KUNGL TEKNISKA HÖGSKOLAN

# RadioNet Driver Implementation for the Mobile INTernet Router

## (Walkstation II project)

**Pascal GUERIN**

**EURECOM Institute**

**June 1994**

The Royal Institute of Technology
Departement of Teleinformatics / TSlab
Electrum 204
S - 164 40 KISTA
Sweden

Institut EURECOM
BP 193
F- 06904 Sophia Antipolis Cedex
FRANCE

# RadioNet Driver Implementation for the Mobile INTernet Router

## (Walkstation II project)

**Pascal GUERIN**
**EURECOM Institute**
**Ecole Nationale Supérieure des Télécommunications de Paris**
**(Telecom University of Paris)**

## January - June 1994

**Supervisors:**

**Prof. Gerald Q. Maguire**
The Royal Institute of Technology
Departement of Teleinformatics / TSlab
Electrum 204
S - 164 40 KISTA
Sweden

**Prof. Pierre Humblet**
Institut EURECOM
C.I.C.A
BP 194
F-06904 SOPHIA ANTIPOLIS Cédex

# Acknowledgements

I would like to express my gratitude to a number of people who have helped me during my work at KTH.

Special thanks go to the head of the Teleinformatics Department of the Royal Institute of Technology (KTH) Professor Björn Pehrson who accepted me as a thesis student in the Telcommunication Systems Laboratory (TSlab) group.

Much credit for this Master Thesis should go to my supervisor Professor Gerald Maguire. He answered many questions in the context of my work with an impressive competence. He gave me the opportunity to work with this cutting edge technology. He also gave me a good guidance and encouragement.

Many thanks to Frank Reichert for the documentation he gave me and for allowing me to include some of his figures into the current report.

I would like also to thank the computer system administrators Jim Svensson and Fredrik Ljungberg who answered my many questions and helped me to find solutions to problems concerning software development.

I would like to thank Daniel Kerek and Teet Saluvere for their cooperation concerning the radio part, the knowledge they taught me and their continous support.

I don't forget Magnus Danielson who worked with me during the two last months and generously offered me his precious help.

Thanks to all the members of the Teleinformatics Department who have welcomed me into their laboratories. Many thanks for the friendly human atmosphere they created for my working environment.

# Table of Contents

# Chapter 1: General Presentation

## 1.1. Introduction

Wireless communication has been developed for voice for many years now, with the implementation of many telephone cellular systems based on different standards (analog and digital), using different techniques of radio communication. Due to the decreasing size of handheld units and to the need of information exchange when moving, cordless telephony is encountering success even though the equipment prices and tariffs are still expensive for non-commercial use.

However, voice communication is not a sufficient means of information exchange because it does not offer enough performance for large data flow. As the wired telephone network has undergone an improvement by becoming partly wireless, it is now expected that the same will occur for computer networks. Thus this subject is being studied by many research centers and laboratories like the KTH in Stockholm, Sweden.

The purpose of the Walkstation II project currently being developed is to allow the connection of a computer to a wired network via a radio channel and also create a complete cellular system for interactive mobile multimedia applications. This connection gives the computer (and therefore the user) mobility. The TS lab, in collaboration with the Columbia University, has improved the IP protocol that supports such mobility, while remaining completely transparent to the user.

The Mobile INTernet router (MINT) is a core element in the Walkstation II network. It plays the role of a Base Station ensuring radio coverage for a whole cell. On one hand, it communicates with mobile hosts (isolated computer or LAN) via air, and on the other hand it is connected to a physical network. Its hardware was developed at Hewlett-Packard Labs, Palo Alto, USA.

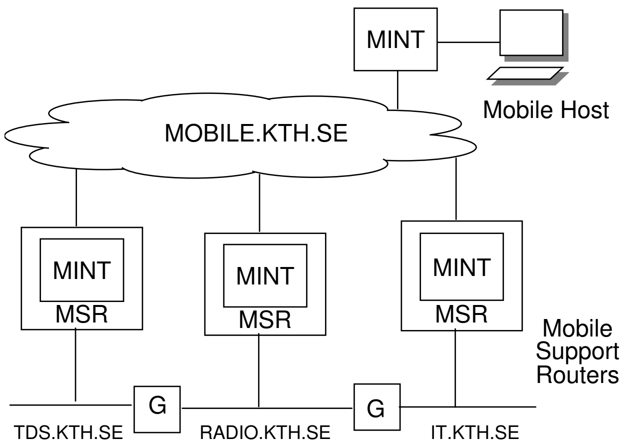



**Figure 1:** Mobile support routers implemented with MINTs

## 1.2. The subject of my thesis

The purpose of my work was to define and create a specific application intended to control one of the several components of a MINT router.

Owing to the fact that the MINT plays the role of a Base Station, it naturally contains a radio part (set of radio emitter and receiver) which has to be “driven” and controlled through many parameters such as the transmit power, the spread spectrum sequence received power (radio tranceiving is based on spread sprectrum) and some others less significant.

A driver looks like a several subroutines in C language, which is then compiled into an executable code for the MINT cpu, after that downloaded into the memory associated to that CPU before finally being run.

The new parts of this driver only deals with the previously noted parameters and does not handle packets from the network which have to be routed on the radio link. The later is the task of the Ethernet driver already done by Mr. Anders Klemets. Actually the RadioNet Driver directly derives from the Ethernet driver.

The project involves implementing such a driver and therefore to produce a complete program which is able to work with applications of higher layers such as the Network Management Application based on SNMP protocol.

All the protocols used in the global project were not yet defined. Concerning the radio part, that was the case and defining it in partnership with people who are responsible of the hardware was part of my task..

## 1.3. Work teams

As shown in the Figure 2 there are in fact three different labs which are involved in the conception of the MINT router.

The *Electronic System Design* (ESD) laboratory is in charge of doing the hardware (radio tranceiver and interface to MINT router). Now it is being subjected to tests in order to make the radio link reliable.

The *Radio Systems* (RS) laboratory is in charge of elaborating algorithms for the radio resource management (power control, channel allocation, acces control, ...). Its results will directly influence Markus Oelhafen’s SNMP manager in terms of what information it must store and control.

The *Telecommunication Systems* (TS) laboratory is in charge of producing all the software that will run on the MINT router. It concerns the SNMP manager which interacts with drivers of sublayers such as the radio device driver (my portion) and the Ethernet device driver (Anders Klemets). Tomoki Osawa is in charge of conceiving the CSMA/CA encoding process (since CSMA/CD is not possible on the radio link) which ensures reliabity in communication between several mobile hosts and base station using the same channel.

A real work of information exchanges between people was led in order to fit the different pieces of the whole project together.

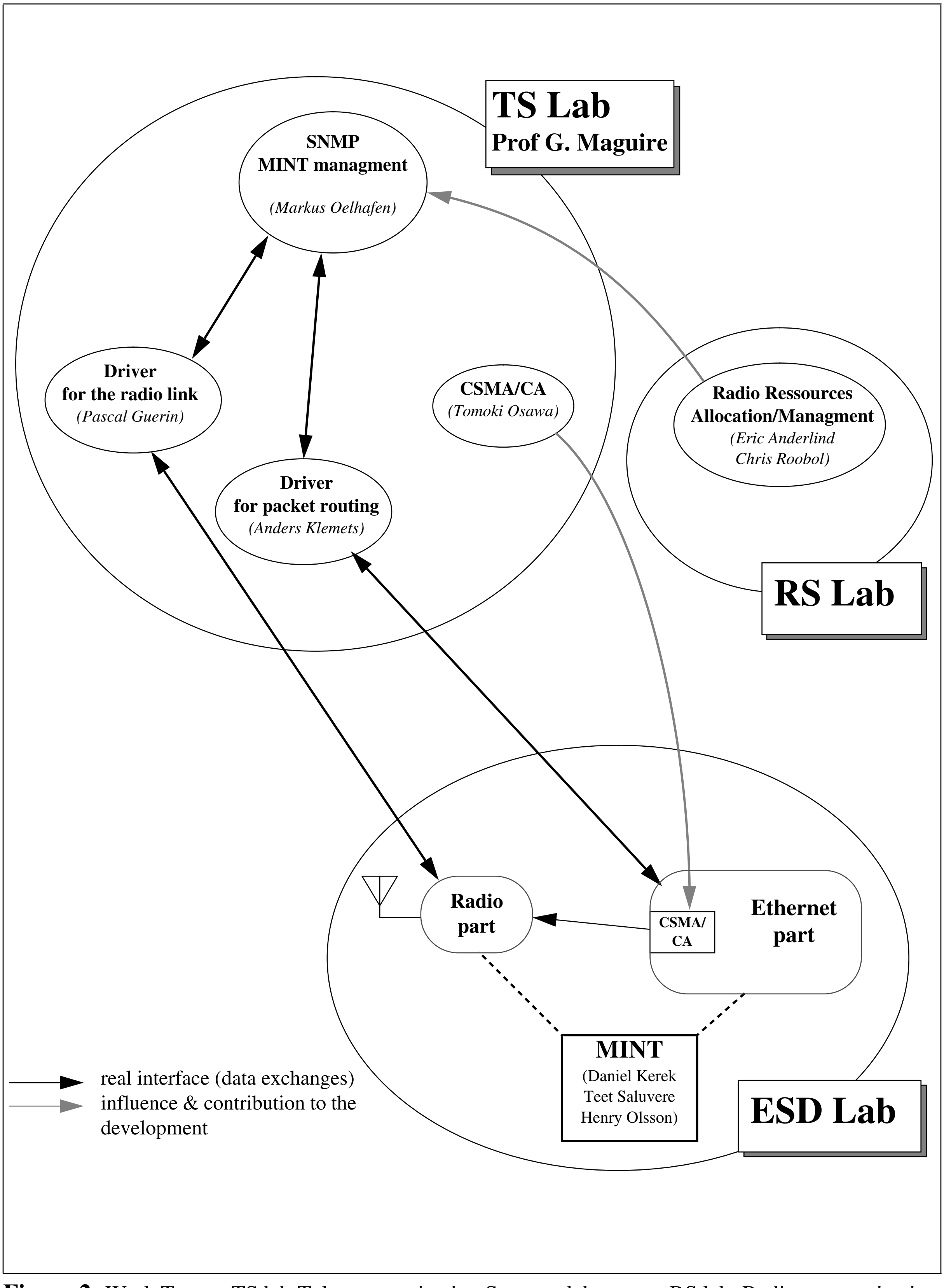



**Figure 2:** Work Teams: TS lab:Telecommunication Systems laboratory, RS lab: Radiocommunication Systems lab, ESD lab: Electronic System Design lab.

# Chapter 2: MINT's features

## 2.1. Introduction

This chapter discusses physical features of the MINT box. It shows its inner structure, compounded of a central unit and many communication devices around, and explains a little bit more about the Ethernet device, forecasting the modifications performed on the system to permit the connection of the radiopart.

## 2.2. Inside the MINT box

It is planned that the MINT router should fit into a small box, plugged on the back of the computer. Small means surface-mounted-device construction and lightwith enough to be used with a portable computer.

However, it is still at the prototype state which means a somewhat large size. The current prototype in illustrate in Figure 3.

The mother board is equiped with a MC68030, 1Mbytes of ROM, and 8 Megabytes of RAM which is addressed via 32 bits. As can be seen in the figure, the Mint box provides a number of different I/O connections allowing data exchanges and control over the device to be performed in a sophisticated way.

As implied by its name, the router obviously contains two LAN devices, one connected to the Ethernet, the other connected to the radio for routing packets to/from the radio link.

The radio has two connections associated with two interfaces: the former is called a *slow interface* (indicated by the (a)-arrow) and is used for the control and configuration of the radio device, i.e. it allows you to set or get some crucial parameters such as the key for spread spectrum encoding, the signal strength of each mobile host received, the total measured power, the power for transmission. The latter is called the *fast interface* (indicated by the (b)-arrow) and is connected to the second LAN interface which is used for routing packets on the radio link. This distinction is made related to the data rate. Indeed, the later in fact carries all the packets (indicated by the (c)-arrow) destinated to mobile hosts (lonely computer or isolated network). So the throughput may be great whereas on the slow interface the exchange of control parameters needs less flow.

The other interfaces (serial and parallel communications) are used now by the development toolkit.

## 2.3. Communication architecture

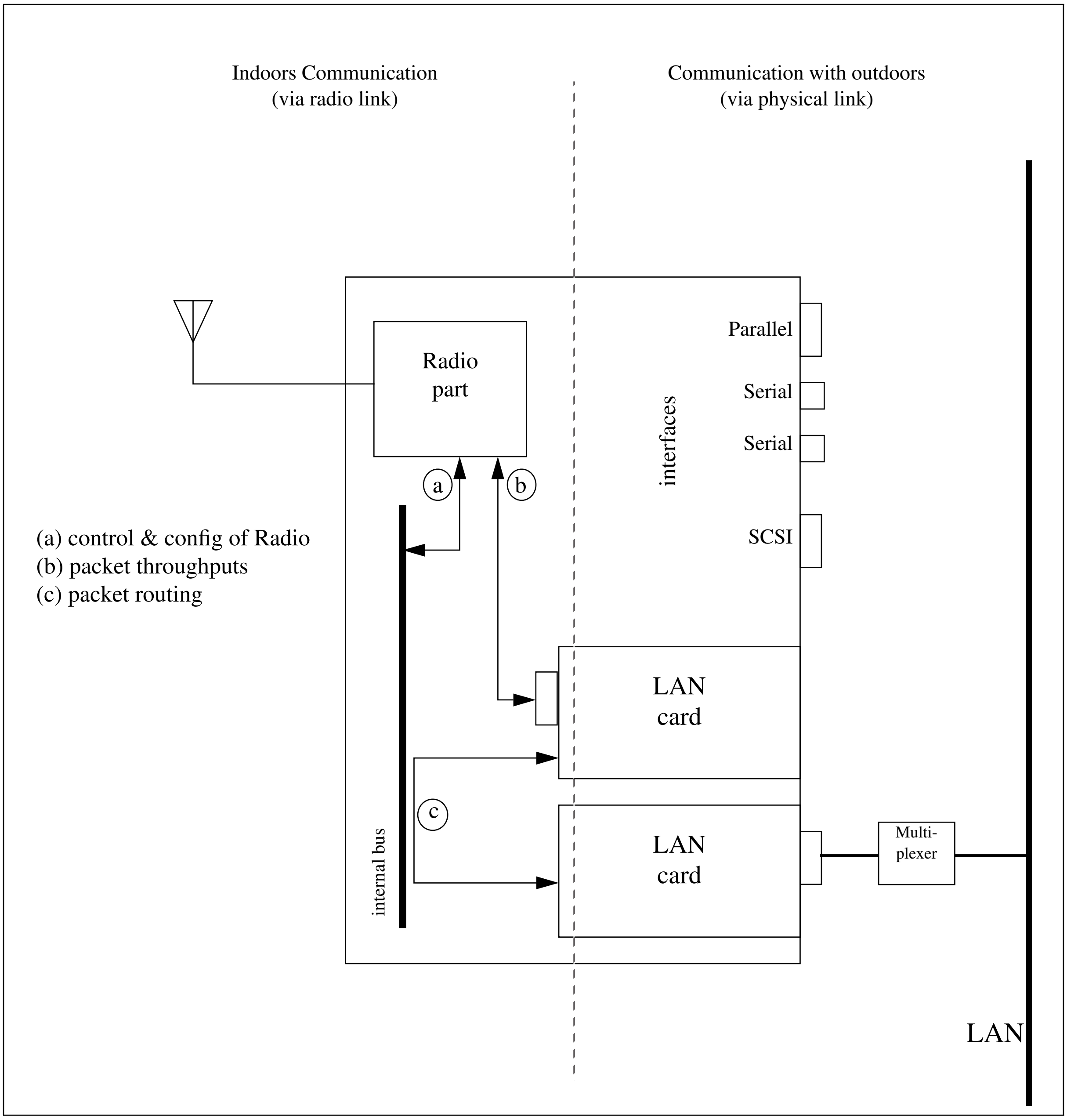
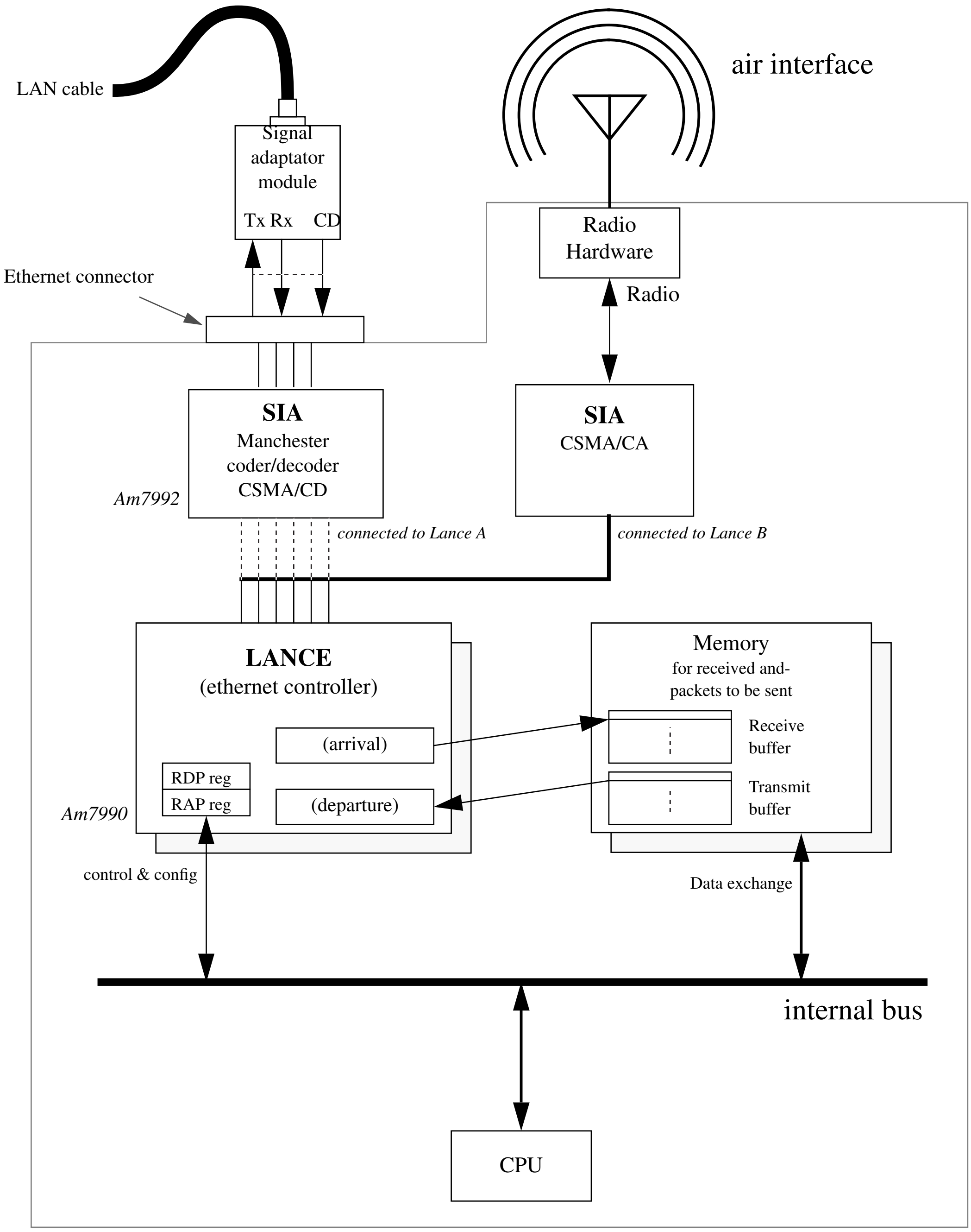



**Figure 3:** The MINT box

Now let us look more deeply into the inner structure of the communication device.

As is shown in Figure 4, there are two Local Area Network Controllers for Ethernet (LANCE) (one in background), one for ensuring communication with the physical local area network, the other for ensuring communication with mobile hosts (via the radio part).

Assuming Lance A is used for the LAN, this one is then followed by a Signal Interface Adaptator (SIA) which encodes data emitted from the Lance into adequat signals in respect to the protocol named Carrier Sense Multiple Access/Collision Detection (CSMA/CD). Collision Detection is feasible by matching the shape of the transmitted carrier and the shape of the carrier curently present on the line because the transmitted signal and the received one from another host have approximately the same




***LEGEND:***

LANCE: Local Area Network Controller for Ethernet
SIA: Serial Interface Adaptator
CSMA/CD: Carrier Sense Multiple Acces/Collision Detection
CSMA/CA: Carrier Sense Multiple Acces/Collision Avoidance

**Memory map:**

| | | |
|---|---|---|
| Lance A: | reg at location | 0xC0000300 |
| | buffer in RAM at | 0x80000000 |
| Lance B: | reg at location | 0xC0000400 |
| | buffer in RAM at | 0x81000000 |

RDP (reg data port): BASE+0x0
RAP (reg address port): BASE+0x2

**Figure 4:** MINT network communication

power level.

On a radio link, owing to the fact that frequency is a scarce resource, one channel is shared among all mobile hosts. The power of the transmitted carrier from your host is much higher than the received signal from another mobile host. Therefore Collision Detection is impossible to realize and so another way of coding packet has to be thought of.

Thus the second Lance (Lance B) is followed by a special SIA (now being developed) which performs the conversion with a new CSMA/CA protocol. CA means Collision Avoidance (low probability of collision).

Each Lance has its own memory location for buffering the packets to receive or to send.

The routing of the packet is the task of the Ethernet Driver (already developped by Anders Klemets) which runs on the CPU. So the control is performed both on the LANCEs (registers to initialize) and on the memory (decoding and encoding packets). The Radio Driver has no role to play in this context.

## 2.4. Memory map

As shown in Figure 5, the memory area is divided into several spaces.

The EPROM contains the code of the monitor along with Bootp and TFTP. When the MINT is switched on, after initializing the processor, the content of the EPROM is copied into the DRAM, from where code is thereafter run. The transfer from the EPROM to the RAM is purely for speed, because the path to the EPROM is 1 byte wide while the data path to memory is 4 bytes wide. The CPU clock runs in this version at 25MHz.

The D(ynamic)RAM is 8Mbytes in size (although an adaptator exists allowing multiple 8 Mbyte SIMMs to be inserted) and will normally be quite sufficient for storing all the future code necessary to make the MINT into a router.

Two buffers in S(tatic)RAM are provided for the LANCEs. They are used to store received packets and all data inherent to the functioning of these Ethernet controllers. DRAM cannot be used in place of SRAM, because the LANCE needs access to it very fast, for reading or writing packets, and the DRAM may not be available due to a refresh.

From 0xC0000000 to 0xC0000700, one finds several spaces reserved for the many devices present on the mother board, such as the LANCE controllers. The Radio Device located between 0xC0000600 and 0xC0000700 is currently being developed.

## 2.5. Some information about the digital transceiver

The modulation[7] is made using the CDMA method, using a 13bit key. As shown in Figure 6, a *Xilinx*[1] chip is in charge of the spread spectrum processing, and comes in connection just after the LANCE chip, in place of the traditional SIA[2]. It generates a clock pulse for the LANCE chip, while the packet is *serially* exchanged.

1. XILINX Inc. is currently the leader in Field Programmable Gate Array (FPGA) devices.
2. Signal Interface Adaptator (Am7992).

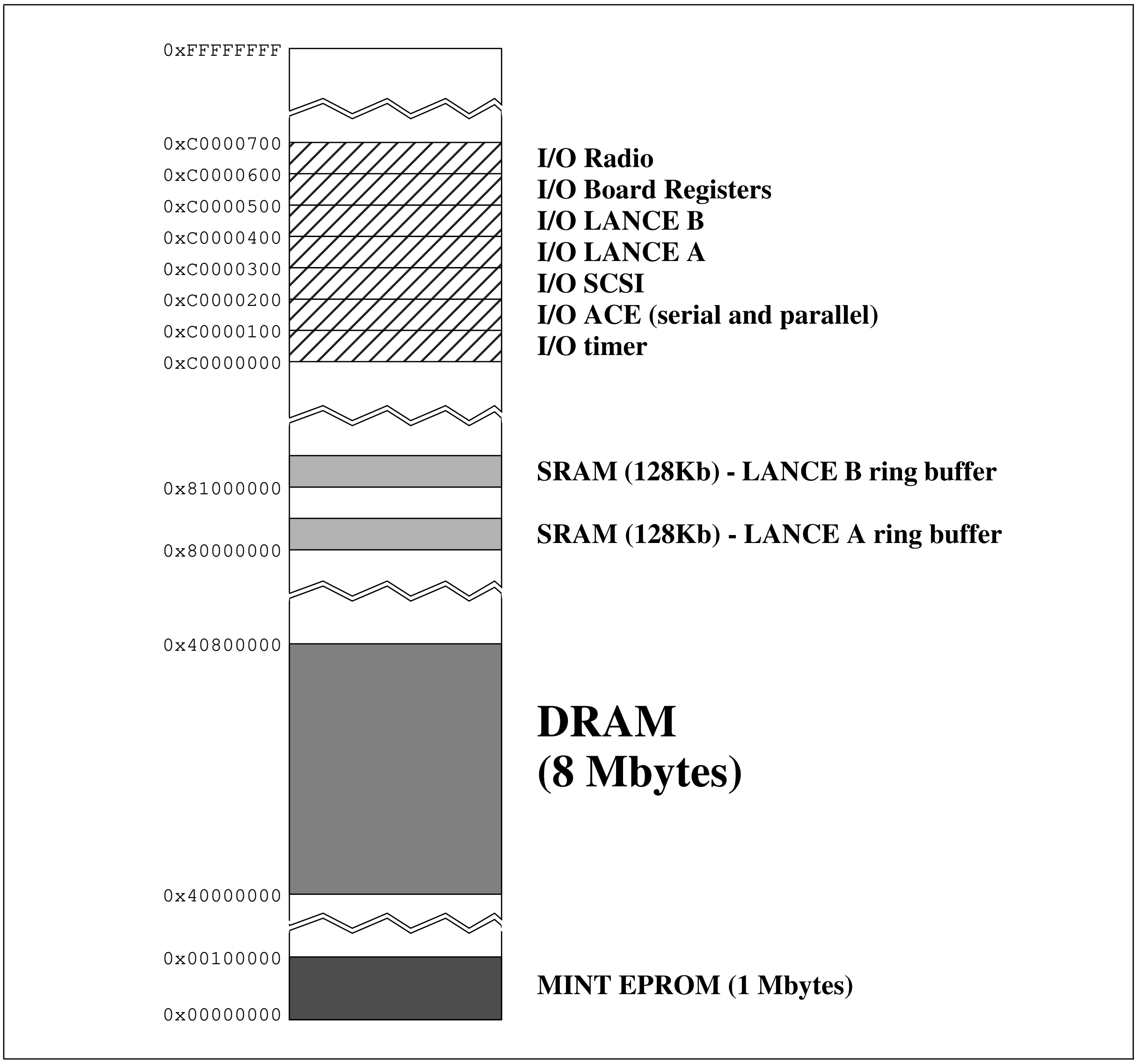



**Figure 5:** MINT memory map

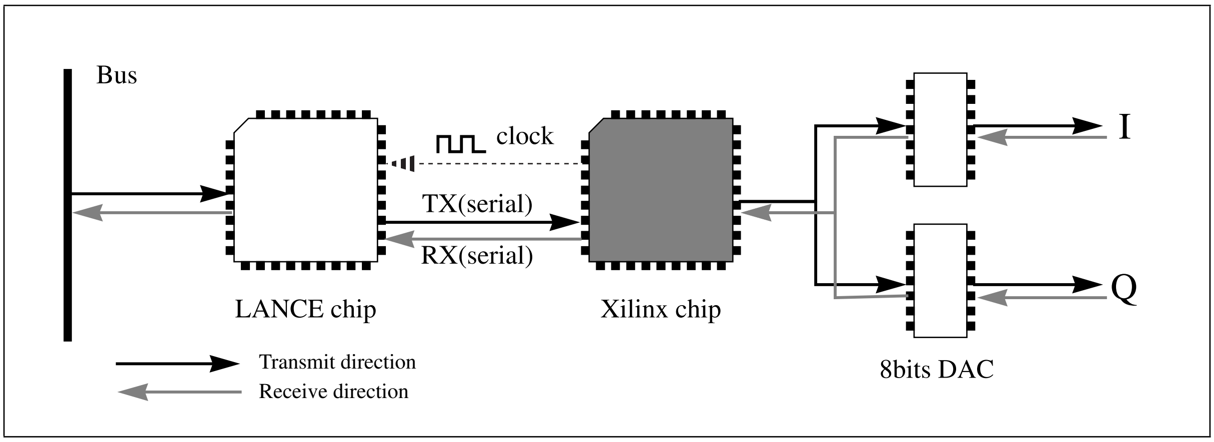



**Figure 6:** Rough scheme of the digital transmitter

In the transmitting case, at the Xilinx, when the packet has been completely memorized, bit after bit is then multiplied[3] by the key sequence (also called *Pseudo Random Noise*). In order to be able to modify the level of the modulating signal (I and Q) in a dynamic way, each bit of the resulting key sequence is translated into a byte, which value determines the amplitude of the analog signal produced after digital to analog conversion.

Two analog signals are thus generated. I is the base signal and is injected into the modulator. Q is the same signal, but is injected into another modulator using quadratic phase shifted carrier[4].

Concerning the demodulator (receiver stage), it is more complex. It would take too much time to explain it[5] and is not really the purpose of this paper. Here we merely state that the hardware the receiver has the same structure as the transmitter.

3. muplication factor equals 1 when bit=1, equals -1 when bit=0.
4. Please refer to Teet Saluvere (ESDlab) Master's Thesis for futher details.
5. Please refer to Teet Saluvere (ESDlab) Master's Thesis for futher details.

# Chapter 3: Working Configuration and Development Tools

## 3.1. The MINT family

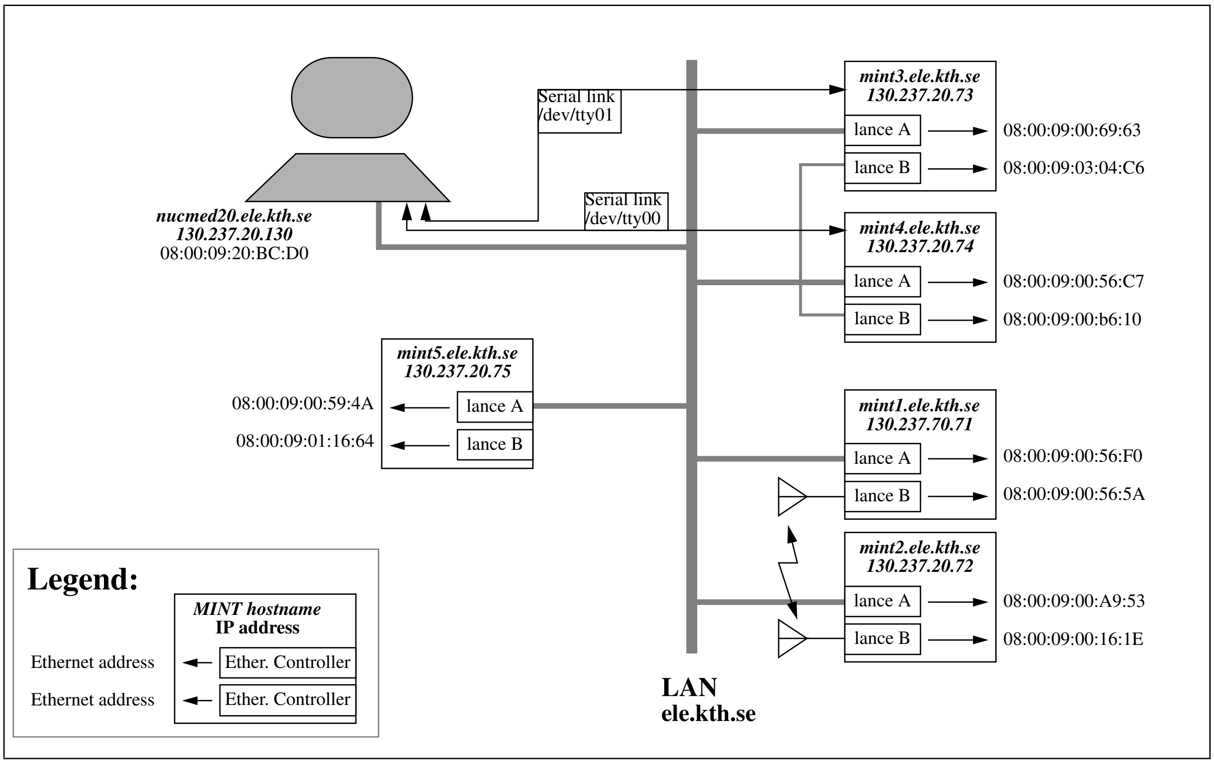



**Figure 7:** MINT family; Ethernet & IP addresses, hostname.

This part of document describes the current project from a technical viewpoint. Both devices and the development environment are described.

MINT is a computer whose architecture is based around the Motorola *MC68030*, a 32bit CPU, whith plenty of RAM (8Mb), ROM (1Mb), and equiped with many devices to allow communication with along others, such as: Ethernet controllers, SCSI port, serial and parallel interfaces. There are several copies of MINT (Figure 7). Currently they are five. All are identical, except for some internal parameters to distinguish them from each others, when they are connected to the same LAN[1].

Nucmed20.ele.kth.se is the controller. It stores the executable files to be downloaded (from a

1. Local Area Network

specific directory), and executes the process which does this transfert. Nucmed20 communicates with MINT during the transfer via the LAN, but has also a link via the serial interface that is used for a control console to the MINT. A ROM monitor is built-in to the MINT, which offers to the developer several simple commands.

dumburken.electrum.kth.se is my workstation. This workstation is located on another LAN which is connected by router to ele.kth.se. Via Telnet on nucmed20, I can get an access to all the resources of the working environment.

## 3.2. Development configuration

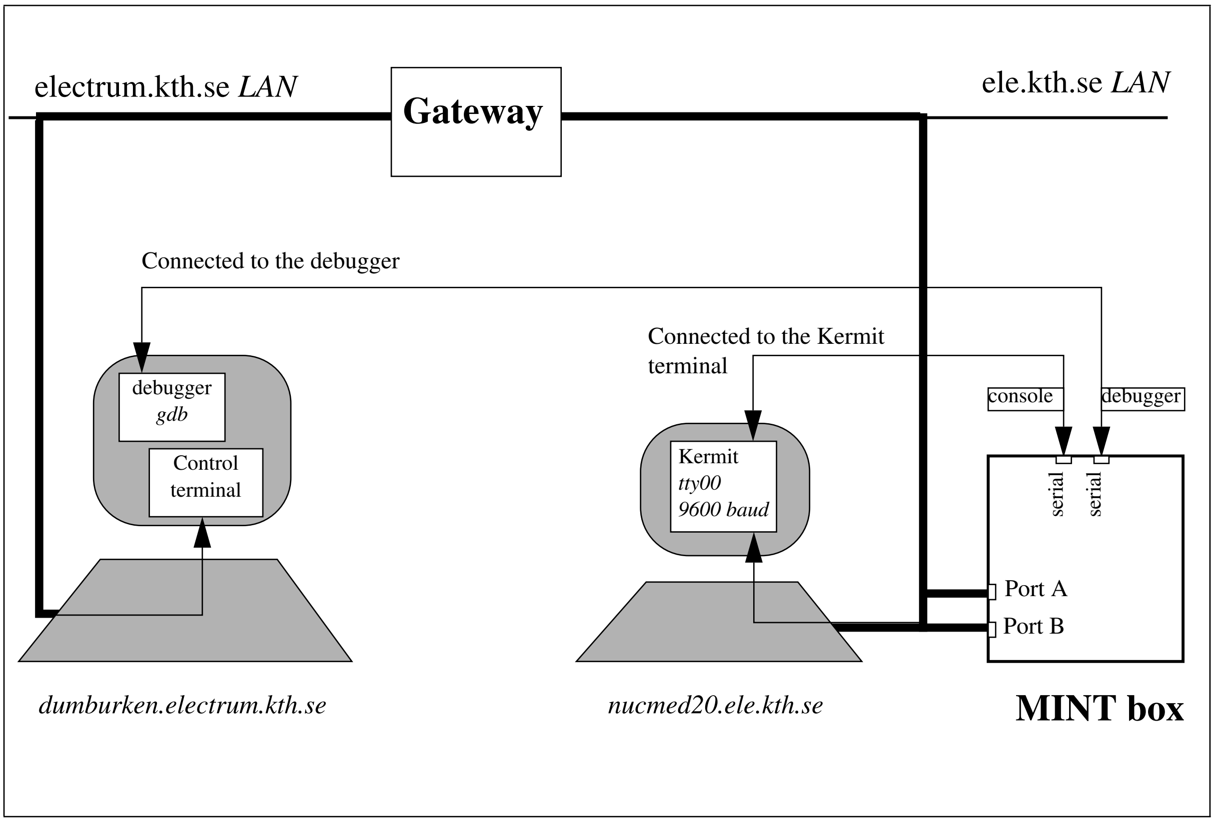



**Figure 8:** Connexion between the MINT box and the workstation

### 3.2.1. Physical configuration

The current development environment (show in Figure 8) provides a very comfortable way of working. In fact it is possible to sit down in front of the workstation and remotely control the MINT box.

The two available serial communication devices of the MINT box are used to facilitate the development of the future applications which will run on it, especially the radio device driver in my case.

The first one (labeled “console”) is associated with the built-in PROM monitor inside the MINT and performs the basic control process, i.e. sending commands and parameters to the monitor, printing on standard output some character strings, etc... As it is plugged in another workstation (nucmed20.ele.kth.se), the logical connection from my workstation to the monitor has to be realized via the network, through a gateway and after a telnet on nucmed20. Then by an intermediate Kermit session, you acces to the monitor.

The second one is used for the GNU debugger and is directly linked to my workstation. Despite the fact that the physical device is several rooms away.

#### 3.2.2. Development tools

A special *cross-compiler* for mc68030 created by GNU has been developed and generates executable code for the MINT cpu.

The *monitor* already contains in its ROM all the operating system to enable data exchanges with the network via its ports (A and B). After compiling your application, you just send to the monitor parameters like the name of the bootfile (i.e., your application), its location on the network giving directory name and ethernet addresses, and then just by ordering it, the monitor proceeds by itself to the dowloading of your application which subsequently begins to run.

The *GNU debugger* allows you to run the dowloaded program step by step, controlling the process via the second serial link (labeled “debugger” on figure).

## 3.3. Creating standalone program

The Mach kernel was not yet stable, so I had to produce and compile my code as a standalone program[2]. C-compiler and linker, especially intended to be used for the MINT (i.e., generating executable code for the MC68030 CPU), have been developed by Anders Klemets (TSlab), and are based on the famous *GNU C compiler*. These tools *are really a cross-compiler* and *cross-linker*, since they run on a Sun Sparcstation, but don’t generate code for the SPARC architecture.

The purpose of the following is not to enter in deep consideration of how this compiler & linker work, or on how to use the many parameters to be set up during the compiling phase. For all these details, I invite the reader to examine the Appendix B section of this document. Here is a summary of what must be done to compile standalone programs for MINT.

### 3.3.1. Typical Makefile

One first creates a Makefile for compiling standalone progams (here assumed to be ‘lr’ for lance_read), although one can invoke both the C-compiler and linker manually. In fact, it is much more judicious to utilize a Makefile, since you will certainly use many existing procedures (previously written for other usage). For instance, printing text on your console (kermit[3]) screen via the serial link is provided by one of those procedures.

A sample Makefile is shown in Figure 9, containing what is necessary to get an executable for any MINT program. Looking at it in detail, one finds:

- CC, AS, LD: the real names of the compiler, assembler and linker to invoke;

2. Program intended to be run on a bare machine, without OS support.
3. See section 3.4.5.1.

```
#
# Makefile for version 2.0 of the MINT monitor code
# (this file is not the original one. It has been shortened, to show just the most important lines to consider).
#
CC                  = ../bin/gcc
AS                  = ../bin/as
LD                  = ../bin/ld

# Mint1
#ETHERFLAGS = -DETHER_A="0x08,0x00,0x09,0x00,0x56,0xf0" -DETHER_B="0x08,0x00,0x09,0x00,0x56,0x5a"
# Mint2
#ETHERFLAGS = -DETHER_A="0x08,0x00,0x09,0x00,0xa9,0x53" -DETHER_B="0x08,0x00,0x09,0x00,0x16,0x1e"
# Mint3
#ETHERFLAGS = -DETHER_A="0x08,0x00,0x09,0x00,0x69,0x63" -DETHER_B="0x08,0x00,0x09,0x03,0x04,0xc6"
# Mint4
ETHERFLAGS = -DETHER_A="0x08,0x00,0x09,0x00,0x56,0xc7" -DETHER_B="0x08,0x00,0x09,0x00,0xb6,0x10"
# Mint5
#ETHERFLAGS = -DETHER_A="0x08,0x00,0x09,0x00,0x59,0x4A" -DETHER_B="0x08,0x00,0x09,0x01,0x16,0x64"

AFLAGS              = -mc68030
CFLAGS              = -Wa,-mc68030 -nostdinc -I. -msoft-float
CCFLAGS             = -O2 $(CFLAGS) $(ETHERFLAGS)
LDFLAGS_STANDALONEPRG = -N -e _start_main -Ttext 40000000

.c.o:
                    $(CC) -c $(CCFLAGS) $*.c -g

.s.o:
                    $(AS) $(AFLAGS) -o $*.o $*.s

OBJS =              mintmon.o all.o ether2.o pmap.o printf.o trap.o monitor.o ace.o rem-debug.o m68k-stub.o file_io.o ramdisk.o time.o
ram.o

clean:
                    /bin/rm -f lr.o

#-----------------------------------------------------------------------------
lance_read:         $(OBJS) lance_read.o mint_packet.o
                    $(LD) $(LDFLAGS_STANDALONEPRG) $(OBJS) lance_read.o mint_packet.o -o lr
#-----------------------------------------------------------------------------
lance_read.o:       lance_read.c ether.c ether.h
```

**Figure 9:** Makefile sample.

- ETHERFLAGS: the Ethernet MAC addresses for the MINT you are currently compiling for; since many routines related to the Ethernet controller use this preset, it is advisable to have it in the compiler command line, even if you don't make use of it. Indeed, it is only used during the preprocessing phase;
- CCFLAGS: the flags for the compiler (see the following section);
- LDFLAGS_STANDALONEPRG : the flag for linking the object files together, in a special way for the MINT (see the following section);
- .c.o: determines how every *.c file should be compiled, resulting in *.o object file;
- .s.o: invokes the assembler for any assembler file to be compiled;
- OBJS: this is just a set of object files, in order to highlight the content of the Makefile, and hence make easier the reading;
- clean: 'make clean' produces the erasing of lr.o; it is up to the user to add whatever file he wants to delete each time it is needed before recompiling;

- Then you put in the commands relevant to your program. For example, lance_read.c is a small application program created for experimenting with the Ethernet Driver. It listens to the network, and captures to memory all the packets that the LANCE[4] has accepted (i.e., the packet explicitly received); after its buffer is full, it displays the content of several fields of each packet (the purpose of such a program was just to visualize a UDP packet sent by a client on nucmed20). On your first line, you put the label (lance_read:), then the dependencies. The following lines represent the commands to perform (compiling & linking). (For further details on Makefile, please consult a suitable book.)
- for generating *.o file, only dependencies are needed, since the macro '.c.o' plays the role in place of the absent command line.

## 3.3.2. Compiler & linker arguments

This section explains the usage of the several arguments present for both compiling and linking commands.

### 3.3.2.1. Assembler

The -mc68030 option tells the Assembler that specific instruction code for the MC68030 CPU are present in the file, since by default it only considers 68020 instructions.

### 3.3.2.2. Compiler

We have seen in the previous section that compiler arguments are collected under a macro named CCFLAGS, and that the compiler is invoked in the following manner : "$(CC) -c $(CCFLAGS) $*.c -g". By replacing some labels with their explicit values, we get the resulting line command :

```
../bin/gcc -c -O2 -Wa,-mc68030 -nostdinc -I. -msoft-float
$(ETHERFLAGS)
```

where the involved arguments have the meaning described below:

`-c` : tells to the compiler not to produce executable code, but only object code. The latter will be linked to be run on the MINT.

`-O2` : is for optimisation, at level 2. The compiler tries to reduce code size and execution time.

`-Wa,-mc68030` : tells the compiler to pass "-mc68030" as an argument to the Assembler, when the latter is invoked.

`-nostdinc` : tells the compiler not to search the standard system directories for header files. Only directories specified with -I option are searched.

`-I` : for indicating the path of include files. For instance, the dot placed just after, specifies that the search directory is the current working directory. It is not possible to use standard include libraries, since there are specific to the workstation you are working on rather than the machine you are compiling for.

`-msoft-float` : for generating output containing library calls for floating point. Since the motherboard is not equiped with a floating point unit (M68851 coprocessor), the compiler shall not make use of corresponding instructions (default procedure), but has to replace such instruction with software library routines.

4. Local Area Network Controller for Ethernet.

`-g` : this argument is optional and in fact only needed for debugging the resulting executable code.

Now, when the compiler has finished its task, you get a *.o file. It still remains to link it properly.

#### 3.3.2.3. Linker

The second step is to invoke the linker with specific parameters, depending on the target computer it will linked for, that is to say, the MINT. Parameters directly relate to the memory configuration and the start address.

The complete command line has the following shape:

```
../bin/ld -N -e _start_main -Ttext 40000000
```

Here are the arguments passed:

`-N` : tells the linker that the data segment should be directly follow the text segment, rather than being allocated at a page boundary. The MINT monitor currently only correctly handles programs linked in this fashion.

`-e _start_main` : informs the linker about the first function to be called in the program.

`-Ttext 40000000` : indicates to the linker at which location in the memory the program should use place. Thus your program can only be loaded at that memory address, otherwise it will crash.

Now that you have got a standalone program, it remains to copy it into the right directory and download it to the desired MINT.

### 3.3.3. Mandatory files - location

Some C-sources have been developed, giving access to many procedures involved in the management of several of the MINT's resources. I strongly suggest the use of this existing code. Indeed, it is so structured that most of them use other procedures, located in other files.

The original files that comprise the *mintmonitor*, can be found in the directory:
`/afs/it.kth.se/home/tslab/klemets/MINT/newmintmon`.
It contains procedures for running the LANCE controller (reading, writing, initializing), printing out and reading characters on the serial interface, etc.

The GNU C compiler and linker can be found in the directory:
`/afs/it.kth.se/home/tslab/klemets/MINT/bin`.

I advise adopting the following configuration (in fact, the configuration I am using): in your current working directory, create a new one in which will be copied all files from `[...]/newmintmon` previously noted. Still in the current one, create a link named `/bin` to the `[...]/MINT/bin` previously noted. When this is done, adopt the former as your new working directory. I guarantee that this works well.

### 3.3.4. Switching between workstation- & MINT- working configuration

Since the GNU compiler & linker have the same name, one can get confused and by mistake invoking the m68k-compiler instead of the "Sun" compiler, when code is to be run on Sun. Thus a solution might be the creation of an alias which swaps the working configuration when necessary (that is to say, changing paths). Since using aliases is much easier, one can create two as following:

```
# Save current path into ipath
set ipath=$path
```

# Defining new aliases

```
alias ws 'set path=(/afs/it.kth.se/home/tslab/klemets/MINT/new-mintmon /afs/it.kth.se/home/tslab/klemets/MINT/bin $ipath)'
alias mint 'set path=(/afs/it.kth.se/system/sun4m_413/usr/include $ipath)'.
```

The above lines should be put in your .login file, so aliases are automatically defined each time you login.

## 3.4. Downloading your standalone program

Downloading a standalone program might be the simpliest thing to do on MINT. The many tasks to realize before are not so difficult. What you need to do is to set the name of the executable file you wish to transfer from one location (say nucmed20) to your MINT (with respect to the direction, such a transfert is called *down*loading), using the ‘set bootfile <your program>’ command on the monitor, to the filename previously placed into the downloading directory (described below on 3.4.4.), then to press the ‘b’ key followed by ‘return’, and the MINT performs the downloading process.

This booting is based on a TFTP (Trivial File Transfert Protocol). On one hand, the MINT acts as a client, sending requests to a server. On the other hand, the server computer which receives the request and acknowledges it, acts and supplies the demanded file via TFTP.

### 3.4.1. BOOTstrap Protocol server

The first computer to engage in the dialog is the MINT (i.e., the MINT you are using). The request is sent as a link level broadcast, since the MINT does not know which computer on its LAN is the server (nor does it know its own IP address). After being reset, the MINT only knows its two ethernet addresses, one for each LANCE, and will complete its identity by receiving information from the server.

BOOTP [10][11], also called Bootstrap protocol, uses UDP (implementation of BOOTP is therefore easy since requests and replies are in UDP datagrams, not special link-layer frames) and normally works in conjunction with TFTP. It is intended as an alternative to RARP for bootsrtapping a diskless system to find its IP address. BOOTP can also return additional information , such as the IP address of a router, the client’s subnet mask, and the IP address of a name server.

Since BOOTP is used in the bootstrap process, a diskless system needs the following protocols implemented in ROM: BOOTP, TFTP, UDP, IP, and a device driver for the local network.

Hence the BOOTP server, when receiving the broadcast packet, extracts the ethernet source address, checks it against its database by comparing it with the several ethernet addresses mentioned in the /etc/bootptab file, and if it finds it, retrieves the hostname of the current MINT.

All explicit data exchanged between MINT and the server are put into a Bootp message, which

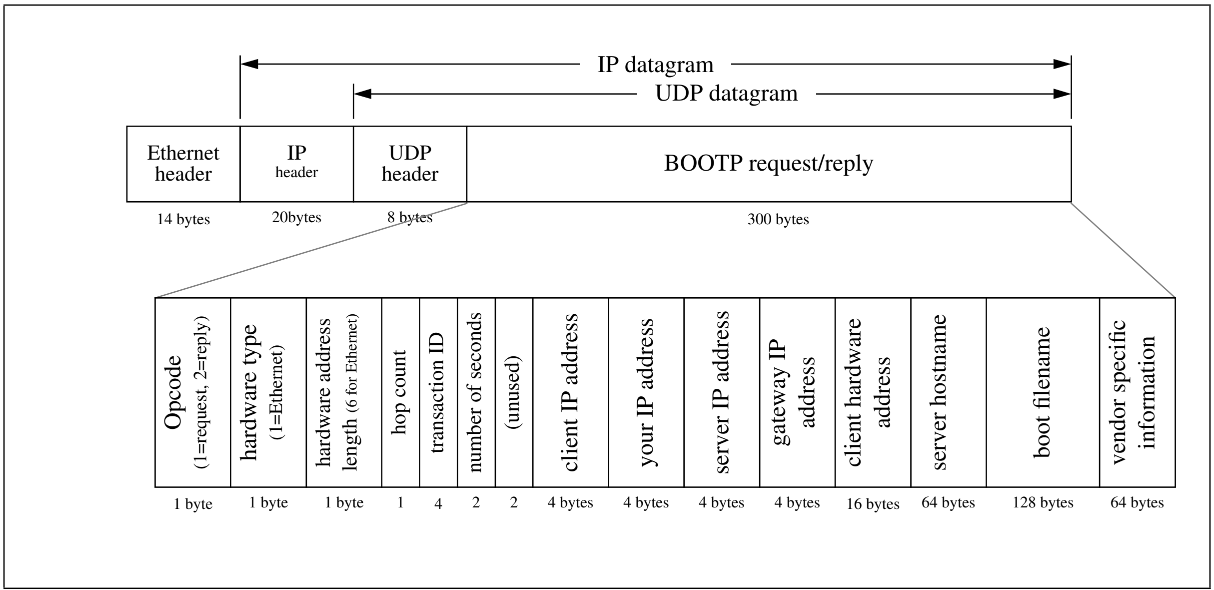



**Figure 10:** Format of BOOTP request and reply

is inserted into a UDP datagram, and then sent to the lower layer via the UDP protocol.

### 3.4.2. Trivial File Transfer Protocol

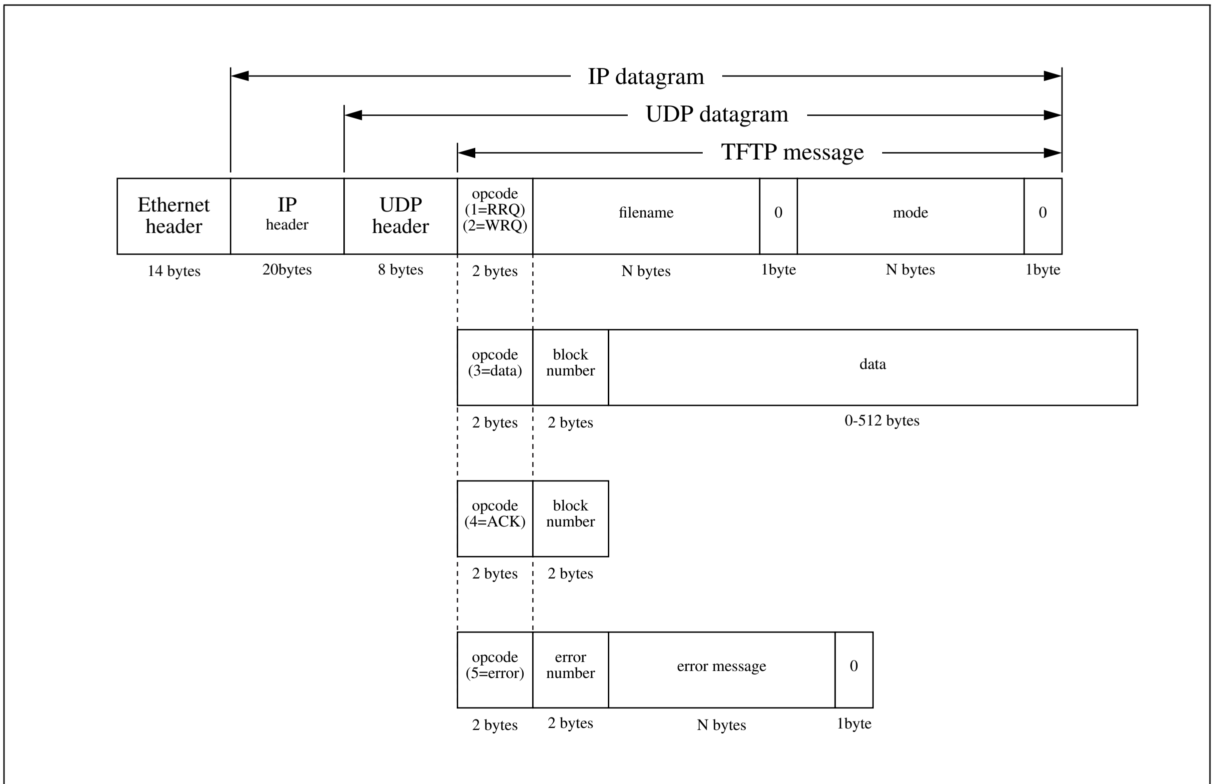



**Figure 11:** Format of a TFTP frame

TFTP [9][11] is a file transfer protocol implemented on top of UDP and is designed to be small and easy to implement, and to fit into ROM. TFTP is used by BOOTP when bootstrapping a diskless system (such as MINT or any X-terminal). The only thing it can do is read and write files from/to a remote server.

Files are transfered between the server and the client (i.e., the MINT) by using TFTP. The file is cut into blocks of 512 bytes (the last block may be less than 512 bytes wide). Each block is then put into a TFTP frame. Each time a TFTP is sent, the server awaits the acknowledgement from the client. If such an acknowledgement is not received, after the timer has expired, the server resends the frame.

This method provides a flexible way of downloading, because TFTP provides retry to compensate for the unreliability of UDP, since UDP packets might be lost, and because it can be adjusted to any kind of machines.

### 3.4.3. bootptab - file content

Associated with the bootp demon, the *bootptab* file is the reference table for all the MINTs that might be connected to the network. For each one, several parameters are recorded, allowing the demon to establish the communication between it and the client (i.e. a mint) in the correct way. The sample below (only part of the file) shows the construction of such a file.

Associated with each name declaration, there is a set of parameters, determining the exact nature of the host. That is to say:

ip =... : ip address,

ha =... : the hardware address, i.e. ethernet physical address (LANCE B (lower controller) is diffentiated from LANCE A (upper controller) by sticking a ‘b’ letter after the name declaration).

ht =... : the hardware type (must precede the ha tag).

Supplemantary data are supplied moreover, such as:

bf =... : the default program to be downloaded if the user has started the downloading process without indicating its name.

tc =... : refers to another paragraph containing additional information and default values.

```
# bootptab entry for testing first MINT box
#
mint1:\
 tc=global.defaults:\
 bf=mintbootfile:\
 hn:\
 ht=ether:\
 ha=0800090056f0:\
 ip=130.237.20.71

mint1b:\
 tc=global.defaults:\
 bf=mintbootfile:\
 hn:\
 ht=ether:\
 ha=08000900565a:\
 ip=130.237.20.71
```

**Figure 12:** Piece of the *bootptab* file

### 3.4.4. Directories

bootptab resides in the `nucmed20.ele.kth.se:/etc` directory. Each time a new MINT is connected to network and might be accessible, the bootptab file must be updated.

Concerning standalone programs to be downloaded, they must be copied into the `nucmed20.ele.kth.se:~maguire/tftpdir` directory. Otherwise, the bootp server will not find it upon a MINT request. This directory is the name directory of pseudo user "tftp" as defined in /etc/passwd.

### 3.4.5. MINT monitor

A monitor has been implemented in the EPROM of the MINT. It contains both the Bootp and TFTP code, and provides the user with rudimentary functions, but some of them are essential (i.e., the boot command).

The monitor awaits order from the serial interface (port #A), which is therefore linked to the master computer (i.e. nucmed20.ele.kth.se in the current configuration).

#### 3.4.5.1. Connection to the MINT's "console" port (via Kermit)

Kermit is a file transfer program for moving files between many machines of different operating systems and architectures. In the case of the MINT, Kermit is used in order to establish the connection between the MINT and nucmed20 via the serial link, while allowing the use to be at this or another workstation rather than at a terminal physically attached to the MINT.

Kermit requires you to set up some parameters before activating the connection. The command "`set line /dev/tty00`" determines the physical port to open, whereas "`set speed 9600`" adjusts the bit flow. The two previous operations can be performed automatically by creating a .kermrc file, filled up with the two commands.

In the case of nucmed20, the latter possesses 2 serial ports, the second denoted by /dev/tty01. This can be useful when working simultaneously on two MINTs.

#### 3.4.5.2. Available commands

Let us begin by resetting the computer. The version number of the EPROM resident monitor appears on the screen, then the prompt (@). Typing '?' outputs a list of possible commands (Figure 13).

```
@?
A -> ALTER bytes
B -> BOOT using TFTP
D -> DISPLAY bytes
G -> Go to address (LOADENTRY default)
I -> Re-INITIALIZE monitor
L -> LIST files in ramdisk
M -> Byte alter using LONG (32 bit) accesses
P -> PRINT environment variables
R -> Registers and flag display
S -> SET environment variables
T -> TRACE using remote GDB
U -> USE stored registers and go to address
W -> Byte alter using WORD (16 bit) accesses
```

**Figure 13:** Monitor commands

The major commands are as follows:

`b :` starts the downloading process of the file you specified. The name of the latter can be passed as first argument to this command; for example: "`b lr`" (where "lr" is an executable file placed in nucmed20.ele.kth.se:~maguire/tftpdir);

`p` & `s` : displays the many variables involved during the downloading process. Each variable can be set up with either the "`s`" or "`set <variable> <value>`" command.

`g` : allows the user to start running the program at a specific address. By default, it will start at the loadentry address, i.e. rerun the program.

`t` : is useful for the debugger. When entered, the monitor gives the control to the remote debugger (see Annex).

`i` : is to re-initialize the monitor. It might be more secure to do it, if ever a bug has occured during the run time, giving back control to the monitor, since innerly data may be affected.

The other commands are less interesting. I invite the reader to test them by himself.

# Chapter 4: MINT's Hardware Modifications

## 4.1. Introduction

This chapter reflects the modifications performed to the MINT motherboard. The MINT was designed prior to the radio interface. Therefore, some programmable chips implemented on the board have been reprogrammed in order to interface to the radio. Moreover, a buffer circuit has been inserted between the data bus and the Xilinx chip which constitutes the digital interface of the radio.

## 4.2. Motherboard updates

Owing to the addition of the radio device to the system, a new memory area must be allocated for this device. The space between 0xC0000600 and 0xC0000700 (which was initially deserved for another device that was not implemented) has been allocated to the radio. Since the motherboard utilizes many PAL chips that integrate specific functions, we had to update two of them - one to allow the device to be selected and one to provide data acknowledgement for bus cycles.

For compiling the corresponding files, we used the *ABEL PAL Assembler*.

### 4.2.1. Changing 'I/O device DECoder' PAL[1] chip (ver2.0)

Referring to the schematics, the iodec (**I/O dec**oder) PAL is intended to deliver a signal, based on the memory space being accessed, to the appropriate device. That is to say, if there is read or write between 0xC0000600 and 0xC0000700, then the radio device is enabled since the PAL sets the $\overline{\text{ENA\_RF}}$ to an active low level.

The PAL source code for I/O device DECoder (PAL chip U105 on the schematics), shown in Figure 14, has been updated in that way.

### 4.2.2. Changing 'DaTa ACKnowledge' PAL chip (ver2.0)

There is a handshake process implemented by the MC68030. Each time a location in the physical addressable space is accessed, for reading or writing, the CPU stops and waits for an acknowledgement on two of its pins, namely $\overline{\text{DSACK0}}$ and $\overline{\text{DSACK1}}$. Those signals are supplied by the *dtack* PAL, which collects independant DTACK signals from all the current devices. We had to implement such a

1. PAL: Programmable Array Logic. Such a electronic chip allows you to *synthesize* any specific functionality you desire, relating to the physical configuration required, in order to fit well on the board, without being obliged to redesign partly or completely, as it would surely be the case when using convential chips.

```
module io_dec flag '-r3'
title 'I/O decoder PAL for MINT'

 iodec device 'P22V10';

" Pin definitions.

"Inputs.
       mfc pin 2;
       as pin 3;
       ma31,ma30,ma24,ma10,ma9,ma8 pin 4,5,6,7,8,9;

"outputs
       ena_rf pin 14 = 'neg,com';
       ena_lan0 pin 15 = 'neg,com';
       ena_lan1 pin 16 = 'neg,com';
       ena_byte pin 17 = 'neg,com';
       ena_ace pin 18 = 'neg,com';
       ena_sr1 pin 19 = 'neg,com';
       ena_sr0 pin 20 = 'neg,com';
       ena_scsi pin 21 = 'neg,com';
       ena_time pin 22 = 'neg,com';

" signal definitions

       H,L,X,CK,Z = 1,0,.X.,.C.,.Z.;

" sets

       myio = [ma10,ma9,ma8];

" Outputs that need to be ENABLED

outputs = [ena_rf, ena_lan0, ena_lan1, ena_byte, ena_ace, ena_sr1, ena_sr0, ena_scsi, ena_time];

" Outputs being used as inputs and should never be ENABLED

equations in iodec

!ena_rf = ma31 & ma30 & (myio == 6) & !as & mfc;
!ena_byte = ma31 & ma30 & (myio == 5) & !as & mfc;
!ena_lan1 = ma31 & ma30 & (myio == 4) & !as & mfc;
!ena_lan0 = ma31 & ma30 & (myio == 3) & !as & mfc;
!ena_scsi = ma31 & ma30 & (myio == 2) & !as & mfc;
!ena_ace = ma31 & ma30 & (myio == 1) & !as & mfc;
!ena_time = ma31 & ma30 & (myio == 0) & !as & mfc;
!ena_sr1 = ma31 & !ma30 & ma24 & !as & mfc;
!ena_sr0 = ma31 & !ma30 & !ma24 & !as & mfc;

ENABLE outputs = (mfc # !mfc); " these should always be enabled"

end io_dec
```

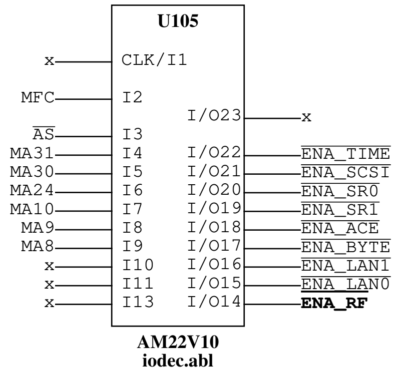



**Figure 14:** *iodec.abl* file to burn the iodec PAL (U 105) (in bold characters are the modifications)

signal for the radio device, shown as rf_dtack in the listing Figure 15.

## 4.3. Physical connection of the digital transceiver.

The next step was the connection of the radio part to the motherboard. Actually there were 12 wires to solde:

- 8 wires for data transfer. It is important to note that the data bus of the MC68030 consists of 32 bits, and when connecting a 8 bit wide data bus from an external device, care must be taken. When handling bytes, the MC68030 processor puts/gets them on/from the lower part of the data bus, that is to say, bits 24 to 31;
- 1 wire linked to the $\overline{\text{ENA\_RF}}$ pin (pin #14) of the I/O device DECoder, which gets active at low level when a physical address is accessed between 0xC000600 and 0xC000700;
- 1 wire for the READ/$\overline{\text{WRITE}}$ signal coming from the processor;
- 2 wires for selecting the internal registers of the digital transceiver. These wires are connected to the

```
module d_dtk flag ‘-r3’
title ‘Dtack generator for MINT’

 dtk device ‘P22V10’;

" Pin definitions.

"Inputs
      mclk pin 1;
      a_dtack pin 2;              "ace
      t_dtack pin 3;              "timer
      dr_dtack pin 4;             "dram
      r_dtack pin 5;              "rom
      s_dtack0 pin 6;             "sram on lance 0
      s_dtack1 pin 7;             "sram on lance 1
      sc_dtack pin 8;             "scsi
      l_dtack0 pin 9;             "lance 0
      l_dtack1 pin 10;            "lance 1
      ds,as pin 11,13;

"outputs as outputs

      dsack0 pin 19 = ‘neg,com’;
      dsack1 pin 18 = ‘neg,com’;

"outputs as inputs

      b_dtack pin 14 = ‘pos,com’;
      rf_dtack pin 15 = ‘pos,com’;  "radio

" signal definitions

 H,L,X,CK,Z = 1,0,.X.,.C.,.Z.;

" sets


" Outputs that need to be ENABLED

      outputs = [dsack0,dsack1];


equations in dtk

" All registers except for LANCE internal registers provide an 8 bit port.
" LANCE registers are 16 bit. All memory types except SRAM provide a
" 32 bit port. SRAM provides a 16 bit port.

      !dsack0 = !a_dtack # !t_dtack # !dr_dtack # !r_dtack # !sc_dtack # !b_dtack # !rf_dtack;

      !dsack1 = !dr_dtack # !r_dtack # !s_dtack0 # !s_dtack1 # !l_dtack0 # !l_dtack1;


" This is needed so that our outputs used as inputs work right

      b_dtack = 1;
      rf_dtack = 1;

      ENABLE outputs = (ds # !ds);  " these should always be enabled"
      ENABLE b_dtack = 0;           " this should never be enabled"
      ENABLE rf_dtack = 0;          " this should never be enabled"

end d_dtk
```

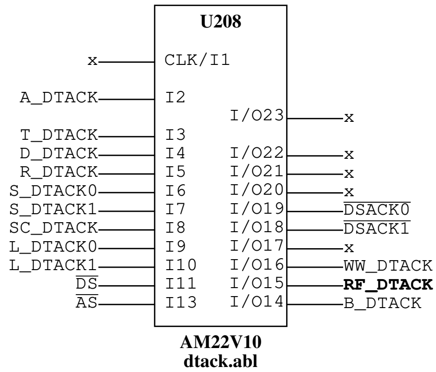



**Figure 15:** *dtack.abl* file to burn the dtack PAL (U 208) (in bold characters are the modifications)

address bus, namely pins ma0 and ma1. At the present, there were not enough pins available on the Xilinx chip to allow management of 8 registers. But as soon as a new version is realized, ma2 (and perhaps ma3) will be used to expand the number of addressable registers in the digital transceiver.

Since the Xilinx was not programed to generate DTACK signal (besides, no pin was available for that task), we simply connected the pin $\overline{\text{ENA\_RF}}$ (pin #14) of the I/O device DECoder PAL chip to the RF_DTACK input pin (pin #15) of the DaTa ACKnowledge PAL chip, since they have the same active low level. This makes the system correctly work.

## 4.4. Radio access protocol (ver1.0)

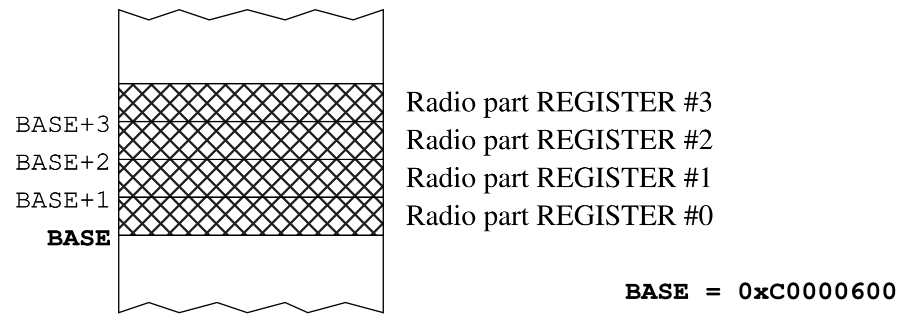



**Figure 16:** Radio part register location in the memory map

The radio device will have many registers (8 at most), but unfortunately, the current device can only support 4. In fact, it does not matter, since the protocol does not take into account the number of registers available.

Registers are located in the lower part of the device memory area. As they are each 8bits long, each register gets an address according to the scheme shown in Figure 16.

To put data into a register, it is sufficient to write to the corresponding address, as shown in the in C-code below:

```
*(BASE + reg_number) = byte_to_write;
```

On the opposite direction, extracting data from a register is simply done by the following line of C-code:

```
byte_read = *(BASE + reg_number);
```

# Chapter 5:
# Getting to know with the Kernel: the Dummy Driver

## 5.1. Introduction

This chapter presents and roughly explains the features of programming at the kernel level, which is somewhat different from creating a user program. The dummy driver is intended to practice with the related notions and shows in concrete what happens on the normaly hidden side of the kernel.

## 5.2. Theoritical approach

A common user process running on top of the kernel, gets access to files and devices via a standard I/O interface. All the resources (keyboard, terminals, audio, disks, network,...) available on a worstation are presented to the user as if they were files, collected by convention in the /dev directory, which one can open, close, read, and write them. This way of proceeding simplifies at most the exchange of data between a process and a resource.

### 5.2.1. For the basic user

From a user viewpoint, using a driver requires opening a filename[1][15], which is represented by a specific icon, as shown in Figure 17, and then accessing the device just as if it were a regular file.

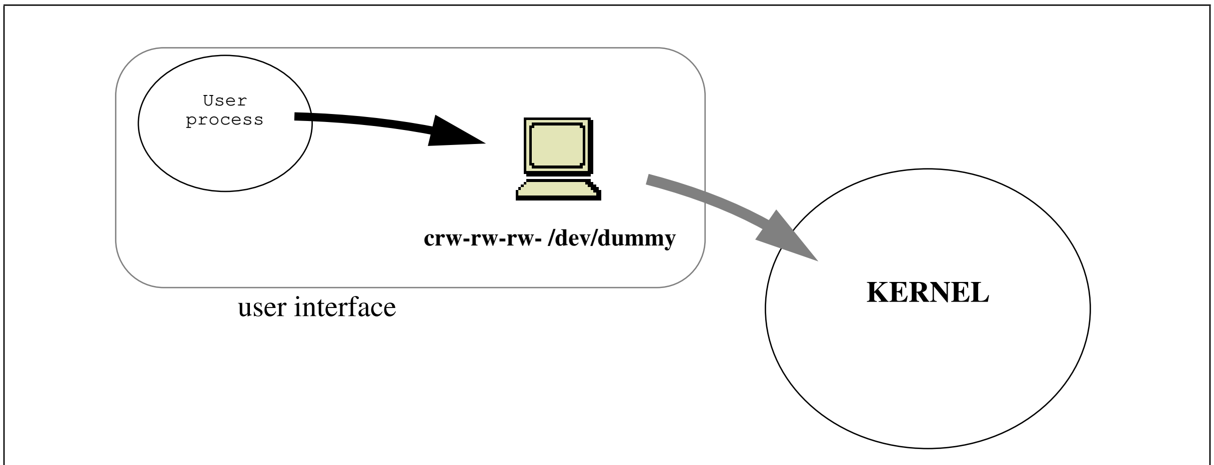



**Figure 17:** Driver file from the user viewpoint

It is therefore very simple for a user to exploit a device, to exchange data with it. The user has not to take care of how the device is designed and how it handles data it is supposed to deliver. The I/O standard interface of the Kernel is in charge of that task. More precisely, it ensures the communication linkage between the user and the driver of the considered device. The driver should be visualized as a kind of bridge flowing data from/to the Kernel which relays them to the demanding user.

The UNIX system contains two types of devices, block devices and raw character devices[1][15]. The dummy device driver belongs to the later category and is recognisable by the first character (i.e., ‘c’) present in the access-rights of the file associated with this device.

For example, assuming that the dummy driver is installed, when one lists the filenames in the directory /dev , regarding the Dummy Driver, it should appear something like that:

```
crw-rw-rw- 1 rootstaff   59, 0    Jun 22 13:55/dev/dummy0
```

The first character of the upper line shows that the filename /dev/dummy0 is a raw character device driver. Then, from left to right, we can find:

- the access-rights which allow read/write for the owner, his group and the other people;
- the size equal to 1 which corroborate the definition of a raw charcater device driver;
- the owner (i.e root) and the group (i.e. staff);
- the major and minor numbers (see 5.2.3.);
- the time of creation;
- and finally, the filename itself.

What actually has the user to know is the filename of the driver and whether he has permission enough to access it. The other parameters are not destinated to him.

### 5.2.2. For the programmer

For the programmer, a driver looks like something different. In fact, the concept of independent process is not valid, since a driver, in terms of machine instructions, is a set of functions intended to be called only by the kernel (that means that there is no main() procedure).

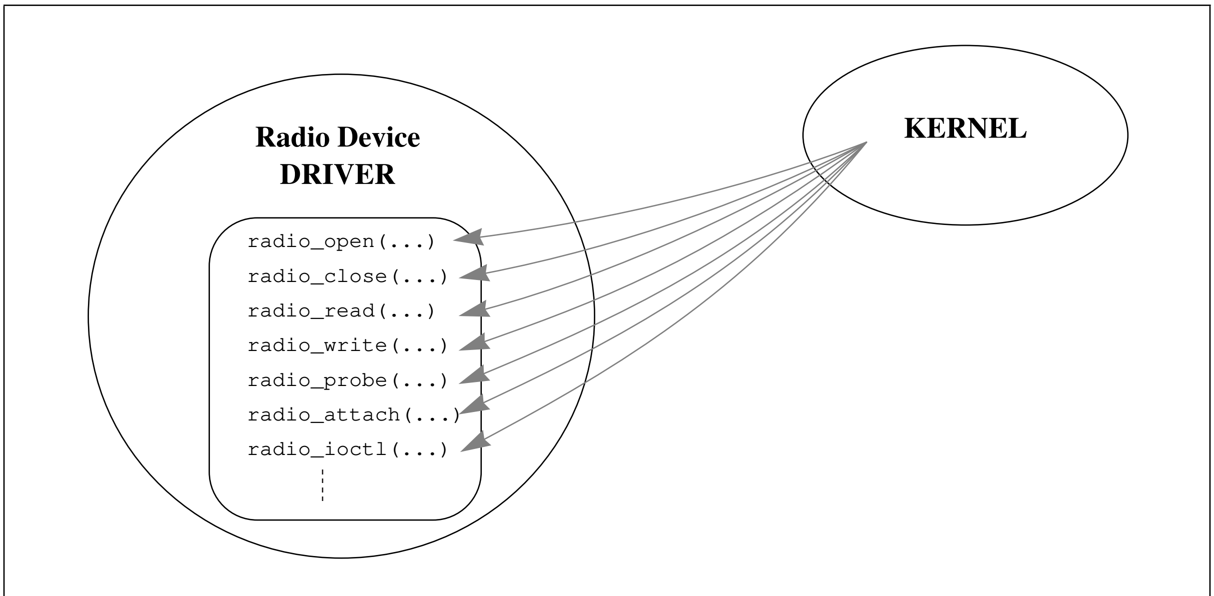



**Figure 18:** Driver file structure from the programmer viewpoint.

Each time a user tries to access the driver (i.e., he performs open, read, write, close on the associated filename), the kernel converts this into appropriate calls to the subroutines which constitute the driver.

The driver programmer has to follow rules. The number of routines that compose the body of the driver is restricted and their functionalities are also predefined. Going deeplier into the kernel source code, the conf.c[1] file is interesting to look at. It shows how drivers are implemented, according to a specific declaration.

The `cdevsw`[2] table specfies the interface routines present for character device drivers. For each character device driver, the associated routine names are grouped in one structure which has the following shape[15]:

```
struct cdevsw = {
     int (*d_open)();         /* routine to call to open the device */
     int (*d_close)();        /* routine to call to close the device */
     int (*d_read)();         /* routine to call to read from the device */
     int (*d_write)();        /* routine to call to write to the device */
     int (*d_ioctl)();        /* special interface routine */
     int (*d_reset)();        /* reset device and recycle its resources */
     int (*d_select)();       /* routine to call to select the device */
     int (*d_mmap)();         /* routine to call to mmap the device*/
     struct streamtab *d_str; /* support for STREAMS */
     int (*d_segmap)();       /* handles mmap devices that support d_mmap */
};
```

Normally, a character device driver may provide seven functions: `xxopen(),xxclose(), xxread(),xxwrite(),xxioctl(),xxselect(),xxmmap()`. But the Dummy Driver only provides the five first ones. By convention the *xx* prefix shall be replaced with the name of the driver, that is to say *dummy* in the case of the Dummy Driver.

Each time a user process issues a system call, like a read on the driver via its filename, the kernel translates it into the call of the corresponding routine provided by the driver. This routine is pointed by the pointer given by the appropriate field in the cdevsw structure. In concrete, regarding the Dummy Driver, we should find the address of the dummyread() routine, i.e. the word *dummyread*, in the (*d_read)() field. For futher details, I advice the reader to consult the SUN AnswerBook[15] and the source code of the Dummy Driver.

The second step of the implementation is to insert this structure into an array which collects the similar structures of all other drivers. This array is defined as `struct cdevsw cdevsw[]`. It can be found in the `conf.c` file. Wherever the Dummy cdevsw structure is placed in the array, it can be indexed. This index thus dertermines the *major* number of the driver.

```
struct cdevsw cdevsw[] =
{
     {
     open,      close,      read,       write,       /*index*/
     ioctl,     reset,      select,     mmap,
     streamtab, segmap,
     },
};
```

Other things have to be done, i.e. settups, updates, in other files belonging to the set of files

1. conf.c is one of the most important file regarding the drivers implemented in the system. This file is among those that are used during the kernel compiling process.
2. cdevsw stands for character device switches.

describing the kernel. But this only concerns permanent device drivers. Such drivers are automatically running up when the system boots. The dummy driver is not particularly useful. It is just built to get familiar with kernel programmation. Moreover it drives no device at all (one calls such driver *pseudo-driver*). Besides, I was not allowed to modify the kernel, nor rebooting the system for my experiments. The section “Writing Device Drivers” of the AnswrBook explains how to add a new permanent device driver in details.

To load the Dummy Driver into the kernel, another solution has been chosen. It exists a program especially intended for this job. Modload and the way to use it are described in section 5.3.

### 5.2.3. Major and minor device numbers

Major and minor numbers[1][15] are two numbers always relating with device drivers.

As been said in the previous section, the *major* number is the index in the device switches array. So it actually selects a driver. When a user process accesses a file, the kernel needs to know which device driver is responsible for it. To make this determination, it must get the name associated with the file. From that name it can derive an i-node[3] and thus a major device number. From this number it retrieves the xdevsw[4] structure of the driver and subsequently calls the appropriate driver’s routine.

The i-node also supplies a *minor* number. This one is passed to the driver as a parameter for local interpretation, and usually determines the unit to control if the device comprises several ones.

The connection between the device name and its major and minor numbers is made by way of the device entry in the /dev directory. The administration command mknod helps to do this connection. The syntax is as follows:

mknod *filename driver_type major minor*

In the case that many units are available on the device, it must be created a filename for each one consequently. Assuming that our dummy driver is supposed to control two units, we would be obliged to generate two filenames as follows:

```
root@host :> mknod /dev/dummy0 c major 0
root@host :> mknod /dev/dummy1 c major 1
```

Now, accessing one of this two files will make the same driver operate, but with a different minor number passed as parameter.

The choice of the major must reflect the index of the driver xdevsw structure in the xdevsw array. For loadable drivers (see section 5.3.), this number is given as an output from the modload program.

## 5.3. modload: loadable device driver

Modload[15] stands for Module Loading and is a tool provided with the SunOS Operating System running of Sun SparcStation. This program allows the UNIX programmer to load into the kernel a new device driver. The advantage of using this method is to avoid shutdown of the computer and there-

3. i-node stands for *information node*.
4. x means either c or b.

fore keep the kernel running during the loading operation.

The other way of implementing a driver into a kernel is to modify the C-source code of the kernel itself. That is to say, setting and adding variables and declarations that relate to the considered driver. However this way of implementation makes the presence of the driver permanent, and the latter will be initialized each time the system is booted. This effect is not wished with the dummy driver, since it will be loaded into the kernel only occasionally. Thus the first method, i.e. using modload, is retained.

For complete manual of the modload program, I invite the reader to consult the AnswerBook[15] or the Man Pages. However, below is a brief description of the command used and its associated parameters for loading the Dummy Driver into the kernel.

The syntax of the command line is as follows:

```
modload dummy.o -conf DUMMY -entry _duminit -exec dummyinstall
```

where:

- `dummy.o` represents the object code of the driver. See section 5.5.3. to know how to build this file.
- `DUMMY` is a file containing information about the system. In fact, in my version it only contains the *charmajor*[5] instruction;
- `_duminit`[6] is the name of the initialization function (i.e. the first function called by the system) run when the module is finished to be loaded. This function makes the linkage of the module to the kernel;

dummyinstall is a shell script executed when the loading has been successful. It actually invokes the mknod command to create a filename for the driver. Modload gives several arguments to the script, the third and fourth ones of them are respectively the block major and character major numbers.
The shell script thus contains the following lines:

```
#!/bin/csh -f
# character driver
if $4 != "0" then
          if ( ! -r /dev/dummy0 ) then
               echo /etc/mknod /dev/dummy0 c $4 0
               /etc/mknod /dev/dummy0 c $4 0
               chmod 666 /dev/dummy0
          endif
endif
```

As it can be seen it uses the fourth argument as the major number, gives 0 as minor number, specifies the type of the driver, i.e. a raw character device driver, and the filename with the unit number as last character to avoid any ambiguity;

Such driver intended to be loaded when keeping the kernel running is called *loadable driver*. Their structure is quite similar to traditional driver's one, but includes other declarations that make the driver be able to be dynamically linked to the kernel. The specific set of variables and functions is called a *wrapper module* (see section 5.5.). The wrapper module besides contains the description of the `xxxinit()`[15] function (`duminit()` in the case of the Dummy Driver).

5. specifies the major number of the driver. But it seems that this number is superceded by the one modload has chosen.
6. do not forget the underscore symbol before the name of the function

### 5.3.1. modunload and modstat

To remove a loadable driver from the kernel a specific command also exists. It is about *modunload* which needs the identification number of the module and the name of a shell script executed when the unloading is successful.

This *id* number is given either by modload as first argument to the shell script (see section 5.3.) (which is also printed on the screen when modload is used with option -v), or by *modstat* which lists all the modules currently integrated in the kernel.

For instance, typing “modstat” on the prompt could give the following result:

```
root@host :> modstat

 Id   Type   Loadaddr   Size    B-major  C-major  Sysnum   Mod Name
 3    Drv    ff1fe000   1000             59.               dummy
 2    Net    ff1a8000   33000                              bf
 1    User   ff13c000   63000                              AFS fs
```

The second line from the top presents data about the Dummy Driver, espaecially its name, its major number displayed in the C-major column (it is therefore a character device driver), its id essential for modunload, its type.

The shell script executed when the unloading of the driver has been a success should remove the filename associated to the driver. For the Dummy Driver it gives:

```
#!/bin/csh -f
echo rm -f /dev/dummy0
rm -f /dev/dummy0
```

For further information about modunload and modstat, please consult the AnswerBook[15] or the Man Pages.

## 5.4. Summing up figure

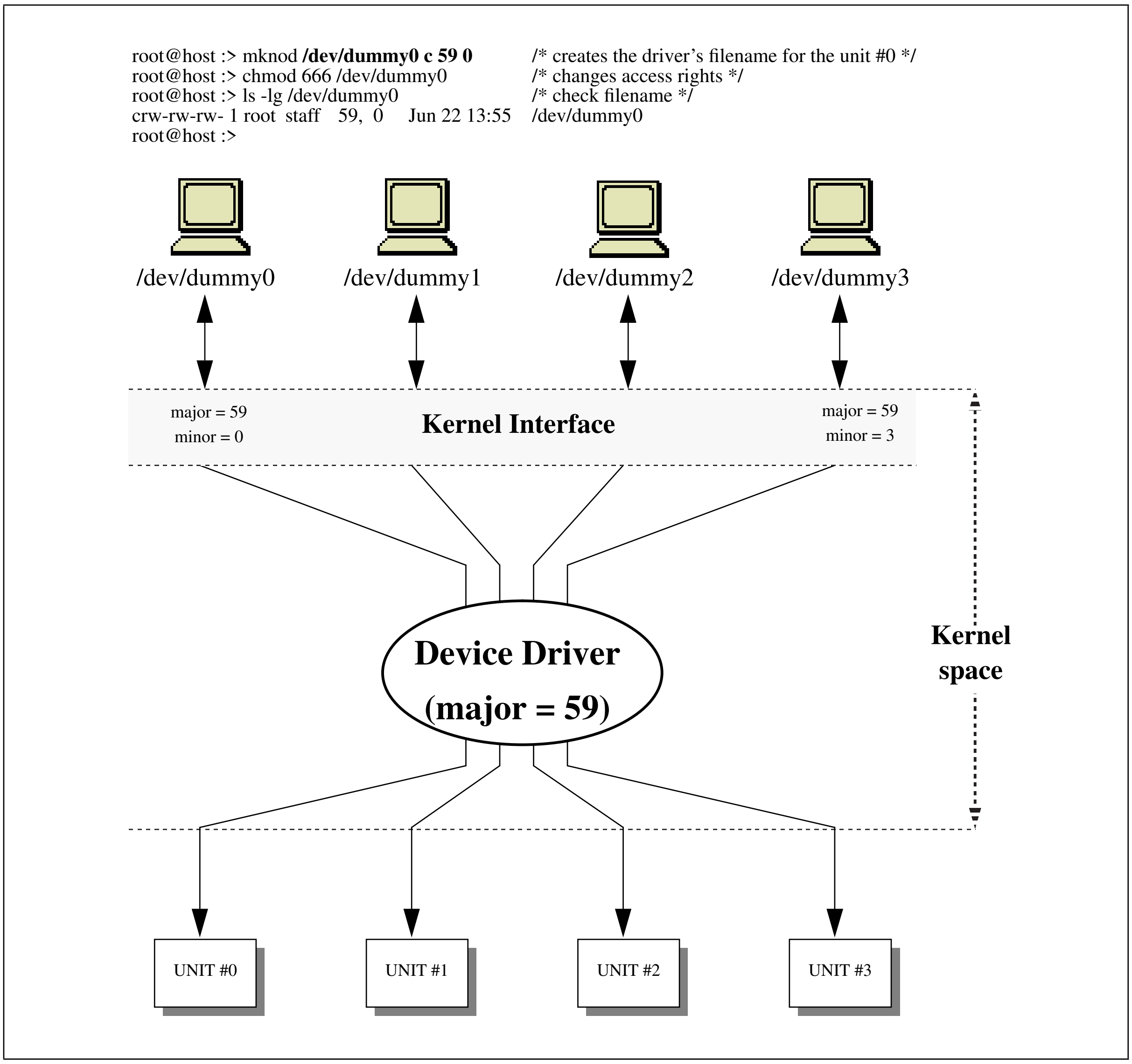



**Figure 19:** Relationship between driver, device units and UNIX filenames. Major and minor numbers have been arbitrary chosen. The major number refers to the position of the `cdevsw` structure in the `cdevsw[]` array. The minor number refers to the unit.
The user gets access to the device unit through its associated filename and the same driver for all units. The filename should reflect the number of the unit by having it at the end (e..g. dummy0 for unit #0).

## 5.5. Implementation of the Dummy Driver

The source code of this loadable driver could have been written in one block, but in order to distinguish the body of the driver itself and the code part which helps to dynimacally link it to the kernel, it has been split into two files.Those files are thereafter separately compiled and then linked.

The first file, i.e. `dummy_driver.c`, describes the functionalities of the driver. The second file contains all the specific variables needed for the dynamic linkage. Such a file is called *wrapper*[15].

The body of this driver has been inspirated from the Skeleton driver described in the Answer-Book. It takes up the global shape with some specific adjustements according to its finality. The sources can be found in `~guerin/PROJECT/dummydriver/` .

### 5.5.1. dummy_driver.c

The body is composed of many routines on one hand, and variables declarations on the other hand. There is no diagram to show here, since a driver is only a set of functions independant and separated from each others. Moreover, there is no main() function within.

This driver only provides simple functions: open, close, read, write, ioctl. In addition, it includes two more specific routines which are dummyprobe() and dummyattach():

- dummyprobe(): this is normally the first function of the driver which is called to initialize the driver when the system is booting.It essentially tells the kernel how much memory space the driver requires for its running. Although it is implemented in the code source and also declared in specific key structure, I have noticed that this routine is never called. It seems that in the case of loadable driver, this routine is in fact useless.
- dummyattach(): does boot-time, device-specific setup and initialization. Its proprer tasks are not limited to the initialization of actual hardware devices - xxattach() is also used to set up and initialization local data structures. In the case of the Dummy Driver, as soon as the module is loaded into the kernel, dummyattach() is called several times (13 times exactly). I have not been able to check whether it was a bug or a normal event, the documentation does not give hints on whatever bug it could occur. Anyway this function is empty.

Beside, we have basic routines called by the kernel when a user process issues a corresponding operation on the filename associated:

- `dummyopen()` is called when the user process opens the filename of the driver. It is passed to the routine the *dev* number which encodes the major and minor numbers; the *flags* number which passes the access mode (O_RDONLY, O_WRONLY, O_RDWR).
- `dummyclose()` is called when the user process closes the filename of the driver. Same arguments as for dummyopen are passed;
- `dummyread()` and `dummywrite()` are called when the user process issues respectively a read and a write on the filename. The dev number is passed, also a special structure *uio* to transfer data between the calling process and the routine, and the flags. Such structure is described in the Answer-Book[15] page 6-13.
- `dummyioctl()` is called when the user process issues a ioctl on the filename. Are passed to the routine, the dev number, an integer cmd as a command, data which is a pointer to an array of maximum 255 bytes, flags.
  It is very important to encode cmd properly. Indeed, it is used by the kernel to manage the data array (size, value) and the access mode (O_RDONLY, O_WRONLY, O_RDWR). For encoding cmd, the library sys/ioccom.h describes macros for that task. It is then up to the routine to do what should be done regarding the value of cmd.
  data is a pointer to an array of 255 bytes located in the kernel space. The data contained within are

copied from the homolog array in the user space, by a length precised by a value encoded in cmd (see sys/ioccom.h for further details).

Concerning the variables declarations, many specific structures have to be filled up:

- `struct mb_driver dummydriver` : this structure contains information relevant to the device driver as a whole, as opposed to information about individual devices or controllers. It contains a number of pointers to driver functions. These pointers, like those in cdevsw and bdevsw, are used by the kernel as entry points into the driver.
- `srtuct mb_device dummyinfo`: it contains infornation about the device and is used in conjuction with the mb_ctrl struct. Since the Dummy Driver does not driver any hardware, those structures are useless. Thusin the fields of other structures which require the address of them, it has been put NULL (or zero) value. For figuring out the real signification of such structures, please refer to the AnswerBook (page 3-13).

The other variables are less important. So I will not stay on explaining them here.

### 5.5.2. dummy_wrapper.c

This piece of code is mandatory for any loadable driver. It permits the dynamical linkage of the module to the running kernel. That is the reason why it must be written with care and seriousness, otherwise it may crash the system when modload tries to load the driver into the kernel.

Two very important structures are declared within this code:

- `struct dev_ops dummy_dev_ops` : as shown in the listing, fieldsof that structure are respectively set up with the routine names of the driver. It is via it that the kernel can make a appeal to the appropriate routine regarding the operation issued by the user process on the driver's filename.
- `struct vdldrv vd` : this one is mandatory in a loadable driver. It is used by modload to establish the linking of the different strutures with the kernel. It mainly comprises a pointer to the dev_ops structure, the name of the module, a pointer to the cdevsw structure, one to the mb_driver structure, and the charmajor number[7]. All other fields non used must be filled with a zero value. Do note, by having a look at the sun/vddrv.h lib, that this structure varies with the type of SUN station you are using.

Then we can find the driver's entry point routine duminit(). This routine is the very first routine executed as soon as the module has been succesfully loaded into the kernel by modload. It performs the linking of the driver's symbols with the kernel symbols table.

### 5.5.3. Building the object file dummy.o

`dummy.o` represents the object code of the driver. It is obtained by compiling `dummy_driver.c` and `dummy_wrapper.c` separately, and then by linking the object code. For the compiler, it must be precised two macros declared by the "`-D`" option as follows:

```
cc -c -DKERNEL -D[machine_type] dummy_driver.c
```

7. It is preferable to let the system choose it. This is done by setting charmajor to zero.

cc -c -DKERNEL -D[machine_type] dummy_wrapper.c

where `machine_type` is the type of the machine you are working on. It can be got by using the command `arch -k`.

`dummy_driver.o` and `dummy_wrapper.o` are then linked with the option `-r` to preserve the relocation of the code, resulting in the definitive object file `dummy.o`.

### 5.5.4. Trial and results produced

Once the driver is loaded, it must be tested. In that sense a testdummy.c program has been created to check the good working of the functionalities of the Dummy Driver. Below is a table comparing in pairs the operation issued by the test program and the reaction of the driver.

| Displayed results of the dummy driver test program | Displayed results from the driver side |
|---|---|
| opening /dev/dummy*: success | *** open() function has been called.<br>dev = 15104 (unit = 0), flags = 3 |
| reading /dev/dummy*: success<br>message got is:<br>"You have performed a read() on the dummy driver!" | *** read() function has been called.<br>dummyread(dev, uio)<br>dev_t dev = 15104; (unit = 0)<br>struct uio *uio = 0xf0505ea4;<br><br>Printing uio structure:<br>struct uio {<br>struct iovec *uio_iov = {<br>caddr_t iov_base -> /**to be filled by the function* */<br>int iov_len = 48/* *length is specified by user* */<br>}<br>int uio_iovcnt = 1<br>off_t uio_offset = 0<br>short uio_segflg = 0<br>short uio_fmode = 3<br>int uio_resid = 48<br>} |
| writing /dev/dummy*: success | *** write() function has been called.<br>dummywrite(dev, uio)<br>dev_t dev = 15104; (unit = 0)<br>struct uio *uio = 0xf0505eac;<br><br>Printing uio structure:<br>struct uio {<br>struct iovec *uio_iov = {<br>caddr_t iov_base -> hello world!<br>int iov_len = 12<br>}<br>int uio_iovcnt = 1<br>off_t uio_offset = 0<br>short uio_segflg = 0<br>short uio_fmode = 3<br>int uio_resid = 12<br>} |

**Figure 20:** Results of the Dummy Driver test program.

| Displayed results of the dummy driver test program | Displayed results from the driver side |
|---|---|
| Before ioctl() call, arg points on:<br>“data passed via ioctl”<br>After ioctl(), now arg points on:<br>“DATA passed via ioctl” | *** ioctl() function has been called.<br>dummyioctl(dev, cmd, data, flags)<br>dev_t dev = 15104 (unit = 0);<br>int cmd = 0xc0150001; (x=0, y=1, t=21)<br>caddr_t data = 0xf0505dc4;<br>int flags = 3;<br><br>Value pointed by *data -><br>:64:61:74:61:20:70:61:73:73:65:64:20:76:69:61:20:69:6f:63:74:6c: |
| closing /dev/dummy*: success | *** close() function has been called. |

**Figure 20:** Results of the Dummy Driver test program.

In all driver’s functions, the arguments passed are displayed in order to have a concrete view of what happens. read and write are more special cases since data have been passed through the `uio` structure. By also displaying its content, and especially the `iovec` structure, we can notice two fields: the first one, `iov_base`, points on the byte array and serves for the data transfer between the user process and the driver. In the case of a read, this array is obviously empty, but full for the write function. The second one is the length of data transfered. In the case of a read, it specifies the length requested by the user process. It is up to the function to fill this array without exceeding this number - otherwise the surplus of data would be ignored by the kernel -, or not to send by returning until the array is full, making the user process issuing the read wait (blocking read).

# Chapter 6:
# The Dummy RadioNet Driver

## 6.1. Introduction

The Dummy Radionet Driver is an intermediate step before accomplishing writing the final RadioNet itself. The purpose of this driver is to simulate the presence of the real RadioNet driver on a SUN Sparcstation. This makes therefore the interfacing between the SNMP agent and the driver be possible and thus helps the progress of the elaboration of this agent whereas the definitive RadioNet is not yet achieved.

## 6.2. Shape of the Dummy RadioNet Driver

Roughly the body of the driver is quite similar to the Dummy Driver's one. What is added is the handling of the data structure involved in the management of the several mobile hosts tranceiving on the radio link.

## 6.3. Data exchange between driver and SNMP agent

Data exchanges (uplink and downlink) will be done under a specific structure, and take place at a cadence of 100Hz minimum. The cadence is imposed by the coherence time. The entire structure is transfered between the driver and the agent.

Such a structure is defined as follows:

```
struct MHOST {
       char      ether[6];  /* the ethernet address of the mobile host */
       char      ip[4];     /* the IP address */
       char      RSS;       /* the strength of the signal received */
       char      ARSS;      /* the average strength of the received signal */
       char      DSS;       /* the strength of the transmitted signal */
       short     key;       /* the key for the spread spectrum */
       int       flag;      /* for indicating if the structure has undergone
                             an update in one of the other fields */
       };
```

This structure is 16 bytes long. Considering a management of 10 mobiles grouped in a table, it makes the latter table be 160 bytes long, which can fit well in the data field of an ioctl call. Let us call this table MHostTab:

```
#define MHostTab 10
struct MHOST     MHostTab[MAXMHOSTS];
```

Both the SNMP agent and the driver handle the same kind of array - this leads to two separated arrays, one placed in each separated memory space -, because they cannot work on the same one at the same time. Indeed, competing accesses could lead to unpredictable results. The solution is therefore to use duplicate tables. Each table is respectively treated by the agent and the driver. Then, every 100th of a second, the tables are exchanged.

The table exchange is made via an ioctl function. Tables are passed by argument `arg` of this function.

## 6.4. ioctl routine

The main functionality of this routine is to perform the tables exchange between the SNMP agent and the driver.

From the agent's viewpoint, such a call is done as follows:

```
ioctl(fd, cmd, table);
```

where `fd` is the file descriptor of the driver. This value is returned by the `open("/dev/RadioNet", O_RDWR)` procedure. It is to notice that the open must be done in READ and WRITE mode, otherwise the exchange of data via `table` cannot be performed correctly.

- `cmd` is a value passed and considered by the driver as an order. It is up to the driver to do whatever regarding this value. In fact, the driver defines the meanings of the different values that `cmd` can take. The user has thus to choose between the proposed set of values. Another very important point is that cmd encodes information extracted by the kernel itself before passing this value to the driver. For encoding `cmd`, a set of macros have been defined in `sys/ioccom.h` library. They take into account the input/output flow direction, a character which can take whatever value but usually refers to the driver, a byte, and the length of the array passed. For instance, for the RadioNet driver, cmd to order an table exchange is encoded in the following way:
  `#define EXCHANGE _IOWRN(r, 0, MAXMHTAB)` , where _IOWRN() is defined in that way:
  `#define _IOWRN(x,y,t) (_IOC_INOUT | (((t)&_IOCPARM_MASK) << 16) | ('x'<<8) | y)`
- `table` is an array of maximum 255 bytes. As said previously, only relevant data are passed to the driver, in a certain length according to the one encoded in cm by the t parameter. This makes the transfer quicker.

From the driver programmer viewpoint, the radionetioctl() routine takes into account four parameters, which are:

- `dev` which encodes the major and minor numbers;
- `cmd`, previously determined;
- `data` which is an array containing the data the SNMP agent sends me;
- `flags` which reminds the access mode on the filename (i.e. O_RDONLY, O_WRONLY, O_RDWR);

If `cmd` has the right value, then the exchanging of the tables is performed. First, the driver must save what the agent sends in a second array different from the current one it is managing. Then it copies its current. array into the `data` array supplied by the kernel. That is the reason why two arrays (known as UserTab1 and UserTab2) for mobile hosts management are declared in the driver's body.

Each time ioctl is called, the driver swaps over its two arrays.

## 6.5. Installing the RadioNet driver on SUN workstation

Since the RadioNet driver is a loadable driver, modload is therefore used to load it into the kernal.

The compiling operation is the same as described for the Dummy Driver. Thus, we do:

```
cc -c -DKERNEL -D`arch -k` radionet_driver.c
cc -c -DKERNEL -D`arch -k` radionet_wrapper.c
ld -r -o radionet.o radionet_driver.o radionet_wrapper.o
```

The resulting radionet.o can now been used by modload[1]. Hence we do:

```
modload radionet.o -v -conf RADIONET -entry _rninit -exec radionetin-
stall
```

- `RADIONET` contains only the “charmajor 0” instruction;
- `rninit()` is the entry point routine of the wrapper module;
- `radionetinstall` is the shell script executed when modload has succesfully loaded the driver into the kernel; it actually makes a new node entry in the node table by using the command: `/etc/mknod /dev/radionet c $4 0` , where $4 returns the major number given by modload as one of this outputs.

Once the driver is definitively installed, the `ls -lg /dev/radio*` command can check its presence.

## 6.6. Tests

A small test program has been created in order to verify if the driver correctly answers.It issuesopen, close and ioctl. ioctl is called several times and time is picked up twice, at the beginning of the loop and at the end. Hence we get an estimation on how long an ioctl call lasts.

The name of the test program is `testradionet.c` and is located in the same directory as the RadioNet driver’s sources.

The time measured for 1000 successive ioctl calls varies between 40ms and 90ms, depending on how busy the processor is, because of the parallel processes and the packet flow on the Ethernet (it generates interrupts with high priority level).

1. You must be root to use modload.

# Chapter 7: The MINT's RadioNet Driver

## 7.1. Introduction

Implementing the real RadioNet Driver is the final step. Nevertheless, the experiment gained with creating the Dummy RadioNet Driver was not enough, since there is a MACH 3.0 kernel ported on MINT instead of the traditional UNIX kernel. Concerning writing device driver, it is a bit different from the way how to proceed in UNIX. The routines supported are not formatted in the same way as for UNIX. At the moment when this report is written, the RadioNet driver is not achieved in terms of C code, but only this additional code implementation has been conducted.

## 7.2. Some words about MACH

MACH is a modern, microkernel-based operating system which has been developed for only a couple of years. Mach is the achievement of a research which aims to demonstrate that operating systems could be structured in a modular way, as a collection of processes that communicated by messages, including over a network.

The first version of Mach was released in 1986. It was a combination of the ealier Mach concept and the 4.2BSD into a single kernel, leading to a large kernel but compatible with the 4.2BSD. Since this time, Mach is now at version 3.0 and has seen its size decreased by removing all the Berkeley UNIX code from the kernel, remaining a microkernel consisting of pure Mach.

Goals of Mach can be summed up as follows:

- providinga base for building other operating systems (e.g., UNIX);
- supporting large sparse address spaces;
- allowingtransparent access to network resources;
- Exploiting parallelism in both the system and the application;
- making Mach portable to a larger collection of machines.

The Mach microkernel has been built as a base upon which UNIX and other operating systems can be emulated.

This emulator is done by a software layer that runs outside the kernel , in user space, as shown in Figure 21. It should be noted that multiple emulator can be running simultaneously.

The Mach kernel provides process management, memory management, communication, and I/O services. Mach 3.0 has inherited the I/O management structure of Mach 2.5, which in turn derived its I/O system from BSD UNIX. Under BSD, devices could only be described as character-oriented or block-oriented devices. This gross characterization made it difficult to share code across functionally

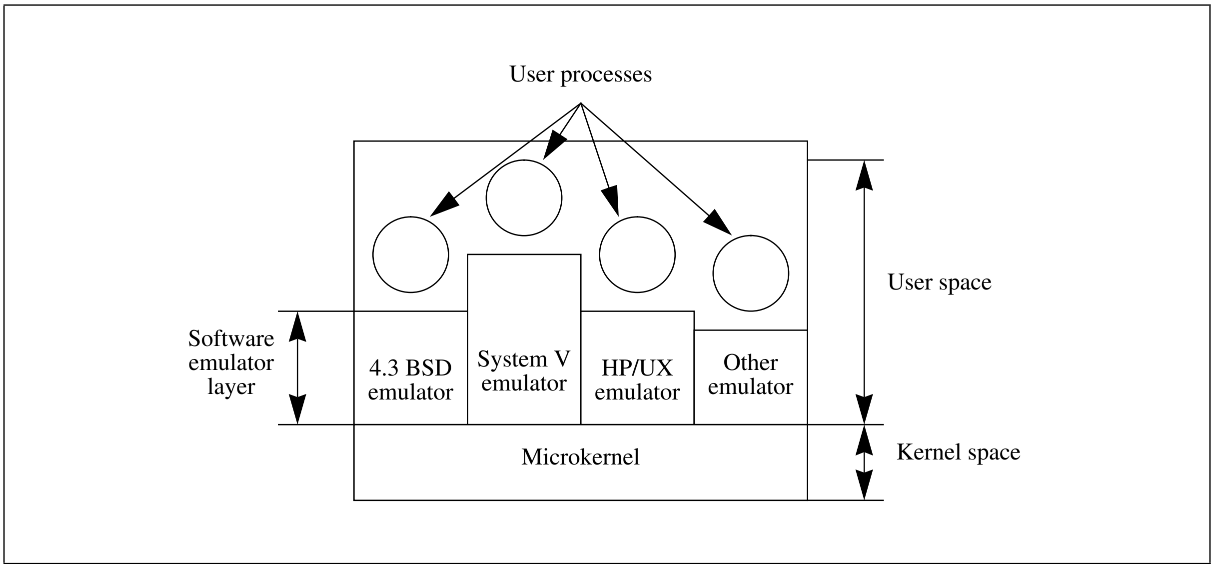



**Figure 21:** The abstract model for UNIX emulator using Mach.

equivalent devices which happened to exist on different system platforms. The I/O system simply had no structure to allow similar devices, to share code, even though the functions provided by the similar devices are nearly identical. The concept of device-independent device drivers decreases the amount of Mach's machine-dependent code, thereby decreasing the size of the operating system and the amount of time required to port the system to a new system architecture. Moreover, maintenance of the system is greatly simplified because changes to the I/O system need only be applied to each class of device, not to each device for each processor architecture for each system.

## 7.3. I/O interface for Mach 3.0

The I/O interface consists of the following remote procedure calls:

- device_open (*master_device_port, mode, name, device*): open procedure, returns a device port;
- device_close(*device*): close procedure;
- device_write(*device, mode, recnum, data, num_bytes, bytes_written*): write procedure, returns the number of bytes actually written;
- device_read(*device, mode, recnum, bytes_wanted, data, bytes_read*): read procedure, returns the
- device_map(*device , protection, offset, size, pager, unmap*): map procedure, returns a port pager for mapping to user space;
- device_set_status(*device, flavor, status*): change the device status, device-specific.
- device_get_status(*device, flavor, status*): inquiry the device status, device-specific.

The ioctl routine is peculiar to the UNIX system. Hence, on Mach, it is translated into the calls of device_set_status() and device_get_status(). The order by which these routines are called is unknown. In fact, I have not found hints in the documentation I possess, and decrypting the UNIX emulator was too hard and thus required much more time than actually remaining. However, I have done the modifications by implementing two types of codes, each with respect to the order of the call.

## 7.4. Implementation in the kernel

To implement the driver into the kernel source code, the way how to proceed is the same as described for the Dummy Driver. The array `dev_name_list[]` lists the structure of all drivers. The location of the RadioNet structure in this array can be thus indexed, giving us the major number of the driver.

```
struct dev_ops dev_name_list[] =
{
     [...]

     {
     name,       open,        close,        read,
     write,      getstat,     setstat,      mmap,
     async_in,   reset,       port_death,   subdev,
     dev_info
     },                                                  /* index */

     [...]
};
```

## 7.5. Data structure

The data structure for the management of the Mobile Hosts is exactly the sane as described for the dummy RadioNet.

## 7.6. Changing code regarding the transmission

Decrypting the C-code made by another person is often a very difficult task to accomplish. During this work, I came on several solutions about where to implement the code specific to the radio part in the Lance driver's source code, according also to the mechanism of the Ethernet controller.

Every incoming packet from the upper layer is handled by the driver and put into the transmitting ring buffer of the Ethernet controller (LANCE), if there is enough place left. Then the LANCE takes a enqueued packet from the buffer, and puts it out onto the network. The operation is automacally performed by the chip, until the buffer is empty.

Since the radio parameters must be updated in the radio part for almost every outgoing packet, the first solution proposed ( (1) in Figure 22) is to intercept the packect that the chip is currently taking from the buffer. Then, according to the Ethernet address and to the MHosts table (see 7.5.), the appropriate radio parameters are given to the radio. This solution can be applied because the chip delivers a interrupt signal for every packet sent out.

The second solution ( (2) in Figure 22) can be that the size of the ring buffer is decreased to 1. That means that the buffer contains at most 1 packet. Thus each time the driver fills it with one packet, it can set up the radio paramaters for that packet at the same time. So we are sure that this packet will be sent on the radio link with the good radio parameters.

The third solution ( (3) in Figure 22) is as follows: instead of putting packets in the LANCE's ring buffer, an additional shadow buffer is used in place of the ring buffer. Firstly, those packets are then associated in a structure with their radio parameters (i.e. the DSS value - the strength of the output

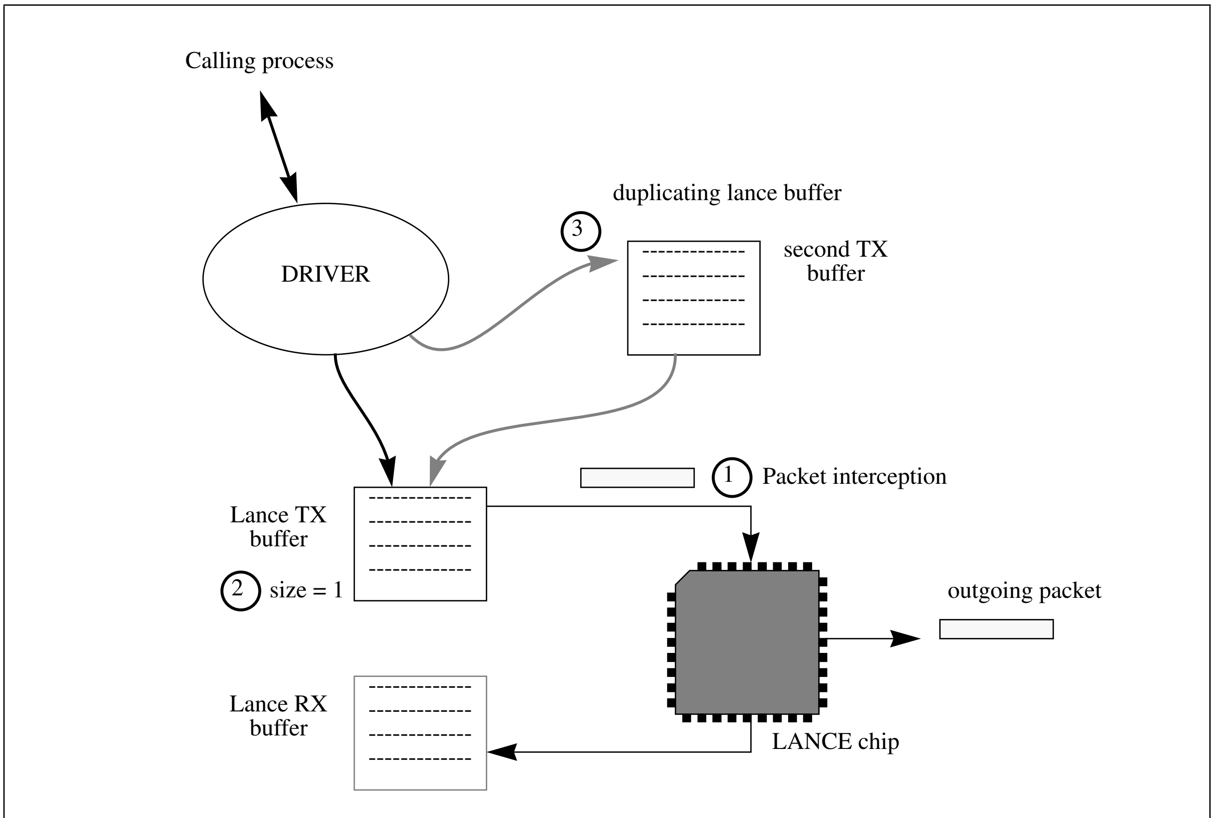



**Figure 22:** Several possibilities of implementation for the transmitting case.

radio signal) and stored in the shadow buffer. Secondly, all the packets associated with the same set of radio parameters (same DSS, same key) are transfered in the LANCE ring buffer while the registers of the radio are updated. The advantage of this solution is that the radio registers are only updated one time whereas several packets are sent out. This makes the system be faster and therefore more efficient. But it is much more complicated to implement the corresponding code than it is for the two first solutions.

## 7.7. Changing code regarding the reception

It is mandatory to get the parameters from the radio each time an incoming packet is delivered by the LANCE chip and transfered into the receiver ring buffer. Fortunately, in such a case, the chip generates an interrupt signal which makes the system run the specific routine for that interupt.

C-code for getting the radio register values shall therefore be inserted in this routine which is `se_rint()`.

# Conclusion

The Mobile INTernet router (MINT for short) is a core element of the Walkstation II Project and ensures the routing of packets from/to mobile hosts. For that task, it has been equiped with many communication devices, especially for Ethernet, and a radio part..

The MINT's motherboard is equiped with a 32bit powerful microprocessor (MC68030), which should make the MINT be powerful enough to support heavy applications, like the SNMP manager that Markus Oelhafen has carried out.

For MINT, special and specific tools have been developed, making possible the porting of the Mach kernel on MINT. With such an operating system, MINT can now be considered as a real workstation, allowing multiusers work at the same time.

For the radio link, the MINT comprises an Ethernet controller and a radio transceiver is hooked on it. Thus for both component, a specific code inserted in the kernel's one must be tought. We call such code a driver. The RadioNet driver is therefore in charge of driving at the same time the Ethernet device for flowing the packets and the radio part to set up or get radio parameters regarding each packet.

My work has not been carried out until the end due to many technical reasons. Firstly, the Mach kernel has been ported to the MINT very late (in the middle of May), and moreover we (my colleague Magnus and I) felt a lot of difficulties to recompile the Mach kernel for our MINT specific environment.After three weeks of labour, we finally succeeded. Secondly, the available MINTs we were using had not enough memory to support Telnet connection call from outside, and a connection trial usually led to a trap exception in the kernel, making the MINT crash.

Now one MINT at least has been equiped with an additional memory, extendingthe size of the dynamic RAM up to 16 megabytes. This should eliminate all the previous problems.

Implementing the code specifically for the radio part shoud not be hard, but actually takes a lot of time, actually more than it remained when I began to work on the RadioNet driver.

# References


[1] Maurice J. Bach, *“The Design of the UNIX Operating System*”, Pentice-Hall, 1986.

[2] Anders Klements, Gerald Q. Maguire, Frank Reichert, Mark T. Smith, “*MINT-A Mobile Internet Router*”, First IEEE International Symposium on GLOBAL DATA NETWORKING, 1993.

[3] Mobile IP Working Group, “*Routing Support for IP Mobile Hosts*”, work in progress, 1993.

[4] Daniel Kerek, Hannu Tenhunen, Gerald Maguire Jr.,Frank Reichert, “*Direct Sequece CDMA Technology and its Application to Future Portable Multimedia Communication Systems*”, IEEE ISSSTA ’94: Third International Symposium on Spread Spectrum Techniques & Applications.

[5] John Ioannidis and Gerald Q. Maguire Jr., “*The Design and Implementation of a Mobile Internetworking Architecture*”, In USENIX Winter 1993 Technical Conference, pages 491-502. USENIX Association, January, 1993.

[6] Tomoki Osawa, “*Realization of CSMA/CA by use of a CSMA/CD chip*”, Telecommunication System Lab, KTH, Feb. 94.

[7] Teet Saluvere, “*FPGA based CDMA Radio*” (Master thesis), Electronic System Design Lab, KTH, Nov. 93.

[8] Raphael Rom, Moshe Sidi, “*Multiple Access Protocols - Performance and Analysis*”, Springer-Verlag, 1990.

[9] K. R. Sollins, *“The TFTP Protocol (Revision 2)”*, RFC-783, June 1981.

[10] Bill Croft, John Gilmore, “*BOOTSTRAP Protocol*”, RFC-951, September 1985

[11] W. Richard Stevens, “*FTP/IP illustrated, Volume 1 - The Protocols*”, Addison-Wesley Professional Computing Series.

[12] Bruce Tuch, “*AN ENGINEER'S SUMMARY of an ISM BAND WIRELESS LAN*”, NCR

Systems Engineering, Utrecht, 1991.

[13] Frank Reichert, “*Integration of Mobile Communication into Fixed Networks*”, Proceedings of NORDUnet'93, Helsinki.

[14] Teet Saluvere, “*FPGA based CDMA Radio*” (Master thesis), Electronic System Design Lab, KTH, Nov. 93.

[15] *”The AnswerBook”*, Sun Microsystems, Inc.

[16] Andrew S. Tanenbaum, *“Modern Operating Systems”*, Prentice-Hall International Editions.

# APPENDIX A:
# A packet Relay between two MINTs

This appendix relates to an experiment done with the MINTs. It actually involves two pieces, both connected to the LAN and to each other. The purpose of the experiment is to relay packets from one MINT to the other. It would have also the opportunity to test the digital transceiver, but due to a lack of time and interface problems, it has been done.

## A.1. Relating to Ethernet

This section deals with the Ethernet environment. Anders Klemets (TSlab) has already written functions allowing the programmer to use the LANCE. I thought first that those functions would be integrated into the RadioNet Driver which will handle with both the LANCE (i.e., LANCE B) and the radio part, but actually it is deriving from the Mach Lance driver.

### A.1.1. A bit deeplier

`ether.c` provides use with several functions to access and control the LANCE chip. But using only three of them is quite sufficient for exploiting the network.

- The first one is the initialisation procedure, which loads the required parameters into the current lance chip. `init()` needs no arguments, but the variable ‘LANCEno’ must be first set up, accordingly to which chip is to be used.
- `int eread()` reads an incoming packet. In fact, it makes the transfer from the SRAM into the DRAM of the first packet stored in the ring buffer of the LANCE. eread() returns a pointer to the buffer in DRAM now containing the packet.
- `int ewrite(buffer, len)` performs the reverse operation, where ‘buffer’ contains the packet and ‘len’ is its length. ewrite() automatically initializes the source ethernet address field, imposes

a minimal length of 64bytes (length required for ethernet packets), or return an error if the lenght is greater than the maximum authorized.

There are many other functions, called internally by the previous three. If the reader likes to decrypt codes, I invite him to have a look at `ether.c`, but they will not be detailed in this document.

### A.1.2. Mini router example

Calling it "mini router" is a bit of an over statement. Indeed, I created a simple program that enables the communication between a MINT and nucmed20 (it could have been whatever other workstation, provided that it is connected to the same LAN (otherwise the communication must transit via a gateway, and it involves specific packet handshaking which hard to handle)).

This little program was firstly intended to handle manually entered ethernet packets, and thereby establishing a communication between a standalone program on MINT and a real client/server on workstation. It is based on UDP protocol (TCP protocol needs handshaking, and hence it would have resulted in a much more complicated standalone program).

## A.2. First experiment

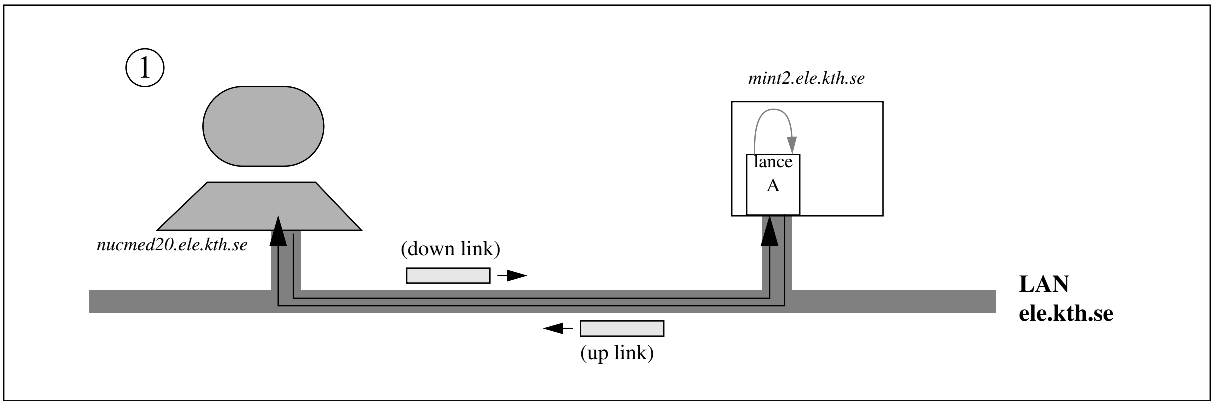



**Figure 23:** Trial of packet exchange between MINT and workstation

For the first experiment (Figure 23), communication in each direction is been tested separately. For the downlink direction, a client sends a message, using UDP (it does not matter which port is used, but the IP address must be correct), on the other hand, the MINT listens to the network during a certain time, stores in memory all the packets receives within this elapsed time, and then displays their content on its console.

By displaying the packets, one can watch how an ethernet packet is built. It might be interesting from an educational viewpoint.

For the uplink direction, a standalone program on MINT encloses a ASCII message into a UDP packet (with the appropriate UDP destination port), which is enclosed into an IP packet (with the right IP address of nucmed20), which is enclosed into a Ethernet packet (with the right Ethernet address of nucmed20), and sends it to nucmed20. While on the latter, a UDP server waits for any packet with the right UDP port. Then the message is displayed when received.

### A.2.1. Unexpected results

Now, here comes most exciting part. The client sends not one packet but a sequence of severals ones without stopping. I got the following result: the first five packets are correctly stored, but the next ones were missing, accordingly to a ratio of 3 missing over 4 packets sent.

This phenomena makes one to wonder what could be the cause. Is the CPU not fast and powerful enough? Is the code of the ethernet routines too heavy, that its execution time is too long? Well! Since I am not a specialist with regard to the motherboard features or the C-source code structure of the ethernet driver, I forwarded these questions to the Anders Klemets, but I haven't got any clear answer.

## A.3. Second experiment

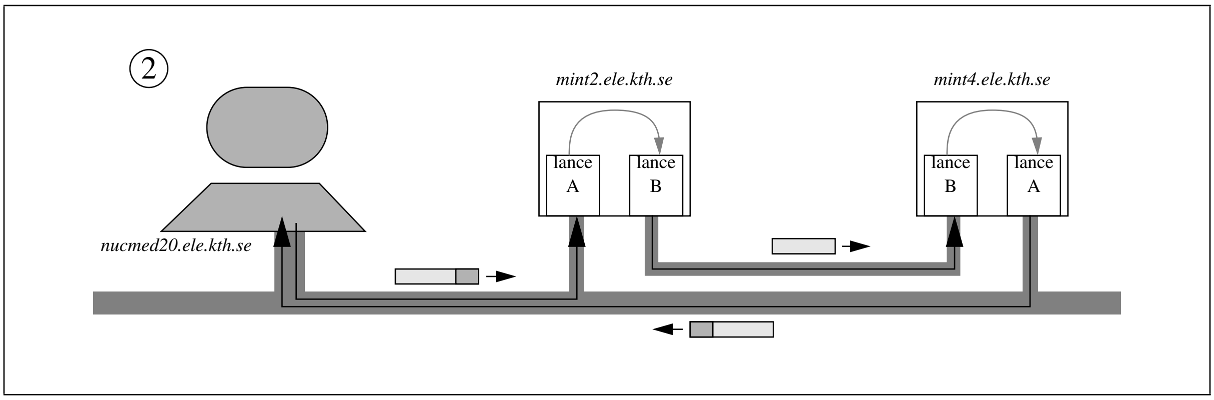



**Figure 24:** Relay between two Mints via an Ethernet cable

The second experiment is a bit more complicated. It involves two MINTs and a workstation, and it consists in relaying packets from one mint to the other, as shown in the Figure 24.

Each mint has its upper ethernet controller (LANCE A) connected to the LAN, since this is the one considered by default when downloading standalone programs, whereas the lower controller (LANCE B) will serve in the direct link between the MINTs. In anticipation of the future, the joining up of the two lance B, via a dedicated cable, symbolizes the radio link. In this manner, it is much easier to set up the experiment, because the tranceiving of the packets is not disturbed by any spurious noise inherent to a radio air channel, to fix potential bugs and to ensure a good running of the whole system, before interfacing the radio.

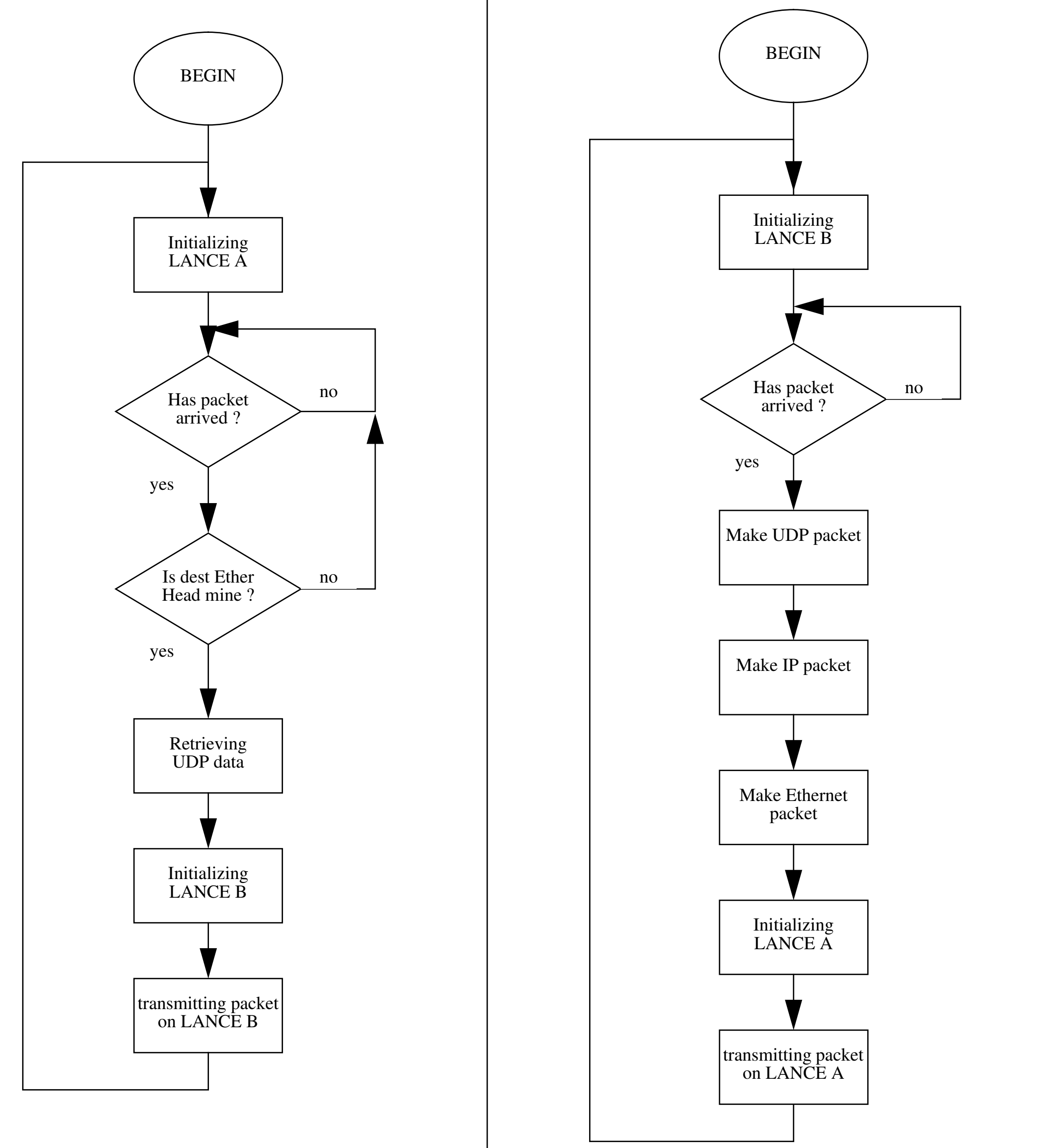
The method is as following: a process running on nucmed20 plays both the role of a client and a server. We would like to know how reliable, in terms of bit error rate, will be the radio link, by circulating test packets between the MINTs. In order to get the maximun degree of liberty, that is to say attribuating to it almost any kind of bit pattern), the packet in question should be encapsulated into a UDP packet, then sent from nucmed20 to the emitting MINT, where it is recovered and crudely put out via its LANCE B. On the other side, the second MINT receives this testpacket, encapsulates it into a new UDP packet, and sends back the whole to nucmed20. There, it will process in a correlation calculation, resulting in a bit error rate estimate.

### A.3.1. Software

Here is the brief explanation of the programs runningon the MINTs and nucmed20. There are thus three different programs created: one on the emitting MINT for getting packets from nucmed20 and sending them to the other MINT; one for the receiving MINT for getting packets from the first MINT and forwarding them to nucmed20; a client/server running on nucmed20 for generating those packets and checking the returning packets.

## A.3.2. Transmitting and receiving MINTs

Below (Figure 25) is a very simple functioning diagram on what happens in each MINT.




**Functioning diagram for the Emitting MINT** **Functioning diagram for the Receiving MINT**

**Figure 25:** Functioning diagrams for the MINTs involved in the relay.

#### A.3.2.1. Transmitting MINT

As shown in the diagram, the transmitting MINT starts running by listening to the network via its LANCE A. The Ethernet header of each incoming packet is check and only the ones which come from nucmed20 are kept, the others are discarded.

From every good packet, the UDP data of the UDP packet is retrieved and since these data already constitute a ethernet packet, there is nothing to change within.

Then LANCE B is initialized, and the packet is sent on LANCE B, that is to say, to the receiving MINT.

Due to ethernet routines provided by the ether.c file, especially some structures relating to the LANCE registers, it must be performed a separated and therefore repetitive initialization of the LANCE chips. This inceases the delay needed to forward the test packets, but it is almost insignificant.

The source code is called `mint_relay_emit.c` and is located in ~guerin/PROJECT/ .

#### A.3.2.2. Receiving MINT

As shown in the diagram, the receiveing MINT starts running by listening to the network via its LANCE B. Every packet arriving on this LANCE are accepted, since they are actually the test packets coming from the transmitting MINT.

Then this packet is put as data in a UDP packet. The latter is put as data in an IP packet. The latter is then put as data in a Ethernet packet, and the whole is sent to nucmed20.

The source code is called `mint_relay_recep.c` and is located in ~guerin/PROJECT/ .

### A.3.3. The relay checker program

The relay checker program plays two roles at the same time. On one hand, it acts as a client, sending a packet to the transmitting MINT. On the other hand, its acts as a server, waiting for the incoming packets from the receiving MINT. Client and server are based on UDP, it makes therefore easy the communication between nucvmed20 and the MINTs, since the latter are not running operating system, but standalone programs.

Running over the diagram, the client is first initialized, then the server in turn, with port equal to 6000. This is a convention, since the receiving MINT will forward its packet in a UDP datagram with 6000 as port number.

The program alternatively behaves as a client and as a server. Thus, after having sent a packet to the transmitting MINT, the program immediately changes state to server, and waits for the returning packet from the receiving MINT. A timer is set up. After a certain timeout, the packet is considered as lost. Hence a new one is sent. Number of lost packets is also recorded.

When a packet is received, then it is compared with the original one sent before, resulting in some statistics. Figure 27 shows what kind of results the program supplies.

Syntax of the launching of the relay_checker program is :

```
relay_checker mint[TX_MINT's_number] [port] [number_of_packets]
```

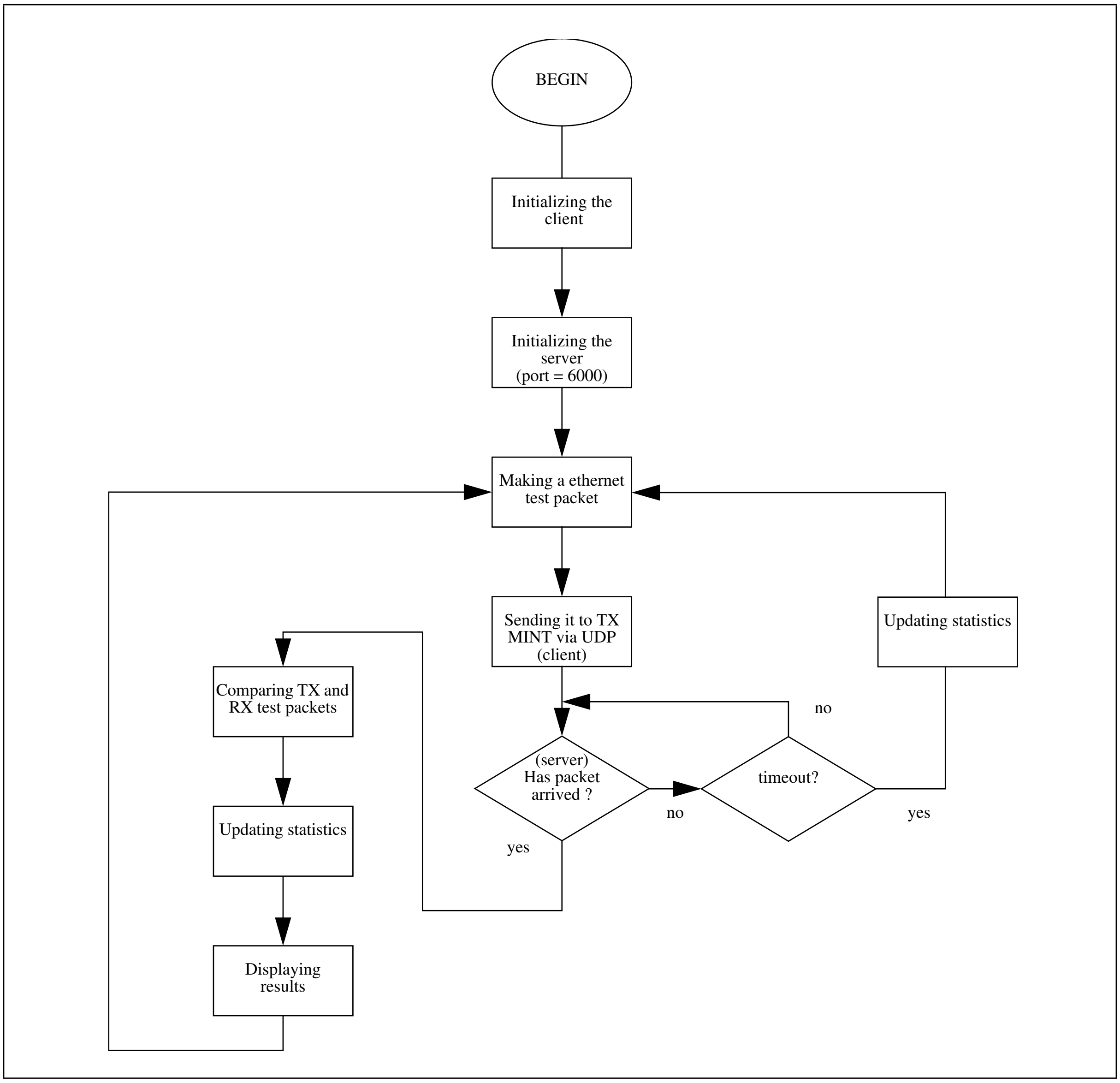



**Figure 26:** Functioning diagram of the relay checker program.

The relay_checker program seems to work fine. The experiment has been made with MINT4 as transmitting MINT, and MINT3 as receiver.

### A.3.4. Hooking on the digital transceiver

We have hooked on the digital transceiver on the LANCE chip. But unfortunately the experiment failed. There might be some unforeseen problems of handshaking with the LANCE chip. Hence the Xilinx must be reprogrammed for fixing the bug.

```
Packet number: 0

Bit Error Rate (for current packet): 0.00 %
Current packet is good : YES

Number of wrong bits: 0
Total number of bits: 0
Bit Error Rate Average: 0.00 %

Number of wrong packets: 0
Total number of packets: 0
Packet Error Rate Average: 0.00 %
Total discarded packets: 1
 timer = 10000

Server: Wait state...
```

**Figure 27:** results displayed by relay_checker.

## A.4. Third experiment planned

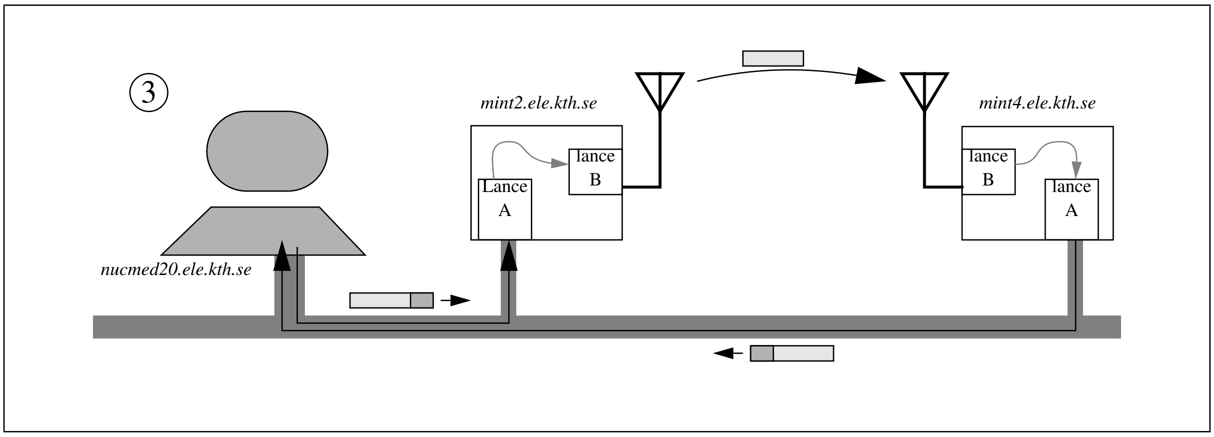



**Figure 28:** Relay between two Mints via the radio interface

Since the analog transceiver is not yet ready (some new instruments are required for adjusting it), the current experiment cannot be set up at the present.

However, due to the Law, it is not permitted to emit radiowaves at 2GHz in the air. A range of 2,4GHz is open within the radio experiment will be continued. Thus the current analog transceiver is now obsolete, and a new one from the market is expected.

# APPENDIX B:

## Notes on how to use the MINT software development environment

## Internal documentation

**Anders Klemets, 940120**

A version of the GNU C compiler has been prepared for use with the MINT. It runs on Sun Sparcstations and generates m68k instructions. It might be used to invoke the GNU m68k assembler and GNU linker. The latter might not be always be desirable.

When compiling a program, the default procedure is that the compiler will automatically invoke the linker. The first routine to be run in the program is called **`start`**, and it normally resides in an object file called **`crt0.o`**. The GNU C compiler will try to link the programs with `crt0.o` and the library `libgcc.a.` Neither of those are available in the current development environment however.

The current workaround is to link the programs by calling the linker explicitly, supplying the names of any libraries if necessary.

When the GNU C compiler encounters a function called `main(),` it will insert a call to a function `_main` which resides in `crt0.o`. The primary function of **`_main`** is to initialize memory management for classes in C++. For C code, it performs no function. However, the call is still there. Since `crt0.o` is not available, a way of eliminating the reference to this function is to name the starting function something else than main, and tell the linker to use that name as the entry point.

Example:

A simple C program, `hello.c`, for the MINT that uses the printf function in the EPROM monitor could look like this:

```
#include "mintromvec.h"
start()
{
      (*romp->v_printf)("Hello, world\n");
```

```
}
```

The `mintromvec.h` file is in the directory:

```
/afs/it.kth.se/home/tslab/klemets/MINT/newmintmon
```

and it defines useful entry points to routines in the EPROM monitor.

The GNU tools are in the directory:

```
afs/it.kth.se/home/tslab/klemets/MINT/bin
```

Assuming that this directory is first in your PATH, you would compile the program in the following manner:

```
gcc -c -O2 -nostdinc -I. -msoft-float hello.c
```

This tells the compiler not to use any of the standard include files in `/usr/include` (since they are specific to SunOS.) Rather, it will pick any include files from the current directory. The last option is a directive to the assembler, telling it not to emit any M68851 instructions. Since the MINT does not have a floating point unit, it will crash if it encounters such an instruction. This option is necessary even if the program does not make use of floating point, because the compiler will sometimes decide to use such an instruction for efficiency reasons. Any real floating point operations will be replaced by calls to software library routines that are currently not available.

If the C program contains explicit MC68030 assembly code instructions, it might be necessary to tell the assembler about this, since it is by default expecting only MC68020 instructions. This can be done by giving the compiler the option “`-Wa,-mc68030`”.

To link the object file `hello.o` into an executable binary named hello, one would do as follows:

```
ld -o hello -N -e _start -Ttext 40000000 hello.o
```

The “`-N`” flag tells the linker that the data segment should directly follow the text segment, rather than being allocated at a page boundary. The MINT monitor currently only correctly handles linked in this fashion. The “`-e _start`” flag informs the linker about the name of the first function to be called in the program.

The MINT monitor will by default load all programs to the start of the RAM area. The programs must be linked for execution at this address. This is done with the “`-Ttext 40000000`” flag.

The program is now ready for downloading to the MINT. The current TFTP downloading area is on a machine called ***nucmed20.ele.kth.se***. The file can be copied there in the following fashion:

```
rcp hello nucmed20.ele.kth.se: ~maguire/tftpdir/hello
```

**Usage of the GNU debugger:**

A version of the *GNU debugger*, **gdb**, is provided for use with the MINT. When debugging, gdb gives more verbose results if the program has been compiled with the "`-g`" flag.

Assuming the same scenario as in the example above, the debugger can be invoked in the following fashion:

```
gdb -b 9600 hello
```

The option “`-b 9600`” tells the debugger that serial line communication will use 9600 bauds. If the

MINT is connected to `/dev/ttya` on the machine that is running the debugger, the debugger would be activated with the GDB command

```
target remote /dev/ttya
```

The debugger will now try to contact the MINT, which it will only succeed with if or when the MINT has gotten an exception, and the MINT is initialized for remote debugging.

To perform a reset of the MINT, one must set the **status register** into supervisor mode with the interrupt stack enabled, and jump to the restart routine of the current monitor. This can be done by invoking the user defined command “`mint-restart`” which is defined in the file `~klemets/.gdbinit`. If such a file is present in your home directory, it will be read automatically by gdb on startup.

For details on how to use GDB, see the online documentation.

**Using the MINT monitor:**

The most important command is perhaps the **`Boot`** command. When issued with no argument, the MINT will use BOOTP to find out the name of a file to download with TFTP. Our environment is currently setup so that the default file name will be “`mintbootfile`”. An optional argument is a file name to override the default name.

The MINT will load the file to the address specified by the environment variable **`Loadstart`**. It defaults to 0x40000000. If the file has an `a.out` header, the MINT will set the environment variable **`Loadentry`** to the entry point specified in that header. Otherwise, it will use the value of `Loadstart` as the entry point.

When the download is completed, the MINT will start executing the file immediately, unless the environment variable Runflag is set to zero. Before calling the address specified by `Loadstart`, however, it will put the value of **`Bootflag`** as an initial argument on the stack.

To start the program manually from the monitor, one can use the command **GO** followed by an optional entry point address. The default address used by `GO` is given by the environment variable `Loadentry`.

To initialize remote debugging, one must execute the **`Trace`** command. The command immediately generates a breakpoint, so execution will not continue until instructed by GDB to do so.

Environment variables can be listed with the **`Printenv`** command and changed using the **`Setenv`** command.