
typedef struct {
  FRAME_HEADER header;
  C_STATE c_state;
unsigned crc;
  /* Stored because it makes things easier */
unsigned c_state_length;
} I_FRAME;

/* I_FRAME field access functions - get and set */
declare_access_functions(I_FRAME, i_frame, crc);

/* returns physical I-frame size in bits */
unsigned i_frame_get_length(I_FRAME *i_frame);

/* returns complete CRC code from an I-frame */
unsigned i_frame_calculate_crc(I_FRAME *frame);
```

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

/* N-frame structure */
typedef struct {
  FRAME_HEADER header;
unsigned char application_data[MAX_APPLICATION_DATA_SIZE];
unsigned crc;
  /* Stored because it makes things easier. */
unsigned application_data_length;
} N_FRAME;

/* N_FRAME field access functions - get and set */
declare_index_access_functions(N_FRAME, n_frame, application_data);
declare_access_functions(N_FRAME, n_frame, crc);

/* returns physical N-frame size in bits */
unsigned n_frame_get_length(N_FRAME *n_frame);

/* Returns CRC code from a N-frame concatenated with C-state */
unsigned n_frame_calculate_crc(N_FRAME *frame, C_STATE *c_state, unsigned
c_state_length);

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

/* union for sharing memory between I_FRAME and N_FRAME */
typedef union {
  N_FRAME n;
  I_FRAME i;
} FRAME_UNION;

/* frame type codes */
typedef enum {
  FRAME_TYPE_N,
  FRAME_TYPE_I
} FRAME_TYPES;

/* frame state codes */
typedef enum {
  FRAME_STATE_NULL,
  FRAME_STATE_RECEIVING,
  FRAME_STATE_COMPLETE
} FRAME_STATES;

/* structure for both frame types */
typedef struct {
  FRAME_TYPES type;
  FRAME_STATES state;
double time_of_delivery;
unsigned noise;                    /* Set on transmission colision */
  FRAME_UNION frame;
} FRAME;

/* returns physical frame size in bits */
unsigned frame_get_length(FRAME *frame);

/* returns 1 if frame is complete, it is an I-frame with valid CRC */
unsigned frame_is_valid_i_frame(FRAME *frame);

/* returns 1 if frame is complete, it is a N-frame with valid CRC */
```

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

/* returns 1 if frame is complete, has valid CRC and its c_state
corresponds to the given one. */
unsigned frame_is_valid(FRAME *frame, C_STATE *c_state, unsigned
c_state_length);


#endif /* #ifndef _FP1FRAME_H */
```

4.18. fp1chan.cpp

```
/*
* 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
*
* fp1chan.c
* Definition of the Bus Channel
*
* version 0.2 - 2001-03-18
*/

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

#include "fp1chan.h"

#include "fp1busg.h"
#include "sp1ctrl.h"
#include "fl1func.h"

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

implement_access_functions(CHANNEL, channel, bus_id, 16);
implement_access_functions(CHANNEL, channel, active, 1);

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

/* Initializes CHANNEL data fields,
  returns channel. */
CHANNEL *channel_init(CHANNEL *channel)
{
  channel->active = 0;
  channel->receiver_count = 0;
  channel->frame.state = FRAME_STATE_NULL;
  channel->bus_id = 0;

  channel->bg_error_callback = NULL;

return channel;
}
```

```
/* Registers a controller to receive next sent frame.
  A callback function is called after receiving the frame.
  The callback function is called: callback(controller, channel).
  Returns 0 on success, nonzero otherwise. */

int channel_receive(CHANNEL *channel, void *controller, PTR_INT_CALLBACK
callback)
{
  assert(channel->receiver_count < MAX_SRU_COUNT);

  ((CONTROLLER *)controller)->ldata.frame[channel->bus_id] = channel-
>frame;

if (!channel->active) {
    channel->receiver[channel->receiver_count] = controller;
    channel->receive_callback[channel->receiver_count] = callback;
    channel->receiver_count++;
    ((CONTROLLER *)controller)->ldata.frame[channel->bus_id].state =
FRAME_STATE_NULL;
  } else {
    ((CONTROLLER *)controller)->ldata.frame[channel->bus_id].state =
FRAME_STATE_RECEIVING;
  }

return channel->active;
}

/* Unregisters a controller so it no more listens on the channel. */

void channel_cancel_receive(CHANNEL *channel, void *controller)
{
unsigned i = 0;

while (i < channel->receiver_count && channel->receiver[i] != controller)
    i++;

if (i < channel->receiver_count)
    channel->receiver_count--;
while (i < channel->receiver_count) {
    channel->receiver[i] = channel->receiver[i + 1];
    channel->receive_callback[i] = channel->receive_callback[i + 1];
    i++;
  }
}


/* The frame is copied from controller's ldata (according to bus_id value).
  Sending controller is excluded from receiving the frame.
  Returns nonzero code in case of aby error, 0 otherwise */

int channel_send_init(CHANNEL *channel, void *controller, PTR_CALLBACK
callback)
{
unsigned id = channel->bus_id;
  FRAME *frame = &((CONTROLLER *)controller)->ldata.frame[id];
unsigned i;
```

```
if (test_bus_guardian(channel, controller))
    return 1;    /* BG Closed */

if (channel->active) {
    assert(0);
    exit(1);
    channel->frame.noise = 1;
    return 2;    /* Silence Violation */
  }

  channel->active = 1;
  channel->sender = controller;
  channel->send_callback = callback;

  /* Handle sender separately */
  frame->noise = 0;
  frame->state = FRAME_STATE_COMPLETE;
  channel_cancel_receive(channel, controller);

  /* Store frame to transmitt */
  channel->frame = *frame;
  channel->frame.state = FRAME_STATE_COMPLETE;

  /* Notify all receivers */
for (i = 0; i < channel->receiver_count; i++)
    ((CONTROLLER *)channel->receiver[i])->ldata.frame[channel-
>bus_id].state = FRAME_STATE_RECEIVING;

return 0;
}

/* Checks if Bus Guardian is open to write and if not then raises a BG
error.
  Returns 0 if BG is open, nonzero code otherwise */

int test_bus_guardian(CHANNEL *channel, void *controller)
{
if (!bus_guardian_is_enabled((BUS_GUARDIAN *)((CONTROLLER *)controller)-
>bus_guardian, channel->bus_id)) {
    if (channel->bg_error_callback != NULL)
      channel->bg_error_callback(controller);
    return 1;
  }
return 0;
}

/* Sets Frame State to COMPLETE if there was no noise while transmiting and
  Sends notifications to all receivers and also to the sender.
  A last clears all transmission fields. */

void channel_send_complete(CHANNEL *channel)
{
unsigned i;

if (channel->frame.noise)
    channel->frame.state = FRAME_STATE_RECEIVING;

for (i = 0; i < channel->receiver_count; i++) {
    ((CONTROLLER *)channel->receiver[i])->ldata.frame[channel->bus_id] =
channel->frame;
    channel->receive_callback[i](channel->receiver[i], channel->bus_id);
   }
  channel->receiver_count = 0;

if (channel->send_callback != NULL)
    channel->send_callback(channel->sender);

  channel->active = 0;
  channel->frame.noise = 0;
  channel->frame.state = FRAME_STATE_NULL;
}

/* Assigns a callback function for BG error events. */

void channel_set_bg_error_callback(CHANNEL *channel, PTR_CALLBACK callback)
{
  channel->bg_error_callback = callback;
}

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

4.19.   fp1chan.h

```
/*
* This software is part of an EU project:
*     FIT - Fault Injection for TTA
*     IST-1999-10748
*
* TTP/C - protocol v.0.1
*
* Design (c) TTTech & TU Vienna
* Implementation (c) UWB Pilsen & CTU Prague
*
* fp1chan.h
* Definition of the Bus Channel
*
* version 0.2 - 2001-03-18
*/

#ifndef _FP1CHAN_H
#define _FP1CHAN_H

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

#include "fp1frame.h"
#include "fl1func.h"

/*************************************************************************
***/
struct channel;

/* channel_send_frame(channel, controller, callback) prototype */
typedef int (*FUNC_CHANNEL_SEND_FRAME)(struct channel *, void *,
PTR_CALLBACK);
```

```
/* CHANNEL data fields required by the protocol */
#define d_channel \
unsigned bus_id; \
int active; \
  FRAME frame; \
  /* Sender */ \
void *sender; \
  /* Receivers */ \
unsigned receiver_count; \
void *receiver[MAX_SRU_COUNT]; \
  /* Callback functions */ \
  PTR_CALLBACK send_callback; \
  PTR_INT_CALLBACK receive_callback[MAX_SRU_COUNT]; \
  PTR_CALLBACK bg_error_callback; \
  /* Virtual functions - implemented in another module */ \
  FUNC_CHANNEL_SEND_FRAME channel_send_frame;

/* CHANNEL - data structure containing all protocol dependant field */
typedef struct channel {
  d_channel
} CHANNEL;

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

declare_access_functions(CHANNEL, channel, bus_id);
declare_access_functions(CHANNEL, channel, active);

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

/* Initializes CHANNEL data fields,
  returns channel. */
CHANNEL *channel_init(CHANNEL *channel);

/* Registers a controller to receive next sent frame.
  A callback function is called after receiving the frame.
  The callback function is called: callback(controller, channel).
  Returns 0 on success, nonzero otherwise. */
int channel_receive(CHANNEL *channel, void *controller, PTR_INT_CALLBACK
callback);

/* Unregisters a controller so it no more listens on the channel. */
void channel_cancel_receive(CHANNEL *channel, void *controller);

/* The frame is copied from controller's ldata (according to bus_id value).
  Sending controller is excluded from receiving the frame.
  Returns nonzero code in case of aby error, 0 otherwise */
int channel_send_init(CHANNEL *channel, void *controller, PTR_CALLBACK
callback);

/* Checks if Bus Guardian is open to write and if not then raises a BG
error.
  Returns 0 if BG is open, nonzero code otherwise */
int test_bus_guardian(CHANNEL *channel, void *controller);

/* Sets Frame State to COMPLETE if there was no noise while transmiting and
  Sends notifications to all receivers and also to the sender.
  A last clears all transmission fields. */
```

```
void channel_send_complete(CHANNEL *channel);

/* Assigns a callback function for BG error events. */
void channel_set_bg_error_callback(CHANNEL *channel, PTR_CALLBACK
callback);

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

#endif /* #ifndef _FP1CHAN_H */
```

4.20. fp1ldata.cpp

```
/*
* 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
*
* fp1ldata.c
* Functions and definitions concerning the LOCAL_DATA structure (controller
private local data)
*
* version 0.11 - 2001-03-18
*/

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

#include "fp1ldata.h"

#include "kr_csim.h"
#include "fl1func.h"

/* local data initialization */

void local_data_initialize(LOCAL_DATA *ldata)
{
unsigned i;

  ldata->last_clock_access = CSIM_TIME();
  ldata->last_mt_fract = 0;

  ldata->clock_correction_time = CSIM_TIME();
  ldata->iack_state = IACK_OK;

for (i = 0; i < SYNC_STACK_SIZE; i++)
    ldata->sync_stack[i] = 0;
  ldata->sync_stack_top = 0;
  ldata->tp_sync_interrupt = 0;

  ldata->immediate_send_permission = 0;
}
```

```
/* stores a new value at the top of the sync_stack, increments top
pointer */

signed local_data_push_sync_stack(LOCAL_DATA *ldata, signed value)
{
  ldata->sync_stack[ldata->sync_stack_top] = value;
  ldata->sync_stack_top = (ldata->sync_stack_top + 1)    % SYNC_STACK_SIZE;
return value;
}
```

4.21. fp1ldata.h

```
/*
* This software is part of an EU project:
*     FIT - Fault Injection for TTA
*     IST-1999-10748
*
* TTP/C - protocol v.0.1
*
* Design (c) TTTech & TU Vienna
* Implementation (c) UWB Pilsen & CTU Prague
*
* fp1ldata.h
* Functions and definitions concerning the LOCAL_DATA structure (controller
private local data)
*
* version 0.13 - 2001-03-18
*/

#ifndef _FP1LDATA_H
#define _FP1LDATA_H

#include "fl1func.h"
#include "fp1frame.h"
#include "kr_csim.h"

/* Implicit Acknowledgement algorithm states */
typedef enum {
  IACK_OK = 0,             /* No check needs to be performed. */
  IACK_SEND = 1,            /* Initial state after sending a frame */
  IACK_CHECK1 = 2,         /* Check I is about to be performed. */
  IACK_CHECK2 = 3,         /* Check II is about to be performed. */
  IACK_ERROR = 4,          /* Both checks performed and the result is that
this controller had failed. */
  IACK_DOUBLE_FAILURE = 5 /* Double failure */
} IACK_STATES;
typedef struct {
  FRAME frame[2];            /* temporary location for received or sent
frames */

  /* These three values are used in impl. ack. algorithm realization. */
  IACK_STATES iack_state;        /* Impl. Ack. Algorithm state. */
unsigned successor;        /* membership position of direct successor (not
counting those who failed) */

unsigned membership_pos;     /* this controller's membership position */

signed sync_stack[SYNC_STACK_SIZE]; /* push-down stack for meassured time
difference */
unsigned sync_stack_top;    /* points above the most recent item in stack
*/

double action_time;        /* Next action time in CSIM time */

double clock_drift;         /* controller'a internal clock drift [sec/sec]
*/
double clock_correction_time;   /* last applied clock correction time */

double last_clock_access;       /* Model time of the last read of the clock
*/
double last_mt_fract;           /* Fractional part of elapsed macroticks in
the time of last_clock_access. */

void *host;                /* Pointer to the host process */

unsigned tp_sync_interrupt;     /* This flag is set when transmission phase
sync. interrupt is about to be raised. */
  PTR_CALLBACK host_interrupt_callback;    /* host interrupt function
address */

int immediate_send_permission;

signed mtd_a, mtd_b;    /* Measured time difference, used by clock sync.
algorithm. */
} LOCAL_DATA;

/* initialize local data with startup values */
void local_data_initialize(LOCAL_DATA *ldata);

/* stores a new value at the top of the sync_stack, increments top pointer
  returns value */
signed local_data_push_sync_stack(LOCAL_DATA *ldata, signed value);


#endif /* #ifndef _FP1LDATA_H */
```

4.22. fp1lname.cpp

```
/*
* 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
*
* fp1lname.c
* Functions and definitions concerning LOGICAL_NAME structure
*
* version 0.5 - 2001-03-13
*/

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

#include "fp1lname.h"
```

```
#include "fl1func.h"

/* LOGICAL_NAME field access functions - get and set */
implement_access_functions(LOGICAL_NAME, logical_name, ps, 1)
implement_access_functions(LOGICAL_NAME, logical_name, slot_position, 6)
implement_access_functions(LOGICAL_NAME, logical_name, multiplex_id, 4)


/* other functions */

/* compares two LOGICAL_NAME structures, returns 1 if equal */
int logical_name_cmp(LOGICAL_NAME *l1, LOGICAL_NAME *l2)
{
return (logical_name_get_ps(l1) == logical_name_get_ps(l2))
    && (logical_name_get_slot_position(l1) ==
logical_name_get_slot_position(l2))
    && (logical_name_get_multiplex_id(l1) ==
logical_name_get_multiplex_id(l2));
}
```

4.23. fp1lname.h

```
/*
* This software is part of an EU project:
*     FIT - Fault Injection for TTA
*     IST-1999-10748
*
* TTP/C - protocol v.0.1
*
* Design (c) TTTech & TU Vienna
* Implementation (c) UWB Pilsen & CTU Prague
*
* fp1lname.h
* Functions and definitions concerning LOGICAL_NAME structure
*
* version 0.5 - 2001-03-13
*/

#ifndef _FP1LNAME_H
#define _FP1LNAME_H

#include "fl1func.h"

/* logical name field structure */
typedef struct {
unsigned _unused; /* 5 unused bits */
unsigned ps;        /* passive */
unsigned slot_position;
unsigned multiplex_id;
} LOGICAL_NAME;

/* LOGICAL_NAME field access functions - get and set */
declare_access_functions(LOGICAL_NAME, logical_name, ps);
declare_access_functions(LOGICAL_NAME, logical_name, slot_position);
declare_access_functions(LOGICAL_NAME, logical_name, multiplex_id);


/* compares two LOGICAL_NAME structures, returns 1 if equal */
int logical_name_cmp(LOGICAL_NAME *l1, LOGICAL_NAME *l2);

#endif /* #ifndef _FP1LNAME_H */
```

4.24. fp1mdlrd.cpp

```
/*
* 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
*
* fp1mdlrd.cpp
* Reading MEDL from a text file
*
* version 0.3 - 2001-03-13
*/


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

#include "fp1mdlrd.h"

#include <stdio.h>
#include <string.h>
#include "fp1medl.h"
#include "fp1const.h"

#define MAX_LINE_LEN        1024
#define MAX_FILE_NAME_LEN    128

#define UNSIGNED_PTR    1
#define DOUBLE_PTR    2

/* Expands given field identifier into an absolute field name */

static int expand(char *s);

/* returns pointer to the name part after the first dot */

static char *get_second_part(char *str);
/* removes the name part after the first dot */

static char *get_first_part(char *str);

/* Finds field position in MEDL and also the type of the field
  medl     - pointer to target MEDL tructure
  id     - field name (absolute)
  ptr    - found position in the MEDL
  type    - found item type (integer, float)
 returns 0 on success */

static int get_ptr_MEDL(MEDL *s, void **ptr, int *type, char *item);
/* Finds field position in MEDL_BGP and also the type of the field. See
get_ptr_MEDL. */

static int get_ptr_MEDL_BGP(MEDL_BGP *s, void **ptr, int *type, char
*item);
```

```
/* Finds field position in MEDL_MAE and also the type of the field. See
get_ptr_MEDL. */

static int get_ptr_MEDL_MAE(MEDL_MAE *s, void **ptr, int *type, char
*item);
/* Finds field position in MEDL_SCE and also the type of the field. See
get_ptr_MEDL. */

static int get_ptr_MEDL_SCE(MEDL_SCE *s, void **ptr, int *type, char
*item);
/* Finds field position in MEDL_ISP and also the type of the field. See
get_ptr_MEDL. */

static int get_ptr_MEDL_ISP(MEDL_ISP *s, void **ptr, int *type, char
*item);
/* Finds field position in MEDL_MCE and also the type of the field. See
get_ptr_MEDL. */

static int get_ptr_MEDL_MCE(MEDL_MCE *s, void **ptr, int *type, char
*item);
/* Finds field position in MEDL_CCP and also the type of the field. See
get_ptr_MEDL. */

static int get_ptr_MEDL_CCP(MEDL_CCP *s, void **ptr, int *type, char
*item);
/* Finds field position in MEDL_RL and also the type of the field. See
get_ptr_MEDL. */

static int get_ptr_MEDL_RL(MEDL_RL *s, void **ptr, int *type, char *item);
/* Finds field position in LOGICAL_NAME and also the type of the field. See
get_ptr_MEDL. */

static int get_ptr_LOGICAL_NAME(LOGICAL_NAME * s, void **ptr, int *type,
char *item);
static char last_valid_id[MAX_LINE_LEN];


/* Number of the line where an error stopped execution */

int medl_read_error_line;


/* Reads MEDL from a text file in the current directory. File name is
derived
  from the specified index, for example: "medl01.txt", "medl12.txt".
  Read data are stored into MEDL prepared at given address.
  Field not present in the source file are set to zero.
  Return code of 0 means successfull completion, any other code is
described
  in MEDL_READ_ERROR_CODES */
MEDL_READ_ERROR_CODES medl_read_from_file(MEDL *medl, unsigned index) {
char fname[MAX_FILE_NAME_LEN];
char line[MAX_LINE_LEN];
  FILE *f = NULL;

#define ERROR(error) do { \
if (f != NULL) \
    fclose(f); \
return (error); \
} while(0)
#define CHECK(error, cond) if (!(cond)) ERROR(error)

  CHECK(MEDL_READ_INDEX_ERROR, index <= 99);
  CHECK(MEDL_READ_NULL_MEDL_ERROR, medl != NULL);

  sprintf(fname, MEDL_READ_FILE_NAME, index);
  f = fopen(fname, "r");
  CHECK(MEDL_READ_FILE_OPEN_ERROR, f != NULL);

  memset(medl, 0, sizeof(MEDL));

  last_valid_id[0] = '\0';

  medl_read_error_line = 0;
while(!feof(f)) {
    char value[MAX_LINE_LEN], name[MAX_LINE_LEN];
    void *ptr;
    int type;

    medl_read_error_line++;
    if (fgets(line, MAX_LINE_LEN, f) == NULL)
      if (!feof(f))
        ERROR(MEDL_READ_FILE_READ_ERROR);

    if (line[0] == '#')
      continue;
    if (sscanf(line, "%s %s", value, name) != 2)
      continue;

    CHECK(MEDL_READ_INVALID_ID_ERROR, expand(name) == 0);

    if (get_ptr_MEDL(medl, &ptr, &type, name) == 0) {
      switch (type) {
      case UNSIGNED_PTR:
        sscanf(value, "%d", (unsigned *)ptr);
        break;

      case DOUBLE_PTR:
        sscanf(value, "%lf", (double *)ptr);
        break;

      default:
        ERROR(MEDL_READ_UNKNOWN_TYPE_ERROR);
      }
    } else
      ERROR(MEDL_READ_UNKNOWN_ID_ERROR);
  }

  fclose(f);

return MEDL_READ_NO_ERROR;
}

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

/* Expands given field identifier into an absolute field name */
```

```
static int expand(char *id)
{
int dot_cnt = 0;
char *c;

if (id[0] != '.') {
    /* OK: Does not begin with a dot -> absolute name */
    strcpy(last_valid_id, id);
    return 0;
  };

if ((id[0] == '.') && (last_valid_id[0] == '\0'))
    /* ERROR: relative field name without parent specification */
    return 1;

  /* Get dot count */
for (c = id; *c; c++)
    if (*c == '.')
      dot_cnt++;

  /* find the end of string */
for (c = last_valid_id; *c; c++)
    ;
  /* find first matching dot */
for( ; c >= last_valid_id; c--)
    if (*c == '.' && --dot_cnt == 0)
      break;

if (c < last_valid_id)
    /* ERROR: relative name contains more dots than absolute name */
    return 2;

if ((c - last_valid_id + strlen(id)) >= MAX_LINE_LEN)
    /* ERROR: expanded name is too long */
    return 3;

  strcpy(c, id);
  strcpy(id, last_valid_id);

return 0;
}

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

/* returns pointer to the name part after the first dot */

static char *get_second_part(char *str)
{
char *c = str;

while ((*c != '.') && (*c != '\0'))
    c++;
if (*c != '\0')
    return c+1;
return c;
}

/* removes the name part after the first dot */
```

```
static char *get_first_part(char *str)
{
char *c = str;

while ((*c != '.') && (*c != '\0'))
    c++;
  *c = '\0';

return str;
}


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

/* Finds field position in MEDL and also the type of the field
  medl     - pointer to target MEDL tructure
  id     - field name (absolute)
  ptr    - found position in the MEDL
  type    - found item type (integer, float)
 returns 0 on success */

static int get_ptr_MEDL(MEDL *medl, void **ptr, int *type, char *id)
{
char *item, *subitems;

  subitems = get_second_part(id);
  item = get_first_part(id);

if (strncmp(item, "implementation_params", 21) == 0) {
    return get_ptr_MEDL_ISP(&(medl->implementation_params), ptr, type,
subitems);

  } else if (strncmp(item, "bus_guardian_cold_start", 23) == 0) {
    return get_ptr_MEDL_BGP(&(medl->bus_guardian_cold_start), ptr, type,
subitems);

  } else if (strncmp(item, "mode_address_entry", 18) == 0) {
    int i1, i2;

    if (sscanf(item, "mode_address_entry[%d][%d]", &i1, &i2) != 2)
      return 2;    /* ERROR */
    return get_ptr_MEDL_MAE(&(medl->mode_address_entry[i1][i2]), ptr, type,
subitems);

  } else if (strncmp(item, "slot_control_entry", 18) == 0) {
    int i1;

    if (sscanf(item, "slot_control_entry[%d]", &i1) != 1)
      return 2;    /* ERROR */
    return get_ptr_MEDL_SCE(&(medl->slot_control_entry[i1]), ptr, type,
subitems);

  } else if (strncmp(item, "bus_guardian_download", 21) == 0) {
    return get_ptr_MEDL_BGP(&(medl->bus_guardian_download), ptr, type,
subitems);

  } else if (strncmp(item, "role_list", 9) == 0) {
    return get_ptr_MEDL_RL(&(medl->role_list), ptr, type, subitems);
  } else if (strncmp(item, "mode_control_entry", 18) == 0) {
    int i1;

    if (sscanf(item, "mode_control_entry[%d]", &i1) != 1)
      return 2;    /* ERROR */
    return get_ptr_MEDL_MCE(&(medl->mode_control_entry[i1]), ptr, type,
subitems);

  } else if (strncmp(item, "controller_cfg_params", 21) == 0) {
    return get_ptr_MEDL_CCP(&(medl->controller_cfg_params), ptr, type,
subitems);

  } else if (strncmp(item, "revision", 8) == 0) {
    *ptr = (void *)&(medl->revision);
    *type = UNSIGNED_PTR;
    return 0;

  } else
    return 1;    /* ERROR */
}

/* Finds field position in MEDL_BGP and also the type of the field
   bgp    - pointer to target MEDL_BGP tructure
   id     - field name (absolute)
   ptr    - found position in the MEDL_BGP
   type    - found item type (integer, float)
  returns 0 on success */

static int get_ptr_MEDL_BGP(MEDL_BGP *bgp, void **ptr, int *type, char *id)
{
char *item;

  item = get_first_part(id);

if (strncmp(item, "crc", 3) == 0) {
    *ptr = (void *)&(bgp->crc);
    *type = UNSIGNED_PTR;
    return 0;

  } else if (strncmp(item, "bg_slot_offset", 14) == 0) {
    *ptr = (void *)&(bgp->bg_slot_offset);
    *type = UNSIGNED_PTR;
    return 0;

  } else if (strncmp(item, "bg_remaining_round", 18) == 0) {
    *ptr = (void *)&(bgp->bg_remaining_round);
    *type = UNSIGNED_PTR;
    return 0;

  } else if (strncmp(item, "bg_slot_duration", 16) == 0) {
    *ptr = (void *)&(bgp->bg_slot_duration);
    *type = UNSIGNED_PTR;
    return 0;

  } else
    return 1;    /* ERROR */
```

```
}

/* Finds field position in LOGICAL_NAME and also the type of the field
  lname     - pointer to target LOGICAL_NAME tructure
  id     - field name (absolute)
  ptr    - found position in the LOGICAL_NAME
  type    - found item type (integer, float)
 returns 0 on success */

static int get_ptr_LOGICAL_NAME(LOGICAL_NAME *lname, void **ptr, int *type,
char *id)
{
char *item;

  item = get_first_part(id);

if (strncmp(item, "multiplex_id", 12) == 0) {
    *ptr = (void *)&(lname->multiplex_id);
    *type = UNSIGNED_PTR;
    return 0;

  } else if (strncmp(item, "ps", 2) == 0) {
    *ptr = (void *)&(lname->ps);
    *type = UNSIGNED_PTR;
    return 0;

  } else if (strncmp(item, "slot_position", 13) == 0) {
    *ptr = (void *)&(lname->slot_position);
    *type = UNSIGNED_PTR;
    return 0;

  } else
    return 1;    /* ERROR */
}

/* Finds field position in MEDL_MAE and also the type of the field
  mae     - pointer to target MEDL_MAE tructure
  id     - field name (absolute)
  ptr    - found position in the MEDL_MAE
  type    - found item type (integer, float)
 returns 0 on success */

static int get_ptr_MEDL_MAE(MEDL_MAE *mae, void **ptr, int *type, char *id)
{
char *item;

  item = get_first_part(id);

if (strncmp(item, "mcp_c", 5) == 0) {
    *ptr = (void *)&(mae->mcp_c);
    *type = UNSIGNED_PTR;
    return 0;

  } else if (strncmp(item, "mcp_d", 5) == 0) {
    int i1;

    if (sscanf(item, "mcp_d[%d]", &i1) != 1)
      return 2;    /* ERROR */
    *ptr = (void *)&(mae->mcp_d[i1]);
    *type = UNSIGNED_PTR;
    return 0;

  } else if (strncmp(item, "eoc", 3) == 0) {
    *ptr = (void *)&(mae->eoc);
    *type = UNSIGNED_PTR;
    return 0;

  } else if (strncmp(item, "crc", 3) == 0) {
    *ptr = (void *)&(mae->crc);
    *type = UNSIGNED_PTR;
    return 0;

  } else if (strncmp(item, "dafla", 5) == 0) {
    *ptr = (void *)&(mae->dafla);
    *type = UNSIGNED_PTR;
    return 0;

  } else if (strncmp(item, "ru1", 3) == 0) {
    *ptr = (void *)&(mae->ru1);
    *type = UNSIGNED_PTR;
    return 0;

  } else if (strncmp(item, "mcp_i", 5) == 0) {
    int i1;

    if (sscanf(item, "mcp_i[%d]", &i1) != 1)
      return 2;     /* ERROR */
    *ptr = (void *)&(mae->mcp_i[i1]);
    *type = UNSIGNED_PTR;
    return 0;

  } else if (strncmp(item, "daflb", 5) == 0) {
    *ptr = (void *)&(mae->daflb);
    *type = UNSIGNED_PTR;
    return 0;

  } else if (strncmp(item, "ru2", 3) == 0) {
    *ptr = (void *)&(mae->ru2);
    *type = UNSIGNED_PTR;
    return 0;

  } else if (strncmp(item, "ia", 2) == 0) {
    *ptr = (void *)&(mae->ia);
    *type = UNSIGNED_PTR;
    return 0;

  } else if (strncmp(item, "cni_address_cha", 15) == 0) {
    *ptr = (void *)&(mae->cni_address_cha);
    *type = UNSIGNED_PTR;
    return 0;

  } else if (strncmp(item, "ib", 2) == 0) {
    *ptr = (void *)&(mae->ib);
    *type = UNSIGNED_PTR;
    return 0;

  } else if (strncmp(item, "cni_address_chb", 15) == 0) {
    *ptr = (void *)&(mae->cni_address_chb);
    *type = UNSIGNED_PTR;
    return 0;

  } else
    return 1;
}

/* Finds field position in MEDL_SCE and also the type of the field
  sce     - pointer to target MEDL_SCE tructure
  id     - field name (absolute)
  ptr    - found position in the MEDL_SCE
  type    - found item type (integer, float)
 returns 0 on success */

static int get_ptr_MEDL_SCE(MEDL_SCE *sce, void **ptr, int *type, char *id)
{
char *item, *subitems;

  subitems = get_second_part(id);
  item = get_first_part(id);

if (strncmp(item, "eot", 3) == 0) {
    *ptr = (void *)&(sce->eot);
    *type = UNSIGNED_PTR;
    return 0;

  } else if (strncmp(item, "sru_slot_duration", 17) == 0) {
    *ptr = (void *)&(sce->sru_slot_duration);
    *type = UNSIGNED_PTR;
    return 0;

  } else if (strncmp(item, "syf", 3) == 0) {
    *ptr = (void *)&(sce->syf);
    *type = UNSIGNED_PTR;
    return 0;

  } else if (strncmp(item, "mm", 2) == 0) {
    *ptr = (void *)&(sce->mm);
    *type = UNSIGNED_PTR;
    return 0;

  } else if (strncmp(item, "logical_name", 12) == 0) {
    return get_ptr_LOGICAL_NAME(&(sce->logical_name), ptr, type, subitems);

  } else if (strncmp(item, "ba", 2) == 0) {
    *ptr = (void *)&(sce->ba);
    *type = UNSIGNED_PTR;
    return 0;

  } else if (strncmp(item, "ra", 2) == 0) {
    *ptr = (void *)&(sce->ra);
    *type = UNSIGNED_PTR;
    return 0;
  } else if (strncmp(item, "medl_pos", 8) == 0) {
    *ptr = (void *)&(sce->medl_pos);
    *type = UNSIGNED_PTR;
    return 0;

  } else if (strncmp(item, "mo", 2) == 0) {
    *ptr = (void *)&(sce->mo);
    *type = UNSIGNED_PTR;
    return 0;

  } else if (strncmp(item, "membership_pos", 14) == 0) {
    *ptr = (void *)&(sce->membership_pos);
    *type = UNSIGNED_PTR;
    return 0;

  } else if (strncmp(item, "delay_correction", 16) == 0) {
    *ptr = (void *)&(sce->delay_correction);
    *type = UNSIGNED_PTR;
    return 0;

  } else if (strncmp(item, "cs", 2) == 0) {
    *ptr = (void *)&(sce->cs);
    *type = UNSIGNED_PTR;
    return 0;

  } else if (strncmp(item, "eor", 3) == 0) {
    *ptr = (void *)&(sce->eor);
    *type = UNSIGNED_PTR;
    return 0;

  } else
    return 1;    /* ERROR */
}

/* Finds field position in MEDL_ISP and also the type of the field
  isp     - pointer to target MEDL_ISP tructure
  id     - field name (absolute)
  ptr    - found position in the MEDL_ISP
  type    - found item type (integer, float)
 returns 0 on success */

static int get_ptr_MEDL_ISP(MEDL_ISP *isp, void **ptr, int *type, char *id)
{
char *item, *subitems;

  subitems = get_second_part(id);
  item = get_first_part(id);

if (strncmp(item, "sync_interval", 13) == 0) {
    *ptr = (void *)&(isp->sync_interval);
    *type = UNSIGNED_PTR;
    return 0;

  } else if (strncmp(item, "listen_timeout_extension", 24) == 0) {
    *ptr = (void *)&(isp->listen_timeout_extension);
    *type = UNSIGNED_PTR;
    return 0;

  } else if (strncmp(item, "i_frame_time", 12) == 0) {
    *ptr = (void *)&(isp->i_frame_time);
    *type = UNSIGNED_PTR;
    return 0;

  } else if (strncmp(item, "listen_cold_start_period_params", 31) == 0) {
    return get_ptr_MEDL_BGP(&(isp->listen_cold_start_period_params), ptr,
type, subitems);

  } else if (strncmp(item, "oversampling_rate", 17) == 0) {
    *ptr = (void *)&(isp->oversampling_rate);
    *type = UNSIGNED_PTR;
    return 0;

  } else if (strncmp(item, "watchdog_slot_length", 20) == 0) {
    *ptr = (void *)&(isp->watchdog_slot_length);
    *type = UNSIGNED_PTR;
    return 0;

  } else if (strncmp(item, "ifg_time", 8) == 0) {
    *ptr = (void *)&(isp->ifg_time);
    *type = UNSIGNED_PTR;
    return 0;

  } else if (strncmp(item, "synchronization_period_params", 29) == 0) {
    return get_ptr_MEDL_BGP(&(isp->synchronization_period_params), ptr,
type, subitems);

  } else if (strncmp(item, "listen_timeout_restart_time", 27) == 0) {
    *ptr = (void *)&(isp->listen_timeout_restart_time);
    *type = UNSIGNED_PTR;
    return 0;

  } else if (strncmp(item, "crc", 3) == 0) {
    *ptr = (void *)&(isp->crc);
    *type = UNSIGNED_PTR;
    return 0;

  } else if (strncmp(item, "entries", 7) == 0) {
    *ptr = (void *)&(isp->entries);
    *type = UNSIGNED_PTR;
    return 0;

  } else if (strncmp(item, "listen_delay", 12) == 0) {
    *ptr = (void *)&(isp->listen_delay);
    *type = UNSIGNED_PTR;
    return 0;

  } else if (strncmp(item, "rom_speed", 9) == 0) {
    *ptr = (void *)&(isp->rom_speed);
    *type = UNSIGNED_PTR;
    return 0;

  } else
    return 1;    /* ERROR */
```

```
}

/* Finds field position in MEDL_MCE and also the type of the field
  mce      - pointer to target MEDL_MCE tructure
  id      - field name (absolute)
  ptr     - found position in the MEDL_MCE
  type     - found item type (integer, float)
 returns 0 on success */

static int get_ptr_MEDL_MCE(MEDL_MCE *mce, void **ptr, int *type, char *id)
{
char *item;

  item = get_first_part(id);

if (strncmp(item, "crc", 3) == 0) {
    *ptr = (void *)&(mce->crc);
    *type = UNSIGNED_PTR;
    return 0;

  } else if (strncmp(item, "deferred_mode", 13) == 0) {
    int i1;

    if (sscanf(item, "deferred_mode[%d]", &i1) != 1)
      return 2;    /* ERROR */
    *ptr = (void *)&(mce->deferred_mode[i1]);
    *type = UNSIGNED_PTR;
    return 0;

  } else
    return 1;    /* ERROR */
}

/* Finds field position in MEDL_RL and also the type of the field
  rl      - pointer to target MEDL_RL tructure
  id      - field name (absolute)
  ptr     - found position in the MEDL_RL
  type     - found item type (integer, float)
 returns 0 on success */

static int get_ptr_MEDL_RL(MEDL_RL *rl, void **ptr, int *type, char *id)
{
char *item, *subitems;

  subitems = get_second_part(id);
  item = get_first_part(id);

if (strncmp(item, "crc", 3) == 0) {
    *ptr = (void *)&(rl->crc);
    *type = UNSIGNED_PTR;
    return 0;

  } else if (strncmp(item, "logical_name", 12) == 0) {
    return get_ptr_LOGICAL_NAME(&(rl->logical_name), ptr, type, subitems);

  } else if (strncmp(item, "bus_guardian", 12) == 0) {
    return get_ptr_MEDL_BGP(&(rl->bus_guardian), ptr, type, subitems);
  } else
    return 1;    /* ERROR */
}

/* Finds field position in MEDL_CCP and also the type of the field
   ccp     - pointer to target MEDL_CCP tructure
   id     - field name (absolute)
   ptr    - found position in the MEDL_CCP
   type    - found item type (integer, float)
  returns 0 on success */

static int get_ptr_MEDL_CCP(MEDL_CCP *ccp, void **ptr, int *type, char *id)
{
char *item;

   item = get_first_part(id);

if (strncmp(item, "free_running_macroticks", 23) == 0) {
     *ptr = (void *)&(ccp->free_running_macroticks);
     *type = UNSIGNED_PTR;
     return 0;

   } else if (strncmp(item, "mmfc", 4) == 0) {
     *ptr = (void *)&(ccp->mmfc);
     *type = UNSIGNED_PTR;
     return 0;

   } else if (strncmp(item, "c_state_length", 14) == 0) {
     *ptr = (void *)&(ccp->c_state_length);
     *type = UNSIGNED_PTR;
     return 0;

   } else if (strncmp(item, "max_cold_start_entry", 20) == 0) {
     *ptr = (void *)&(ccp->max_cold_start_entry);
     *type = UNSIGNED_PTR;
     return 0;

   } else if (strncmp(item, "app_version", 11) == 0) {
     *ptr = (void *)&(ccp->app_version);
     *type = UNSIGNED_PTR;
     return 0;

   } else if (strncmp(item, "physical_sru_name", 17) == 0) {
     *ptr = (void *)&(ccp->physical_sru_name);
     *type = UNSIGNED_PTR;
     return 0;

   } else if (strncmp(item, "crc", 3) == 0) {
     *ptr = (void *)&(ccp->crc);
     *type = UNSIGNED_PTR;
     return 0;

   } else if (strncmp(item, "controller_version", 18) == 0) {
     *ptr = (void *)&(ccp->controller_version);
     *type = UNSIGNED_PTR;
     return 0;
   } else if (strncmp(item, "cf", 2) == 0) {
     *ptr = (void *)&(ccp->cf);
     *type = UNSIGNED_PTR;
     return 0;

   } else if (strncmp(item, "startup_timeout", 15) == 0) {
     *ptr = (void *)&(ccp->startup_timeout);
     *type = UNSIGNED_PTR;
     return 0;

   } else if (strncmp(item, "micro_to_macro", 14) == 0) {
     *ptr = (void *)&(ccp->micro_to_macro);
     *type = DOUBLE_PTR;
     return 0;

   } else if (strncmp(item, "app_id", 6) == 0) {
     *ptr = (void *)&(ccp->app_id);
     *type = UNSIGNED_PTR;
     return 0;

   } else if (strncmp(item, "tdma_round_length", 17) == 0) {
     *ptr = (void *)&(ccp->tdma_round_length);
     *type = UNSIGNED_PTR;
     return 0;

   } else
     return 1;    /* ERROR */
}
```

4.25. fp1mdlrd.h

```
/*
* This software is part of an EU project:
*     FIT - Fault Injection for TTA
*     IST-1999-10748
*
* TTP/C - protocol v.0.1
*
* Design (c) TTTech & TU Vienna
* Implementation (c) UWB Pilsen & CTU Prague
*
* fp1mdlrd.h
* Reading MEDL from a text file
*
* version 0.3 - 2001-03-13
*/


#ifndef _FP1MDLRD_H
#define _FP1MDLRD_H

#include "fp1medl.h"

/* Return codes of function read_medl_from_file().
  When an error occurs while reading a line, the line number is stored
  in an external variable called "medl_read_error_line".
  System errors store error code into external variable "errno". */
```

```
typedef enum {
  MEDL_READ_NO_ERROR = 0,
  MEDL_READ_INDEX_ERROR,         /* File index is not in range 0-99. */
  MEDL_READ_NULL_MEDL_ERROR,      /* MEDL address is NULL pointer. */
  MEDL_READ_FILE_OPEN_ERROR,     /* File containing MEDL does not exist, see
variable "errno". */
  MEDL_READ_FILE_READ_ERROR,     /* Error reading from file, see variable
"errno". */
  MEDL_READ_INVALID_ID_ERROR,      /* Line does not contain a valid field
identifier. */
  MEDL_READ_UNKNOWN_TYPE_ERROR,     /* File contains field of unknown type.
Should never happen. */
  MEDL_READ_UNKNOWN_ID_ERROR,     /* File constains unknown field
identifier. */
  MEDL_READ_BAD_FEOF_ERROR    /* Premature end of file. Should never
happen. */
} MEDL_READ_ERROR_CODES;

/* Number of the line where an error stopped execution */
extern int medl_read_error_line;

/* Reads MEDL from a text file in the current directory. File name is
derived
  from the specified index, for example: "medl01.txt", "medl12.txt".
  Read data are stored into MEDL prepared at given address.
  Field not present in the source file are set to zero.
  Return code of 0 means successfull completion, any other code is
described
  in MEDL_READ_ERROR_CODES */
extern MEDL_READ_ERROR_CODES medl_read_from_file(MEDL *medl, unsigned
index);


#endif
```

4.26. fp1medl.cpp

```
/*
* 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
*
* fp1medl.c
* Functions and definitions concerning the MEDL structure
*
* version 0.5 - 2001-03-13
*/

#include "fp1medl.h"
#include "fl1func.h"

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

/* MEDL_CCP (controller configuration parameters) field access functions -
get and set */
```

```
implement_access_functions(MEDL_CCP, medl_ccp, physical_sru_name, 8)
implement_access_functions(MEDL_CCP, medl_ccp, controller_version, 8)
implement_access_functions(MEDL_CCP, medl_ccp, app_id, 8)
implement_access_functions(MEDL_CCP, medl_ccp, app_version, 8)
implement_access_functions(MEDL_CCP, medl_ccp, cf, 1)
implement_access_functions(MEDL_CCP, medl_ccp, c_state_length, 4)
implement_access_functions(MEDL_CCP, medl_ccp, tdma_round_length, 15)
implement_access_functions(MEDL_CCP, medl_ccp, startup_timeout, 16)
implement_access_functions(MEDL_CCP, medl_ccp, free_running_macroticks, 16)
implement_access_functions(MEDL_CCP, medl_ccp, mmfc, 8)
implement_access_functions(MEDL_CCP, medl_ccp, max_cold_start_entry, 8)
implement_access_functions(MEDL_CCP, medl_ccp, crc, 16)

/* get MEDL_CCP micro_to_macro field */

double medl_ccp_get_micro_to_macro(MEDL_CCP *medl_ccp)
{
return medl_ccp->micro_to_macro;
}

/* set MEDL_CCP micro_to_macro field */

void medl_ccp_set_micro_to_macro(MEDL_CCP *medl_ccp, double value)
{
  medl_ccp->micro_to_macro = value;
}

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

/* MEDL_BGP (bus guardian parameters) field access functions - get and set
*/
implement_access_functions(MEDL_BGP, medl_bgp, bg_slot_offset, 16)
implement_access_functions(MEDL_BGP, medl_bgp, bg_slot_duration, 16)
implement_access_functions(MEDL_BGP, medl_bgp, bg_remaining_round, 16)
implement_access_functions(MEDL_BGP, medl_bgp, crc, 16)

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

/* MEDL_RL (role list) field access functions - get and set */
implement_access_functions(MEDL_RL, medl_rl, crc, 16)

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

/* MEDL_MCE (mode controle entry) field access functions - get and set */
implement_index_access_functions(MEDL_MCE, medl_mce, deferred_mode, 5)
implement_access_functions(MEDL_MCE, medl_mce, crc, 16)

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

/* MEDL_ISP (implementation specific parameters) field access functions -
get and set */
implement_access_functions(MEDL_ISP, medl_isp, entries, 10)
implement_access_functions(MEDL_ISP, medl_isp, rom_speed, 4)
implement_access_functions(MEDL_ISP, medl_isp, i_frame_time, 16)
implement_access_functions(MEDL_ISP, medl_isp, ifg_time, 16)
implement_access_functions(MEDL_ISP, medl_isp, listen_delay, 16)
implement_access_functions(MEDL_ISP, medl_isp, watchdog_slot_length, 16)
implement_access_functions(MEDL_ISP, medl_isp, listen_timeout_extension, 8)
implement_access_functions(MEDL_ISP, medl_isp, listen_timeout_restart_time,
8)
implement_access_functions(MEDL_ISP, medl_isp, oversampling_rate, 8)
implement_access_functions(MEDL_ISP, medl_isp, sync_interval, 8)
implement_access_functions(MEDL_ISP, medl_isp, crc, 16)

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

/* MEDL_SCE (slot control entry) field access functions - get and set */
implement_access_functions(MEDL_SCE, medl_sce, sru_slot_duration, 16)
implement_access_functions(MEDL_SCE, medl_sce, mo, 1)
implement_access_functions(MEDL_SCE, medl_sce, membership_pos, 6)
implement_access_functions(MEDL_SCE, medl_sce, medl_pos, 9)
implement_access_functions(MEDL_SCE, medl_sce, mm, 1)
implement_access_functions(MEDL_SCE, medl_sce, eor, 1)
implement_access_functions(MEDL_SCE, medl_sce, ra, 1)
implement_access_functions(MEDL_SCE, medl_sce, eot, 1)
implement_access_functions(MEDL_SCE, medl_sce, cs, 1)
implement_access_functions(MEDL_SCE, medl_sce, syf, 1)
implement_access_functions(MEDL_SCE, medl_sce, ba, 1)
implement_access_functions(MEDL_SCE, medl_sce, delay_correction, 6)

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

/* MEDL_MAE (mode address entry) field access functions - get and set */
implement_access_functions(MEDL_MAE, medl_mae, ru1, 1)
implement_access_functions(MEDL_MAE, medl_mae, ia, 1)
implement_access_functions(MEDL_MAE, medl_mae, cni_address_cha, 14)
implement_access_functions(MEDL_MAE, medl_mae, ru2, 1)
implement_access_functions(MEDL_MAE, medl_mae, ib, 1)
implement_access_functions(MEDL_MAE, medl_mae, cni_address_chb, 14)
implement_index_access_functions(MEDL_MAE, medl_mae, mcp_i, 1)
implement_index_access_functions(MEDL_MAE, medl_mae, mcp_d, 1)
implement_access_functions(MEDL_MAE, medl_mae, mcp_c, 1)
implement_access_functions(MEDL_MAE, medl_mae, eoc, 1)
implement_access_functions(MEDL_MAE, medl_mae, dafla, 4)
implement_access_functions(MEDL_MAE, medl_mae, daflb, 4)
implement_access_functions(MEDL_MAE, medl_mae, crc, 16)

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

/* MEDL (message descriptor list) field access functions - get and set */
implement_access_functions(MEDL, medl, revision, 16)

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

/* other MEDL functions */

/* checks all CRC code in the MEDL and return 1 if ok */

int medl_validity_check(MEDL *medl)
{
  /* TODO: CRC check */
return (medl != NULL);
}
```

4.27. fp1medl.h

```
/*
* This software is part of an EU project:
*     FIT - Fault Injection for TTA
*     IST-1999-10748
*
* TTP/C - protocol v.0.1
*
```

```
* Design (c) TTTech & TU Vienna
* Implementation (c) UWB Pilsen & CTU Prague
*
* fp1medl.h
* Functions and definitions concerning the MEDL structure
*
* version 0.5 - 2001-03-13
*/

#ifndef _FP1MEDL_H
#define _FP1MEDL_H

#include "fl1func.h"
#include "fp1const.h"
#include "fp1lname.h"

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

/* MEDL - cotroller configuration parameters structure */
typedef struct {
unsigned physical_sru_name;
unsigned controller_version;
unsigned app_id;
unsigned app_version;
unsigned cf;               /* cold start flag */
unsigned c_state_length;
unsigned tdma_round_length;
unsigned startup_timeout;
unsigned free_running_macroticks;
unsigned mmfc;                /* max. membership failure count */
double micro_to_macro;        /* microticks/macrotick integral +
microticks/macrotick fraction */
unsigned max_cold_start_entry;
unsigned crc;
} MEDL_CCP;

/* MEDL_CCP field access functions - get and set */
declare_access_functions(MEDL_CCP, medl_ccp, physical_sru_name);
declare_access_functions(MEDL_CCP, medl_ccp, controller_version);
declare_access_functions(MEDL_CCP, medl_ccp, app_id);
declare_access_functions(MEDL_CCP, medl_ccp, app_version);
declare_access_functions(MEDL_CCP, medl_ccp, cf);
declare_access_functions(MEDL_CCP, medl_ccp, c_state_length);
declare_access_functions(MEDL_CCP, medl_ccp, tdma_round_length);
declare_access_functions(MEDL_CCP, medl_ccp, startup_timeout);
declare_access_functions(MEDL_CCP, medl_ccp, free_running_macroticks);
declare_access_functions(MEDL_CCP, medl_ccp, mmfc);
declare_access_functions(MEDL_CCP, medl_ccp, max_cold_start_entry);
declare_access_functions(MEDL_CCP, medl_ccp, crc);

/* get MEDL_CCP micro_to_macro field */
double medl_ccp_get_micro_to_macro(MEDL_CCP *medl_ccp);
/* set MEDL_CCP micro_to_macro field */
void medl_ccp_set_micro_to_macro(MEDL_CCP *medl_ccp, double value);

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

/* MEDL - bus guardian configuration parameters structure */
typedef struct {
unsigned bg_slot_offset;
unsigned bg_slot_duration;
unsigned bg_remaining_round;
unsigned crc;
} MEDL_BGP;

/* MEDL_BGP field access functions - get and set */
declare_access_functions(MEDL_BGP, medl_bgp, bg_slot_offset);
declare_access_functions(MEDL_BGP, medl_bgp, bg_slot_duration);
declare_access_functions(MEDL_BGP, medl_bgp, bg_remaining_round);
declare_access_functions(MEDL_BGP, medl_bgp, crc);

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

/* MEDL - role list structure */
typedef struct {
  LOGICAL_NAME logical_name;
  MEDL_BGP bus_guardian;
unsigned crc;
} MEDL_RL;

/* MEDL_RL field access functions - get and set */
declare_access_functions(MEDL_RL, medl_rl, crc);

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

/* MEDL - mode control entry structure */
typedef struct {
unsigned deferred_mode[3];
unsigned crc;
} MEDL_MCE;

/* MEDL_MCE field access functions - get and set */
declare_index_access_functions(MEDL_MCE, medl_mce, deffered_mode);
declare_access_functions(MEDL_MCE, medl_mce, crc);

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

/* MEDL - implementation specific parameters structure */
typedef struct {
unsigned entries;
unsigned rom_speed;
unsigned i_frame_time;
unsigned ifg_time;
unsigned listen_delay;
unsigned watchdog_slot_length;
unsigned listen_timeout_extension;
unsigned listen_timeout_restart_time;
unsigned oversampling_rate;
unsigned sync_interval;
unsigned crc;
  MEDL_BGP synchronization_period_params;
  MEDL_BGP listen_cold_start_period_params;
} MEDL_ISP;

/* MEDL_ISP field access functions - get and set */
declare_access_functions(MEDL_ISP, medl_isp, entries);
declare_access_functions(MEDL_ISP, medl_isp, rom_speed);
declare_access_functions(MEDL_ISP, medl_isp, i_frame_time);
declare_access_functions(MEDL_ISP, medl_isp, ifg_time);
declare_access_functions(MEDL_ISP, medl_isp, listen_delay);
declare_access_functions(MEDL_ISP, medl_isp, watchdog_slot_length);
declare_access_functions(MEDL_ISP, medl_isp, listen_timeout_extension);
declare_access_functions(MEDL_ISP, medl_isp, listen_timeout_restart_time);
declare_access_functions(MEDL_ISP, medl_isp, oversampling_rate);
declare_access_functions(MEDL_ISP, medl_isp, sync_interval);
declare_access_functions(MEDL_ISP, medl_isp, crc);

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

/* MEDL - slot control entry structure */
typedef struct {
unsigned sru_slot_duration;
unsigned mo;        /* multiplexed SRU with own membership */
unsigned membership_pos;
unsigned medl_pos;
unsigned mm;        /* multiplexed SRU with multiplexed membership */
unsigned eor;        /* end of round */
unsigned ra;        /* reintegration allowed */
unsigned eot;        /* end of transmission cycle */
unsigned cs;        /* clock synchronization */
unsigned syf;        /* synchronization frame */
unsigned ba;        /* BG arm */
unsigned delay_correction;
  LOGICAL_NAME logical_name;
} MEDL_SCE;

/* MEDL_SCE field access functions - get and set */
declare_access_functions(MEDL_SCE, medl_sce, sru_slot_duration);
declare_access_functions(MEDL_SCE, medl_sce, mo);
declare_access_functions(MEDL_SCE, medl_sce, membership_pos);
declare_access_functions(MEDL_SCE, medl_sce, medl_pos);
declare_access_functions(MEDL_SCE, medl_sce, mm);
declare_access_functions(MEDL_SCE, medl_sce, eor);
declare_access_functions(MEDL_SCE, medl_sce, ra);
declare_access_functions(MEDL_SCE, medl_sce, eot);
declare_access_functions(MEDL_SCE, medl_sce, cs);
declare_access_functions(MEDL_SCE, medl_sce, syf);
declare_access_functions(MEDL_SCE, medl_sce, ba);
declare_access_functions(MEDL_SCE, medl_sce, delay_correction);

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

/* MEDL - mode address entry structure */
typedef struct {
unsigned ru1;        /* raise user interrupt 1 */
unsigned ia;        /* i-frame on channel A */
unsigned cni_address_cha;
unsigned ru2;        /* raise user interrupt 2 */
unsigned ib;        /* i-frame on channel B */
unsigned cni_address_chb;

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

unsigned eoc;        /* end of cluster cycle */
unsigned dafla;        /* data field length on channel A */
unsigned daflb;        /* data field length on channel B */
unsigned crc;
} MEDL_MAE;
```

```
/* MEDL_MAE field access functions - get and set */
declare_access_functions(MEDL_MAE, medl_mae, ru1);
declare_access_functions(MEDL_MAE, medl_mae, ia);
declare_access_functions(MEDL_MAE, medl_mae, cni_address_cha);
declare_access_functions(MEDL_MAE, medl_mae, ru2);
declare_access_functions(MEDL_MAE, medl_mae, ib);
declare_access_functions(MEDL_MAE, medl_mae, cni_address_chb);
declare_index_access_functions(MEDL_MAE, medl_mae, mcp_i);
declare_index_access_functions(MEDL_MAE, medl_mae, mcp_d);
declare_access_functions(MEDL_MAE, medl_mae, mcp_c);
declare_access_functions(MEDL_MAE, medl_mae, eoc);
declare_access_functions(MEDL_MAE, medl_mae, dafla);
declare_access_functions(MEDL_MAE, medl_mae, daflb);
declare_access_functions(MEDL_MAE, medl_mae, crc);

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

/* MEDL - message descriptor list structure */
typedef struct {
unsigned revision;

  /* MEDL - configuration parameters */
  MEDL_CCP controller_cfg_params;
  MEDL_BGP bus_guardian_cold_start;
  MEDL_BGP bus_guardian_download;
  MEDL_RL role_list;
  MEDL_MCE mode_control_entry[MAX_MODE_COUNT];
  MEDL_ISP implementation_params;

  /* MEDL - trasmission block */
  MEDL_SCE slot_control_entry[MAX_TRANSMISSION_CYCLE_LENGTH];
  MEDL_MAE mode_address_entry[MAX_MODE_COUNT +
1][MAX_TRANSMISSION_CYCLE_LENGTH];
} MEDL;

/* MEDL field access functions - get and set */
declare_access_functions(MEDL, medl, revision);


/* other MEDL functions */

/* checks all CRC code in the MEDL and return 1 if ok */
int medl_validity_check(MEDL *medl);

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

#endif /* #ifndef _FP1MEDL_H */
```

4.28. sa1fault.cpp

```
/*
* 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
*
* sa1failt.c
* FAULT_INJECTOR C-sim process
*
* version 0.5 - 2001-06-07
*/

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

#include "sa1fault.h"

#include <math.h>
#include <string.h>
#include "sp1ctrl.h"
#include "sl1func.h"
#include "kr_csim.h"
#include "fp1cni.h"


/**************************************************************************
***********/
static void derive_fault_index(FAULT_INDEX *fault_index); static void
fault_injector_destructor(FAULT_INJECTOR *p_my); static int
is_enabled(FAULT_INJECTOR *fi);

/* TODO: destructor not assigned */

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

/* returns 0 if no mem_area is enabled or lambda parameter is zero */
static int is_enabled(FAULT_INJECTOR *fi)
{
int ret = 0;
unsigned i;

if (fi->lambda <= 0.0)
    return 0;

for (i = 0; !ret && i < fi->area_count; i++)
    ret = fi->area_enabled[i];

return ret;
}


/* Fills index and bit_count fields from the fault_mask */

static void derive_fault_index(FAULT_INDEX *fault_index)
{
unsigned *mask = fault_index->mask;
unsigned *index = fault_index->index;
unsigned i, x, count;

  count = 0;
for (i = 0; i < fault_index->size; i++) {
    for (x = mask[i]; x > 0; x >>= 1)
      count += x & 1;
    index[i] = count;
  }
  fault_index->bits = count;
}

/* FAULT_INJECTOR process - program code */

static s_program(FAULT_INJECTOR, fault_injector_prog)
for (;;) {
    if (is_enabled(p_my)) {
      fault_injector_inject_fault(p_my);
      hold(negexp(p_my->lambda / 1000000.0));
    } else {
      passivate();
    }
  }
s_end_program


/* Frees allocated dynamic memory */

static begin_destructor(FAULT_INJECTOR, fault_injector_destructor)
if (p_my->fault_index.index != NULL)
    free(p_my->fault_index.index);
if (p_my->area_ptr != NULL)
    free(p_my->area_ptr);
if (p_my->area_enabled != NULL)
    free(p_my->area_enabled);
end_destructor

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

/* Global functions. */

/* creates new FAULT_INJECTOR process and initializes atributtes */
FAULT_INJECTOR *fault_injector_create(char *name)
{
  FAULT_INJECTOR *fi;

  fi = s_new_process(FAULT_INJECTOR, fault_injector_prog);
  set_name(fi, name);
  set_destructor((PROCESS *)fi, fault_injector_destructor);
  fi->area_count = 0;
  fi->area_enabled = NULL;
  fi->fault_index.size = 0;
  fi->fault_index.index = NULL;

  fi->fault_type = FT_FLIP;
  fi->lambda = 0.0;
  fi->fault_count = 0;

return fi;
}

/* Reschedule the FAULT_INJECTOR process to the new lambda parameter. */
void fault_injector_reschedule(FAULT_INJECTOR *fi, double lambda)
{
  fi->lambda = lambda;

if (is_enabled(fi))
```

```
    process_activate_delay((PROCESS *) fi, negexp(lambda / 1000000));
}

/* Inject fault of chosen type into specified CNI (or CNIs) */

void fault_injector_inject_fault(FAULT_INJECTOR *fi)
{
unsigned a, b, bit;
unsigned x, i;

if (fi->fault_index.bits > 0) {
    fi->fault_count++;

    a = 0;
    b = fi->fault_index.size - 1;
    bit = (int)(uniform(1, fi->fault_index.bits) + 0.5);
//        print_debug((fi, ">> Injecting random fault at bit %d!\n", bit));

    while (a < b) {
      if (fi->fault_index.index[(a + b) / 2] >= bit)
        b = (a + b) / 2;
      else
        a = (a + b) / 2 + 1;
    }

    bit -= (a != 0) ? fi->fault_index.index[a - 1] : 0;

//        print_debug((fi, ">> Addr = %i, bit = %i\n", a, bit));

    x = 1;
    do {
      if ((fi->fault_index.mask[a] & x) != 0)
        bit--;
      if (bit > 0)
        x <<= 1;
      assert(x != 0);
    } while (bit > 0);

    for (i = 0; i < fi->area_count; i++) {
      if (fi->area_enabled[i])
        switch (fi->fault_type) {
          case FT_FLIP: fi->area_ptr[i][a] ^= x;
            break;
          case FT_ZERO: fi->area_ptr[i][a] &= ~x;
            break;
          case FT_ONE: fi->area_ptr[i][a] |= x;
            break;
        }
    }
   }
}

/* Sets a new fault mask - 'size' if the byte size of the memory area,
'ptr' points to mask. */

void fault_injector_set_mask(FAULT_INJECTOR *fi, unsigned size, unsigned
*ptr)
{
```

```
if (fi->fault_index.index != NULL)
    free(fi->fault_index.index);
  fi->fault_index.index = (unsigned *)malloc(size);
  fi->fault_index.size = size / sizeof(unsigned);
  fi->fault_index.mask = ptr;

  derive_fault_index(&fi->fault_index);
}

/* Drops all memory areas previously assigned to the fault injector. */
void fault_injector_clear_areas(FAULT_INJECTOR *fi)
{
if (fi->area_ptr != NULL)
    free(fi->area_ptr);
if (fi->area_enabled != NULL)
    free(fi->area_enabled);

  fi->area_ptr = NULL;
  fi->area_enabled = NULL;
  fi->area_count = 0;
}

/* Selects a memory area for fault injection.
  The application should first use the _set_mask() function to specify the
area size and fault mask.
  Return the index of the new memory area, -1 if an error occurs. */

int fault_injector_add_mem_area(FAULT_INJECTOR *fi, unsigned *ptr)
{
unsigned **ptr_buf;
int *bool_buf;

  ptr_buf = (unsigned **)malloc((fi->area_count + 1) * sizeof(unsigned *));
  bool_buf = (int *)malloc((fi->area_count + 1) * sizeof(int));
if (fi->area_ptr != NULL) {
    memcpy(ptr_buf, fi->area_ptr, fi->area_count * sizeof(unsigned *));
    memcpy(bool_buf, fi->area_enabled, fi->area_count * sizeof(int));
    free(fi->area_ptr);
    free(fi->area_enabled);
  }
  fi->area_ptr = ptr_buf;
  fi->area_enabled = bool_buf;
  fi->area_ptr[fi->area_count] = ptr;
  fi->area_enabled[fi->area_count] = 0;

return fi->area_count++;
}

/* Enables or disables fault injection into specified memory area. */

void fault_injector_enable_area(FAULT_INJECTOR *fi, unsigned index, int
enable)
{
  assert(index < fi->area_count);

  fi->area_enabled[index] = enable;
}
```

```
/* Return if the specified area is has fault injection enabled. */

int fault_injector_area_enabled(FAULT_INJECTOR *fi, unsigned index)
{
  assert(index < fi->area_count);

return fi->area_enabled[index];
}
```

4.29. sa1fault.h

```
/*
* This software is part of an EU project:
*     FIT - Fault Injection for TTA
*     IST-1999-10748
*
* TTP/C - protocol v.0.1
*
* Design (c) TTTech & TU Vienna
* Implementation (c) UWB Pilsen & CTU Prague
*
* sa1fault.h
* Process for fault injection into deliberate memory area
*
* version 0.5 - 2001-06-07
*/

#ifndef _SA1FAULT_H
#define _SA1FAULT_H

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

/**************************************************************************
***********/
typedef enum {
  FT_ONE,
  FT_ZERO,
  FT_FLIP
} FAULT_TYPES;

/* Structure that enables bit faults in the memory and optimizes access */
typedef struct {
unsigned size;            /* size of fault injection area in integers */
unsigned *mask;            /* bit mask where 1 = fault enabled */
unsigned *index;        /* index[i] = number of bit faults enabled until i-
th integer */
unsigned bits;            /* total number of bit faults enabled (value of
last index field) */
} FAULT_INDEX;

/* FAULT_INJECTOR process data structure */
derived_process(fault_injector)
unsigned area_count;    /* Number of areas designated for fault injection
*/
unsigned **area_ptr;    /* Pointers to fault injected memory areas */
int *area_enabled;      /* if cleared the respective area is NOT subjected
to fault injection */
```

```
double lambda;            /* Fault intensity (mean faults per sec) */
  FAULT_TYPES fault_type; /* One of three fault types */
  FAULT_INDEX fault_index;

unsigned fault_count;    /* number of injected faults */
end_derived(FAULT_INJECTOR);

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

/* Creates new FAULT_INJECTOR process and initializes its atributtes.
  Returns pointer the new process. */
FAULT_INJECTOR *fault_injector_create(char *name);

/* Reshedules the FAULT_INJECTOR process when lambda parameter changes */
void fault_injector_reschedule(FAULT_INJECTOR *fi, double lambda);

/* Inject fault of chosen type into the selected memory areas. */
void fault_injector_inject_fault(FAULT_INJECTOR *fi);

/* Sets a new fault mask - 'size' if the byte size of the memory area,
'ptr' points to mask. */
void fault_injector_set_mask(FAULT_INJECTOR *fi, unsigned size, unsigned
*ptr);

/* Drops all memory areas previously assigned to the fault injector. */
void fault_injector_clear_areas(FAULT_INJECTOR *fi);

/* Selects a memory area for fault injection.
  The application should first use the _set_mask() function to specify the
area size and fault mask.
  Return the index of the new memory area, -1 if an error occurs. */
int fault_injector_add_mem_area(FAULT_INJECTOR *fi, unsigned *ptr);

/* Enables or disables fault injection into specified memory area. */
void fault_injector_enable_area(FAULT_INJECTOR *fi, unsigned index, int
enable);

/* Return if the specified area is has fault injection enabled. */
int fault_injector_area_enabled(FAULT_INJECTOR *fi, unsigned index);

#endif
```

4.30. sl1func.cpp

```
/*
* 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
*
* sl1func.cpp
* Global functions and macro definitions for the simulation environment
*
* version 0.2 - 2001-06-07
*/
```

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

#include "sl1func.h"

#include "kr_csim.h"

/* Returns whether the process is planned in the scheduler. */

int process_is_planned(PROCESS *process)
{
  PROC_STATE pstate;

  assert(process != NULL);
  pstate = state(process);
return (pstate == PLANNED || pstate == NEW_PLANNED);
}

/* Returns whether the process is planned in the scheduler. */

int process_is_active(PROCESS *process)
{
  assert(process != NULL);
//    return (state(process) == ACTIVE);
return (process == current());
}

/* Returns whether the process is terminated. */

int process_is_terminated(PROCESS *process)
{
  assert(process != NULL);
return (state(process) == TERMINATED);
}

/* If the process is planned, then it is removed from scheduler.
  Returns 0 on success, 1 if the process is currently active. */

int process_cancel(PROCESS *process)
{
  assert(process != NULL);

if (process_is_active(process))
    return 1;

if (process_is_planned(process))
    cancel(process);

return 0;
}

/* Schedules the process to specified time. If there was an earlier
schedule it is rewriten.
  Returns '0' on success, '1' if process is ACTIVE and '2' if TERMINATED.
*/

int process_activate_at(PROCESS *process, double time)
{
  assert(process != NULL);

if (process_is_active(process))
    return 1;
if (process_is_terminated(process))
    return 2;

  reactivate_at(process, time);
return 0;
}

/* Schedules the process to specified delta time. If there was an earlier
schedule it is rewriten.
  Returns '0' on success, '1' if process is ACTIVE and '2' if TERMINATED.
*/

int process_activate_delay(PROCESS *process, double time)
{
  assert(process != NULL);

if (process_is_active(process))
    return 1;
if (process_is_terminated(process))
    return 2;

  reactivate_delay(process, time);
return 0;
}
```

4.31. sl1func.h

```
/*
* This software is part of an EU project:
*     FIT - Fault Injection for TTA
*     IST-1999-10748
*
* TTP/C - protocol v.0.1
*
* Design (c) TTTech & TU Vienna
* Implementation (c) UWB Pilsen & CTU Prague
*
* sl1func.cpp
* Global functions and macro definitions for the simulation environment
*
* version 0.1 - 2001-04-10
*/

#ifndef _SL1FUNC_H
#define _SL1FUNC_H

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

#include "kr_csim.h"

/* Returns whether the process is planned in the scheduler. */
int process_is_planned(PROCESS *process);

/* Returns whether the process is planned in the scheduler. */
```

```
int process_is_active(PROCESS *process);

/* Returns whether the process is terminated. */
int process_is_terminated(PROCESS *process);

/* If the process is planned, then it is removed from scheduler.
  Returns 0 on success, 1 if the process is currently active. */
int process_cancel(PROCESS *process);

/* Schedules the process to specified time. If there was an earlier
schedule it is rewriten.
  Returns '0' on success, '1' if process is ACTIVE and '2' if TERMINATED.
*/
int process_activate_at(PROCESS *process, double time);

/* Schedules the process to specified delta time. If there was an earlier
schedule it is rewriten.
  Returns '0' on success, '1' if process is ACTIVE and '2' if TERMINATED.
*/
int process_activate_delay(PROCESS *process, double time);


#endif /* #ifndef SL1FUNC_H */
```

4.32. sp1busg.cpp

```
/*
* 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
*
* sp1busg.c
* Functions and definitions concerning the BUS_GUARDIAN c-sim proces
*
* version 0.7 - 2001-04-10
*/

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

#include "sp1busg.h"

#include "fp1busg.h"
#include "kr_csim.h"
#include "sl1func.h"


inline static double bg_macrotick_to_csim(BUS_GUARDIAN_MODEL *bg, unsigned
bg_time)
{
return BUS_GUARDIAN_CLK_PER_SEC * bg_time * (1.0 + bg->clock_drift) /
1000000.0;
}

#define bg_hold(bg_model, mt) do { \
  print_debug((bg_model->process, "Holding: %0.4f\n", \
    bg_macrotick_to_csim(bg_model, mt))); \
  hold(bg_macrotick_to_csim(bg_model, mt)); \
} while(0)

/* BUS_GUARDIAN process - program code */
#define model (p_my->model)
#define protocol (p_my->protocol) static s_program(BUS_GUARDIAN_PROCESS,
prog_bus_guardian)
for (;;) {
    switch (protocol->bg_state) {
      case BGS_INITIALIZE:
        /* set default configuration */
        bus_guardian_init_state(protocol);
        break;
      case BGS_CONFIGURE:
        bus_guardian_configure_state(protocol);
        passivate();
        break;
      case BGS_ARM:
        bus_guardian_arm_state(protocol);
        if (protocol->bg_state == BGS_ARM)
          passivate();
        break;
      case BGS_START_TIMER:
        if (bus_guardian_get_tso(protocol) > 0) {
          bg_hold(model, bus_guardian_get_tso(protocol));
          if (protocol->bg_state != BGS_START_TIMER)
            break;
        }
        bus_guardian_start_timer_state(protocol);
        break;
      case BGS_ENABLE:
        print_debug((p_my, "Bus access enabled.\n"));
        if (bus_guardian_get_tsd(protocol) > 0) {
          bg_hold(model, bus_guardian_get_tsd(protocol));
          if (protocol->bg_state != BGS_ENABLE)
            break;
        }
        bus_guardian_enable_state(protocol);
        break;
      case BGS_DISABLE:
        print_debug((p_my, "Bus access disabled.\n"));
        if (bus_guardian_get_trr(protocol) > 0) {
          bg_hold(model, bus_guardian_get_trr(protocol));
          if (protocol->bg_state != BGS_DISABLE)
            break;
        }
        bus_guardian_disable_state(protocol);
        break;
      case BGS_NODE_FAILURE:
        print_debug((p_my, "Node Failure.\n"));
        passivate();
    }
  }
s_end_program
#undef model
```

```
#undef protocol
static begin_destructor(BUS_GUARDIAN_PROCESS, bus_guardian_destroy)
  free(p_my->model);
end_destructor

/* set BUS_GUARDIAN arm bit, check for errors */

static void bus_guardian_arm_signal(BUS_GUARDIAN *bus_guardian, unsigned
value)
{
  BUS_GUARDIAN_MODEL *bus_guardian_model = (BUS_GUARDIAN_MODEL
*)bus_guardian;
unsigned old_value = bus_guardian_get_am(bus_guardian);

  value &= 1;
  bus_guardian_set_am(bus_guardian, value);

if (value == 1 && old_value != 1 && bus_guardian->bg_state == BGS_ARM) {
    print_debug((bus_guardian_model->process, "Arm. Open scheduled at
%10.4f.\n",
      bg_macrotick_to_csim(bus_guardian_model,
bus_guardian_get_tso(bus_guardian)) + CSIM_TIME()
    ));
    process_activate_delay((PROCESS *)bus_guardian_model->process, 0);
  }
}

/* create and initialize BUS_GUARDIAN process */
BUS_GUARDIAN_MODEL *bus_guardian_create(char *name)
{
  BUS_GUARDIAN_MODEL *bus_guardian_model;

  bus_guardian_model = (BUS_GUARDIAN_MODEL
*)malloc(sizeof(BUS_GUARDIAN_MODEL));

  bus_guardian_model->arm_signal = bus_guardian_arm_signal;

  bus_guardian_model->process = s_new_process(BUS_GUARDIAN_PROCESS,
prog_bus_guardian);
  bus_guardian_model->process->protocol = (BUS_GUARDIAN
*)bus_guardian_model;
  bus_guardian_model->process->model = bus_guardian_model;
  set_name(bus_guardian_model->process, name);
  set_destructor(bus_guardian_model->process, bus_guardian_destroy);

  bus_guardian_set_clock_drift(bus_guardian_model, 0);
  bus_guardian_initialize((BUS_GUARDIAN *)bus_guardian_model);

return bus_guardian_model;
}

/* attach controller to the bus_guardian */

void bus_guardian_attach_controller(BUS_GUARDIAN_MODEL* bus_guardian_model,
void *controller)
{
  bus_guardian_model->controller = controller;
}
```

```
/* Sets the clock drift [sec/sec] for the bus guardian. */

void bus_guardian_set_clock_drift(BUS_GUARDIAN_MODEL *bus_guardian_model,
double new_drift)
{
  bus_guardian_model->clock_drift = new_drift;
}
```

4.33. sp1busg.h

```
/*
* This software is part of an EU project:
*     FIT - Fault Injection for TTA
*     IST-1999-10748
*
* TTP/C - protocol v.0.1
*
* Design (c) TTTech & TU Vienna
* Implementation (c) UWB Pilsen & CTU Prague
*
* sp1busg.h
* Functions and definitions concerning the BUS_GUARDIAN c-sim proces
*
* version 0.6 - 2001-03-13
*/

#ifndef _SP1BUSG_H
#define _SP1BUSG_H

#include "kr_csim.h"
#include "fp1busg.h"
#include "fl1func.h"
struct bus_guardian_model;

derived_process(bus_guardian_process)
  BUS_GUARDIAN *protocol;
struct bus_guardian_model *model;
end_derived(BUS_GUARDIAN_PROCESS);
typedef struct bus_guardian_model {
  d_bus_guardian;
  BUS_GUARDIAN_PROCESS *process;
double clock_drift;     /* Internal clock drift [sec / sec] */
} BUS_GUARDIAN_MODEL;


/* create and initialize BUS_GUARDIAN process */
BUS_GUARDIAN_MODEL *bus_guardian_create(char *name);

/* attach controller to the bus_guardian */
void bus_guardian_attach_controller(BUS_GUARDIAN_MODEL *bus_guardian_model,
void *controller);

/* Sets the clock drift [sec/sec] for the bus guardian. */
void bus_guardian_set_clock_drift(BUS_GUARDIAN_MODEL *bus_guardian_model,
double new_drift);
```

```
#endif /* #ifndef _SP1BUSG_H */
```

4.34. sp1cni.cpp

```
/*
* 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
*
* sp1cni.c
* Functions and definitions concerning the CNI structure
*
* version 0.11 - 2001-03-13
*/

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

#include "sp1cni.h"

#include "fp1cni.h"
#include "fl1func.h"
#include "sp1ctrl.h"

/* Calculates (and stores) the local cluster time field according to actual
model time. */

static unsigned static_cni_get_cluster_time(CNI *cni)
{
  ((CONTROLLER *)cni->controller)->update_cluster_time((CONTROLLER *)cni-
>controller);
return     cni->cluster_time;
}


/* set the CO field in the CNI, possibly wake the controller */

static unsigned static_cni_set_co(CNI *cni, unsigned value)
{
if (cni->co == value)
    return value;
  cni->co = value & bit_mask(16);
  controller_co_changed((CONTROLLER *)cni->controller);

return cni->co;
}

/* set the CA field in the CNI, possibly wake the controller */

static unsigned static_cni_set_ca(CNI *cni, unsigned value)
{
if (cni->ca == value)
    return value;
  cni->ca = value & bit_mask(16);
  controller_ca_changed((CONTROLLER *)cni->controller);

return cni->ca;
}
static unsigned static_cni_set_timer1(CNI *cni, unsigned value)
{
if (cni->timer1 == value)
    return value;
  cni->timer1 = value & bit_mask(16);
  controller_timer_changed((CONTROLLER_MODEL *)(cni->controller), 0);

return cni->timer1;
}
static unsigned static_cni_set_timer2(CNI *cni, unsigned value)
{
if (cni->timer2 == value)
    return value;
  cni->timer2 = value & bit_mask(16);
  controller_timer_changed((CONTROLLER_MODEL *)(cni->controller), 1);

return cni->timer2;
}

/* Fixme: ERROR (need controller pointer in CNI_IE */

static unsigned static_cni_ie_set_ti1(CNI *cni, unsigned value)
{
if (cni->interrupt_enable.ti1 == value)
    return value;
  cni->interrupt_enable.ti1 = value & bit_mask(1);
  controller_timer_changed((CONTROLLER_MODEL *)(cni->controller), 0);

return cni->interrupt_enable.ti1;
}

/* Fixme: ERROR (need controller pointer in CNI_IE */

static unsigned static_cni_ie_set_ti2(CNI *cni, unsigned value)
{
if (cni->interrupt_enable.ti2 == value)
    return value;
  cni->interrupt_enable.ti2 = value & bit_mask(1);
  controller_timer_changed((CONTROLLER_MODEL *)(cni->controller), 1);

return cni->interrupt_enable.ti2;
}

void cni_module_initialize()
{
  cni_get_cluster_time = static_cni_get_cluster_time;
  cni_set_co = static_cni_set_co;
  cni_set_ca = static_cni_set_ca;
  cni_set_timer1 = static_cni_set_timer1;
  cni_set_timer2 = static_cni_set_timer2;
  cni_set_timer1 = static_cni_set_timer1;
  cni_set_timer2 = static_cni_set_timer2;
  cni_ie_set_ti1 = static_cni_ie_set_ti1;
  cni_ie_set_ti2 = static_cni_ie_set_ti2;
}
```

4.35. sp1cni.h

```
/*
* This software is part of an EU project:
*     FIT - Fault Injection for TTA
*     IST-1999-10748
*
* TTP/C - protocol v.0.1
*
* Design (c) TTTech & TU Vienna
* Implementation (c) UWB Pilsen & CTU Prague
*
* sp1cni.h
* Functions and definitions concerning the CNI structure
*
* version 0.10 - 2001-03-13
*/

#ifndef _SP1CNI_H
#define _SP1CNI_H

#include "fl1func.h"
#include "fp1cni.h"
void cni_module_initialize();

#endif /* #ifndef _SP1CNI_H */
```

4.36. sp1ctrl.cpp

```
/*
* 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
*
* sp1ctrl.c
* Functions and definitions concerning the controller c-sim process
*
* version 0.7 - 2001-04-10
*/

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

#include "sp1ctrl.h"

#include <math.h>
#include <string.h>
#include "kr_csim.h"
#include "fp1const.h"
#include "fp1ctrl.h"
#include "fp1cni.h"
#include "fp1medl.h"
```

```
#include "fp1frame.h"
#include "sp1cni.h"
#include "sp1busg.h"
#include "sl1func.h"

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

/* Time to wait after receive start when garbage is received */
#define CONTROLLER_TIME_TO_RECOGNIZE_BAD_RECEIVE   100


#define controller_hold(controller, mt) do { \
  print_debug((controller->process, "Holding: ")); \
  hold(get_hold_time(controller, mt)); \
} while (0)


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

/* Initializes a Controller Model structure. Returns pointer to the
structure. */

static CONTROLLER_MODEL *controller_model_init(CONTROLLER_MODEL
*controller_model); static void model_receive_callback(void *controller,
unsigned channel); static double controller_micro_to_csim(CONTROLLER_MODEL
*controller, double micro); static double get_hold_time(CONTROLLER_MODEL
*controller, unsigned macroticks);

/**********************************************************************
***/
static void attach_timer_process(CONTROLLER_MODEL *controller, unsigned
index, TIMER_PROCESS *timer)
{
  controller->timer[index] = timer;
  timer->controller = (void *)controller;
  timer->timer = index;
}

static s_program(TIMER_PROCESS, prog_timer)
for (;;) {
    print_debug((p_my, "Timer activated\n"));
    controller_raise_timer_interrupt((CONTROLLER *)p_my->controller, p_my-
>timer);
    passivate();
  }
s_end_program
static TIMER_PROCESS *timer_process_create(char *name)
{
  TIMER_PROCESS *timer;

  timer = s_new_process(TIMER_PROCESS, prog_timer);
  set_name((PROCESS *)timer, name);

return timer;
}

/* Shedule the timer process or cancel its shedule if timer is disabled. */
```

```
void controller_timer_changed(CONTROLLER_MODEL *controller, int timer)
{
  CNI *cni = &controller->cni;
unsigned mt;

if (timer == 0) {
    mt = (cni_get_timer1(cni) - cni_get_cluster_time(cni)) & bit_mask(16);
    if (mt > 0 && cni_ie_get_ti1(&cni->interrupt_enable))
      process_activate_delay((PROCESS *)controller->timer[0],
        get_hold_time(controller, mt) + controller_micro_to_csim(controller,
2.0));
    else
      process_cancel((PROCESS *)controller->timer[0]);
  }
if (timer == 1) {
    mt = (cni_get_timer2(cni) - cni_get_cluster_time(cni)) & bit_mask(16);
    if (mt > 0 && cni_ie_get_ti2(&cni->interrupt_enable))
      process_activate_delay((PROCESS *)controller->timer[1],
        get_hold_time(controller, mt) + controller_micro_to_csim(controller,
2.0));
    else
      process_cancel((PROCESS *)controller->timer[1]);
  }
}


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


/* return the duration of n microticks of the controller in CSIM time */
static double controller_micro_to_csim(CONTROLLER_MODEL *controller, double
micro)
{
double x, p;

  p = controller->ldata.clock_drift / (1 + controller->ldata.clock_drift);

  x = micro / medl_ccp_get_micro_to_macro(&controller-
>medl.controller_cfg_params)
    * (double)MACROTICK_DURATION;
return x - x * p;
}

/* return the number of microticks equal to time in CSIM */

static double controller_csim_to_micro(CONTROLLER_MODEL *controller, double
time)
{
double x;

  x = time / (double)MACROTICK_DURATION
    * medl_ccp_get_micro_to_macro(&controller->medl.controller_cfg_params);
return x + x * controller->ldata.clock_drift;
}

/* return the microtick duration of the controller in microseconds */
static double controller_macro_to_csim(CONTROLLER_MODEL *controller, double
macro)
```

```
{
double x, p;

  p = controller->ldata.clock_drift / (1 + controller->ldata.clock_drift);

  x = macro * (double)MACROTICK_DURATION;
return x - x * p;
}

/* returns macroticks equal to CSIM duration */

static double controller_csim_to_macro(CONTROLLER_MODEL *controller, double
time)
{
double x;

  x = time / (double)MACROTICK_DURATION;
return x + x * controller->ldata.clock_drift;
}

/* Returns clock correction that will apply during dt time (CSIM time) */
static signed get_clock_correction(CONTROLLER_MODEL *controller, double dt)
{
signed n, c, f;
double ft;

  c = cni_ps_get_csct(&controller->lcni->protocol_status);
if (c != 0 && dt > 0.0) {
    f = medl_ccp_get_free_running_macroticks(&controller-
>medl.controller_cfg_params) + 1;
    ft = controller_macro_to_csim(controller, f);

    n = (unsigned)(dt / ft) + 1;
    if (n < abs(c))
      c = (c > 0) ? n : -n;
  }

return c;
}

/* Returns CSIM hold time equal to n macroticks in cluster time.
  The hold() will allways end at macrotick boundary. */

static double get_hold_time(CONTROLLER_MODEL *controller, unsigned
macroticks)
{
  LOCAL_DATA *ldata = &controller->ldata;
double wait_time, dt;

  /* Update Cluster Time field */
  cni_get_cluster_time(controller->lcni);

  wait_time = controller_macro_to_csim(controller, macroticks)
    - (CSIM_TIME() - ldata->last_clock_access + ldata->last_mt_fract);
  dt = CSIM_TIME() - ldata->clock_correction_time + wait_time;
  wait_time -= controller_micro_to_csim(controller,
get_clock_correction(controller, dt));

  print_debug((controller->process, "Computed Hold Time: %0.4f\n",
wait_time));

return wait_time;
}

/* Compute next Action Time (in CSIM time) and store it in local data */
static void set_next_action_time(CONTROLLER_MODEL *controller)
{
unsigned t1 = cni_get_cluster_time(controller->lcni);
unsigned t2 = c_state_get_time(&controller->lcni->c_state);

  controller->ldata.action_time =    CSIM_TIME()
    + get_hold_time(controller, (t2 - t1) & bit_mask(16));
  print_debug((controller->process, "CState-Time = %d, Cluster-Time = %d,
Next Action Time = %0.4f\n",
    t2, t1, (controller)->ldata.action_time));
  print_debug((controller->process, "External rate correction = %d\n",
    (controller)->cni.external_rate_correction));
}

/* Wait until Transmission Phase ends
  This must be a macro because hold() is illegal outside
s_program...s_end_program block. */
#define wait_for_tp_end(controller) \
  controller_hold(controller, bit_mask(16) & ( \
    c_state_get_time(&controller->lcni->c_state) \
    + medl_sce_get_sru_slot_duration(get_current_sce((CONTROLLER
*)controller))    \
    - medl_isp_get_ifg_time(&controller->medl.implementation_params) \
    - cni_get_cluster_time(controller->lcni) \
  ))

/* Wait until next Transmission Phase starts
  This must be a macro because hold() is illegal outside
s_program...s_end_program block. */
#define wait_for_tp_start(controller) \
  controller_hold(controller, bit_mask(16) & ( \
    c_state_get_time(&controller->lcni->c_state) \
    - cni_get_cluster_time(controller->lcni) \
  ))

/* Controller initialization (INIT state):
   - clears CNI, sets Null Frame vect, Personality
   - checks MEDL
   - transits into LISTEN state */

static void model_init_state(CONTROLLER_MODEL *controller)
{
  process_cancel((PROCESS *)((BUS_GUARDIAN_MODEL *)controller-
>bus_guardian)->process);
  controller_init_state((CONTROLLER *)controller);
}

/* Initializes the LISTEN state of the controller.
   - checks controller's state
   - sets BG params for cold start nodes
   - sets wake flags
  Returns zero if all conditions are OK, nonzero otherwise (state
transition follows). */
```

```
static int model_prepare_listen_state(CONTROLLER_MODEL *controller)
{
int result = controller_prepare_listen_state((CONTROLLER *)controller);

if (result == 0) {
    controller->wake_on_co = 1;
    controller->wake_on_ca = 1;
    controller->wake_on_receive = 1;
  }

return result;
}

/* Finishes the LISTEN state of the controller.
  - stops receive, clears wake flags
  - checks bus for I-Frames
  - synchronizes on valid I-Frame
  - enter Cold Start if enabled and bus is silent */

static void model_finish_listen_state(CONTROLLER_MODEL *controller)
{
  controller->wake_on_receive = 0;
  controller->wake_on_co = 0;
  controller->wake_on_ca = 0;

  controller_finish_listen_state((CONTROLLER *)controller);
}

/* Initializes controller's Cold Start state (1st part):
  - checks controller's state
  - initializes C-State, stores CSim time
  - sends cold start frame, increments I-Frame switch
  Returns zero if everything is OK, otherwise nonzero code (state
transition) */

static int model_prepare_cold_start_state(CONTROLLER_MODEL *controller)
{
int result;

  controller->saved_csim_time = CSIM_TIME();
  result =  controller_prepare_cold_start_state((CONTROLLER *)controller);

if (result == 0) {
    /* Prepare cold start timeout */
    controller->wake_on_co = 1;
    controller->wake_on_ca = 1;
    controller->wake_on_receive = 1;
  }

return result;
}

/* Finishes the COLD_START state of the controller.
  - stops receive, clears wake flags
  - updates time fields and MEDL position
  - checks bus for I-Frames - synchronizes on valid I-Frame
  Returns zero if valid I-Frame was received - synchronize, nonzero
otherwise */
```

```
static int model_finish_cold_start_state(CONTROLLER_MODEL *controller)
{
  controller->wake_on_receive = 0;
  controller->wake_on_co = 0;
  controller->wake_on_ca = 0;

  /* update C-State Time and Medl position fields */
  controller_run_cold_start_state((CONTROLLER *)controller,
    (unsigned)controller_csim_to_macro(controller, CSIM_TIME() -
controller->saved_csim_time));

return controller_finish_cold_start_state((CONTROLLER *)controller);
}

/* Prints controller's state information, used only for debuging. */
#define print_debug_controller_state(controller) \
  print_debug((controller->process, "%s MV=[%u%u%u%u] NFV=[%u%u%u%u] A=%u
F=%u I=%u, TI1=%u, Timer1=%u\n", \
    get_protocol_state_name((controller)->lcni-
>protocol_status.protocol_state), \
    (controller)->lcni->c_state.membership[0], \
    (controller)->lcni->c_state.membership[1], \
    (controller)->lcni->c_state.membership[2], \
    (controller)->lcni->c_state.membership[3], \
    (controller)->lcni->null_frame[0], \
    (controller)->lcni->null_frame[1], \
    (controller)->lcni->null_frame[2], \
    (controller)->lcni->null_frame[3], \
    (controller)->lcni->protocol_status.accept_counter, \
    (controller)->lcni->protocol_status.failed_counter, \
    (controller)->lcni->protocol_status.invalid_counter, \
    (controller)->cni.interrupt_enable.ti1, \
    (controller)->cni.timer1 \
  ))

/* CONROLLER process - program code
  Implements main state machine. */
#define model (p_my->model)
#define protocol (p_my->protocol) static s_program(CONTROLLER_PROCESS,
prog_controller)
#define test_co \
if (cni_get_co(&protocol->cni) != CONTROLLER_ON) { \
    set_state(protocol, PS_FREEZE); \
    break; \
  }
for (;;) {
    print_debug_controller_state(model);

    switch (cni_ps_get_protocol_state(&protocol->lcni->protocol_status)) {

    case PS_INIT:
      model_init_state(model);
      break;

    case PS_LISTEN:
      if (model_prepare_listen_state(model))
        break;
      if (!start_receive(protocol)) {
        if (is_cold_start_allowed(protocol)) {
          /* cold start allowed: start listen timeout */
          print_debug((p_my, "Starting listen timeout.\n"));
          update_cni(protocol);
          controller_hold(model, get_listen_timeout(protocol));
          test_co;
        } else {
          /* The bus is silent, we must wait passively for something to
happen. */
          print_debug((p_my, "Waiting for I-Frame.\n"));
          update_cni(protocol);
          passivate();
          test_co;
        }
      } else {
        /* There is a garbage on the bus, wait a while to recognize it. */
        print_debug((p_my, "Bad receive.\n"));
        update_cni(protocol);
        controller_hold(model, CONTROLLER_TIME_TO_RECOGNIZE_BAD_RECEIVE);
        test_co;
      }

      model_finish_listen_state(model);
      break;

    case PS_COLD_START:
      if (model_prepare_cold_start_state(model))
        break;
      controller_hold(model,
        medl_sce_get_sru_slot_duration(get_current_sce(protocol))
        - medl_isp_get_ifg_time(&protocol->medl.implementation_params)
      );
      test_co;

      if (!start_receive(protocol)) {
        /* The bus is silent, hold on for the cold start timeout period. */
        controller_hold(model, medl_isp_get_ifg_time(&protocol-
>medl.implementation_params));
        test_co;
        if (!cold_start_set_arm(protocol)) {
          controller_hold(model,
            get_cold_start_timeout(protocol)
            - medl_sce_get_sru_slot_duration(get_current_sce(protocol))
          );
          test_co;
        }
      } else {
        /* There is a garbage on the bus, wait a while to recognize it. */
        print_debug((p_my, "Bad receive.\n"));
        update_cni(protocol);
        controller_hold(model, CONTROLLER_TIME_TO_RECOGNIZE_BAD_RECEIVE);
        test_co;
      }

      model_finish_cold_start_state(model);
      break;
    case PS_STARTUP:
      /* FIXME: Don't test the Host lifesign field in the first TDMA round
or something like that */
    case PS_APPLICATION:
    case PS_READY:
    case PS_PASSIVE:
      test_co;
      /* Send message if in the controller is in its sending slot or
         receive message from the current sender.
         Checks TP synchornous interrupts.
         This macro is used only in synchronised state. */
      update_cni(protocol);
      set_next_action_time(model);

      if (psp(protocol)) {
        /* Current slot is sending slot, frames are prepared */
        wait_for_tp_start(model);
        test_co;
        raise_tp_sync_interrupt(protocol);
        if (protocol->ldata.immediate_send_permission) {
          model->bus_guardian->arm_signal(protocol->bus_guardian, 1);
          print_debug((model->process, "Hold 0\n"));
          hold(0);
          test_co;
        }
        send_frame(protocol, 0);
        send_frame(protocol, 1);
        wait_for_tp_end(model);
        test_co;
        set_normal_bg_params(protocol);
      } else {
        wait_for_tp_start(model);
        test_co;
        if (is_current_sender(protocol))
          protocol->bus_guardian->arm_signal(protocol->bus_guardian, 0);
        /* Not a sending slot: If there is any interrupt or */
        /* if Arm signal is required, then wait for action time */
        if (protocol->ldata.tp_sync_interrupt ||
medl_sce_get_ba(get_current_sce(protocol))) {
          /* We are already in action time. */
          raise_tp_sync_interrupt(protocol);
          if (!protocol->ldata.immediate_send_permission)
            set_bg_arm(protocol);
        }
        wait_for_tp_end(model);
        test_co;
      }
      if (cni_get_co(&protocol->cni) == CONTROLLER_ON) {
        prp(protocol);
      }

      start_receive(protocol);
      break;

    case PS_AWAIT:
      update_cni(protocol);
      passivate();
      set_state(protocol, PS_FREEZE);
      break;

    case PS_SELFTEST:
      update_cni(protocol);
      passivate();
      set_state(protocol, PS_FREEZE);
      break;

    case PS_FREEZE:
      channel_cancel_send(protocol->channel[0], protocol);
      channel_cancel_send(protocol->channel[1], protocol);
      process_cancel((PROCESS *)model->timer[0]);
      process_cancel((PROCESS *)model->timer[1]);
      if (cni_get_co(&protocol->cni) == CONTROLLER_ON)
        set_state(protocol, PS_INIT);
      else {
        cni_ps_set_controller_ready(&protocol->lcni->protocol_status, 0);
        cni_ps_set_c_state_valid(&protocol->lcni->protocol_status, 0);
        update_cni(protocol);
        passivate();
      }
      break;

    case PS_DOWNLOAD:
      passivate();
      set_state(protocol, PS_FREEZE);
      break;
    default:
      passivate();
      set_state(protocol, PS_FREEZE);
      break;
    }
  }
s_end_program
#undef model
#undef protocol
static begin_destructor(CONTROLLER_PROCESS, controller_destroy)
  free(p_my->model);
end_destructor


/* calcullates actual new value of the cluster time */
/* warning: direct access to cni->cluster_time */

static void controller_update_cluster_time(CONTROLLER *controller)
{
  CONTROLLER_MODEL *controller_model = (CONTROLLER_MODEL *)controller;
  LOCAL_DATA *ldata = &controller->ldata;
  MEDL_CCP *medl_ccp = &controller->medl.controller_cfg_params;
signed c;
unsigned n;
double dt, x;

  /* Clock correction */
  dt = CSIM_TIME() - ldata->clock_correction_time;
  c = get_clock_correction(controller_model, dt);
  cni_ps_increment_csct(&controller->lcni->protocol_status, -c);
  ldata->clock_correction_time += abs(c) *
    controller_macro_to_csim(controller_model,
medl_ccp_get_free_running_macroticks(medl_ccp));

  /* x is the number of macroticks that elapsed since the last call to this
function,
    counting the clock drift and clock correction "c" */
  x = CSIM_TIME() - ldata->last_clock_access + ldata->last_mt_fract;
  x = controller_csim_to_macro(controller_model, x);
  x += (double)c / medl_ccp_get_micro_to_macro(medl_ccp);

  /* last_clock_access is now current time */
  ldata->last_clock_access = CSIM_TIME();
  ldata->last_mt_fract = controller_macro_to_csim(controller_model,
fract(x));

  /* Increment cluster time by the integral part of elapsed macroticks. */
  n = (unsigned)x;
if (controller->lcni->cluster_time + n > bit_mask(16))
    cni_increment_time_overflow_counter(&controller->cni, n >> 16);
  controller->lcni->cluster_time = (controller->lcni->cluster_time + n) &
bit_mask(16);
  controller->cni.cluster_time = controller->lcni->cluster_time;

    /* If a clock correction has just been applied, timer interrupt
processes must be replanned. */
    if (c != 0) {
/*              controller_timer_changed((CONTROLLER_MODEL *)controller,
0);
    controller_timer_changed((CONTROLLER_MODEL *)controller, 1);*/
    }
}

/* called upon change of cni.co field */

void controller_co_changed(CONTROLLER *controller)
{
  CONTROLLER_MODEL *controller_model = (CONTROLLER_MODEL *)controller;
unsigned value = cni_get_co(&controller->cni);

if (value == CONTROLLER_ON || controller_model->wake_on_co)
    process_activate_delay((PROCESS *)controller_model->process, 0);

if (value == CONTROLLER_ON && cni_ps_get_protocol_state(&controller->lcni-
>protocol_status) != PS_FREEZE)
    cni_ps_set_protocol_state(&controller->lcni->protocol_status,
PS_FREEZE);
}

/* called upon change of cni.ca field */

void controller_ca_changed(CONTROLLER *controller)
{
  CONTROLLER_MODEL *controller_model = (CONTROLLER_MODEL *)controller;
unsigned value = cni_get_ca(&controller->cni);

if (value == CONTROLLER_AWAIT && controller_model->wake_on_ca)
```

```
    process_activate_delay((PROCESS *)controller_model->process, 0);
}

/* Initializes a Controller Model structure. Returns pointer to the
structure. */

static CONTROLLER_MODEL *controller_model_init(CONTROLLER_MODEL
*controller_model)
{
#define controller ((CONTROLLER *)controller_model)

char *s, t[] = "BG - ";
int i;
char *ts[2];

  controller_init(controller);

  controller_model->receive_callback = model_receive_callback;
  controller_model->update_cluster_time = controller_update_cluster_time;

  controller_model->wake_on_co = 0;
  controller_model->wake_on_ca = 0;
  controller_model->wake_on_receive = 0;

  s = (char *)malloc(strlen(t) + strlen(controller_model->process->name) +
1);
  strcpy(s, t);
  strcat(s, controller_model->process->name);
  controller->bus_guardian = (BUS_GUARDIAN *)bus_guardian_create(s);
  bus_guardian_attach_controller((BUS_GUARDIAN_MODEL *)controller-
>bus_guardian, controller);
  process_activate_delay((PROCESS *)((BUS_GUARDIAN_MODEL *)controller-
>bus_guardian)->process, 0);

  ts[0] = (char *)malloc(strlen(controller_model->process->name) + 10);
  ts[1] = (char *)malloc(strlen(controller_model->process->name) + 10);
  strcpy(ts[0], controller_model->process->name);
  strcat(ts[0], ": Timer 1");
  strcpy(ts[1], controller_model->process->name);
  strcat(ts[1], ": Timer 2");
for (i = 0; i < 2; i++)
    attach_timer_process(controller_model, i, timer_process_create(ts[i]));

return controller_model;
#undef controller
}


/* Called by the channel after a frame is received. */

static void model_receive_callback(void *controller, unsigned channel)
{
#define ctrl ((CONTROLLER_MODEL *)controller)
  LOCAL_DATA *ldata = &ctrl->ldata;
  FRAME *frame = &ldata->frame[channel];

  controller_receive_callback(controller, channel);
  ctrl->ldata.mtd_a = (signed)controller_csim_to_micro(ctrl,
    ldata->frame[0].time_of_delivery - ldata->action_time);
  ctrl->ldata.mtd_b = (signed)controller_csim_to_micro(ctrl,
    ldata->frame[1].time_of_delivery - ldata->action_time);

if (ctrl->wake_on_receive && frame->state == FRAME_STATE_COMPLETE)
    process_activate_delay((PROCESS *)((CONTROLLER_MODEL *)controller)-
>process, 0);
#undef ctrl
}

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

/* Create and initialize CONTROLLER process */
CONTROLLER_MODEL *controller_create(char *name)
{
  CONTROLLER_MODEL *controller_model;

  controller_model = (CONTROLLER_MODEL *)malloc(sizeof(CONTROLLER_MODEL));

  controller_model->process = s_new_process(CONTROLLER_PROCESS,
prog_controller);
  controller_model->process->protocol = (CONTROLLER *)controller_model;
  controller_model->process->model = controller_model;
  set_name(controller_model->process, name);
  set_destructor(controller_model->process, controller_destroy);

return controller_model_init(controller_model);
}
```

4.37. sp1ctrl.h

```
/*
* This software is part of an EU project:
*     FIT - Fault Injection for TTA
*     IST-1999-10748
*
* TTP/C - protocol v.0.1
*
* Design (c) TTTech & TU Vienna
* Implementation (c) UWB Pilsen & CTU Prague
*
* sp1ctrl.h
* Functions and definitions concerning the controller c-sim process
*
* version 0.6 - 2001-03-13
*/

#ifndef _SP1CTRL_H
#define _SP1CTRL_H

#include "kr_csim.h"
#include "fl1func.h"
#include "fp1ctrl.h"
#include "sp1chan.h"
```

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

derived_process(timer_process)
void *controller;        /* Parent controller */
int timer;            /* Number of the timer (0,1)*/
end_derived(TIMER_PROCESS);
struct controller_model;

derived_process(controller_process)
  CONTROLLER *protocol;            /* Pointer to protocol fields */
struct controller_model *model; /* Pointer to model fields */
end_derived(CONTROLLER_PROCESS);
typedef struct controller_model {
  d_controller
  CONTROLLER_PROCESS *process;

unsigned wake_on_co;      /* Should the cni wake the controller up upon co
change? */
unsigned wake_on_ca;      /* Should the cni wake the controller up upon ca
change? */
unsigned wake_on_receive; /* Should the cni wake the controller up upon
receive of a frame? */

double saved_csim_time;   /* temporary stoarge field */

  TIMER_PROCESS *timer[2];  /* timer processes */
} CONTROLLER_MODEL;

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

/* create and initialize CONTROLLER process */
CONTROLLER_MODEL *controller_create(char *name);

/* called upon change of cni.co field */

void controller_co_changed(CONTROLLER *controller);

/* called upon change of cni.ca field */

void controller_ca_changed(CONTROLLER *controller);

/* Shedule the timer process or cancel its shedule if timer is disabled. */
void controller_timer_changed(CONTROLLER_MODEL *controller, int timer);


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

#endif /* #ifndef _SP1BUSG_H */
```

4.38. sp1chan.cpp

```
/*
* 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
*
* sp1chan.c
* Functions and definitions concerning the CHANNEL c-sim process
*
* version 0.14 - 2001-04-10
*/

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

#include "sp1chan.h"

#include "fp1chan.h"
#include "kr_csim.h"
#include "fp1frame.h"
#include "sp1ctrl.h"
#include "sl1func.h"

/* Number of bits transfered in one microsecond */
#define CHANNEL_BIT_DURATION    ( (double)1000000 / CHANNEL_SPEED )

/* Length of Start of Frame pattern transmitted on the bus (in bits) */
#define CHANNEL_SOF_LENGTH        8
static void do_fault_injection(CHANNEL_MODEL *channel); static void
channel_model_init(CHANNEL_MODEL *channel);

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

/* Injects random faults into the transmitted frame. */

static void do_fault_injection(CHANNEL_MODEL *channel)
{
if (channel->faults_to_inject > 0) {
    print_debug((channel->process, ">> Transmission error (bit flip)\n"));
    channel->frame.frame.i.header.i_n_frame ^= 1;
    channel->injected_faults++;
    channel->faults_to_inject--;
  }
}

/* Initializes model simulation specific parameters of the model.
  Also runs protocol specific initialization*/

static void channel_model_init(CHANNEL_MODEL *channel)
{
  channel_init((CHANNEL *)channel);

  channel->faults_to_inject = 0;
  channel->injected_faults = 0;
}

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

```
#define model (p_my->model)
#define protocol (p_my->protocol)
/* Channel process - program code */

static s_program(CHANNEL_PROCESS, channel_prog)
for (;;) {
    do_fault_injection(model);

    if (!test_bus_guardian(protocol, protocol->sender)) {
      channel_send_complete(protocol);
    } else {
      model->active = 0;
      model->frame.noise = 0;
      model->frame.state = FRAME_STATE_NULL;
    }

    passivate();
  }
s_end_program
#undef model
#undef protocol

/* Destructor for Channel model structure.
  This function is run by the kernel after the channel process terminates.
*/

static begin_destructor(CHANNEL_PROCESS, channel_destroy)
  free(p_my->model);
end_destructor


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

/* Creates and initializes Channel model.
  Returns pointer to the created model structure. */
CHANNEL_MODEL *channel_create(char *name)
{
  CHANNEL_MODEL *channel = (CHANNEL_MODEL *)malloc(sizeof(CHANNEL_MODEL));

  channel->channel_send_frame = channel_send_frame;

  CHANNEL_PROCESS *process = s_new_process(CHANNEL_PROCESS, channel_prog);

  channel->process = process;
  process->protocol = (CHANNEL *)channel;
  process->model = channel;
  set_destructor(process, channel_destroy);
  set_name(process, name);
  channel_model_init(channel);

return channel;
}

/* Sends a frame from the specified controller on the channel.
  Transmission duration is calculated from bus speed and frame length.
  Returns nonzero code if nay error occurs, 0 otherwise */
```

```
int channel_send_frame(CHANNEL *channel, void *controller, PTR_CALLBACK
callback)
{
double duration;

  ((CONTROLLER *)controller)->ldata.frame[channel->bus_id].time_of_delivery
= CSIM_TIME();
if (channel_send_init((CHANNEL *)channel, controller, callback))
    return 1;

  duration = (CHANNEL_SOF_LENGTH + frame_get_length(&channel->frame)) *
CHANNEL_BIT_DURATION;
  process_activate_delay((PROCESS *)((CHANNEL_MODEL *)channel)->process,
duration);

  print_debug((((CHANNEL_MODEL *)channel)->process, "Sending %c-frame
(duration %0.2f, arival at %0.2f)\n",
    (channel->frame.type == FRAME_TYPE_I) ? 'I' : 'N',
    duration,
    CSIM_TIME() + duration
  ));

return 0;
}

/* Called by the sender when it stops sending. */

void channel_cancel_send(CHANNEL *channel, void *controller)
{
    if (controller == channel->sender && channel->active) {
    channel->frame.state = FRAME_STATE_RECEIVING;

    process_cancel((PROCESS *)((CHANNEL_MODEL *)channel)->process);

    if (!test_bus_guardian(channel, channel->sender)) {
      channel_send_complete(channel);
    } else {
      channel->active = 0;
      channel->frame.noise = 0;
      channel->frame.state = FRAME_STATE_NULL;
    }

  }
}
```

4.39. sp1chan.h

```
/*
* This software is part of an EU project:
*     FIT - Fault Injection for TTA
*     IST-1999-10748
*
* TTP/C - protocol v.0.1
*
* Design (c) TTTech & TU Vienna
* Implementation (c) UWB Pilsen & CTU Prague
*
* sp1chan.h
* Functions and definitions concerning the CHANNEL c-sim proces
*
* version 0.7 - 2001-03-18
*/

#ifndef _SP1CHAN_H
#define _SP1CHAN_H

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

#include "fp1chan.h"
#include "kr_csim.h"
struct channel_model;

/* C-Sim extension of CHANNEL data structure */
derived_process(channel_process)
  CHANNEL *protocol;               /* Pointer to protocol fields */
struct channel_model *model;     /* Pointer to model fields */
end_derived(CHANNEL_PROCESS);

/* Channel data structure enhanced with simulation oriented fields. */
typedef struct channel_model {
  d_channel                  /* Inherited protocol specified fields */
  CHANNEL_PROCESS *process;  /* Pointer to C-Sim process simulating this
channel */
unsigned faults_to_inject; /* Number of faults to inject in following
transmissions */
unsigned injected_faults;  /* Count of injeted faults */
} CHANNEL_MODEL;


/* Creates and initializes Channel model.
  Returns pointer to the created model structure. */
CHANNEL_MODEL *channel_create(char *name);

/* Sends a frame from the specified controller on the channel.
  Transmission duration is calculated from bus speed and frame length.
  Returns nonzero code if nay error occurs, 0 otherwise */
int channel_send_frame(CHANNEL *channel, void *controller, PTR_CALLBACK
callback);

/* Called by the sender when it stops sending. */
void channel_cancel_send(CHANNEL *channel, void *controller);

#endif /* #ifndef _SP1CHAN_H */
```