Single-Chip-Linux-Computer for the HADES-experiment Data-Acquisition

Radoslaw Trebacz
Jagiellonian University in Cracow, ul. Golebia 24, 31-007 Cracow
rtrebacz@poczta.onet.pl

This document discusses the chip Etrax 100LX (which is planned to be used in HADES at GSI). This paper described the cross-compiler and the compile process on example "Hello World" application. Next are described a client-server application with the performance of the UDP transport protocol, reading a external memory and access extenal devices via linux-modules.

1 Introduction

The **H**igh-**A**cceptance **D**i-**E**lectron **S**pectometer (HADES) is a detector system for lepton pair spectroscopy. HADES is planning an extenstion. A new subdetector is beeing built and one needs a DAQ for that. The idea is: using the "cheap" Ethernet as data-transport-medium and doing the TDC-conversion and data-transport at the detector. Boards are currently built[1].

2 Description of ETRAX 100LX & AXIS 82 DEVELOPER BOARD

Fig. 1: *Axis 82 - the developer board for ETRAX 100LX MCM 2+8*

The ETRAX 100LX Multi Chip Module (MCM) is using High Density Packaging (HDP) technology. HDP enables the integration of naked dies (i.e. chips without their capsules) and other components (like resistors). Thanks to that one gets smaller and more cost-effective products.
The MCM integrates an ETRAX 100LX, 2MB Flash, 8 MB SDRAM and many more components. It is almost a complete (capable of running embedded-Linux) computer on a single chip. The AXIS 82 DEVELOPER BOARD is the development platform for the ETRAX 100LX MCM chip. It's equipped with external connectors for dual 100 MBit Ethernet, USB, COM1, COM2, and RS-485 connection. This board is equipped with additional 4 Mbytes Flash and 8 Mbytes SDRAM.

3 How to Install the Developer Board 82 Software[2]

One has to download the cris compiler tools from `ftp://ftp.axis.com/pub/axis/tools/cris/compiler-kit/` or `http://developer.axis.com/download/compiler/`. One should now enter to the home directory and move all downloaded packeges to this directory and install cris compiler by typing:

```
tar -zxvf cris-dist-x.y.tar.gz
cd cris-dist-x.y
tar -zxvf ../cris-dist-glibc-x.y.tar.gz
tar -zxvf ../cris-dist-linux-headers-x.y.tar.gz
./install-cris-tools
```

When a question about a directory is asked, one should point at home directory.

This software does not include the Linux kernel. The file `linux-2.4.20.tar.gz` should be downloaded from `http://www.kernel.org/pub/linux/kernel/v2.4/` or `ftp://ftp.kernel.org/pub/linux/kernel/v2.4/`. Then one has to download the file `devboard_82-Rx_yz.tgz` from the directory http://developer.axis.com/download/devboard_82/latest/. One can move these files to for exapmle to home directory.
Next a some variables have to set by the script`init_env` in `~axis/devboard_82/` :

```
cd ~axis/devboard_82
. ./init_env or
source ./init_env
```

Next the downloaded tgz-file have to be ungzip and untar in the home directory: `cd ~`

```
tar -zxvf ~devboard_82-Rx_y_z.tgz
```

Contests of file axis/devboard_82/README:
Developer Board 82 Software version 1.92
This product uses Linux 2.4.26, but needs some of the functionality of future versions of Linux (e.g. changes not yet committed to arch/cris in the official Linux source code tree). Therefore a patch has been made to add this functionality.
Run the install script (./install) to patch the kernel source and build the software.
The installation script needs to know where is the source code for it. One has to ungzip and untar it in the directory axis/devboard_82/os:

```
cd axis/devboard_82/os
tar -zxvf /path/to/linux-2.4.20.tar.gz
```

If the source already exists in for example /usr/src/linux, one can leave it there and tell the install script where it is (one will be asked). The install script will then create a symbolic link (axis/devboard_82/os/linux) to the directory of existing Linux 2.4.20 source code.
The install script should be run in the`~axis/devboard_82/` directory:

```
cd axis/devboard_82
./install
```

One has installed the full developer board software in a directory called axis/devboard_82/.

4 Embedded system & Cross-Compiler

This chapter provides information about embedded and cross-platform software development where the host system on which the software is developed and the target system on which the software runs are different. The issues in this type of development are slightly different than a non-cross-environment although basic programming methods and techniques remain the same.
An embedded system is a special-purpose computer system that is used for a particular task. There is no need to add new applications to the system. Therefore programs and operating system are loaded only once into read-only memory. There is possibility to put files into Developer Board without rebuilding the whole Operating System. Under /mnt/flash is mounted flash memory, where we can put new applications, file etc.
A cross compiler is a compiler that creates executable code for a machine which is different than the machine on which the compiler is running. The host machine is the machine on which one can write and compile programs. All of the development tools are installed on this machine. When the code is rewarding, the executables can be build and transferred to the target machine. If the host and target machines are different processors, the final executable cannot be executed on the host machine where it was built. When one writes applications for a target machine there must be the following steps:

- Writing code
- Cross compilation
- Transferring binaries to the target
- Debugging

5 How to Write and Build Applications-"Hello World"

Here there will be explanation how to write a simple application that writes "Hello World" to standard output and run it on the developer board.
First one has to create a subdirectory in the ~axis/devboard_82/apps/[1]

```
cd ~axis/devboard_82/apps
mkdir hello
```

Next the source code should be written and saved to a file called

[1] directory, where the development software is installed. One can download this software from http://www.axis.com

~axis/devboard_82/apps/hello/hello.c.
There are only a few lines in the application:

```
#include < stdio.h >
int main(int argc, char *argv[]){
  printf("Hello World \n");
  return 0;
}
```

One can see that the code above is the same as for classic a non-cross-compiler.
One can write Makefile and save it in the ~axis/devboard*/apps/hello directory (note: make requires tabs instead of spaces in some places):

```
AXIS_USABLE_LIBS = GLIBC UCLIBC
include $(AXIS_TOP_DIR)/tools/build/
Rules.axis
OBJ=hello
$(OBJ):
  $(CC) $(LDFLAGS) -lcrypt $@.c -o
$@
all:  $(OBJ)
clean:
  rm -f $(OBJ) *.o core
```

The variable OBJ is the name of file (without .c), in which is a source code.

To build the executable file for the developer board one first need to set some additional environment variables by run the init_env script that should exist in the product directory of the source code tree:

```
cd ~axis/devboard*/
.  ./init_env or
source ./init_env
```

.target-makefrag A file named `.target-makefrag` is created in this directory, where the script will be made by typing:
`make cris-axis-linux-gnu`
For example, if one will type `make cris-axis-linux-gnu` in the `~axis/devboard*/apps/hello/`, in this directory will be created the file `.target-makefrag` Depending on the contents of this file .target-makefrag (there are a many files in the developer system, which are so called, but the proper file is in the same directory as the Makefile, which is using by the building the exectutable file) the executable will be built for the developer board or host operating system. When one creates the file .target-makefrag, there is "AXIS_BUILDTYPE=host " inside. In this case the executable for the developer board can be built.

To prepare for building the application for PC (host) one has to type:

```
cd ~axis/devboard*/apps/hello
make host
```

To change this for building the application for Etrax:
`make elinux`
Now the hello executable can be built using make:
`make clean && make`
If no errors occur an executable named hello will be created.

6 Getting the Executable into the Developer Board[3]

There are a few ways of getting the executable into the developer board:
•Using the top-level Makefile1 to build a new ROM file system for the developer board containing the executable. •Using the top-level Makefile2 to build a new ROM file system for the developer board containing the executable.
•Using ftp to transfer the executable to the flash file system (/mnt/flash) of the developer board.

6.1 Using the top-level Makefile1

To add an application put the source code directory in the devboard_82/apps directory and add it to the "SUBDIRS" section in the Makefile. Then build it like this:

```
cd ~axis/devboard_82
make cris-axis-linux-gnu
make install
make images
./kernel
```

(set the board in boot mode=¿see below)
Then the executable file will be appear in the /bin/ directory on Etrax.

6.2 Using the top-level Makefile2

There are two ways: •In the directory `~axis/devboard*/apps/Hello/` the script have to be run by typing:

```
 cd ~axis/devboard*/apps/Hello/
 make install
```


•Instead above-mentioned one can just copy the executable file manually somewhere into the target directory in ~axis/devboard*/target/cris-axis-linux-gnu/., which will be used as the root directory of the compressed ROM file system image.

After this the images have to built:

```
cd ~axis/devboard*/
make images
```

The executable will then be installed in ~axis/devboard*/target/cris-axis-linux-gnu/bin.
`./kernel`
(set the board in boot mode)
Then the executable file will be appear in the analog directory as the directory, where one puts the executable file(see few line above) or in the /bin/ directory on Etrax.

6.3 setting the board in boot mode

When the `make images` is done new images have been created. To test the new ROM filesystem first put your developer board in boot mode by pressing the boot button, pressing and releasing the reset button and then switching off and switching on the power (i.e. the boot button must be kept down while resetting the developer board). Then run the script ktest [2]:
`cd ~axis/devboard*`
(set the board in boot mode)
`./ktest`
When ktest is executed the image file ~axis/devboard*/kimage containing the kernel and the ROM file system (created by the toplevel-Makefile) will be sent to RAM on the developer board by the ETRAX 100 bootloader(etrax100boot) and the board will execute it. Everything sent to the developer board using ktest is stored in RAM on the board and therefore lost upon reboot. To make the changes permanent use ./flashit instead of ./ktest. This will send the image file ~axis/devboard*/fimage containing the kernel, the ROM file system and the flash file system to the RAM on the developer board and the board will then write it to flash before execute it.[3] Note that the flash file system of the board is then rewritten including configuration files in the /etc directory, which will result in the network settings of the board being replaced with those in ~axis/devboard*/packages/initscripts/network. One can edit this file and run make install and make images again from the devboard* directory to make the flash file system image contain network settings. When the board is up and running one can telnet to it and execute the application by typing hello in the shell of the developer board.

[2] from Documentation[3]:this requires that the developer board is on the same network segment as your computer, otherwise use FTP

6.4 Using FTP

When using FTP for transfering files to the developer board, one have to put them in the flash file system, because the ROM file system cannot be written. The flash file system is mounted at /mnt/flash on the board. Since transferring a file using FTP will not make the file executable one has to set the execute bits manually using the chmod command from the FTP client. Make sure you transfer the file in binary mode (using `bin` or `binary` command). Example:

```
ftp hades18
Connected to hades18
Name:  root Password:  ***
ftp> cd /mnt/flash
ftp> bin
ftp> put hello
ftp> chmod 755 hello
ftp> quit
```

7 How to Use the Top-Level Makefile(Axis 82)[4]

The working directory should be changed to the product directory in your source code tree:

FIRST STEP:
Set variables by typing:
`cd ~axis/devboard_82/`

[3] from Documentation[3]: If your computer is not using the eth0 interface to connect with the network where the developer board is, you must specify which network interface to use with ktest (or flashit). You do this by adding the -d option in the kflash (or ktest) script. For example:
`cd ~axis/devboard*`
(set the board in boot mode)
`./flashit -d eth1`

```
.  init_env
```

From documentation [4]. I did not use that[4] *CHOOSE WHAT DO YOU WANT TO DO*:

Reconfiguring the Linux kernel
The Linux kernel is configured using the menuconfig or xconfig targets of the Makefile in the Linux source code directory:

```
cd ~/axis/devboard_82/os/linux
make xconfig or make menuconfig
```

For example the option `Loadable module support ⇒ Set version information on all module symbols` should be unmarked. In this case we can use the name of kernel function without the special ending, for example: one can use printk instead printk_cc0327151. There are kernel function's names in /proc/ksyms directory on the developer board. Then use the top-level Makefile to build the kernel and images:

```
cd ~/axis/devboard_82
make kernel
make images
```

Modifying the Linux kernel When one has modified source code in the Linux kernel that does not require reconfiguring the kernel one just has to rebuild it like this:

```
cd ~/axis/devboard_82
make kernel
make images
```

Adding an application In order to add an application one has to put the source code directory(for example "hello") in the devboard_82/apps directory and add it to the "SUBDIRS" section in the Makefile(which is in ~/axis/devboard_82) (see How to Write and Build Applications for Linux for details). Then build it like this:

```
cd ~/axis/devboard_82
make cris-axis-linux-gnu
make install
make images
```

Modifying an application If the source of an application (for example devboard_82/apps/hello) was modified one needs to install that application separately before building new images:

```
cd ~/axis/devboard_82
.  ./init_env  cd ~/axis/devboard_82/hello
make install
```

Then the top-level Makefile will be using to build the kernel and images: `cd ~/axis/devboard_82`

```
make images
```

Removing an application In order to remove an application, one should just remove it (or comment it out) from the Makefile file and run:

```
cd ~/axis/devboard_82
make install
make images
```

Adding a file In order to add a single file it has to be added in the proper directory in the devboard_82/files directory and add it to the Makefile file (not in the subdirs section). If one, for example, wants to add the file file.html to the directory /usr/html on the developer board, it has to be put in the directory /axis/devboard_82/files/usr/html and the following line has to be added under the "LOCALS" section of the Makefile file:

```
files/usr/html/file.html 0644
```

The option 0644 means file access permission 0644 (i.e. rw-r–r–). Then run:

```
cd ~/axis/devboard_82
make files
make images
```

``make install'' can be run instead of make files but it is not necessary (make install also does a make files among other things). **Modifying a file** If a some file was modified under the devboard_82/files directory, then the following steps should be run: `cd ~/axis/devboard_82`

```
make files
make images
```

Removing a file If one wants remove a file under the devboard_82/files directory and/or in the Makefile file, one cannot use the files target because it just adds files. The install target has to be used instead:

```
cd ~/axis/devboard_82
make install
make images
```

The install target removes the entire target/cris-axis-linux-gnu directory before installing everything listed in the Makefile again. *WHAT IS NEXT?* All of the tasks above result in two images: kimage and fimage. The kimage contains the linux kernel and the read-only root partition (cramfs). The fimage contains a complete flash image (rescue partition, partition table, decompressor, compressed kernel, read-only root partition and JFFS partition). They can be transfered to the board using one of four scripts:

- ktest: network boots the kimage from RAM.
- kflash: writes the kernel and read-only root partition to flash.
- flashit: writes all of the fimage except the rescue partition to flash.
- flashitall: writes the entire fimage to flash.

Before any of these scripts will be run, first devboard should be put in boot mode by pressing the boot button, pressing and releasing the reset button and then releasing the boot button (i.e. the boot button must be kept down while resetting the developer board). The ktest and flashit scripts are the most useful. The rescue partition contains the serial number which is used as the ethernet address, so you probably do not want to overwrite that, i.e. don't use the flashitall script unless you really want to overwrite the rescue partition. A new kernel can be tested by typing:

```
cd ~/axis/devboard_82
(set the board in boot mode)
./ktest
```

A new fimage (except the rescue partition) could be flash by typing:

```
cd ~/axis/devboard_82
(set the board in boot mode)
./flashit
```

If ethernet interface (on PC) is not eth0 the -d option should be used to the scripts. For example flashit should be run on eth1 like this:

```
cd ~/axis/devboard_82
(set the board in boot mode)
./flashit -d eth1
```

[4] The first thing that needs to be built are the applications in the tools/build directory. These are used through out the reset of the installation. Simply type:

```
cd ~/axis/devboard_82
make tools
```


8 Client-Server:see source code AxisServer.c, DEPC152Client.c

This part explains how to write a simple client/server system in Linux using UDP. Client/server describes the relationship between two computer programs in which one program, the client, makes a service request to another program, the server, which fulfills the request. Now there will be presented what one should do to get our system to work.

CLIENT:

```
int sd=socket(PF_INET,SOCK_DGRAM,0);
```

Here we create a socket(a communication end point). PF_INET says that it will be an internal socket

int socket(int domain, int type, int protocol);

domain:

AF_UNIX – file system

AF_INET – internet address

type:

SOCK_STREAM – reliable connect-oriented, byte stream TCP

SOCK_DGRAM – unreliable connectionless UDP

protocol: 0, non-zero for a specific protocol

```
struct sockaddr_in caddr;
caddr.sin_family = PF_INET;
caddr.sin_addr.s_addr = INADDR_ANY;
caddr.sin_port=htons(CLIENTPORT);
```

Protocol family is chosen as PF_INET (Internet IP Protocol). IP address is chosen as to be INADDR_ANY instead of a specific address, as Client will accept packets from any interface/computers. One sets the port using htons() to convert value from host byte order to network byte order.

```
bind(sd,(struct sockaddr *)&caddr, sizeof(caddr));
```

This line tells the system that the socket sd should be bound to the address caddr.

bind: attach an address to a socket

int bind(int socket, const struct sockaddr *address, size_t address_len)

```
struct sockaddr_in *sent;
sent = gethostbyname(ETRAX's address);
```

This function returns the IP address corresponding to the machine where the server runs.

```
struct sockaddr_in saddr;
saddr.sin_family = PF_INET;
saddr.sin_port = htons(SRVPORT);
bcopy(sent→h_addr, (char*) &saddr.sin_addr, sent→h_lenght);
```

Analog as above. Function bcopy() copy from sent→h_add to saddr.sin_addr.

```
recvfrom(sd,SizeNumber,2*sizeof(int), 0,(struct sockaddr *)&caddr,&len);
for(i=0; i< SizeNumber[1]; i++){
  recvfrom(sd,msg,SizeNumber[0],0, (struct sockaddr *)&caddr,&len); }
```

Recvfrom: receive data

ssize_t recvfrom(int sockfd, void *buff, size_t nbytes, int flags, struct sockaddr *from, socklen_t *addrlen);

One receives the information about the number of packages and on this base one constructs the loop, in which these packages are stored.

```
close(sd);
```

At the end socket is closed.

Server:

```
while(1){
  recvfrom(sd, buf, sizeof(buf),0,
(struct sockaddr *)&caddr, &len);
  sendto(sd,SizeNumber, 2*sizeof
(int),0,(struct sockaddr *)&caddr,
sizeof(caddr));
  for(i=0; i<SizeNumber[1] ; i++){
    sendto(sd,packet,sizetableofint
*sizeof(int),0,(struct sockaddr *)
&caddr, sizeof(caddr));
  }
}
```

Sendto: send data

ssize_t sendto(int sockfd, void *buff, size_t nbytes, int flags, const struct sockaddr *to, socklen_t addrlen);

Most of the things in server's code is the same like in the client's code. There is one difference: in server.s code data are sending instead of receiving. The server application goes into endless loop and wait for client's request. When it comes, the data about number of packages and these packages can be sent.

8.1 Performance

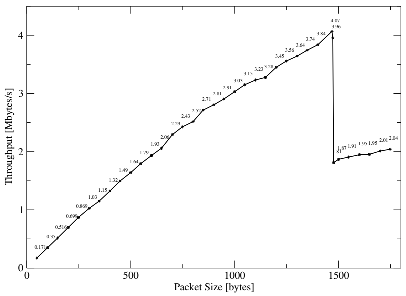

Fig. 2: *Performance of Etrax in UDP transfer*

Above applications were used to measure the speed of UDP connection to/from Etrax. 10MB of data was sent at different sizes of the UDP-packets. Figure 2 indicates that the best result – 4.07 MBytes/s one gets when the data are sent in the packages, in which the raw(without headers) data occupy 1468 bytes. When we increase the size of the package more than this value, the UDP-protocol splits the message into two smaller messages.

9 Reading Memory:see source code AxisServer.c, DEPC152Client.c

This section explains how one can read the external memory. There is external memory on the Developer-Board.

In a user application one can access memory through the pointer with the help of the following functions:

```
int fd = open(``/dev/ram0'',O_RDWR);
```

This is creating the file descriptor to the device "ram0", which represents external memory and which can be opened with read-write permission.

```
unsigned char *virt =
mmap((unsigned int*) ADDRESS,
SIZE, PROT_READ| PROT_WRITE|
PROT_EXEC, MAP_SHARED, fd, OFF-
SET);
```

mmap() allows the mapping of device memory directly into a user process's address space. There is mapping SIZE bytes starting at OFFSET from the file specified by the file descriptor fd into memory, preferably at address ADDRESS. Third parameter describes the desired memory protection. Pages may be read, written and executed. The fourth parameter specifies the type of the mapped object.

```
close(fd);
```

One can now close the file descriptor, because it is no longer needed.

```
unsigned char *cp = virt;
*cp = LetterToWrite;
printf(``Read letter::%c\n'',*cp);
```

Pointer can be used to read or write the memory unit. The type of pointer should be the same as the type of reading memory.

```
munmap(virt, SIZE);
```

The mappings for the specified address range has to be deleted before the end of program.

10 Device Driver

A device driver is a collection of subroutines and data within the kernel that constitutes the software interface to an I/O device. When the kernel recognizes that a particular action is required from the device, it calls the appropriate driver routine, which passes control from the user process to the driver routine. Control is returned to the user process when the driver routine has finished[5].

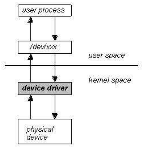

Fig. 3: *The relationship between device driver and the Linux system*

10.1 Hello World

Loadable Kernel Modules are code which can be dynamically inserted into the linux kernel without recompilation. LKM's are mostly used for device drivers.

To program a LKM to display greeting and good-bye the following code should by written:

```
#define __KERNEL__
#define MODULE
#include <linux/module.h>
int init_module(void){
printk("<1>Hello, world\n"); re-
turn 0; }
void cleanup_module(void){
printk("<1>Goodbye world\n"); }
```

First of all, it is needed to declare MODULE and __KERNEL__ to let the compiler know that a device driver should be implemented as linux module. The include's for an lkm are different from that of normal applications. They use the directory /usr/src/linux/include. Moreover, normal libc functions like printf, scanf..etc cannot be used. One has to restrict ourselves to the kernel API. A full list of the kernel functions can be found in /proc/ksyms. These functions(init_module & cleanup_module) have to exist. init_module will be started when one load the module into the kernel memory and cleanup_module will be started when the device driver will be remove from memory.

The init_module function is an entry point into the lkm. It executes when the lkm is first loaded into memory. This function is usually used to get arguments, register devices,protocols and other things which need to be done once.

The cleanup_module function is an exit point in the lkm. It executes when the lkm is unloaded from memory. Usually used to unregister all the things registered in init_module.

Next the module is loaded by 'insmod hello' and one can see that on the screen 'Hello World' appears.

One can check for the loaded module by lsmod or by /proc/modules.

The module can be unloaded by 'rmmod hello'. This will now call cleanup_module and then remove the module from memory.

10.2 Reading external memory

There are two types of devices. One of them is the block device, which one can access via the buffer cache and which may be randomly accessed, that is any block can be read or written no matter where it is on the device. Block devices can be accessed via their device special file(dev/xxx) but more commonly they are accessed via the filesystem(mount).

Each device is accessed through a file located in the /dev directory, which contains all device special files. There are the major and minor number for each file. The major number identifies the driver associated with the device. The minor number is for a driver to control several devices.

Device memory regions must be allocated prior to use by the following functions:

```
int check_mem_region(unsigned
long start, unsigned long len);
void request_mem_region(unsigned
long start, unsigned long len,
char *name);
void release_mem_region(unsigned
long start, unsigned long len);
```

check_mem_region may be called to see if a range of memory is available for allocation; request_mem_region will actually allocate the memory range; to release the memory, driver should call release_mem_region, when one is done with them.

References

[1] http://www.axis.com

[2] http://developer.axis.com/doc/software/devboard_82/install-howto.html

[3] http://developer.axis.com/doc/software/apps/apps-howto.html

[4] http://developer.axis.com/doc/software/devboard_82/makefile-howto.html

[5] Alessandro Rubini, Jonathan Corbet, Linux Device Drivers, 2nd Edition, O'Reilly, 2001