Programming and Using a DSP-Based Parallel Computer under Linux

Studienarbeit von Dirk Bächle

Technische Universität Hamburg-Harburg
Technische Informatik VI (Verteilte Systeme)
Prof. Dr. Georg-Friedrich Mayer-Lindenberg

9. Dezember 2003

Contents

1 Introduction

While system configurations like COW (*Cluster of Workstations*) and *multi computer architectures* like *Beowulf* gain more and more ground in the area of parallel processing, the classic parallel computers still have their share. Consisting of many tightly-coupled processing nodes that work in concert, these highly parallel machines can solve large application problems and—if properly designed—are eligible to special purposes like digital signal processing or 2D/3D imaging.
The ER2 (fig. 1) is a parallel computer developed at the department 'Distributed Systems' of the Technical University Hamburg-Harburg. This MIMD machine is based on several types of *Digital Signal Processors* and has a very flexible structure. Its network is built on a layer of *2181 modules* (see 2.2). This *base layer* can be extended by attaching appropriate add-on boards like the *SHARC modules*, which are further described in 2.3.
One possible configuration of the ER2 uses the SHARC clusters for breaking down a complex task, on top of the *network of 2181 nodes.* The latter connect the fast serial links of the SHARCs via special crossbar switches, such that the exchange of data or synchronization messages is possible. These connections are preferably set up once and kept static during runtime. However, the dynamic reconfiguration of crossbar links is also possible and a major feature. It enriches the area of research for the ER2 with some interesting topics like *fault tolerance* and *reconfigurable networks.*
Unfortunately, due to its heterogeneous structure the software support for the ER2 is rather poor. So far, only the *Fifth* system (see [13, 14]) can be used to interact with the single processing elements. Via a normal PC—running the *Fifth* IDE—and a special interface called *host adapter*, data and code can be loaded to the 2181 processors and the SHARCs. A small *parallel runtime system* (PRS), developed in *Fifth*, provides a set of communication primitives, the so called *messages.* This communication is solely based on the processor the host adapter is attached to. The *root processor* can send messages to other nodes, but not vice versa. No form of arbitrary *message passing* is supported. Hence, the mentioned scenario with the SHARCs as *processing layer* and the 2181 as *connection layer* appears to be an acceptable alternative and is regarded as the ER2 standard configuration for the rest of this document.
Another slight drawback is the fact that *Fifth* is a stack-oriented language. Published algorithms however, are often presented in some form of pseudo-code similar to high-level infix languages like C, Pascal or even Matlab. So, the user would have to restructure the given algorithm instead of implementing it in a straightforward manner.

The aim of this work is to provide a small *Linux* programming environment that enables the user to write C programs for the single SHARC nodes on the ER2, easy loading and executing of the generated code included. For the general access of the parallel computer via the *host adapter*, a special device driver is developed. It supports both kinds of existing interfaces—parallel port and ISA bus—and establishes a doorway to the *Fifth* PRS. As already stated, the SHARC clusters partly depend on the 2181 modules regarding the configuration of the crossbar switches. Thus, additional support is provided for this *base layer*. A library written in C offers various functions, ranging from the detection of the network topology and read/write operations up to a complete system for automatically configuring and testing an arbitrary set of crossbar links. In the next step, an equivalent library for the SHARC modules is implemented. Accompanied by several functions for exchanging data with the 2181 processors, it can boot and detect the 2106x boards. It is complemented by some SHARC assembler routines that build a small runtime system for loading code and data, also enabling the restart of programs. For programming the SHARCs in C, the *G21k* utils are selected. They stem from an early version of the `gcc`—and relating programs—by Analog Devices, which were made publicly available on the Internet. The *G21k* utils consist of the C compiler `g21k` and several useful tools like a SHARC assembler and linker. Several patches are applied to them and an own archiver is developed from a copy of the linker sources. It is then used to create a small Standard C and Math library for the compiler `g21k`, based on the assembler sources from Analog Devices. The libraries offer a set of common functions like `asin` and `pow`, supporting floating point arithmetics for the data types `float` and `double`. The ease of programming that results from the achieved *status quo* is shown by creating an example application. An unidirectional ring of eight SHARC clusters computes the *Cholesky factorization* of a real matrix.
The structure of this text is as follows: section 2 introduces the reader to the *hardware*, i.e. the basic layout of the ER2. Then, sections 3–5 outline the development and usage of the necessary device driver and the supporting C libraries. How to compile and link code for the SHARCs is explained in section 6 where the *G21k binutils* and the C compiler `g21k` are described. Finally, the algorithm and the implementation of the application example are presented (sec. 7).
In the appendices, the reader finds an index of available functions for the G21k Standard C libraries and a CD-ROM which serves as a backup for all the data related to this work. Among source code, documentation and examples it also contains the internal reports [3, 16, 6, 5, 4, 7], where the latter three are *Literate Programming* documents. More information on this technique and the used tool NOWEB can be found in [5].

2 The parallel computer ER2

2.1 Basic structure

The ER2 is a parallel computer built in a modular fashion. On each of its 16 backplanes one discovers 16 connectors for little plug-in processor boards, which are designed around an ADSP-2181. If fully assembled and by the aid of the *SHARC modules* the ER2 makes it up to a total of 512 processors (256 ADSP-2181, 256 SHARC-2106x).

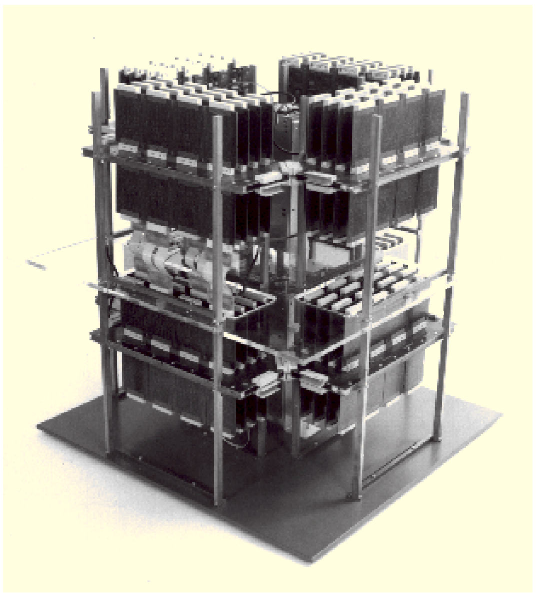


Figure 1: The parallel computer ER2 (Source: TI6, TUHH)

By inserting a processor board to the backplane it is directly connected to its four neighbours in the north, west, south and east (see also fig. 10, p. 19). Additionally, one can use the crossbar switches to establish links between dedicated nodes. This configuration can be done during runtime.
Thus, a large variety of graphs can be mapped to the network topology of the ER2. The DSPs offer a computing power of up to 12+16 GOPS, especially for digital signal processing applications.

2.2 ADSP-2181 modules

The *ADSP-2181 modules* (fig. 2) are built around the ADSP-2181 (see [1]). They also contain a *crossbar switch* (I-Cube IQ160), controlled by an FPGA.

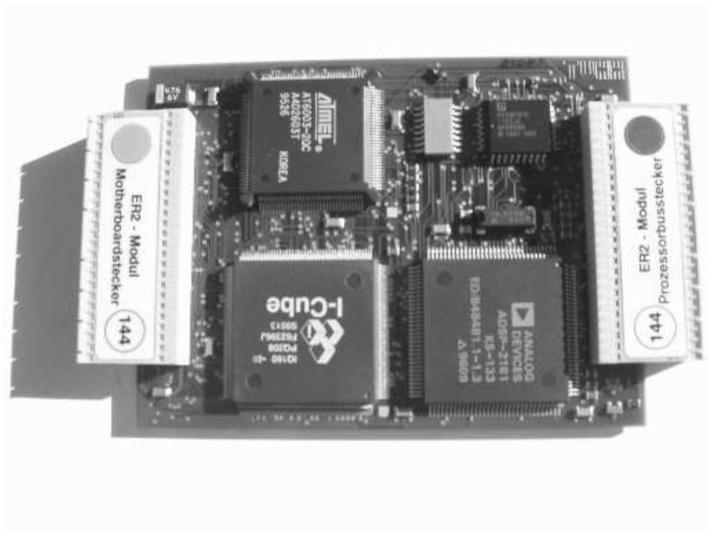


Figure 2: 2181 module

The ADSP-2181 is a 16-bit integer DSP with 16k words of internal memory, for data and program memory each. It is based on a Harvard architecture and offers separate data buses to *Program Memory* (PM) and *Data Memory* (DM). In connection with the internal *accumulator*, the processor is capable of performing up to two *data fetches*, one *multiply* and one *add* operation in a single cycle. Hence, building *scalar products*—i.e. matrix-vector or even matrix-matrix multiplications—is often its main task. This operation is broadly used in the area of *Digital Signal Processing*, e.g. for digital filters.
At boot time, the ADSP-2181 is fed from the Flash-EPROM with a small *parallel runtime system* (PRS) which was written in *Fifth*. It provides a basic layer of routines for reading/writing from/to the memory of the single DSPs.
One of the connectors shown in figure 2 fits into the backplane of the ER2. The other can be used to extend the hardware by attaching additional boards like the *SHARC modules*.

2.3 SHARC modules

The SHARC processors are built around a Harvard architecture, too. In contrast to the ADSP-2181 they provide more internal memory—1MBit for the 21061 and 2MBit for the 21062—and support 40-bit floating-point operations directly.
A very interesting feature of the ADSP-2106x series is the support for multiprocessing. Accessing a common address and data bus, each SHARC in an

embedded system gets a unique ID. This setup does not only include automatic bus arbitration. The SHARCs can also access all internal memories that are mapped to the common address space accordingly (see [2, 5-10]). Thus, data can be distributed very quickly among the single processors. Additionally, the 21062 offers six fast serial links that can be used for dedicated remote connections.
The SHARC clusters for the ER2 contain two ADSP-21062 and two ADSP-21061 each (see fig. 3).

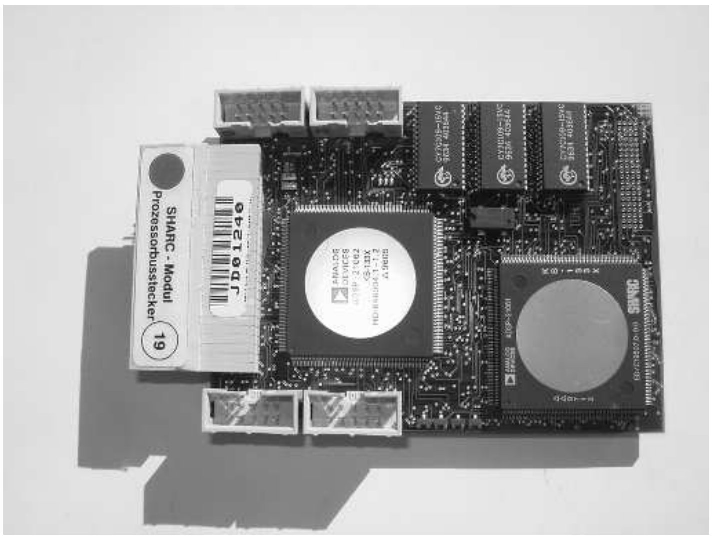


Figure 3: SHARC cluster

As is shown in fig. 4 they build a small multiprocessing system with an external memory bank of 128k words to share.
The serial links L0, L3, L4, L5 of the SHARC #1 and L3, L4 of #2 are directly connected to pins of the I-Cube chip on the 2181 board underneath. By configuring the *crossbar switch* a path, i.e. a physical link, can be established between remote processors. Unfortunately, these connections did not prove to be as reliable as they should. Various experiments with example applications showed that the ER2 is suffering from severe electrical problems which lead to an unstable and unpredictable behaviour (see 7.3.1 on pp. 39).

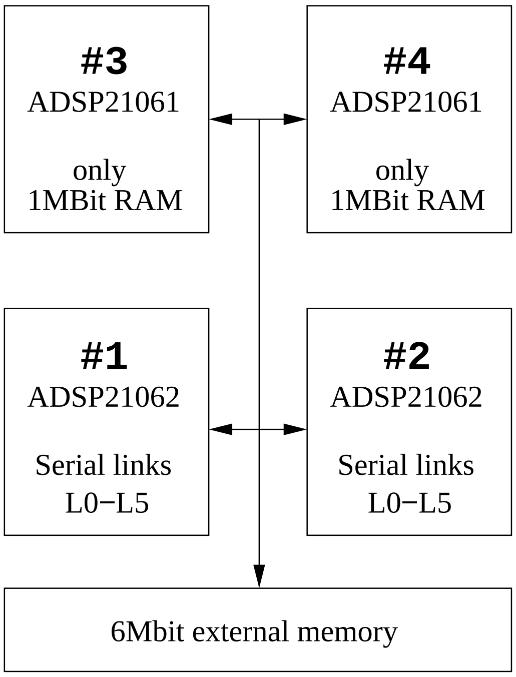


Figure 4: Basic layout of a SHARC cluster

3 The device driver `er2p`

3.1 Why a device driver?

The connection between the ER2 and a PC is established via an interface—also called *host adapter*. Triggered by special commands from an application on the PC, it reads/writes data from/to the *root processor* of the ER2 via the IDMA port (see fig. 5).

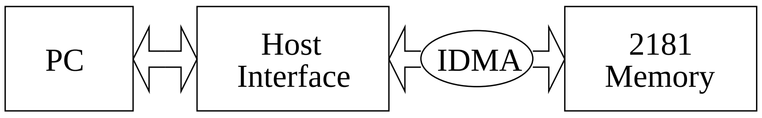


Figure 5: Access to the memory of the *root processor* via the host adapter

At the moment, two different *host adapters* exist. One is an ISA bus card that has to be inserted to the PC directly. The other host interface can simply be attached to the parallel port connector of the computer.
Considering the functionality of the two interfaces, no differences exist be-

tween them. The parallel port interface is only much slower (150 KByte/s max.) than the ISA card (750 KByte/s max.), resulting from the fact that it can not handle full 16-bit words, but operates with 8 bit instead (see [3]). It has to be noted that the ISA card interface operates together with the 2181 modules only. A write to a SHARC cluster via the *Fifth* PRS is immediately followed by a reset of the ER2 if the ISA card is used.
Both devices are controlled via writing to and reading from IO ports with `inb` and `outb` functions. Hence, programs in Linux that want to access the devices, i.e. get permissions for the appropriate ports, normally need to be run as *root*. This is, of course, somewhat awkward because every interested party should be enabled to develop applications for the ER2 without giving him the password of the superuser.
There exist several solutions for this problem, including *workarounds* like setting the *root execution* bit or using a daemon.
The normal way—and definitely the best—is to write a device driver. Being closely related to file handling (`open`, `read`, `write` ...) it does not only use a well defined interface to the kernel but also provides a useful and portable interface to applications. By hiding the necessary IO port accesses within the device driver one is able to write programs that will always work, regardless of which interface is used at the moment. If a new host adapter is ever developed, only the device driver has to be changed. All other source code stays exactly the same.

3.2 Basic structure

Linux handles external devices by the help of the *Virtual Filesystem Switch* (VFS). This is a part of the kernel that is, in short terms, responsible for translating general system calls like a `read` into the equivalent sequence of commands for the addressed device. This device could be a hard disk, a scanner or an external zip drive.
The readily compiled device driver contains appropriate functions that can handle requests like `read` or `open`. Additionally, a special `struct` is defined that keeps pointers to these *file operations*. When the device driver is loaded to the kernel, they are registered with the VFS and can then be used to access the device.
For each type of peripheral device, a special *device file* is created with an unique number—the so called *major* ID. On every access to a device file, the VFS checks the major number and immediately knows which function it has to call.
The device driver `er2p` supports both types of host adapters for the ER2. It is organized in two layers (see fig. 6) where the lower layer 1 consists of basic

IO routines for each host interface.

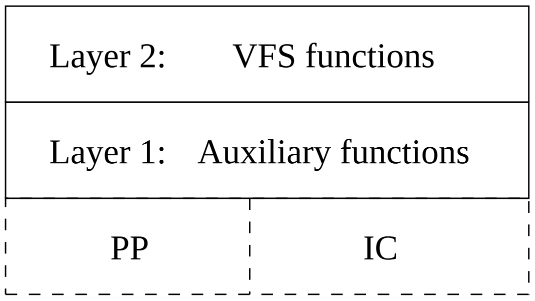


Figure 6: Layer structure of the device driver

In the following text and the source code of the driver, 'PP' is an abbreviation for 'Parallel Port', while 'IC' is short for 'ISA Card'. The auxiliary functions ensure that both interface cards can be accessed on the same level, regarding word size. Their main task is to translate the reads and writes of 16-bit words or addresses to two 8-bit transfers for the parallel port host adapter.
The layer 2 on top implements the *file operations* as needed by the VFS. All actions like reading a 16-bit data word or resetting the ER2 are triggered by a special VFS function called `ioctl` (short for Input Output ConTroL). It is normally used to transpose the device into another state, regarding the transfer of data. For example, the driver of a modem at the serial port might use an `ioctl` call to change the current baud rate. The `ioctl` routine can handle an arbitrary number of subroutines, that are specified by an ID given as parameter to the function.
The device driver `er2p` supports the following `ioctl` calls:

Command	**Description**
`IOCTL_ER2_RESET`	Resets the ER2
`IOCTL_ER2_IRQ`	Generates IRQ2
`IOCTL_ER2_WRITE_WORDS`	Writes 16-bit words
`IOCTL_ER2_WRITE_ADDRESS`	Sets 16-bit address
`IOCTL_ER2_READ_WORDS`	Reads 16-bit words
`IOCTL_ER2_SET_LENGTH`	Sets the number of words to read/write
`IOCTL_ER2_SET_INTERFACE`	Sets the used interface

The last `ioctl` subroutine specifies the adapter type—which can be either `PARALLEL_PORT` or `ISA_CARD`—that should be used for the following accesses. Each VFS file operation routine (layer 2) of the driver contains a switch

statement that executes the appropriate function from layer 1, depending on the selected host interface.
Figure 7 shows how the request for a `read` of a 16-bit data word is translated to the special auxiliary functions, depending on the selected adapter type.

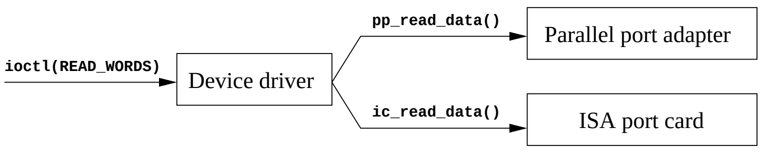


Figure 7: Control flow for a `read` operation

Since both interfaces are combined into one device driver, all IO ports have to be requested at once. Unfortunately, `ioctl` calls can only be sent to already opened device files. Thus, every time an `open` call reaches the device driver, it tries to claim the port regions:

Interface	**Port region**
ISA Card	`0x320-0x32F`
Parallel Port	`0x378-0x37A`

They are released as soon as the device file is closed again.

3.3 Using the device driver

The device driver `er2p` is compiled as kernel driver module, which means that it can be loaded and unloaded during runtime. For this, *root access* is needed. Then, the module can be added to the kernel by the command:

```
modprobe er2p
```

If everything went fine and the module was properly inserted, it should appear in `/proc/modules`. This can be checked with either

```
cat /proc/modules
```

or

```
lsmod
```

For removing the module again, one has to type:

```
rmmod er2p
```

As already mentioned, a *device file* with the proper *major number* is needed, too. It can be created by the command

```
mknod /dev/er2p c 219 0
```

if it does not already exist.
More information about the interna of the device driver `er2p`, its proper installation and the VFS in general can be found in [5, 17].

4 The ER2 library

4.1 General description

Via the device driver, memory locations of the *root processor* can be read or written. This enables the access to the *Fifth* PRS that is running on the single ER2 nodes after startup.
The *root processor* emits small data packets, which are then distributed around in the network. Single remote processors can be addressed but broadcasts to all nodes—or just a predefined group—are also possible.
The ER2 library collects routines that encapsulate these so called *messages* in higher-level functions. Following the guidelines in [15], most of the available PRS functions are supported, including

- network detection,
- memory access to remote processors and
- starting of programs on remote processors.

It hides the device driver accesses to single memory locations of the *root processor* from the programmer, offering a more abstracted view on the ER2 (see fig. 8).

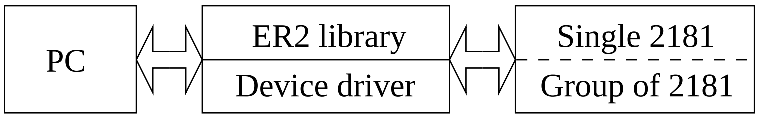


Figure 8: View on the ER2 via the library `er2`

Instead of only writing to the memory of the *root processor*, he can now address all nodes in the network directly.
The prototypes of the functions were defined, trying to meet the following requirements:

1. consistent naming of function parameters,

2. verbose function names and

3. support of error handling, where appropriate.

Additionally, all variables and functions for internal processing were declared *static*. The user has to call special *accessor functions* for reading or changing them. For example, two internal variables named `errors` and `verbose` exist that control the output of error and info messages, respectively. Their state can not be changed directly, the user has to call the function `errors_off()` in order to suppress the display of errors.
All together, the ER2 library can be seen as the *instance* of a *class*. It contains *private* variables and data structures, representing the internal state of the *object*, and offers *accessors* that can change this state. The underlying concept of *encapsulation* was borrowed from OOP (*Object-oriented programming*) and is also applied to the SHARC library in section 5 later on. The resulting *access control* helps in defining what the client programmer can use and separates the *interface* from the *implementation*. Anything that is not *public*, can easily be changed without requiring modifications to client code (see [8, pp. 261]).

4.2 Basic structure

The ER2 library can be roughly divided into two layers (see fig. 9). Layer 3 contains the routines for basic IO via the device file, i.e. the device driver `er2p`. None of them is available to the user, they are kept private. The upper layer 4 offers the higher-level functions—as listed in section 4.6, starting on page 22—to the programmer, enabling him to take actions like distributing data or loading code to some processors.
Among the user functions, the `message` plays a very important role. These already mentioned data packets, consist of a sequence of 16-bit data words, specifying what kind of action should be triggered on which processor. The lengths of the provided messages, i.e. the number of data words, can range from 1 up to 66.
In order to send a message, its data words have to be written subsequently into a destined location of the root processor's memory. For each word an

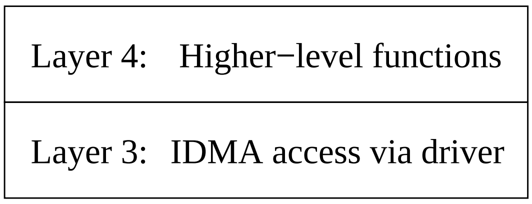

Layer 4: Higher−level functions
Layer 3: IDMA access via driver

Figure 9: Layer structure of the ER2 library

interrupt is generated, the 2181 then reads the memory location and decides what to do next, based on the contents of the new word (see [15, p. 5]).
The ER2 library uses these messages within higher-level functions like `broadcast_memory`, which transfers code or data to a remote processor. Inside the function, the appropriate message is constructed and then sent via the device driver.
A lot of library functions require the user to specify the processor and/or the destination address—beside saying which data to transfer. For giving the necessary information to the library, the following paragraphs explain some general concepts regarding the management of addresses and IDs throughout the `er2` functions.
The single boards can be distinguished by their *physical address*—a unique number in the range 0–255. These *physical IDs* are used by the *Fifth* PRS to route the message data packets through the network. While detecting the network topology at the startup of the ER2, the library also initializes a second array of *logical addresses* from *0* to *(n-1)*, where *n* is the number of nodes in the network. This is especially useful for loop constructions like

```
for (i = 0; i < get_number_of_nodes(); i++)
{
  /* Any action(s) involving the processor
          get_physical_address(i)
   */
}
```

where `get_number_of_nodes` returns the number of detected 2181 modules and `get_physical_address` translates from logical to physical addresses.
While specifying an address in the 2181 memory, again the concept of logical vs. physical addresses is used. The memory of the ADSP-2181 on an ER2 module ranges from physical address `0x0000` up to physical address `0x7FFF`. It can be split into 24-bit program memory (PM) from physical address `0x0000` to physical address `0x3FFF` and 16-bit data memory (DM) from physical address `0x4000` to physical address `0x7FFF`.

Following the standard imposed by the ADSP DOS-IDE utilities—which are further described in section 4.4.1—, logical addresses from 0x0000 to 0x3FFF are used in both: program **and** data memory. This suits the definitions in the architecture files (*.ach) best and means that logical addresses in PM will stay the same when converted to physical addresses. Logical addresses in DM are mapped by the library as follows:

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

In order to distinguish between program and data memory a special data type called memory_class is defined, which accepts the two values dat and prog.
The following call of broadcast_memory transfers the first 4 integer words of the array data to the single processor #0x34. Their destination is the logical address 0x1000 in data memory (DM), which results in the physical address 0x5000:

```
broadcast_memory(SINGLE, 0x34, 0x1000, dat, data, 4);
```

Instead of talking to only a single processor, data broadcasts to a whole group of nodes are also possible. An 2181 module can join a group by the function join_group. It takes the physical address of the processor and the group number—ranging from 0 to 63—as arguments. Unfortunately, the *Fifth* PRS does not allow processors to join more than one group but this is still sufficient for speeding up and simplifying certain data transfers, e.g. loading the same program to several nodes.
The call

```
broadcast_memory(14, 0x1, 0x1000, dat, data, 4);
```

differs from the one above in the first two arguments. Data is now loaded to all processors of the group #14 simultaneously. Since the special group 'SINGLE' is not given, the physical address, i.e. the second argument, is neglected and can be set to an arbitrary value.

4.3 Example program

A minimal wrapper for programs that want to use the ER2 looks like this:

```
#include <er2gdef.h>
#include <er2.h>

int main(void)
{
  startup_er2(PARALLEL_PORT);

  /* ... other actions ... */

  shutdown_er2();

  return(0);
}
```

First, the necessary headers are included. The file `er2.h` contains the prototypes of all functions in the ER2 library. Some general defines—common to the device driver and both libraries—are kept in `er2gdef.h`. It provides the definitions for the different host adapter types (`PARALLEL_PORT` and `ISA_CARD`) and for the function return values `OK` and `ERROR`.
Please, regard that the directory of the include files and of the library itself has to be known to the compiler. Assuming they are installed in their default places at `/usr/local/include/er2` and `/usr/local/lib/er2`, the options

```
-I/usr/local/include/er -L/usr/local/lib/er2
```

have to be added to the compiler call.
In `startup_er2` the device file `er2p` is opened. Then a special routine detects the network and initializes the routing table and the arrays for the translation between logical and physical processor addresses. Based on the function argument, either the `PARALLEL_PORT` or the `ISA_CARD` host adapter is used for this.
The section "other actions" may contain any access to the ER2 or another arbitrary C function. For example, the function `display_routing_table` could be called. It loops through the internal routing table and outputs its contents to `stdout`.
The function `shutdown_er2` at the end has only one task: it closes the device file again which results in releasing the claimed IO ports.

4.4 Loading and starting programs

4.4.1 DOS-IDE for the ADSP-2181

For the ADSP-2181 processors a small DOS-IDE by Analog Devices exists. This set of tools for developing assembler and C programs contains:

- the assembler `asm21`,
- the linker `ld21`,
- the C compiler `g21` and
- several simulators for the 21xx series.

Due to the lack of appropriate Linux tools this environment—more exactly, the *ADDS-21xx-SW-PC Development Software, Release 5.1*—is directly supported in the ER2 library. Generated programs can be loaded to ER2 nodes and started with a single function call each.
All of the ADSP tools can be run in *Dosemu*, the DOS emulator of Linux, without problems. Thus, the user still has only one operating system to manage during development.

4.4.2 Developing assembler programs

This is the architecture file that can be used to create assembler programs for the 2181 on the ER2:

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

If its name is `er2.sys` it can be packed to a `*.ach` file by the command:

```
bld21 er2.sys
```

The memory areas `0x0000`–`0x0FD7` in PM and `0x0000`–`0x0FFF` in DM contain routines of the *Fifth* PRS, so the architecture description leaves them untouched. In the range `0x0FD8`–`0x0FFF` (PM) the *Interrupt Vector Table* (IVT) of a program is located, right before the executable itself starts at address `0x1000`.
A corresponding assembler program should look like this:

```
.MODULE/RAM/ABS=0x0FD8  serial_receive;
.VAR/DM/RAM rcv_flag;
.INIT rcv_flag: 0;
.ENTRY init_port;
```

```
{set up interrupt table}
NOP; NOP; NOP; NOP;
NOP; NOP; NOP; NOP;
NOP; NOP; NOP; NOP;
NOP; NOP; NOP; NOP;
NOP; NOP; NOP; NOP;
NOP; NOP; NOP; NOP;
NOP; NOP; NOP; NOP;
NOP; NOP; NOP; NOP;
NOP; NOP; NOP; NOP;
JUMP RECVE; NOP; NOP; NOP;

init_port:
           {main program, initializations}
           AX0 = 1;
           DM(rcv_flag) = AX0;
           IDLE;
END:       RTS;

{interrupt routine}
RECVE:     AX0 = DM(rcv_flag);
           AR = AX0 AND 0x0001;
           IF NE JUMP SEND_ACK;
           RTI;
```

After some assembler directives for defining variables and the entry point of the 'main' routine, the first 40 commands represent the IVT. A '`NOP`' means: "Do not overwrite the current entry in the IVT.". The program—starting at the label `init_port`—runs into an IDLE loop. Upon a triggered interrupt from the serial port, the subroutine `RECVE` checks the flag `rcv_flag`. If it is not equal to zero, an acknowledge is sent. Please, note that the required routine `SEND_ACK` is not included in this listing.
The source file can be assembled by:

```
asm21 ser_rec
```

and then linked to an executable with the command:

```
ld21 ser_rec.obj -a er2 -e ser_rec.exe
```

4.4.3 Executing programs

The executables, output by the Analog Devices assembler `asm21`, consist of simple ASCII text lines. Single sections tell which data or program code should go to which memory location. For example, the small program

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

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

results in the executable file:

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

```
#12300002000
IIo
```

Quickly, one discovers the simple structure of these files. They consist of *blocks*—either for data or program memory—supplied with an address. The program memory blocks contain ADSP-2181 instructions in hexadecimal format, whereas data memory blocks care about the initialization of variables.
The function `read_program` is able to read an ASCII executable and loads it to the specified processor. Afterwards, it can be started by the routine `broadcast_start_program` from the ER2 library.
The following program reads the executable `toggle.exe` and loads it to processor #171 in the network. Afterwards, the process is started at address `0x1000` in program memory.

```
#include <er2gdef.h>
#include <er2.h>

int main(void)
{
  startup_er2(PARALLEL_PORT);

  read_program("toggle.exe", 0x0, 0x1);
  broadcast_start_program(SINGLE, 171, 0x1000, 0xFF);

  /* ... other actions ... */

  shutdown_er2();

  return(0);
}
```

4.5 Configuring and testing crossbar links

The crossbar switches can be used to connect ports of two ADSP-2181 processors e.g. for serial communication. These connections between two distant modules can be viewed as *edges* of a *communication graph*—regardless of their width in number of bits. For embedding this graph into the *crossbar switch structure* of the ER2, an *edge* may have to cross several intermediate processors, where *pass-through* connections have to be defined. These *subparts* of an *edge* are called *links* from now on.
A special *message* is provided, configuring a single *link* for a 3-bit wide serial communication *edge* (see [15, p. 6]). Support for this *link message* is not included in the current version of the ER2 library.

Instead, a different, more basic, approach is selected that enables the user to switch 1-bit connection *edges*. Several of these can be combined to 3-bit serial communication *paths* (or even 6-bit for the SHARCs) again. Additionally, the single connections can be tested.
By inserting modules into an ER2 backplane, they are physically connected to their *neighbour's pins* as shown in figure 10, which was derived from [18]. The small hexadecimal numbers denote the accessible crossbar ports and their antipodes in the four directions.

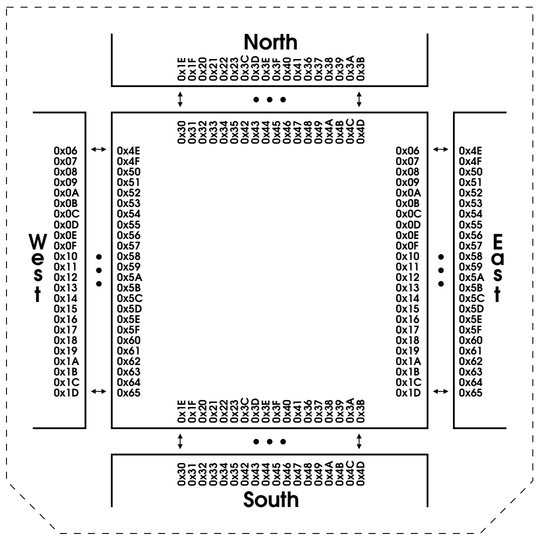

North
0x1E 0x1F 0x20 0x21 0x22 0x23 0x3C 0x3D 0x3E 0x3F 0x40 0x41 0x36 0x37 0x38 0x39 0x3A 0x3B
0x30 0x31 0x32 0x33 0x34 0x35 0x42 0x43 0x44 0x45 0x46 0x47 0x48 0x49 0x4A 0x4B 0x4C 0x4D
West
0x06 0x07 0x08 0x09 0x0A 0x0B 0x0C 0x0D 0x0E 0x0F 0x10 0x11 0x12 0x13 0x14 0x15 0x16 0x17 0x18 0x19 0x1A 0x1B 0x1C 0x1D
0x4E 0x4F 0x50 0x51 0x52 0x53 0x54 0x55 0x56 0x57 0x58 0x59 0x5A 0x5B 0x5C 0x5D 0x5E 0x5F 0x60 0x61 0x62 0x63 0x64 0x65
East
South

Figure 10: Available pin connections

All information about the *communication graph* is stored in a linear list, where each processor gets an entry, consisting of two additional linear lists—one for *links*, the other for *edges*.
A *link* is simply a pair of port numbers:

```
struct link
{
  /** First port of the crossbar connection */
  int link_porta;
  /** Second port of the crossbar connection */
  int link_portb;
  /** Pointer to next entry */
  struct link *next_link;
};
```

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

```
struct edge
{
  /** The port of the actual processor that is connected with
  another processor in the network */
  int port;
  /** The physical address of the connected processor */
  int conn_proc;
  /** The port of the connected processor */
  int conn_port;
  /** Pointer to next entry */
  struct edge *next_edge;
};
```

Again, all these data structures are internal and not visible to the user. New entries to the tree of configuration data have to be added by calling the two functions `insert_link` and `insert_edge`.
If a single *edge* is specified as: "Let the port `0x0D` of processor `0x45` be connected to port `0x64` of processor `0x32`.", this also implies a connection in the opposite direction. Crossbar switches work bidirectional, so for every *hardware* path one gets two *logical* connections, i.e. each call of `insert_edge` actually inserts two entries.
Once that the configuration of all processors is complete, the function `configure_er2` can be called. In a first pass it loads a special assembler program to all nodes that are concerned. Then, the list of *links* is transferred to each single 2181. Finally, the program is started in parallel on all boards. It reads the list of crossbar connections and switches them automatically.
In a similar manner, the made connections are tested by the function `test_all_edges`. Depending on the *edge* lists specified by the user, it loads an assembler program to the processors and tests all hardware lines. Returned are the number of errors, i.e. unsuccessful tests. More information about the assembler programs can be found in [4].
So far, the user has to provide a function—or at least some kind of loop—calling `insert_link` and `insert_edge` repeatedly for constructing the *communication graph.* If the configuration is static, it is desirable to relieve the user from the burden of manually setting up the *data tree* each time.
For this, the function `read_configuration_data` was developed. It reads a simple text file that contains the needed informations for building the *configuration tree.*

The syntax for this file—given in BNF—is:

```
configuration: list_of_blocks

list_of_blocks: block
              | block NL list_of_blocks

block: processor NL link_block NL edge_block

link_block: '0'
          | number NL links

links: link
     | link NL links

edge_block: '0'
          | number NL edges

edges: edge
     | edge NL edges

link: HexInt HexInt

edge: HexInt HexInt HexInt

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

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

The terminal NL denotes a *newline*, so each entry has to stand on a single line.
If the externally created file could be read successfully, the call of configure_er2 sets up the desired connections. The following basic wrapper program reads the configuration file test.cnf, configures the specified crossbar connections and tests them automatically:

```
#include <er2gdef.h>
#include <er2.h>

int main(void)
{
  verbose_on();
  startup_er2(PARALLEL_PORT);

  read_configuration_data("test.cnf");
  configure_er2();
  test_all_edges();

  /* ... other actions ... */

  free_configuration_list();
  shutdown_er2();

  return(0);
}
```

The function `free_configuration_list` should be called before or after shutting down the ER2. It frees all memory for the *link* and *edge* entries that were allocated during configuration.

4.6 List of available functions

At the end of this section, a list of all available functions for the ER2 library is presented. The prototypes can be found in the header file `er2.h` or the *Literate Programming* document [4], which also gives further insight to the operation of the single routines.

Function	Description
verbose_on	Switches on the display of verbose messages
verbose_off	Switches off the display of verbose messages
errors_on	Switches on the display of error messages
errors_off	Switches off the display of error messages
root_write_address	Sets current IDMA address for root processor
root_write_data	Writes data to root processor via IDMA bus
root_read_data	Reads data from root processor via IDMA bus

continued on next page

continued from previous page

Function	**Description**
message	Sends a message with no result
message_wfr	Sends a message and returns the 24-bit result
message_wfr_x	Sends a message and returns the 48-bit result
request_memory	Reads data from the memory of a remote processor
broadcast_memory	Transfers data to the memory of a remote processor
join_group	Sets the group ID for a node
display_routing_table	Displays the detected network topology
get_neighbour	Returns the ID of the neighbour node in the given direction
get_number_of_edges	Returns the number of edges in the network
get_number_of_nodes	Returns the number of nodes in the network
get_physical_address	Returns the physical address for a node
get_logical_address	Returns the logical address for a node
read_program	Reads an 2181 assembler executable to the memory of a remote processor
read_program_c	Reads an 2181 C compiler executable to the memory of a remote processor
root_start_program	Starts a program on the root processor
broadcast_start_program	Starts a program on a remote processor
insert_link	Adds a link to the configuration data
insert_edge	Adds an edge to the configuration data
free_configuration_list	Frees all memory, allocated for the configuration data
configure_er2	Sets up all specified crossbar connections from the configuration data
read_configuration_data	Reads the configuration data from an external file
load_test_program	Loads the program for testing connections to a remote processor

continued on next page

continued from previous page

Function	Description
start_test_program	Starts the program for testing connections on a remote processor
test_edge	Tests a single connection between two remote nodes
stop_test_program	Stops the program for testing connections on a remote processor
test_all_edges	Automatically tests all specified connections of the configuration data
startup_er2	Automatically opens the device file, resets the ER2 and detects the current network topology
shutdown_er2	Closes the device file

5 The SHARC library

5.1 General description

Having gained control over the 2181 modules, the next logical step is to provide support for the SHARCs. Figure 11 shows the layer structure of the SHARC library `er2sh`.

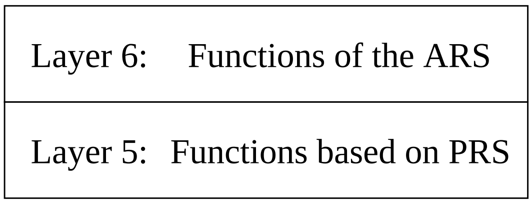

Layer 6: Functions of the ARS
Layer 5: Functions based on PRS

Figure 11: Layer structure of the SHARC library

The layer 6 is built by some special routines, supporting an extension to the *Fifth* PRS that is described in section 5.3. At the bottom, layer 5 supports the *Fifth* PRS and encapsulates various functions for loading data or starting programs on the SHARC clusters.
Especially the boot sequence for the SHARCs is worth mentioning: After a reset of the ER2 no 2181 module initially knows whether a SHARC cluster is attached or not. As described in [16], two binary files have to be loaded to each ER2 node. The file `s21cde.dat` contains a boot loader program for the 2181, while `shcde.dat` contains the *Fifth* PRS routines for the SHARCs.

They are distributed in the network by a single broadcast each. This is achieved by first building a special group `WOROOT`, which contains all 2181 modules except the root processor. Upon the start of the boot loader, the contents of `shcde.dat` are transferred to the SHARC cluster—whether it is present or not—and the SHARCs are started. A small delay is added, to ensure that the boot process has finished on all clusters. This whole process is combined to the single function `boot_sharcs`.
If a cluster is attached it sets the bit #8 of the variable `MASTER`, located at `0x222` in the data memory of the 2181. The routine `detect_sharcs` checks this variable on each ER2 node and enters all booted SHARC clusters into the ID tables of the library `er2sh`. Again, two internal tables and a set of accessor functions enable the translation between *physical addresses* and *logical addresses*.
In contrast to the 2181 modules, the SHARC clusters do not have an own unique number that can be used for addressing the single boards. Thus, the ID of the underlying ER2 node is used instead.
The following call of `write_to_sharc` transfers the program word `0x0A3E00000000` (= ReTurn from Subroutine) to the single processor #`0x25`:

```
write_to_sharc(SINGLE, 0x25, 0x2, 0x20004, 0x0A3E, 0x0000, 0x0000);
```

According to the structure of the equivalent message in [16, p. 2] the third argument specifies the *address code*. The SHARC address codes range from 1–7, where 1–4 are the internal memories of the single 2106x. Codes 6 and 7 specify the external RAM, starting at address `0x00400000`.
The next argument gives the destination address, within the selected address code range. Here, the data is written to the physical address `0x20004` of SHARC #2.

5.2 Example program

A minimal wrapper for programs that want to use the SHARC library `er2sh` looks like this:

```
#include <er2gdef.h>
#include <er2.h>
#include <er2sh.h>

int main(void)
{
  startup_er2(PARALLEL_PORT);
  boot_sharcs();
  detect_sharcs();

  /* ... other actions ... */

  shutdown_er2();

  return(0);
}
```

After the startup of the ER2, the function boot_sharcs loads the files 21cde.dat and shcde.dat to all nodes and starts the SHARC boot loader as described in [16]. Then, the routine detect_sharcs detects all booted SHARC clusters.
In “other actions” any routine for the ER2 or from the SHARC library er2sh can be called, except the ARS routines which own the prefix “ars_”.

5.3 Additional runtime system (ARS)

5.3.1 Basic layout

A dump of the SHARC IVT shows that it got completely overwritten by the routines of the *Fifth* PRS—i.e. the contents of shcde.dat—, up to address 0x200FD (see also [7, pp. 22].
A simple load of an executable, created by the SHARC C compiler g21k would destroy the routines that are used to communicate with the 2181 module below. The address space 0x20000–0x201FF is used for the interrupt vector 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.
Hence, a few additional routines—called ARS (*Additional Runtime System*) from now on—were developed, holding the conditions:

1. No interrupts get overwritten or redirected, except the reset vector. So the C programs that are to be executed can use them as they like.

2. All *interrupt vector tables* (IVT) and internal memories of the single SHARCs are unoccupied, except a small area from 0x20100–0x20300.

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.

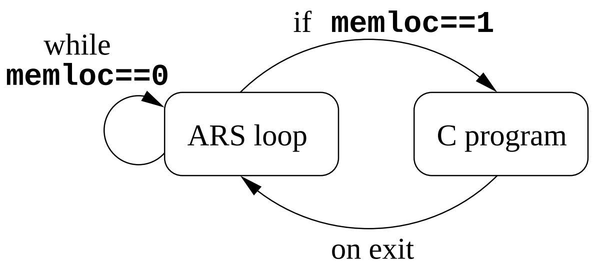


Figure 12: Basic loop for the ARS

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.

5. Loading data and starting processes per '*broadcast*' is possible, i.e. with a single call all four SHARCs in a cluster—or a nonempty subset—can be supplied with a new program.

Like in the original *Fifth* PRS, complemented by the boot code `shcde.dat`, the single processors each run in a small loop (see fig. 12). If a special memory location gets written, they jump to the specified start address and execute the loaded program. After the C program is finished, the loop of the *Additional Runtime System* is entered again.
Although all SHARCs have direct access to the 2181 memory via IDMA in principle, only #4 on each cluster is 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 may 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 signaling 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 *process synchronization* functions, described in [7] and 5.3.2—takes place within the C programs itself.
Some basic functions were written in SHARC assembler and are made available in C via the G21k ARS library (see [6]). Especially assembler programs might need a more direct access to these routines, so the following table lists all addresses occupied by the ARS:

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

5.3.2 Synchronizing processes within a SHARC cluster

The ARS functions `idma_read` and `idma_write` can be used to read parameters or return results to the 2181 modules. They access the 16-bit IDMA port of the DSP as a gateway to all internal memory locations. Some address lines of the SHARC cluster are used as control signals and every 2106x is able to use the IDMA port. Unfortunately, this parallel port can not be shared, so the user has to ensure that no two SHARCs try to claim it at the same time.
In general, the possibility to *synchronize* processes on a cluster can be regarded as a very helpful feature. Thus, a few ARS functions were developed that fulfill this task. The pair `ars_signal` and `ars_wait` can be used to synchronize two different SHARCs on a cluster. One SHARC, the *sender*, *signals a request* to the *receiver*. Once, the receiver enters his matching `ars_wait` function it detects the request and *sends an acknowledge*. Both functions block the execution of each program until the acknowledge is exchanged.
In a similar way the routine `ars_barrier` can be called and synchronizes all four SHARCs on a cluster. Since separate memory locations are used for the internal synchronization protocols, different calls to `signal/wait` may overlap in respect to the time axis: If a SHARC sends a signal to #3, it can meanwhile receive a signal from #1.
Information about the internal structure of the ARS synchronization functions can be found in [7, pp. 30]. Several of the examples on the ER2 CD (see Appendix B) demonstrate their usage.

5.3.3 Using the ARS routines

A minimal wrapper for programs that want to use the ARS routines from the SHARC library looks like this:

```
#include <er2gdef.h>
#include <er2.h>
#include <er2sh.h>

int main(void)
{
  startup_er2(PARALLEL_PORT);
  boot_ars();

  /* ... other actions ... */

  shutdown_er2();

  return(0);
}
```

The function `boot_ars` first calls the routines `boot_sharcs` and `detect_sharcs`. Additionally, every 2181 module that has a detected SHARC cluster attached, joins the predefined group `ALL_CLUSTERS`. Then, the ARS is installed and started.
In "other actions" any routine from the ER2 library or an ARS function, starting with "`ars_`", may be called. Please, note that the plain *Fifth* PRS functions for the SHARCs do not work anymore. At this point the control flow is already transferred to the ARS loops on the single clusters, so the *Fifth* PRS does not listen to any requests.

5.4 List of available functions

At the end of this section, a list of all available functions for the SHARC library `er2sh` is presented. The prototypes can be found in the header file `er2sh.h`. The *Literate Programming* document [6] explains the design of the ARS and its single functions in greater detail.

Function	Description
boot_sharcs	Tries to boot the SHARC clusters
detect_sharcs	Detects for which nodes the SHARC boot was successful

continued on next page

continued from previous page	
Function	**Description**
get_number_of_clusters	Returns the number of detected SHARC clusters
get_physical_cluster_address	Returns the physical address of the SHARC cluster
get_logical_cluster_address	Returns the logical address of the SHARC cluster
write_to_sharc	Writes a single 48-bit word to a SHARC
read_from_sharc	Reads a single 48-bit word from a SHARC
load_sharc_hex_data	Loads a file with hexadecimal data to a SHARC
start_sharc_program	Starts a loaded program on a SHARC
boot_ars	Installs and boots the ARS
ars_broadcast_memory	Transfers memory to a remote processor via ARS
ars_load_program	Loads an executable to a cluster via ARS
ars_start_cluster	Starts a loaded program on a cluster via ARS

6 The G21k utils

6.1 Compiling C programs for the SHARC

The G21k C compiler stems from an early version of the `gcc`. Adapted to the 2106x processor series by Analog Devices, its source code is now publicly available on the Internet. Together with some binary tools, the so called *binutils*, the whole G21k package provides:

- an assembler (`asm21k`),
- a linker (`ld21k`),
- a C compiler (`g21k`) and
- a tool for dumping 21k COFF executables (`cdump21k`).

The development cycle for executables follows the usual paths as depicted by figure 13.
The C compiler, i.e. the linker `ld21k`, supports several processor types and needs to know certain details about the architecture of the machine it generates code for. The memory layout for the current processor is defined in a

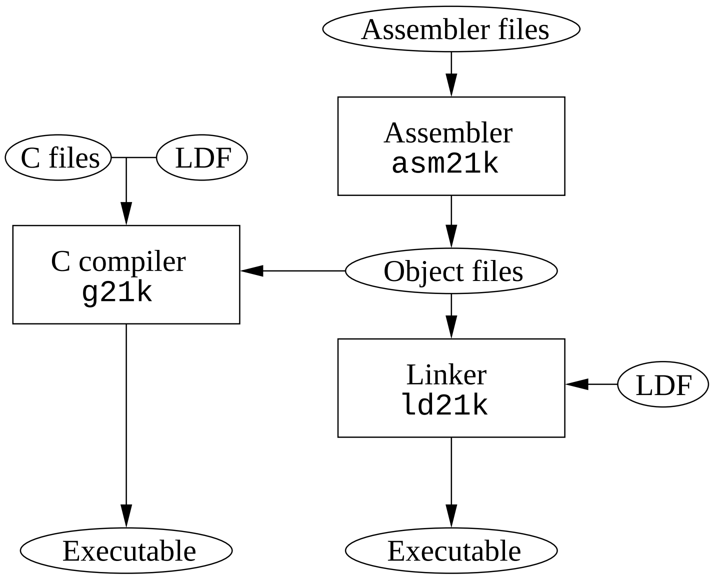


Figure 13: Program development with the G21k utils

Linker Description File (LDF) that is given as argument to the linker call. Two development scenarios are possible in general:

1. Only the *Fifth* PRS is used to load data and start programs. Then, the executables have to be developed in Assembler only. The architecture file should exclude the *Fifth* PRS section by defining a segment

   ```
   .segment/ram/begin=0x00020000 /end=0x000200ff /pm seg_rth;
   ```

   that is not used within the program. The user is free to define additional segments for his code and data. No restrictions are lying upon the usage of memory block #1 as for scenario 2.

2. C programs are to be developed with the compiler `g21k`. Loading and starting the executables requires that the ARS is used, so the segment

   ```
   .segment/ram/begin=0x00020100 /end=0x000202ff /pm seg_ars;
   ```

   needs to be defined in the LDF. It reserves the memory space for the ARS routines. The user has to be aware of some restrictions that are put on the C *Stack* and *Heap* as described at the end of section 6.3.

Complete architecture files (LDFs) can be found in the 'examples' on the ER2 CD (see Appendix B). Together with the accompanying source code and the `Makefiles` they provide a starting point for new applications.

6.2 The archiver `ar21k`

A small problem was left out of the discussion so far. The G21k utils do not include a Standard C library (`libc.a`) which is needed by the compiler—at least for basic functions like `exit`. Even worse, an appropriate *archiver* for creating the required file is also missing.
Based on the source code for the linker `ld21k` a simple librarian utility, named `ar21k`, was developed. Its syntax is:

```
ar21k [-h] -o archive file1.obj file2.obj ...
```

In the first pass, the program reads each object file and tries to detect the symbol table for each contained section. If a symbol directory was found, all labels—like entry points, functions, variables—that are supposed to be global are stored. This is done in `process_symbol_table` (`syms.c`) where global symbols are recognized by their special type `C_EXT` and then added to the list of string entries for the current file. During this whole process, additional information—like the size of the single object files—is collected.
In the second pass, the archive header is written to the new file. It contains some general info about the library and the list of symbols that are made available by the archive. For each object file, a special header is constructed, followed by a dump of the file itself (fig. 14).

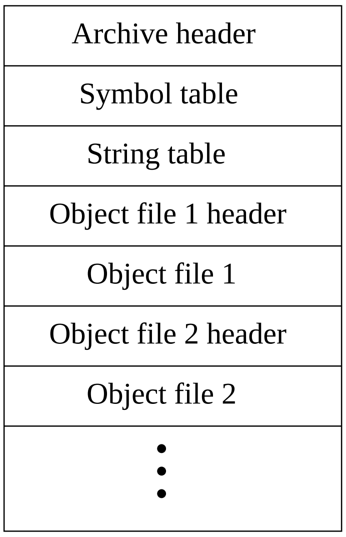


Figure 14: General structure of an archive file

6.3 Creating the Standard C and Math library

The archiver `ar21k` was used to create a Standard C library `libc.a` and a Math library `libm.a`, based on the assembler sources from the VisualDSP IDE. Several 'patches' to the assembler and C sources had to be applied for this. For their documentation in the rest of this section the following abbreviations are introduced:

Macro	**Expanded path**
`ASM`	`er2cd/sharc/g21k-binutils-PL4/asm`
`LINK`	`er2cd/sharc/g21k-binutils-PL4/link`
`INC`	`er2cd/sharc/g21k-binutils-PL4/include`
`LIBC`	`er2cd/sharc/libs/libc`
`LIB060`	`er2cd/sharc/libs/lib060`

They can be regarded as macro defines that expand to the full path specification wherever they are used.

1. All C++ style comments "`//`" had to be replaced by normal C comments.
2. Several labels contained a '.' within their name, especially at the end of a routine (example: `._abs.end`). The dots were replaced by underscores.
3. The three macros `write_instr_and_call`, `write_first_instr` and `write_second_and_call` (defined in `LIBC/irptl.h`) were expanded in the ASM files that used them, because `asm21k` does not support define macros longer than one line or with parameters.

A lot of global symbols had a name longer than 8 characters and would not be exported by the original version of the assembler `asm21k`. So, the maximum length for symbols was increased from 8 to 64 by:

- changing `SYMNMLEN` in `INC/a_out.h`,
- changing `MAX_SYMBOL_NAME_SIZE` in `ASM/symbol.h` and
- adding a dummy string named `x_dummy_name` in `INC/auxent.h` to the union `auxent`, since it needs to have exactly the same size as the struct `syment` from `INC/syms.h`.

This means that object files and archives—created by `ld21k` and `ar21k`, respectively—are incompatible to other linkers, even if they use the COFF format too.
The *Heap* and *Stack* spaces also underly certain restrictions:

1. The *Stack* may reside in DM only.

2. Only one *Heap* is supported at the moment.

The values for initializing the *Heap* and *Stack* are exported via a temporary object file that is created during the linking phase (`LINK/create_object.c`). It contains the symbols

```
ldf_stack_space
ldf_stack_length
ldf_heap_space
ldf_heap_length
ldf_heap_type.
```

They are used in `LIBC/set_env.asm` to set the variables

```
___lib_stack_space
___lib_stack_length
___lib_heap_descriptions
```

to their initial values.
Since the *Stack* may reside in DM only, the mode for memory block #1 is set to 32-bit—instead of the 40-bit mode used for the *Fifth* PRS. Otherwise the function calls in C programs would never return because the 40-bit mode disturbs the stack management. The appropriate setting of the `SYSCON` register is done in `LIB060/060_proc.asm`.
Together with a collection of ARS routines (`libars.a`) the created libraries can be used by the G21k C compiler or linker. So, if a function like `exp` is used within the C program, the option "`-lm`" should be added to the call of `g21k`.
More information about the required C headers and the syntax of single functions is available in [6].

7 Example application: Cholesky factorization

7.1 Prerequisites

The *Cholesky factorization* of a matrix $A \in \mathbb{R}^{n \times n}$ is the decomposition to

$$A = GG^T$$

where $G \in \mathbb{R}^{n \times n}$ is referred to as the *Cholesky triangle.* As proven in [9, p. 143], this factorization exists if A is *symmetric positive definite*, i.e.

$$a_{ij} = a_{ji}$$

and

$$x^T A x > 0 \quad \forall \quad x \in \mathbb{R}^n \setminus \{\mathbf{0}\}.$$

Analogous to *LU decomposition*, the *Cholesky factorization* is one of the basic algorithms for scientific numeric computing. It is used in a wide range of applications and gains its importance from the fact that a lot of problems—and therefore the describing matrices—are symmetric by nature.
For example, in [19] the factorization is used to compute the inverse of a matrix within every step of the *Karmarkar* algorithm. This is an algorithm for solving large scale linear programming (LP) problems, which are often encountered in communication, transportation industries and military operations. The number of variables for typical problems like *tele-traffic routing* runs high, usually well-above a million with the number of constraints exceeding a hundred thousand:

$$\text{Minimize} \quad \sum_{q=1}^{Q} M_q Y_q \ ,$$

$$\text{subject to} \quad \sum_{j=1}^{J} Z_{jk}^{h} = G_k^h \ ,$$

$$\sum_{j=1}^{J} P_{jk}^{qh} Z_{jk}^{h} \leq Y_q \ ,$$

$$0.0 \ll Z_{jk}^{h} \leq UB_{jk}^{h} \ , \quad \text{where}$$

M_q	:	incremental link cost in terms of \$ per carried load on link q
Y_q	:	maximum carried load on link q
Z_{jk}^h	:	carried load during period h, through path j of node-pair k
G_k^h	:	total carried load between node-pair k during period h
P_{jk}^{qh}	:	"1" if link q is on path j of node-pair k during period h, "0" else
UB_{jk}^h	:	upper bound value on Z_{jk}^h

Following, a few general uses: the *Cholesky factorization* of a matrix A can speed up solving the *linear equation*

$$Ax = GG^Tx = b$$

for several different right hand sides b_i. First, *forward substitution* is applied to $Gy = b$ where $y = G^Tx$. Then, the latter equation is solved by *backward substitution.*
Another use is the *symmetric-definite generalized eigenproblem*

$$Ax = \lambda Bx; \quad A, B \in \mathbb{R}^{n \times n}$$

for the quite common case that B is not only *symmetric* like A, but also *positive definite.* Then, as shown in [9, pp. 463] and [11, pp. 307] the *Cholesky factorization* helps in reducing the problem.
Third, as hinted at in [10, p. 225] one can find out if a *symmetric* matrix A is *positive definite* by applying a *Cholesky algorithm* to it, instead of computing all *eigenvalues.* If the factorization finishes with strictly positive square roots, the matrix is in fact *positive definite.*

7.2 The parallel algorithm

The following descriptions and their notation refer to 4.2.4 and 6.3.1 in [9, pp. 143, pp. 300] where a parallel algorithm for the *Cholesky factorization* is presented.
Starting with

$$A = GG^T$$

the jth column can be expressed as

$$A(:, j) = \sum_{k=1}^{j} G(j, k)G(:, k)$$

while using common Matlab notation. $A(i, j)$ equals the element a_{ij} of the matrix A. The ":" is used to specify a range, so $A(1{:}3, 5)$ is a column vector built from the first three elements in column 5 of matrix A. A single ":" is an abbreviation for the range "1:n".
Drawing out the case $k = j$ of the sum and using the fact that G is lower triangular yields

$$G(j, j)G(j{:}n, j) = A(j{:}n, j) - \sum_{k=1}^{j-1} G(j, k)G(j{:}n, k) \equiv v(j{:}n). \qquad (1)$$

Since obviously $G(j,j) = \sqrt{v(j)}$, the jth column of the Cholesky matrix can be computed by

$$G(j{:}n, j) = \frac{v(j{:}n)}{\sqrt{v(j)}}. \tag{2}$$

The basic idea of the parallel algorithm is to distribute the computation of the n columns evenly among the p processors in a ring. Each single processor $\mu \in \{1..p\}$ is responsible for a maximum of $L_{max} = \lceil n/p \rceil$ columns. Receiving the needed column vectors with $k < \mu$ from the left, it sums them up in a local array according to (1). Then, the scaling from (2) is applied and the new Cholesky column is passed on to the right neighbours.
The following C code for the node μ assumes that the column vectors

$$a_\mu \ldots a_{\mu+(L-1)p}$$

are stored in the local array `aloc`, where L is the number of G-columns to be produced. This kind of 'folded' assignment provides a better load balancing. The integer n is set to the rank of the matrix A, while `right` keeps the index of the right neighbour in the ring. The index of the last column that '`right`' has to compute is stored in `rfin`. The functions `send_column_to_right` and `receive_column_from_left` send and receive the current G-column in the array `gloc`, respectively. For the decision whether to pass a column on or not, it is important to know which processor generated it. Basically, this can be computed since the algorithm is known but it seems easier to 'mark' the sent columns with the local ID. The function `set_gloc_id` appends an appropriate integer value to the array `gloc` before sending it. This processor index can be read out with `get_gloc_id` again. Finally, the routine `col` returns the index of the local column in the total matrix A.

```
int j = 0;        /* Next column to receive */
int q = 0         /* Next column to compute */
int i, k, r;      /* Counters */
float ftemp;      /* Stores sqrt(A(j,q)) */

while (q < L)
{
  if (j == col(q))
  {
    /* Form new G-column G(j:n,j) */
    ftemp = sqrtf(aloc[j*L+q]);
    for (i = j; i < n; i++)
    {
      aloc[i*L+q] /= ftemp;
      gloc[i] = aloc[i*L+q];
    }
    if (j < (n-1))
    {
      /* Mark column with ID */
      set_gloc_id(mu);
      /* Send new column */
      send_column_to_right();
    }
    j++;

    /* Update local columns */
    for (k = (q+1); k < L; k++)
    {
      r = col(k);
      for (i = r; i < n; i++)
        aloc[i*L+k] -= aloc[r*L+q]*aloc[i*L+q];
    }
    q++;
  }
  else
  {
    /* Receive new column */
    receive_column_from_left();
    if ((right != get_gloc_id()) && (j < rfin))
    {
      /* Send column */
      send_column_to_right();
    }

    /* Update local columns */
    for (k = q; k < L; k++)
    {
      r = col(k);
      for (i = r; i < n; i++)
        aloc[i*L+k] -= gloc[r]*gloc[i];
    }
    j++;
  }
}
```

7.3 Implementation on the SHARCs

7.3.1 Electrical problems

Originally, it was planned to embed the ring of processors on the ER2 as sketched in the introduction: Computations are done on the SHARCs which are coupled by dedicated serial link connections via the 2181 modules and their crossbar switches.
However, running the first examples showed that the ER2 suffers from severe electrical problems. They result in an unstable and unpredictable behaviour, e.g. several restarts may be necessary to get a program to run.
As a pre-stage to the *Cholesky* program a special test was developed and carried out. Two SHARC clusters, one as *sender* the other as *receiver*, transfer a single column of 512 words. This transfer is done via their fast serial links that are connected by the crossbar switches of the 2181 modules below. Both modules are direct neighbours on the backplane, so the lengths of the single data lines are the minimum of what can be achieved. Additionally, a small program called `probe.exe` was developed for the ADSP-2181. It can probe single crossbar ports for their state (`HIGH` or `LOW`) and also apply a signal to a dedicated line (see directory `probe` in the 'examples').
For each of the six test configurations that follow, the program was executed 50 times, trying to transfer data from one SHARC to the other.

Test A : Link fully configured, edges tested in both directions.

Test B : Link fully configured, edges not tested at all.

Test C : Link fully configured, edges tested in one direction (from sender to receiver).

Test D : Link fully configured, edges tested in one direction, `ACK` tested from receiver to sender, all other lines from sender to receiver.

Test E : `ACK` set to `HIGH` for the sender.

Test F : `ACK` set to `LOW` for the sender.

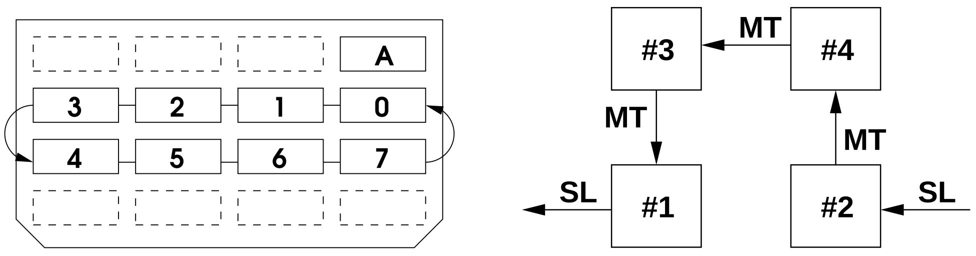
The following table shows the results:

Test	Success	Faulty	No receive	No send/receive
A	38	1	11	1
B	2	0	48	0
C	25	1	24	1
D	37	2	11	2
E	46	2	4	0
F	0	0	0	50

The column '**Faulty**' gives the number of successful but faulty transfers, i.e. all data was sent and received but a single nibble was either gobbled or wrongly inserted, preferably while transmitting the first 32-bit word.
Very interesting is the fact that there are no result levels in between. Either all words are received, or none.
The 'trigger' function of the program `probe.exe` was used to additionally check the lines of the SHARC links without connecting them at all. It verified that all data lines `D0`–`D3` and the `CLK` of the *sender* are working and driving the used crossbar ports `0x6E`–`0x72`. The `ACK` from the *receiver* keeps the port `0x7F` at high state like it should—signaling that it is ready to receive data.
The tests show that crossbar switches provide no reliable connection between two remote SHARC clusters. The problem does not seem to be connected to the configuration of the link ports and buffers. Neither do the setup or the testing of the crossbar switches have a negative influence on the following transmission. The mentioned fact that either all data words are received or nothing, might hint at a problem with the bidirectional crossbar switches. Perhaps they decide to drive some lines—probably the `ACK`—the 'wrong way' under certain conditions. How these conditions look like, should be further investigated.
Based on the test results, no configuration of crossbars is involved in the example program. Instead, the necessary serial lines are established by short flat-band cables that are attached to the external connectors for link #1 on the SHARCs #1 and #2, respectively.

7.3.2 Arrangement on backplane

Figure 15(a) documents the arrangement of the single SHARC clusters on the used ER2 backplane for the example program `ring8_cholesky`. The '`A`' stands for the host adapter, the clusters 0–3 build the *back row* and 4–7 the *front row*.
They are directly connected to an unidirectional ring by flat-band cables. The counter-clockwise direction of data flow, i.e. the direction in which computed columns are passed on, is indicated by the arrows.
Within a cluster at the *back row*, the columns from the *previous* neighbour are received by SHARC #2. Then, the data is sent to #4, #3 and finally to #1 that passes the columns to the *next* neighbour in the ring of clusters (see fig. 15(b)). At the *front row*, this direction is reversed.
Since only #1 and #2 are equipped with serial links (SL), the other 'virtual links' are replaced by direct memory transfers (MT) to the destination SHARC.



(a) On the backplane, top view (b) On a SHARC cluster, back row

Figure 15: Direction of data flow

7.4 Runtime comparisons

The program was run on several PCs and the ER2 with the matrix dimension n varying from 75–200. The used computers and architectures were:

PC1 : Pentium MMX, 200MHz, 64MB RAM, SuSE Linux 8.0, Matrix Template Library (MTL), no compiler optimization switches

PC2 : AMD K6-III, 400MHz, 256 MB RAM, SuSE Linux 9.0, Matrix Template Library (MTL), no compiler optimization switches

PC3 : Athlon XP 2500+ (Barton), 1.83GHz, 1GB RAM, SuSE Linux 9.0, Matrix Template Library (MTL), used g++ switches: `-O3 -march=athlon-xp`

4C : ER2, unidirectional ring of 4 SHARC clusters ≡ 16 processors

8C : ER2, unidirectional ring of 8 SHARC clusters ≡ 32 processors

The times `CT` (computation time) and `TT` (total time, including transfer of data and computation) are given in seconds. `S` is the achieved speedup, relative to `PC1`.

n	Conf.	CT	TT	S
75	PC1	0.279704		1.0
	4C	0.034049	15.972000	8.2

continued on next page

continued from previous page

n	Conf.	CT	TT	S
	8C	0.030853	16.087000	9.1
	PC2	0.095601		2.9
	PC3	0.000247		1132.4
100	PC1	0.687065		1.0
	4C	0.056327	28.273600	12.2
	8C	0.048862	28.392400	14.1
	PC2	0.223540		3.1
	PC3	0.000514		1336.7
125	PC1	1.296860		1.0
	4C	0.080189	44.186500	16.2
	8C	0.059463	44.165600	21.8
	PC2	0.436092		3.0
	PC3	0.000939		1381.1
150	PC1	2.373250		1.0
	4C	0.161830	86.197600	14.7
	8C	0.087019	63.488700	27.3
	PC2	0.756340		3.1
	PC3	0.001545		1536.1
175	PC1	3.903350		1.0
	8C	0.121883	86.287100	32.0
	PC2	1.208131		3.2
	PC3	0.002390		1633.2
200	PC1	6.148070		1.0
	8C	0.160046	112.635000	38.4
	PC2	1.798086		3.4
	PC3	0.003805		1615.8

The *Cholesky* ring with four SHARC clusters (4C) could only be tested up to the matrix dimension $n = 150$. For higher values the electrical problems prevented the program from finishing successfully.
Figure 16 shows the speedups of the single test configurations against the slowest competitor PC1. Regard, that PC3 (Athlon XP, 1.8GHz) is not displayed because it outdoes all other machines by far and would distort the graphic too much.
Compared to the older Pentium and AMD processors, the ER2 is able to keep pace with their performance. But in relation to newer PCs, the ER2 is definitely not *state of the art* anymore, regarding its computational power. Even worse are the long times needed for data transfer from and to the SHARCs via the host adapter.

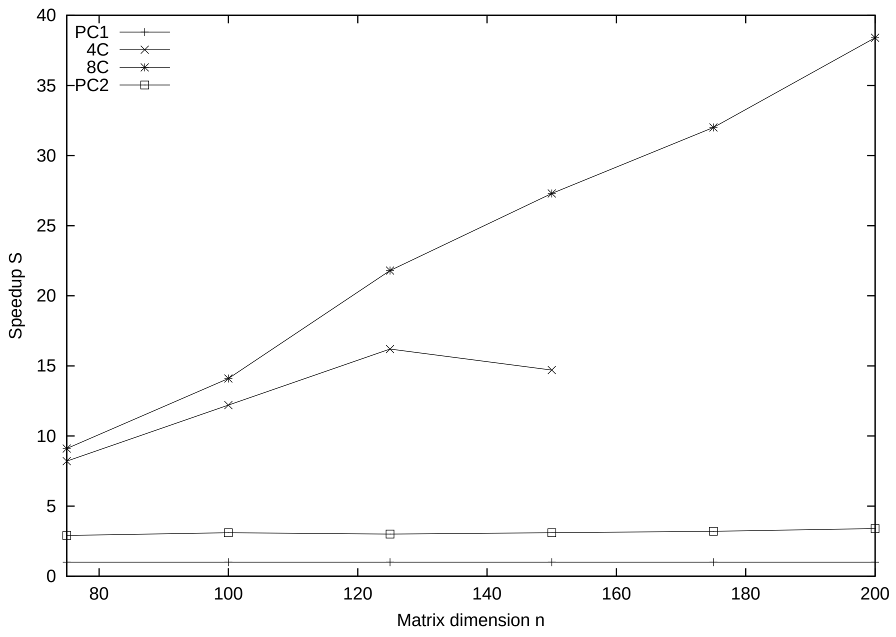


Figure 16: Speedups, relative to PC1

8 Conclusion

In this work a set of tools and utilities has been presented that should give rise to further development and usage of the ER2. A Linux device driver and two supporting C libraries have been developed. The *Parallel Runtime System* of the 2181 modules got complemented by a set of routines, the so called *Additional Runtime System*. A C compiler, assembler and linker for the SHARCs have been adapted to the ER2 and an own archiver utility was added. It was used to create a Standard C and Math library for the compiler `g21k`.
All together, a layered software structure–depicted in figure 17—is now offered to the user.
It enables working with the ER2 by programming in the high-level language C and lends itself to improvements like:

- Speeding up data transfers between the ER2 and the host PC by adding support for the SHARC-PCI interface, presented in [12].
- Integrating the tool `cnct`, which was developed in [20] for embedding graphs to the crossbar network of the ER2 automatically.

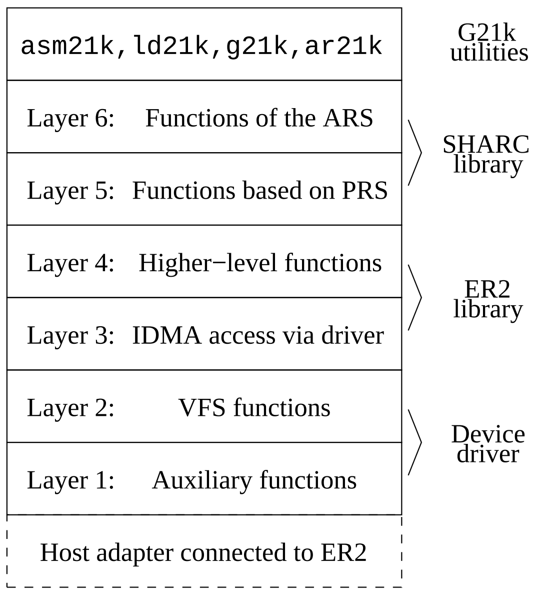

asm21k,ld21k,g21k,ar21k
G21k utilities
Layer 6: Functions of the ARS
Layer 5: Functions based on PRS
SHARC library
Layer 4: Higher−level functions
Layer 3: IDMA access via driver
ER2 library
Layer 2: VFS functions
Layer 1: Auxiliary functions
Device driver
Host adapter connected to ER2

Figure 17: Resulting software layers

- Providing a higher-level group management, where ER2 nodes may belong to several groups simultaneously.

Additionally, some features for the G21k utils are still missing:

- Adding the insertion and removal of object files to the archiver `ar21k`.
- Supporting more than one *Heap* in the C Runtime environment.
- Restructuring the mechanism for passing the *Heap* and *Stack* parameters from the *Linker Description Files* to the linker.
- Adding routines to the ARS library `libars.a`, for transferring `double` values from the SHARCs to the host PC and vice versa.

While the accompanying examples demonstrate that applications for the ER2 can be built quite easily now, the tests in section 7.3.1 and the runtime results manifest the weaknesses of this parallel computer. Thus, its near future—regarding the range of application—remains unclear.
Reducing the amount of data going in and out, by picking appropriate problems and algorithms, appears to be feasible. Another plan would be to concentrate on *network reconfiguration* issues or research in the area of *packet*

routing and *message passing*, respectively. However, all of these options require to get a grip on the electrical problems and intricacies of the ER2 as soon as possible.

References

[1] Analog Devices. *ADSP-2100 Family User's Manual*, third edition, 1995.

[2] Analog Devices. *ADSP-2106x Family User's Manual*, third edition, 1999.

[3] Dirk Bächle. *Ermittlung der Datentransferraten für den alten und neuen Host-Adapter des ER2*, 2000. Internal report.

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

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

[6] Dirk Bächle. *Reference to the Standard C, Math and ARS Library for the SHARC Compiler* `g21k`, 2003. Internal report.

[7] Dirk Bächle. *A SHARC Library in C. Functions for Accessing the SHARC Modules on the ER2*, 2003. Internal report.

[8] Bruce Eckel. *Thinking in Java.* Prentice-Hall, second edition, 2000.

[9] Gene H. Golub and Charles F. van Loan. *Matrix Computations.* Johns Hopkins University Press, third edition, 1996.

[10] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms.* SIAM, Philadelphia, 1996.

[11] Wolfgang Mackens and Heinrich Voß. *Mathematik I für Studierende der Ingenieurwissenschaften.* HECO, Alsdorf, 1993.

[12] Meder Mamutoff. *Softwarearchitecture for Transparent Integration of PC-Periphery for a Parallel Computer System via a Fast Serial Interface*, 2002. Studienarbeit.

[13] Georg-Friedrich Mayer-Lindenberg. *Das Fifth-Programmiersystem für den IBM-PC und kompatible Rechner*, 1990. Internal report, v1.1.90.

[14] Georg-Friedrich Mayer-Lindenberg. *Dokumentation zur Fifth-Implementierung auf dem ADSP 2181*, 1996. Internal report.

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

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

[17] Peter Jay Salzman and Ori Pomerantz. *Linux Kernel Module Programming Guide*, 2003. Version 2.4.0, (This document is available at `http://tldp.org/LDP/lkmpg/lkmpg.pdf`).

[18] Arnd Seeger. *Schematic of the ER2 Backplane*, 1996. Internal document "er2_mo7".

[19] M. Torabi. Decomposed Block Cholesky Factorization in the Karmarkar Algorithm. In Ervin Y. Rodin, editor, *Computers and Mathematics with Applications*, volume 20, pages 1–7. Pergamon Press, 1990.

[20] Otto Wohlmuth. *Konzepte zur Konstruktion und anwendungsspezifischen Konfiguration von Prozessornetzwerken.* PhD thesis, Technische Universität Hamburg-Harburg, 2000.

Appendix A: G21k C library functions

This is a short list of all available functions, contained in the provided *Standard C Library* `libc.a`, the *Math Library* `libm.a` and the *ARS Library* `libars.a`.
Please, refer to [6] for a complete reference to description, syntax and needed headers of the single C routines.

The library `libc.a` offers the functions:

> `abs`, `atexit`, `atof`, `atoff`, `atoi`, `atol`, `avg`, `bsearch`, `calloc`, `clip`, `div`, `exit`, `free`, `fmaxf`, `fminf`, `fsign`, `fsignf`, `getenv`, `idle`, `interrupt`, `isalnum`, `isalpha`, `iscntrl`, `isdigit`, `isgraph`, `islower`, `isprint`, `ispunct`, `isspace`, `isupper`, `isxdigit`, `labs`, `ldiv`, `lmax`, `lmin`, `malloc`, `max`, `memchr`, `memcmp`, `memcpy`, `memmove`, `memset`, `min`, `poll_flag_in`, `qsort`, `raise`, `rand`, `realloc`, `set_flag`, `srand`, `sign`, `signal`, `strcat`, `strchr`, `strcmp`, `strcpy`, `strcspn`, `strlen`, `strncat`, `strncmp`, `strncpy`, `strpbrk`, `strspn`, `strstr`, `strtod`, `strtodf`, `strtol`, `strtoul`, `strtok`, `tolower`, `toupper`

The library `libm.a` offers the functions:

> `acos`, `acosf`, `asin`, `asinf`, `atan`, `atanf`, `atan2`, `atan2f`, `ceil`, `ceilf`, `cos`, `cosf`, `cosh`, `coshf`, `exp`, `expf`, `fabs`, `fabsf`, `floor`, `floorf`, `fmod`, `fmodf`, `frexp`, `frexpf`, `isinf`, `isinff`, `isnan`, `isnanf`, `ldexp`, `ldexpf`, `log`, `log10`, `logf`, `log10f`, `modf`, `modff`, `pow`, `powf`, `sin`, `sinf`, `sinh`, `sinhf`, `sqrt`, `sqrtf`, `tan`, `tanf`, `tanh`, `tanhf`

The library `libars.a` offers the functions:

> `ars_barrier`, `ars_get_id`, `ars_signal`, `ars_wait`, `idma_read`, `idma_read_float`, `idma_write`, `idma_write_float`, `led_off`, `led_on`, `pack_float`

Appendix B: CD

This CD contains all data—i.e. source codes, examples, documentation and tools—related to this work. Each directory includes a HTML index file, so a common browser can be used to traverse the folder structure.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, daß ich meine Studienarbeit “Programming and Using a DSP-Based Parallel Computer under Linux” selbständig ohne fremde Hilfe angefertigt habe und daß ich alle von anderen Autoren wörtlich übernommenen Stellen, wie auch die sich an die Gedanken anderer Autoren eng anlehnenden Ausführungen meiner Arbeit besonders gekennzeichnet und die Quellen nach den mir vom Prüfungsamt angegebenen Richtlinien zitiert habe.

Hamburg, den 09.12.2003

(Unterschrift)