An ER2 Library in C

A Collection of Functions for Using and Configuring the ER2

Dirk Bächle
TI6 (Distributed Systems)
Technical University Hamburg-Harburg

December 2, 2003

Contents

1 Introduction

The functions that are developed throughout this text rely on the Linux device driver `er2p` and the device file `/dev/er2p` being installed. How to do this is described in [2], including a short introduction to the ER2, device drivers and NOWEB. Most of the following descriptions are based on [3]. It deals with the addresses of the Fifth protocol, the bootstrap and the message passing in detail.

2 Headers

2.1 The C library er2.c

The basic structure of the library looks like this:

1a ⟨*er2.c* 1a⟩≡

```
⟨Header 1b⟩
⟨Include files 2a⟩
⟨Defines 8c⟩
⟨Structs 54b⟩
⟨Global variables 3b⟩
⟨Functions 3a⟩
```

The library starts with a disclaimer and some general information.

1b ⟨*Header* 1b⟩≡ (1a) 16b▷

```
⟨Disclaimer 1c⟩

/** \file er2.c
An ER2 library in C. A collection of functions for using and configuring
the ER2.
\author Dirk Baechle
\version 3.0
\date 26.11.2003
*/
```

Defines:
`ER2`, used in chunks 2c, 14c, 66a, 70c, 78a, 92b, 94d, and 95.
Uses `file` 2c.

1c ⟨*Disclaimer* 1c⟩≡ (1b 2c)

```
/* This file was created automatically from the file er2.nw by NOWEB. */
/* If you want to make changes, please edit the source file er2.nw. */
/* A complete documentation is in er2.tex, i.e. er2.dvi and er2.ps. */
/* Read it to understand why things are as they are. Thank you! */
```

Uses `file` 2c.

Next come the include files.

2a ⟨*Include files* 2a⟩≡ (1a) 94c▷

```
#include <stdio.h>
#include "../device_driver/er2gdef.h"
#include "er2.h"
#include "../device_driver/er2p.h"
```

Defines, structs and global variables are introduced step by step throughout the text.

2.2 The header file er2.h

The appropriate header file **er2.h** has the following structure:

2b ⟨*er2.h* 2b⟩≡

```
⟨HF: Header 2c⟩

#ifndef _ER2_H
#define _ER2_H

⟨HF: Defines 24a⟩
⟨HF: Typedefs 7a⟩
#ifdef __cplusplus
extern "C" {
#endif
⟨HF: Function prototypes 4b⟩
#ifdef __cplusplus
}
#endif

#endif
```

Defines:
  _ER2_H, never used.

2c ⟨*HF: Header* 2c⟩≡ (2b)

```
⟨Disclaimer 1c⟩

/** \file er2.h
Header file for the ER2 C library ''er2.c'' and all programs using it.
\author Dirk Baechle
\version 3.0
\date 26.11.2003
*/
```

Defines:
  file, used in chunks 1, 5b, 41c, 43–50, 75–78, 94d, and 95.
Uses ER2 1b.

3 Functions

The *weaving* of the functions is done by splitting them up a little. First of all, one needs to talk to the device driver. Then, one would like to detect the current network and be able to translate between the logical and the detected physical addresses. Informations about the network—like the number of nodes and edges—are always helpful, as much as functions for configuring the crossbars of the nodes and for testing the switched connections. Of course, one would also be very happy if one could read and write pieces of data to and from the processors, loading and executing of programs included. For this, sending the so called *messages* is needed. The device file has to be opened in order to start and initialize the ER2. After all work with the ER2 is done, the device is released again.

3a ⟨*Functions* 3a⟩≡ (1a)
⟨*Stdout Control* 4a⟩
⟨*Basic IO* 6a⟩
⟨*Sending Messages* 15a⟩
⟨*Reading and Writing* 21a⟩
⟨*Network Informations* 28b⟩
⟨*Translating Addresses* 38a⟩
⟨*Loading Programs* 41a⟩
⟨*Executing Programs* 52b⟩
⟨*Configuring Single Connections* 57b⟩
⟨*Testing Connections* 86a⟩
⟨*Startup* 94d⟩
⟨*Shutdown* 95⟩

3.1 Stdout control

For all this functions some kind of error handling is needed. Because this library is thought of as being readily compiled for the user, no defines should be used in order to specify how much verbose the output will be.
Instead, two global variables—one for additional informations and one for errors—are provided that can be set by the user with the help of function calls. These *flags* are responsible for the output behaviour of the single functions.

3b ⟨*Global variables* 3b⟩≡ (1a) 5b▷

```
/** Should error messages be output or not? If the variable is set
* to 1, errors will be printed on stdout. Default value is 1.
*/
static int errors = 1;

/** Should additional messages be output or not? If the variable is set
* to 1, additional information will be printed on stdout. Default value is 0.
*/
static int verbose = 0;
```

Defines:

errors, used in chunks 4a, 5a, and 92b.
verbose, used in chunks 4a and 5a.
Uses error 5a.

4a ⟨*Stdout Control* 4a⟩≡ (3a) 5a▷

```
/** Switches on the output of additional messages/informations for
* all other functions.
*/
void verbose_on(void)
{
        verbose = 1;
}

/** Switches off the output of additional messages/informations for
* all other functions.
*/
void verbose_off(void)
{
        verbose = 0;
}

/** Switches on the output of error messages for
* all other functions.
*/
void errors_on(void)
{
        errors = 1;
}

/** Switches off the output of error messages for
* all other functions.
*/
void errors_off(void)
{
        errors = 0;
}
```

Defines:
errors_off, used in chunk 4b.
errors_on, used in chunk 4b.
verbose_off, used in chunk 4b.
verbose_on, used in chunks 4b and 96b.
Uses error 5a, errors 3b, and verbose 3b.

The function prototypes are added to the header file to make them available for the user.

4b ⟨*HF: Function prototypes* 4b⟩≡ (2b) 12d▷

```
extern void verbose_on(void);
extern void verbose_off(void);
extern void errors_on(void);
```

```
extern void errors_off(void);
```

Uses errors_off 4a, errors_on 4a, verbose_off 4a, and verbose_on 4a.

Additionally, functions for the output of errors and infos, respectively, are defined.

5a ⟨*Stdout Control* 4a⟩+≡ (3a) ◁4a

```
/** Outputs an error message to stderr if the flag [[errors]] is set.
* @param errmsg The error message
*/
static void error(char *errmsg)
{
  if (errors)
  {
    fprintf(stderr, "Error: %s\n", errmsg);
  }
}

/** Outputs an info message to stdout if the flag [[verbose]] is set.
* @param infomsg The info message
*/
static void info(char *infomsg)
{
  if (verbose)
  {
    fprintf(stdout, "%s\n", infomsg);
  }
}
```

Defines:
error, used in chunks 3b, 4a, 10c, 12c, 15–23, 25–27, 37, 41–50, 53c, 57–60, 62–64, 71–73, 75–78, 86–89, and 94d.
info, used in chunks 42–45, 49, 70d, 73c, 88c, and 92–95.
Uses errors 3b, message 15d, and verbose 3b.

3.2 Basic IO

For talking to the device driver a file handle is needed. The device itself will be opened/closed during the startup/shutdown of the ER2.

5b ⟨*Global variables* 3b⟩+≡ (1a) ◁3b 9a▷

```
/** Handle for the device file. */
static int device_file = -1;
```

Defines:
device_file, used in chunks 6, 14c, 94d, and 95.
Uses file 2c.

Now the single functions like reading or writing a word/address can be implemented, based on [2]:

6a ⟨*Basic IO* 6a⟩≡ (3a) 6b▷

```
/** Writes a 16-bit address word to the IDMA port of the
* ADSP-2181 processor.
* @param address The 16-bit address word
*/
static void ifc_write_address_word(int address)
{

        ioctl(device_file, IOCTL_ER2_WRITE_ADDRESS, &address);

}
```

Defines:
  `ifc_write_address_word`, used in chunk 8a.
Uses `device_file` 5b.

While reading and writing data words, the length of the buffer has to be set first.

6b ⟨*Basic IO* 6a⟩+≡ (3a) ◁6a 6c▷

```
/** Writes a 16-bit word data array to the IDMA port of the
* ADSP-2181 processor.
* @param data Pointer to the 16-bit word data array
* @param length Length of the array
*/
static void ifc_write_data_words(int *data, int length)
{
        /* Is the length > 1? */
        if (length > 1)
        {
                /* Set length for device */
                ioctl(device_file, IOCTL_ER2_SET_LENGTH, &length);
        }

        /* Write data words */
        ioctl(device_file, IOCTL_ER2_WRITE_WORDS, data);

}
```

Defines:
  `ifc_write_data_words`, used in chunks 9b and 10a.
Uses `device_file` 5b.

6c ⟨*Basic IO* 6a⟩+≡ (3a) ◁6b 7b▷

```
/** Reads a 16-bit word data array from the IDMA port of the
* ADSP-2181 processor.
* @param data Pointer to the 16-bit word data array
```

```
* @param length Length of the array
*/
static void ifc_read_data_words(int *data, int length)
{
        /* Is the length > 1? */
        if (length > 1)
        {
                /* Set length for device */
                ioctl(device_file, IOCTL_ER2_SET_LENGTH, &length);
        }

        /* Read data words */
        ioctl(device_file, IOCTL_ER2_READ_WORDS, data);

}
```

Defines:
`ifc_read_data_words`, used in chunks 11b and 12a.
Uses `device_file` 5b.

Some routines for reading from and writing to the root processor directly shall be added, so the concept of logical vs. physical addresses is introduced. The memory of the ADSP-2181 on an ER2 module ranges from physical address `0x0000` up to physical address `0x7FFF`. It can be split into 24-bit program memory (PM) from physical address `0x0000` to physical address `0x3FFF` and 16-bit data memory (DM) from physical address `0x4000` to physical address `0x7FFF`.
In the following, logical addresses from `0x0000` to `0x3FFF` shall be used in both: program **and** data memory. This suits the definitions in the architecture file (`*.ach`) best and means that logical addresses in PM will stay the same when converted to physical addresses. Logical adresses in DM have to be mapped as follows:

	Logical address	**Physical address**
PM	`0x0000–0x3FFFF`	`0x0000–0x3FFF`
DM	`0x0000–0x3FFFF`	`0x4000–0x7FFF`

In order to distinguish between program and data memory a new type called `memory_class` is defined.

7a ⟨*HF: Typedefs* 7a⟩≡ (2b) 29b▹

```
/** Classification of the memory type */
enum memory_class {prog, dat};
typedef enum memory_class memory_class;
```

Defines:
`memory_class`, used in chunks 8a, 9c, 11c, 12d, 21b, 23b, 25d, 28a, 44a, 45c, and 50c.

7b ⟨*Basic IO* 6a⟩+≡ (3a) ◃6c 14c▹
⟨*Root Write Address* 8a⟩
⟨*Root Write Data Words* 8b⟩
⟨*Root Read Data Words* 11a⟩

For mapping the logical address to the physical one, the Bit 14 is set if *data memory* (DM) is specified.

8a ⟨*Root Write Address* 8a⟩≡ (7b)

```
/** Converts a logical address (0x0000--0x3FFF) in either program or
* data memory to a physical address and sets this address for the root
* processor. Logical program addresses 0x0000--0x3FFFF will result in
* physical addresses 0x0000--0x3FFF. The logical data addresses
* 0x0000--0x3FFF are mapped to the physical addresses 0x4000--0x7FFF.
* @param address The 16-bit address word
* @param mem_class The type of memory (prog/dat)
*/
void root_write_address(int address, memory_class mem_class)
{
        if (mem_class == prog)
        {
          ifc_write_address_word(address);
        }
        else
        {
          ifc_write_address_word(0x4000 | address);
        }
}
```

Defines:
root_write_address, used in chunks 12d, 16, 18–20, 28d, 29a, 34c, and 53.
Uses ifc_write_address_word 6a and memory_class 7a.

At this point the next *translation problem* arises. The user is supposed to deliver an array of integer values `data` for writing to the memory of the root processor. If he wants to write to data memory everything is OK and a single call of the `ifc_write_data_words` function can be used.
But while writing to program memory each integer value of the `data` array is interpreted as 24-bit entity and has to be split up in two 16-bit words for `ifc_write_data_words`. Thus, each *program memory array* is divided up into small pieces of fixed size called *chunks*. Writing them subsequently until the whole array is transferred, the possible problem of running out of memory is avoided.

8b ⟨*Root Write Data Words* 8b⟩≡ (7b)
⟨*Root Write Program Chunk* 9b⟩
⟨*Root Write Array of Data Words* 9c⟩

A define for the size of a *program memory chunk* is introduced, valid for both *write* and *read* operations. It is measured in the number of 24-bit integers that can be transferred.

8c ⟨*Defines* 8c⟩≡ (1a) 14a▷

```
/** Specifies the size of a program memory chunk. */
#define ROOT_MEM_CHUNK_SIZE            40
```

Defines:
`ROOT_MEM_CHUNK_SIZE`, used in chunks 9–12.

The translated values are held in a global array.

9a ⟨*Global variables* 3b⟩+≡ (1a) ◁5b 30b▷

```
/** Array for the translation between 24-bit and 16-bit integers
while writing/reading to/from root processor memory. */
static int translated[2 * ROOT_MEM_CHUNK_SIZE];
```

Defines:
`translated`, used in chunks 9b and 11b.
Uses `ROOT_MEM_CHUNK_SIZE` 8c.

9b ⟨*Root Write Program Chunk* 9b⟩≡ (8b)

```
/** Writes an array of 24-bit integer values with the maximum size of
* ROOT_MEM_CHUNK_SIZE to the program memory of the root processor.
* @param data Pointer to the array of 24-bit integers
* @param size The size of the array
*/
static void root_write_program_chunk(int *data, int size)
{
  int i;

  /* Translating the values */
  for (i = 0; i < size; i++)
  {
    /* MSW - Most significant word is first */
    translated[2*i] = (data[i] >> 8) & 0xFFFF;
    /* LSW - Least significant word */
    translated[2*i+1] = data[i] & 0xFF;
  }

  /* Writing the values to the root processor */
  ifc_write_data_words(translated, 2 * size);

}
```

Defines:
`root_write_program_chunk`, used in chunk 10b.
Uses `ifc_write_data_words` 6b, `ROOT_MEM_CHUNK_SIZE` 8c, and `translated` 9a.

9c ⟨*Root Write Array of Data Words* 9c⟩≡ (8b)

```
/** Writes an array of integer values to the data or program memory
* of the root processor.
* @param data Pointer to the array of integers
* @param size The size of the array
* @param mem_class The type of memory (prog/dat)
```

```
*/
void root_write_data(int *data, int size, memory_class mem_class)
{

  switch (mem_class)
  {

    ⟨write words to data memory 10a⟩
    ⟨write words to program memory 10b⟩
    ⟨write to unknown type of memory 10c⟩

  }
}
```

Defines:
  root_write_data, used in chunks 10c, 12d, 16a, 18a, 28d, 29a, and 53.
Uses memory_class 7a.

10a ⟨*write words to data memory* 10a⟩≡ (9c)

```
/* Write to data memory */
case dat: ifc_write_data_words(data, size);
          break;
```

Uses ifc_write_data_words 6b.

10b ⟨*write words to program memory* 10b⟩≡ (9c)

```
/* Write to program memory */
case prog: while (size >= ROOT_MEM_CHUNK_SIZE)
           {
             root_write_program_chunk(data, ROOT_MEM_CHUNK_SIZE);
             size -= ROOT_MEM_CHUNK_SIZE;
             data += ROOT_MEM_CHUNK_SIZE;
           }

           /* Is there still a piece left? */
           if (size > 0)
           {
             root_write_program_chunk(data, size);
           }
           break;
```

Uses ROOT_MEM_CHUNK_SIZE 8c and root_write_program_chunk 9b.

10c ⟨*write to unknown type of memory* 10c⟩≡ (9c)

```
/* Error: Unknown memory type */
default: error("<root_write_data> : Unknown memory type!");
         break;
```

Uses error 5a and root_write_data 9c.

The same holds for the reading of data words from the root processor. A *read* from program memory has to be translated into two subsequent reads of 16-bit words.

11a ⟨*Root Read Data Words* 11a⟩≡ (7b)
  ⟨*Root Read Program Chunk* 11b⟩
  ⟨*Root Read Array of Data Words* 11c⟩

11b ⟨*Root Read Program Chunk* 11b⟩≡ (11a)

```
/** Reads an array of 24-bit integer values with the maximum size of
* ROOT_MEM_CHUNK_SIZE from the program memory of the root processor.
* @param data Pointer to the array of 24-bit integers
* @param size The size of the array
*/
static void root_read_program_chunk(int *data, int size)
{
  int i;

  /* Reading the values from the root processor */
  ifc_read_data_words(translated, 2 * size);

  /* Translating the values */
  for (i = 0; i < size; i++)
  {
    /*        ((           MSW           ) << 8) | (           LSW           ) */
    data[i] = ((translated[2*i] & 0xFFFF) << 8) | (translated[2*i+1] & 0xFF);
  }

}
```

Defines:
  root_read_program_chunk, used in chunk 12b.
Uses ifc_read_data_words 6c, ROOT_MEM_CHUNK_SIZE 8c, and translated 9a.

11c ⟨*Root Read Array of Data Words* 11c⟩≡ (11a)

```
/** Reads an array of integer values from the data or program memory
* of the root processor.
* @param data Pointer to the array of integers
* @param size The size of the array
* @param mem_class The type of memory (prog/dat)
*/
void root_read_data(int *data, int size, memory_class mem_class)
{

  switch(mem_class)
  {

    ⟨read words from data memory 12a⟩
```

⟨*read words from program memory* 12b⟩
⟨*read from unknown type of memory* 12c⟩

```
  }
}
```

Defines:
`root_read_data`, used in chunks 12, 16c, 18–20, and 34c.
Uses `memory_class` 7a.

12a ⟨*read words from data memory* 12a⟩≡ (11c)

```
/* Read from the data memory */
case dat: ifc_read_data_words(data, size);
          break;
```

Uses `ifc_read_data_words` 6c.

12b ⟨*read words from program memory* 12b⟩≡ (11c)

```
/* Read from the program memory */
case prog: while (size >= ROOT_MEM_CHUNK_SIZE)
           {
             root_read_program_chunk(data, ROOT_MEM_CHUNK_SIZE);
             size -= ROOT_MEM_CHUNK_SIZE;
             data += ROOT_MEM_CHUNK_SIZE;
           }

           /* Is there still a piece left? */
           if (size > 0)
           {
             root_read_program_chunk(data, size);
           }
           break;
```

Uses `ROOT_MEM_CHUNK_SIZE` 8c and `root_read_program_chunk` 11b.

12c ⟨*read from unknown type of memory* 12c⟩≡ (11c)

```
/* Error: Unknown memory type */
default: error("<root_read_data> : Unknown memory type!");
         break;
```

Uses `error` 5a and `root_read_data` 11c.

12d ⟨*HF: Function prototypes* 4b⟩+≡ (2b) ◁4b 20d▷

```
extern void root_write_address(int, memory_class);
extern void root_write_data(int *, int, memory_class);
extern void root_read_data(int *, int, memory_class);
```

Uses `memory_class` 7a, `root_read_data` 11c, `root_write_address` 8a, and `root_write_data` 9c.

3.3 Sending messages

Through the host interface only the root processor of the ER2 can be reached directly by writing to or reading from its memory. All communication to and from distant processors is done via sending the so called *messages*. They are used to distribute data around the network or gather information about its topology. A message consists of a sequence of 16-bit data words, specifying what kind of action should be triggered on which processor. The lengths of the provided messages, i.e. the number of data words, can range from 1 up to 66.

3.3.1 Writing a message data word

In order to send a message, its data words have to be written subsequently into a destined location of the root processor's memory. This is the address `0x0208` in DM (Data Memory) also called `HCMD`. The following steps have to be performed for **each single** data word of the message (see [3, p. 5]):

- Write the data word to address `0x0208`—also known as `HCMD`— in DM.
- Write the value `0x0001` to the address `0x0209`—also called `HNRDY`— in DM.
- Trigger an interrupt (`IRQ2`).
- Wait until the root processor sets `HNRDY`—or address `0x0209` in DM—back to `0x0000` to signal that the data word has been sent.

Then, the next data word can be *sent*.
If a message delivers a result, the 24-bit value—remember, it could be the data from a read in program memory—appears bytewise in the memory locations

Address in DM	**Name**	Result from DM	**Result from PM**
`0x0210`	`RPARH`	Bits 15–8	Bits 23–16
`0x0211`	`RPARL`	Bits 7–0	Bits 15–8
`0x0212`	`RPARB`		Bits 7–0

of the root processor.
If the result comes from a SHARC a second 24-bit result value is delivered into three additional memory places like this:

Address in DM	**Name**	Result from DM	**Result from PM**
`0x0213`	`XPARH`	Bits 15–8	Bits 23–16
`0x0214`	`XPARL`	Bits 7–0	Bits 15–8
`0x0215`	`XPARB`		Bits 7–0

3.3.2 When is a message result valid?

After writing a message to the root processor the data words are automatically sent to the specified processor. Depending on the topology of the network and the distance to the *root* the latency can be high or low. How long does one have to wait for the result?
The answer lies in memory location `0x020F`—also called `RPAR1`— in DM of the root processor. If `RPAR1` is initialized to `0x8000` before sending a message, the most significant bit is set to zero, once the results are valid.

Thus, after sending a message one has to wait until `RPAR1` is set back to `0x0000` again and then the results can be processed.
The mentioned memory locations are added as defines...

14a ⟨*Defines* 8c⟩+≡ (1a) ◁8c 15b▷
⟨*Defines: ADSP2181 (DM)* 14b⟩
⟨*Defines: ADSP2181 (PM)* 52c⟩

14b ⟨*Defines: ADSP2181 (DM)* 14b⟩≡ (14a) 28c▷

```
/* Memory locations in ADSP2181 Data Memory */
#define HCMD                            0x0208
#define HNRDY                           0x0209
#define RPAR1                           0x020F
#define RPARH                           0x0210
#define RPARL                           0x0211
#define RPARB                           0x0212
#define XPARH                           0x0213
#define XPARL                           0x0214
#define XPARB                           0x0215
```

Defines:
`HCMD`, used in chunk 16a.
`HNRDY`, used in chunk 16.
`RPAR1`, used in chunks 17 and 18.
`RPARB`, used in chunks 17c and 19.
`RPARH`, used in chunks 17c, 19, and 29a.
`RPARL`, used in chunks 17c and 19.
`XPARB`, used in chunks 19c and 20c.
`XPARH`, used in chunks 19c and 20c.
`XPARL`, used in chunks 19c and 20c.

Since triggering an interrupt on the ER2 is needed, the appropriate *Basic IO* function is added. The specified `address` is just a calling convention of the device driver, no data is changed or used.

14c ⟨*Basic IO* 6a⟩+≡ (3a) ◁7b

```
/** Triggers an interrupt on the ER2.
* @param address Dummy pointer
*/
static void ifc_irq_er2(int *address)
{

        ioctl(device_file, IOCTL_ER2_IRQ, &address);

}
```

Defines:
`ifc_irq_er2`, used in chunk 16a.
Uses `device_file` 5b and `ER2` 1b.

3.3.3 Defining the message functions

Three different message functions are provided. One without any results, one with a single and one with two results.

15a ⟨*Sending Messages* 15a⟩≡ (3a)
  ⟨*Message* 15d⟩
  ⟨*Message with result* 17c⟩
  ⟨*Message with two results* 19c⟩

First, the message without any results. After sending a single word of the message the affirmation of the ADSP2181 is expected. If—for some unknown reason—no acknowledge is received, the code should not loop forever. So, a maximum number of tries is specified for checking.

15b ⟨*Defines* 8c⟩+≡ (1a) ◁14a 17b▷
  ⟨*Defines: Retries* 15c⟩

15c ⟨*Defines: Retries* 15c⟩≡ (15b) 17a▷

```
/* Defines for Retries */
#define MAX_HNRDY_ATTEMPTS      1000
```

Defines:
  `MAX_HNRDY_ATTEMPTS`, used in chunk 16c.

15d ⟨*Message* 15d⟩≡ (15a)

```
/** Sends a message and continues immediately without waiting
* for any message results.
* @param word Pointer to an array containing the message
* @param size Size of the array (length of the message)
* @return ERROR if an error occured, OK else
*/
int message(int *word, int size)
{
  int data[1];
  int i, j;

  /* Send a message */
  for (j = 0; j < size; j++)
  {
    ⟨send single message data word 16a⟩
    ⟨wait for data word sent acknowledge 16c⟩
    ⟨check for message acknowledge error 16d⟩
  }

  return(OK);
}
```

Defines:
  `message`, used in chunks 5a, 16–20, 23b, and 27b.
Uses `error` 5a.

16a ⟨*send single message data word* 16a⟩≡ (15d 17c 19c)

```
/* Write data word to HCMD */
data[0] = word[j];
root_write_address(HCMD, dat);
root_write_data(data, 1, dat);

/* Initialize HNRDY variable, set it to 0x0001 */
root_write_address(HNRDY, dat);
data[0] = 1;
root_write_data(data, 1, dat);

/* Trigger interrupt IRQ2 */
ifc_irq_er2(data);
```

Uses `HCMD` 14b, `HNRDY` 14b, `ifc_irq_er2` 14c, `root_write_address` 8a, and `root_write_data` 9c.

16b ⟨*Header* 1b⟩+≡ (1a) ◁1b

```
// #define NO_TIMEOUT
```

16c ⟨*wait for data word sent acknowledge* 16c⟩≡ (15d 17c 19c)

```
root_write_address(HNRDY, dat);
root_read_data(data, 1, dat);

/* Check the HNRDY variable */
#ifndef NO_TIMEOUT
for (i = 0; (i < MAX_HNRDY_ATTEMPTS) && (data[0] != 0); i++)
#else
while (data[0] != 0)
#endif
{
  root_write_address(HNRDY,dat);
  root_read_data(data, 1, dat);
}
```

Uses `HNRDY` 14b, `MAX_HNRDY_ATTEMPTS` 15c, `root_read_data` 11c, and `root_write_address` 8a.

16d ⟨*check for message acknowledge error* 16d⟩≡ (15d)

```
if (data[0] != 0)
{
  error("<message> : Timeout while sending a message!");
  return(ERROR);
}
```

Uses `error` 5a and `message` 15d.

Now, the sending of a message with a result gets prepared. It has to be checked whether the result is valid or not. Again, only a finite number of attempts are allowed.

17a ⟨*Defines: Retries* 15c⟩+≡ (15b) ◁15c

```
#define MAX_RPAR1_ATTEMPTS      1000
```

Defines:
MAX_RPAR1_ATTEMPTS, used in chunk 18c.

As a second define, the initial value for the **RPAR1** memory location is specified.

17b ⟨*Defines* 8c⟩+≡ (1a) ◁15b 42a▷

```
/** Initial value for RPAR1 */
#define RPAR1_INIT              0x8000
```

Defines:
RPAR1_INIT, used in chunks 17–19.
Uses RPAR1 14b.

17c ⟨*Message with result* 17c⟩≡ (15a)

```
/** Sends a message, waits for the results in RPARL, RPARH and RPARB and
* returns them.
* @param word Pointer to an array containing the message
* @param size Size of the array (length of the message)
* @param rparl Pointer to the result for RPARL
* @param rparh Pointer to the result for RPARH
* @param rparb Pointer to the result for RPARB
* @return ERROR if an error occured, OK else
*/
int message_wfr(int *word, int size, int *rparl, int *rparh, int *rparb)
{

  int data[1];
  int i, j;

  ⟨initialize RPAR1 value 18a⟩

  /* Send the message */
  for (j = 0; j < size; j++)
  {
    ⟨send single message data word 16a⟩
    ⟨wait for data word sent acknowledge 16c⟩
    ⟨check for message-wfr acknowledge error 18b⟩
  }

  ⟨wait for valid results 18c⟩

  if ((data[0] & RPAR1_INIT) == RPAR1_INIT)
```

```
  {
    ⟨message-wfr results error 19a⟩
  }
  else
  {
    ⟨return first message result 19b⟩
  }
  return(OK);
}
```

Defines:
  `message_wfr`, used in chunks 18–20, 22c, and 29a.
Uses `error` 5a, `message` 15d, `RPAR1_INIT` 17b, `RPARB` 14b, `RPARH` 14b, and `RPARL` 14b.

18a ⟨*initialize RPAR1 value* 18a⟩≡ (17c 19c)

```
/* Initialize RPAR1 value */
data[0] = RPAR1_INIT;
root_write_address(RPAR1, dat);
root_write_data(data, 1, dat);
```

Uses `root_write_address` 8a, `root_write_data` 9c, `RPAR1` 14b, and `RPAR1_INIT` 17b.

18b ⟨*check for message-wfr acknowledge error* 18b⟩≡ (17c)

```
if (data[0] != 0)
{
  error("<message_wfr> : Timeout while sending a message!");
  return(ERROR);
}
```

Uses `error` 5a, `message` 15d, and `message_wfr` 17c.

18c ⟨*wait for valid results* 18c⟩≡ (17c 19c)

```
/* Check the RPAR1 variable = wait for valid results */
i = 0;
#ifndef NO_TIMEOUT
while ((i < MAX_RPAR1_ATTEMPTS) && ((data[0] & RPAR1_INIT) == RPAR1_INIT))
#else
while ((data[0] & RPAR1_INIT) == RPAR1_INIT)
#endif
{
  root_write_address(RPAR1, dat);
  root_read_data(data, 1, dat);

  i++;
}
```

Uses `MAX_RPAR1_ATTEMPTS` 17a, `root_read_data` 11c, `root_write_address` 8a, `RPAR1` 14b,
  and `RPAR1_INIT` 17b.

19a ⟨*message-wfr results error* 19a⟩≡ (17c)

```
error("<message_wfr> : Timeout while waiting for valid results!");
return(ERROR);
```

Uses `error` 5a and `message_wfr` 17c.

19b ⟨*return first message result* 19b⟩≡ (17c 19c)

```
/* Return valid first result */
root_write_address(RPARL, dat);
root_read_data(rparl, 1, dat);
root_write_address(RPARH, dat);
root_read_data(rparh, 1, dat);
root_write_address(RPARB, dat);
root_read_data(rparb, 1, dat);
```

Uses `root_read_data` 11c, `root_write_address` 8a, `RPARB` 14b, `RPARH` 14b, and `RPARL` 14b.

19c ⟨*Message with two results* 19c⟩≡ (15a)

```
/** Sends a message, waits for the results in RPARL, RPARH, RPARB, XPARL,
* XPARH and XPARB and
* returns them.
* @param word Pointer to an array containing the message
* @param size Size of the array (length of the message)
* @param rparl Pointer to the result for RPARL
* @param rparh Pointer to the result for RPARH
* @param rparb Pointer to the result for RPARB
* @param xparl Pointer to the result for XPARL
* @param xparh Pointer to the result for XPARH
* @param xparb Pointer to the result for XPARB
* @return ERROR if an error occured, OK else
*/
int message_wfr_x(int *word, int size, int *rparl, int *rparh,
                  int *rparb, int *xparl, int *xparh, int *xparb)
{
  int data[1];
  int i, j;

  ⟨initialize RPAR1 value 18a⟩

  /* Send the message */
  for (j = 0; j < size; j++)
  {
    ⟨send single message data word 16a⟩
    ⟨wait for data word sent acknowledge 16c⟩
    ⟨check for message-wfr-x acknowledge error 20a⟩
  }
```

```
  ⟨wait for valid results 18c⟩

  if ((data[0] & RPAR1_INIT) == RPAR1_INIT)
  {
    ⟨message-wfr-x results error 20b⟩
  }
  else
  {
    ⟨return first message result 19b⟩
    ⟨return second message result 20c⟩
  }
  return(OK);
}
```

Defines:
  message_wfr_x, used in chunk 20.
Uses **error** 5a, **message** 15d, **RPAR1_INIT** 17b, **RPARB** 14b, **RPARH** 14b, **RPARL** 14b, **XPARB** 14b, **XPARH** 14b, and **XPARL** 14b.

20a ⟨*check for message-wfr-x acknowledge error* 20a⟩≡ (19c)

```
if (data[0] != 0)
{
  error("<message_wfr_x> : Timeout while sending a message!");
  return(ERROR);
}
```

Uses **error** 5a, **message** 15d, and **message_wfr_x** 19c.

20b ⟨*message-wfr-x results error* 20b⟩≡ (19c)

```
error("<message_wfr_x> : Timeout while waiting for valid results!");
return(ERROR);
```

Uses **error** 5a and **message_wfr_x** 19c.

20c ⟨*return second message result* 20c⟩≡ (19c)

```
/* Return valid second result */
root_write_address(XPARL, dat);
root_read_data(xparl, 1, dat);
root_write_address(XPARH, dat);
root_read_data(xparh, 1, dat);
root_write_address(XPARB, dat);
root_read_data(xparb, 1, dat);
```

Uses **root_read_data** 11c, **root_write_address** 8a, **XPARB** 14b, **XPARH** 14b, and **XPARL** 14b.

20d ⟨*HF: Function prototypes* 4b⟩+≡ (2b) ◁12d 28a▷

```
extern int message(int *, int);
extern int message_wfr(int *, int, int *, int *, int *);
extern int message_wfr_x(int *, int,
                         int *, int *, int *,
                         int *, int *, int *);
```

Uses `message` 15d, `message_wfr` 17c, and `message_wfr_x` 19c.

These messages can be used immediately for reading and writing to the memory of an arbitrary processor, i.e. node.

21a ⟨*Reading and Writing* 21a⟩≡ (3a)
⟨*Request Memory* 21b⟩
⟨*Broadcast Memory Piece* 23b⟩
⟨*Broadcast Memory* 25d⟩

Unfortunately, there is no *message* for reading a number of consecutive values. Single integers have to be requested repeatedly in order to fill the data array.

21b ⟨*Request Memory* 21b⟩≡ (21a)

```
/** Reads the memory of a processor at the given address.
* @param netaddress Physical address of the processor
* @param address Logical address in the memory
* @param mem_class The type of memory (prog/dat)
* @param data Pointer to the data array
* @param length Number of values to read
* @return ERROR if an error occured, OK else
*/
int request_memory(int netaddress, int address, memory_class mem_class,
                   int *data, int length)
{
  int rqmsg[2], res=0;
  int i, rparl[1], rparh[1], rparb[1];

  ⟨rq: check for valid memory class 22a⟩

  for (i = 0; i < length; i++)
  {
    ⟨rq: read single data value 22b⟩
  }

  return(OK);
}
```

Defines:
`request_memory`, used in chunks 22a, 28a, and 88c.
Uses `error` 5a and `memory_class` 7a.

22a ⟨*rq: check for valid memory class* 22a⟩≡ (21b)

```
/* Correct memory class type? */
if ((mem_class != prog) && (mem_class != dat))
{
  error("<request_memory> : Unknown data type!");
  return(ERROR);
}
```

Uses `error` 5a and `request_memory` 21b.

22b ⟨*rq: read single data value* 22b⟩≡ (21b)

```
⟨rq: send read message 22c⟩

if (res == OK)
{
  /* Data or program memory? */
  switch (mem_class)
  {
    ⟨rq: read from data memory 22d⟩
    ⟨rq: read from program memory 23a⟩
  }
}
else
{
  return(ERROR);
}
```

Here, the *message* for reading a single word is constructed. For details see [3, p. 6].

22c ⟨*rq: send read message* 22c⟩≡ (22b)

```
/* Request memory */
rqmsg[0] = 0x8200 + netaddress;
rqmsg[1] = ((int) mem_class) ? (address+i): (address+i) + 0x4000;
res = message_wfr(rqmsg, 2, rparl, rparh, rparb);
```

Uses `message_wfr` 17c.

22d ⟨*rq: read from data memory* 22d⟩≡ (22b)

```
/* Data memory  */
case dat:
  data[i] = ((rparh[0] & 0xFF) << 8) + rparl[0];
  break;
```

23a ⟨*rq: read from program memory* 23a⟩≡ (22b)

```
/* Program memory */
case prog:
  data[i] = ((rparh[0] & 0xFF) << 16) +
            ((rparl[0] & 0xFF) << 8) + rparb[0];
  break;
```

The *message* for writing values to a processor has a lot more to offer. Not only does it enable the user to write a number of values at once, but code/data can also be broadcasted to a whole predefined group of processors simultaneously. For details see [3, p. 7].
Since a maximum number of 64 values (16-bit integer) can be written, a function is defined for broadcasting an array with a length ranging from 1 to the upper bound of 32 (PM) or 64 (DM).

23b ⟨*Broadcast Memory Piece* 23b⟩≡ (21a)

```
/** Loads a piece of program or data code with maximum size LOAD_MAX
* to a single processor or a group of processors.
* @see broadcast_memory.
* @param groupnr The group number
* @param netaddress Physical address of the processor if groupnr is SINGLE
* @param address Logical address in memory
* @param mem_class Type of memory (prog/dat)
* @param program Pointer to the data array
* @param length The size of the code piece
* @return ERROR if an error occurs, OK else
*/
static int broadcast_memory_piece(int groupnr, int netaddress,
                                  int address, memory_class mem_class,
                                  const int *program, int length)
{
  int bcpmsg[64 + 2];
  int mlength;
  int i, mi;

  ⟨bcp: construct write message header 24b⟩

  switch (mem_class)
  {
    ⟨bcp: load data code 25a⟩
    ⟨bcp: load program code 25b⟩
  }

  /* Send write message */
  message(bcpmsg, mlength+2);

  return(OK);
}
```

Defines:
  `broadcast_memory_piece`, used in chunk 27.
Uses `broadcast_memory` 25d 71a, `error` 5a, `memory_class` 7a, `message` 15d, and `SINGLE` 24a.

While writing the header for the *write message* a broadcast to a group and a transfer to a single processor have to be differentiated. Therefore, a group number called `SINGLE` is defined, outside of the standard range 0–63.
If a broadcast is done and the given group number is `SINGLE` the data is written to a single node. Otherwise the `netaddress` is disregarded and `groupnr` is used instead.

24a ⟨*HF: Defines* 24a⟩≡ (2b) 30a▷

```
/** Value that specifies the ''group'' SINGLE (processor) */
#define SINGLE          128
```

Defines:
  `SINGLE`, used in chunks 23–26, 37c, 41c, 43–45, 50c, 53c, 71–73, and 86–89.

Great care has to be taken while differentiating between `PM` (Program Memory) and `DM` (Data Memory) addresses in a message. As stated in 3.2 the memory space of the ADSP-2181 is separated into `PM` below address `0x4000` and `DM` ranging from `0x4000-0x7FFF`.
However, in a *message* it is exactly the other way round, due to the Fifth protocol ([3, p. 6])!
The variable `mlength` specifies the number of 16-bit words needed for the message, in contrast to `length` which is the given number of words to transfer. For a write to data memory both variables are equal, but for program memory they differ. Two consecutive 24-bit words are packed into three 16-bit words within a message.

24b ⟨*bcp: construct write message header* 24b⟩≡ (23b)

```
if (mem_class == prog)
{
  mlength = (3 * length + 1) / 2;
  bcpmsg[1] = address + 0x4000;
}
else
{
  mlength = length;
  bcpmsg[1] = address;
}

if (groupnr != SINGLE)
  bcpmsg[0] = (mlength-1)*0x200+0x100+groupnr;
else
  bcpmsg[0] = (mlength-1)*0x200+netaddress;
```

Uses `SINGLE` 24a.

25a ⟨*bcp: load data code* 25a⟩≡ (23b)

```
/* Load data code */
case dat:
  for (i = 0; i < length; i++)
    bcpmsg[i+2] = program[i] & 0xFFFF;
  break;
```

25b ⟨*bcp: load program code* 25b⟩≡ (23b)

```
/* Load program code */
case prog:
  i = 0;
  mi = 2;
  while (i < length)
  {
    ⟨bcp: pack program words 25c⟩
  }
  break;
```

Packing program words is done after the following scheme: Assume, there are two program words `A` and `B` each consisting of three 8-bit parts `M`, `B` and `L`. `M` is the most significant 8-bit word, while `L` is the least significant. For a message the 24-bit words `A` and `B` are now stored in three 16-bit words as:

Word 1	:	`AM`, `AB`
Word 2	:	`BM`, `AL`
Word 3	:	`BB`, `BL`

25c ⟨*bcp: pack program words* 25c⟩≡ (25b)

```
/* Pack first 24-bit word */
bcpmsg[mi++] = ((program[i] >> 8) & 0xFFFF);
bcpmsg[mi] = (program[i++] & 0xFF);
/* Does a second 24-bit word exist? */
if (i < length)
{
  /* Yes, so pack it, too */
  bcpmsg[mi++] |= ((program[i] >> 16) & 0xFF) << 8;
  bcpmsg[mi++] = (program[i++] & 0xFFFF);
}
```

Now, arrays of arbitrary length can be transferred by calling `broadcast_memory_piece` repeatedly.

25d ⟨*Broadcast Memory* 25d⟩≡ (21a)

```
/** Loads a piece of program or data code
* to a single processor or a group of processors.
* @param groupnr The group number
```

```
* @param netaddress Physical address of the processor if groupnr is SINGLE
* @param address Logical address in memory
* @param mem_class Type of memory (prog/dat)
* @param program Pointer to the data array
* @param length The size of the code
* @return ERROR if an error occurs, OK else
*/
int broadcast_memory(int groupnr, int netaddress, int address,
                     memory_class mem_class, const int *program, int length)
{
  int i, load_max;

  ⟨bc: check for valid group 26a⟩
  ⟨bc: check for valid memory class 26b⟩
  ⟨bc: check for valid length 27a⟩
  ⟨bc: assign maximum number of load values 27b⟩

  ⟨bc: load data piecewise 27c⟩

  ⟨bc: check for last piece of data 27d⟩

  return(OK);
}
```

Defines:
broadcast_memory, used in chunks 23b, 26–28, 37c, 43c, 44a, 53c, 72a, 86b, 88, and 89.
Uses **error** 5a, **memory_class** 7a, and **SINGLE** 24a.

26a ⟨*bc: check for valid group* 26a⟩≡ (25d)

```
/* Valid group number? */
if (((groupnr > 63) || (groupnr < 0)) && (groupnr != SINGLE))
{
  error("<broadcast_memory> : Invalid group number! (Must be 0-63 or SINGLE)");
  return(ERROR);
}
```

Uses **broadcast_memory** 25d 71a, **error** 5a, and **SINGLE** 24a.

26b ⟨*bc: check for valid memory class* 26b⟩≡ (25d)

```
/* Correct memory class type? */
if ((mem_class != prog) && (mem_class != dat))
{
  error("<broadcast_memory> : Unknown data type!");
  return(ERROR);
}
```

Uses **broadcast_memory** 25d 71a and **error** 5a.

27a ⟨*bc: check for valid length* 27a⟩≡ (25d)

```
if (length < 1)
{
  error("<broadcast_memory> : Length of code is less than 1!");
  return(ERROR);
}
```

Uses `broadcast_memory` 25d 71a and `error` 5a.

27b ⟨*bc: assign maximum number of load values* 27b⟩≡ (25d)

```
/* Assign the maximum number of values to */
/* load with a single message */
if (mem_class == dat)
  load_max = 64;
else
  load_max = 42;
```

Uses `message` 15d.

27c ⟨*bc: load data piecewise* 27c⟩≡ (25d)

```
/* Load code to processor/s piecewise */
i = 0;
while (i < (length/load_max))
{
  if (broadcast_memory_piece(groupnr, netaddress, address+i*load_max,
                             mem_class, program+i*load_max,
                             load_max) == ERROR)
  {
    return(ERROR);
  }

  i++;
}
```

Uses `broadcast_memory_piece` 23b.

27d ⟨*bc: check for last piece of data* 27d⟩≡ (25d)

```
/* Is there still a piece of code with length < load_max? */
if ((i * load_max) < length)
{
  if (broadcast_memory_piece(groupnr, netaddress, address+i*load_max,
                             mem_class, program+i*load_max,
                             length-i*load_max) == ERROR)
  {
    return(ERROR);
  }
```

```
}
```

Uses broadcast_memory_piece 23b.

Finally, the prototypes are added to the header file.

28a ⟨*HF: Function prototypes* 4b⟩+≡ (2b) ◁20d 37d▷

```
extern int request_memory(int, int, memory_class, int *, int);
extern int broadcast_memory(int, int, int, memory_class, const int *, int);
```

Uses broadcast_memory 25d 71a, memory_class 7a, and request_memory 21b.

3.4 Detecting the network topology

So far, so good. Sending data and program code to any processor is supported. But which processors are available in the network and how are they connected? The next functions deal with these problems—and several similar ones.

28b ⟨*Network Informations* 28b⟩≡ (3a)

⟨*Mark Root Node* 28d⟩
⟨*Request Neighbour* 29a⟩
⟨*Detect Neighbours* 30e⟩
⟨*Pre-initialize Routing Table* 33b⟩
⟨*Initialize Routing Table* 33c⟩
⟨*Display Single Neighbour* 34d⟩
⟨*Display Routing Table* 35a⟩
⟨*Get Neighbour* 35c⟩
⟨*Get Number of Edges* 36a⟩
⟨*Get Number of Nodes* 36b⟩
⟨*Join Group* 37a⟩

The *root node*, i.e. the node that is directly attached to the host interface, has to be distinguishable from the other processors. He is *marked* by setting the variable 0x0222 (=MASTER) in DM (see [3, p. 2]).

28c ⟨*Defines: ADSP2181 (DM)* 14b⟩+≡ (14a) ◁14b 34b▷

```
#define MASTER                          0x0222
```

Defines:
MASTER, used in chunk 28d.

28d ⟨*Mark Root Node* 28d⟩≡ (28b)

```
/** Marks the root processor.
*/
static void mark_root_node(void)
{
  int data[1];

  data[0] = 8;
  root_write_address(MASTER, dat);
```

```
    root_write_data(data, 1, dat);
  }
```

Defines:
  mark_root_node, used in chunk 94d.
Uses MASTER 28c, root_write_address 8a, and root_write_data 9c.

For detecting the network topology there exists a special *message*. It returns the physical address of the processor's neighbour in the given direction (see [3, p. 5]). If there is no neighbour present, the result variable RPARH does not get written. Thus, RPARH is set to the netaddress initially. If it is still the same after the request, no connection in the direction dir was found.

29a ⟨*Request Neighbour* 29a⟩≡ (28b)

```
  /** Detects the physical address of a neighbour processor in the given
  * direction.
  * @param netaddress Physical address of a processor
  * @param dir The direction
  * @return Physical address of the neighbour processor
  */
  static int request_neighbour(int netaddress, direction dir)
  {
    int dummy;
    int data, nb;

    /* Initialize RPARH to processor address */
    nb = netaddress;
    root_write_address(RPARH, dat);
    root_write_data(&nb, dat, 1);

    /* Message: Get the neighbour processor in the given direction */
    data = 0xc300 + netaddress + 0x100 * dir;
    message_wfr(&data, 1, &dummy, &nb, &dummy);

    return(nb);
  }
```

Defines:
  request_neighbour, used in chunks 31b and 32a.
Uses direction 29b, message_wfr 17c, root_write_address 8a, root_write_data 9c, and RPARH 14b.

At the moment, only the directions "West", "North", "East", "South" and "Bus" are supported.

29b ⟨*HF: Typedefs* 7a⟩+≡ (2b) ◁7a

```
  /** Direction from current processor to another */
  enum direction {West, South, East, North, Bus};
  typedef enum direction direction;
```

Defines:
  direction, used in chunks 29a, 30e, 34d, 35c, 38d, 92b, and 94a.

In order to use **request_neighbour** for scanning the network, a *routing table* is defined. This is a twodimensional array of integers where the *graph* can be stored.

30a ⟨*HF: Defines* 24a⟩+≡ (2b) ◁24a 30d▷

```
/** Maximum number of nodes in the network */
#define MAX_NODES      256
/** Maximum number of directions for each processor */
#define MAX_DIR        5
```

Defines:
MAX_DIR, used in chunks 30–33.
MAX_NODES, used in chunks 30 and 33.

30b ⟨*Global variables* 3b⟩+≡ (1a) ◁9a 30c▷

```

/** The routing table. */
static int table[MAX_NODES][MAX_DIR];
```

Defines:
table, used in chunks 30–35, 41c, 43, 47b, 49–51, and 94d.
Uses MAX_DIR 30a and MAX_NODES 30a.

Additionally, two arrays are defined for translating between logical processor addresses and the detected physical ones.

30c ⟨*Global variables* 3b⟩+≡ (1a) ◁30b 31a▷

```

/** Array that holds the logical addresses of the processors
with the physical address as index. */
static int logical_address[MAX_NODES];
/** Array that holds the physical addresses of the processors
with the logical address as index. */
static int physical_address[MAX_NODES];
```

Defines:
logical_address, used in chunks 31–34 and 38c.
physical_address, used in chunks 32–35 and 38b.
Uses MAX_NODES 30a.

An *empty* entry in the routing table is indicated by a "-1". Another define is added for this...

30d ⟨*HF: Defines* 24a⟩+≡ (2b) ◁30a 94a▷

```
/** Value of an empty entry */
#define EMPTY          -1
```

Defines:
EMPTY, used in chunks 32–34.

Now, the connections between single processors are detected and the entries in our routing table and the *address translation* arrays are filled. The following *algorithm* assumes that **pre_init_routing_table** has been called, i.e. all entries in **logical_address** are **EMPTY**.

30e ⟨*Detect Neighbours* 30e⟩≡ (28b)

```
/** Detects all processors starting at the ''root'' and writes
* them into the routing table.
* @param rootaddress Physical address of the root processor
*/
static void detect_neighbours(int rootaddress)
{
  int i, new_node, entry_left;
  direction dir;

  ⟨dn: initialize loop 31b⟩

  do
  {
    entry_left = 0;
    for (i = 0; i < MAX_NODES; i++)
    {
      ⟨dn: has this processor not been requested yet? 32a⟩
    }
    ⟨dn: prepare next "stage" 33a⟩
  } while (entry_left == 1);

}
```

Defines:
`detect_neighbours`, used in chunk 33c.
Uses `direction` 29b, `MAX_NODES` 30a, and `table` 30b.

While inserting the table entries, the function keeps track of the number of nodes and edges in the graph...

31a ⟨*Global variables* 3b⟩+≡ (1a) ◁30c 41b▷

```
/** The number of detected edges in the network. */
static int number_of_edges;
/** The number of detected nodes in the network. */
static int number_of_nodes;
```

Defines:
`number_of_edges`, used in chunks 31b, 32b, 34a, and 36a.
`number_of_nodes`, used in chunks 32a and 34–36.

Detecting the network is done in *stages*. In each $(n+1)$-stage, the processors found in stage (n) are checked for new neighbours. A processor that was found in the last stage is marked by a "-2" in the table `logical_address`. So, initializing the loop is done by *marking* the neighbours of the root processor. In subsequent stages, these markers propagate through the array `logical_address`—i.e. through the network graph—until no new connections could be found.

31b ⟨*dn: initialize loop* 31b⟩≡ (30e)

```
/* Detect all neigbours of the root processor */
for (dir = West; dir < MAX_DIR; dir++)
{
```

```
    new_node = request_neighbour(rootaddress, dir);
    if (new_node != rootaddress)
    {
      /* New processor found, mark it */
      logical_address[new_node] = -2;
      table[rootaddress][dir] = new_node;
      /* Increase number of edges */
      number_of_edges++;
    }
  }
```

Uses **logical_address** 30c, **MAX_DIR** 30a, **number_of_edges** 31a, **request_neighbour** 29a, and **table** 30b.

If a *marked* processor is found, his logical and physical address are set. Then, his neighbours are detected.

32a ⟨*dn: has this processor not been requested yet?* 32a⟩≡ (30e)

```
  if (logical_address[i] == -2)
  {
    /* Undetected entry left */
    entry_left = 1;
    logical_address[i] = number_of_nodes;
    physical_address[number_of_nodes] = i;
    number_of_nodes++;
    /* Detect all neigbours of the actual processor */
    for (dir = West; dir < MAX_DIR; dir++)
    {
      new_node = request_neighbour(i, dir);

      if (new_node != i)
      {
        ⟨dn: set table entries 32b⟩
      }
    }
  }
```

Uses **logical_address** 30c, **MAX_DIR** 30a, **number_of_nodes** 31a, **physical_address** 30c, and **request_neighbour** 29a.

If the logical address of the found node is `EMPTY`, a new edge has been detected. *Marking* the processors needs some synchronization. The logical address can not be set to "-2" before the end of the current stage. If, for example, a connection from #57 to #161 was found, the #161 would be processed immediately, which is unwanted. Thus, the detected nodes are *earmarked* by setting their logical address to "-3".

32b ⟨*dn: set table entries* 32b⟩≡ (32a)

```
  table[i][dir] = new_node;
  if (logical_address[new_node] == EMPTY)
  {
```

```
    number_of_edges++;
    logical_address[new_node] = -3;
  }
```

Uses EMPTY 30d, logical_address 30c, number_of_edges 31a, and table 30b.

Serving as transition to the next stage, all "-3" are converted to a "-2".

33a ⟨*dn: prepare next "stage"* 33a⟩≡ (30e)

```
  for (i = 0; i < MAX_NODES; i++)
  {
    if (logical_address[i] == -3)
      logical_address[i] = -2;
  }
```

Uses logical_address 30c and MAX_NODES 30a.

The entries of the routing table are initialized to *empty* before using it.

33b ⟨*Pre-initialize Routing Table* 33b⟩≡ (28b)

```
  /** Initializes the entries of the routing table.
  */
  static void pre_init_routing_table(void)
  {
    int i, j;

    for (i = 0; i < MAX_NODES; i++)
    {
      physical_address[i] = logical_address[i] = EMPTY;
      for (j = 0; j < MAX_DIR; j++)
      {
        table[i][j] = EMPTY;
      }
    }
  }
```

Defines:
  pre_init_routing_table, used in chunk 34a.
Uses EMPTY 30d, logical_address 30c, MAX_DIR 30a, MAX_NODES 30a, physical_address 30c, and table 30b.

For scanning the complete network the *root* processor has to be detected. Then, detect_neighbours is called with its address as argument.

33c ⟨*Initialize Routing Table* 33c⟩≡ (28b)

```
  /** Detects the network and copies the informations to the internal
  * routing table.
  */
  static void init_routing_table(void)
  {
    int adr[1];
```

```
    ⟨irt: pre-initialize routing table 34a⟩
    ⟨irt: detect root processor 34c⟩

    /* Detect network */
    detect_neighbours(adr[0]);

  }
```

Defines:
init_routing_table, used in chunk 94d.
Uses detect_neighbours 30e and table 30b.

34a ⟨*irt: pre-initialize routing table* 34a⟩≡ (33c)

```
  /* Pre-initialize routing table */
  number_of_nodes = number_of_edges = 0;
  pre_init_routing_table();
```

Uses number_of_edges 31a, number_of_nodes 31a, pre_init_routing_table 33b, and table 30b.

The ID of the root processor is stored at NWADR (0x207 in DM, see [3, p. 1]).

34b ⟨*Defines: ADSP2181 (DM)* 14b⟩+≡ (14a) ◁28c 36c▷

```
  #define NWADR                              0x0207
```

Defines:
NWADR, used in chunk 34c.

34c ⟨*irt: detect root processor* 34c⟩≡ (33c)

```
  /* Get network address of the root processor */
  root_write_address(NWADR, dat);
  root_read_data(adr, 1, dat);

  /* Enter root processor in the routing table */
  logical_address[adr[0]] = 0;
  physical_address[0] = adr[0];
  number_of_nodes++;
```

Uses logical_address 30c, number_of_nodes 31a, NWADR 34b, physical_address 30c, root_read_data 11c, root_write_address 8a, and table 30b.

Now, the routing table is complete and its contents can be printed to stdout.

34d ⟨*Display Single Neighbour* 34d⟩≡ (28b)

```
  /** Displays the physical address of the neighbour processor in the given
  * direction on stdout.
  * @param physical_address Physical address of a processor
  * @param dir The direction to the neighbour processor
  */
  static void display_single_neighbour(int physical_address, direction dir)
```

```
{
  if (table[physical_address][dir] == EMPTY)
    printf(" ---");
  else
    printf(" %3d", table[physical_address][dir]);

}
```

Defines:
display_single_neighbour, used in chunk 35b.
Uses direction 29b, EMPTY 30d, physical_address 30c, and table 30b.

35a ⟨*Display Routing Table* 35a⟩≡ (28b)

```
/** Displays the entries of the internal routing table on stdout.
*/
void display_routing_table(void)
{
  int i, phys_i;

  printf("                                                W   S   E   N   B\n");
  for (i = 0; i < number_of_nodes; i++)
  {
    phys_i = physical_address[i];
    printf("Processor %3d (%3d) is directly connected with", phys_i, i);

    ⟨drt: display neighbours 35b⟩

    printf("\n");

  }
}
```

Defines:
display_routing_table, used in chunks 38d and 96b.
Uses number_of_nodes 31a, physical_address 30c, and table 30b.

35b ⟨*drt: display neighbours* 35b⟩≡ (35a)

```
display_single_neighbour(phys_i, West);
display_single_neighbour(phys_i, South);
display_single_neighbour(phys_i, East);
display_single_neighbour(phys_i, North);
display_single_neighbour(phys_i, Bus);
```

Uses display_single_neighbour 34d.

The neighbour of a processor in the given direction is returned to the user by get_neighbour.

35c ⟨*Get Neighbour* 35c⟩≡ (28b)

```
/** Gets the physical address of a neighbour processor in the given direction.
* @param processor The physical address of a processor
* @param dir The direction to a neighbour processor
* @return The physical address of the neighbour processor
*/
int get_neighbour(int processor, direction dir)
{
        return(table[processor][dir]);
}
```

Defines:
`get_neighbour`, used in chunk 38d.
Uses `direction` 29b and `table` 30b.

The earlier defined internal variables `number_of_nodes` and `number_of_edges` can be fetched by the two following functions.

36a ⟨*Get Number of Edges* 36a⟩≡ (28b)

```
/** Returns the number of network edges.
* @return The number of edges
*/
int get_number_of_edges(void)
{
        return(number_of_edges);
}
```

Defines:
`get_number_of_edges`, used in chunk 38d.
Uses `number_of_edges` 31a.

36b ⟨*Get Number of Nodes* 36b⟩≡ (28b)

```
/** Returns the number of network processors (nodes).
* @return The number of processors
*/
int get_number_of_nodes(void)
{
        return(number_of_nodes);
}
```

Defines:
`get_number_of_nodes`, used in chunk 38d.
Uses `number_of_nodes` 31a.

The function `join_group` helps in defining groups of processors. It sets the variable `OWNGRP` at `0x0206` in DM to the given group number (see [3, p. 1]).

36c ⟨*Defines: ADSP2181 (DM)* 14b⟩+≡ (14a) ◁34b

```
#define OWNGRP                          0x0206
```

Defines:
`OWNGRP`, used in chunk 37.

37a ⟨*Join Group* 37a⟩≡ (28b)

```
/** Sets the variable OWNGRP (0x206 in DM) for processor
[[netaddress]] to the given [[group]].
@param netaddress Processor that wants to join the group
@param group Group number
@return ERROR if an error occurs, OK else
*/
int join_group(int netaddress, int group)
{
  int data;

  ⟨jg: check group number 37b⟩
  ⟨jg: set new group number 37c⟩

}
```

Defines:
join_group, used in chunk 37.
Uses error 5a and OWNGRP 36c.

37b ⟨*jg: check group number* 37b⟩≡ (37a)

```
/* Valid group number? */
if ((group > 63) || (group < 0))
{
  error("<join_group> : Invalid group number! (Must be 0-63)");
  return(ERROR);
}
```

Uses error 5a and join_group 37a.

The user specifies group numbers in the range 0–63. Internally, the Fifth protocol handles groups as normal processor addresses, but in the range 0x100–0x13F. So, for a broadcast to group `0x2A` the message is sent to the network address `0x12A`. This is why the given group number is `OR`ed with `0x100` before writing it to `OWNGRP`.

37c ⟨*jg: set new group number* 37c⟩≡ (37a)

```
data = (group | 0x100);

return(broadcast_memory(SINGLE, netaddress, OWNGRP,
                        dat, &data, 1));
```

Uses broadcast_memory 25d 71a, OWNGRP 36c, and SINGLE 24a.

37d ⟨*HF: Function prototypes* 4b⟩+≡ (2b) ◁28a 38d▷

```
extern int join_group(int, int);
```

Uses join_group 37a.

Arrays for the translation between logical and physical processor addresses are already defined. In order to make them available for the user, two appropriate functions are provided...

38a ⟨*Translating Addresses* 38a⟩≡ (3a)
⟨*Get Physical Address* 38b⟩
⟨*Get Logical Address* 38c⟩

38b ⟨*Get Physical Address* 38b⟩≡ (38a)

```
/** Converts the logical address to the physical address of a processor.
* @param logical The logical address of a processor
* @return The physical address of the processor
*/
int get_physical_address(int logical)
{
        return(physical_address[logical]);
}
```

Defines:
get_physical_address, used in chunk 38d.
Uses physical_address 30c.

38c ⟨*Get Logical Address* 38c⟩≡ (38a)

```
/** Converts the physical address to the logical address of a processor.
* @param physical The physical address of a processor
* @return The logical address of the processor
*/
int get_logical_address(int physical)
{
        return(logical_address[physical]);
}
```

Defines:
get_logical_address, used in chunk 38d.
Uses logical_address 30c.

38d ⟨*HF: Function prototypes* 4b⟩+≡ (2b) ◁37d 52a▷

```
extern void display_routing_table(void);
extern int get_neighbour(int, direction);
extern int get_number_of_edges(void);
extern int get_number_of_nodes(void);
extern int get_physical_address(int);
extern int get_logical_address(int);
```

Uses direction 29b, display_routing_table 35a, get_logical_address 38c, get_neighbour 35c, get_number_of_edges 36a, get_number_of_nodes 36b, and get_physical_address 38b.

3.5 Loading programs

If one uses the AnalogDevices Assembler `asm21`, an executable file is output. Consisting of simple ASCII text lines, it tells which data or program code should go to which memory location. For example, the small program

```
.MODULE LED_BLINK;
.VAR/DM/RAM/ABS=0x2000 dataflag;
.INIT dataflag: 0;
.ENTRY start_blink;

start_blink:      toggle fl0;
                  CNTR = 1000;
                  DO loop1 UNTIL CE;
loop1:            NOP;
                  toggle fl0;
                  CNTR = 1000;
                  DO loop2 UNTIL CE;
loop2:            NOP;
                  toggle fl0;
                  CNTR = 1000;
                  DO loop3 UNTIL CE;
loop3:            NOP;
                  toggle fl0;
                  RTS;
.ENDMOD
```

results in the executable file

```
IIi
@PA
1000
02004F
3E7105
15003E
000000
02004F
3E7105
15007E
000000
02004F
3E7105
1500BE
000000
02004F
0A000F
#123010C65D4
@DA
2000
0000
#12300002000
```

IIo

.

Quickly, one discovers the simple structure of these files. They consist of *blocks*—either for data or program memory—supplied with an address. The program memory blocks contain ADSP-2181 instructions in hexadecimal format, whereas data memory blocks care about the initialization of variables.
Wouldn't it be nice to have a function for loading these *ASCII executables* to our processors directly?
Of course, but first a few words have to be lost about the interrupt vector table in connection with the Fifth protocol:
This Fifth protocol already uses the addresses `0x0000`–`0x1000` in DM and PM. So programs have to start at `0x1000` in PM, right? Not exactly, but quite. In the Fifth protocol the range `0x0FD8`–`0x0FFF` is reserved for the interrupt vector table (IVT) which is 28 words long for the ADSP-2181. Hence, programs that want to use interrupts should begin at `0x0FD8` with the IVT and then continue with the first real instruction for the DSP at `0x1000` in PM.
An appropriate architecture description (`*.sys`) for the 2181 on the ER2 looks like this:

```
.SYSTEM auto;
.ADSP2181;
.MMAP0;
.SEG/PM/RAM/ABS=0x0FD8/CODE/DATA     int_pm[0x3024];
.SEG/DM/RAM/ABS=0x1000/DATA          int_dpm[0x2FFF];
.ENDSYS;
```

This file has to be translated into a `*.ach` file by the program `bld21`. Let's assume the file is named `er2.sys`. Then call

```
bld21 er2
```

.

Now, the new architecture file `er2.ach` can be used while generating an executable—say `test.exe`—with the commands:

```
asm21 test
ld21 test.obj -a er2 -e test.exe
```

But what if one does not want to use any interrupts at all in a program or want to leave some of them untouched? Well, for the *read an interrupt vector table* function—which is about to be declared pretty soon—the following holds:
If an entry in the IVT of the program is a `NOP` (*no operation*) instruction, this `NOP` is not written to the DSP, i.e. the interrupt vector entry in the 2181 remains unchanged!
So, if the interrupts are not to be changed, an IVT consisting of `NOP`s only should be provided.
Looks like everything can be put together for making it work. Unfortunately, there is a little more to come regarding the executables created by the `g21` C compiler...
For reading an executable file—produced by the `asm21` assembler—into the memory of an ADSP-2181, single blocks of data are read while determining

the address. If an interrupt vector table is detected, the NOP entries have to be skipped.

41a ⟨*Loading Programs* 41a⟩≡ (3a)
  ⟨*Read Interrupt* 41c⟩
  ⟨*Read Block* 44a⟩
  ⟨*Read Program* 45c⟩
  ⟨*Scan Interrupt* 49⟩
  ⟨*Read C Program* 50c⟩

Because the device driver should be used in an efficient way by transferring whole code pieces *en bloc*, a global array is allocated as buffer...

41b ⟨*Global variables* 3b⟩+≡ (1a) ◁31a 42d▷

```
/** Array as buffer for code pieces */
static int prg[0x2fff];
```

Defines:
  prg, used in chunks 42–44.

read_interrupt assumes that the executable file is opened and an interrupt vector table was detected at the actual position within the file.

41c ⟨*Read Interrupt* 41c⟩≡ (41a)

```
/** Reads the detected interrupt vector table from the already opened
* file into the program memory of a single processor or a
* group of processors.
* @param stream Pointer to the opened file
* @param filename Pointer to the name of the file
* @param lineptr Pointer to a line buffer
* @param groupnr The group number
* @param netaddress Physical address of the processor if groupnr is SINGLE
* @return ERROR if an error occurs, OK else
*/
static int read_interrupt(FILE *stream, char *filename,
                          char *lineptr, int groupnr, int netaddress)
{
  int i=0, j, length;
  char *tailptr;

  ⟨ri: read table 42b⟩
  ⟨ri: fill with zeros 42c⟩
  ⟨ri: check table length 43a⟩
  ⟨ri: info 43b⟩
  ⟨ri: load program to processor 43c⟩

  return(OK);
}
```

Defines:

read_interrupt, used in chunks 43a and 47b.
Uses error 5a, file 2c, SINGLE 24a, and table 30b.

While reading the IVT its length INT_LENGTH is checked. All text lines are required to have a maximum length of EXE_LINE_LEN.

42a ⟨*Defines* 8c⟩+≡ (1a) ◃17b 69b▹

```
/** Starting address of the IVT */
#define INT_BASE        0x0FD8
/** Length of the IVT */
#define INT_LENGTH      0x0028
/** Maximum length of a text line */
#define EXE_LINE_LEN    15
```

Defines:
EXE_LINE_LEN, used in chunks 42b, 44b, 45c, 47a, 48c, 50, and 51a.
INT_BASE, used in chunks 43c and 47.
INT_LENGTH, used in chunks 42b, 43a, and 47c.

42b ⟨*ri: read table* 42b⟩≡ (41c)

```
fgets(lineptr, EXE_LINE_LEN, stream);
while ((i < INT_LENGTH) && (feof(stream) == 0))
{
  prg[i] = strtol(lineptr, &tailptr, 16);

  /* Read the new line */
  fgets(lineptr, EXE_LINE_LEN, stream);
  i++;
}
```

Uses EXE_LINE_LEN 42a, INT_LENGTH 42a, and prg 41b.

42c ⟨*ri: fill with zeros* 42c⟩≡ (41c)

```
/* Fill up the rest of the interrupt vector with zeros if necessary */
if ((i % 4) != 0)
{
  j = 4 - (i % 4);
  i += j;
  for (; j > 0; j--)
    prg[i-j-1] = 0;
}
```

Uses prg 41b.

Sometimes the output of arguments is needed while displaying information or an error. For this purpose a static array of chars is defined that can be used in connection with sprintf.

42d ⟨*Global variables* 3b⟩+≡ (1a) ◁41b 57a▷

```
/** Char array for info and error messages */
static char msg[150];
```

Defines:
`msg`, used in chunks 43–50, 70d, 76–78, and 88c.
Uses `error` 5a and `info` 5a.

43a ⟨*ri: check table length* 43a⟩≡ (41c)

```
/* Length of the interrupt vector table */
if (i != INT_LENGTH)
{
  sprintf(msg, "<read_interrupt> : Invalid length %d of the interrupt\
          vector table in file %s! Should be %d.", i, filename, INT_LENGTH);
  error(msg);
  return(ERROR);
}
```

Uses `error` 5a, `file` 2c, `INT_LENGTH` 42a, `msg` 42d, `read_interrupt` 41c, and `table` 30b.

43b ⟨*ri: info* 43b⟩≡ (41c)

```
if (groupnr == SINGLE)
{
  sprintf(msg, "Load interrupt vector table to processor 0x%X.", netaddress);
  info(msg);
}
else
{
  sprintf(msg, "Load interrupt vector table to processors with group\
          number 0x%X.", groupnr);
  info(msg);
}
```

Uses `info` 5a, `msg` 42d, `SINGLE` 24a, and `table` 30b.

43c ⟨*ri: load program to processor* 43c⟩≡ (41c)

```
/* Load program to processor */
for (j = 0; j < i/4; j++)
{
  if (prg[4*j] != 0 || prg[4*j+1] != 0 || prg[4*j+2] != 0 || prg[4*j+3] != 0)
  {
    sprintf(msg, "%X: %x %x %x %x", 4*j + INT_BASE, prg[4*j], prg[4*j+1],
            prg[4*j+2], prg[4*j+3]);
    info(msg);
    if (broadcast_memory(groupnr, netaddress, 4*j + INT_BASE, prog,
        prg+4*j, 4) == ERROR)
      return(ERROR);
```

```
    }
  }
```

Uses **broadcast_memory** 25d 71a, **info** 5a, **INT_BASE** 42a, **msg** 42d, and **prg** 41b.

If a memory *block* is read it simply gets transferred to the specified address in PM or DM.

44a ⟨*Read Block* 44a⟩≡ (41a)

```
/** Reads a detected block of data from the already opened
* file into the memory of a single processor or a
* group of processors.
* @param stream Pointer to the opened file
* @param filename Pointer to the name of the file
* @param lineptr Pointer to a line buffer
* @param groupnr The group number
* @param netaddress Physical address of the processor if groupnr is SINGLE
* @param address Logical address in the program/data memory
* @param mem_class Type of memory (prog/dat)
* @return ERROR if an error occurs, OK else
*/
static int read_block(FILE *stream, char *filename, char *lineptr,
               int groupnr, int netaddress,
               int address, memory_class mem_class)
{
  int i=0, j, length;
  char *tailptr;

  sprintf(msg, "<rb>: g: %d n: %d a: %X",
          groupnr, netaddress, address);
  info(msg);

  ⟨rb: read block 44b⟩
  ⟨rb: check end sequence 45a⟩

  length = i;

  ⟨rb: info 45b⟩

  /* Load data to processor(s) */
  return(broadcast_memory(groupnr, netaddress, address,
                          mem_class, prg, length));
}
```

Defines:
read_block, used in chunks 45a, 48a, and 51b.
Uses **broadcast_memory** 25d 71a, **error** 5a, **file** 2c, **info** 5a, **memory_class** 7a, **msg** 42d, **prg** 41b, and **SINGLE** 24a.

44b ⟨*rb: read block* 44b⟩≡ (44a)

```
fgets(lineptr, EXE_LINE_LEN, stream);
while ((feof(stream) == 0) && (lineptr[0] != '#'))
{
  prg[i] = strtol(lineptr, &tailptr, 16);

  /* Read new line */
  i++;
  fgets(lineptr, EXE_LINE_LEN, stream);
}
```

Uses EXE_LINE_LEN 42a and prg 41b.

45a ⟨*rb: check end sequence* 45a⟩≡ (44a)

```
/* Check for end sequence (= #) */
if (lineptr[0] != '#')
{
  sprintf(msg, "<read_block> : Missing end sequence of data block in\
          file %s. Should be '#\\n'.", filename);
  error(msg);
  return(ERROR);
}
```

Uses error 5a, file 2c, msg 42d, and read_block 44a.

45b ⟨*rb: info* 45b⟩≡ (44a)

```
if (groupnr != SINGLE)
{
  sprintf(msg, "Loading data to the %s memory of the processors with\
          group number %d address: 0x%X length: 0x%X",
          (mem_class == prog)? "program": "data", groupnr, address, length);
  info(msg);
}
else
{
  sprintf(msg, "Loading data to the %s memory of processor 0x%X\
          address: 0x%X length: 0x%X", (mem_class == prog)? "program": "data",
          netaddress, address, length);
  info(msg);
}
```

Uses info 5a, msg 42d, and SINGLE 24a.

Now, a complete executable is parsed, detecting single data blocks and extracting their start address and memory type.

45c ⟨*Read Program* 45c⟩≡ (41a)

```
/** Reads an ADSP-2181 program --- produced by the assembler ---
* from a file into the memory of a single processor or a
* group of processors.
* @param filename Pointer to the name of the file
* @param groupnr The group number
* @param netaddress Physical address of the processor if groupnr is SINGLE
* @return ERROR if an error occurs, OK else
*/
int read_program(char *filename, int groupnr, int netaddress)
{
  FILE *stream;
  long int address = 1;
  char lineptr[EXE_LINE_LEN];
  char *tailptr;
  memory_class typ;

  ⟨rp: open file 46a⟩
  ⟨rp: check start sequence 46b⟩
  ⟨rp: read data blocks 47a⟩
  ⟨rp: check end sequence 48d⟩

  return(OK);
}
```

Defines:
read_program, used in chunks 46–48 and 52a.
Uses error 5a, EXE_LINE_LEN 42a, file 2c, memory_class 7a, and SINGLE 24a.

46a ⟨*rp: open file* 46a⟩≡ (45c 50c)

```
if ((stream = fopen(filename, "r")) == NULL)
{
   sprintf(msg, "<read_program> : Could not open file %s!", filename);
   error(msg);
   return(ERROR);
}
```

Uses error 5a, file 2c, msg 42d, and read_program 45c.

46b ⟨*rp: check start sequence* 46b⟩≡ (45c 50c)

```
/* Detect ESCESCi as start sequence */
if (fscanf(stream, "\033\033i\n") < 0)
{
  sprintf(msg, "<read_program> : Invalid start sequence in file %s!\
          Should be \033\033i.", filename);
  error(msg);
  return(ERROR);
}
```

Uses **error** 5a, **file** 2c, **msg** 42d, and **read_program** 45c.

47a ⟨*rp: read data blocks* 47a⟩≡ (45c)

```
fgets(lineptr, EXE_LINE_LEN, stream);
while ((feof(stream) == 0) && (strncmp(lineptr, "\033\033o",3) != 0))
{
  ⟨rp: detect type of memory 48b⟩
  ⟨rp: get start address 48c⟩

  if ((address == INT_BASE) && (typ == prog))
  {
    ⟨rp: load interrupt table 47b⟩
  }
  else
  {
    ⟨rp: check start address 47c⟩
    ⟨rp: load memory block 48a⟩
  }

  /* Read next line */
  fgets(lineptr, EXE_LINE_LEN, stream);
}
```

Uses **EXE_LINE_LEN** 42a and **INT_BASE** 42a.

47b ⟨*rp: load interrupt table* 47b⟩≡ (47a)

```
/* Load interrupt vector table */
if (read_interrupt(stream, filename, lineptr,
                    groupnr, netaddress) == ERROR)
  return(ERROR);
```

Uses **read_interrupt** 41c and **table** 30b.

47c ⟨*rp: check start address* 47c⟩≡ (47a)

```
if (address < INT_BASE+INT_LENGTH)
{
  sprintf(msg, "<read_program> : Address 0x%X of the program in\
          file %s should be 0x%X at least!", address, filename,
          INT_BASE+INT_LENGTH);
  error(msg);
  return(ERROR);
}
```

Uses **error** 5a, **file** 2c, **INT_BASE** 42a, **INT_LENGTH** 42a, **msg** 42d, and **read_program** 45c.

48a ⟨*rp: load memory block* 48a⟩≡ (47a)

```
/* Read data block to memory */
if (read_block(stream, filename, lineptr,
               groupnr, netaddress, address, typ) == ERROR)
  return(ERROR);
```

Uses **read_block** 44a.

48b ⟨*rp: detect type of memory* 48b⟩≡ (47a 51a)

```
/* Detect type of memory (data/program) */
if ((strncmp(lineptr, "@DA", 3) != 0) && (strncmp(lineptr, "@PA", 3) != 0))
{
  sprintf(msg, "<read_program> : Invalid type of memory in\
          file %s! Should be @DA or @PA.", filename);
  error(msg);
  return(ERROR);
}

typ = (lineptr[1] == 'D')? dat: prog;
```

Uses **error** 5a, **file** 2c, **msg** 42d, and **read_program** 45c.

48c ⟨*rp: get start address* 48c⟩≡ (47a 51a)

```
/* Read address */
fgets(lineptr, EXE_LINE_LEN, stream);
if (feof(stream) != 0)
{
  sprintf(msg, "<read_program> : Unexpected end of file %s while\
          reading address line!", filename);
  error(msg);
  return(ERROR);
}

/* Calculate start address */
address = strtol(lineptr, &tailptr, 16);
```

Uses **error** 5a, **EXE_LINE_LEN** 42a, **file** 2c, **msg** 42d, and **read_program** 45c.

48d ⟨*rp: check end sequence* 48d⟩≡ (45c 50c)

```
/* Detect end sequence ESCESCo */
if (strncmp(lineptr, "\033\033o",3) != 0)
{
  sprintf(msg, "<read_program> : Invalid end sequence in file %s!\
          Should be \033\033o.", filename);
  error(msg);
  return(ERROR);
```

```
}
```

Uses **error** 5a, **file** 2c, **msg** 42d, and **read_program** 45c.

As already mentioned, the `g21` C compiler does not only provide a nice platform for developing *advanced* DSP programs but also confronts the *ER2 user* with some intricacies.
The main problem is, that it wants to write an interrupt vector table at `0x0000` in PM. If one specifies an architecture file without enough space for an IVT at this place, the compiler complains and refuses further work. No compiler option seems to help in this situation.
Hence, this architecture file `er2c.sys` should be used

```
.SYSTEM auto;
.ADSP2181;
.MMAP0;
.SEG/PM/RAM/ABS=0x0000/CODE/DATA     int_pmi[0x0030];
.SEG/PM/RAM/ABS=0x1000/CODE/DATA     int_pm[0x2FFF];
.SEG/DM/RAM/ABS=0x1000/DATA          int_dm[0x2FFF];
.ENDSYS;
```

for creating ADSP-2181 programs with the `g21` compiler by issueing the command

```
g21 <C file> [DSP files] -o <output file> -a er2c -v -map
```

Then, while reading the executable, the IVT at `0x0000` is detected and simply skipped. If interrupts shall be used, they have to be initialized within the program itself by appropriate assembler commands using the `asm` function.
But there is even more work to do. In general, programs from the assembler `asm21` start at `0x1000` in PM due to the architecture file. The C compiler `g21` includes several header files, library routines for floating point arithmetic a.s.o., to the programs and rearranges the single pieces of code in a, more or less, arbitrary manner. The starting point of the executable is not bound to `0x1000` in PM anymore. Thus, it is necessary to extract the start address from the interrupt vector `RESET` (`IRQ #0`) of the IVT.

49 ⟨*Scan Interrupt* 49⟩≡ (41a)

```
/** Skips the detected interrupt vector table in the already opened
* file and extracts the start address of the program.
* @param stream Pointer to the opened file
* @param filename Pointer to the name of the file
* @param lineptr Pointer to a line buffer
* @param start Pointer to the start address
* @return ERROR if an error occurs, OK else
*/
static int scan_interrupt(FILE *stream, char *filename,
                          char *lineptr, int *start)
{
```

```
  int i = 0;
  char **tailptr = NULL;

  ⟨si: skip IVT and extract start address 50a⟩
  ⟨si: check for end sequence 50b⟩

  sprintf(msg, "The interrupt vector table is skipped. Extracted\
          start address of the program: 0x%X", *start);
  info(msg);

  return(OK);
}
```

Defines:
  scan_interrupt, used in chunks 50b and 51b.
Uses **error** 5a, **file** 2c, **info** 5a, **msg** 42d, and **table** 30b.

50a ⟨*si: skip IVT and extract start address* 50a⟩≡ (49)

```
fgets(lineptr, EXE_LINE_LEN, stream);
lineptr[strlen(lineptr) - 2] = 0;
while ((feof(stream) == 0) && (lineptr[0] != '#'))
{
  if (i == 1)
    *start = (int) ((strtol(lineptr, tailptr, 16) >> 4 ) & 0x03FFF);

  /* Read new line */
  i++;
  fgets(lineptr, EXE_LINE_LEN, stream);
  lineptr[strlen(lineptr) - 2] = 0;
}
```

Uses **EXE_LINE_LEN** 42a.

50b ⟨*si: check for end sequence* 50b⟩≡ (49)

```
/* Check for end sequence */
if (lineptr[0] != '#')
{
  sprintf(msg, "<scan_interrupt> : Unexpected end of file %s while\
          reading interrupt vector table!", filename);
  error(msg);
  return(ERROR);
}
```

Uses **error** 5a, **file** 2c, **msg** 42d, **scan_interrupt** 49, and **table** 30b.

So, while reading a C program, the IVT is skipped. The user has to ensure that the program is started at the correct address, returned in `start`.

50c ⟨*Read C Program* 50c⟩≡ (41a)

```
/** Reads an ADSP-2181 program --- produced by the C compiler ---
* from a file into the memory of a single processor or a
* group of processors. The interrupt vector table is skipped
* and the start address of the program is extracted.
* @param netaddress Physical address of the processor if groupnr is SINGLE
* @param start Pointer to the start address
* @return ERROR if an error occurs, OK else
*/
int read_program_c(char *filename, int groupnr,
                   int netaddress, int *start)
{
  FILE *stream;
  long int address = 1;
  char lineptr[EXE_LINE_LEN];
  char *tailptr;
  memory_class typ;

  ⟨rp: open file 46a⟩
  ⟨rp: check start sequence 46b⟩
  ⟨rpc: read data blocks 51a⟩
  ⟨rp: check end sequence 48d⟩

  return(OK);
}
```

Defines:
  read_program_c, used in chunk 52a.
Uses error 5a, EXE_LINE_LEN 42a, file 2c, memory_class 7a, SINGLE 24a, and table 30b.

51a ⟨*rpc: read data blocks* 51a⟩≡ (50c)

```
fgets(lineptr, EXE_LINE_LEN, stream);
while ((feof(stream) == 0) && (strncmp(lineptr, "\033\033o",3) != 0))
{
  ⟨rp: detect type of memory 48b⟩
  ⟨rp: get start address 48c⟩

  ⟨rpc: read block or skip IVT 51b⟩

  /* Read next line */
  fgets(lineptr, EXE_LINE_LEN, stream);

}
```

Uses EXE_LINE_LEN 42a.

51b ⟨*rpc: read block or skip IVT* 51b⟩≡ (51a)

```
if ((address == 0) && (typ == prog))
```

```
{
  /* Skip interrupt vector table and extract start address */
  if (scan_interrupt(stream, filename, lineptr, start) == ERROR)
    return(ERROR);
}
else
{
  /* Read data block */
  if (read_block(stream, filename, lineptr,
                 groupnr, netaddress, address, typ) == ERROR)
    return(ERROR);
}
```

Uses **read_block** 44a, **scan_interrupt** 49, and **table** 30b.

52a ⟨*HF: Function prototypes* 4b⟩+≡ (2b) ◁38d 54a▷

```
extern int read_program(char *, int, int);
extern int read_program_c(char *, int, int, int *);
```

Uses **read_program** 45c and **read_program_c** 50c.

3.6 Starting programs

All programs loaded and ready to run? Let's start them!

52b ⟨*Executing Programs* 52b⟩≡ (3a)
⟨*Root Start Program* 52d⟩
⟨*Broadcast Start Program* 53c⟩

Starting a loaded program on the root processor is done by writing the start address to the memory location `0x02E` in `PM` also called `KVP` (see [3, p. 4]).

52c ⟨*Defines: ADSP2181 (PM)* 52c⟩≡ (14a)

```
/* Memory locations in ADSP2181 Program Memory */
#define KVP                             0x002E
```

Defines:
KVP, used in chunk 53.

52d ⟨*Root Start Program* 52d⟩≡ (52b)

```
/** Starts an already loaded program on the root processor.
* @param start_address The start address of the program
*/
void root_start_program(int start_address)
{

  int data = 0;
```

```
    ⟨rsp: write parameters 53a⟩
    ⟨rsp: write start address to KVP 53b⟩

  }
```

Defines:
`root_start_program`, used in chunk 54a.

53a ⟨*rsp: write parameters* 53a⟩≡ (52d)

```
  /* Write parameters */
  root_write_address(0xff, dat);
  root_write_data(&data, 1, dat);
```

Uses `root_write_address` 8a and `root_write_data` 9c.

53b ⟨*rsp: write start address to KVP* 53b⟩≡ (52d)

```
  /* Write KVP */
  root_write_address(KVP, prog);
  start_address &= 0xFFFF;
  data = (start_address << 8) + 0xff;
  root_write_data(&data, 1, prog);
```

Uses `KVP` 52c, `root_write_address` 8a, and `root_write_data` 9c.

53c ⟨*Broadcast Start Program* 53c⟩≡ (52b)

```
/** Starts an already loaded program
* on a single processor or a group of processors.
* @param groupnr The group number
* @param netaddress Physical address of the processor if groupnr is SINGLE
* @param start_address Start address in program memory
* @param par_adr Address for the parameters (should be 0xFF)
* @return ERROR if an error occurs, OK else
*/
int broadcast_start_program(int groupnr, int netaddress,
                            int start_address, int par_adr)
{
  int data[6];

  data[0] = ((start_address & 0xFFFF) << 8) + par_adr;
  broadcast_memory(groupnr, netaddress, KVP, prog, data, 1);

  return(OK);
}
```

Defines:
`broadcast_start_program`, used in chunks 54a, 73a, and 87b.
Uses `broadcast_memory` 25d 71a, `error` 5a, `KVP` 52c, and `SINGLE` 24a.

54a ⟨*HF: Function prototypes* 4b⟩+≡ (2b) ◁52a 65a▷

```
extern void root_start_program(int start_address);
extern int broadcast_start_program(int groupnr,
                                   int netaddress,
                                   int start_address,
                                   int par_adr);
```

Uses `broadcast_start_program` 53c and `root_start_program` 52d.

3.7 Configuring single connections

The crossbar switches can be used to connect ports of two ADSP-2181 processors e.g. for serial communication. In the following, connections between two distant modules are viewed as *edges* of a *communication graph*—regardless of their width in number of bits. For embedding this graph into the *crossbar switch structure* of the ER2, an *edge* may have to cross several intermediate processors, where *pass-through* connections have to be defined. These *subparts* of an *edge* are called *links* from now on.

A special *message* is provided, configuring a single *link* for a 3-bit wide serial communication *edge* (see [3, p. 6]). Support for this *link message* is not included in the current version of this library.

Instead, a different, more basic, approach is selected that enables the user to switch 1-bit connection *edges*. Several of these can be combined to 3-bit serial communication *paths* (or even 6-bit for the SHARCs) again. Additionally, the single connections can be tested.

By inserting modules into an ER2 backplane, they are mechanically connected to their *neighbour's pins* as shown in figure 1, which was derived from [4]. The small hexadecimal numbers denote the accessible crossbar ports and their antipodes in the four directions.

3.7.1 Storing *links* and *edges*

Since a *testing facility* for the made connections shall be provided, all information about the *communication graph* is stored in a linear list of type `conf_data`. Each processor gets an entry, consisting of two additional linear lists—one for `links`, the other for `edges`.

A `link` is simply a pair of port numbers:

54b ⟨*Structs* 54b⟩≡ (1a) 56a▷

```
/** Struct that holds the crossbar link informations of the network configuration
in a linear list.
*/
struct link
{
  /** First port of the crossbar connection */
  int link_porta;
  /** Second port of the crossbar connection */
  int link_portb;
```

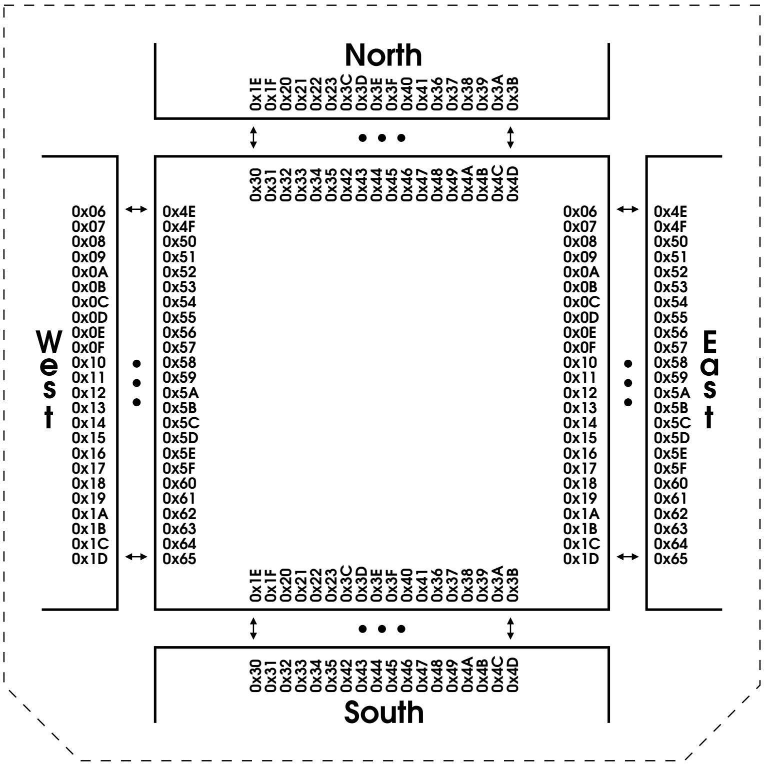

Figure 1: Available *neighbour pins* on an ADSP-2181 module

```
  /** Pointer to the next crossbar link information in the linear list */
  struct link *next_link;
};
```

An `edge` connects a `port` of the actual processor, with a distant processor `conn_proc`. For *testing*, the number of the distant port `conn_port` needs to be known, too.

56a ⟨*Structs* 54b⟩+≡ (1a) ◁54b 56b▷

```
/** Struct that holds the edge informations of the network configuration
in a linear list.
*/
struct edge
{
  /** The port of the actual processor that is connected with
  another processor in the network */
  int port;
  /** The physical address of the connected processor */
  int conn_proc;
  /** The port of the connected processor */
  int conn_port;
  /** If the connection has been tested once, this variable is set to 1 */
  int tested;
  /** Pointer to the next edge information in the linear list */
  struct edge *next_edge;
};
```

56b ⟨*Structs* 54b⟩+≡ (1a) ◁56a

```
/** Struct that holds all the informations of the network configuration
in a linear list.
*/
struct conf_data
{
  /** The physical address of a processor */
  int proc_number;
  /** The number of crossbar links for this processor */
  int no_links;
  /** The number of edges (i.e. the number of connections to other processors)
  for this processor */
  int no_edges;

  /** Pointer to the linear list of crossbar link configurations */
  struct link *links;
  /** Pointer to the linear list of edge informations */
  struct edge *edges;
  /** Pointer to the next configuration struct in the linear list */
  struct conf_data *next_data;
```

```
};
```

57a ⟨*Global variables* 3b⟩+≡ (1a) ◁42d 69c▷

```
/** The root pointer for the linear list of network configuration entries.
*/
static struct conf_data *list_root = NULL;
```

Defines:
list_root, used in chunks 57c, 59a, 60c, 62c, 66a, 70–72, and 90–93.

A few functions for adding/removing *links* and *edges* are defined, that operate on the list list_root directly.

57b ⟨*Configuring Single Connections* 57b⟩≡ (3a) 70a▷

⟨*Insert Link* 57c⟩
⟨*Insert Single Edge* 60c⟩
⟨*Insert Edge* 64b⟩
⟨*Free Links* 65b⟩
⟨*Free Edges* 65c⟩
⟨*Free Configuration List* 66a⟩

57c ⟨*Insert Link* 57c⟩≡ (57b)

```
/** Inserts a crossbar link information for the given processor
* into the linear list of configuration data.
* @param l_proc Physical address of the processor
* @param porta Port A of the crossbar link
* @param portb Port B of the crossbar link
* @return ERROR if an error occurs, OK else
*/
int insert_link(int l_proc, int porta, int portb)
{
  struct conf_data *conf_help=list_root, *conf_new;
  struct link *new_link, *help_link;


  /* Are there any entries? */
  if (conf_help != NULL)
  {
    ⟨il: list contains entries already 58a⟩
  }
  else
  {
    ⟨il: insert first processor 59a⟩
  }

  /* Enter crossbar link information */
  help_link = conf_help->links;
```

```
    if (help_link != NULL)
    {
      ⟨il: processor ''owns'' some links already 59b⟩
    }
    else
    {
      ⟨il: insert first link 60b⟩
    }

    /* Increase number of link informations */
    conf_help->no_links = conf_help->no_links + 1;

    return(OK);
  }
```

Defines:
`insert_link`, used in chunks 58–60, 65a, and 76a.
Uses `error` 5a and `list_root` 57a.

58a ⟨*il: list contains entries already* 58a⟩≡ (57c)

```
  /* Search for the processor number */
  while ((conf_help->next_data != NULL) &&
         (conf_help->proc_number != l_proc))
  {
    conf_help = conf_help->next_data;
  }

  if (conf_help->proc_number != l_proc)
  {
    ⟨il: append new processor at EOL 58b⟩
    ⟨il: link new pointer to list 58c⟩
  }
```

58b ⟨*il: append new processor at EOL* 58b⟩≡ (58a)

```
  /* Append new processor at the end of the list */
  conf_new = (struct conf_data *) malloc(sizeof(struct conf_data));
  if (conf_new == NULL)
  {
    error("<insert_link> : Could not allocate memory for conf_new pointer!");
    return(ERROR);
  }
```

Uses `error` 5a and `insert_link` 57c.

58c ⟨*il: link new pointer to list* 58c⟩≡ (58a)

```
  /* Link new pointer to the list */
  conf_help->next_data = conf_new;
```

```
conf_help = conf_new;
conf_help->next_data = NULL;
conf_help->proc_number = l_proc;
conf_help->no_links = 0;
conf_help->no_edges = 0;
conf_help->links = NULL;
conf_help->edges = NULL;
```

59a ⟨*il: insert first processor* 59a⟩≡ (57c)

```
/* Insert first processor in the list */
list_root = (struct conf_data *) malloc(sizeof(struct conf_data));
if (list_root == NULL)
{
  error("<insert_link> : Could not allocate memory for list_root pointer!");
  return(ERROR);
}

/* Initialise new pointer */
list_root->next_data = NULL;
list_root->proc_number = l_proc;
list_root->no_links = 0;
list_root->no_edges = 0;
list_root->links = NULL;
list_root->edges = NULL;

conf_help = list_root;
```

Uses **error** 5a, **insert_link** 57c, and **list_root** 57a.

59b ⟨*il: processor "owns" some links already* 59b⟩≡ (57c)

```
/* Check whether link already exists or not */
while (help_link->next_link != NULL)
{
  ⟨il: is the link already "registered"? 59c⟩

  help_link = help_link->next_link;
}

⟨il: is the link already "registered"? 59c⟩
⟨il: append new link information 60a⟩
```

59c ⟨*il: is the link already "registered"?* 59c⟩≡ (59b)

```
if (((help_link->link_porta==porta) && (help_link->link_portb==portb)) ||
    ((help_link->link_porta==portb) && (help_link->link_portb==porta)))
{
```

```
    /* Link already exists */
    return(OK);
  }
```

60a ⟨*il: append new link information* 60a⟩≡ (59b)

```
  /* Append new link information */

  /* Allocate memory for new_link pointer */
  new_link = (struct link *) malloc(sizeof(struct link));
  if (new_link == NULL)
  {
    error("<insert_link> : Could not allocate memory for new_link pointer!");
    return(ERROR);
  }

  help_link->next_link = new_link;
  help_link = new_link;
  help_link->link_porta = porta;
  help_link->link_portb = portb;
  help_link->next_link = NULL;
```

Uses **error** 5a and **insert_link** 57c.

60b ⟨*il: insert first link* 60b⟩≡ (57c)

```
  /* Insert first link information */
  /* Allocate memory for conf_help->links pointer */
  conf_help->links = (struct link *) malloc(sizeof(struct link));
  if (conf_help->links == NULL)
  {
    error("<insert_link> : Could not allocate memory for conf_help->links pointer!");
    return(ERROR);
  }

  help_link = conf_help->links;
  help_link->next_link = NULL;
  help_link->link_porta = porta;
  help_link->link_portb = portb;
```

Uses **error** 5a and **insert_link** 57c.

If a single *edge* is specified as: "Let the port `0x0D` of processor `0x45` be connected to port `0x64` of processor `0x32`.", this also implies a connection in the opposite direction. Crossbar switches work bidirectional, so for every *hardware* path one gets two *logical* connections. This is taken into account by first defining a static function that inserts a single *logical* edge into the list of configurations.

60c ⟨*Insert Single Edge* 60c⟩≡ (57b)

```
  /** Inserts a single edge information
```

```
  * into the linear list of configuration data.
  * @param l_proc Physical address of the local processor
  * @param l_port Port of the connection at the local processor
  * @param c_proc Physical address of the remote processor
  * @param c_port Port of the connection at the remote processor
  * @return ERROR if an error occurs, OK else
  */
  static int insert_single_edge(int l_proc, int l_port, int c_proc, int c_port)
  {
    struct conf_data *conf_help=list_root, *conf_new;
    struct edge *new_edge, *help_edge;

    /* Are there any entries? */
    if (conf_help != NULL)
    {
      ⟨ie: list contains entries already 61⟩
    }
    else
    {
      ⟨ie: insert first processor 62c⟩
    }

    /* Insert edge information */
    help_edge = conf_help->edges;

    if (help_edge != NULL)
    {
      ⟨ie: processor "owns" some edges already 63a⟩
    }
    else
    {
      ⟨ie: insert first edge 64a⟩
    }

    /* Increase number of edge informations */
    conf_help->no_edges = conf_help->no_edges + 1;

    return(OK);

  }
```

Defines:
`insert_single_edge`, used in chunks 62–64.
Uses `error` 5a and `list_root` 57a.

61 ⟨*ie: list contains entries already* 61⟩≡ (60c)

```
  /* Search for the processor number */
  while ((conf_help->next_data != NULL) &&
         (conf_help->proc_number != l_proc))
```

```
{
  conf_help = conf_help->next_data;
}

if (conf_help->proc_number != l_proc)
{
  ⟨ie: append new processor at EOL 62a⟩
  ⟨ie: link new pointer to list 62b⟩
}
```

62a ⟨*ie: append new processor at EOL* 62a⟩≡ (61)

```
/* Append new processor at the end of the list */
conf_new = (struct conf_data *) malloc(sizeof(struct conf_data));
if (conf_new == NULL)
{
  error("<insert_single_edge> : Could not allocate memory for conf_new pointer!");
  return(ERROR);
}
```

Uses **error** 5a and **insert_single_edge** 60c.

62b ⟨*ie: link new pointer to list* 62b⟩≡ (61)

```
/* Link new pointer to the list */
conf_help->next_data = conf_new;
conf_help = conf_new;
conf_help->next_data = NULL;
conf_help->proc_number = l_proc;
conf_help->no_links = 0;
conf_help->no_edges = 0;
conf_help->links = NULL;
conf_help->edges = NULL;
```

62c ⟨*ie: insert first processor* 62c⟩≡ (60c)

```
/* Insert first processor in the list */
list_root = (struct conf_data *) malloc(sizeof(struct conf_data));
if (list_root == NULL)
{
  error("<insert_single_edge> : Could not allocate memory for list_root pointer!");
  return(ERROR);
}

/* Initialise new pointer */
list_root->next_data = NULL;
list_root->proc_number = l_proc;
list_root->no_links = 0;
list_root->no_edges = 0;
```

```
  list_root->links = NULL;
  list_root->edges = NULL;

  conf_help = list_root;
```

Uses `error` 5a, `insert_single_edge` 60c, and `list_root` 57a.

63a ⟨*ie: processor "owns" some edges already* 63a⟩≡ (60c)

```
  /* Check whether edge already exists or not */
  while (help_edge->next_edge != NULL)
  {
    ⟨ie: is the edge already "registered"? 63b⟩

    help_edge = help_edge->next_edge;
  }

  ⟨ie: is the edge already "registered"? 63b⟩
  ⟨ie: append new edge information 63c⟩
```

63b ⟨*ie: is the edge already "registered"?* 63b⟩≡ (63a)

```
  if ((help_edge->port==l_port) && (help_edge->conn_proc==c_proc) &&
      (help_edge->conn_port==c_port))
  {
    /* Edge already exists */
    return(OK);
  }
```

63c ⟨*ie: append new edge information* 63c⟩≡ (63a)

```
  /* Insert new edge information */

  /* Allocate memory for new_edge pointer */
  new_edge = (struct edge *) malloc(sizeof(struct edge));
  if (new_edge == NULL)
  {
    error("<insert_single_edge> : Could not allocate memory for new_edge pointer!");
    return(ERROR);
  }

  help_edge->next_edge = new_edge;
  help_edge = new_edge;
  help_edge->port = l_port;
  help_edge->conn_proc = c_proc;
  help_edge->conn_port = c_port;
  help_edge->tested = 0;
  help_edge->next_edge = NULL;
```

Uses **error** 5a and **insert_single_edge** 60c.

64a ⟨*ie: insert first edge* 64a⟩≡ (60c)

```
/* Insert first edge information */
/* Allocate memory for conf_help->edges pointer */
conf_help->edges = (struct edge *) malloc(sizeof(struct edge));
if (conf_help->edges == NULL)
{
  error("<insert_single_edge> : Could not allocate memory for conf_help->edges pointer!"
  return(ERROR);
}

help_edge = conf_help->edges;
help_edge->port = l_port;
help_edge->conn_proc = c_proc;
help_edge->conn_port = c_port;
help_edge->tested = 0;
help_edge->next_edge = NULL;
```

Uses **error** 5a and **insert_single_edge** 60c.

The user can only insert *hardware* edges, so `insert_edge` calls the function `insert_single_edge` twice.

64b ⟨*Insert Edge* 64b⟩≡ (57b)

```
/** Inserts an edge information
* into the linear list of configuration data for both processors.
* @param l_proc Physical address of the local processor
* @param l_port Port of the connection at the local processor
* @param c_proc Physical address of the remote processor
* @param c_port Port of the connection at the remote processor
* @return ERROR if an error occurs, OK else
*/
int insert_edge(int l_proc, int l_port, int c_proc, int c_port)
{
  if (insert_single_edge(l_proc, l_port, c_proc, c_port) == ERROR)
  {
    return(ERROR);
  }

  if (insert_single_edge(c_proc, c_port, l_proc, l_port) == ERROR)
  {
    return(ERROR);
  }
  return(OK);
}
```

Defines:
insert_edge, used in chunks 65a and 77a.
Uses **error** 5a and **insert_single_edge** 60c.

65a ⟨*HF: Function prototypes* 4b⟩+≡ (2b) ◁54a 66b▷

```
extern int insert_link(int l_proc,
                       int porta,
                       int portb);
extern int insert_edge(int l_proc,
                       int l_port,
                       int c_proc,
                       int c_port);
```

Uses **insert_edge** 64b and **insert_link** 57c.

If the configuration data is not needed anymore, the memory should be deallocated. . .

65b ⟨*Free Links* 65b⟩≡ (57b)

```
/** Removes the linear list of crossbar link informations from the memory.
* @param link_root Pointer to the linear list
*/
static void free_links(struct link *link_root)
{
  struct link *help_link = link_root->next_link;

  while (help_link != NULL)
  {
    free(link_root);
    link_root = help_link;
    help_link = link_root->next_link;
  }

  free(link_root);
}
```

Defines:
free_links, used in chunk 66a.

65c ⟨*Free Edges* 65c⟩≡ (57b)

```
/** Removes the linear list of edge informations from the memory.
* @param link_root Pointer to the linear list
*/
static void free_edges(struct edge *edge_root)
{
  struct edge *help_edge = edge_root->next_edge;

  while (help_edge != NULL)
  {
    free(edge_root);
    edge_root = help_edge;
    help_edge = edge_root->next_edge;
```

```
    }

    free(edge_root);

  }
```

Defines:
`free_edges`, used in chunk 66a.

66a ⟨*Free Configuration List* 66a⟩≡ (57b)

```
  /** Removes the linear list of configuration data for the ER2 network
  * from the memory.
  */
  void free_configuration_list(void)
  {
    struct conf_data *help_list;

    if (list_root == NULL)
      return;

    help_list = list_root->next_data;
    while (help_list != NULL)
    {
      if (list_root->links != NULL) free_links(list_root->links);
      if (list_root->edges != NULL) free_edges(list_root->edges);
      free(list_root);
      list_root = help_list;
      help_list = list_root->next_data;
    }

    if (list_root->links != NULL) free_links(list_root->links);
    if (list_root->edges != NULL) free_edges(list_root->edges);
    free(list_root);

    list_root = NULL;
  }
```

Defines:
`free_configuration_list`, used in chunks 66b and 95.
Uses `ER2` 1b, `free_edges` 65c, `free_links` 65b, and `list_root` 57a.

66b ⟨*HF: Function prototypes* 4b⟩+≡ (2b) ◁65a 74▷

```
  extern void free_configuration_list(void);
```

Uses `free_configuration_list` 66a.

3.7.2 Connecting crossbar switch ports

A *message* for a single crossbar connection does not exist in [3]. So, how can the crossbar switch be induced to arrange the contact of two pins?

The switch is controlled by an FPGA that listens on special *IO ports* of the DSP processor ([5, p. 2]). These *ports* are memory locations in the I/O space of the ADSP-2181 and can be accessed via the `IO` assembler directive ([1, p. 15-74]). The following assembler code—using the *IO ports* `15` and `7`—provides two functions for connecting/disconnecting crossbar ports:

67a ⟨*crossbar.dsp* 67a⟩≡

```
MODULE code_for_crossbars;
{Some routines for switching/disconnecting crossbar links}

.ENTRY connect_cb;
.ENTRY disconnect_cb;
.ENTRY cb_delay;

{Connects the port AX0(HIGH) with port AX0(LOW)}
connect_cb:     IO(15) = AX0;
                CALL cb_delay;
                RTS;

{Disconnects the port AX0(HIGH) from port AX0(LOW)}
disconnect_cb:  IO(7) = AX0;
                CALL cb_delay;
                RTS;

{The delay to ensure that the (dis)connection is made}
cb_delay:       CNTR = 0x20;
                DO delay_loop UNTIL CE;
                delay_loop:     NOP;

                RTS;

.ENDMOD;
```

Defines:
`code_for_crossbars`, never used.

The added delay ensures that the FPGA gets enough cycles for his work.
The functions `connect_cb` and `disconnect_cb` can now be used to develop a more sophisticated assembler program. It takes an arbitrary number of crossbar connections and switches them automatically, once it is started.

67b ⟨*config.dsp* 67b⟩≡

```
.MODULE/RAM/ABS=0x1000  code_to_config_links;
⟨CC: Header 68a⟩
⟨CC: Variables 68b⟩
⟨CC: Entry Points 68c⟩
⟨CC: Functions 68d⟩
.ENDMOD;
```

68a ⟨*CC: Header* 68a⟩≡ (67b)

```
{Configures the crossbar links given in an array of data}
{which starts at address 0x1000 in DM.}
{Requires CROSSBAR.DSP for linking}

{Dirk Baechle, TUHH TI6, 14.05.2003}
```

The table of link ports starts at `0x1000` in DM with the number of entries. This value is initialized to zero.

68b ⟨*CC: Variables* 68b⟩≡ (67b)

```
.VAR/DM/RAM/ABS=0x1000 cblinks;

.INIT cblinks: 0;
```

The functions from `crossbar.dsp` are made known to this module...

68c ⟨*CC: Entry Points* 68c⟩≡ (67b)

```
.ENTRY conf_links;
.EXTERNAL connect_cb;
.EXTERNAL disconnect_cb;
.EXTERNAL cb_delay;
```

68d ⟨*CC: Functions* 68d⟩≡ (67b)

⟨*setup registers* 68e⟩
⟨*read number of links to configure* 68f⟩
⟨*configure links* 69a⟩

```
end_prog:       RTS;
```

The start address and the modifiers for the DAG are set up appropriately.

68e ⟨*setup registers* 68e⟩≡ (68d)

```
{set up I,M, and L registers}

conf_links:     I0 = 0x1000;  {I0 contains address of link list}
                M0 = 1;       {fill every location}
                L0 = 0;       {no circular buffer}
```

If the number of links at `0x1000` is zero the program is stopped immediately. Else, the counter register is set.

68f ⟨*read number of links to configure* 68f⟩≡ (68d)

```
{read number of links to configure}

                AX0 = DM(I0,M0);
                AY0 = 0x0000;
                AR = AX0 OR AY0;
                IF EQ JUMP end_prog;

                CNTR = AX0;
```

Now, the loop starts to step through the array. It configures the links by calling `connect_cb` until the counter expires.

69a ⟨*configure links* 69a⟩≡ (68d)

```
{configure links}

                DO conf_loop UNTIL CE;
                AX0 = DM(I0,M0);
                CALL connect_cb;
conf_loop:      NOP;
```

The resulting executable is added to the library as an array of integers:

69b ⟨*Defines* 8c⟩+≡ (1a) ◁42a 70b▷

```
/** Start address of the program ''config.exe'' */
#define CRSSCONF_START          0x1000
/** Length of the program ''config.exe'' */
#define CRSSCONF_LENGTH         23
```

Defines:
`CRSSCONF_LENGTH`, used in chunks 69c and 72a.
`CRSSCONF_START`, used in chunks 72a and 73a.

69c ⟨*Global variables* 3b⟩+≡ (1a) ◁57a 85e▷

```
/** The program ''config.exe'' for switching crossbar ports */
static const int crssconf_exe[CRSSCONF_LENGTH] = {
  0x350000, 0x340014, 0x340008, 0x600000, 0x400004,
  0x23A00F, 0x1900C0, 0x0D0C50, 0x1500BE, 0x600000,
  0x1D00DF, 0x000000, 0x0A000F, 0x0180F0, 0x1D013F,
  0x0A000F, 0x018070, 0x1D013F, 0x0A000F, 0x3C0205,
  0x15015E, 0x000000, 0x0A000F};
```

Defines:
`crssconf_exe`, used in chunk 72a.
Uses `CRSSCONF_LENGTH` 69b.

3.7.3 Configuring specified connections

70a ⟨*Configuring Single Connections* 57b⟩+≡ (3a) ◁57b 71b▷
⟨*Write Crossbar Data* 70c⟩

`write_crossbar_data` scans the linear list of the network configuration. For every found processor entry the port *pairs* are written to its memory, starting at `0x1000` in DM. The data is collected in the array `link_table` which has a fixed length of `LT_MAX`.

70b ⟨*Defines* 8c⟩+≡ (1a) ◁69b 85d▷

```
/** Maximum length of the crossbar configuration data */
#define LT_MAX 1000
```

Defines:
`LT_MAX`, used in chunk 70.

70c ⟨*Write Crossbar Data* 70c⟩≡ (70a)

```
/** Writes all crossbar informations from the linear list of configuration
* data for the ER2 network to the data memory (0x1000) of the processors.
*/
static void write_crossbar_data(void)
{
  struct conf_data *help_data = list_root;
  struct link *help_link;
  int link_table[LT_MAX];
  int table_cnt=0;

  while (help_data != NULL)
  {
    ⟨wcd: collect crossbar data for processor 70d⟩
    ⟨wcd: write data to memory 71a⟩

    /* Next processor */
    help_data = help_data->next_data;
    table_cnt = 0;
  }

}
```

Defines:
`write_crossbar_data`, used in chunks 70d and 73c.
Uses `ER2` 1b, `list_root` 57a, and `LT_MAX` 70b.

70d ⟨*wcd: collect crossbar data for processor* 70d⟩≡ (70c)

```
help_link = help_data->links;
```

```
/* Get the number of crossbar links */
link_table[table_cnt] = help_data->no_links;
table_cnt++;

/* Write crossbar links */
while (help_link != NULL)
{
  if (table_cnt == LT_MAX)
  {
    sprintf(msg, "<write_crossbar_data> : Maximum number of connections\
            was exceeded for processor %d!", help_data->proc_number);
    info(msg);
    info("<write_crossbar_data> : Not all data could be written,\
          which may cause problems!");
    help_link = NULL;
  }
  else
  {
    /* Enter in data buffer */
    link_table[table_cnt] = ((help_link->link_porta << 8) |
                             help_link->link_portb);
    help_link = help_link->next_link;
    table_cnt++;
  }
}
```

Uses **info** 5a, **LT_MAX** 70b, **msg** 42d, and **write_crossbar_data** 70c.

71a ⟨*wcd: write data to memory* 71a⟩≡ (70c)

```
/* Write the configuration data to the data memory of */
/* the processor at address 0x1000 */
broadcast_memory(SINGLE, help_data->proc_number, 0x1000,
                 dat, link_table, table_cnt);
```

Defines:
broadcast_memory, used in chunks 23b, 26–28, 37c, 43c, 44a, 53c, 72a, 86b, 88, and 89.
Uses **SINGLE** 24a.

In the next step, a function is defined for loading the configuration program `config.exe` to the processors.

71b ⟨*Configuring Single Connections* 57b⟩+≡ (3a) ◁70a 72b▷
⟨*Load Configuration Program* 71c⟩

71c ⟨*Load Configuration Program* 71c⟩≡ (71b)

```
/** Loads the program ''config.exe'' to all processors that are to be
* configured.
* @return ERROR if an error occurs, OK else
```

```
*/
static int load_configuration_programs(void)
{
  struct conf_data *help_list = list_root;

  while (help_list != NULL)
  {
    ⟨lcp: load config program to processor 72a⟩

    /* Next processor */
    help_list = help_list->next_data;
  }

  return(OK);
}
```

Defines:
load_configuration_programs, used in chunk 73c.
Uses error 5a and list_root 57a.

72a ⟨*lcp: load config program to processor* 72a⟩≡ (71c)

```
/* Load the program data to the processor */
if (broadcast_memory(SINGLE, help_list->proc_number,
    CRSSCONF_START, prog, crssconf_exe,
    CRSSCONF_LENGTH) == ERROR)
{
  return(ERROR);
}
```

Uses broadcast_memory 25d 71a, crssconf_exe 69c, CRSSCONF_LENGTH 69b, CRSSCONF_START 69b, and SINGLE 24a.

Starting all loaded programs is also required.

72b ⟨*Configuring Single Connections* 57b⟩+≡ (3a) ◁71b 73b▷
⟨*Start Configuration Program* 72c⟩

72c ⟨*Start Configuration Program* 72c⟩≡ (72b)

```
/** Starts the program config.exe on all processors
* that are to be configured.
* @return ERROR if an error occurs, OK else
*/
static int start_configuration_programs(void)
{
  struct conf_data *help_list = list_root;

  while (help_list != NULL)
  {
    ⟨scp: start program on processor 73a⟩
```

```
    /* Next processor */
    help_list = help_list->next_data;
  }

  return(OK);
}
```

Defines:
`start_configuration_programs`, used in chunk 73c.
Uses `error` 5a and `list_root` 57a.

73a ⟨*scp: start program on processor* 73a⟩≡ (72c)

```
/* Start program on processor */
if (broadcast_start_program(SINGLE, help_list->proc_number,
    CRSSCONF_START, 0xFF) == ERROR)
{
  return(ERROR);
}
```

Uses `broadcast_start_program` 53c, `CRSSCONF_START` 69b, and `SINGLE` 24a.

Finally, a function `configure_er2` is made available to the user. It combines all preceding steps for switching the defined *communication graph* automatically.

73b ⟨*Configuring Single Connections* 57b⟩+≡ (3a) ◁72b 75a▷
⟨*Configure ER2* 73c⟩

73c ⟨*Configure ER2* 73c⟩≡ (73b)

```
/** Loads the configuration program ``config.exe'' to all processors that are
* to be configured. Writes the configuration data from the linear list to
* the memories of the processors and starts the configuration program in order
* to connect the crossbar ports as needed.
* Afterwards, the switched crossbar links are ready for testing.
* @return ERROR if an error occurs, OK else
*/
int configure_er2(void)
{

  /* Load ``config.exe'' to all processors */
  info("Loading program ``config.exe'' to all processors...");
  if (load_configuration_programs() == ERROR)
  {
    return(ERROR);
  }

  /* Write configuration data to memories of the processors */
  info("Writing configuration data to all processors...");
  write_crossbar_data();
```

```
    /* Start the configuration programs */
    info("Starting the configuration programs...");
    if (start_configuration_programs() == ERROR)
    {
      return(ERROR);
    }

    info("Crossbars are (hopefully) switched now...");

    return(OK);
  }
```

Defines:
`configure_er2`, used in chunk 74.
Uses `error` 5a, `info` 5a, `load_configuration_programs` 71c, `start_configuration_programs` 72c, and `write_crossbar_data` 70c.

74 ⟨*HF: Function prototypes* 4b⟩+≡ (2b) ◁66b 79a▷

```
  extern int configure_er2(void);
```

Uses `configure_er2` 73c.

3.7.4 Reading configuration from a file

So far, the user has to provide a function—or at least some kind of *loop*—calling `insert_link` and `insert_edge` repeatedly for constructing the *communication graph*. If the configuration is static, it is desirable to relieve the user from the burden of manually setting up the *data tree* each time.
In the following *chunks*, the function `read_configuration_data` is developed. It reads a simple text file that contains the needed informations for building the *configuration tree*. The syntax for this file is:

```
configuration: list_of_blocks

list_of_blocks: block
              | block NL list_of_blocks

block: processor NL link_block NL edge_block

link_block: '0'
          | number NL links

links: link
     | link NL links

edge_block: '0'
          | number NL edges

edges: edge
```

```
    | edge NL edges

link: HexInt HexInt

edge: HexInt HexInt HexInt

HexInt: [0-F] [0-F]

processor:
number : [0-F]+
```

The terminal `NL` denotes a *newline*, so each entry has to stand on a single line.

75a ⟨*Configuring Single Connections* 57b⟩+≡ (3a) ◃73b
  ⟨*Read Links* 75b⟩
  ⟨*Read Edges* 76c⟩
  ⟨*Read Configuration Data* 78a⟩

`read_links` reads the `links` from an already opened file.

75b ⟨*Read Links* 75b⟩≡ (75a)

```
/** Reads the crossbar links to configure for a single processor from
* the already opened file into the linear list of configuration data.
* @param cfile Pointer to the already opened file
* @param filename Pointer to the name of the file
* @param processor Physical address of the processor
* @param max_links Number of crossbar links to read
* @return ERROR if an error occurs, OK else
*/
static int read_links(FILE *cfile, char *filename,
                      int processor, int max_links)
{
  int link_cnt=0, dummy=0, porta=0, portb=0;

  while ((link_cnt < max_links) && (feof(cfile) == 0))
  {
    ⟨rl: read and insert next link 76a⟩

    link_cnt++;

  }

  ⟨rl: check for EOF 76b⟩

  return(OK);

}
```

Defines:

read_links, used in chunks 76b and 78c.
Uses error 5a and file 2c.

A link specification consists of 4 hexadecimal digits—two for the first port (00-FF) and two for the second one (00-FF).

76a ⟨*rl: read and insert next link* 76a⟩≡ (75b)

```
/* Read next link line */
fscanf(cfile,"%X",&dummy);
porta = (dummy >> 8);
portb = (dummy & 255);
if (insert_link(processor, porta, portb) == ERROR)
{
  return(ERROR);
}
```

Uses insert_link 57c.

76b ⟨*rl: check for EOF* 76b⟩≡ (75b)

```
if (feof(cfile) != 0)
{
  sprintf(msg, "<read_links> : Unexpected end of file %s\
          while reading %d crossbar links for processor \
          0x%X!", filename, max_links, processor);
  error(msg);
  return(ERROR);
}
```

Uses error 5a, file 2c, msg 42d, and read_links 75b.

In an analogous fashion read_edges inserts the edges into the *tree.*

76c ⟨*Read Edges* 76c⟩≡ (75a)

```
/** Reads the edge informations for a single processor from
* the already opened file into the linear list of configuration data.
* @param cfile Pointer to the already opened file
* @param filename Pointer to the name of the file
* @param processor Physical address of the processor
* @param max_edges Number of edge informations
* @return ERROR if an error occurs, OK else
*/
static int read_edges(FILE *cfile, char *filename,
                      int processor, int max_edges)
{
  struct edge *new_edge, *root_edge, *help_edge;
  int edge_cnt=0, edge_data, l_port, c_port, c_proc;

  while ((edge_cnt < max_edges) && (feof(cfile) == 0))
  {
    ⟨re: read and insert next edge 77a⟩
```

```
      edge_cnt++;

    }

    ⟨re: check for EOF 77b⟩

    return(OK);

  }
```

Defines:
  read_edges, used in chunks 77b and 78c.
Uses **error** 5a and **file** 2c.

An *edge line* has 6 hexadecimal digits, e.g. `4F1157`. The two leftmost digits specify the remote processor. Then, the port of the local processor follows and the two rightmost digits correspond to the port of the remote crossbar switch. So, in the given example port `0x11` of the current processor should be connected to port `0x57` of the crossbar on module `0x4F`.

77a ⟨*re: read and insert next edge* 77a⟩≡ (76c)

```
  /* Read next edge information */
  fscanf(cfile,"%X",&edge_data);

  /* Bits 0-7 (LSBs) are the port of the remote processor */
  c_port = (edge_data & 255);
  edge_data = edge_data >> 8;
  /* Bits 8-15 are the port of the local processor */
  l_port = (edge_data & 255);
  /* Bits 16-31 (MSBs) is the physical address of the remote processor */
  c_proc = (edge_data >> 8);

  if (insert_edge(processor, l_port, c_proc, c_port) == ERROR)
  {
    return(ERROR);
  }
```

Uses **insert_edge** 64b.

77b ⟨*re: check for EOF* 77b⟩≡ (76c)

```
  if (feof(cfile) != 0)
  {
    sprintf(msg, "<read_edges> : Unexpected end of file %s\
            while reading %d edge informations for processor\
            0x%X!", filename, max_edges, processor);
    error(msg);
    return(ERROR);
  }
```

Uses error 5a, file 2c, msg 42d, and read_edges 76c.

78a ⟨*Read Configuration Data* 78a⟩≡ (75a)

```
/** Reads the data for the configuration of the ER2 network from a file
* into a linear list.
* @param filename Pointer to the name of the configuration file
* @return ERROR if an error occurs, OK else
*/
int read_configuration_data(char *filename)
{
  FILE *cfile;
  int processor, n_links, n_edges;

  ⟨rcd: open file 78b⟩

  /* Read the first set of data */
  fscanf(cfile,"%X",&processor);

  while (feof(cfile) == 0)
  {
    ⟨rcd: read links and edges 78c⟩
  }

  /* Close file */
  fclose(cfile);

  return(OK);
}
```

Defines:
read_configuration_data, used in chunks 78b and 79a.
Uses ER2 1b, error 5a, and file 2c.

78b ⟨*rcd: open file* 78b⟩≡ (78a)

```
/* Open file */
cfile = fopen(filename,"rb");
if (cfile == NULL)
{
  sprintf(msg, "<read_configuration_data> : Could not open file %s!", filename);
  error(msg);

  return(ERROR);
}
```

Uses error 5a, file 2c, msg 42d, and read_configuration_data 78a.

78c ⟨*rcd: read links and edges* 78c⟩≡ (78a)
```
fscanf(cfile,"%X",&n_links);

/* Read the link informations */
if (read_links(cfile, filename, processor, n_links) == ERROR)
  return(ERROR);

/* Read the number of edges */
fscanf(cfile,"%X",&n_edges);

/* Read the edge informations */
if (read_edges(cfile, filename, processor, n_edges) == ERROR)
  return(ERROR);

/* Read the next set of data */
fscanf(cfile,"%X",&processor);
```

Uses `read_edges` 76c and `read_links` 75b.

79a ⟨*HF: Function prototypes* 4b⟩+≡ (2b) ◁74 94b▷

```
extern int read_configuration_data(char *);
```

Uses `read_configuration_data` 78a.

3.8 Testing configured connections

The basic idea for *testing* a crossbar *path* is to apply a signal to one end of the connection and listen at the other...
Again, a small assembler program is developed, which uses the `FI` (Flag In) and `FO` (Flag Out) pin. They are part of the serial port SPORT1 on the ADSP-2181.

79b ⟨*test.dsp* 79b⟩≡

```
.MODULE/RAM/ABS=0x1000  code_to_test_links;
⟨TC: Header 79c⟩
⟨TC: Variables 80d⟩
⟨TC: Init Variables 80e⟩
⟨TC: Entry Points 82d⟩
⟨TC: Functions 80b⟩

end_prog:       RTS;

.ENDMOD;
```

79c ⟨*TC: Header* 79c⟩≡ (79b)

```
{Tests a switched crossbar ''edge''}
{starting at port linkno.}
```

```
{Requires CROSSBAR.DSP for linking}

{Dirk Baechle, TUHH TI6, 14.05.2003}
```

At the start of the program the serial port has to be reconfigured as *flags and interrupts*, according to [1, p. 5-7]. The `FO` pin is set to "LOW" initially.

80a ⟨*TC: Entry points* 80a⟩≡

```
.ENTRY test_links;
```

80b ⟨*TC: Functions* 80b⟩≡ (79b) 80f▷
⟨*reconfigure serial port* 80c⟩

80c ⟨*reconfigure serial port* 80c⟩≡ (80b)

```
{set up serial port as flags and interrupts}

test_links:     AR = DM(0x3FFF);
                AY0 = 0x0800;
                AR = AR OR AY0;
                AY0 = 0xFBFF;
                AR = AR AND AY0;
                DM(0x3FFF) = AR;

                {Set FlagOut pin to LOW}
                reset FO;
```

Now, the program runs in a loop. It waits until the PC wrote the necessary data into specified memory locations (DM) and signalled an *acknowledge*. If the variable `dataflag` at `0x2002` is set to "1" by the host computer, the test starts.

80d ⟨*TC: Variables* 80d⟩≡ (79b) 81b▷

```
.VAR/DM/RAM/ABS=0x2002 dataflag;
```

80e ⟨*TC: Init Variables* 80e⟩≡ (79b) 81c▷

```
.INIT dataflag: 0;
```

80f ⟨*TC: Functions* 80b⟩+≡ (79b) ◁80b 81d▷
⟨*wait for "start" acknowledge* 81a⟩

81a ⟨*wait for "start" acknowledge* 81a⟩≡ (80f)

```
{-------------- Testing links -------------}

                {Wait until PC wrote data into RAM (<=> dataflag=1)}
test_loop:      AX0 = DM(0x2002);
                AY0 = 0x0001;
                AR = AX0 AND AY0;
                IF EQ JUMP test_loop;

                {Reset dataflag}
                AX0 = 0x0000;
                DM(0x2002) = AX0;
```

The port number `linkno` is stored at `0x2000` in DM. If it is equal to 256 (`0x100`), the loop is left and the program stops.

81b ⟨*TC: Variables* 80d⟩+≡ (79b) ◁80d 81f▷

```
.VAR/DM/RAM/ABS=0x2000 linkno;
```

81c ⟨*TC: Init Variables* 80e⟩+≡ (79b) ◁80e 82a▷

```
.INIT linkno: 0;
```

81d ⟨*TC: Functions* 80b⟩+≡ (79b) ◁80f 82b▷
⟨*check port number "linkno"* 81e⟩

81e ⟨*check port number "linkno"* 81e⟩≡ (81d)

```
                {Is the port number >= 256? => abort test loop}
                AX0 = DM(0x2000);
                AY0 = 0x0100;
                AR = AX0 AND AY0;
                IF NE JUMP end_prog;
```

Else, the test continues. In order to avoid unnecessary broadcasts, the same program is run on all processors, regardless of whether they are the *sender* or the *receiver*.
The variable `inout` at `0x2001` tells whether the current processor is at the *sending* or *receiving* end.

81f ⟨*TC: Variables* 80d⟩+≡ (79b) ◁81b 83a▷

```
.VAR/DM/RAM/ABS=0x2001 inout;
```

82a ⟨*TC: Init Variables* 80e⟩+≡ (79b) ◁81c 83b▷

```
.INIT inout: 0;
```

82b ⟨*TC: Functions* 80b⟩+≡ (79b) ◁81d
⟨*"send" or "receive"?* 82c⟩
⟨*send test signal* 82e⟩
⟨*receive test signal* 84a⟩

82c ⟨*"send" or "receive"?* 82c⟩≡ (82b)

```
                reset FL0;

                {Send or receive?}
                AX0 = DM(0x2001);
                AY0 = 0x0001;
                AR = AX0 AND AY0;
                IF EQ JUMP receive_ts;
```

Sending a test signal means connecting the `FO` pin to the specified port `linkno` (`0x2000`), so the routines from `crossbar.dsp` are needed again.

82d ⟨*TC: Entry Points* 82d⟩≡ (79b)

```
.EXTERNAL connect_cb;
.EXTERNAL disconnect_cb;
.EXTERNAL cb_delay;
```

An *active low* signal is used—remember, `FO` was set to "LOW" at the start of the program—because the crossbar switch provides a "HIGH" for all ports that are not driven, i.e. in high impedance state.

82e ⟨*send test signal* 82e⟩≡ (82b) 83c▷
⟨*connect FO to specified port* 82f⟩

82f ⟨*connect FO to specified port* 82f⟩≡ (82e)

```
{-------------- Sending -------------}

                {Connect FO with specified port}
send_ts:        AX0 = DM(0x2000);
                AY0 = 0x6B00;
                AR = AX0 OR AY0;
                AX0 = AR;
                CALL connect_cb;
```

Afterwards, the processor waits for the *signal received* acknowledge by the PC. If a "1" is written to the variable `linkflag` at `0x2003`, it is set to zero again, the `FO` pin is disconnected and the loop continues, i.e. waits for the next *test*.

83a ⟨*TC: Variables* 80d⟩+≡ (79b) ◁81f

```
.VAR/DM/RAM/ABS=0x2003 linkflag;
```

83b ⟨*TC: Init Variables* 80e⟩+≡ (79b) ◁82a

```
.INIT linkflag: 0;
```

83c ⟨*send test signal* 82e⟩+≡ (82b) ◁82e

⟨*wait for "signal received" acknowledge* 83d⟩
⟨*reset "linkflag" to zero* 83e⟩
⟨*disconnect FO pin* 83f⟩

```
                JUMP test_loop;
```

83d ⟨*wait for "signal received" acknowledge* 83d⟩≡ (83c)

```
                {Wait for acknowledge from PC}
send_ack_pc:    AX0 = DM(0x2003);
                AY0 = 0x0001;
                AR = AX0 AND AY0;
                IF EQ JUMP send_ack_pc;
```

83e ⟨*reset "linkflag" to zero* 83e⟩≡ (83c)

```
send_lf_reset:  {Reset linkflag to zero}
                AX0 = 0x0000;
                DM(0x2003) = AX0;
```

83f ⟨*disconnect FO pin* 83f⟩≡ (83c)

```
                {Disconnect FO}
send_disconn:   AX0 = DM(0x2000);
                AY0 = 0x6B00;
                AR = AX0 OR AY0;
                AX0 = AR;
                CALL disconnect_cb;

                set FL0;
```

The *receiver* connects its FI pin to the given port.

84a ⟨*receive test signal* 84a⟩≡ (82b) 84c▷
⟨*connect FI to specified port* 84b⟩

84b ⟨*connect FI to specified port* 84b⟩≡ (84a)

```
{------------- Receiving ------------}


                 {Connect FI with specified port}
receive_ts:      AX0 = DM(0x2000);
                 AY0 = 0x6A00;
                 AR = AX0 OR AY0;
                 AX0 = AR;
                 CALL connect_cb;
```

Now, it waits for the FI pin to get "LOW", but not forever. The PC can signal a *timeout* by setting the dataflag at 0x2002 to "1". If this happens, the dataflag is reset to "0" and the FI pin is disconnected immediately.

84c ⟨*receive test signal* 84a⟩+≡ (82b) ◁84a 84f▷
⟨*wait for FI to get "LOW"* 84d⟩
⟨*timeout occured, so reset dataflag* 84e⟩

84d ⟨*wait for FI to get "LOW"* 84d⟩≡ (84c)

```
                 {Wait for FI pin to get LOW}
receive_fi:      AX0 = DM(0x2002);
                 AY0 = 0x0001;
                 AR = AX0 AND AY0;
                 IF NE JUMP rec_time_out;
                 IF FLAG_IN JUMP receive_fi;

                 JUMP receive_linkfl;
```

84e ⟨*timeout occured, so reset dataflag* 84e⟩≡ (84c)

```
                 {Time out occured}
rec_time_out:    {reset dataflag}
                 AX0 = 0x0000;
                 DM(0x2002) = AX0;
                 JUMP receive_disc;
```

If the test signal was received, the linkflag at 0x2003 is set. Then, the program waits, until the PC acknowledges by resetting the linkflag to "0".

84f ⟨*receive test signal* 84a⟩+≡ (82b) ◁84c 85b▷
⟨*set linkflag and wait for acknowledge* 85a⟩

85a ⟨*set linkflag and wait for acknowledge* 85a⟩≡ (84f)

```
                  {Set linkflag as acknowledge for PC}
receive_linkfl:   AX0 = 0x0001;
                  DM(0x2003) = AX0;

                  {Wait for PC to reset linkflag to zero}
receive_ack_pc:   AX0 = DM(0x2003);
                  AY0 = 0x0001;
                  AR = AX0 AND AY0;
                  IF NE JUMP receive_ack_pc;
```

Afterwards, the `FI` pin is disconnected and the loop starts with a new *test*.

85b ⟨*receive test signal* 84a⟩+≡ (82b) ◁84f
⟨*disconnect FI pin* 85c⟩

85c ⟨*disconnect FI pin* 85c⟩≡ (85b)

```
                  {Disconnect FI}
receive_disc:     AX0 = DM(0x2000);
                  AY0 = 0x6A00;
                  AR = AX0 OR AY0;
                  AX0 = AR;
                  CALL disconnect_cb;

                  set FL0;

                  JUMP test_loop;
```

Again, the resulting executable is added to the library as an array of integers:

85d ⟨*Defines* 8c⟩+≡ (1a) ◁70b

```
/** Start address of the program ``test.exe'' */
#define TEST_START             0x1000
/** Length of the program ``test.exe'' */
#define TEST_LENGTH            78
```

Defines:
`TEST_LENGTH`, used in chunks 85e and 86b.
`TEST_START`, used in chunks 86b and 87b.

85e ⟨*Global variables* 3b⟩+≡ (1a) ◁69c

```
/** The program ``test.exe'' for testing crossbar links/edges */
static const int test_exe[TEST_LENGTH] = {
0x83FFFA, 0x408004, 0x23A20F, 0x4FBFF4, 0x23820F,
0x93FFFA, 0x02002F, 0x820020, 0x400014, 0x23800F,
0x190070, 0x400000, 0x920020, 0x820000, 0x401004,
```

```
0x23800F, 0x190431, 0x02008F, 0x820010, 0x400014,
0x23800F, 0x190280, 0x820000, 0x46B004, 0x23A00F,
0x0D000A, 0x1D044F, 0x820030, 0x400014, 0x23800F,
0x1901B0, 0x400000, 0x920030, 0x820000, 0x46B004,
0x23A00F, 0x0D000A, 0x1D047F, 0x0200CF, 0x19007F,
0x820000, 0x46A004, 0x23A00F, 0x0D000A, 0x1D044F,
0x820020, 0x400014, 0x23800F, 0x190331, 0x0302D6,
0x19036F, 0x400000, 0x920020, 0x1903CF, 0x400010,
0x920030, 0x820030, 0x400014, 0x23800F, 0x190381,
0x820000, 0x46A004, 0x23A00F, 0x0D000A, 0x1D047F,
0x0200CF, 0x19007F, 0x0A000F, 0x0180F0, 0x1D04AF,
0x0A000F, 0x018070, 0x1D04AF, 0x0A000F, 0x3C0205,
0x1504CE, 0x000000, 0x0A000F
};
```

Defines:
test_exe, used in chunk 86b.
Uses **TEST_LENGTH** 85d.

In the next step, a function is defined for loading the program `test.exe` to a single processor.

86a ⟨*Testing Connections* 86a⟩≡ (3a) 87a▷
⟨*Load Test Program* 86b⟩

86b ⟨*Load Test Program* 86b⟩≡ (86a)

```
/** Loads the program TEST.EXE to the specified processor.
* @param processor Physical address of the processor
* @return ERROR if an error occurs, OK else
*/
int load_test_program(int processor)
{
  int data[5] = {0, 0, 0, 0, 0};

  /* Load the program data to the processor */
  if (broadcast_memory(SINGLE, processor,
      TEST_START, prog, test_exe, TEST_LENGTH) == ERROR)
  {
    return(ERROR);
  }

  /* Initialize variables */
  if (broadcast_memory(SINGLE, processor,
      0x2000, dat, data, 5) == ERROR)
  {
    return(ERROR);
  }

  return(OK);
}
```

Defines:
load_test_program, used in chunks 92c and 94b.
Uses broadcast_memory 25d 71a, error 5a, SINGLE 24a, test_exe 85e, TEST_LENGTH 85d, and TEST_START 85d.

Starting the test program on an involved ER2 module is also required.

87a ⟨*Testing Connections* 86a⟩+≡ (3a) ◁86a 87c▷
⟨*Start Test Program* 87b⟩

87b ⟨*Start Test Program* 87b⟩≡ (87a)

```
/** Starts the program TEST.EXE on a processor that is to be tested.
* @param processor Physical address of the processor
* @return ERROR if an error occurs, OK else
*/
int start_test_program(int processor)
{

  return(broadcast_start_program(SINGLE, processor,
                                 TEST_START, 0xFF));
}
```

Defines:
start_test_program, used in chunks 92c and 94b.
Uses broadcast_start_program 53c, error 5a, SINGLE 24a, and TEST_START 85d.

Now, a single *edge* can be tested.

87c ⟨*Testing Connections* 86a⟩+≡ (3a) ◁87a 89c▷
⟨*Test Edge* 87d⟩

87d ⟨*Test Edge* 87d⟩≡ (87c)

```
/** Tests a single configured crossbar path (=edge) between two
* different processors or two different ports.
* The program TEST.EXE has to be running on both processors.
* @param send_proc Physical address of the sender processor
* @param send_port The crossbar port of the path at the sender
* @param rec_proc Physical address of the receiver processor
* @param rec_port The crossbar port of the path at the receiver
* @return ERROR if an error occurs, OK else
*/
int test_edge(int send_proc, int send_port, int rec_proc, int rec_port)
{
  int adsp_data[2];

  if ((send_proc == rec_proc) && (send_port == rec_port))
    return(OK);

  ⟨te: initialize data for sender and receiver 88a⟩
  ⟨te: start "test" loop on ADSPs 88b⟩
```

⟨*te: check if "test" signal was received* 88c⟩

```
}
```

Defines:
test_edge, used in chunks 88c, 93a, and 94b.
Uses **error** 5a.

88a ⟨*te: initialize data for sender and receiver* 88a⟩≡ (87d)

```
/* Write port and ''send'' flag to the sender */
adsp_data[0] = send_port;
adsp_data[1] = 1;
broadcast_memory(SINGLE, send_proc, 0x2000, dat, adsp_data, 2);

/* Write port and ''receive'' flag to the receiver */
adsp_data[0] = rec_port;
adsp_data[1] = 0;
broadcast_memory(SINGLE, rec_proc, 0x2000, dat, adsp_data, 2);
```

Uses **broadcast_memory** 25d 71a and **SINGLE** 24a.

88b ⟨*te: start "test" loop on ADSPs* 88b⟩≡ (87d)

```
/* Initialize data */
adsp_data[0] = 1;

/* Start the test loop on the sender processor */
broadcast_memory(SINGLE, send_proc, 0x2002, dat, adsp_data, 1);
/* Start the test loop on the receiver processor */
broadcast_memory(SINGLE, rec_proc, 0x2002, dat, adsp_data, 1);
```

Uses **broadcast_memory** 25d 71a and **SINGLE** 24a.

88c ⟨*te: check if "test" signal was received* 88c⟩≡ (87d)

```
/* Check the result */
request_memory(rec_proc, 0x2003, dat, adsp_data, 1);

/* Test successful? */
if (adsp_data[0] == 1)
{
  sprintf(msg, "<test_edge> : Connection (%d/0x%X)<->(%d/0x%X) is available.",
          send_proc, send_port, rec_proc, rec_port);
  info(msg);
```

⟨*te: send acknowledge to sender and receiver* 89a⟩

```
  return(OK);
}
```

```
else
{

  sprintf(msg, "<test_edge> : No connection for (%d/0x%X)<->(%d/0x%X)!",
          send_proc, send_port, rec_proc, rec_port);
  error(msg);

  ⟨te: send timeout to sender and receiver 89b⟩

  return(ERROR);
}
```

Uses **error** 5a, **info** 5a, **msg** 42d, **request_memory** 21b, and **test_edge** 87d.

89a ⟨*te: send acknowledge to sender and receiver* 89a⟩≡ (88c)

```
/* Set ``linkflag'' to ``1'' for sender */
adsp_data[0] = 1;
broadcast_memory(SINGLE, send_proc, 0x2003, dat, adsp_data, 1);

/* Set ``linkflag'' to ``0'' for receiver */
adsp_data[0] = 0;
broadcast_memory(SINGLE, rec_proc, 0x2003, dat, adsp_data, 1);
```

Uses **broadcast_memory** 25d 71a and **SINGLE** 24a.

89b ⟨*te: send timeout to sender and receiver* 89b⟩≡ (88c)

```
/* Set ``linkflag'' to ``1'' for sender */
adsp_data[0] = 1;
broadcast_memory(SINGLE, send_proc, 0x2003, dat, adsp_data, 1);

/* Set ``dataflag'' to ``1'' for receiver */
broadcast_memory(SINGLE, rec_proc, 0x2002, dat, adsp_data, 1);
```

Uses **broadcast_memory** 25d 71a and **SINGLE** 24a.

After all *testing* is done, the program on an ADSP-2181 can be stopped by specifiying a port number of `0x100`.

89c ⟨*Testing Connections* 86a⟩+≡ (3a) ◁87c 90a▷

```
⟨Stop Test Program 89d⟩
```

89d ⟨*Stop Test Program* 89d⟩≡ (89c)

```
/** Stops the program TEST.EXE on the given processor.
* @param processor Physical address of the processor
* @return ERROR if an error occurs, OK else
*/
```

```
int stop_test_program(int processor)
{
  int adsp_data;

  /* Write port ''0x100'' to processor */
  adsp_data = 0x100;
  broadcast_memory(SINGLE, processor, 0x2000, dat, &adsp_data, 1);

  /* Start the test loop on the processor */
  /* -> stops the program */
  adsp_data = 1;
  broadcast_memory(SINGLE, processor, 0x2002, dat, &adsp_data, 1);

}
```

Defines:
  stop_test_program, used in chunks 93b and 94b.
Uses **broadcast_memory** 25d 71a, **error** 5a, and **SINGLE** 24a.

Now, most of the preceding *testing* functions are combined for verifying all defined *edges* at once. Since hardware connections do not need to be tested twice, `mark_edge` is used to set the *was already tested* flag for both ends of the communication path.

90a ⟨*Testing Connections* 86a⟩+≡ (3a) ◁89c 92a▷
  ⟨*Mark Edge* 90b⟩
  ⟨*Unmark All Edges* 91⟩

90b ⟨*Mark Edge* 90b⟩≡ (90a)

```
/** Marks an edge as ''already tested''.
* @param l_proc Physical address of the local processor
* @param l_port Port of the connection at the local processor
* @param c_proc Physical address of the remote processor
* @param c_port Port of the connection at the remote processor
*/
static void mark_edge(int l_proc, int l_port, int c_proc, int c_port)
{
  struct conf_data *conf_help=list_root;
  struct edge *help_edge;

  /* Are there any entries? */
  if (conf_help != NULL)
  {
    /* Search the processor */
    while ((conf_help->next_data != NULL) &&
           (conf_help->proc_number != l_proc))
    {
      conf_help = conf_help->next_data;
    }
```

```
      if (conf_help->proc_number == l_proc)
      {
        /* Search edge */
        help_edge = conf_help->edges;
        while (help_edge != NULL)
        {
          if ((help_edge->port == l_port) &&
              (help_edge->conn_proc == c_proc) &&
              (help_edge->conn_port == c_port))
          {
            /* Mark it */
            help_edge->tested = 1;
            return;
          }

          help_edge = help_edge->next_edge;
        }
      }
    }

  }
```

Defines:
  `mark_edge`, used in chunk 93a.
Uses `list_root` 57a.

`unmark_all_edges` resets the *tested* flag for all currently specified *edges*.

91 ⟨*Unmark All Edges* 91⟩≡ (90a)

```
  /** Sets back the ''tested'' flag for all edges in the linear
  * list of configuration data.
  */
  static void unmark_all_edges(void)
  {
    struct conf_data *conf_help=list_root;
    struct edge *help_edge;

    /* All processors */
    while (conf_help != NULL)
    {
      /* All edges */
      help_edge = conf_help->edges;
      while (help_edge != NULL)
      {
        /* Set back ''tested'' flag */
        help_edge->tested = 0;

        help_edge = help_edge->next_edge;
      }

      conf_help = conf_help->next_data;
```

```
  }

}
```

Defines:
  unmark_all_edges, used in chunk 92b.
Uses list_root 57a.

test_all_edges loads the program **test.exe** to all processors that are contained in the list of configuration data. It starts them and checks all defined edges, keeping track of the number of errors. Afterwards, all *tested* flags are reset by **unmark_all_edges** and the *test* programs are stopped.

92a ⟨*Testing Connections* 86a⟩+≡ (3a) ◁90a
  ⟨*Test All Edges* 92b⟩

92b ⟨*Test All Edges* 92b⟩≡ (92a)

```
/** Tests all edges in the linear list of configuration data for
* the ER2 network. If \a test_direction is set to ONE_DIRECTION,
* all paths are only tested once in a single direction.
* @param test_direction If ''ONE_DIRECTION'' paths are tested only
* in one direction, for ''BOTH_DIRECTIONS'' each direction is checked.
* @return The number of errors
*/
int test_all_edges(int test_direction)
{
  struct conf_data *conf_help;
  struct edge *help_edge;
  int test_errors = 0;

  ⟨tae: load and start "test.exe" 92c⟩
  ⟨tae: test edges 93a⟩
  ⟨tae: stop "test.exe" 93b⟩

  /* Unmark edges */
  unmark_all_edges();

  return(test_errors);

}
```

Defines:
  test_all_edges, used in chunk 94b.
Uses direction 29b, ER2 1b, errors 3b, ONE_DIRECTION 94a, and unmark_all_edges 91.

92c ⟨*tae: load and start "test.exe"* 92c⟩≡ (92b)

```
info("Loading ''test.exe''...");
/* All processors */
conf_help = list_root;
```

```
while (conf_help != NULL)
{
  load_test_program(conf_help->proc_number);
  start_test_program(conf_help->proc_number);

  conf_help = conf_help->next_data;
}
```

Uses info 5a, list_root 57a, load_test_program 86b, and start_test_program 87b.

93a ⟨*tae: test edges* 93a⟩≡ (92b)

```
info("Testing all configured edges...");
/* All processors */
conf_help = list_root;
while (conf_help != NULL)
{
  /* All edges */
  help_edge = conf_help->edges;
  while (help_edge != NULL)
  {
    if (help_edge->tested == 0)
    {
      /* Mark edges */
      mark_edge(conf_help->proc_number, help_edge->port,
                help_edge->conn_proc, help_edge->conn_port);
      if (test_direction == ONE_DIRECTION)
      {
        mark_edge(help_edge->conn_proc, help_edge->conn_port,
                  conf_help->proc_number, help_edge->port);
      }
      /* Test connection */
      if (test_edge(conf_help->proc_number, help_edge->port,
          help_edge->conn_proc, help_edge->conn_port) == ERROR)
        test_errors++;
    }

    help_edge = help_edge->next_edge;
  }

  conf_help = conf_help->next_data;
}
```

Uses info 5a, list_root 57a, mark_edge 90b, ONE_DIRECTION 94a, and test_edge 87d.

93b ⟨*tae: stop "test.exe"* 93b⟩≡ (92b)

```
info("Stopping ``test.exe''...");
/* All processors */
conf_help = list_root;
```

```
while (conf_help != NULL)
{
  stop_test_program(conf_help->proc_number);

  conf_help = conf_help->next_data;
}
```

Uses info 5a, list_root 57a, and stop_test_program 89d.

94a ⟨*HF: Defines* 24a⟩+≡ (2b) ◁30d

```
/** Value that specifies a test of all switched crossbar links
in only one direction */
#define ONE_DIRECTION          0
/** Value that specifies a test of all switched crossbar links
in both directions */
#define BOTH_DIRECTIONS        1
```

Defines:
BOTH_DIRECTIONS, never used.
ONE_DIRECTION, used in chunks 92b and 93a.
Uses direction 29b.

94b ⟨*HF: Function prototypes* 4b⟩+≡ (2b) ◁79a 96a▷

```
extern int load_test_program(int);
extern int start_test_program(int);
extern int test_edge(int, int, int, int);
extern int stop_test_program(int);
extern int test_all_edges(int);
```

Uses load_test_program 86b, start_test_program 87b, stop_test_program 89d, test_all_edges 92b, and test_edge 87d.

3.9 Startup and shutdown

sleep needs the header unistd.

94c ⟨*Include files* 2a⟩+≡ (1a) ◁2a

```
#include <unistd.h>
```

94d ⟨*Startup* 94d⟩≡ (3a)

```
/** Initializes the ER2. Opens the device file, selects the interface type,
* resets all configuration data, marks the root processor,
* detects the complete network and sets up the internal routing table.
* @param ifc_type Type of interface that is to be used
```

```
* (PARALLEL_PORT/ISA_CARD)
* @return ERROR if an error occurs, OK else
*/
int startup_er2(int ifc_type)
{
  int data = ifc_type;

  /* try to open device file */
  device_file = open("/dev/er2p", 0);
  if (device_file < 0)
  {
    error("<startup_er2>: Can not open device file ‘‘/dev/er2p’’!");
    return(ERROR);
  }

  /* Set interface type */
  ioctl(device_file, IOCTL_ER2_SET_INTERFACE, &data);

  /* Reset ER2 */
  info("Resetting ER2...");
  ioctl(device_file, IOCTL_ER2_RESET, 0);

  /* Rest a while... */
  sleep(2);

  /* Mark root node */
  info("Marking root node...");
  mark_root_node();

  /* Initialize routing table */
  info("Initializing routing table...");
  init_routing_table();

  return(OK);
}
```

Defines:
`startup_er2`, used in chunk 96.
Uses `device_file` 5b, `ER2` 1b, `error` 5a, `file` 2c, `info` 5a, `init_routing_table` 33c, `mark_root_node` 28d, and `table` 30b.

95 ⟨*Shutdown* 95⟩≡ (3a)

```
/** Shuts the ER2 down. Closes the device file and frees the list of
* configuration data.
*/
void shutdown_er2(void)
{
  info("Shutting down ER2...");

  /* Close device file */
```

```
    close(device_file);

    /* Free list of configuration data */
    free_configuration_list();
  }
```

Defines:
  `shutdown_er2`, used in chunk 96.
Uses `device_file` 5b, `ER2` 1b, `file` 2c, `free_configuration_list` 66a, and `info` 5a.

96a ⟨*HF: Function prototypes* 4b⟩+≡ (2b) ◁94b

```
  extern int startup_er2(int);
  extern void shutdown_er2(void);
```

Uses `shutdown_er2` 95 and `startup_er2` 94d.

4 Building and installing the library

For compiling the library please use the prepared `Makefile` by issueing the command

```
make
```

After successful compilation, the library `liber2.a` and the header `er2.h` should be installed by

```
make install
```

assuming that you changed to *root* mode.

5 Test program

96b ⟨*er2test.c* 96b⟩≡

```
  #include <stdio.h>
  #include "../device_driver/er2gdef.h"
  #include "er2.h"

  int main(void)
  {
    verbose_on();

    startup_er2(PARALLEL_PORT);

    display_routing_table();

    shutdown_er2();

    return(0);
```

```
}
```

Defines:
  `main`, never used.
Uses `display_routing_table` 35a, `shutdown_er2` 95, `startup_er2` 94d, and `verbose_on` 4a.

List of code chunks

This list was generated automatically by NOWEB. The numeral is that of the first definition of the chunk.

Index

This is a list of identifiers used, and where they appear. Underlined entries indicate the place of definition.

References

[1] Analog Devices. *ADSP-2100 Family User's Manual*, third edition, 1995.

[2] Dirk Bächle. *A Linux Device Driver for the Parallel Port and the ISA Card Host Interface of the ER2*, 2003. Internal report.

[3] Georg-Friedrich Mayer-Lindenberg. *Message Passing im ER2 und Funktionen der Laufzeitkerne*, 1997. Internal report.

[4] Arnd Seeger. *Schematic of the ER2 Backplane*, 1996. Internal document "er2_mo7".

[5] Arnd Seeger. *Schematic of the ER2 Module*, 1996. Internal document "er2_lu1".