A SHARC Library in C

Functions for Accessing the SHARC Modules on the ER2

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

December 2, 2003

Contents

1 Introduction

This collection of functions relies on the Linux device driver `er2p` and the ER2 library `er2` (see [3, 2]). It supports the so called SHARC modules, by providing automatic bootstrapping. Additionally, memory access is handled by appropriate subroutines. Most of the following descriptions refer to [5]. It explains the communication between the ADSP-2181 and the ADSP-2106x.

2 Headers

2.1 The C library `er2sh.c`

The basic structure of the library looks like this:

1a ⟨*er2sh.c* 1a⟩≡
  ⟨*Header* 1b⟩
  ⟨*Include files* 1d⟩
  ⟨*Defines* 6a⟩
  ⟨*Global variables* 6b⟩
  ⟨*Functions* 3a⟩

The library starts with a disclaimer and some general information.

1b ⟨*Header* 1b⟩≡ (1a)

  ⟨*Disclaimer* 1c⟩

```
/** \file er2sh.c
A C library, providing several useful functions for accessing the
''SHARC modules'' on the ER2.
\author Dirk Baechle
\version 1.0
\date 26.11.2003
*/
```

Defines:
  `C`, used in chunks 2, 10b, 37c, 39b, and 42.
Uses `file` 2b.

1c ⟨*Disclaimer* 1c⟩≡ (1b 2b)

```
/* This file was created automatically from the file er2sh.nw by NOWEB. */
/* If you want to make changes, please edit the source file er2sh.nw. */
/* A complete documentation is in er2sh.tex, i.e. er2sh.dvi and er2sh.ps. */
/* Read it to understand why things are as they are. Thank you! */
```

Uses `file` 2b.

Next come the include files.

1d ⟨*Include files* 1d⟩≡ (1a) 72a▷

```
#include <stdio.h>
#include "../device_driver/er2gdef.h"
#include "../er2lib/er2.h"
#include "er2sh.h"
```

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

2.2 The header file `er2sh.h`

The appropriate header file **er2sh.h** has the following structure:

2a ⟨*er2sh.h* 2a⟩≡

```
⟨HF: Header 2b⟩

#ifndef _ER2SH_H
#define _ER2SH_H

⟨HF: Defines 2c⟩
#ifdef __cplusplus
extern "C" {
#endif
⟨HF: Function prototypes 11b⟩
#ifdef __cplusplus
}
#endif

#endif
```

Defines:
  _ER2SH_H, never used.
Uses **C** 1b.

2b ⟨*HF: Header* 2b⟩≡ (2a)

```
⟨Disclaimer 1c⟩

/** \file er2sh.h
Header file for the SHARC C library ``er2sh.c'' and all programs using it.
\author Dirk Baechle
\version 1.0
\date 26.11.2003
*/
```

Defines:
  `file`, used in chunks 1, 16b, 26d, 50b, 70d, and 73b.
Uses **C** 1b.

2c ⟨*HF: Defines* 2c⟩≡ (2a) 3c▷

3 Functions using the Fifth PRS

All functions in this section rely on the Fifth *parallel runtime system* (PRS) and the *message* service it offers. In a later section an own bootloader is developed that supports the direct loading of executables, IDMA access to the 2181 from the SHARCs and more.

3a ⟨*Functions* 3a⟩≡ (1a)
  ⟨*Booting SHARCs* 3b⟩
  ⟨*Reading and Writing* 14b⟩
  ⟨*Network Informations* 11c⟩
  ⟨*Loading Programs* 16a⟩
  ⟨*Executing Programs* 20b⟩

3.1 Booting the SHARCs

As specified in [5] there already exist two binary files `21cde.dat` and `shcde.dat`, containing a small runtime system that helps in booting the SHARCs and accessing their memory.
These have to be loaded into the memory of the attached ADSP-2181 and then the boot code at address `0x975` has to be executed.

3b ⟨*Booting SHARCs* 3b⟩≡ (3a)

```
/** Loads ``21cde.dat'' and ``shcde.dat'' to all
* nodes and tries to boot the SHARCs.
* @return ERROR if an error occurs, OK else
*/
int boot_sharcs(void)
{
  int cnt;
  ⟨bs: Additional variables 10a⟩

  ⟨bs: build group without root 4a⟩
  ⟨bs: load 21cde to all nodes 7a⟩
  ⟨bs: load shcde to all nodes 10b⟩
  ⟨bs: start shcde on all nodes 11a⟩

  return(OK);
}
```

Defines:
  `boot_sharcs`, used in chunks 11b, 21b, 23, 46, 48c, 57b, and 65a.
Uses `cnt` 26a 47a and `shcde` 7c.

For the following subroutines the *root* processor has to be excluded. It definitely has no SHARC cluster attached to it, but the host interface. The same holds for the *bridges* that connect the upper and lower half of the ER2. Thus, a special group is defined.

3c ⟨*HF: Defines* 2c⟩+≡ (2a) ◁2c 65b▷

```
/** Group of all ADSP-2181 without the ``root''
and ``bridge'' processors */
#define WOROOT          63
```

Defines:
`WOROOT`, used in chunks 4a, 7a, 10b, and 11a.

4a ⟨*bs: build group without root* 4a⟩≡ (3b)

```
for (cnt = 1; cnt < get_number_of_nodes(); cnt++)
{
  if (get_neighbour(get_physical_address(cnt), Bus) == EMPTY)
    join_group(get_physical_address(cnt), WOROOT);
}
```

Uses `cnt` 26a 47a and `WOROOT` 3c.

Instead of loading the binary boot code from an external file, they are directly included to the SHARC library in an ER2-readable form.
For converting the files, the following program `convert.c` was used:

4b ⟨*convert.c* 4b⟩≡

```
#include <stdio.h>

extern int fileSize(FILE *);
extern int *fileConvertChar2Int(FILE *, int *, int, int);

int main(void)
{
  ⟨conv: Variables 4c⟩

  ⟨conv: convert 21cde 5a⟩
  ⟨conv: convert shcde 5b⟩

  return(0);
}
```

Defines:
`main`, never used.

The functions `fileSize` and `fileConvertChar2Int` are defined in the file `er2file.cpp`. It is part of the work by Derik Schröter who wrote some routines for booting and accessing the SHARCs under Windows.

4c ⟨*conv: Variables* 4c⟩≡ (4b)

```
FILE *pFile_prog2181, *pFile_progSharc;
int *prog2181=NULL, *progSharc=NULL;
int wordLength_prog2181  = 24;
int wordLength_progSharc = 16;
```

```
  int size_prog2181=0, size_progSharc=0;
  int littleEndian=0, bigEndian=1;
  int icnt = 0;
```

Defines:
- `icnt`, used in chunk 5.
- `littleEndian`, never used.
- `pFile_prog2181`, used in chunk 5a.
- `pFile_progSharc`, used in chunk 5b.
- `prog2181`, used in chunk 5a.
- `size_prog2181`, used in chunk 5a.
- `wordLength_prog2181`, used in chunk 5a.
- `wordLength_progSharc`, used in chunk 5b.

5a ⟨*conv: convert 21cde* 5a⟩≡ (4b)

```
  /* Convert 21CDE.DAT */
  if( (pFile_prog2181 = fopen("21cde.dat", "r")) == NULL )
  {
    fprintf(stderr, "Could not open 21CDE.DAT!\n");
    return(1);
  }

  prog2181 = fileConvertChar2Int( pFile_prog2181, &size_prog2181,
                                  wordLength_prog2181, bigEndian );

  fclose(pFile_prog2181);

  if (prog2181 == NULL)
  {
    fprintf(stderr, "No memory for converting 21CDE.DAT!\n");
    return(1);
  }

  /* Write converted data */
  if( (pFile_prog2181 = fopen("21cde.hex", "w")) == NULL )
  {
    fprintf(stderr, "Could not open 21cde.hex!\n");
    return(1);
  }

  for (icnt = 0; icnt < size_prog2181; icnt++)
  {
    fprintf(pFile_prog2181, "0x%6X,\n", prog2181[icnt]);
  }

  fclose(pFile_prog2181);
```

Uses `data` 26a 47d, `icnt` 4c, `pFile_prog2181` 4c, `prog2181` 4c, `size_prog2181` 4c, and `wordLength_prog2181` 4c.

5b ⟨*conv: convert shcde* 5b⟩≡ (4b)

```
/* Convert SHCDE.DAT */
if( (pFile_progSharc = fopen("shcde.dat", "r")) == NULL )
{
  fprintf(stderr, "Could not open SHCDE.DAT!\n");
  return(1);
}

progSharc = fileConvertChar2Int( pFile_progSharc, &size_progSharc,
                                 wordLength_progSharc, bigEndian );

fclose(pFile_progSharc);

/* Write converted data */
if( (pFile_progSharc = fopen("shcde.hex", "w")) == NULL )
{
  fprintf(stderr, "Could not open shcde.hex!\n");
  return(1);
}

for (icnt = 0; icnt < size_progSharc; icnt++)
{
  fprintf(pFile_progSharc, "0x%4X,\n", progSharc[icnt]);
}

fclose(pFile_progSharc);
```

Uses `data` 26a 47d, `icnt` 4c, `pFile_progSharc` 4c, `shcde` 7c, and `wordLength_progSharc` 4c.

Now, the contents of `21cde.dat` are added to the library.

6a ⟨*Defines* 6a⟩≡ (1a) 7b▷

```
/** Length of the ``21cde.dat'' data */
#define S21CDE_LENGTH          83
```

Defines:
`S21CDE_LENGTH`, used in chunks 6b and 7a.
Uses `data` 26a 47d.

6b ⟨*Global variables* 6b⟩≡ (1a) 7c▷

```
/** The data of ``21cde.dat'' in ER2-readable form. */
static int s21cde[S21CDE_LENGTH] = {
0x09001E, 0x83FE50, 0x78000D, 0x70000D, 0x4FFEF4,
0x23800F, 0x7800AE, 0xB0008D, 0x70000F, 0x70004D,
0x23A00F, 0x7800AD, 0x70000F, 0x93FE50, 0x83FE60,
0x400184, 0x23A00F, 0x93FE6A, 0x0A000F, 0x3C0045,
0x14972E, 0x09001E, 0xB0000D, 0x70000F, 0x70004D,
0x78000E, 0x78004D, 0x400049, 0x70008F, 0x13108D,
0x0E080F, 0x7800FD, 0x09001E, 0x7800ED, 0x70000F,
0x70004D, 0x78000E, 0x78004D, 0x70000F, 0x018100,
0x09001F, 0x0A000F, 0x400000, 0x93FE00, 0x400000,
0x902040, 0x1C94BF, 0x3C3005, 0x1498AE, 0x0D0345,
0x403000, 0x22E00F, 0x09001E, 0x7800AD, 0x70000D,
0x408004, 0x22600F, 0x7800AD, 0x70000D, 0x0D0800,
0x700001, 0x78000D, 0x1C95EF, 0x000000, 0x83FE50,
0x400104, 0x23A00F, 0x93FE5A, 0x09001E, 0xB0064D,
0x1C108F, 0x09001E, 0x83FE00, 0x78000E, 0xB4300D,
0x70004F, 0x70000F, 0x2AE0AA, 0x1899D1, 0x401000,
0x902220, 0x0A000F, 0x0A000F
};
```

Defines:
  s21cde, used in chunk 7a.
Uses data 26a 47d and S21CDE_LENGTH 6a.

7a ⟨*bs: load 21cde to all nodes* 7a⟩≡ (3b)

```
    broadcast_memory(WOROOT, 0x1, 0x94B, prog, s21cde, S21CDE_LENGTH);
```

Uses s21cde 6b, S21CDE_LENGTH 6a, and WOROOT 3c.

The file shcde.dat contains a lot of adjacent memory locations that are set to 0xBBBB. In order to save some space the array shcde stores only the data that is different from this value, using the following mapping:

Part	Array index	Load to address 0x800 plus	Length
A	0-8	12	9
B	9-23	48	15
C	24-38	84	15
D	39-209	192	171
E	210-593	384	384

7b ⟨*Defines* 6a⟩+≡ (1a) ◁6a 11d▷

```
/** Length of the ''shcde.dat'' data */
#define SHCDE_LENGTH            594
```

Defines:
  SHCDE_LENGTH, used in chunk 7c.
Uses data 26a 47d and shcde 7c.

Please, note that the following hex data is no direct representation of the binary file shcde.dat but of its patched version, developed in section 6.7 (pp. 51)!

7c ⟨*Global variables* 6b⟩+≡ (1a) ◁6b 12a▷

```
/** The data of ''shcde.dat'' in ER2-readable form. */
static int shcde[SHCDE_LENGTH] = {
0x0000, 0x0000, 0x0000, 0x06BE, 0x0002,
0x00C2, 0x063E, 0x0002, 0x00E3, 0x1208,
0x0002, 0x0002, 0x013E, 0x0002, 0x9880,
0x1108, 0x0002, 0x0002, 0x0B3E, 0x0000,
0x0000, 0x0A3E, 0x0000, 0x0000, 0x1208,
0x0002, 0x0002, 0x013E, 0x0002, 0xA880,
0x1108, 0x0002, 0x0002, 0x0B3E, 0x0000,
0x0000, 0x0A3E, 0x0000, 0x0000, 0x0B3E,
0x0000, 0x0000, 0x0A3E, 0x0000, 0x0000,
0xB0DB, 0x0000, 0x0000, 0x047E, 0x0000,
0x0000, 0x13DB, 0x0010, 0x0004, 0x13DB,
0x0018, 0x0004, 0x13DB, 0x0020, 0x0004,
0x013E, 0x0000, 0x2001, 0x0620, 0x0002,
0x004A, 0x0A3E, 0x0000, 0x0000, 0x063E,
0x0002, 0x0042, 0x0F00, 0x0000, 0xA308,
0x1100, 0x0038, 0x0000, 0x0F00, 0x0000,
0x0021, 0x1100, 0x0010, 0x001C, 0x1100,
0x0018, 0x001C, 0x1100, 0x0020, 0x001C,
0x0F18, 0x0002, 0x0000, 0x0F00, 0x0000,
0x0100, 0x0F28, 0x0000, 0x0001, 0x0F38,
0x0000, 0x0000, 0x0F01, 0x0000, 0x0001,
0x063E, 0x0002, 0x0042, 0x0F0B, 0x0000,
0xFFFF, 0x013E, 0x0004, 0x0AAB, 0x0F0B,
0x0000, 0x0001, 0x013E, 0x0000, 0x299B,
0x023E, 0x0033, 0x0209, 0x0610, 0x0002,
0x0064, 0x013E, 0x0000, 0x299B, 0x013E,
0x0004, 0x099B, 0x023E, 0x0000, 0x1099,
0x023E, 0x0030, 0x1699, 0x013E, 0x0004,
0x199A, 0x709F, 0x8C00, 0x0000, 0x0A3E,
0x0000, 0x0000, 0x013E, 0x0000, 0x199B,
0x023E, 0x0000, 0x1399, 0x0F0B, 0x0002,
0x0000, 0x013E, 0x0004, 0x199B, 0x013E,
0x0004, 0x199A, 0x709F, 0x8C00, 0x0000,
0x0A3E, 0x0000, 0x0000, 0x0F08, 0x0000,
0x4209, 0x1108, 0x00CE, 0x0000, 0x1208,
0x00C8, 0x0000, 0x023E, 0x0033, 0x0008,
0x0610, 0x0002, 0x0071, 0x063E, 0x0002,
0x006B, 0x0F08, 0x0000, 0x4208, 0x1108,
0x00CE, 0x0000, 0x1109, 0x00C4, 0x0000,
0x0F09, 0x0000, 0x0001, 0x1109, 0x00C4,
0x0000, 0x1109, 0x0140, 0x0000, 0x0A3E,
0x0000, 0x0000, 0x0A3E, 0x0000, 0x0000,
0x013E, 0x0002, 0x9CC0, 0x0F0B, 0x0000,
0x00FF, 0x013E, 0x0004, 0x0CCB, 0x0F0B,
0x0000, 0x4300, 0x013E, 0x0000, 0x1CCB,
0x110C, 0x00CE, 0x0000, 0x0A3E, 0x0000,
0x0000, 0x0F0C, 0x0000, 0x4215, 0x110C,
0x00CE, 0x0000, 0x120C, 0x00C8, 0x0000,
0x0F09, 0x0000, 0x4000, 0x013E, 0x0000,
0x1CC9, 0x110C, 0x00CE, 0x0000, 0x1208,
0x00C8, 0x0000, 0x06BE, 0x0002, 0x0080,
0x120A, 0x00C8, 0x0000, 0x0F09, 0x0000,
0x7FFF, 0x013E, 0x0004, 0x0989, 0x023E,
0x0033, 0x0F08, 0x0630, 0x0002, 0x00AC,
0x0F08, 0x0000, 0x00FF, 0x013E, 0x0004,
0x0898, 0x013E, 0x0002, 0xA880, 0x023E,
0x0000, 0xF899, 0x06BE, 0x0002, 0x0057,
0x06BE, 0x0002, 0x009B, 0x0B3E, 0x0000,
0x0000, 0x06BE, 0x0002, 0x0080, 0x1209,
0x00C8, 0x0000, 0x023E, 0x0000, 0x1099,
0x06BE, 0x0002, 0x0080, 0x120A, 0x00C8,
0x0000, 0x0F0B, 0x0000, 0xFFFF, 0x013E,
0x0004, 0x0AAB, 0x013E, 0x0004, 0x199A,
0x709F, 0xEE80, 0x0000, 0x06BE, 0x0002,
0x0080, 0x12DC, 0x00C8, 0x0000, 0xB1DB,
0x0000, 0x0000, 0x047E, 0x0000, 0x0000,
0x0F09, 0x0000, 0x0003, 0x013E, 0x0000,
0x2889, 0x0620, 0x0002, 0x009B, 0x0A3E,
0x0000, 0x0000, 0x06BE, 0x0002, 0x0057,
0xB0DB, 0x0000, 0x0000, 0x7DDF, 0x8480,
0x0000, 0x023E, 0x0000, 0xE899, 0x0F08,
0x0000, 0x8000, 0x023E, 0x0030, 0x0F99,
0x06BE, 0x0002, 0x006B, 0x7DDF, 0x8480,
0x0000, 0x023E, 0x0000, 0xF899, 0x06BE,
0x0002, 0x006B, 0x7DDF, 0x8480, 0x0000,
0x0F08, 0x0000, 0x00FF, 0x013E, 0x0004,
0x0989, 0x0F08, 0x0000, 0x8100, 0x013E,
0x0004, 0x1989, 0x06BE, 0x0002, 0x006B,
0x7DCF, 0x8480, 0x0000, 0x06BE, 0x0002,
0x006B, 0x0B3E, 0x0000, 0x0000, 0x0000,
0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
0x0000, 0x0000, 0x0000, 0x0F27, 0x0000,
0x0001, 0x0F26, 0xFFFF, 0xFFFF, 0x0F37,
0x0000, 0x0000, 0x0F2F, 0x0000, 0x0000,
0x0F00, 0x01AD, 0x6B41, 0x1100, 0x0000,
0x0002, 0x0F00, 0x0000, 0x000E, 0x1100,
0x0000, 0x0018, 0x0F00, 0x0000, 0xA308,
0x1100, 0x0000, 0x0000, 0x0F00, 0x0000,
0x0000, 0x1100, 0x0000, 0x00C6, 0x0F00,
0x0000, 0x0000, 0x1100, 0x0000, 0x001C,
0x1100, 0x0002, 0x0000, 0x1200, 0x0000,
0x0003, 0x0F01, 0x0000, 0x0600, 0x013E,
0x0004, 0x0101, 0x0620, 0x0002, 0x00DF,
0x0F00, 0x0000, 0x4300, 0x1100, 0x00CE,
0x0000, 0x06BE, 0x0002, 0x004B, 0x06BE,
0x0002, 0x00FC, 0x0FDE, 0x0001, 0x0000,
0x0F7A, 0x0000, 0x8024, 0x0F79, 0x0000,
0x0000, 0x0F7D, 0x0000, 0x0050, 0x0F7C,
0x0000, 0x0000, 0x0A3E, 0x0000, 0x0000,
0x0F7A, 0x0000, 0x8002, 0x0F7D, 0x0000,
0x0080, 0x0F7C, 0x0000, 0x0000, 0x0A3E,
0x0000, 0x0000, 0x0F7B, 0x0000, 0x0000,
0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
0x0000, 0x12DB, 0x0002, 0x0000, 0x7DDF,
0x8000, 0x0000, 0x7DCF, 0x8080, 0x0000,
0x0F7B, 0x0000, 0x1000, 0x013E, 0x0004,
0x1000, 0x0600, 0x0002, 0x00FA, 0x023E,
0x0030, 0x1111, 0x701F, 0x8B80, 0x0000,
0x700F, 0x8F80, 0x0000, 0xAE0F, 0x0000,
0x0000, 0x08BF, 0xF800, 0x0000, 0xAF0F,
0x0000, 0x0000, 0x0F7B, 0x0000, 0x0000,
0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
0x0000, 0x717F, 0xEE00, 0x0000, 0x0FDD,
0x0000, 0x0000, 0x13DB, 0x0002, 0x0000,
0x0F7B, 0x0000, 0x1000, 0x063E, 0x0002,
0x00E3, 0x063E, 0x0002, 0x00E3, 0x0A3E,
0x0000, 0x0000, 0x0FDD, 0x063E, 0x0002,
0x0FDC, 0x0002, 0x0087, 0x13DB, 0x0002,
0x0018, 0x0A3E, 0x0000, 0x0000
};
```

Defines:
  `shcde`, used in chunks 3b, 5b, 7b, and 10b.
Uses `data` 26a 47d and `SHCDE_LENGTH` 7b.

The array `init` is used to initialize the whole memory from `0x800` to `0xDFF` to the value `0xBBBB`.

10a ⟨*bs: Additional variables* 10a⟩≡ (3b)

```
int init[16] = {0xBBBB, 0xBBBB, 0xBBBB, 0xBBBB,
                0xBBBB, 0xBBBB, 0xBBBB, 0xBBBB,
                0xBBBB, 0xBBBB, 0xBBBB, 0xBBBB,
                0xBBBB, 0xBBBB, 0xBBBB, 0xBBBB};
```

Defines:
  `init`, used in chunk 10b.

Afterwards, the parts containing the data are loaded on top of it from the array `shcde`.

10b ⟨*bs: load shcde to all nodes* 10b⟩≡ (3b)

```

for (cnt = 0; cnt < 96; cnt++)
{
  broadcast_memory(WOROOT, 0x1, 0x800 + (cnt * 16), dat, init, 16);
}

/* Load part A */
broadcast_memory(WOROOT, 0x1, 0x800+12, dat, shcde, 9);
/* Load part B */
broadcast_memory(WOROOT, 0x1, 0x800+48, dat, shcde+9, 15);
/* Load part C */
broadcast_memory(WOROOT, 0x1, 0x800+84, dat, shcde+24, 15);
/* Load part D */
broadcast_memory(WOROOT, 0x1, 0x800+192, dat, shcde+39, 171);
/* Load part E */
broadcast_memory(WOROOT, 0x1, 0x800+384, dat, shcde+210, 384);
```

Uses `C` 1b, `cnt` 26a 47a, `init` 10a, `shcde` 7c, and `WOROOT` 3c.

After starting the boot loader, a small delay is added to ensure that the loader has finished.

11a ⟨*bs: start shcde on all nodes* 11a⟩≡ (3b)

```
broadcast_start_program(WOROOT, 0x1, 0x975, 0xFF);
sleep(2);
```

Uses `WOROOT` 3c.

11b ⟨*HF: Function prototypes* 11b⟩≡ (2a) 14a▷

```
extern int boot_sharcs(void);
```

Uses `boot_sharcs` 3b.

3.2 Network informations

Once *booted*, the SHARC clusters signal their presence by setting the bit #8 of the variable `MASTER` (`0x222` in DM). This can be used to detect all attached SHARC modules.

11c ⟨*Network Informations* 11c⟩≡ (3a) 13b▷
  ⟨*Detect SHARCs* 12b⟩

11d ⟨*Defines* 6a⟩+≡ (1a) ◁7b 11e▷

```
/** Variable ``MASTER'' in DM */
#define MASTER          0x222
```

Defines:
  `MASTER`, used in chunk 12b.

Similar to the `er2` library, the number of detected SHARC clusters and their logical and physical addresses are made available to the user.
According to the Fifth protocol for reading/writing data words from/to the SHARCs, the physical address of a SHARC module is the same as the 2181 board it is attached to.

11e ⟨*Defines* 6a⟩+≡ (1a) ◁11d 17▷

```
/** Maximum number of SHARC clusters in the network */
#define MAX_CLUSTERS            64
```

Defines:
  `MAX_CLUSTERS`, used in chunk 12.

12a ⟨*Global variables* 6b⟩+≡ (1a) ◁7c 59▷
```
/** The number of detected SHARC clusters in the network. */
static int number_of_clusters;

/** Array that holds the logical addresses of the clusters
with the physical address as index. */
static int logical_cluster_address[MAX_NODES];
/** Array that holds the physical addresses of the clusters
with the logical address as index. */
static int physical_cluster_address[MAX_CLUSTERS];
```

Defines:
`logical_cluster_address`, used in chunks 12 and 13.
`number_of_clusters`, used in chunks 12 and 13.
`physical_cluster_address`, used in chunks 12 and 13.
Uses `MAX_CLUSTERS` 11e.

12b ⟨*Detect SHARCs* 12b⟩≡ (11c)
```

/** Checks the memory location MASTER (0x222 in DM) for a
* present SHARC cluster for all detected ADSP-2181 nodes.
*/
void detect_sharcs(void)
{
  int i;
  int data[1];

  ⟨ds: initialize address arrays 12c⟩
  for (i = 1; i < get_number_of_nodes(); i++)
  {
    request_memory(get_physical_address(i), MASTER, dat, data, 1);
    if ((data[0] & 0x100) == 0x100)
    {
      ⟨ds: add detected cluster 13a⟩
    }
  }
}
```

Defines:
`detect_sharcs`, used in chunks 14a, 21b, 23, 46, 48c, 57b, and 65a.
Uses `data` 26a 47d and `MASTER` 11d.

12c ⟨*ds: initialize address arrays* 12c⟩≡ (12b)
```
number_of_clusters = 0;
for (i = 0; i < MAX_NODES; i++)
  logical_cluster_address[i] = EMPTY;
for (i = 0; i < MAX_CLUSTERS; i++)
  physical_cluster_address[i] = EMPTY;
```

Uses `logical_cluster_address` 12a, `MAX_CLUSTERS` 11e, `number_of_clusters` 12a,
and `physical_cluster_address` 12a.

13a ⟨*ds: add detected cluster* 13a⟩≡ (12b)
```
physical_cluster_address[number_of_clusters] = get_physical_address(i);
logical_cluster_address[get_physical_address(i)] = number_of_clusters;
number_of_clusters++;
```

Uses `logical_cluster_address` 12a, `number_of_clusters` 12a, and `physical_cluster_address` 12a.

13b ⟨*Network Informations* 11c⟩+≡ (3a) ◁11c
⟨*Get Number of Clusters* 13c⟩
⟨*Get Logical Cluster Address* 13d⟩
⟨*Get Physical Cluster Address* 13e⟩

13c ⟨*Get Number of Clusters* 13c⟩≡ (13b)
```
/** Returns the number of detected SHARC clusters.
* @return The number of clusters
*/
int get_number_of_clusters(void)
{
        return(number_of_clusters);
}
```

Defines:
`get_number_of_clusters`, used in chunks 14a, 21b, 46, 48c, 65a, and 67b.
Uses `number_of_clusters` 12a.

13d ⟨*Get Logical Cluster Address* 13d⟩≡ (13b)
```
/** Converts the physical address to the logical address of a cluster.
* @param physical The physical address of a cluster
* @return The logical address of the cluster
*/
int get_logical_cluster_address(int physical)
{
        return(logical_cluster_address[physical]);
}
```

Defines:
`get_logical_cluster_address`, used in chunk 14a.
Uses `logical_cluster_address` 12a.

13e ⟨*Get Physical Cluster Address* 13e⟩≡ (13b)
```
/** Converts the logical address to the physical address of a cluster.
* @param logical The logical address of a cluster
* @return The physical address of the cluster
*/
int get_physical_cluster_address(int logical)
{
        return(physical_cluster_address[logical]);
}
```

Defines:
  get_physical_cluster_address, used in chunks 14a, 22c, 26b, 47b, 48c, 50b, 65a, and 67b.
Uses physical_cluster_address 12a.

14a ⟨*HF: Function prototypes* 11b⟩+≡ (2a) ◁11b 21a▷

```
extern void detect_sharcs(void);
extern int get_number_of_clusters(void);
extern int get_logical_cluster_address(int);
extern int get_physical_cluster_address(int);
```

Uses detect_sharcs 12b, get_logical_cluster_address 13d, get_number_of_clusters 13c,
  and get_physical_cluster_address 13e.

3.3 Reading and writing

The functions for reading/writing from/to the SHARCs via the Fifth PRS are easily defined. The appropriate *message*—according to [5]—is set up and then sent.

14b ⟨*Reading and Writing* 14b⟩≡ (3a)
  ⟨*Write to SHARC* 14c⟩
  ⟨*Read from SHARC* 15⟩

14c ⟨*Write to SHARC* 14c⟩≡ (14b)

```
/** Writes a single 48-bit data word, consisting
* of the three 16-bit words \a msw, \a bmw and \a lsw,
* to the SHARC cluster on top of the
* 2181 module \a netaddress or the group \a groupnr.
* @param groupnr The group number
* @param netadress Physical address of the 2181 node if groupnr is SINGLE,
* where the SHARC cluster is attached
* @param address_code The ``memory bank'' that the address refers to.
* Address codes range from 1--7, where 1--4 are the internal memories
* of the single SHARCs. Codes 6 and 7 specify the external RAM, starting
* at 0x00400000.
* @param address Address where the 48-bit word should be written to
* @param msw Most significant word
* @param bmw Middle significant word
* @param lsw Least significant word
*/
void write_to_sharc(int groupnr, int netaddress,
                    int address_code, int address,
                    int msw, int bmw, int lsw)
{
  int lData[6];
  int rparl, rparh, rparb, xparl, xparh, xparb;

  if (groupnr == SINGLE)
  {
    lData[0] = 512*1*3+netaddress;
  }
  else
  {
    lData[0] = 512*1*3+0x100+groupnr;
  }

  lData[1] = address_code | 0x8000;
  lData[2] = address;
  lData[3] = msw;
  lData[4] = bmw;
  lData[5] = lsw;

  message_wfr_x(lData, 6, &rparl, &rparh, &rparb,
                        &xparl, &xparh, &xparb);

}
```

Defines:
  `write_to_sharc`, used in chunks 21a, 47c, and 58.
Uses `data` 26a 47d.

15 ⟨*Read from SHARC* 15⟩≡ (14b)

```
/** Reads a single 48-bit data word, consisting
* of the three 16-bit words \a msw, \a bmw and \a lsw,
* from the SHARC cluster on top of the
* 2181 module \a netaddress.
* @param netadress Physical address of the 2181 node, where the
* SHARC cluster is attached
* @param address_code The ``memory bank'' that the address refers to.
* Address codes range from 1--7, where 1--4 are the internal memories
* of the single SHARCs. Codes 6 and 7 specify the external RAM, starting
* at 0x00400000.
* @param address Address where the 48-bit word is read from
* @param msw Most significant word
* @param bmw Middle significant word
* @param lsw Least significant word
*/
void read_from_sharc(int netaddress,
                     int address_code, int address,
                     int *msw, int *bmw, int *lsw)
{
  int lData[6];
  int rparl, rparh, rparb, xparl, xparh, xparb;
```

```
    lData[0] = 0x8800 + (512*address_code) + netaddress;
    lData[1] = address;

    message_wfr_x(lData, 2, &rparl, &rparh, &rparb,
                            &xparl, &xparh, &xparb);

    *msw = (rparb << 8) | rparh;
    *bmw = (rparl << 8) | xparb;
    *lsw = (xparh << 8) | xparl;

  }
```

Defines:
  `read_from_sharc`, used in chunk 21a.
Uses `data` 26a 47d.

3.4 Loading programs

As a first approach, a routine for loading data from a file—consisting of ASCII lines with hex numbers—is developed. Small programs can easily be created with the help of the *G21k* utils assembler `asm21k`. Afterwards, the tool `cdump21k` can be used to dump the object file to text format. The relevant program words can then be copied to a new file, which is loaded to a SHARC by the following function.
This development strategy is a little awkward, of course. Later on, a function for reading binary code, i.e. executables created by the linker `ld21k`, is presented but for now it seems a good way to start. Additionally, `load_sharc_hex_data` provides an interface to all external programs that generate hex data...

16a ⟨*Loading Programs* 16a⟩≡ (3a) 57a▷
  ⟨*Load HEX Data to SHARC* 16b⟩

16b ⟨*Load HEX Data to SHARC* 16b⟩≡ (16a)
```
  /** Loads a file consisting of single lines with
  * SHARC commands in hex format to the SHARC cluster
  * on top of the 2181 module \a netaddress or the
  * group \a groupnr.
  * @param hexfile Name of the hex data file
  * @param groupnr The group number
  * @param netadress Physical address of the 2181 node, where the
  * SHARC cluster is attached
  * @param address_code The ``memory bank'' that the address refers to.
  * Address codes range from 1--7, where 1--4 are the internal memories
  * of the single SHARCs. Codes 6 and 7 specify the external RAM, starting
  * at 0x00400000.
  * @param address Address where the 48-bit words are written to
  */
  int load_sharc_hex_data(char *hexfile,
                          int groupnr,
                            int netaddress,
                            int address_code,
                            int address)
{
  FILE *fHex;
  char input[HEX_LINE_LEN+2];
  char *spos, *epos;
  char hexvalue[HEXVALUE_LEN];
  char hexword[HEXWORD_LEN];
  int slen, msw, bmw, lsw;

  fHex = fopen(hexfile, "r");
  if (fHex == NULL)
  {
    return(ERROR);
  }

  fgets(input, HEX_LINE_LEN, fHex);
  spos = input;
  while ((feof(fHex) == 0) &&
         (*spos != 0) && (*spos != 10))
  {
    ⟨lshd: extract first hex word from line 18a⟩
    while (*spos != 0)
    {
      ⟨lshd: extract pure hex numbers 18b⟩
      ⟨lshd: load hex value to SHARC 19c⟩
      ⟨lshd: extract next hex word from line 20a⟩
    }
    fgets(input, HEX_LINE_LEN, fHex);
    spos = input;
  }

  fclose(fHex);

  return(OK);
}
```

Defines:
  `load_sharc_hex_data`, used in chunks 21a, 22c, 26b, 47, and 48c.
Uses `data` 26a 47d, `file` 2b, `HEX_LINE_LEN` 17, `HEXVALUE_LEN` 17, and `HEXWORD_LEN` 17.

17 ⟨*Defines* 6a⟩+≡ (1a) ◁11e 58e▷

```
/** Maximum number of characters for a line of hex numbers */
#define HEX_LINE_LEN            256
/** Maximum number of characters for a single hex number */
#define HEXVALUE_LEN            16
/** Maximum number of characters for a 16-bit hex word */
```

```
#define HEXWORD_LEN              6
```

Defines:
  HEX_LINE_LEN, used in chunk 16b.
  HEXVALUE_LEN, used in chunk 16b.
  HEXWORD_LEN, used in chunk 16b.

It is assumed that the input line starts with a hex digit, which is set as starting position for the new hex word. The pointer `epos` is set to the end of this word.

18a ⟨*lshd: extract first hex word from line* 18a⟩≡ (16b)

```
epos = spos;
while (isxdigit(*epos) ||
      (*epos == 'x'))
  epos++;
```

The length of the single hex numbers in a line gets restricted to a maximum of 15 digits. This count includes the optional "0x" for specifying a hexadecimal number, which is removed in the next step.

18b ⟨*lshd: extract pure hex numbers* 18b⟩≡ (16b)

```
if ((epos-spos) < 16)
{
  strncpy(hexvalue, spos, epos-spos);
  hexvalue[epos-spos] = 0;
}
else
{
  strncpy(hexvalue, spos, 15);
  hexvalue[15] = 0;
}

/* Remove leading ''0x'' if necessary */
if ((hexvalue[0] == '0') &&
    (hexvalue[1] == 'x'))
  memmove(hexvalue, hexvalue+2, strlen(hexvalue) - 1);
```

⟨*lshd: process MSW of hexvalue* 18c⟩
⟨*lshd: process BMW of hexvalue* 19a⟩
⟨*lshd: process LSW of hexvalue* 19b⟩

In order to be as flexible as possible, it is not required that single hex words have a unified length of 12 hex digits. For the most significant word (MSW) the length of the current `hexvalue` is checked. If it is larger than 8, the hex digits 8–12, as far as they exist, set the value of the MSW. Then, the rest of the digits, i.e. 0–7, are shifted to the left of the string. This guarantees a maximum length of 8 hex characters for processing the middle word.

18c ⟨*lshd: process MSW of hexvalue* 18c⟩≡ (18b)

```
slen = strlen(hexvalue);
if (slen > 8)
{
  strncpy(hexword, hexvalue, slen - 8);
  hexword[slen - 8] = 0;
  sscanf(hexword, "%X", &msw);
  memmove(hexvalue, hexvalue + slen - 8, 8);
  hexvalue[8] = 0;
}
else
  msw = 0x0;
```

Once again the length of the remaining string is taken into account and only the digits 7–4 of the `hexvalue` set the middle word—if they exist.

19a ⟨*lshd: process BMW of hexvalue* 19a⟩≡ (18b)

```
slen = strlen(hexvalue);
if (slen > 4)
{
  strncpy(hexword, hexvalue, slen - 4);
  hexword[slen - 4] = 0;
  sscanf(hexword, "%X", &bmw);
  memmove(hexvalue, hexvalue + slen - 4, 4);
  hexvalue[4] = 0;
}
else
  bmw = 0x0;
```

Now, the `hexvalue` is at most 4 characters long and can be directly converted to the LSW.

19b ⟨*lshd: process LSW of hexvalue* 19b⟩≡ (18b)

```
strncpy(hexword, hexvalue, 4);
hexword[4] = 0;
sscanf(hexword, "%X", &lsw);
```

19c ⟨*lshd: load hex value to SHARC* 19c⟩≡ (16b)

```
write_to_sharc(groupnr, netaddress,
               address_code, address,
               msw, bmw, lsw);
address++;
```

Defines:
  `write_to_sharc`, used in chunks 21a, 47c, and 58.

Starting at the end position within the current line, the pointer spos is progressed until a hex digit or the end of the line is encountered. Then, the new hex word is skipped by the pointer epos. Thus, the single hex words in a line may be separated by whitespace, semicolon or a comma (or any other non-hex character).

20a ⟨*lshd: extract next hex word from line* 20a⟩≡ (16b)

```
  spos = epos;
  while (!isxdigit(*spos) &&
        (*spos != 0))
    spos++;

  epos = spos;
  while ((*epos != 0) &&
         (isxdigit(*epos) ||
          (*epos == 'x')))
    epos++;
```

3.5 Executing programs

20b ⟨*Executing Programs* 20b⟩≡ (3a) 74d▷
  ⟨*Start SHARC program* 20c⟩

Once again just the call of the right *message*.

20c ⟨*Start SHARC program* 20c⟩≡ (20b)

```
  /** Starts an already loaded program on the SHARC with
  * the number \a sharc_id (1--4) from the SHARC cluster
  * on top of the 2181 module \a netaddress or the
  * group \a groupnr.
  * @param groupnr The group number
  * @param netadress Physical address of the 2181 node if groupnr
  * is single, where the SHARC cluster is attached
  * @param sharc_id ID of the single SHARC (1--4)
  * @param start_address_msw Most significant word of the
  * 32-bit start address
  * @param start_address_lsw Least significant word of the
  * 32-bit start address
  */
  void start_sharc_program(int groupnr, int netaddress, int sharc_id,
                           int start_address_msw,
                           int start_address_lsw)
  {
    int lData[6];
    int rparl, rparh, rparb, xparl, xparh, xparb;

    if (groupnr == SINGLE)
```

```
      lData[0] = (512*3) + netaddress;
    else
      lData[0] = (512*3) + 0x100 + groupnr;
    lData[1] = sharc_id | 0x8000;
    lData[2] = 0x0000;
    lData[3] = start_address_msw;
    lData[4] = start_address_lsw;
    lData[5] = 0xffff;

    message_wfr_x(lData, 6, &rparl, &rparh, &rparb,
                            &xparl, &xparh, &xparb);

  }
```

Defines:
  start_sharc_program, used in chunks 21a, 22c, 26b, 47e, 50a, and 58d.

21a ⟨*HF: Function prototypes* 11b⟩+≡ (2a) ◁14a 65c▷

```
  extern void start_sharc_program(int, int, int, int, int);
  extern void write_to_sharc(int, int, int, int, int, int, int);
  extern int load_sharc_hex_data(char *, int, int, int, int);
  extern void read_from_sharc(int, int, int, int *, int *, int *);
```

Uses load_sharc_hex_data 16b, read_from_sharc 15, start_sharc_program 20c,
  and write_to_sharc 14c 19c.

4 A first SHARC program

This little C program loads and starts a small SHARC executable, which switches off the lower of the four LEDs on the board.

21b ⟨*shtest.c* 21b⟩≡

```
  #include <stdio.h>
  #include "../device_driver/er2gdef.h"
  #include "../er2lib/er2.h"
  #include "er2sh.h"

  int main(void)
  {
    verbose_on();

    startup_er2(PARALLEL_PORT);

    printf("Booting SHARCs...\n");
    boot_sharcs();

    printf("Detecting SHARCs...\n");
    detect_sharcs();
  if (get_number_of_clusters() > 0)
  {
    ⟨Toggling LED1 on first SHARC 22c⟩
  }

  shutdown_er2();

  return(0);
}
```

Defines:
  `main`, never used.
Uses `boot_sharcs` 3b, `detect_sharcs` 12b, and `get_number_of_clusters` 13c.

The assembler code that is used in the next to follow chunk was derived from this short assembler program for toggling the LED at Flag0 of SHARC #1.

22a ⟨*led.asm* 22a⟩≡

```
/* MODE2 register */
/* Bit 15: FLAG0 1=output 0=input */
#define FLG0O   0x00008000
/* ASTAT register */
/* Bit 19: FLAG0 value */
#define FLG0  0x00080000

.SEGMENT/PM     seg_pmco;

.GLOBAL         toggle_led;

toggle_led: bit set mode2 FLG0O;
            bit set astat FLG0;

            rts;

.ENDSEG;
```

Defines:
  `FLG0`, never used.
  `FLG0O`, never used.

The corresponding hex data file—created with the assembler `asm21k` and `cdump21k`—looks like this:

22b ⟨*led.hex* 22b⟩≡

```
140A00008000
140C00080000
0A3E00000000
```

22c ⟨*Toggling LED1 on first SHARC* 22c⟩≡ (21b)

```
int proc = get_physical_cluster_address(0);
load_sharc_hex_data("led.hex", SINGLE, proc, 6, 0x0);

/* Start prog */
start_sharc_program(SINGLE, proc, 1, 0x0040, 0x0000);
```

Defines:
  `proc`, used in chunks 26, 47, 48, and 50a.
Uses `get_physical_cluster_address` 13e, `load_sharc_hex_data` 16b, and `start_sharc_program` 20c.

5 Dumping the SHARC IVT

Based on the program `shtest.c` and the assembler file `led.asm`, a small set of routines is developed for dumping the program memory of the first SHARC in the cluster to a file.

23 ⟨*dumpivt.c* 23⟩≡

```
#include <stdio.h>
#include "../device_driver/er2gdef.h"
#include "../er2lib/er2.h"
#include "er2sh.h"

int main(void)
{
  ⟨dit: Variables 26a⟩
  verbose_on();

  startup_er2(PARALLEL_PORT);

  printf("Booting SHARCs...\n");
  boot_sharcs();

  printf("Detecting SHARCs...\n");
  detect_sharcs();

  ⟨dit: start ASM prog on SHARC 26b⟩
  ⟨dit: dump IVT to file 26d⟩

  shutdown_er2();

  return(0);
}
```

Defines:
  `main`, never used.
Uses `boot_sharcs` 3b and `detect_sharcs` 12b.

24a ⟨*dumpivt.asm* 24a⟩≡

```
.SEGMENT/PM     seg_pmco;

.GLOBAL         dump_ivt;

dump_ivt: R7 = 0x400;       /* Number of program words to dump */
          ⟨dit: init DAGs 24b⟩
dmp_wrd:
          ⟨dit: dump program word 24c⟩
          R7 = R7 - 1;
          IF NE JUMP 0x400005;
          ⟨dit: set “finished” flag 25a⟩
          RTS;

.ENDSEG;
```

Uses NE 39b 39b 40a 41a.

24b ⟨*dit: init DAGs* 24b⟩≡ (24a)

```
I8 = 0x20000;     /* Start address */
M8 = 0x1;         /* Step one ahead, each time */
L8 = 0x0;         /* No circular buffer */
R8 = 0x5001;      /* Dumped data starts at 0x1001 DM on 2181 */
```

Uses data 26a 47d.

The single 48-bit program words are subdivided into three 16-bit words. The most significant word (MSW = Bits 47–32) is written first, then the middle word (BMW = Bits 31–16) and finally the least significant word (LSW = Bits 15–0).

24c ⟨*dit: dump program word* 24c⟩≡ (24a)

```
PX = PM(I8, M8);          /* Read program word from IVT to PX */
R9 = PX2;                 /* Copy PX2 to R9 */
R11 = 0xFFFF;
R9 = R9 AND R11;          /* Get lower bits of PX2 = BMW */
R10 = PX2;                /* Copy PX2 to R10 */
R12 = -16;
R10 = LSHIFT R10 BY R12;  /* Get upper bits of PX2 = MSW */
DM(0xCE0000) = R8;        /* Write IDMA address to 2181 */
DM(0xC40000) = R10;       /* Write MSW to 2181 */
R8 = R8 + 1;              /* Step ahead pointer to 2181 RAM */
DM(0xCE0000) = R8;        /* Write IDMA address to 2181 */
DM(0xC40000) = R9;        /* Write BMW to 2181 */
R8 = R8 + 1;              /* Step ahead pointer to 2181 RAM */
DM(0xCE0000) = R8;        /* Write IDMA address to 2181 */
DM(0xC40000) = PX1;       /* Write PX1 = LSW to 2181 */
R8 = R8 + 1;              /* Step ahead pointer to 2181 RAM */
```

The writes to the IDMA bus are based on the following table, showing the connections between the address and data buses of the SHARC cluster to the IDMA port of the 2181:

	SHARC		Socket		2181	
Function	Pin#	Signal	Pin#	Signal	Pin#	Function
A17	36	A17 (IAL)	4d	$\overline{\text{IAL}}$	125	$\overline{\text{IAL}}$
A18	37	A18 ($\overline{\text{IRD}}$)	5e	$\overline{\text{IRD}}$	100	$\overline{\text{IRD}}$
A19	40	A19 ($\overline{\text{IWR}}$)	4e	$\overline{\text{IWR}}$	99	$\overline{\text{IWR}}$
$\overline{\text{IRQ1}}$	237	ESTR	21d	ESTR	81 FPGA	ESTR
$\overline{\text{MS1}}$	56	$\overline{\text{MS1}}$	5d	$\overline{\text{ICS}}$	124	$\overline{\text{IS}}$
$\overline{\text{MS2}}$	55	$\overline{\text{MS2}}$	21c	$\overline{\text{IRQ2}}$	34	$\overline{\text{IRQ2}}$
$\overline{\text{RESET}}$	234	$\overline{\text{RESET}}$	13c	$\overline{\text{RESET}}$	58	$\overline{\text{RESET}}$
$\overline{\text{IRQ2}}$	236	IFLG	5c	IFLG	120	FLAG6
$\overline{\text{DMAR1}}$	70	$\overline{\text{DMAR1}}$	9b	$\overline{\text{IACK}}$	37	$\overline{\text{IACK}}$
D16	158	D16	1b	IAD0	118	IAD0
D17	157	D17	1c	IAD1	117	IAD1
D18	155	D18	1d	IAD2	116	IAD2
D19	154	D19	2a	IAD3	115	IAD3
D20	153	D20	2b	IAD4	114	IAD4
D21	151	D21	2c	IAD5	113	IAD5
D22	150	D22	2d	IAD6	110	IAD6
D23	149	D23	2e	IAD7	109	IAD7
D24	147	D24	3a	IAD8	108	IAD8
D25	146	D25	3b	IAD9	107	IAD9
D26	145	D26	3c	IAD10	106	IAD10
D27	142	D27	3d	IAD11	105	IAD11
D28	141	D28	3e	IAD12	104	IAD12
D29	140	D29	4a	IAD13	103	IAD13
D30	138	D30	4b	IAD14	102	IAD14
D31	137	D31	4c	IAD15	101	IAD15

To signal that all data locations were written, the memory location `0x1000` (DM) of the 2181 is set to “1”.

25a ⟨*dit: set “finished” flag* 25a⟩≡ (24a)

```
R8 = 0x5000;
R9 = 0x1;
DM(0xCE0000) = R8;       /* Set address for IDMA access */
DM(0xC40000) = R9;       /* Write a ''1'' to 2181 */
```

The corresponding hex data file—created with the assembler `asm21k` and `cdump21k`—looks like this:

25b ⟨*dumpivt.hex* 25b⟩≡

```
0F0700000400 0F1800020000 0F2800000001
0F3800000000 0F0800005001 503F6D800000
7DDF84800000 0F0B0000FFFF 013E0004099B
7DDF85000000 0F0CFFFFFFF0 013E00200AAC
110800CE0000 110A00C40000 013E00029880
110800CE0000 110900C40000 013E00029880
110800CE0000 11DC00C40000 013E00029880
013E0002A770 062000400005 0F0800005000
```

```
0F0900000001 110800CE0000 110900C40000
0A3E00000000
```

Now the program is loaded to the SHARC cluster that was detected first during bootup. . .

26a ⟨*dit: Variables* 26a⟩≡ (23) 26c▷

```
int proc, cnt;
int data[4];
```

Defines:
`cnt`, used in chunks 3b, 4a, 10b, 26, 47c, 48c, 65–67, 70, and 71.
`data`, used in chunks 5–7, 12b, 14–16, 24b, 26, 29, 39b, 47e, 48b, 50b, 58e, 59, 66–68, 74, and 75.
`proc`, used in chunks 26, 47, 48, and 50a.

26b ⟨*dit: start ASM prog on SHARC* 26b⟩≡ (23)

```

proc = get_physical_cluster_address(0);
/* Reset ‘finished’ flag and memory */
data[0] = 0x0;
data[1] = 0x0;
data[2] = 0x0;
for (cnt = 0; cnt < 0x400; cnt++)
  broadcast_memory(SINGLE, proc, 0x1001+cnt*3, dat, data, 3);

/* Load program to SHARC */
load_sharc_hex_data("dumpivt.hex", SINGLE, proc, 6, 0x0);

/* Start program */
start_sharc_program(SINGLE, proc, 1, 0x0040, 0x0000);

/* Wait for ‘finished’ flag */
while (data[0] != 1)
{
  request_memory(proc, 0x1000, dat, data, 1);
}
```

Uses `cnt` 26a 47a, `data` 26a 47d, `get_physical_cluster_address` 13e, `load_sharc_hex_data` 16b, `proc` 22c 26a 47a, and `start_sharc_program` 20c.

26c ⟨*dit: Variables* 26a⟩+≡ (23) ◁26a

```
FILE *fDump;
```

Defines:
`fDump`, used in chunk 26d.

The file `ivt.dmp` is opened and the read contents of the SHARCs memory from `0x20000`–`0x20400` are dumped to it.

26d ⟨*dit: dump IVT to file* 26d⟩≡ (23)

```
fDump = fopen("ivt.dmp", "wb");
if (fDump == NULL)
{
  fprintf(stderr, "Could not open ivt.dmp for writing!\n");
  return(1);
}

/* Read data and dump it to file */
for (cnt = 0; cnt < 0x400; cnt++)
{
  request_memory(proc, 0x1001+cnt*3, dat, data, 3);
  fprintf(fDump, "%04X%04X%04X\n", data[0], data[1], data[2]);
}

fclose(fDump);
```

Uses `cnt` 26a 47a, `data` 26a 47d, `fDump` 26c, `file` 2b, and `proc` 22c 26a 47a.

6 Additional runtime system

6.1 Basic layout

The dump of the SHARC IVT shows that it got completely overwritten by the routines of the Fifth PRS, up to address `0x200FD`.
A simple load of an executable, created by the C compiler `g21k` or the linker `ld21k`, would destroy the routines that are used to communicate with the 2181 module below. The address space `0x20000–0x201FF` is used for the interrupt table and some basic tasks, e.g. setting up the processor and the C runtime environment. In the worst case, programs could not even be started once they are loaded.
A few additional routines are now developed, holding the conditions:

1. No use of interrupts. So the C programs that are to be executed can use them as they like.
2. All IVTs and internal memories of the single SHARCs are unoccupied, except a small area from `0x20100–0x202FF`.
3. If they finished, programs can be restarted without a reset of the ER2. This would mean to reboot all SHARCs and load the code again, which should be avoided.
4. While waiting for the next (or first) start of a program, SHARC #4 of the cluster is also responsible for loading new code and data to the single processors.

Like in the original Fifth PRS, complemented by the boot code `shcde.dat`, the single processors each run in a small loop (see fig. 1). If a special memory location gets written, they jump to the specified start address and execute the

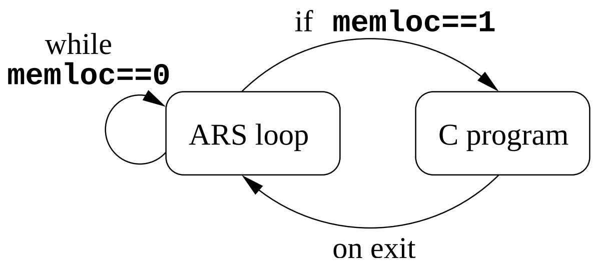

Figure 1: Basic loop for the ARS

loaded program. After the C program is finished, the loop of the *additional runtime system*—called ARS in the following—is entered again.
Although all SHARCs have direct access to the 2181 memory via IDMA in principle, only #4 on each cluster will be responsible for loading data and finally starting the single processes. Otherwise, there would have to be some sort of synchronization between the SHARCs and the 2181. Additionally, the total time for transmitting all data to a cluster could not be reduced this way.
The single C programs can be different and run independently, but they are started at the same time. Since #4 is responsible for the general control flow, it also sets up the common 'go' signal.

> Within the ARS, no direct routines are provided for signalling to the user that a single process has finished. If necessary, this can be achieved by a special result word that is written to the memory of the 2181. It is further assumed that if several processes want to output their results, some sort of synchronization—preferably using the functions from section 6.3—takes place within the C programs itself.

6.2 IDMA reads and writes

The section 5 already presented a table of the available IDMA lines, connected to the SHARC. They can be accessed by all processors of the cluster and be used for data transfer. The development of the ARS loops starts with some embedded routines for the IDMA access.
The memory map for the ARS looks as follows:

Address	**Description**
`0x20100`	Sharc ID
`0x20101`	"1" = Program is running, "0" = Waiting for next start
`0x20102–0x20105`	Reserved for *token passing*
`0x20106`	Reserved for *ARS barrier* synchronization
`0x20107–0x2010F`	Reserved for future use
`0x20110–0x20116`	Function `idma_read`
`0x20117–0x2011B`	Function `idma_write`
`0x2011C–0x20123`	Function `ars_get_id`
`0x20122–0x2012C`	Function `ars_signal`
`0x2013D–0x20157`	Function `ars_wait`
`0x20158–0x2017F`	Function `ars_barrier`
`0x20180–0x20300`	ARS loop

Initial setup for Fifth PRS:

DM address	**Register name**	**Function**	**Initial value**
`0x00`	syscon	System Control	`0x0000A308`
`0x02`	wait	Wait State Configuration	`0x01AD6B41`
`0x18`	bmax	Bus Time-Out Maximum	`0x0000000E`
`0x1C`	dmac6	DMA6 Control Register	`0x00000000`
`0xC6`	lctl	Link Buffer Control	`0x00000000`

Accessing memory via IDMA is done using the specifications in [1, pp. 11-17] and the table from section 5. The `IACK` line is low by default, so the SHARC does not have to set it. `IRD`, `IWR` and `IS` are asserted/deasserted by using an appropriate bit mask for the address...

29a ⟨*ars: Setting the IDMA address* 29a⟩≡ (29)

```
DM(0xCE0000) = R4;     /* Write IDMA address to 2181 */
R0 = DM(0x4C0000);     /* Deassert IAL, assert IS */
```

29b ⟨*ars: Reading 16-bit word via IDMA* 29b⟩≡ (35a)

```
⟨ars: Setting the IDMA address 29a⟩
R0 = PM(0xC80000);     /* Request data, set IS and IRD low */
R8 = PM(0x4C0000);     /* Rise IS and IRD again */
R8 = 0xFFFF;
R0 = R0 AND R8;
RTS;
```

Uses `data` 26a 47d.

29c ⟨*ars: Writing 16-bit word via IDMA* 29c⟩≡ (35a)

```
⟨ars: Setting the IDMA address 29a⟩
DM(0xC40000) = R8;     /* Write data to 2181, set IS and IWR low */
R0 = PM(0x4C0000);     /* Rise IS and IWR again */
RTS;
```

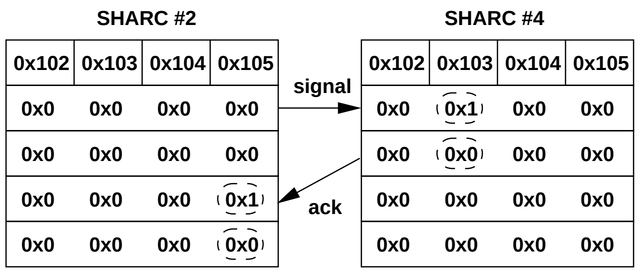

Figure 2: The functions *signal* and *wait*

Uses `data` 26a 47d.

6.3 Synchronizing processes

For synchronizing processes on the SHARC cluster various actions have to be taken. Additionally, the necessary steps depend on the current processor IDs, i.e. which SHARC wants to sync with which remote counterpart.
The following routine reads the SHARCs ID from memory location `0x20100` and stores it in the `R0` register.

30a ⟨*ars: Getting SHARC ID* 30a⟩≡ (35a)
```
  I8 = 0x20100;
  M8 = 0x0;
  L8 = 0x0;
  PX = PM(I8,M8);
  R0 = PX2;
  RTS;
```

It is assumed that SHARC # 2 wants to synchronize with # 4.
He sets `0x20103` at # 4 to "1". Now, # 4 has to enter its "wait" loop, sees the "1" from #2 and sets it back to zero. Then #4 sets `0x20105` at # 2 to "1" and continues.
If # 2 gets this acknowledge, he resets `0x20105` and continues, too.

30b ⟨*ars: Signal* 30b⟩≡ (35a)
  ⟨*arssignal: check ID* 31a⟩
  ⟨*arssignal: compute remote address* 31b⟩
  ⟨*arssignal: send signal* 31c⟩
  ⟨*arssignal: get acknowledge* 31d⟩

```
  signal_end:
    RTS;
```

If the SHARC is to synchronize with itself, no further action is taken.

31a ⟨*arssignal: check ID* 31a⟩≡ (30b)

```
    CALL ars_get_id;
    R11 = R0 - R4;
    IF EQ JUMP signal_end;
```

Uses `EQ` 37a.

The address in *multiprocessor memory* that should be set as flag is computed. The base address is `0x20101` plus the ID of the current processor (R0). Then R4 times `0x80000` is added to reach the correct memory block for the remote processor.

31b ⟨*arssignal: compute remote address* 31b⟩≡ (30b)

```
    R11 = R4;
    M8 = 0x0;
    L8 = 0x0;
    R8 = 0x20101;
    R8 = R8 + R0;
    R9 = 0x80000;
  send_address:
    R8 = R8 + R9;
    R11 = R11 - 1;
    IF NE JUMP send_address;
```

Uses `NE` 39b 39b 40a 41a.

Now, the signal is sent by setting the calculated address to `0x1`.

31c ⟨*arssignal: send signal* 31c⟩≡ (30b)

```
    I8 = R8;
    PX1 = 0x0000;
    PX2 = 0x00000001;
    PM(I8, M8) = PX;
```

After sending a signal, the SHARC waits for the acknowledge from the remote processor. Once he received the necessary flag it is immediately reset to zero.

31d ⟨*arssignal: get acknowledge* 31d⟩≡ (30b)

```
    R8 = 0x20101;
    R8 = R8 + R4;
    I8 = R8;
    R9 = 0x1;
  get_rveack:
    PX = PM(I8, M8);
    R11 = PX2;
    R11 = R11 AND R9;
    IF EQ JUMP get_rveack;
    PX2 = 0x00000000;
    PM(I8, M8) = PX;
```

Uses **EQ** 37a.

On the other side of the synchronization process, the **wait** function has to be called.

32a ⟨*ars: Wait* 32a⟩≡ (35a)

```
  ⟨arswait: check ID 32b⟩
  ⟨arswait: get signal 32c⟩
  ⟨arswait: compute remote address 32d⟩
  ⟨arswait: send acknowledge 33a⟩

  wait_end:
    RTS;
```

It also starts by checking the ID against its own. If they are equal, nothing has to be done for now.

32b ⟨*arswait: check ID* 32b⟩≡ (32a)

```
    CALL ars_get_id;
    R11 = R0 - R4;
    IF EQ JUMP wait_end;
```

Uses **EQ** 37a.

Else, the remote SHARC waits for the signal from the sender process. Once received, it is set to zero immediately.

32c ⟨*arswait: get signal* 32c⟩≡ (32a)

```
    M8 = 0x0;
    L8 = 0x0;
    R8 = 0x20101;
    R8 = R8 + R4;
    R9 = 0x1;
    I8 = R8;
  get_sigack:
    PX = PM(I8, M8);
    R11 = PX2;
    R11 = R11 AND R9;
    IF EQ JUMP get_sigack;
    PX2 = 0x00000000;
    PM(I8, M8) = PX;
```

Uses **EQ** 37a.

Then, the address into multiprocessor memory space for the acknowledge flag is generated. . .

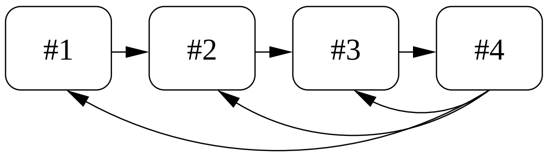

Figure 3: The *barrier* function

32d ⟨*arswait: compute remote address* 32d⟩≡ (32a)

```
    R11 = R4;
    R8 = 0x20101;
    R8 = R8 + R0;
    R9 = 0x80000;
  wait_address:
    R8 = R8 + R9;
    R11 = R11 - 1;
    IF NE JUMP wait_address;
```

Uses NE 39b 39b 40a 41a.

... and the acknowledge is sent.

33a ⟨*arswait: send acknowledge* 33a⟩≡ (32a)

```
    I8 = R8;
    PX1 = 0x0000;
    PX2 = 0x00000001;
    PM(I8, M8) = PX;
```

A *barrier* is a special function whose concept was borrowed from PVM. It does not synchronize two processors against each other but waits until all four SHARCs have reached a marked point in their control flow.

33b ⟨*ars: Barrier* 33b⟩≡ (35a)

```
  ⟨arsbarrier: check ID 34a⟩
  ⟨arsbarrier: wait for signal 34b⟩
  ⟨arsbarrier: send signal 34c⟩
  ⟨arsbarrier: get acknowledge 34d⟩
  ⟨arsbarrier: send acknowledge 34e⟩
  barrier_end:
    RTS;
```

The basic layout of this routine is as follows: SHARC #1 sends a signal to #2, then #2 to #3 and #3 signals to #4. After sending their signals, the SHARCs #1–3 wait until the barrier flag at address `0x20106` is set (see fig. 3). This is done by SHARC #4 after synchronizing with #3. Then, all SHARCs reset the barrier flag and continue with the execution of their current program.
If the ID of the current SHARC is #4, no signal has to be sent. Instead, the barrier acknowledge is transmitted after receiving the signal from #3.

If the ID is #1, the queue of signals is started by syncing with #2.

34a ⟨*arsbarrier: check ID* 34a⟩≡ (33b)

```
    CALL ars_get_id;
    R12 = R0;
    R9 = 0x1;
    R4 = 0x4;
    R8 = R0 AND R4;
    IF NE JUMP barrier_ack;
    R4 = 0x1;
    R8 = R0 - R4;
    IF EQ JUMP barrier_sig;
```

Uses `EQ` 37a and `NE` 39b 39b 40a 41a.

Else,—for #2 and #3—the processor waits for the signal from #1 or #2, respectively.

34b ⟨*arsbarrier: wait for signal* 34b⟩≡ (33b)

```
    R4 = R12 - R9;
    CALL ars_wait;
```

34c ⟨*arsbarrier: send signal* 34c⟩≡ (33b)

```
  barrier_sig:
    R12 = R12 + 1;
    R4 = R12;
    CALL ars_signal;
```

34d ⟨*arsbarrier: get acknowledge* 34d⟩≡ (33b)

```
    I8 = 0x20106;
    M8 = 0x0;
    R9 = 0x1;
  barrier_wait:
    PX = PM(I8, M8);
    R4 = PX2;
    R4 = R4 AND R9;
    IF EQ JUMP barrier_wait;
    PX2 = 0x00000000;
    PM(I8,M8) = PX;
    JUMP barrier_end;
```

Uses `EQ` 37a.

34e ⟨*arsbarrier: send acknowledge* 34e⟩≡ (33b)

```
  barrier_ack:
    R4 = 0x3;
    CALL ars_wait;
    M8 = 0x0;
    L8 = 0x0;
    I8 = 0x0A0106;
    PX1 = 0x0000;
    PX2 = 0x00000001;
    PM(I8,M8) = PX;  /* Set 'barrier' flag for #1 */
    I8 = 0x120106;
    PM(I8,M8) = PX;  /* Set 'barrier' flag for #2 */
    I8 = 0x1A0106;
    PM(I8,M8) = PX;  /* Set 'barrier' flag for #3 */
    I8 = 0x20106;
    PX2 = 0x00000000;
    PM(I8,M8) = PX;  /* Reset 'barrier' flag for #4 */
```

The created routines are now combined to the so called *ARS runtime header* which is then included to the ARS loops.

35a ⟨*ars: Runtime header* 35a⟩≡ (35b 38a)

```
  idma_read:
            ⟨ars: Reading 16-bit word via IDMA 29b⟩
  idma_write:
            ⟨ars: Writing 16-bit word via IDMA 29c⟩
  ars_get_id:
            ⟨ars: Getting SHARC ID 30a⟩
  ars_signal:
            ⟨ars: Signal 30b⟩
  ars_wait:
            ⟨ars: Wait 32a⟩
  ars_barrier:
            ⟨ars: Barrier 33b⟩
```

Uses **idma_read** 39b 39c 39c 39c 40a 40a 40a and **idma_write** 38c 41a.

6.4 Main loops

First, the simpler loop for the SHARCs #1–#3 is developed.

35b ⟨*arssimp.asm* 35b⟩≡

```
  .SEGMENT/PM     seg_ars;

  .GLOBAL         ars_simp;

  ⟨ars: Runtime header 35a⟩
  ars_simp:
            ⟨arss: detach SHARC from IRQs 36a⟩
```

```
                ⟨arss: init 36b⟩
                ⟨arss: set SHARC ID 36c⟩
                ⟨arss: set up DAG registers 36d⟩
ars_loop:
                ⟨arss: wait for start signal 37a⟩
                ⟨arss: redirect C exit routine 37b⟩
                ⟨arss: start C program 37c⟩
                RTS;

.ENDSEG;
```

Uses ars_loop 39b 41a.

The IRQ2 routine of the Fifth PRS uses the same registers R8 and R9 and interferes with the ARS routines in general since all SHARCs listen to this interrupt simultaneously. In a first step, each SHARC gets his interrupts disabled completely. When the C programs start, the necessary IRQs will be activated again by the C runtime header.

36a ⟨*arss: detach SHARC from IRQs* 36a⟩≡ (35b)

```
IMASK = 0x0; /* Disable all interrupts */
```

The address `0x20101` of the internal memory is used as start flag for the single SHARCs. This variable is set to `0x0` at first.

36b ⟨*arss: init* 36b⟩≡ (35b)

```
I8 = 0x20101;
M8 = 0x0;
PX1 = 0x0000;
PX2 = 0x00000000;
PM(I8,M8) = PX;         /* Reset ‘start’ flag */
```

Especially for synchronization purposes, it seems to be essential that each SHARC knows its own ID. The multiprocessor ID is hard-wired on the board and available via the bits 8–10 of the `SYSTAT` register. For being able to access the ID without extracting bits and using the shifter each time, the address `0x20100`—right after the IVT—is used to store this important datum.

36c ⟨*arss: set SHARC ID* 36c⟩≡ (35b)

```
I8 = 0x20100;
R8 = DM(0x03);
R8 = FEXT R8 BY 8:3;
PX1 = 0x0000;
PX2 = R8;
PM(I8,M8) = PX;
```

After receiving the ‘start’ command from the 2181, the flag is set to `0x1` by SHARC #4 and the programs start.

36d ⟨*arss: set up DAG registers* 36d⟩≡ (35b)
```
  I8 = 0x20101;    /* Address of the start flag */
  M8 = 0x0;        /* Do not step further in memory */
  L8 = 0x0;        /* No circular buffer */
  R9 = 0x1;        /* Used for the wait loop */
```

37a ⟨*arss: wait for start signal* 37a⟩≡ (35b)
```
  PX = PM(I8,M8);
  R8 = PX2;
  R8 = R8 AND R9;
  IF EQ JUMP ars_loop;
```

Defines:
`EQ`, used in chunks 31, 32, 34, 40b, 42, and 43.
Uses `ars_loop` 39b 41a.

The general C runtime reset vector looks like this (see `060_hdr.asm`):

```
___lib_RSTI:      NOP;
                  CALL ___lib_setup_c;
                  JUMP _main;
___lib_prog_term: JUMP ___lib_prog_term;
```

After setting up the C runtime environment, the `main` routine is called. If the program finishes, the `exit` routine passes the control flow to the label `___lib_prog_term` which is an infinite loop.

> The jump to `___lib_prog_term` is now redirected to our ARS loops starting at `0x20180`.

37b ⟨*arss: redirect C exit routine* 37b⟩≡ (35b)
```
  PX1 = 0x0180;      /* Start relative to 0x20000 */
  PX2 = 0x063E0002; /* Upper part of 'jump' */
  I8 = 0x20007;      /* Address of infinite loop */
  PM(I8, M8) = PX;  /* Redirect jump */
```

The C program is then started by a direct jump to the reset vector.

37c ⟨*arss: start C program* 37c⟩≡ (35b)
```
  JUMP 0x20004;      /* Start C program */
```

Uses `C` 1b.

The main loop for SHARC #4 is a bit more complicated and supports the actions:

1. Starting the loaded programs on all SHARCS of the cluster.
2. Reading a specified number of program words from the memory of the 2181 and writing it to a SHARCs memory.

38a ⟨*arsmain.asm* 38a⟩≡

```
  .SEGMENT/PM      seg_ars;

  .GLOBAL          ars_main;

  ⟨ars: Runtime header 35a⟩
  ars_main:
            ⟨arsm: detach SHARC from IRQs 38b⟩
            ⟨arsm: init 38c⟩
            ⟨arsm: set SHARC ID 39a⟩
  ars_loop:
            ⟨arsm: wait for a signal 39b⟩
  ars_read:
            ⟨arsm: set up DAG registers 39c⟩
  ars_word:
            ⟨arsm: read program word 40a⟩
  ars_broad:
            ⟨arsm: broadcast program word 40b⟩
  ars_check:
            ⟨arsm: all data read? 41a⟩
  ars_run:
            ⟨arsm: redirect C exit routine 41b⟩
            ⟨arsm: start C programs 42⟩
            RTS;

  .ENDSEG;
```

Uses **ars_check** 40a and **ars_loop** 39b 41a.

SHARC #4 is detached from all IRQs, too.

38b ⟨*arsm: detach SHARC from IRQs* 38b⟩≡ (38a)

```
  IMASK = 0x0;       /* Disable all interrupts */
```

Location `0x1000` in the data memory (DM) of the 2181 is used for the synchronization between the two modules. At the start of the ARS main loop, SHARC #4 sets it to zero.

38c ⟨*arsm: init* 38c⟩≡ (38a)

```
  R4 = 0x5000;              /* Write a ``0'' to 2181 */
  R8 = 0x0;
  CALL idma_write;
  I8 = 0x20101;
  M8 = 0x0;
  PX1 = 0x0000;
  PX2 = 0x00000000;
  PM(I8,M8) = PX;          /* Reset `start' flag */
```

Defines:
  idma_write, used in chunk 35a.

39a ⟨*arsm: set SHARC ID* 39a⟩≡ (38a)

```
  I8 = 0x20100;
  PX1 = 0x0000;
  PX2 = 0x00000004;
  PM(I8,M8) = PX;
```

If the host PC—via the Fifth PRS—writes a "2" to this location, the SHARC #4 starts the loaded C programs. A "1" in `0x1000` (DM) means that data is ready to be loaded, starting at address `0x1001`.

39b ⟨*arsm: wait for a signal* 39b⟩≡ (38a)

```
  R4 = 0x5000;            /* Read 'sync' flag */
  CALL idma_read;
  R10 = R0;               /* Copy it for later use */
  R11 = 0x1;
  R0 = R0 AND R11;        /* Is it a ''1''? */
  IF NE JUMP ars_read;    /* Then read data from 2181 */
  R11 = 0x2;
  R0 = R10 AND R11;       /* Is it a ''2''? */
  IF NE JUMP ars_run;     /* Then start C programs */
  JUMP ars_loop;          /* Else, keep on waiting */
```

Defines:
  `ars_loop`, used in chunks 35b, 37a, and 38a.
  `idma_read`, used in chunk 35a.
  `NE`, used in chunks 24a, 31b, 32d, 34a, and 42.
Uses `C` 1b and `data` 26a 47d.

The 24-bit destination address is read from the memory locations `0x1001` (=MSW) and `0x1002` (=LSW). The next variable at `0x1003` contains the number of 48-bit program words to read.

39c ⟨*arsm: set up DAG registers* 39c⟩≡ (38a)

```
  R4 = 0x5001;            /* Read MSW of address */
  CALL idma_read;
  R9 = R0;                /* Copy MSW */
  R4 = 0x5002;            /* Read LSW of address */
  CALL idma_read;
  R12 = 16;
  R9 = LSHIFT R9 BY R12;  /* Shift MSW to right position... */
  R9 = R9 OR R0;          /* ...and add the LSW. */
  I8 = R9;                /* Set destination address */
  M8 = 0x1;               /* Step one forward at each write */
  L8 = 0x0;               /* No circular buffer */
  R4 = 0x5003;            /* Read number of 48-bit words */
  CALL idma_read;
  R7 = R0;                /* Copy to word counter R7 */
  R4 = 0x5004;            /* Set IDMA address to the start */
```

Defines:
  `idma_read`, used in chunk 35a.

Then the 48-bit words follow, splitted into three 16-bit words each. The *most significant word* (MSW) comes first, then the *middle word* (BMW) and finally the *least significant word* (LSW).

40a ⟨*arsm: read program word* 40a⟩≡ (38a)
```
  CALL idma_read;          /* Read MSW from 2181... */
  R9 = R0;                 /* ... and copy it */
  R4 = R4 + 1;             /* Step ahead pointer to 2181 RAM */
  CALL idma_read;          /* Read middle word (BMW) */
  R12 = 16;
  R9 = LSHIFT R9 BY R12;   /* Shift MSW to right position... */
  R9 = R9 OR R0;           /* ...and add the middle word. */
  PX2 = R9;                /* Set upper program word */
  R4 = R4 + 1;             /* Step ahead pointer to 2181 RAM */
  CALL idma_read;          /* Read LSW from 2181 */
  PX1 = R0;                /* Set lower program word */
  R12 = 0x4;
  R12 = R10 AND R12;
  IF NE JUMP ars_broad;    /* Was a broadcast specified? */
  PM(I8, M8) = PX;         /* Write program word */
  JUMP ars_check;          /* Skip broadcast routine */
```

Defines:
  ars_check, used in chunk 38a.
  idma_read, used in chunk 35a.
  NE, used in chunks 24a, 31b, 32d, 34a, and 42.

By additionally setting bit #2 of the sync value, which equals a '5' in `0x1000` (DM), a broadcast is specified. It can be used to write data to several SHARCs simultaneously.
The single SHARCs are then addressed by the bits 4–7, the address in `0x1001` and `0x1002` may point to internal memory locations only!
Some examples:

Sync value	**Taken action**
0x1	Broadcast data to address `0x1001`, `0x1002`
0x95	Broadcast data to SHARC #1 and #4
0x55	Broadcast data to SHARC #1 and #3
0x65	Broadcast data to SHARC #2 and #3

40b ⟨*arsm: broadcast program word* 40b⟩≡ (38a)
```
    R11 = I8;                /* Save current address */
    R12 = 0x10;
    R12 = R10 AND R12;
    IF EQ JUMP ars_data_b;   /* Skip SHARC #1 */
    R12 = 0x80000;
    R9 = R11 + R12;
    I8 = R9;
    PM(I8,M8) = PX;          /* Write to #1 */
  ars_data_b:
    R12 = 0x20;
    R12 = R10 AND R12;
    IF EQ JUMP ars_data_c;   /* Skip SHARC #2 */
    R12 = 0x100000;
    R9 = R11 + R12;
    I8 = R9;
    PM(I8,M8) = PX;          /* Write to #2 */
  ars_data_c:
    R12 = 0x40;
    R12 = R10 AND R12;
    IF EQ JUMP ars_data_d;   /* Skip SHARC #3 */
    R12 = 0x180000;
    R9 = R11 + R12;
    I8 = R9;
    PM(I8,M8) = PX;          /* Write to #3 */
  ars_data_d:
    R12 = 0x80;
    R12 = R10 AND R12;
    IF EQ JUMP ars_data_end; /* Skip SHARC #4 */
    I8 = R11;
    PM(I8,M8) = PX;          /* Write to #4*/
  ars_data_end:
    R11 = R11 + 1;           /* Progress address counter */
    I8 = R11;                /* And assign it again */
```

Uses `EQ` 37a.

The number of 48-bit words is decremented and checked against zero. If all words have been read, the flag at `0x1000` is reset to zero and the ARS starts to wait for the next command.

41a ⟨*arsm: all data read?* 41a⟩≡ (38a)

```
  R4 = R4 + 1;             /* Step ahead pointer to 2181 RAM */
  R7 = R7 - 1;
  IF NE JUMP ars_word;
  R4 = 0x5000;             /* Write a ''0'' to 2181 */
  R8 = 0x0;
  CALL idma_write;
  JUMP ars_loop;           /* Continue with wait loop */
```

Defines:
`ars_loop`, used in chunks 35b, 37a, and 38a.
`idma_write`, used in chunk 35a.
`NE`, used in chunks 24a, 31b, 32d, 34a, and 42.

The infinite loop at `0x20007` is redirected to the main ARS loop.

41b ⟨*arsm: redirect C exit routine* 41b⟩≡ (38a)

```
  PX1 = 0x0180;      /* Start relative to 0x20000 */
  PX2 = 0x063E0002; /* Upper part of 'jump' */
  I8 = 0x20007;      /* Address of infinite loop */
  PM(I8, M8) = PX;  /* Redirect jump */
```

A “1” is written to the internal memory locations `0x20101` as ‘go’ signal to the other SHARCs.
Similar to the data broadcast, only some of the SHARCs can be started. This feature is selected by addtionally setting bit #3 of the sync value, which equals a ‘A’ in `0x1000` (DM).
The single SHARCs that should be started are then addressed by the bits 4–7.
Some examples:

Sync value	**Taken action**
0x2	Starting all SHARCs simultaneously
0x8A	Start SHARC #4
0x3A	Start SHARC #1 and #2
0xEA	Start SHARC #2, #3 and #4

42 ⟨*arsm: start C programs* 42⟩≡ (38a)

```
    M8 = 0x0;
    PX1 = 0x0000;
    PX2 = 0x00000001;
    R12 = 0x8;
    R12 = R10 AND R12;
    IF NE JUMP ars_start_a; /* Start feature, specified? */
    R12 = 0xF0;
    R10 = R10 OR R12;       /* No, so start all SHARCs */
  ars_start_a:
    R12 = 0x10;
    R12 = R10 AND R12;
    IF EQ JUMP ars_start_b; /* Skip SHARC #1 */
    I8 = 0x0A0101;
    PM(I8,M8) = PX;         /* Set ‘start’ flag for #1 */
  ars_start_b:
    R12 = 0x20;
    R12 = R10 AND R12;
    IF EQ JUMP ars_start_c; /* Skip SHARC #2 */
    I8 = 0x120101;
    PM(I8,M8) = PX;         /* Set ‘start’ flag for #2 */
  ars_start_c:
    R12 = 0x40;
    R12 = R10 AND R12;
    IF EQ JUMP ars_start_d; /* Skip SHARC #3 */
    I8 = 0x1A0101;
    PM(I8,M8) = PX;         /* Set ‘start’ flag for #3 */
  ars_start_d:
    R12 = 0x80;
    R12 = R10 AND R12;
    IF EQ JUMP ars_main;    /* Skip SHARC #4, restart wait loop */
    I8 = 0x20101;
    PX1 = 0x0000;
    PX2 = 0x00000001;
    PM(I8,M8) = PX;         /* Set ‘start’ flag for #4 */
    JUMP 0x20004;           /* Start C program */
```

Uses `C` 1b, `EQ` 37a, and `NE` 39b 39b 40a 41a.

6.5 Short test programs

This assembler program toggles the LED 0, depending on the value of memory location `0x20400`.

43 ⟨*ledtog.asm* 43⟩≡

```
/* MODE2 register */
/* Bit 15: FLAG0 1=output 0=input */
#define FLG0O   0x00008000
/* ASTAT register */
/* Bit 19: FLAG0 value */
#define FLG0  0x00080000

.SEGMENT/PM     seg_init;

.GLOBAL         toggle_led;

toggle_led:
            bit set mode2 FLG0O;
            I8 = 0x20400;
            M8 = 0x0;
            L8 = 0x0;
            PX = PM(I8, M8);
            R8 = PX2;
            R9 = 0x1;
            R9 = R8 AND R9;
            IF EQ JUMP set_on;
            bit clr astat FLG0;
            PX1 = 0x0000;
            R9 = 0x0;
            PX2 = R9;
            PM(I8, M8) = PX;
            JUMP end;
set_on:
            PX1 = 0x0000;
            R9 = 0x1;
            PX2 = R9;
            bit set astat FLG0;
            PM(I8, M8) = PX;
end:
            rts;

.ENDSEG;
```

Defines:

`FLG0`, never used.
`FLG00`, never used.
Uses `EQ` 37a.

6.6 Boot code for ARS

This assembler program redirects the program loops of the Fifth PRS to the ARS loops on all the processors.

44 ⟨*bootars.asm* 44⟩≡

```
.SEGMENT/PM     seg_init;

.GLOBAL         boot_ars;

boot_ars:
            M8 = 0x1;
            L8 = 0x0;
            I8 = 0x120004;  /* Boot #2 */
            PX2 = 0x00000000;
            PX1 = 0x0000;
            PM(I8, M8) = PX;
            PM(I8, M8) = PX;
            PM(I8, M8) = PX;
            PX2 = 0x063E0002;
            PX1 = 0x0180;
            PM(I8, M8) = PX;
            I8 = 0x120102;
            PX2 = 0x00000000;
            PX1 = 0x0000;
            PM(I8, M8) = PX;
            PM(I8, M8) = PX;
            PM(I8, M8) = PX;
            PM(I8, M8) = PX;
            PM(I8, M8) = PX;
            PX2 = 0x00020180;
            PX1 = 0xFFFF;
            I8 = 0x120000;
            PM(I8, M8) = PX;    /* Write address -> start */

            I8 = 0x1A0004;  /* Boot #3 */
            PX2 = 0x00000000;
            PX1 = 0x0000;
            PM(I8, M8) = PX;
            PM(I8, M8) = PX;
            PM(I8, M8) = PX;
            PX2 = 0x063E0002;
            PX1 = 0x0180;
            PM(I8, M8) = PX;
            I8 = 0x1A0102;
```

```
PX2 = 0x00000000;
PX1 = 0x0000;
PM(I8, M8) = PX;
PM(I8, M8) = PX;
PM(I8, M8) = PX;
PM(I8, M8) = PX;
PM(I8, M8) = PX;
PX2 = 0x00020180;
PX1 = 0xFFFF;
I8 = 0x1A0000;
PM(I8, M8) = PX;    /* Write address -> start */

I8 = 0x220004;  /* Boot #4 */
PX2 = 0x00000000;
PX1 = 0x0000;
PM(I8, M8) = PX;
PM(I8, M8) = PX;
PM(I8, M8) = PX;
PX2 = 0x063E0002;
PX1 = 0x0180;
PM(I8, M8) = PX;
I8 = 0x220102;
PX2 = 0x00000000;
PX1 = 0x0000;
PM(I8, M8) = PX;
PM(I8, M8) = PX;
PM(I8, M8) = PX;
PM(I8, M8) = PX;
PM(I8, M8) = PX;
PX2 = 0x00020180;
PX1 = 0xFFFF;
I8 = 0x220000;
PM(I8, M8) = PX;    /* Write address -> start */

I8 = 0x20004;  /* Boot #1 */
PX2 = 0x00000000;
PX1 = 0x0000;
PM(I8, M8) = PX;
PM(I8, M8) = PX;
PM(I8, M8) = PX;
PX2 = 0x063E0002;
PX1 = 0x0180;
PM(I8, M8) = PX;
I8 = 0x20102;
PX2 = 0x00000000;
PX1 = 0x0000;
PM(I8, M8) = PX;
PM(I8, M8) = PX;
PM(I8, M8) = PX;
PM(I8, M8) = PX;
```

```
            PM(I8, M8) = PX;

            JUMP 0x20180;
end:
            rts;

.ENDSEG;
```

Uses `boot_ars` 65a.

The program **`arstest`** installs the ARS and tests the "restart" feature. On all four SHARCs the program `ledtog` is run and restarted again. Thus, with every press of the **`Return`** key, all LEDs of the cluster are switched on and off, respectively.

46 ⟨*arstest.c* 46⟩≡

```
#include <stdio.h>
#include "../device_driver/er2gdef.h"
#include "../er2lib/er2.h"
#include "er2sh.h"

int main(void)
{
  ⟨at: Variables 47a⟩
  verbose_on();

  startup_er2(PARALLEL_PORT);

  display_routing_table();

  printf("Booting SHARCs...\n");
  boot_sharcs();

  printf("Detecting SHARCs...\n");
  detect_sharcs();
  printf("%d SHARC clusters found!\n", get_number_of_clusters());

  printf("Transferring ASR...\n");
  ⟨at: transfer ASR to SHARCs 47b⟩
  printf("Patching IVTs and loading programs...\n");
  ⟨at: patch IVTs and load progs 47c⟩
  ⟨at: start ASR on SHARCs 47e⟩
  ⟨at: restart loop 48b⟩

  shutdown_er2();

  return(0);
}
```

Defines:

main, never used.
Uses boot_sharcs 3b, detect_sharcs 12b, and get_number_of_clusters 13c.

47a ⟨*at: Variables* 47a⟩≡ (46) 47d▷
```
int proc, cnt;
```

Defines:
cnt, used in chunks 3b, 4a, 10b, 26, 47c, 48c, 65–67, 70, and 71.
proc, used in chunks 26, 47, 48, and 50a.

47b ⟨*at: transfer ASR to SHARCs* 47b⟩≡ (46)
```

proc = get_physical_cluster_address(0);
/* Load programs to SHARC */
load_sharc_hex_data("arssimp.hex", SINGLE, proc, 3, 0x20110);
load_sharc_hex_data("arssimp.hex", SINGLE, proc, 2, 0x20110);
load_sharc_hex_data("arssimp.hex", SINGLE, proc, 1, 0x20110);
load_sharc_hex_data("arsmain.hex", SINGLE, proc, 4, 0x20110);
```

Uses arsmain 59, arssimp 59, get_physical_cluster_address 13e, load_sharc_hex_data 16b, and proc 22c 26a 47a.

47c ⟨*at: patch IVTs and load progs* 47c⟩≡ (46)
```

for (cnt = 1; cnt < 5; cnt++)
{
  load_sharc_hex_data("ledtog.hex", SINGLE, proc, cnt, 0x20300);
  write_to_sharc(SINGLE, proc, cnt, 0x20005, 0x06BE, 0x0002, 0x0300);
  write_to_sharc(SINGLE, proc, cnt, 0x20006, 0x0000, 0x0000, 0x0000);
  write_to_sharc(SINGLE, proc, cnt, 0x20007, 0x063E, 0x0002, 0x0007);
  write_to_sharc(SINGLE, proc, cnt, 0x20400, 0x0000, 0x0000, 0x0000);
  write_to_sharc(SINGLE, proc, cnt, 0x20101, 0x0000, 0x0000, 0x0000);
  write_to_sharc(SINGLE, proc, cnt, 0x20102, 0x0000, 0x0000, 0x0000);
  write_to_sharc(SINGLE, proc, cnt, 0x20103, 0x0000, 0x0000, 0x0000);
  write_to_sharc(SINGLE, proc, cnt, 0x20104, 0x0000, 0x0000, 0x0000);
  write_to_sharc(SINGLE, proc, cnt, 0x20105, 0x0000, 0x0000, 0x0000);
  write_to_sharc(SINGLE, proc, cnt, 0x20106, 0x0000, 0x0000, 0x0000);
}
```

Uses cnt 26a 47a, load_sharc_hex_data 16b, proc 22c 26a 47a, and write_to_sharc 14c 19c.

47d ⟨*at: Variables* 47a⟩+≡ (46) ◁47a 48a▷
```
int data[4];
```

Defines:
data, used in chunks 5–7, 12b, 14–16, 24b, 26, 29, 39b, 47e, 48b, 50b, 58e, 59, 66–68, 74, and 75.

47e ⟨*at: start ASR on SHARCs* 47e⟩≡ (46)

```
/* Start programs */
fprintf(stderr, "ARS installed ... ready to start? \n");
fgets(ans, 4, stdin);

start_sharc_program(SINGLE, proc, 2, 0x0002, 0x0180);
start_sharc_program(SINGLE, proc, 3, 0x0002, 0x0180);
start_sharc_program(SINGLE, proc, 4, 0x0002, 0x0180);
start_sharc_program(SINGLE, proc, 1, 0x0002, 0x0180);

fprintf(stderr, "ARS hopefully running...ready to start LED program?\n");
fgets(ans, 4, stdin);

data[0] = 0x2;
broadcast_memory(SINGLE, proc, 0x1000, dat, data, 1);
```

Uses `ans` 48a, `data` 26a 47d, `proc` 22c 26a 47a, and `start_sharc_program` 20c.

48a ⟨*at: Variables* 47a⟩+≡ (46) ◁47d

```
char ans[8];
```

Defines:
`ans`, used in chunks 47, 48, and 50a.

48b ⟨*at: restart loop* 48b⟩≡ (46)

```
ans[0] = 0;
while (1)
{

  fprintf(stderr, "Waiting for end of programs...");
  data[0] = 0x1;
  while (data[0] != 0x0)
    request_memory(proc, 0x1000, dat, data, 1);

  fprintf(stderr, "Enter q to quit loop, else programs are restarted.\n");
  fgets(ans, 4, stdin);
  if (ans[0] == 'q')
    break;


  data[0] = 0x2;
  broadcast_memory(SINGLE, proc, 0x1000, dat, data, 1);

}
```

Uses `ans` 48a, `data` 26a 47d, and `proc` 22c 26a 47a.

The program `simptest` was used for debugging and led to the discovery of a small error in `shcde.dat` (see section 6.7).

48c ⟨*simptest.c* 48c⟩≡

```
#include <stdio.h>
#include "../device_driver/er2gdef.h"
#include "../er2lib/er2.h"
#include "er2sh.h"

⟨st: dump data memory to file 50b⟩

int main(void)
{
  char ans[6];
  int proc;
  int cnt = 1;
  int msw, bmw, lsw;

  verbose_on();

  startup_er2(PARALLEL_PORT);

  display_routing_table();

  printf("Booting SHARCs...\n");
  boot_sharcs();

  printf("Detecting SHARCs...\n");
  detect_sharcs();
  printf("%d SHARC clusters found!\n", get_number_of_clusters());

  proc = get_physical_cluster_address(0);

  printf("Using cluster on top of %d ...\n", proc);

  load_sharc_hex_data("ledtog.hex", SINGLE, proc, 1, 0x20300);
  load_sharc_hex_data("ledtog.hex", SINGLE, proc, 2, 0x20300);
  load_sharc_hex_data("ledtog.hex", SINGLE, proc, 3, 0x20300);
  load_sharc_hex_data("ledtog.hex", SINGLE, proc, 4, 0x20300);

  printf("Programs loaded...\n");
  fgets(ans, 4, stdin);

  do
  {
    ⟨st: start all progs 50a⟩
  }
  while (ans[0] != 'q');

  shutdown_er2();

  return(0);
}
```

Defines:
  main, never used.
Uses **ans** 48a, **boot_sharcs** 3b, **cnt** 26a 47a, **detect_sharcs** 12b, **get_number_of_clusters** 13c, **get_physical_cluster_address** 13e, **load_sharc_hex_data** 16b, and **proc** 22c 26a 47a.

50a ⟨*st: start all progs* 50a⟩≡ (48c)

```
start_sharc_program(SINGLE, proc, 4, 0x0002, 0x0300);
/*
start_sharc_program(SINGLE, proc, 3, 0x0002, 0x0300);
start_sharc_program(SINGLE, proc, 2, 0x0002, 0x0300);
start_sharc_program(SINGLE, proc, 1, 0x0002, 0x0300);
*/
fgets(ans, 4, stdin);
```

Uses **ans** 48a, **proc** 22c 26a 47a, and **start_sharc_program** 20c.

50b ⟨*st: dump data memory to file* 50b⟩≡ (48c)

```
void file_dump(int filecnt)
{
  char fname[50];
  FILE *fout;
  int address = 0x0;
  int data[5];

  sprintf(fname, "%d", filecnt);
  strcat(fname, ".dat");

  fout = fopen(fname, "w");
  if (fout == NULL)
  {
    fprintf(stderr, "Could not open file <%s>!\n", fname);
    exit(1);
  }

  for (; address < 0x1000; address++)
  {
    request_memory(get_physical_cluster_address(0), address, dat, data, 1);
    fprintf(fout, "0x%04X: %06X\n", address, data[0]);
  }

  fclose(fout);

}
```

Defines:
  file_dump, never used.
Uses **data** 26a 47d, **file** 2b, and **get_physical_cluster_address** 13e.

6.7 Patching the SHARC bootcode `shcde.dat`

Several tests with the given simple 'LED toggle' programs and `simptest.c` showed that code could not safely be executed on any of the four processors. A single program could be restarted exactly 18 times by the function `start_sharc_program`, then it would fail.
Further investigation of the *runtime systems* for the ADSP-2181 and the SHARCs revealed a bug in `shcde.dat`.
For the communication to the SHARCs a small memory area `0x300-0x3FF` (DM) on the 2181 is used (see [4, p. 2]). It is organized as circular buffer with `0x213` and `0x215` (DM) holding the buffer pointer, respectively.
Every time a 48-bit word is sent to the SHARC—either for normal transfer of data, or for starting a program—it is split up into three 16-bit words. They are written to the buffer, then an `IRQ2` is triggered on the SHARC.
The SHARC gets the pointer into the circular buffer by an IDMA read from `0x215` (DM) in the 2181s memory. It then reads the three 16-bit words via IDMA and combines them to a 48-bit word again.
Problems arise because of the necessary wrap-around at the end of the buffer. Its length 256 gives a remainder on division by 3, so it happens that the pointer is set to `0x3FD` or `0x3FE` when trying to transfer a 48-bit word. The 2181 performs the wrap-around correctly by using:

```
[04b4] modify (i7,m6)
[04b5] ax0= dm(0x0213)      ; get p
[04b6] ay0 = 0x0001
[04b7] ar = ax0 + ay0       ; p = p + 1
[04b8] dm(i7,m5) = ar
[04b9] ax0 = dm(i7,m5)
[04ba] ay0 = 0x00ff         ; p = p & 0xFF
[04bb] ar = ax0 and ay0
[04bc] dm(i7,m6) = ar
[04bd] dm(i7,m5) = 0x0300
[04be] ax0 = dm(i7,m7)
[04bf] ay0 = dm(i7,m5)
[04c0] ar = ax0 or ay0      ; p = p | 0x300
[04c1] dm(i7,m5) = ar
[04c2] ax0 = dm(i7,m5)
[04c3] dm(0x0213) = ax0     ; assign new p
```

every time a new 16-bit value is added to the buffer (see `prs/2181/asm.dmp`). The SHARC reads the address to the communication buffer only once from `0x215` (DM)—right at the start of the IRQ2 routine (see `prs/shcde/shcde.dis`).

```
irq2:
  r8=0x4215;
  dm(0xce0000)=r8;
  r8=pm(0xc80000);
  r9=0x4000;
  r8=r8+r9;
  dm(0xce0000)=r8;
  r8=pm(0xc80000);
  r9=0x7fff;
  r9=r8 and r9;
  btst r8 by 0xf;
  if not sz jump read_sharc;
  r8=0xff;
  r8=r9 and r8;
  r8=r8-1;
  r9=lshift r9 by 0xf8;
  r10=pm(0xc80000);
  call 0x20057;
  call read_2181;
  rti;
```

Then, if a read from the 2181 memory to the SHARC is specified, the function `read_2181` is called (see `prs/shcde/shcde.dis`):

```
read_2181:
  r9=pm(0xc80000);
  r9=lshift r9 by 0x10;
  r10=pm(0xc80000);
  r11=0xffff;
  r10=r10 and r11;
  r9=r9 or r10;
  px2=r9;
  px1=pm(0xc80000);
  pm(0,i8)=px;
  modify(i8,m8);
  r9=0x3;
  r8=r8-r9;
  if ne jump read_2181;
  rts;
```

While reading the three 16-bit values via IDMA, no wrap-around is performed. So, if the current 48-bit word starts at `0x3FE` or `0x3FF` the SHARC

reads	**instead of**
0x3FE, 0x3FF, 0x400	0x3FE, 0x3FF, 0x300
0x3FF, 0x400, 0x401	0x3FF, 0x300, 0x301

The same holds for a read from the SHARC to the ADSP-2181 which starts with:

```
read_sharc:
  r10=pm(0xc80000);
  call 0x20057;
  px=pm(0,i8);
     ...
```

If the communication pointer is set to `0x3FF`, the SHARC reads `0x400` (DM) to the register `R10`, instead of `0x300`.

The following chunks present the new part E (`0x20080`–`0x200FF`) of the patched SHARC runtime system `shcde.dat`.

53a ⟨*shcdee.asm* 53a⟩≡
```
.SEGMENT/PM      seg_init;

.GLOBAL          shcde_parte;

shcde_parte:
```
⟨*shcdee: increase buffer pointer* 54a⟩
⟨*shcdee: IRQ2 routine* 53b⟩
⟨*shcdee: read a 48-bit word from 2181 memory* 54b⟩
⟨*shcdee: read a 48-bit word from SHARC memory* 55a⟩
⟨*shcdee: shifting NOPs* 54c⟩
⟨*shcdee: general setup routine* 55b⟩
⟨*shcdee: set IRQ1I edge-sensitive* 56a⟩
⟨*shcdee: wait for start address* 56b⟩
⟨*shcdee: redirect IRQ2* 54d⟩
```
.ENDSEG;
```

Another register `R12` has to be used, for storing the pointer to the communication buffer.
It is set at the start of the IRQ2 routine:

53b ⟨*shcdee: IRQ2 routine* 53b⟩≡ (53a)
```
irq2:
  r12=0x4215;
  dm(0xce0000)=r12;
  r12=pm(0xc80000);
  r9=0x4000;
  r12=r12+r9;
  dm(0xce0000)=r12;
  r8=pm(0xc80000);
  call inc_pointer;
  r10=pm(0xc80000);
  r9=0x7fff;
  r9=r8 and r9;
  btst r8 by 0xf;
  if not sz jump read_sharc;
  r8=0xff;
  r8=r9 and r8;
  r8=r8-1;
  r9=lshift r9 by 0xf8;
  call 0x20057;
  call read_2181;
  rti;
```

The routine `inc_pointer` is responsible for increasing the pointer to the communication buffer by one. A wrap-around is performed and the new IDMA address is set.

54a ⟨*shcdee: increase buffer pointer* 54a⟩≡ (53a)

```
inc_pointer:
  r12=r12+1;
  r11=0xff;
  r12=r12 and r11;
  r11=0x4300;
  r12=r12+r11;
  dm(0xce0000)=r12;
  rts;
```

54b ⟨*shcdee: read a 48-bit word from 2181 memory* 54b⟩≡ (53a)

```
read_2181:
  call inc_pointer;
  r9=pm(0xc80000);
  r9=lshift r9 by 0x10;
  call inc_pointer;
  r10=pm(0xc80000);
  r11=0xffff;
  r10=r10 and r11;
  r9=r9 or r10;
  px2=r9;
  call inc_pointer;
  px1=pm(0xc80000);
  pm(0,i8)=px;
  modify(i8,m8);
  r9=0x3;
  r8=r8-r9;
  if ne jump read_2181;
  rts;
```

The following three NOPs are added as "buffer" that shifts the remaining functions towards the end of part E. This way `setup` and `shcde_wait` keep their addresses and the calls in the reset vector (`0x20005` and `0x20006`) do not have to be changed.

54c ⟨*shcdee: shifting NOPs* 54c⟩≡ (53a)

```
shift_nops:
  nop;
  nop;
  nop;
```

The address of the IRQ2 routine changed from `0x2009B` to `0x20087`.

54d ⟨*shcdee: redirect IRQ2* 54d⟩≡ (53a)

```
  redirect_irq2:
    px2=0x63e0002;
    px1=0x20087;
    pm(0x20018)=px;
    rts;
```

The command

```
r10=pm(0xc80000);
```

at the start of `read_sharc` was shifted to the IRQ2 routine.

55a ⟨*shcdee: read a 48-bit word from SHARC memory* 55a⟩≡ (53a)

```
  read_sharc:
    call 0x20057;
    px=pm(0,i8);
    r9=px2;
    r9=lshift r9 by 0xe8;
    r8=0x8000;
    r9=bset r9 by 0xf;
    call 0x2006b;
    r9=px2;
    r9=lshift r9 by 0xf8;
    call 0x2006b;
    r9=px2;
    r8=0xff;
    r9=r8 and r9;
    r8=0x8100;
    r9=r8 or r9;
    call 0x2006b;
    r9=px1;
    call 0x2006b;
    rti;
```

The rest of the functions were kept unchanged.

55b ⟨*shcdee: general setup routine* 55b⟩≡ (53a)

```
  setup:
    m7=0x1;
    m6=0xffffffff;
    l7=0;
    m15=0;
    r0=0x1ad6b41;
    dm(0x2)=r0;
    r0=0xe;
    dm(0x18)=r0;
    r0=0xa308;
    dm(0)=r0;
    r0=0;
    dm(0xc6)=r0;
    r0=0;
    dm(0x1c)=r0;
    dm(0x20000)=r0;
    r0=pm(0x3);
    r1=0x600;
    r1=r0 and r1;
    if ne jump set_irq1i;
    r0=0x4300;
    dm(0xce0000)=r0;
    call 0x2004b;
    call redirect_irq2;
    tperiod=0x10000;
    mode2=0x8024;
    irptl=0;
    imask=0x50;
    astat=0;
    rts;
```

56a ⟨*shcdee: set IRQ1I edge-sensitive* 56a⟩≡ (53a)

```
  set_irq1i:
    mode2=0x8002;
    imask=0x80;
    astat=0;
    rts;
```

56b ⟨*shcdee: wait for start address* 56b⟩≡ (53a)

```
  shcde_wait:
    mode1=0;
    nop;
    nop;
    px=pm(0x20000);
    r0=px2;
    r1=px1;
    mode1=0x1000;
    r0=r0 or r0;
    if eq jump add_wait_jump;
    r1=bset r1 by 0x11;
    i7=r1;
    i15=r0;
    r15=dm(0,i7);
    call(m15,i15);
    dm(0,i7)=r15;
  mode1=0;
  nop;
  nop;
  px1=i7;
  px2=0;
  pm(0x20000)=px;
  mode1=0x1000;
  jump shcde_wait;
add_wait_jump:
  jump shcde_wait;
  rts;
```

Address	Description
0x20080–0x20086	Function inc_pointer
0x20087–0x20097	Function read_2181
0x20098–0x2009B	Function redirect_irq2
0x2009C–0x200AF	Function irq2
0x200B0–0x200C3	Function read_sharc
0x200C4–0x200DF	Function setup
0x200E0–0x200E3	Function set_irq1i
0x200E4–0x200FB	Function shcde_wait

6.8 Final ARS start routine

57a ⟨*Loading Programs* 16a⟩+≡ (3a) ◁16a 65d▷

```
  ⟨Start ARS 57b⟩
  ⟨Boot with ARS 65a⟩
```

57b ⟨*Start ARS* 57b⟩≡ (57a)

```
/** Installs the ARS on all detected SHARC clusters.
* @warning Assumes that boot_sharcs and detect_sharcs have been called before!
*/
static void install_ars()
{
  int wordCnt;

  ⟨ia: load arssimp.hex 58a⟩
  ⟨ia: load arsmain.hex 58b⟩
  ⟨ia: load bootars.hex 58c⟩
  ⟨ia: start bootars.hex 58d⟩
}
```

Defines:
  install_ars, used in chunk 65a.
Uses boot_sharcs 3b and detect_sharcs 12b.

58a ⟨*ia: load arssimp.hex* 58a⟩≡ (57b)
```
/* Load arssimp.hex to #1--#3 */
for (wordCnt = 0; wordCnt < ARSSIMP_LENGTH/3; wordCnt++)
{
  write_to_sharc(ALL_CLUSTERS, 0x1, 3, 0x20110 + wordCnt,
                 arssimp[wordCnt*3],
                 arssimp[wordCnt*3+1],
                 arssimp[wordCnt*3+2]);
  write_to_sharc(ALL_CLUSTERS, 0x1, 2, 0x20110 + wordCnt,
                 arssimp[wordCnt*3],
                 arssimp[wordCnt*3+1],
                 arssimp[wordCnt*3+2]);
  write_to_sharc(ALL_CLUSTERS, 0x1, 1, 0x20110 + wordCnt,
                 arssimp[wordCnt*3],
                 arssimp[wordCnt*3+1],
                 arssimp[wordCnt*3+2]);
}
```
Uses `ALL_CLUSTERS` 65b, `arssimp` 59, `ARSSIMP_LENGTH` 58e, and `write_to_sharc` 14c 19c.

58b ⟨*ia: load arsmain.hex* 58b⟩≡ (57b)
```
/* Load arsmain.hex to #4 */
for (wordCnt = 0; wordCnt < ARSMAIN_LENGTH/3; wordCnt++)
  write_to_sharc(ALL_CLUSTERS, 0x1, 4, 0x20110 + wordCnt,
                 arsmain[wordCnt*3],
                 arsmain[wordCnt*3+1],
                 arsmain[wordCnt*3+2]);
```
Uses `ALL_CLUSTERS` 65b, `arsmain` 59, `ARSMAIN_LENGTH` 58e, and `write_to_sharc` 14c 19c.

58c ⟨*ia: load bootars.hex* 58c⟩≡ (57b)
```
/* Load boot program bootars.hex to #1 ... */
for (wordCnt = 0; wordCnt < BOOTARS_LENGTH/3; wordCnt++)
  write_to_sharc(ALL_CLUSTERS, 0x1, 1, 0x20300 + wordCnt,
                 bootars[wordCnt*3],
                 bootars[wordCnt*3+1],
                 bootars[wordCnt*3+2]);
```
Uses `ALL_CLUSTERS` 65b, `bootars` 59, `BOOTARS_LENGTH` 58e, and `write_to_sharc` 14c 19c.

58d ⟨*ia: start bootars.hex* 58d⟩≡ (57b)
```
/* ... and start it. */
start_sharc_program(ALL_CLUSTERS, 0x1, 1, 0x0002, 0x0300);
```
Uses `ALL_CLUSTERS` 65b and `start_sharc_program` 20c.

Now, the contents of `arsmain.hex`, `arssimp.hex` and `bootars.hex` are added to the library.

58e ⟨*Defines* 6a⟩+≡ (1a) ◁17 69a▷

```
/** Length of the ``arsmain.hex'' data */
#define ARSMAIN_LENGTH          714
/** Length of the ``arssimp.hex'' data */
#define ARSSIMP_LENGTH          414
/** Length of the ``bootars.hex'' data */
#define BOOTARS_LENGTH          252
```

Defines:
  ARSMAIN_LENGTH, used in chunks 58b and 59.
  ARSSIMP_LENGTH, used in chunks 58a and 59.
  BOOTARS_LENGTH, used in chunks 58c and 59.
Uses **arsmain** 59, **arssimp** 59, **bootars** 59, and **data** 26a 47d.

59 ⟨*Global variables* 6b⟩+≡ (1a) ◁12a 70c▷

```
/** The data of ``arsmain.hex'' in ER2-readable form. */
static int arsmain[ARSMAIN_LENGTH] = {
0x1104, 0x00CE, 0x0000, 0x1000, 0x004C,
0x0000, 0x1200, 0x00C8, 0x0000, 0x1208,
0x004C, 0x0000, 0x0F08, 0x0000, 0xFFFF,
0x013E, 0x0004, 0x0008, 0x0A3E, 0x0000,
0x0000, 0x1104, 0x00CE, 0x0000, 0x1000,
0x004C, 0x0000, 0x1108, 0x00C4, 0x0000,
0x1200, 0x004C, 0x0000, 0x0A3E, 0x0000,
0x0000, 0x0F18, 0x0002, 0x0100, 0x0F28,
0x0000, 0x0000, 0x0F38, 0x0000, 0x0000,
0x503F, 0x6D80, 0x0000, 0x7DDF, 0x8000,
0x0000, 0x0A3E, 0x0000, 0x0000, 0x06BE,
0x0002, 0x011C, 0x013E, 0x0000, 0x2B04,
0x0600, 0x0002, 0x013C, 0x704F, 0x8580,
0x0000, 0x0F28, 0x0000, 0x0000, 0x0F38,
0x0000, 0x0000, 0x0F08, 0x0002, 0x0101,
0x013E, 0x0000, 0x1880, 0x0F09, 0x0008,
0x0000, 0x013E, 0x0000, 0x1889, 0x013E,
0x0002, 0xABB0, 0x0620, 0x0002, 0x012B,
0x708F, 0x8C00, 0x0000, 0x0FDC, 0x0000,
0x0000, 0x0FDD, 0x0000, 0x0001, 0x503F,
0xED80, 0x0000, 0x0F08, 0x0002, 0x0101,
0x013E, 0x0000, 0x1884, 0x708F, 0x8C00,
0x0000, 0x0F09, 0x0000, 0x0001, 0x503F,
0x6D80, 0x0000, 0x7DDF, 0x8580, 0x0000,
0x013E, 0x0004, 0x0BB9, 0x0600, 0x0002,
0x0136, 0x0FDD, 0x0000, 0x0000, 0x503F,
0xED80, 0x0000, 0x0A3E, 0x0000, 0x0000,
0x06BE, 0x0002, 0x011C, 0x013E, 0x0000,
0x2B04, 0x0600, 0x0002, 0x0157, 0x0F28,
0x0000, 0x0000, 0x0F38, 0x0000, 0x0000,
0x0F08, 0x0002, 0x0101, 0x013E, 0x0000,
0x1884, 0x0F09, 0x0000, 0x0001, 0x708F,
0x8C00, 0x0000, 0x503F, 0x6D80, 0x0000,
0x7DDF, 0x8580, 0x0000, 0x013E, 0x0004,
0x0BB9, 0x0600, 0x0002, 0x0146, 0x0FDD,
0x0000, 0x0000, 0x503F, 0xED80, 0x0000,
0x704F, 0x8580, 0x0000, 0x0F08, 0x0002,
0x0101, 0x013E, 0x0000, 0x1880, 0x0F09,
0x0008, 0x0000, 0x013E, 0x0000, 0x1889,
0x013E, 0x0002, 0xABB0, 0x0620, 0x0002,
0x0150, 0x708F, 0x8C00, 0x0000, 0x0FDC,
0x0000, 0x0000, 0x0FDD, 0x0000, 0x0001,
0x503F, 0xED80, 0x0000, 0x0A3E, 0x0000,
0x0000, 0x06BE, 0x0002, 0x011C, 0x700F,
0x8600, 0x0000, 0x0F09, 0x0000, 0x0001,
0x0F04, 0x0000, 0x0004, 0x013E, 0x0004,
0x0804, 0x0620, 0x0002, 0x0170, 0x0F04,
0x0000, 0x0001, 0x013E, 0x0000, 0x2804,
0x0600, 0x0002, 0x0163, 0x013E, 0x0000,
0x24C9, 0x06BE, 0x0002, 0x013D, 0x013E,
0x0002, 0x9CC0, 0x70CF, 0x8200, 0x0000,
0x06BE, 0x0002, 0x0122, 0x0F18, 0x0002,
0x0106, 0x0F28, 0x0000, 0x0000, 0x0F09,
0x0000, 0x0001, 0x503F, 0x6D80, 0x0000,
0x7DDF, 0x8200, 0x0000, 0x013E, 0x0004,
0x0449, 0x0600, 0x0002, 0x0169, 0x0FDD,
0x0000, 0x0000, 0x503F, 0xED80, 0x0000,
0x063E, 0x0002, 0x017F, 0x0F04, 0x0000,
0x0003, 0x06BE, 0x0002, 0x013D, 0x0F28,
0x0000, 0x0000, 0x0F38, 0x0000, 0x0000,
0x0F18, 0x000A, 0x0106, 0x0FDC, 0x0000,
0x0000, 0x0FDD, 0x0000, 0x0001, 0x503F,
0xED80, 0x0000, 0x0F18, 0x0012, 0x0106,
0x503F, 0xED80, 0x0000, 0x0F18, 0x001A,
0x0106, 0x503F, 0xED80, 0x0000, 0x0F18,
0x0002, 0x0106, 0x0FDD, 0x0000, 0x0000,
0x503F, 0xED80, 0x0000, 0x0A3E, 0x0000,
0x0000, 0x0F7D, 0x0000, 0x0000, 0x0F04,
0x0000, 0x5000, 0x0F08, 0x0000, 0x0000,
0x06BE, 0x0002, 0x0117, 0x0F18, 0x0002,
0x0101, 0x0F28, 0x0000, 0x0000, 0x0FDC,
0x0000, 0x0000, 0x0FDD, 0x0000, 0x0000,
0x503F, 0xED80, 0x0000, 0x0F18, 0x0002,
0x0100, 0x0FDC, 0x0000, 0x0000, 0x0FDD,
0x0000, 0x0004, 0x503F, 0xED80, 0x0000,
0x0F04, 0x0000, 0x5000, 0x06BE, 0x0002,
0x0110, 0x700F, 0x8500, 0x0000, 0x0F0B,
0x0000, 0x0001, 0x013E, 0x0004, 0x000B,
0x0620, 0x0002, 0x0197, 0x0F0B, 0x0000,
0x0002, 0x013E, 0x0004, 0x00AB, 0x0620,
0x0002, 0x01DA, 0x063E, 0x0002, 0x018D,
0x0F04, 0x0000, 0x5001, 0x06BE, 0x0002,
0x0110, 0x700F, 0x8480, 0x0000, 0x0F04,
0x0000, 0x5002, 0x06BE, 0x0002, 0x0110,
0x0F0C, 0x0000, 0x0010, 0x013E, 0x0020,
0x099C, 0x013E, 0x0004, 0x1990, 0x709F,
0x8C00, 0x0000, 0x0F28, 0x0000, 0x0001,
0x0F38, 0x0000, 0x0000, 0x0F04, 0x0000,
0x5003, 0x06BE, 0x0002, 0x0110, 0x700F,
0x8380, 0x0000, 0x0F04, 0x0000, 0x5004,
0x06BE, 0x0002, 0x0110, 0x700F, 0x8480,
0x0000, 0x013E, 0x0002, 0x9440, 0x06BE,
0x0002, 0x0110, 0x0F0C, 0x0000, 0x0010,
0x013E, 0x0020, 0x099C, 0x013E, 0x0004,
0x1990, 0x709F, 0xEE80, 0x0000, 0x013E,
0x0002, 0x9440, 0x06BE, 0x0002, 0x0110,
0x700F, 0xEE00, 0x0000, 0x0F0C, 0x0000,
0x0004, 0x013E, 0x0004, 0x0CAC, 0x0620,
0x0002, 0x01B6, 0x503F, 0xED80, 0x0000,
0x063E, 0x0002, 0x01D3, 0x718F, 0x8580,
0x0000, 0x0F0C, 0x0000, 0x0010, 0x013E,
0x0004, 0x0CAC, 0x0600, 0x0002, 0x01BE,
0x0F0C, 0x0008, 0x0000, 0x013E, 0x0000,
0x19BC, 0x709F, 0x8C00, 0x0000, 0x503F,
0xED80, 0x0000, 0x0F0C, 0x0000, 0x0020,
0x013E, 0x0004, 0x0CAC, 0x0600, 0x0002,
0x01C5, 0x0F0C, 0x0010, 0x0000, 0x013E,
0x0000, 0x19BC, 0x709F, 0x8C00, 0x0000,
0x503F, 0xED80, 0x0000, 0x0F0C, 0x0000,
0x0040, 0x013E, 0x0004, 0x0CAC, 0x0600,
0x0002, 0x01CC, 0x0F0C, 0x0018, 0x0000,
0x013E, 0x0000, 0x19BC, 0x709F, 0x8C00,
0x0000, 0x503F, 0xED80, 0x0000, 0x0F0C,
0x0000, 0x0080, 0x013E, 0x0004, 0x0CAC,
0x0600, 0x0002, 0x01D1, 0x70BF, 0x8C00,
0x0000, 0x503F, 0xED80, 0x0000, 0x013E,
0x0002, 0x9BB0, 0x70BF, 0x8C00, 0x0000,
0x013E, 0x0002, 0x9440, 0x013E, 0x0002,
0xA770, 0x0620, 0x0002, 0x01A6, 0x0F04,
0x0000, 0x5000, 0x0F08, 0x0000, 0x0000,
0x06BE, 0x0002, 0x0117, 0x063E, 0x0002,
0x018D, 0x0FDC, 0x0000, 0x0180, 0x0FDD,
0x063E, 0x0002, 0x0F18, 0x0002, 0x0007,
0x503F, 0xED80, 0x0000, 0x0F28, 0x0000,
0x0000, 0x0FDC, 0x0000, 0x0000, 0x0FDD,
0x0000, 0x0001, 0x0F0C, 0x0000, 0x0008,
0x013E, 0x0004, 0x0CAC, 0x0620, 0x0002,
0x01E6, 0x0F0C, 0x0000, 0x00F0, 0x013E,
0x0004, 0x1AAC, 0x0F0C, 0x0000, 0x0010,
0x013E, 0x0004, 0x0CAC, 0x0600, 0x0002,
0x01EB, 0x0F18, 0x000A, 0x0101, 0x503F,
0xED80, 0x0000, 0x0F0C, 0x0000, 0x0020,
0x013E, 0x0004, 0x0CAC, 0x0600, 0x0002,
0x01F0, 0x0F18, 0x0012, 0x0101, 0x503F,
0xED80, 0x0000, 0x0F0C, 0x0000, 0x0040,
0x013E, 0x0004, 0x0CAC, 0x0600, 0x0002,
0x01F5, 0x0F18, 0x001A, 0x0101, 0x503F,
0xED80, 0x0000, 0x0F0C, 0x0000, 0x0080,
0x013E, 0x0004, 0x0CAC, 0x0600, 0x0002,
0x0180, 0x0F18, 0x0002, 0x0101, 0x0FDC,
0x0000, 0x0000, 0x0FDD, 0x0000, 0x0001,
0x503F, 0xED80, 0x0000, 0x063E, 0x0002,
0x0004, 0x0A3E, 0x0000, 0x0000
};

/** The data of ``arssimp.hex'' in ER2-readable form. */
static int arssimp[ARSSIMP_LENGTH] = {
0x1104, 0x00CE, 0x0000, 0x1000, 0x004C,
0x0000, 0x1200, 0x00C8, 0x0000, 0x1208,
0x004C, 0x0000, 0x0F08, 0x0000, 0xFFFF,
0x013E, 0x0004, 0x0008, 0x0A3E, 0x0000,
0x0000, 0x1104, 0x00CE, 0x0000, 0x1000,
0x004C, 0x0000, 0x1108, 0x00C4, 0x0000,
0x1200, 0x004C, 0x0000, 0x0A3E, 0x0000,
0x0000, 0x0F18, 0x0002, 0x0100, 0x0F28,
0x0000, 0x0000, 0x0F38, 0x0000, 0x0000,
0x503F, 0x6D80, 0x0000, 0x7DDF, 0x8000,
0x0000, 0x0A3E, 0x0000, 0x0000, 0x06BE,
0x0002, 0x011C, 0x013E, 0x0000, 0x2B04,
0x0600, 0x0002, 0x013C, 0x704F, 0x8580,
0x0000, 0x0F28, 0x0000, 0x0000, 0x0F38,
0x0000, 0x0000, 0x0F08, 0x0002, 0x0101,
0x013E, 0x0000, 0x1880, 0x0F09, 0x0008,
0x0000, 0x013E, 0x0000, 0x1889, 0x013E,
0x0002, 0xABB0, 0x0620, 0x0002, 0x012B,
0x708F, 0x8C00, 0x0000, 0x0FDC, 0x0000,
0x0000, 0x0FDD, 0x0000, 0x0001, 0x503F,
0xED80, 0x0000, 0x0F08, 0x0002, 0x0101,
0x013E, 0x0000, 0x1884, 0x708F, 0x8C00,
0x0000, 0x0F09, 0x0000, 0x0001, 0x503F,
0x6D80, 0x0000, 0x7DDF, 0x8580, 0x0000,
0x013E, 0x0004, 0x0BB9, 0x0600, 0x0002,
0x0136, 0x0FDD, 0x0000, 0x0000, 0x503F,
0xED80, 0x0000, 0x0A3E, 0x0000, 0x0000,
0x06BE, 0x0002, 0x011C, 0x013E, 0x0000,
0x2B04, 0x0600, 0x0002, 0x0157, 0x0F28,
0x0000, 0x0000, 0x0F38, 0x0000, 0x0000,
0x0F08, 0x0002, 0x0101, 0x013E, 0x0000,
0x1884, 0x0F09, 0x0000, 0x0001, 0x708F,
0x8C00, 0x0000, 0x503F, 0x6D80, 0x0000,
0x7DDF, 0x8580, 0x0000, 0x013E, 0x0004,
0x0BB9, 0x0600, 0x0002, 0x0146, 0x0FDD,
0x0000, 0x0000, 0x503F, 0xED80, 0x0000,
0x704F, 0x8580, 0x0000, 0x0F08, 0x0002,
0x0101, 0x013E, 0x0000, 0x1880, 0x0F09,
0x0008, 0x0000, 0x013E, 0x0000, 0x1889,
0x013E, 0x0002, 0xABB0, 0x0620, 0x0002,
0x0150, 0x708F, 0x8C00, 0x0000, 0x0FDC,
0x0000, 0x0000, 0x0FDD, 0x0000, 0x0001,
0x503F, 0xED80, 0x0000, 0x0A3E, 0x0000,
0x0000, 0x06BE, 0x0002, 0x011C, 0x700F,
0x8600, 0x0000, 0x0F09, 0x0000, 0x0001,
0x0F04, 0x0000, 0x0004, 0x013E, 0x0004,
0x0804, 0x0620, 0x0002, 0x0170, 0x0F04,
0x0000, 0x0001, 0x013E, 0x0000, 0x2804,
0x0600, 0x0002, 0x0163, 0x013E, 0x0000,
0x24C9, 0x06BE, 0x0002, 0x013D, 0x013E,
0x0002, 0x9CC0, 0x70CF, 0x8200, 0x0000,
0x06BE, 0x0002, 0x0122, 0x0F18, 0x0002,
0x0106, 0x0F28, 0x0000, 0x0000, 0x0F09,
0x0000, 0x0001, 0x503F, 0x6D80, 0x0000,
0x7DDF, 0x8200, 0x0000, 0x013E, 0x0004,
0x0449, 0x0600, 0x0002, 0x0169, 0x0FDD,
0x0000, 0x0000, 0x503F, 0xED80, 0x0000,
0x063E, 0x0002, 0x017F, 0x0F04, 0x0000,
0x0003, 0x06BE, 0x0002, 0x013D, 0x0F28,
0x0000, 0x0000, 0x0F38, 0x0000, 0x0000,
0x0F18, 0x000A, 0x0106, 0x0FDC, 0x0000,
0x0000, 0x0FDD, 0x0000, 0x0001, 0x503F,
0xED80, 0x0000, 0x0F18, 0x0012, 0x0106,
0x503F, 0xED80, 0x0000, 0x0F18, 0x001A,
0x0106, 0x503F, 0xED80, 0x0000, 0x0F18,
0x0002, 0x0106, 0x0FDD, 0x0000, 0x0000,
0x503F, 0xED80, 0x0000, 0x0A3E, 0x0000,
0x0000, 0x0F7D, 0x0000, 0x0000, 0x0F18,
0x0002, 0x0101, 0x0F28, 0x0000, 0x0000,
0x0FDC, 0x0000, 0x0000, 0x0FDD, 0x0000,
0x0000, 0x503F, 0xED80, 0x0000, 0x0F18,
0x0002, 0x0100, 0x1008, 0x0000, 0x0003,
0x023E, 0x0010, 0xC888, 0x0FDC, 0x0000,
0x0000, 0x708F, 0xEE80, 0x0000, 0x503F,
0xED80, 0x0000, 0x0F18, 0x0002, 0x0101,
0x0F28, 0x0000, 0x0000, 0x0F38, 0x0000,
0x0000, 0x0F09, 0x0000, 0x0001, 0x503F,
0x6D80, 0x0000, 0x7DDF, 0x8400, 0x0000,
0x013E, 0x0004, 0x0889, 0x0600, 0x0002,
0x0190, 0x0FDC, 0x0000, 0x0180, 0x0FDD,
0x063E, 0x0002, 0x0F18, 0x0002, 0x0007,
0x503F, 0xED80, 0x0000, 0x063E, 0x0002,
0x0004, 0x0A3E, 0x0000, 0x0000
};
```

```
/** The data of ``bootars.hex'' in ER2-readable form. */
static int bootars[BOOTARS_LENGTH] = {
0x0F28, 0x0000, 0x0001, 0x0F38, 0x0000,
0x0000, 0x0F18, 0x0012, 0x0004, 0x0FDD,
0x0000, 0x0000, 0x0FDC, 0x0000, 0x0000,
0x503F, 0xED80, 0x0000, 0x503F, 0xED80,
0x0000, 0x503F, 0xED80, 0x0000, 0x0FDD,
0x063E, 0x0002, 0x0FDC, 0x0000, 0x0180,
0x503F, 0xED80, 0x0000, 0x0F18, 0x0012,
0x0102, 0x0FDD, 0x0000, 0x0000, 0x0FDC,
0x0000, 0x0000, 0x503F, 0xED80, 0x0000,
0x503F, 0xED80, 0x0000, 0x503F, 0xED80,
0x0000, 0x503F, 0xED80, 0x0000, 0x503F,
0xED80, 0x0000, 0x0FDD, 0x0002, 0x0180,
0x0FDC, 0x0000, 0xFFFF, 0x0F18, 0x0012,
0x0000, 0x503F, 0xED80, 0x0000, 0x0F18,
0x001A, 0x0004, 0x0FDD, 0x0000, 0x0000,
0x0FDC, 0x0000, 0x0000, 0x503F, 0xED80,
0x0000, 0x503F, 0xED80, 0x0000, 0x503F,
0xED80, 0x0000, 0x0FDD, 0x063E, 0x0002,
0x0FDC, 0x0000, 0x0180, 0x503F, 0xED80,
0x0000, 0x0F18, 0x001A, 0x0102, 0x0FDD,
0x0000, 0x0000, 0x0FDC, 0x0000, 0x0000,
0x503F, 0xED80, 0x0000, 0x503F, 0xED80,
0x0000, 0x503F, 0xED80, 0x0000, 0x503F,
0xED80, 0x0000, 0x503F, 0xED80, 0x0000,
0x0FDD, 0x0002, 0x0180, 0x0FDC, 0x0000,
0xFFFF, 0x0F18, 0x001A, 0x0000, 0x503F,
0xED80, 0x0000, 0x0F18, 0x0022, 0x0004,
0x0FDD, 0x0000, 0x0000, 0x0FDC, 0x0000,
0x0000, 0x503F, 0xED80, 0x0000, 0x503F,
0xED80, 0x0000, 0x503F, 0xED80, 0x0000,
0x0FDD, 0x063E, 0x0002, 0x0FDC, 0x0000,
0x0180, 0x503F, 0xED80, 0x0000, 0x0F18,
0x0022, 0x0102, 0x0FDD, 0x0000, 0x0000,
0x0FDC, 0x0000, 0x0000, 0x503F, 0xED80,
0x0000, 0x503F, 0xED80, 0x0000, 0x503F,
0xED80, 0x0000, 0x503F, 0xED80, 0x0000,
0x503F, 0xED80, 0x0000, 0x0FDD, 0x0002,
0x0180, 0x0FDC, 0x0000, 0xFFFF, 0x0F18,
0x0022, 0x0000, 0x503F, 0xED80, 0x0000,
0x0F18, 0x0002, 0x0004, 0x0FDD, 0x0000,
0x0000, 0x0FDC, 0x0000, 0x0000, 0x503F,
0xED80, 0x0000, 0x503F, 0xED80, 0x0000,
0x503F, 0xED80, 0x0000, 0x0FDD, 0x063E,
0x0002, 0x0FDC, 0x0000, 0x0180, 0x503F,
0xED80, 0x0000, 0x0F18, 0x0002, 0x0102,
0x0FDD, 0x0000, 0x0000, 0x0FDC, 0x0000,
0x0000, 0x503F, 0xED80, 0x0000, 0x503F,
0xED80, 0x0000, 0x503F, 0xED80, 0x0000,
  0x503F, 0xED80, 0x0000, 0x503F, 0xED80,
  0x0000, 0x063E, 0x0002, 0x0180, 0x0A3E,
  0x0000, 0x0000
  };
```

Defines:
  arsmain, used in chunks 47b and 58.
  arssimp, used in chunks 47b and 58.
  bootars, used in chunk 58.
Uses ARSMAIN_LENGTH 58e, ARSSIMP_LENGTH 58e, BOOTARS_LENGTH 58e, and data 26a 47d.

65a ⟨*Boot with ARS* 65a⟩≡ (57a)

```
  /** Boots the SHARCs, detects the number of
  * SHARC clusters and installs the ARS.
  * All 2181 modules that have a SHARC cluster attached
  * are combined to a new group ALL_CLUSTERS.
  */
  void boot_ars()
  {
    int cnt;

    boot_sharcs();
    detect_sharcs();

    for (cnt = 0; cnt < get_number_of_clusters(); cnt++)
      join_group(get_physical_cluster_address(cnt), ALL_CLUSTERS);

    install_ars();
  }
```

Defines:
  boot_ars, used in chunks 44 and 65c.
Uses ALL_CLUSTERS 65b, boot_sharcs 3b, cnt 26a 47a, detect_sharcs 12b,
  get_number_of_clusters 13c, get_physical_cluster_address 13e, and install_ars 57b.

65b ⟨*HF: Defines* 2c⟩+≡ (2a) ◁3c 66a▷

```

  /** Group of all ADSP-2181 with an attached SHARC cluster */
  #define ALL_CLUSTERS            62
```

Defines:
  ALL_CLUSTERS, used in chunks 58, 65a, and 67b.

65c ⟨*HF: Function prototypes* 11b⟩+≡ (2a) ◁21a 70a▷

```
  extern void boot_ars();
```

Uses boot_ars 65a.

7 Loading data via ARS

65d ⟨*Loading Programs* 16a⟩+≡ (3a) ◁57a 70b▷

⟨*Broadcast memory piece via ARS* 66b⟩
⟨*Broadcast memory via ARS* 68d⟩

66a ⟨*HF: Defines* 2c⟩+≡ (2a) ◁65b

```
/** ID value for SHARC #1 */
#define SHARC1                  0x1
/** ID value for SHARC #2 */
#define SHARC2                  0x2
/** ID value for SHARC #3 */
#define SHARC3                  0x4
/** ID value for SHARC #4 */
#define SHARC4                  0x8
/** ID value for all SHARCs together, i.e. a broadcast */
#define ALL_SHARCS              0xF
```

Defines:
ALL_SHARCS, never used.
SHARC1, never used.
SHARC2, never used.
SHARC3, never used.
SHARC4, never used.

66b ⟨*Broadcast memory piece via ARS* 66b⟩≡ (65d)

```
/** Loads a piece of program code from the given array \a program
* with maximum size ARS_DATA_MAX to a single SHARC or a group
* of SHARCs on a cluster.
* Size here refers to the number of 48-bit program words.
* @see ars_broadcast_memory.
* @param groupnr The group number
* @param netadress Physical address of the 2181 node if groupnr is
* SINGLE, where the SHARC cluster is attached
* @param netadress Physical address of the 2181 node, where the
* SHARC cluster is attached
* @param ID Index value for addressing the single SHARCs (0x1=#1, 0x2=#2,
* 0x4=#3, 0x8=#4)
* @param address_msw Most significant word of the
* 32-bit address
* @param address_lsw Least significant word of the
* 32-bit address
* @param program Array with program words (MSW, BMW, LSW)
* @param piece_length Number of program words to dump
*/
static void ars_broadcast_memory_piece(int groupnr,
                                       int netaddress,
                                       int ID,
                                       int address_msw,
                                       int address_lsw,
                                       int *program,
                                       int piece_length)
```

```
  {
    int data[4];
    int cnt;

    if (groupnr == SINGLE)
    {
      ⟨abmp: wait until SHARC is ready 67a⟩
    }
    else
    {
      ⟨abmp: wait until most distant SHARC is ready 67b⟩
    }
    ⟨abmp: initialize data 68a⟩

    ⟨abmp: read program words from array 68b⟩

    ⟨abmp: set start signal for reading 68c⟩
    if (groupnr == SINGLE)
    {
      ⟨abmp: wait until SHARC is ready 67a⟩
    }
    else
    {
      ⟨abmp: wait until most distant SHARC is ready 67b⟩
    }

  }
```

Defines:
  ars_broadcast_memory_piece, used in chunks 68d, 69d, and 71c.
Uses ars_broadcast_memory 68d, cnt 26a 47a, and data 26a 47d.

67a ⟨*abmp: wait until SHARC is ready* 67a⟩≡ (66b)

```
  data[0] = 0x800;
  while (data[0] != 0x0)
    request_memory(netaddress, 0x1000, dat, data, 1);
```

Uses data 26a 47d.

If the group ALL_CLUSTERS of processors is defined, the cluster that was detected last during bootup is checked. The assumption is that it needs the longest time to finish because of its maximum distance to the root processor.

67b ⟨*abmp: wait until most distant SHARC is ready* 67b⟩≡ (66b)

```
  if (groupnr == ALL_CLUSTERS)
  {
    cnt = get_physical_cluster_address(get_number_of_clusters()-1);
    data[0] = 0x800;
    while (data[0] != 0x0)
      request_memory(cnt, 0x1000, dat, data, 1);
```

```
}
```

Uses **ALL_CLUSTERS** 65b, **cnt** 26a 47a, **data** 26a 47d, **get_number_of_clusters** 13c, and **get_physical_cluster_address** 13e.

68a ⟨*abmp: initialize data* 68a⟩≡ (66b)

```
data[0] = address_msw & 0xFFFF;
data[1] = address_lsw & 0xFFFF;
data[2] = piece_length & 0xFFFF;

broadcast_memory(groupnr, netaddress, 0x1001, dat, data, 3);
```

Uses **data** 26a 47d.

68b ⟨*abmp: read program words from array* 68b⟩≡ (66b)

```
broadcast_memory(groupnr, netaddress, 0x1004, dat, program, 3*piece_length);
```

68c ⟨*abmp: set start signal for reading* 68c⟩≡ (66b)

```
data[0] = 0x5 | (ID << 4);
broadcast_memory(groupnr, netaddress, 0x1000, dat, data, 1);
```

Uses **data** 26a 47d.

68d ⟨*Broadcast memory via ARS* 68d⟩≡ (65d)

```
/** Loads the program code from the given array \a program
* to a single SHARC or a group of SHARCS on a cluster.
* The address is interpreted relative to SHARC 4.
* @see ars_broadcast_memory_piece.
* @param groupnr The group number
* @param netadress Physical address of the 2181 node if groupnr
* is SINGLE, where the SHARC cluster is attached
* @param ID Index value for addressing the single SHARCs (0x1=#1, 0x2=#2,
* 0x4=#3, 0x8=#4)
* @param address_msw Most significant word (Upper 16 bit) of the
* 32-bit address
* @param address_lsw Least significant word (Lower 16 bit) of the
* 32-bit address
* @param program Array with program words (MSW, BMW, LSW)
* @param length Number of 48-bit program words to dump
*/
void ars_broadcast_memory(int groupnr,
                          int netaddress,
                          int ID,
                          int address_msw,
                          int address_lsw,
                          int *program,
                          int length)
```

```
{
  int last_addr;
  int i = 0;

  while (i < (length/ARS_MAX_DATA))
  {
    ⟨abm: load piece of data 69b⟩
    ⟨abm: update addresses 69c⟩
    i++;
  }

  ⟨abm: load last piece of data 69d⟩
}
```

Defines:
  `ars_broadcast_memory`, used in chunks 66b and 70a.
Uses `ars_broadcast_memory_piece` 66b 69b 71a and `ARS_MAX_DATA` 69a.

69a ⟨*Defines* 6a⟩+≡ (1a) ◁58e

```
/** Maximum number of 48-bit program words that can be loaded to a
* SHARC via ARS at once */
#define ARS_MAX_DATA    0x800
```

Defines:
  `ARS_MAX_DATA`, used in chunks 68–71.

69b ⟨*abm: load piece of data* 69b⟩≡ (68d)

```
ars_broadcast_memory_piece(groupnr, netaddress, ID, address_msw,
                           address_lsw, program, ARS_MAX_DATA);
```

Defines:
  `ars_broadcast_memory_piece`, used in chunks 68d, 69d, and 71c.
Uses `ARS_MAX_DATA` 69a.

69c ⟨*abm: update addresses* 69c⟩≡ (68d)

```
last_addr = address_lsw;
address_lsw = (address_lsw + ARS_MAX_DATA) & 0xFFFF;
if (address_lsw < last_addr)
  address_msw++;
program += 3*ARS_MAX_DATA;
```

Uses `ARS_MAX_DATA` 69a.

69d ⟨*abm: load last piece of data* 69d⟩≡ (68d)

```
if ((i * ARS_MAX_DATA) < length)
  ars_broadcast_memory_piece(groupnr, netaddress, ID, address_msw,
                             address_lsw, program,
                             length-i*ARS_MAX_DATA);
```

Uses `ars_broadcast_memory_piece` 66b 69b 71a and `ARS_MAX_DATA` 69a.

70a ⟨*HF: Function prototypes* 11b⟩+≡ (2a) ◁65c 74c▷

```
extern void ars_broadcast_memory(int, int, int, int, int, int *, int);
```

Uses `ars_broadcast_memory` 68d.

8 Loading executables via ARS

70b ⟨*Loading Programs* 16a⟩+≡ (3a) ◁65d

⟨*Dump section via ARS* 70d⟩
⟨*Load executable via ARS* 72b⟩

70c ⟨*Global variables* 6b⟩+≡ (1a) ◁59

```
/** Array used for loading executables to a SHARC cluster */
static int transfer[ARS_MAX_DATA*3];
```

Defines:
`transfer`, used in chunk 71.
Uses `ARS_MAX_DATA` 69a.

70d ⟨*Dump section via ARS* 70d⟩≡ (70b)

```
/** Loads the program code from the given file \a fObject
* to a single SHARC or a group of SHARCs on a cluster.
* @param groupnr The group number
* @param netadress Physical address of the 2181 node if the groupnr
* is SINGLE, where the SHARC cluster is attached
* @param ID Index value for addressing the single SHARCs (0x1=#1, 0x2=#2,
* 0x4=#3, 0x8=#4)
* @param address_msw Most significant word (Upper 16 bit) of the
* 32-bit address
* @param address_lsw Least significant word (Lower 16 bit) of the
* 32-bit address
* @param fObject File with program code
* @param length Number of 48-bit program words to load
*/
void ars_dump_section(int groupnr,
                      int netaddress,
                      int ID,
                      int address_msw,
                      int address_lsw,
                            FILE *fObject,
                            int length)
{
  int last_addr;
  int i = 0;
  int cnt;
  char cUpper, cLower;

  while (i < (length/ARS_MAX_DATA))
  {
    ⟨ads: load piece of data 71a⟩
    ⟨ads: update addresses 71b⟩
    i++;
  }

  ⟨ads: load last piece of data 71c⟩
}
```

Defines:
  ars_dump_section, used in chunk 74a.
Uses ARS_MAX_DATA 69a, cnt 26a 47a, and file 2b.

71a ⟨*ads: load piece of data* 71a⟩≡ (70d)

```
for (cnt = 0; cnt < (ARS_MAX_DATA*3); cnt++)
{
  fread(&cUpper, 1, 1, fObject);
  fread(&cLower, 1, 1, fObject);
  transfer[cnt] = ((cUpper & 0xFF) << 8) + (cLower & 0xFF);
}

ars_broadcast_memory_piece(groupnr, netaddress, ID, address_msw,
                           address_lsw, transfer, ARS_MAX_DATA);
```

Defines:
  ars_broadcast_memory_piece, used in chunks 68d, 69d, and 71c.
Uses ARS_MAX_DATA 69a, cnt 26a 47a, and transfer 70c.

71b ⟨*ads: update addresses* 71b⟩≡ (70d)

```
last_addr = address_lsw;
address_lsw = (address_lsw + ARS_MAX_DATA) & 0xFFFF;
if (address_lsw < last_addr)
  address_msw++;
```

Uses ARS_MAX_DATA 69a.

71c ⟨*ads: load last piece of data* 71c⟩≡ (70d)

```
  if ((i * ARS_MAX_DATA) < length)
  {
    for (cnt = 0; cnt < ((length - i*ARS_MAX_DATA)*3); cnt++)
    {
      fread(&cUpper, 1, 1, fObject);
      fread(&cLower, 1, 1, fObject);
      transfer[cnt] = ((cUpper & 0xFF) << 8) + (cLower & 0xFF);
    }

    ars_broadcast_memory_piece(groupnr, netaddress, ID, address_msw,
                               address_lsw, transfer, length-i*ARS_MAX_DATA);
  }
```

Uses `ars_broadcast_memory_piece` 66b 69b 71a, `ARS_MAX_DATA` 69a, `cnt` 26a 47a, and `transfer` 70c.

72a ⟨*Include files* 1d⟩+≡ (1a) ◁1d

```
  #include "../config.h"
  #include "a_out.h"
  #include "coff_io.h"
```

72b ⟨*Load executable via ARS* 72b⟩≡ (70b)

```
  /** Reads the single sections from the given executable
  * \a fName and loads them to a single SHARC or a group of
  * SHARCs on a cluster.
  * @param groupnr The group number
  * @param netadress Physical address of the 2181 node if groupnr
  * is SINGLE, where the SHARC cluster is attached
  * @param ID Index value for addressing the single SHARCs (0x1=#1, 0x2=#2,
  * 0x4=#3, 0x8=#4)
  * @param fName Filename of the executable
  */
  void ars_load_program(int groupnr, int netaddress, int ID, char *fName)
  {
    FILE *fExe;
    FILHDR file_header;
    SCNHDR section_header;
    int scnt;

    ⟨alp: open file 73a⟩
    ⟨alp: read header 73b⟩
    ⟨alp: load sections 73c⟩
    ⟨alp: close file 74b⟩

  }
```

Defines:
ars_load_program, used in chunk 74c.

73a ⟨*alp: open file* 73a⟩≡ (72b)

```
fExe = fopen(fName, "r");
if (fExe == NULL)
{
  fprintf(stderr, "ER2SH: Could not open executable %s!\n", fName);
  return;
}
```

73b ⟨*alp: read header* 73b⟩≡ (72b)

```

if( !read_file_header( &file_header, fExe) )
{
  fprintf(stderr, "ER2SH: Could not read file header of executable %s!\n", fName);
  return;
}
if( M_21000 != file_header.f_magic)
{
  fprintf(stderr, "ER2SH: Bad magic number in executable %s! (Should be 0x%2)\n", fName,
  return;
}
```

Uses `file` 2b.

73c ⟨*alp: load sections* 73c⟩≡ (72b)

```
for (scnt = 0; scnt < file_header.f_nscns; scnt++)
{
  ⟨alpls: get section header 73d⟩
  ⟨alpls: load section if not empty 74a⟩
}
```

73d ⟨*alpls: get section header* 73d⟩≡ (73c)

```
if( fseek(fExe, (long)(FILHSZ +
                        file_header.f_opthdr +
                        ( scnt * SCNHSZ )), SEEK_SET))
{
  fprintf(stderr, "ER2SH: Could not seek to section header %d of executable %s!\n", scnt
  return;
}

if( !read_section_header( &section_header, fExe) )
{
  fprintf(stderr, "ER2SH: Could not read header of section %s in executable %s!\n",
                 section_header.s_name, fName);
  return;
}
```

74a ⟨*alpls: load section if not empty* 74a⟩≡ (73c)
```
  if (section_header.s_size > 0)
  {
    /* Seek to raw section data */
    if( fseek(fExe, section_header.s_scnptr, SEEK_SET) )
    {
      fprintf(stderr, "ER2SH: Could not seek to section %d of executable %s!\n", scnt, fNa
      return;
    }
    ars_dump_section(groupnr, netaddress, ID,
                     (int) ((section_header.s_paddr >> 16) & 0xFFFF),
                     (int) (section_header.s_paddr & 0xFFFF),
                     fExe, section_header.s_size/6);
  }
```
Uses **ars_dump_section** 70d and **data** 26a 47d.

74b ⟨*alp: close file* 74b⟩≡ (72b)
```
  fclose(fExe);
```

74c ⟨*HF: Function prototypes* 11b⟩+≡ (2a) ◁70a 75c▷
```
  extern void ars_load_program(int, int, int, char *);
```
Uses **ars_load_program** 72b.

9 Starting programs via ARS

74d ⟨*Executing Programs* 20b⟩+≡ (3a) ◁20b
  ⟨*Start cluster via ARS* 74e⟩

74e ⟨*Start cluster via ARS* 74e⟩≡ (74d)
```
  /** Starts the cluster on top of the 2181 \a netaddress or the group
  * \a groupnr via ARS.
  * @param groupnr The group number
  * @param netadress Physical address of the 2181 node, where the
  * SHARC cluster is attached
  * @param ID Index value for addressing the single SHARCs (0x1=#1, 0x2=#2,
  * 0x4=#3, 0x8=#4)
  */
  void ars_start_cluster(int groupnr, int netaddress, int ID)
  {
    int data[2];

    if (groupnr == SINGLE)
    {
      ⟨asc: wait until SHARC cluster is ready 75a⟩
    }
    ⟨asc: start cluster 75b⟩
  }
```

Defines:
  `ars_start_cluster`, used in chunk 75c.
Uses `data` 26a 47d.

75a ⟨*asc: wait until SHARC cluster is ready* 75a⟩≡ (74e)

```
  data[0] = 0x800;
  while (data[0] != 0x0)
    request_memory(netaddress, 0x1000, dat, data, 1);
```

Uses `data` 26a 47d.

75b ⟨*asc: start cluster* 75b⟩≡ (74e)

```
  data[0] = 0xA | (ID << 4);
  broadcast_memory(groupnr, netaddress, 0x1000, dat, data, 1);
```

Uses `data` 26a 47d.

75c ⟨*HF: Function prototypes* 11b⟩+≡ (2a) ◁74c

```
  extern void ars_start_cluster(int, int, int);
```

Uses `ars_start_cluster` 74e.

10 Building and installing the library

For compiling the library please use the prepared `Makefile` by issueing the command

```
make
```

After successful compilation, the library `liber2sh.a` and the header `er2sh.h` should be installed by

```
make install
```

assuming that you changed to *root* mode.

List of code chunks

This list was generated automatically by NOWEB. The numeral is that of the first definition of the chunk.

⟨*Write to SHARC* 14c⟩

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. *An ER2 Library in C. A Collection of Functions for Using and Configuring the ER2*, 2003. Internal report.

[3] Dirk Bächle. *A Linux Device Driver for the Parallel Port and the ISA Card Host Interface of the ER2*, 2003. Internal report.

[4] Georg-Friedrich Mayer-Lindenberg. *Message Passing im ER2 und Funktionen der Laufzeitkerne*, 1997. Internal report.

[5] Georg-Friedrich Mayer-Lindenberg. *Bootstrap, Programmdownload, Prozeßstart und Kommunikation mit den Sharcs im ER2*, 1999. Internal report.