united nations
educational, scientific
and cultural
organization

international atomic
energy agency

the
abdus salam
international centre for theoretical physics

The Sixth College on Microprocessor-Based Real-Time Systems in Physics

9 October - 3 November 2000

LECTURE NOTES

Volume II

MIRAMARE-TRIESTE

October 2000

Editors:
Abhaya S. Induruwa
Catharinus Verkerk

These are preliminary lecture notes intended only for distribution to participants

strada costiera, 11 - 34014 trieste italy - tel. +39 040 2240111 fax +39 040 224163 - sci_info@ictp.trieste.it - www.ictp.trieste.it

Acknowledgements

The Sixth College on Microprocessor-based Real-time Systems in Physics is the result of the dedicated contributions by a large number of people associated with the planning, organising and running of these Colleges. The appropriateness of the theme and the timeliness of the topics covered are evident from the demand for participation and the enthusiasm of the participants. Laboratory work forms an important and integral part of the College and having access to a range of purpose built hardware systems provides an excellent opportunity for the participants to have hands-on experience and serves to reinforce the theoretical aspects taught in the College.

We sincerely thank Professor Miguel A. Virasoro, Director of the Abdus Salam International Centre for Theoretical Physics for his interest in the course and the laboratory work, and for his support. Special thanks are due to late Professor Ines Wesley-Tanaskovic and Professor Luciano Bertocchi for their encouragement, support and long standing association with the College. We are grateful to the United Nations University and to the Abdus Salam ICTP for their respective financial contributions.

The organisation and smooth running of an activity of this scale requires support and assistance, both before and during the College. Our thanks go in particular to Italo Birri and Mohammed Iqbal of the Microprocessor Laboratory, Marco Zorzini of the Scientific Computing Section and the College Secretary Ms Stanka Tanaskovic for their continued assistance to the College. We also thank all the staff of the Abdus Salam ICTP who work behind the scenes to make the running of the College possible.

We wish to acknowledge the contributions of the lecturers and tutors who in addition to preparing and making presentations, contributed by running the laboratory and giving useful and friendly assistance to the participants. Our sincere thanks go to Imtiaz Ahmed, Chu Suan Ang, Paul Bartholdi, Razaq Ijaduola, Carlos Kavka, Anton Lavrentev, Ulrich Raich, Pablo Santamarina, Olexiy Tykhomyrov and Jim Wetherilt. A number of them made particularly important contributions in preparing enhanced hardware and software for this course.

The hard work and dedication of the participants made the interaction with them an enriching experience for the teaching staff. We are confident that they will return to their respective home countries and institutions equipped with new knowledge and experience in the subject matter taught in the College. We wish them success and full satisfaction in their professional life.

Abhaya S. Induruwa,
Catharinus Verkerk,
Directors of the College,
Trieste, October 2000.

Contents

Software Design

Sixth College on Microprocessor-based Real-time Systems in Physics

Abdus Salam ICTP, Trieste, October 9 — November 3, 2000

Paul Bartholdi and Denis Mégevand
Geneva Observatory
51, chemin des Maillettes
CH-1290 Sauverny
Switzerland

e-mail: Paul.Bartholdi@obs.unige.ch
Denis.Megevand@obs.unige.ch
URL: http://obswww.unige.ch/

Abstract

In this chapter, we will look at various topics concerning Software Design, from program documentation to very specific aspects of real-time. It contains also an introduction to shell programming and the use of various Unix tools.

1 Documentation

Some program are used once and never used again.

However most programs

- will be used many times;
- will be changed, upgraded;
- will go to other users;
- will contain undetected errors.

Maintaining, upgrading, using again, debugging, cost more time and money **after** a program is "finished" than **before**.

Good programming + Good documentation = lower total cost

1.1 Various Types of Documentation

Documentation will serve many goals, and be read by many different users.

It should be

- Useful, that is concise and readable;
- Consistent, any change should be time stamped;
- Maintainable, indexes and cross-references should be produced automatically;
- Up-to-date, in parallel with the codes.

Here is a *short* list of various situations:

1. Source Code Comments
2. Maintenance Manual
3. User's Guide (Tutorial)
4. Reference Manual

5. Reference Card

6. Administrator's Guide

7. Teaching Notes

8. General Index

Depending on the importance of the system, some of these points may be ignored, or be part of others. For large project, they should be independent documents.

1.2 Internal Documentation to the Code

Goal: Document each module at the local level for the programmer. It should be short and informative (not paraphrase), easily readable on a screen.

Header

- name + descriptive title
- programmer's name and affiliation
- date and version of revisions with changes
- short description of what it does and how
- input expected, limits
- output produced
- error conditions, special cases
- other modules called

In-line comments

- should help to follow execution
- break into sub-sections
- indent if useful
- use meaningful names
- do not duplicate code

1.3 Maintenance Manual – Programme Logic

Goal: Present a global view of the product to a programmer, at the functional and structural level.

- table of contents
- program purpose, what it does and how
- names and purpose of principal modules
- cross-reference between modules
- name and purpose of main variables
- flow chart of main activities, dynamical behavior
- debugging aids, how to use them
- interface for new modules
- index

It should complement the internal documentation (not duplicate it)

Look at your program from above, think about it as an outsider.

1.4 User's Guide

Goal: Should help the **user**, present him a global overview of the product and how to use it!

- Table of contents
- how to use the documentation
- how to contact author/maintainer (E-Mail) addresses, phones etc
- acknowledgments
- program name(s)
- what it does (briefly)
- explanation of the main notions and concepts used

- references (how it does it)
- how to start and stop the programs
- input expected, controls available
- unusual conditions, errors, limitations
- sample run with input, output and comments
- index

1.5 Reference Manual

Goal: Present an exhaustive and formal description for the various elements of the product.

- table of contents
- table of function, with a short description
- reference pages: list of all functions in a standard form, with a complete description similar to the module headers
- table of global variables with complete description and cross-indexing
- glossary for all specific words
- table of errors
- table of drivers
- annexes
- index

1.6 Reference Card

Goal: Single sheet with formal references for rapid consultation.

List of all commands, with their syntax, ordered by subject. Should be produced automatically from the Reference Manual and User's Guide.

1.7 Administrator's Guide

Goal: Easy installation and maintenance of the product in various environments.

- Table of contents
- minimum configuration and necessary associated products
- installation
- documentation production
- updates
- des-installation procedure
- list of supported machines and configurations
- list of attached files
- table of variables
- index

1.8 Teaching Manual, Primer

Goal: Easier understanding and learning of the product.

Step by step introduction of the various concepts and commands of the system, with examples, exercises, answers etc

It will depend considerably on the product. It could be part of the User's Guide.

As a rule, make suggestions for serial execution, avoid to force the reader on a given path, let him try whatever he wants, put data files at his disposition. In my opinion, many *Introduction to ...* are far too restrictive in this sense.

1.9 General Index

Goal: Find information anywhere in the documentation.

Should be prepared at the same time as the various documents.

1.10 Reference Page Contents

Here is a quite exhaustive list of fields for a reference page:

name		
list of commands	linkages to other products	
short description	long description	remarks
synopsis	(BNF) syntax	return value(s)
options	global variables	context
input parameters	output parameters	optional parameters
author	version	date
examples	keywords	optional keywords
known bugs	limitations	cross-references
errors	level of errors	bibliography
algorithms	precision	complexity
input files	library files	external references
temporary files	used files	modified files

1.11 Literate Programming

Knuth, while writing his set of books on TEX , that is the TEX text processing system, in parallel with the design of the product, has build a new concept for the documentation of codes, where the text around the code is the main object of attention.

The code, written in the middle of the documentation, can be extracted automatically and passed untouched to the compiler. It is not intended for human reading, even less for editing, this has to be done in the documentation file.

The printed documentation produce code listing that is particularly easy to read.

`cweb` is well adapted to `C` programming.

Here is a small extract from a *cweb file*:

```
@ Most \.{CWEB} programs share a common structure.
It's probably a good idea to state the overall structure
explicitly at the outset, even though the various parts
could all be introduced in unnamed sections of the code
if we wanted to add them piecemeal.

Here, then, is an overview of the file \.{wc.c} that is defined
```

```
by this \.{CWEB} program \.{wc.w}:
\index{c!cweb example}

@c
@<Header files to include@>@/
@<Global variables@>@/
@<Functions@>@/
@<The main program@>

@ We must include the standard I/O definitions, since we want
to send formatted output to |stdout| and |stderr|.

@<Header files...@>=
#include <stdio.h>

@  The |status| variable will tell the operating system if the
run was successful or not, and |prog_name| is used in case
there's an error message to be printed.

@d OK 0 /* |status| code for successful run */
@d usage_error 1 /* |status| code for improper syntax */
@d cannot_open_file 2 /* |status| code for file access error */

@<Global variables@>=
int status=OK; /* exit status of command, initially |OK| */
char *prog_name; /* who we are */
```

From this code, two files can be extracted, a `.tex` for the printed document, and a `.c` file for the compiler.

Here is the corresponding extract in printed form:

2 AN EXAMPLE OF CWEB WC §1

2. Most CWEB programs share a common structure. It's probably a good idea to state the overall structure explicitly at the outset, even though the various parts could all be introduced in unnamed sections of the code if we wanted to add them piecemeal.

Here, then, is an overview of the file `wc.c` that is defined by this CWEB program `wc.w`:

⟨ Header files to include 3 ⟩
⟨ Global variables 4 ⟩
⟨ Functions 20 ⟩
⟨ The main program 5 ⟩

3. We must include the standard I/O definitions, since we want to send formatted output to *stdout* and *stderr*.

⟨ Header files to include 3 ⟩ ≡
```
#include <stdio.h>
```
This code is used in section 2.

4. The *status* variable will tell the operating system if the run was successful or not, and *prog_name* is used in case there's an error message to be printed.

```
#define OK 0 /* status code for successful run */
#define usage_error 1 /* status code for improper syntax */
#define cannot_open_file 2 /* status code for file access error */
```
⟨ Global variables 4 ⟩ ≡
```
int status = OK; /* exit status of command, initially OK */
char *prog_name; /* who we are */
```
See also section 14.
This code is used in section 2.

and the C code:

```
#define OK 0
#define usage_error 1
#define cannot_open_file 2 \

#define READ_ONLY 0 \

#define buf_size BUFSIZ \

#define print_count(n)printf("%8ld",n) \

/*2:*/
#line 30 "wc.w"

/*3:*/
#line 39 "wc.w"

#include <stdio.h>

/*:3*/
#line 31 "wc.w"

/*4:*/
#line 50 "wc.w"
```

```
int status= OK;
char*prog_name;

/*:4*//*14:*/
#line 150 "wc.w"

long tot_word_count,tot_line_count,tot_char_count;
```

2 Quality Assurance

The goal of Quality Assurance is to systematize the process of verification and validation:

- Verification: *Are we building the the product right?*
- Validation: *Are we building the right product?*

2.1 Standards, Practices and Conventions

Will depend on the environment (ex. programming language). It should be

- generally agreed on,
- then followed by every one.

In general:

- The code should reflect the problem, not the solution;
- the methods used has to be predictable;
- the style has to be consistent throughout the program;
- special features of the programming language or hardware environment should be used very carefully, or avoided altogether;
- the program should be written for a reader as much as for a computer.

2.2 Software Quality Factors

Correctness does it satisfy its specifications and fulfill the objectives?
Does it do what I want?

Reliability does it perform its intended functions?
Does it do it accurately all the time?

Efficiency Amount of resources required
Will it run on a given hardware as well it can?

Security controlled access to the code and data
Is it secure?

Usability Effort required to learn, operate, upgrade the code
Can I run it in the long term?

Maintainability Effort required to locate and fix errors in the code
Can I fix it?

Flexibility Effort required to modify an operational program
Can I upgrade it?

Testability Effort required to test fully a program
Can I test/trust it?

Portability Effort required to transfer the program to another system
Will I be able to change my OS or hardware?

Re-usability Reuse of parts of a program in another application
Can I reuse some of my work?

Interoperability Effort required to couple one system to another
Can I interface my program to another system?

2.3 Review and Audits

An innocent view on your work can be very useful to

- uncover errors in function, logic or implementation;
- verify that it meets the requirements;
- agree with accepted standards;

- achieve consistency with other works;
- ease management.

A technical review should take place each time a module of a reasonable size has been completed, or results from some extensive test exist.

The review team should be small: 2–3 persons. E-Mail has the advantage that everything will be documented.

Imaginary checklist for a review:

1. System engineering: definitions, interfaces, performances, limitations, consistency, alternative solutions
2. Project planning: budgets, deadlines, schedules
3. Software requirements
4. Software design: modularity, functional dependencies, interfaces, data structures, algorithms, exception handling, dependencies, documentation, maintainability
5. Testing: identification of test phases, resources, tools, record keeping, error handling, performance, tolerance
6. Maintenance: side effects, documentation, change evaluation and approval ...

2.4 Testing

1. Executing a program with the intent of finding an error
2. Successful test: one that uncovers an as-yet undiscovered error, with minimum amount of time and effort.
3. Testing cannot prove the absence of defects

2.4.1 Black Box Testing

Using only the specified functions and input/output description, demonstrate that each function is fully operational in all circumstances, and has no defective side effects.

Some questions:

- Which functions are tested?
- Which classes of inputs are used?
- Is the system sensitive to input values? to user errors?
- What data rates and volumes can be accepted?
- How does it affect system operations?

2.4.2 White Box Testing

Using not only the external specifications, but also the internal working of the modules, demonstrate that it does work in the expected way, exercising all internal components.

All procedural details should be closely examined.

Exhaustive testing is generally impossible for large modules.

Some questions:

- Do the data structures maintain their integrity during the execution?
- Which paths are exercised, which are not?
- How are "special paths" executed?
- How is error handling executed?
- How does the system react to stress, deliberate attacks?

2.5 Defensive Programming in the Laboratory

The previous section is mainly valid for large projects, in particular when a team of many people is involved with external requirements.

Here are a few hints that can be applied during the exercises in the laboratory:

- Try to explain clearly what you are doing to your colleague. It is not far from a psychiatric experience. You will find your own errors that way.
- Do not trust anything!
  - Print the status for all file operations,

  - When you open a file, verify that it exists,
  - When you read a record, check that you are not at `eof`, , check that the data are valid
  - When you write a record, check that you have write permissions,
  - When you do some complex calculation, check that the results are in the right order,
  - If some input data must be on a given range, check its bounds,
  - If anything may last more than a few seconds, print some flags or indications,
  - When your program has terminated, check the size and contents of every file involved (it may not be a bad idea to print inside the program a summary of all written files with their length).
- Keep a backup of all important (a constantly changing concept) files
- Use the facilities of UNIX, like `make`, `grep`, `tee`, `diff` ...

2.6 Debugging

Almost all programmes contain errors (= bugs in relay). You can help the detection of them:

- add guards while coding,
- prepare simulated input, first simple (easy to trace by hand), then more complex (difficult),
- Debug each module alone, then in small integration,
- chose critical points where you know what you should get if previous step are correct,
- advance by small steps,
  - from input forward,
  - from output backward,
- analyze wrong results to see what/where this value comes from,
- try all (very) improbable cases.

Rules :

- if some thing can go wrong, it will !
- if an error can be damaging, it will !
- if it is very improbable, it will still exist !

2.7 Murphy's Laws

Murphy was an American engineer whose pessimism paid — his famous law, "If anything can go wrong, it will," should remain a model of conservative system design. Many scientists were inspired by him (as seen from the following):

- Any given program, when running, is obsolete.
- Any given program costs more and takes longer to develop.
- If any program is useful, it will have to be changed.
- If a program is useless, it will have to be documented.
- Any given program will expand to fill all available memory.
- The value of a program is proportional to the weight of its output.
- Program complexity grows until it exceeds the capability of the programmer who must maintain it.
- If the input editor has been designed to reject all bad input, an ingenious idiot will discover a method to get bad data past it.
- Make it possible for programmers to write in English and you will find the programmers cannot write in English.
- *Bolub's Fourth Law of Computerdom*: Project teams detest weekly progress reporting because it so vividly manifests their lack of progress.
- *The Briggs/Chase Law of Program Development*: To determine how long it will take to write and debug a program, take your best estimate, multiply that by two, add one, and convert to the next higher units.

- Computers are unreliable, but humans are even more unreliable.
- Any system which depends on human reliability is unreliable.
- A carelessly planned project takes three times longer to complete than expected; A carefully planned project takes only twice as long.
- *Grosch's Law*: Computing power increases as the square of the cost.
- *Putt's-Brook's Law*: Adding manpower to a late software project only makes it later.
- *Shaw's Principle*: Build a system that even a fool can use, and only a fool will want to use it.
- *Weinberg's First Law*: If builders built buildings the way programmers wrote programs, then the first woodpecker that came along would destroy civilization.
- *Weinberg's Second Law*: A computer can make more mistakes in 2 seconds than 50 mathematicians in 200 years.
- Efforts in improving a program's "user friendliness" invariably lead to work in improving user's "computer literacy".
- "But I only changed one line and it won't affect anything!"

3 UNIX Tools

The goal of this section is not to introduce UNIX *per se*, but to show how some UNIX tools can help in the production of good software.

3.1 UNIX as a Programming Language

Forty years ago, much programming was done in assembler, if not with wires. Then higher level languages like Fortran, C, Cobol etc. permitted the development of codes more or less independent of the hardware and operating system, that is much easier to read, that can be developed in reusable modules. Yet, the basic building blocks are still relatively low level instructions that are combined into higher and higher modules to form a single large program, where the modules are 'hard' interconnected.

The pipes and redirections, the very large number of simple standard tools available in UNIX and the facilities to build newer tools in the same spirit, and then interconnect them into streams and shells, make UNIX an ideal interactive programming environment.

3.2 Pipes and Redirections

Pipes permit to write small modules dedicated to simple tasks, and to interconnect them through standard input/output. Such modules are much simpler to develop and test individually, while the pipe checks for the interfaces. When fully tested, these modules can be put together in larger ones. The number of successive pipes is practically unlimited.

```
ls -l | less
```

Redirection is a good way to have all input data (including test ones) in files that can be text-edited. Output redirection, in particular using `tee`, builds sets of files against which future version's output can be compared (use `diff` for that).

```
prog < test.data > test.results
```

3.3 Aliases and functions

Every complex command that may be used regularly could be aliased into a simple mnemonic name :

```
alias mnemonic='equivalent string'
```

The exact form of *alias* depends on the shell used. Here I have adopted the *bash* form.

Many examples of aliases are given below.

Aliasing into usual UNIX command should be carefully avoided if the use of the original version can be dangerous when the aliased one is expected.

```
alias rm='/bin/rm -i'
```

is a typical example. In another environment, `rm` will not ask you for confirmation when you expected it.

Inversely, tools that require a mode, should specify so: use "`~/bin/rm -f`" and not "`rm`".

But:

```
alias ls='/bin/ls -CF'
```

is a perfectly acceptable one.

Notice that, except in *csh* and *tcsh*, it is not possible to pass parameters to an alias. In most cases, but not with *csh* or *tcsh*, a function can replace an alias. In *ksh* and *bash*, a function is defined by:

```
function name() { commands }
```

Inside the function body (*commands*), parameters are referred by their positions in the calling statement, that is `$1` for the first parameter, and so on. See the examples bellow, page 20.

The keyword `function` itself is optional in *bash*.

Another method, probably safer than an alias and shell independent, is to have a reserved `~/bin` directory, and a corresponding scripts for each alias:

Put in your `.login` file a command:

```
PATH=~/bin:$PATH
```

and then:

```
echo "/bin/rm -i" > ~/bin/rmi
```

This will create a file, usually called a script (do not forget to make it executable), instead of the `alias` command. Typing `rmi` will execute that file.

3.4 Searching Tools

`grep` is a very powerful tool to do all sorts of searches and filters, in particular as part of a pipe stream. It looks for all occurrences of a pattern inside a set of files, and print the corresponding lines. It has many options (see the man pages), among them three avec very useful: `-h` suppress the prefixing of filenames on output, `-i` ignore case distinctions, and `-v` to select non-matching lines.

For example, finding all files that use `stdio.h`:

```
grep stdio.h *.c *.h   or              *.[ch]
```

Printing error messages only, with full output into a file:

```
test < test.data | tee test.res | egrep -i error
```

`grep` can also be used very effectively to "search" through a "data base". Suppose that you have a file with names, phone numbers and remarks, more

or less in free form, another with hints on different subjects concerning your programs etc.

Then you can define the following aliases:

```
function help(){egrep -ih $1 ~/.help ./.help}
function tel()={egrep -ih \!* ~/.phones /share/phones}
```

`help` *xxx* will print all lines from `~/.help` and `./.help` that contains the string "*xxx*".

and `tel` *abc* will do the same for the phone files. With `tel` or `help` you can look for anything, not necessarily name or first name, but also for partial phone numbers etc. `tel 0039` will list all entries in Italy, while `tel rinus` will find our director's one.

Here is another application, to list only the files that have been modified this day in the current directory:

```
alias today 'set TODAY=`date +"%h %d"`; ls -al | egrep $TODAY'
```

A similar command to see all files modified this day, in alphabetical order:

```
alias Today 'find .  -ctime 0 -print | sort'
```

3.5 Looking for parts of a file

`head` and `tail` can be used to select only a few useful lines:

To see only the first line of a set of subroutines:

```
head -1 *.c
```

To see only the largest (or the most recently modified) files:

```
ls -l | sort +4n -5 | tail -16
ls -rtl | tail
```

Long output could also be piped into `more` (or `less`, `most`).

`uniq` can be combined efficiently with `sort` to find "*words*" that are rarely used, and so possibly wrong (`sort -u` would do the same).

3.6 Stream Editor: `sed` and `gawk`

`sed` is a very simple but powerful editor that can be inserted in the middle of a stream. `gawk` can be used in the same way for very complex text manipulation. The simplest using of `sed` looks like:

```
... | sed -e 's/abc/efgh/g' | ...
```

It will simply replace everywhere the pattern "*abc*" with "*efgh*". The first character after s will be used as the separator, it is not necessarily a /.

Here is a more elaborated example:

```
#!/bin/csh
# add a new user (board), in group 501
# and set the same encrypted passwd as in other machines.
\index{password}

/usr/sbin/adduser -g501 board
set today=`date +'%Y%m%d:%H%M'`
cd /etc
cp passwd passwd_$today
sed -e 's/board\:\!/board\:7s0kry.rn.dco/' passwd_$today > passwd
```

gawk is a very complex program for which the manual is more than 300 pages, but it can also be used very effectively as a single line program. In the simplest case, gawk is used to reorder the *fields* or choose among the fields in every lines. For example:

```
awk '{ print $3 $7}' file
```

will leave only the third and seventh fields.

Here is little more complex example:

```
#!/bin/tcsh
# lock all users with no password
# put a ! in the second field of the file /etc/passwd
# if that field is empty.

cd /etc
set today=`date +'%Y%m%d:%H%M'`
cp passwd passwd_${today}
/bin/awk ' BEGIN {FS=":"; OFS=":"} \
      { if(NF>=7) { if($2=="") $2="!";
                    print $0 } } ' passwd_${today} > passwd
```

3.7 Character conversion using `tr`

`tr` is intended to do all sorts of character conversion, including special characters, and optionally to replace strings of the same character by a single one.

Normally, two strings of characters are given as parameters to `tr`. `tr` will replace all occurances of characters in the first string by the corresponding character in the second string.

If the option `-d` is given, then the characters from the first string are deleted.

If the option `-s` is given, strings of the same characters are replaced by a single one.

Special characters are represented with \ and a character.

\a	ctrl G	bell
\f	ctrl L	form feed
\n	ctrl J	new line
\r	ctrl M	carriage return
\t	ctrl I	tab
\v	ctrl K	vertical tab
\nnn	octal value	

It is also possible to give a *class* of characters instead of a string. In this case, the *string* should have the form `'[:`*class*`:]'` , where *class* is one of `alnum alpha digit cntrl blank lower upper punct`.

Here are a few examples of using `tr`. The first three are equivalent. The last two can be very usefull if you have a mix of PC, Mac and Unix machines.

```
cat myfiile.c | tr ABC...Z abcd...z > ...
cat myfiile.c | tr A-Z a-z > ...
cat myfiile.c | tr '[:upper:]' '[:lower:]' > ...

tr '\r' ''   < dos_file > unix_file
tr '\r' '\n' < mac_file > unix_file
```

3.8 Use of the `history`

`tcsh` keeps a log of the last *n* commands. *n* is defined with the command `set history=`*n* in the file `.cshrcr` or `.tcshrc` .

This log can be used in the following ways:

`history`	prints (on screen) the list of the last n commands executed,
`fc`	repeats the last command,
`fc` *n*	repeats a given command,
`fc` *-n*	repeats a given relative command,
`fc` *abc*	repeats the last command starting with the same letters,
`fc -s`*old/new*`/g`	repeats the last command with editing (substitution [+global]),

Part of the repeated command can be re-used by the following *word designators*:

0	word 0 (= command)
n	n^{th} word
^	first word
$	last word
$m - n$	words m through n
-	words 0 through but last
*	words 1 through last
%	word matched by the string

The word designators can be modified by appending a modifier to the specifier: <*specifier*>:<*modifier*>

r	root of the file name
e	extension of the file name
h	head of the path (but last comp)
t	tail of the path
s/old/new/	substitution
g	global (comes before s)
p	print but no execution
q	quote words
u	make first lower case letter upper
l	make first upper letter case letter lower

3.9 Command/file name completion

After you have typed a few letters of a command or file name:

<*TAB*> will complete it if possible and unique,

<*ctrl*>d will list all possible completions.

3.9.1 Finding something in a large directory tree – `find`

`find` allows to search through any directory tree, looking for matching file names or files modified before or after a given date for example, and then

execute any sort of command, like printing file name with full path, deleting, executing a `grep` on them etc.

`find` has many options, but we will see only four. Refer to the man pages for all other ones.

`find . -name` <*file_name, possibly with wild card*> `-print`

`find . -ctime` <*n*> `-exec` <*command*>

In the command, use {} to replace the file name, ending the command with \;

The first parameter (".") is the starting point, root of the directory we are searching.

The second is the selection criteria, according to file names or times.

Then comes the execution for all files that match the selection criteria.

In facts, many selection criteria and many executions can be used simultaneously. Selection criteria can possibly be joined by `or` or `and` and `not`.

`find` without any execution part simply produces a list of the selected files on the standard output. This can be used with `cpio` to copy directories recursively.

Examples:

1. Remove all core files, printing their full path:

   ```
   find . -name core -exec rm -f {} \;
   ```

2. List all files created today in any subdirectory:

   ```
   find . -ctime 0 -print
   ```

3. Search for use of `stdio.h` in all c files:

   ```
   find . -name \*\.c -exec grep stdio {} \;
   ```

4. copy a directory tree on an other place:

   `find` <*source directory*> `| cpio -dpm` <*destination directory*>

3.10 Executing just What is Necessary, using `make`

When a project gets larger, it becomes more and more difficult to track which compilations, link and execution are necessary.

`make` permits to do such operations automatically, based on declared dependencies and last modification time. The set of commands executed in each case is completely open and not restricted in any way to compilation or link. Further, the dependencies can be given explicitly, supplied by compilers like `gcc -M`, or even assumed implicitly by `make` itself in many cases from the file suffixes.

The use of implicit assumptions make it faster to write but more difficult to read the dependency file.

The general form of a dependency file (usually named `Makefile`) is the following:

```
target(s): dependencies
<TAB>      commands to produce the target(s)
```

`make` without a parameter will check the first target for dependencies, and then recursively through the file. If a target is older than a dependency, then the corresponding commands are executed.

If `make` is used with a parameter (a target in the `Makefile`), then the search starts from this target.

Here is a small example of a `Makefile`

```
all:     prog test

prog:    main.o  sub.o
         $(LINK.c) -o $@ main.o sub.o

main.o:  incl.h main.c
         gcc -c main.c

sub.o:   incl.h sub.c
         gcc -c sub.c

test:    prog test.data
         prog < test.data > test.results
```

`touch` can be used to change the date of last modification.

`make` can also be used as a simple user interface for commands, when there are dependencies among them. Suppose that you have a dBase on which you can edit, make extraction, preformat, visualize or print. The user could then say: `make visualize` or `make edit`, and all necessary operations will be done automatically. Here is the corresponding makefile:

```
all : catalogue stickers

catalogue : Catalogue.dvi
        dvips -Php0d Catalogue

stickers : Stickers.dvi
        dvips -Php0 Stickers

catalogue.win : Catalogue.dvi
        xdvi Catalogue &

stickers.win : Stickers.dvi
        xdvi Stickers &

Catalogue.ps : Catalogue.dvi
        dvips Catalogue -o

Stickers.ps : Stickers.dvi
        dvips Stickers -o

Catalogue.dvi : Catalogue.tex catalogue.tex
        latex Catalogue

Stickers.dvi : Stickers.tex stickers.tex
        latex Stickers

catalogue.tex : m.rdb
        report catalogue.report < m.rdb > catalogue.tex

stickers.tex : m.rdb
        report stickers.report < m.rdb > stickers.tex

m.rdb : mediatheque.rdb
        cp mediatheque.rdb m.rdb

mediatheque.rdb : mediatheque.db
        m.awk mediatheque

clear :
        rm catalogue.tex stickers.tex Catalogue.dvi Stickers.dvi \
        Catalogue.ps Stickers.ps Catalogue.log Stickers.log      \
        Catalogue.aux Stickers.aux
```

If the files reside on more than one machine (using NFS for example), they should all be synchronized with `ntp` or similar time protocols. See section 10.4.2.

For very large projects, when many persons are involved in the development,

`make` is not sufficient. `make` ignores the notion of version or file locking that are necessary in these circumstances.

Other tools exists for them, in particular `sccs`, `RCS` or `CV`. `diff` and `patch` can be used to keep track of incremental updates and versions (including the recovery of previous code).

3.11 RCS and SCCS: Automatic Revision Control

RCS and SCCS designate sets of tools that help maintaining revisions of a product. Only RCS will be discussed; SCCS offers approximately the same capabilities while having an older, clumsier syntax. CVS is intended for the simultaneous update of files by many users.

If a program of a certain importance is being developed, it is essential to keep *all* versions of the source code — not just the last, or the ten last. All versions should be numbered; a log file should account for all the modifications made between two numbers; version numbers should be allowed to ramify in a tree-like manner; the binary code produced should be stamped with the version number; and if many people work on the same project, there should be some coordinating means between them.

RCS is a set of tools for UNIX that manages automatically these tasks. Text files are normally hidden by RCS. A developer may *check a file out*, that is make it visible in his directory for modification, while locking other developer's access to it; edit it, write appropriate logging information; and *check it in back*. Initially, a file `f.c` is placed under RCS' supervision with

```
ci f.c
```

with initial version 1.1. The file is moved to a special directory, usually `~/RCS/`. An edit cycle would now be:

```
co f.c
edit f.c
ci f.c
```

If you have EMACS, you may use its built-in capabilities to simplify this process: edit the file using its true path (`~/RCS/f.c`), and type *Ctrl-X* and *Ctrl-Q* to check the file in and out respectively.

It is not necessary to modify your `Makefiles`, as `make` automatically checks out and deletes files it doesn't find. If you really wanted to, you would just put:

```
...
f.c: /home/mickeymouse/RCS/f.c
<TAB>   co $<
...
```

RCS can stamp source and object code with special identification strings. To obtain them, place the marker "`$Id$`" somewhere inside your source file. `co` will automatically replace it with `$Id:` *filename revision_number date time author state locker*`$` and the marker "`$Log$`" is replaced by the log messages that are requested during a check-in.

RCS keeps all your previous versions through *reverse deltas*, i.e. keeps the last version in full, and reverse diff's to obtain previous revisions. These are accessed through

```
co -r<revision #>
```

and a sub-branch, new level major release etc. may be defined with

```
ci -r<new revision #>
```

Besides `ci` and `co`, RCS provides a few commands:

`ident`	extracts identification markers
`rlog`	extracts log information about an RCS file
`rcs`	changes an RCS file's attribute
`rcsdiff`	compares revisions

Refer to the manual pages for more detail.

3.11.1 RCS in a multiuser environment

UNIX by itself provides no file lock, neither file access control. But all the nuts and bolts are present.

For a good multiuser system with personalized file access control,

- create a user `rcs`, without terminal access (no shell) and locked password (`*LK*` in `passwd` file),
- make RCS directories belonging to `rcs`,
- for each file, use `rcs -a` to give access to every authorized users.

3.11.2 Remarks concerning RCS

1. The directory `~/RCS` is **not** made automatically (use `mkdir RCS`)
2. `ci` will not move `...c,v` files automatically to `RCS` (use `mv`)
3. `co` and `ci` will look automatically in `~/RCS/` if the file is not found in the current directory, and `~/RCS` exists.
4. `co` and `ci` will **not** lock automatically the files, use `co -l` instead.
5. `co` and `ci` work also on wild card. For example, `co -l *.c` will extract all `.c` files at once.
6. `rcs -l` *file* will lock the file. This is necessary if you modified a non locked file.
7. `rcs -U` / `rcs -L` *file* will enable/disable the file, doing strict locking.

4 Shell programming

When a set of commands is repeated more than 2 or 3 times, then it is usually worth putting them into a file and executing the file, passing possibly parameters. Such files are called script files in UNIX.

All UNIX shells offer lots of usual programming constructions, as variables, conditionals and loops, input and output, even some rudimentary arithmetic. Shell programming cannot replace C programming, in particular it is much slower, but it can be very effective to organize together the repetitive and possibly conditional execution of programs.

Writing script files can have two other advantages:

– They can be edited until it works, even once ...

– They keep track of what was done, either as a log, or as an example for a similar problem in the future.

To be executable, a file just needs the `x` bit set. This is done with the `chmod +x` *script* command.

As many different shells can be used in UNIX, it is preferable to add as a first line a comment telling the system which one is used. So the first line of a script file should look like `#!/bin/sh` or whatever other shell is used (remember they have different syntax, and should not be confused).

4.0.3 Comments

Any character between the # and the end-of-line is treated as a comment. The example just above is really a comment, and is understood by the shell as a possible indication about which shell should be used. In such a case, the # is called the *magic number*.

4.0.4 Quotes

Two quotes symbols can be used: ' and ".

Inside ' ', no special character is interpreted.

Inside " ", then $, `, !, and \ are the only ones interpreted.

Any special character can be transformed into a normal one with a \ in front.

Try:

```
Test="NoGood"
echo 1. Test          # just ascii string
echo 2. $Test         # $    in front
echo 3. \$Test        # \$   in front
echo 4. \\$Test       # \\$  in front
```

4.0.5 Parameter passing

A command can be followed by parameters as "words" separated with spaces or tabs. The end-of-line, a ;, redirections or pipes end the command.

Inside a script, $n, where n is a digit, will be replaced by the corresponding parameter. Notice that $0 corresponds to the name of the command itself.

As a very simple example, here is a script that will compile a C program, and execute it immediately. The name of the program is passed as a parameter.

```
#!/bin/sh -x
gcc -O3 -o $1 $1.c
$1
```

To compile and execute threads.c, one would type `ccc threads` .

4.0.6 Variables

Variables can be defined inside a shell. Except if exported, they are not seen outside the shell. Variable names are made of letters, digits and underscores only, starting with a letter or an underscore.

They can be defined with =, without any space around the = sign, or read from the terminal or a file.

```
Test="Order==$1"
read answer
```

and used, as for parameters, with a $ in front for them to be replaced with their content.

```
if [ "x$answer" = "xY" ]; then
   SetPower $level
fi
select "$Test"
```

4.0.7 Environment variables PATH , MANPATH and LD_LIBRARY_PATH

When the name of a program (a file name effectively) is given for execution, the system will look in successive directories, and execute the first one found.

In the same way, man looks in successive directories and prints the first corresponding pages found, and the loader looks in the list of directories for dynamic libraries.

These lists of directories are given in the variables LD_LIBRARY_PATH, MANPATH and PATH.

The directory names are separated with colon (":") characters.

To add a new directory, use command (in *bash*):

PATH=${PATH}:<*my_dir*>

or

PATH=<*my_dir*>:${PATH}

The first version puts the new directory at the end, the second in front of the list. Both versions have some advantages.

tcsh keeps a hash table of all executables found in the PATH. This table is setup at login, but it is not automaticaly updated when PATH changes. The command rehash can be used to update manually the hash table.

- a "generous" PATH is predefined in most *Linux* systems
- the current directory "." is usually part of the PATH . It is better to put it at the end of the list to avoid replacing a system program.
- you can put all your executables in a directory called ~/bin and add ~/bin to your PATH . (in the file ~/.login or ~/.profile).
- you can do the same for your personal man pages.
- to see the full PATH as defined now, use the command:

  ```
  echo $PATH
  ```

- to see all environment variables:

  ```
  env
  ```

- to find where an executable is:

  `which` *my_program*

- to find where are all copies of a program (in the list defined by PATH):

  `whereis` *your_program*

  You may have to redefine whereis in an alias to search the full PATH :

  ```
  alias=whereis "whereis -B $PATH -f"
  ```

- If you add directories in an uncontrolled way, the same directory may appear in different places ... To avoid this, you can use the PD program envv :

  `eval 'envv add PATH` *my_dir* `1'`

  The last number, if present, indicates the position of the new directory in the list. Without a number, the new directory is put at the right end of the list.

  Notice that envv is insensitive to the shell used (same syntax in *tcsh*, *bash* and *ksh*.

4.0.8 Reading data

Variables can be read from the keyboard with the read command as seen above. Any file can be redirected to the standard input with the command exec 0<file. Then the read command gets lines form the file into the variables. The arguments can be individualy recovered with the set command:

```
exec 0< Classes
read head
set $head
echo The heads are: $1 $2 $3
```

4.0.9 Loop – `foreach` command

In `bash`, the command `for` permits to loop over many commands with a variable taking successive values from a list (See section 4.1 for a `csh` equivalent).

The syntax is:

`for` <*variable name*> `in` <*list of values*> ; `do`)
<*commands*>
<*commands*>
...
`done`

Here are a few examples using `foreach` in `csh` scripts. Try to rewrite them in `ksh` ones.

1. Repeat 10 times a benchmark:

```
for bench in 1 2 3 4 5 6 7 8 9 10 ; do
   echo Benchmark Nb: $bench
   benchmark | tee bench.log_$bench
done
```

2. Doing `ftp` to a set of machines. We assume that the commands for `ftp` have been prepared in a file `ftp.cmds`:

```
for station in 1 2 3 7 13 19 27 ; do
   echo "Connecting to station infolab-$station"
   ftp infolab-$station < ftp.cmds
done
```

   Such commands enable us to update a lot of stations in a relatively easy way.

4.0.10 File name modifiers

The variable names can be modified with the following modifiers:

<*variable name*>:`r` suppresses all the possible suffixes.

<*variable name*>:`s/`<*old*>`/`<*new*>`/` substitutes <*new*> for <*old*>.

Many more modifiers exist, look in the man pages of `csh` for a complete list.

Example: Save all executables and recompile:

```
for file in *.c ; do
   echo $file
   cp $file:r $file:r_org
   gcc -g -o $file:r $file
done
```

4.1 bash and csh command syntax compared

Today, many people use `tcsh` for interactive work. Other prefer `bash` or `ksh`. It has so many goodies. But for shell programming, writing scripts, the choice is really open between `sh` and its offspring (`ksh`, `bash`...) on one side, and `csh` on the other. `ksh` or `bash` are now the default standard on Linux, probably the simpler yet most powerful of all. `csh` on the other end has the advantage of being a subset of `tsch`, with which the user is probably more comfortable. As with many other choices with computers, it has become a question of religion. Make your mind!

If your problem is more complex, if you need arrays, if you manipulate many files, then probably neither `bash` or `csh` are sufficient.

`awk` is almost ideal to manipulate text in any form, but it is not really intended for shell programming. It has only few interactions with the system, with the file system etc.

`perl` provides almost everything you may ever whish, including, in the script language, all facilities of `awk` and `sed`, both indexed and context addressed arrays etc. `perl 5` is now available with most Linux distributions. As for `tsch`, it is not part of the system and has to be installed specifically by the "system manager".

The following pages compare the main commands used in `bash` and `csh`. As you will see, some are missing on one or the other side, others are definitely simpler on one side, and many are quite similar.

ksh	csh
Arithmetic	
`$(( ... ))`	`@`*var*`=`*expr*
`expr` *expression*	
Loops	
`for` *id* `in` *words* `;` `do`	`foreach` *var* `(` *words* `)`
list `;`	...
`done`	`end`
Repeated command	
–	`repeat` *count command*
Menu input	
`select` *id* `in` *words* `;`	–
`do` *list* `;`	
`done`	
Case	
`case` *word* `in`	`switch (` *string* `)`
pattern `)` *list* `;;`	`case` *label* `:`
pattern `)` *list* `;;`	...
`*` `)` *list* `;;`	`breaksw`
`esac`	`default:`
	`endsw`
Conditionals	
`if` *list* `;` `then`	`if (`*expression* `)` `then`
list `;`	...
`elif`	`else if (`*expression* `)` `then`
list `;`	...
`else`	`else`
list `;`	...
`fi`	`endif`
Conditional loops	
`while` *list* `;` `do`	`while (` *expression* `)`
list `;`	...
`done`	`end`
`until` *list* `;` `do`	
list `;`	
`done`	
Function	
`function` *id* `()` `{` *list* `;` `}`	
Signal capture	
`trap` *command signal*	`onintr` *label*
Breaking loops	
–	`break`
	`continue`

4.1.1 Signals used with shells

The main signals used in shells are: `INT` (2), `QUIT` (3), `KILL` (9), `TERM` (15), `STOP` (23) and `CONT` (25). `KILL` can not be caught or ignored, and will bring your shell to an end. `STOP` and `CONT` allows to stop temporarely a shell (or any task) and then restart it without loosing anything.

4.1.2 Sample shell scripts

The following pages list some shell scripts that present various aspect of shell programming. Almost every construction is present, though not necessarely with every options. Some are just toy scripts (`calc`), some real programs used daily for system maintenance (`crlicense`, `png1` and `png2`). `flist` has been used to create this listing.

Here is a table of commands and corresponding scripts where they are used. The scripts bellow are in alphabetical order. Their names appear in the listing at the right, after a long dash line separating the various scripts.

arithmetic	calc calc2 guess1 guess2 minutes
loops	convert convert2 flist tolower toupper
select	term1 term2
case	convert minutes term2
if	convert ddmf_check filinfo flist grep2 guess1 guess2 term1 term2
while	calc2 convert guess1 guess2 minutes
function	convert3
trap	calc2 guess1

```
Tue Oct 3 11:41:33 MEST 2000
------------------------------------------------------------   KillKillMeAfter
#!/bin/bash -f
# Kill the KillMeAfter started by pid $1
# Also kill the sleep started by KillMeAfter

ostype="`uname -mrs | tr ' ' '_'`"
GAWK=/unige/gnu/${ostype}/bin/gawk

KMApid=`ps -ef | \
  tr -s ' ' | \
  egrep KillMeAfter | \
  $GAWK -v pid=$1 '$10 == pid { echo $2  } ' `
```

```
sleeppid=‘ps -ef | \
  tr -s ’ ’ | \
  egrep sleep | \
  $GAWK -v pid=$KMApid ’$3 == pid { echo $2  } ’ ‘

# echo "$0 : KillMeAfter pid : $KMApid"
# echo "$0 : sleep pid       : $sleeppid"

if [ "X$KMApid" != "X" ] ; then
  # echo "killing pid : $KMApid  and $sleeppid"
  kill -9 $KMApid $sleeppid
fi

exit 0
-------------------------------------------------------------   KillMeAfter
#!/bin/bash
# called by some script, with  pid as parameter $1,
# expected to kill it after $2 sec

# echo $0 : pid=$1
# echo $0 go to sleep for $2 sec
sleep $2
# echo $0 weak up
if ‘ps -ef -o pid | egrep $1 > /dev/null ‘ ; then
  kill -9 $1
#   echo pid : $1 should be dead now
# else
#   echo pid : $1 was already killed
fi
exit 0
-------------------------------------------------------------   calc
#!/bin/bash
# Very simple calculator - one expression per command

echo $(($*))
exit 0
-------------------------------------------------------------   calc2
#!/bin/bash
# simple calculator, multiple expressions until ^C

trap ’echo Thank you for your visit ’ EXIT

while read expr’?expression ’; do
     echo $(($expr))
done
exit 0
-------------------------------------------------------------   convert
#!/bin/bash
# convert tiff files to ps
```

```
echo there are $# files to convert :
echo $*
echo Is this correct ?

done=false
while [[ $done == false ]]; do
  done=true
  {
    echo 'Enter y for yes'
    echo 'Enter n for no'
  } >&2
  read REPLY?'Answer ?'
  case $REPLY in
    y ) GO=y ;;
    n ) GO=n ;;
    * ) echo '***** Invalid'
        done=falase ;;
  esac
done
if [[ "$GO" = y\"y" ]]; then
  for filename in "$@"; do
    newfile=${filename%.tiff}.ps
    eval convert $filename $newfile
  done
fi
exit 0
```

convert2

```
#!/bin/bash
# simple program to convert tiff files into ps

for filename in "$@" ; do
  psfile=${filename%.tiff}.ps
  eval convert $filename $psfile
done
exit 0
```

convert3

```
#!/bin/bash
# simple program to convert tiff files into ps

function tops {
  psfile=${1%.tiff}.ps
  echo $1 $psfile
  convert $1 $psfile
  }

for filename in "$@" ; do
  tops $filename
done
```

```
exit 0
-------------------------------------------------------------    copro
#!/bin/bash
# coprocess in ksh

ed - memo |&
echo -p /world/
read -p search
echo "$search"
exit 0
-------------------------------------------------------------    copro2
#!/bin/bash
# coprocess 2 in ksh

search=eval echo /world/ | ed - memo
echo "$search"
exit 0
-------------------------------------------------------------    filinfo
#!/bin/bash
# print informations about a file

if [[ ! -a $1 ]] ; then
  echo "file $1 does not exist !"
  return 1
fi

if [[ -d $1 ]] ; then
  echo -n "$1 is a directory that you may"
  if [[ ! -x $1 ]] ; then
    echo -n " not "
  fi
  echo "search."
elif [[ -f $1 ]] ; then
  echo "$1 is a regular file."
else
  echo "$1 is a special file."
fi

if [[ -O $1 ]] ; then
  echo "You own this file."
else
  echo "You do not own this file."
fi

if [[ -r $1 ]] ; then
  echo "You have read permission on this file."
fi

if [[ -w $1 ]] ; then
```

```
  echo "You have write permission on this file."
fi

if [[ -x $1 ]] ; then
  echo "You have execute permission on this file."
fi
exit 0
------------------------------------------------------------    flist
#!/bin/ksh

# list files separated with name and date as header

ECHO=/unige/gnu/bin/echo

narg=$#
if test $# -eq 0
then
  $ECHO "No file requested for listing"
  exit
fi

if test $# -eq 2
then
  head=$1
  shift
fi

$ECHO `date`
for i in $* ; do
  $ECHO ' '
  $ECHO -n '------------------------------------------------------------    '
  if test $narg -ne -1
  then head=$i
  fi
  $ECHO $head
  cat $i
done
$ECHO ' '
$ECHO '------------------------------------------------------------    end'

exit 0
------------------------------------------------------------    grep2
#!/bin/ksh

# search for two words in a file

filename=$1
word1=$2
word2=$3
```

```
if grep -q $word1 $filename && grep -q $word2 $filename
then
  echo "'$word1' and '$word2' arre both in file:  $filename."
fi
exit 0
```

---

guess1

```
#!/bin/ksh

# simple number guessing program

trap 'echo Thank you for playing !' EXIT

magicnum=$(($RANDOM%10+1))

echo 'Guess a number between 1 and 10 : '

while read guess'?number> '; do
   sleep 1
   if (( $guess == $magicnum )) ; then
     echo 'Right !!!'
     exit
   fi
   echo 'Wrong !!!'
done
exit 0
```

---

guess2

```
#!/bin/ksh

# an other number guessing program

magicnum=$(($RANDOM%100+1))

echo 'Guess a number between 1 and 100 :'

while read guess'?number > '; do
  if (( $guess == $magicnum )); then
    echo 'Right !!!'
    exit
  fi
  if (( $guess < $magicnum )); then
    echo 'Too low !'
  else
    echo 'Too high !'
  fi
done
exit 0
```

---

minutes

```
#!/bin/bash
# count to 1 minute
```

```
i=1
date
while test $i -le 60; do
  case  $(($i%10)) in
    0 ) j=$(($i/10))
        echo -n "$j" ;;
    5 ) echo -n '+' ;;
    * ) echo -n '.' ;;
  esac
  sleep 1
  let i=i+1
done
echo
date
```

---

term1

```
#!/bin/bash
# setting terminal using select

PS3='terminal? '
oldterm=$TERM
select term in vt100 vt102 vt220 xterm dtterm ; do
  if [[ -n $term ]]; then
    TERM=$term
    echo TERM was $oldterm, is now $TERM
    break
  else
    echo '***** Invalid !!!'
  fi
done
```

---

term2

```
#!/bin/bash
# set terminal using select and case

PS3='terminal? '
oldterm=$TERM
select term in 'DEC vt100' 'DEC vt220' xterm dtterm; do
  case $REPLY in
    1 ) TERM=vt100 ;;
    2 ) TERM=vt220 ;;
    3 ) TERM=xterm ;;
    4 ) TERM=dtterm ;;
    * ) echo '***** Invalid !' ;;
  esac
  if [[ -n $term ]]; then
    echo TERM is now $TERM
    break
  fi
done
```

```
------------------------------------------------------------    tolower
#!/bin/bash
# convert file names to lower case

for filename in "$@" ; do
   typeset -l newfile=$filename
   eval mv $filename $newfile
done
------------------------------------------------------------    toupper
#!/bin/ksh
# convert file names to upper case

for filename in "$@" ; do
   typeset -u newfile=$filename
   echo $filename $newfile
   eval mv $filename $newfile
done
------------------------------------------------------------    end
```

4.2 Use of the `history`

`tcsh` keeps a log of the last n commands. n is defined with the command `set history=`n in the file `.cshrcr` or `.tcshrc` .

This log can be used in the following ways:

`history`	prints (on screen) the list of the last n commands executed,
`fc`	repeats the last command,
`fc` n	repeats a given command,
`fc -`n	repeats a given relative command,
`fc` *abc*	repeats the last command starting with the same letters,
`fc -s/`*old*`/`*new*`/g`	repeats the last command with editing (substitution [+global]),

Part of the repeated command can be re-used by the following *word designators*:

0	word 0 (= command)
n	n^{th} word
^	first word
$	last word
$m - n$	words m through n
-	words 0 through but last
*	words 1 through last
%	word matched by the string

The word designators can be modified by appending a modifier to the specifier: <*specifier*>:<*modifier*>

r	root of the file name
e	extension of the file name
h	head of the path (but last comp)
t	tail of the path
s/old/new/	substitution
g	global (comes before s)
p	print but no execution
q	quote words
u	make first lower case letter upper
l	make first upper letter case letter lower

4.3 Command/file name completion

After you have typed a few letters of a command or file name:

< *TAB* > will complete it if possible and unique,

< *ctrl* >d will list all possible completions.

5 Very High Level Programming

Many tools exist now where the basic data unit is not numbers or words, but vectors, matrices, records or files, whose internal structure and detailed manipulation can be ignored by the user.

`matlab`, `SciLab`, `Yorick`, `Python` or `SuperMongo` are good examples of very high level programming environments for graphic, vector and matrix manipulation.

`/rdb` is a similar environment to manipulate relational tables.

For example, here is a small program in `SM`, that reads a file, does some computation, and draws a graph with points of various sizes:

```
data cluster.dat
read{ size 1  viscosity 2  temperature 5 }
set LogT = lg(temperature)
set size = 0.1 + 2 * viscosity
expand viscosity
Diag size LogT
```

and another that selects some columns and rows from a table, using their

names and a selection criteria, then prepares a file for later processing with LaTeX.

```
column name first_name institute < ictp.rdb | \
row ' country == "India" || country == "China" ' | \
jointable -j1 institute - addresses.rdb | \
tabletotex > addresses.tex
```

The commands `column`, `row`, `jointable` etc. are part of the *Perl* `rdb` set of commands that are also used for the exercises.

5.1 Public Domain Software for High Level Programming

Programs like *Mathematica*, *matlab*, *ingres* etc. are very good indeed, but also very expensive, even for universities. Most of them cost now more than even powerful computer stations.

They have an other major drawback: They are produced and maintained generally in the United-States, and the users never get involved in their development. In some sense they are passive consumers.

Since the advent of GNU and more recently of LINUX, the users have the possibility not only to get *free* software of high quality, but more important, in particular for developing countries, to get involved actively in their development, maintenance etc.

If LINUX, gcc, samba, apache and many other products around GNU are so powerful and robust, it is mostly thanks to the very large number of users that participate in their development, find bugs and correct them, exchange idea to improve them etc. This could be a very cheap way to develop strong software competences in your country. When the package is installed in your machine, it requires only access to e-mail to exchange informations... and *your* manpower and basic knowledges. Big supporting organization are not necessary.

Here is a small list of some of the most often used ones, with their equivalent commercial names:

LINUX	Commercial	Comment
octave	matlab	Matrix + 2D and 3D graphics, use the same M-files
scilab		idem, strong for simulation
jacal	mathematica	Symbolic mathematics
maxima	maxima	idem, GNU version of maxima
R	S	Statistics, very complete
gnuplot		2D graphics
pgplot		idem
Yorick	IDL	Data analysis and graphics
Python		idem
RDB	/rdb	UNIX relational database
postgres	ingres	Powerful Relational Database System
msql		idem
MySql		idem
...		

Thousand of public domain applications are available. To have a wider look at the projects you may get involved with, consult:

`http://rpmfind.net/linux/RPM/`

or

`http://sal.kachinatech.com/sal2.shtml`

5.2 Notes about Relational Data Bases

Data Base systems are not part of this course, but it is difficult to build real time systems without producing data that must be stored for later analysis. Environmental parameters, usually noted in log books, should also be put in files.

Many models have been invented to organize (some very large) sets of data, the final goal being to be able to extract rapidly part of these data according to given criteria (see the example in page 45).

The relational model is probably the simplest to understand and use, the only one where mathematical proofs can be used and for which a standard interrogation language (SQL) has been defined.

5.2.1 The relational model

The relational model was introduced by E. F. Codd of IBM in 1970. Its main characteristics are:

- it is mathematically defined
- it is always coherent
- it is fully predictable
- it contains no redundancy

Many commercial or not relational data Bases are now available, for example DB2, Informix, Ingres, Oracle, Postgres, Sybase, /rdb ...

In a relational RdB the data are organized in sets of rectangular tables:

PIN	name	surname	birth	...
9318	Weber	Luc	610711	

PIN	Insurance
9318	Medica

Test	Blood	Sugar
316	...	...
...		
495	...	

PIN	Diag	Interv	**Test**
9318	...	...	316
9318	...	...	495

Some columns (in bold) are key columns. Usually, each row has a different value in them. They do not depend on another one. Non key columns depend on a key one.

The rule behind the choice of columns and the structure of tables, is that no information should appear twice or more anywhere.

5.2.2 RdB basic commands

The basic commands are: insert, delete, sort, search, edit, append and join.

The **join** commands combine two or more tables whose records match on a given column.

Example: `Join Personal Medical on PIN`
`Join Medical Lab on Test`

SQL, the Standard Query Language, is a standard way to do interrogation on a RdB. SQL commands can be embedded into C or Fortran programs, but this is not standardized. See above, page 45 for a small example of a relational database system entirely written in *Perl*, and so very transportable. It is freely available at:

`http://obswww.unige.ch/~bartho/RDB.tgz`

5.2.3 Real Time RdB

Concept: Associate with critical columns a trigger function(s) that is executed whenever an entry is added or changed in it.

The trigger has access to any other data, and can start any operation, including modification in the dB that may start another trigger.

Example of applications:

- stock exchange
- patient monitoring
- central control for complex instruments
- storage monitoring ($\Delta t > 1$ day)

Real time dBs are good examples of the concept of "Objects = Data + Functions".

6 Use of network

The network concepts are part of another chapter. Here are just a few notes on how to use the network for file transfer and remote connection.

6.1 File transfer

File transfer between two computers can be done with the program `ftp` (*file transfer protocol*)

`ftp` <*remote host name*>

On some computers (including infolab-*n*), `ncftp` is available with some extra facilities. It will record all recent hosts you have been connected to and in which directory you worked. It will reuse this information the next time you connect to the same host. Hosts can have short nick names.

6.1.1 Host names

The computer you want to connect to can be local, part of your local network, or nonlocal, part of the rest of the world.

For a local host, the host name is sufficient.

For a non local host, the fully qualified name of the *host.domain.country* is necessary.

For example: `infolab-27` is locally acceptable, but `obsmp2.unige.ch` must be given in full.

Every computer on the Internet has an IP number, made of 4 groups of digits (1-255). For example, infolab-20 has the number `140.105.28.186` .

Both full name and IP number are unique in the world, and must stay so! They can usually be used interchangeably.

See the chapter on Network for the new ipv6 (current is ipv4) protocol and addressing schema.

6.1.2 User names

If you have an account on the remote computer, then use your own *username* and your own *password* on that machine to transfer files back and forth between your local and your remote computer.

If the remote machine is an `anonymous` server, from which you intend to fetch or send files, then you must use `anonymous` as user name, and your email address, in the form `user@host.domain.country` as password. Some servers will accept anything as password, some others will check that it is a valid address. In any case, politeness dictate that you use your true email address, or at least your name and host.

6.1.3 Going to the right directory

When you are connected to the remote computer, you can use the usual `cd` and `ls` or `dir` command to locate your files.

Note that on anonymous servers, directories ready to accept files from anonymous users are usually not readable! ... but you can still fetch a file from them if you know its name and place.

6.1.4 Setting the mode of transfer

The files can be transmitted either in `ascii`, possibly with code conversion if necessary, or in `binary` mode. The `tenex` mode is for binary files with very long records.

6.1.5 Getting files

`get` <*remote file*> <*local file name*> will fetch the file.

`mget` <*first file*> <*second file*> ... fetch a set of files.

`reget` <*remote file*> <*local file name*> will restart the transfer of the file **after** the last previously transferred block (after a problem on the line ...).

6.1.6 Putting files

`put` <*local name*> <*remote file name*> will transfer the file to the remote host.

`mput` <*first file*> <*second file*> ... will transfer a set of files to the remote host.

6.1.7 Compression and tar files

Some servers are set to compress files before transferring them. They can also tar a complete directory and even compress it before sending.

To use these facilities, one must add `.gz` , `.tar` or `.tar.gz` after the file or directory names.

6.1.8 Decompressing a file or directory

`gzip -d` <*compressed file*> will decompress that file.

`tar xzvf` <*compressed tar file*> will decompress and detar the full tar file.

`gzip -dc` <*compressed tar file*> `| tar vxf -` will do it if the decompression is not available within `tar`.

6.2 Working on another computer

To do so, you MUST have an account on the remote machine. No anonymous user is possible (On infolab-*nn* machines, the username `public`, possibly with password `public` can be used in a way similar to anonymous!).

`telnet` < *remote host name* > will establish the connection to the remote host.

`rlogin -l` < *username* > < *remote host name* > will establish a new session for you on the remote host.

6.2.1 Password transfer

If you have in your home directory a file called `.rhosts` with entry lines in the form:

< *host1* > < *username* >
< *host2* > < *username* >

with your current host name on the left part of this file, then the remote system will not ask you for your password if you use the `rlogin` connection.

6.3 Executing a command on a remote host

It is possible to execute a line of commands on a remote station with:

`rsh` < *remote host* > "< *command line* >"

Your local host should be present in the `.rhosts` file in your remote home directory.

If more than one command is present on the line, they should be separated with ";" characters.

For example, to list your files in the directory `tbl` on the remote host `infolab-21`, use the command:

```
rsh infolab-21 "cd tbl; ls -l"
```

6.4 Remote copying a file

`rcp` < *local file* > < *remote host* >:< *remote file* > will copy the local file onto the remote system. Your local host should be present in the `.rhosts` file in your remote home directory.

6.5 Displaying on another station

To have a process running on a station with a X11 display on another, you must:

On the display station: give the permission to write on its screen with the command:

`xhost` <*process station name or IP address*>

(`xhost +` will give permission to any computer in the world. This can be dangerous ...)

On your process station, you may have to redefine the global variable `DISPLAY` with the command:

`setenv DISPLAY` <*display address*>`:0.0`

Then on, all your X11 output will go to the screen of the display station.

6.6 Secure remote commands

If you use `rsh` or `rcp` over the Internet without a `.rlogin` file on the remote station, your password will be transmitted in clear ascii and many spying programs will be able to catch it. With the very large number of nodes traversed by your packets, it is impossible to guarantee any confidentiality, even for sensitive ones.

`ssh` was developed to replace `rsh` and `rcp` while encrypting (and compressing) every packet. X11 packets are also automatically encrypted and compressed. `ssh` use public key encryption in a very clever way. It has to be installed by `root` on the target machine (server), but the client part can reside in the normal user files. Except for the initial "`r`" or "`s`" in their names, the original and securised commands are used in the same way. No extra password is needed. They may be just faster because of the compression on slow, non compressed lines.

You can find informations on `ssh` on the following URL:

`http://obswww.unige.ch/isdc/SSH/ssh-1.2.26.tar.gz`

`http://www-itg.lbl.gov/info/ssh/`

The syntax for `scp` is as follow:

`scp` [-C -c blowfish] [[username]*hostname*:]<source> [[username]*hostname*:]<destination>

Recently, a new version of `ssh` has been developed, that is free to any one and do not contains any proprietary or patented code. It is available for

LINUX and SOLARIS and most other UNIX versions. It should replace very soon the proprietary ssh as above.

The master address for this OPENSSH is:

`http://www.openssh.com/portable.html`

The next URL contains information on OPIE, a password system where the users get a list of passwords that are usable only once, making spying useless. This effectively replace `telnet`.

`http://obswww.unige.ch/isdc/OTP/opie-2.32.tar.gz`

Both `Openssh` and `OPIE` are public domain softwares available for linux and Unix in general.

7 Structured Design

7.1 Introduction

The continued improvement of computer performances have permitted to develop more and more complex programs, leading to a posteriori misunderstanding of the code, and to difficulties in the support and modification of the program.

This has lead Dijkstra in 1965 to the concept of *structured programming*, which can be understood as creating programs recursively consisting of modules of lower level complexity. The modules should describe a whole, a logical entity, at all levels. The operations involved in each module should be described in the most general terms available at this level, and hide the unnecessary details.

7.2 Program Development Phases

Any software project goes through a series of phases, possibly with many loop-backs to previous steps. This is true for both simple modules and large projects as a whole.

The main steps are:

user's requirements , identification of what are the data, what has to be done, which results are expected, what is the time-scale for the project, what money, what hardware is available?

system definitions , formalization of the previous informations

system analysis , looking for solutions to the requirements

program design , software architecture for the adopted solution

program coding , implementation of the architecture

testing , verification against definitions and requirements

improving , smoothing the bottlenecks, getting better user's interface

upgrading , to new requirements, new hardware available etc

The analysis phase is very important as it should lead to a good design for simple programming, maintenance and should enable anyone to further enhance the program without having been involved in the original programming work.

The question is how to decompose the problem in modules ?

There are two main ways in the decomposition process:

7.3 Ascending Design and Programming

Ascending approach is the construction of a complex system by combining modules from the lowest level operations to the complete system, in increasing order of complexity. This is also called Bottom-Up design.

Pros: The modules can be tested in their real functioning at the time they are built.

Cons: We can't know at the module's programming time if it will best fit the next level module.

We don't have a general sight of the problem to be presently solved.

The interfaces are difficult to fix from below.

7.4 Descending Design and Programming

Descending approach is the construction of a complex system by expressing it in terms of simpler layers, with stepwise refinement. This is also called Top-Down design. The descending design presents exactly the reverse situation for the programmer, that is:

Pros: General problem is more correctly decomposed in sub-problems.

Good sight of the problem or sub-problem to be solved at any time of the design.

Design error can be detected and corrected at programming time quite easily.

The interfaces between modules are defined from above.

Cons: Testing the already built modules (which are the higher level modules) need to write drivers submodules simulating the input-output behavior of the real submodules.

In practice, a mixture of both approaches is often used, by combining a descending analysis and design with a ascending programming phase. This mixture can be a good practical way *as long as the analysis and design phase are kept detailed and precise enough to avoid design errors.* If one has to correct the design at the programming time, this one should also be descending.

7.5 Structured Design Principles

Structured programming enables to:

- give a program a better clarity, so that future enhancements may be easily done.
- augment the reliability of the code, because modules can be tested as soon as they are built.
- hide unnecessary details.

The principle of structured programming is to give the programmer tools enabling him to express his problem in structured blocks.

A structured block is a module (at design level), or a piece of code (at programming level) which stands on its own and has only 1 input and 1 output.

The content of this block may be very simple **or** very complex, in which case it should be decomposed itself into other structured blocks.

This can be done with the flow control instructions. With these instructions, the `GOTO` instruction is not needed anymore, so that one can avoid the unverifiable and multiple paths in a program.

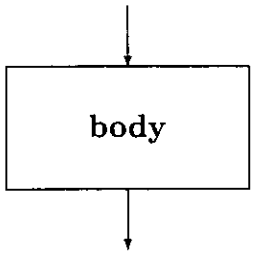


Figure 1: Structured block construction

7.6 Flow Controlling

Each language defines it's own set of flow control instructions, and renames them differently. In this section, we will describe the main flow control instructions, which can be separated into three groups:

Conditional instructions

7.6.1 IF...THEN...ELSE...

Only one of two possible blocks is executed (figure 2):

if *condition* **then** *body_true*
else *body_false*
end

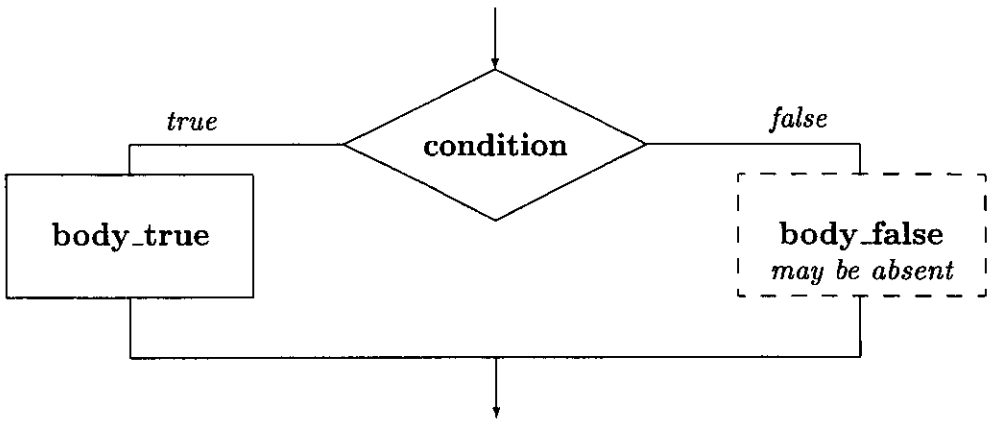


Figure 2: `if...then...else` construction

7.6.2 CASE...OF...

Only one of many possible blocks is executed (figure 3):

case *expression* **of**
    value_1:= *body_1*
    value_2:= *body_2*
    ⋮
end

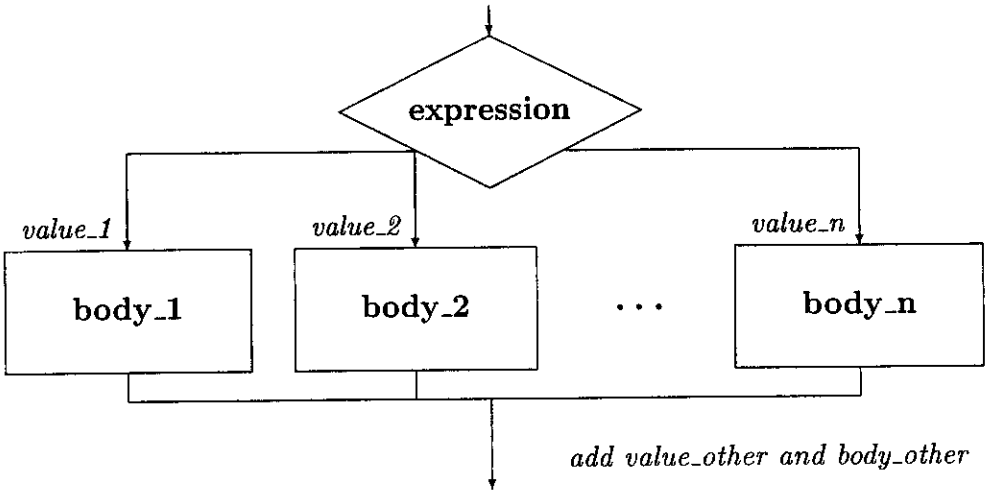


Figure 3: case...of construction

Please take notice that *expression* and *values* are sometimes replaced by *conditions*.

Counting loops

7.6.3 FOR...DO...

A given block is executed an exact number of times (figure 4):

for *variable*:= *first_value* **to** *last_value* **do**
    loop_body
end

Please take notice that *loop_body* is usually executed at least once.

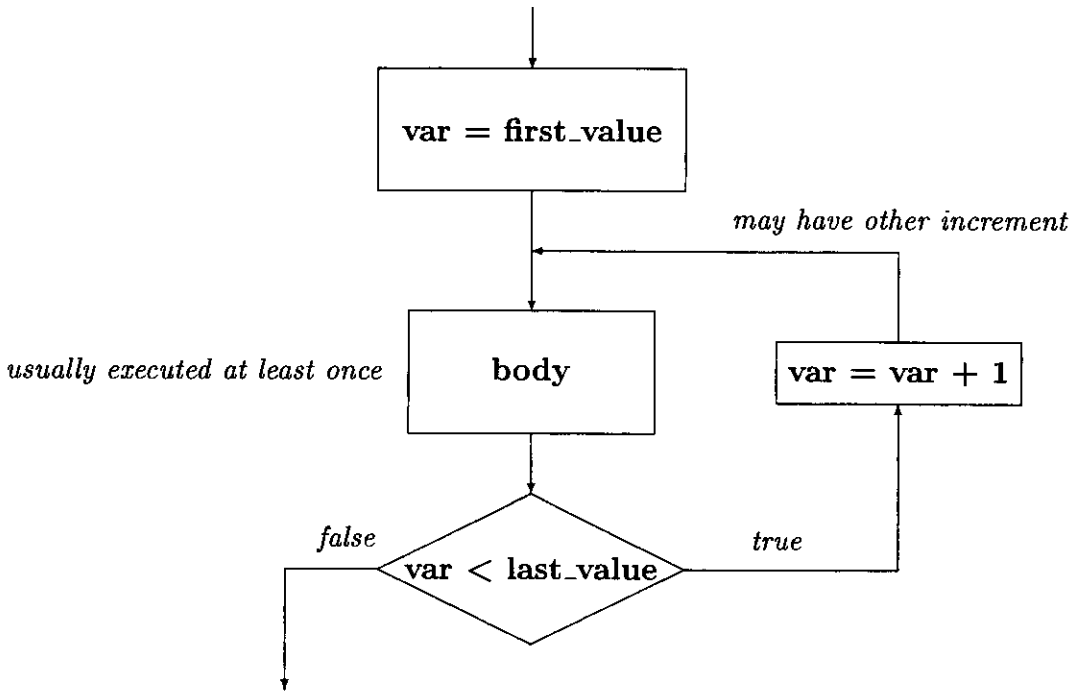


Figure 4: `for...do` construction

Conditional loops A given block may or may not be executed many times depending on a condition. The condition may be set inside the block.

7.6.4 WHILE...DO...

while *condition* **do**
    conditional_body
end

Please take notice that *condition* is tested before the first execution of the *conditional_body* (figure 5).

If the condition is the constant 1, then the loop will go for ever. You will have to use `break` to get out of it.

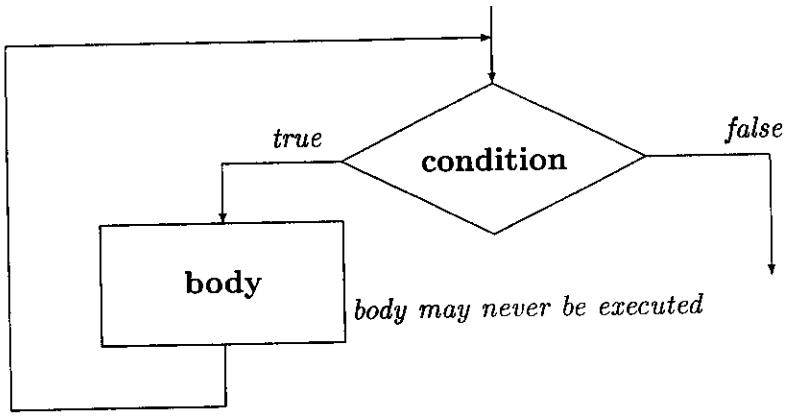


Figure 5: `while...do` construction

7.6.5 REPEAT...UNTIL...

repeat
    conditional_body
until *condition*

Please take notice that *condition* is tested after the first execution of the *conditional_body* (figure 6).

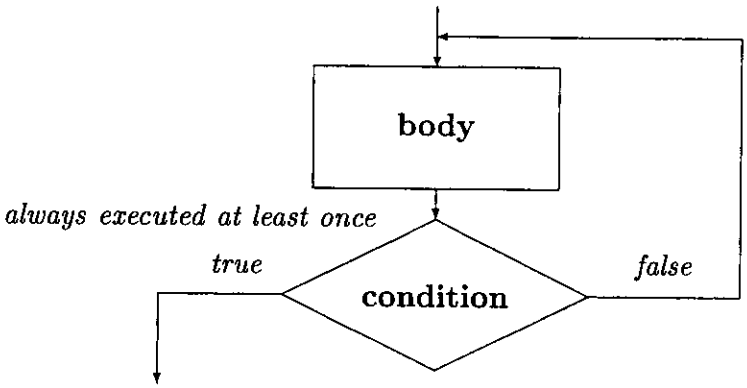


Figure 6: `repeat...until` construction

7.6.6 REPEAT...WHILE...DO...

repeat
body_1
while *condition* **do**
body_2
end

Please take notice that *condition* is tested after the first execution of the *body_1*, but before the first execution of the *body_2* (figure 7.

Consider the following real situation (in pseudo code):

```
read;
if not EOF  do computations
read again
```

We can solve it in three ways:

1. as in Pascal:
   `s=read( );`
   `while(s!=EOF) { calculations; s=read( )};`
   `read` is used twice, and appear illogically after the calculations ...
2. `while((s=read( ))!=EOF) { calculations } ;`
   Now we have side effects in the condition, doing two things in one statement;

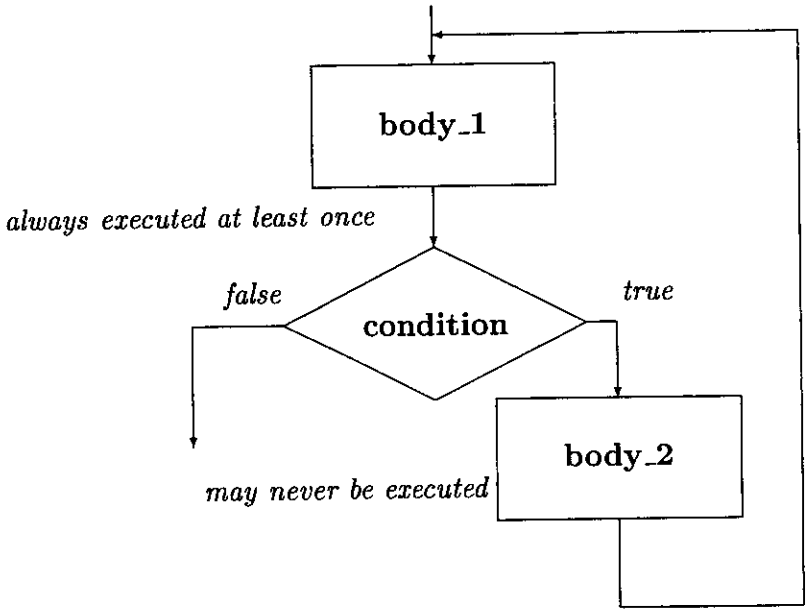


Figure 7: `repeat...while...do` construction

3. `while(1) { s=read( );`
   `if(s==EOF) break;`
   `calculations`
   This matches the logic much better, though the code is longer.

7.6.7 Side effects

In C and many other languages, the tested condition can be any expression, possibly with strong side effects, that is variables get changed by the condition evaluation. For example, consider the expression, given in the C notes:

```
while ( *S1++ = *S2++ ) ;
```

Such expressions are very compact, but rather difficult to read, and quite prone to errors.

7.7 Implementation Addresses

Some languages are more appropriate than others to structured programming, and amongst the procedural languages, let's cite as examples the Pascal, C, Modula and Ada languages.

These languages offer all the preceding possibilities by specialized instructions, except the last one which should be programmed with a loop and an internal if instructions. Moreover, Ada language doesn't support the `repeat...until` structure.

Other branching instructions complete the set, enabling the program to interrupt or skip an occurrence of a loop.

Most of the procedural languages offer a `goto` instruction, just in case..., but to avoid using it will lead to better design, and maintenance.

7.8 Weaknesses of the Structured Approach

The modules are based on their functionality, and define procedures and functions, while variables are often passed as parameters, or are globally (on the outside) defined.

This leads to

- logically incomplete modules.
- difficulty to reuse a module in a slightly different way
- variables can be modified from the outside of the module.

7.9 Practical remarks concerning the exercises

1. All system calls and standard library routines return a value indicating the success or failure of the operation. The error code is also returned in the variable `errno`.

   This value should always be checked, with an error message and appropriate action (continue with default, do it again, exit ...) in case of failure (See the examples below).

2. `stdio.h` and other header files (including your own) contain list of declarations like `#DEFINE OEF (-1)` or even `#DEFINE NULL (0)` and also `typedef ... { ... } FILE;`

Use them! They help you hide something and make the code easier to read, check and understand.

Examples:

```
FILE *Pn
      Pn = fopen("/ds", "w" );  /* not "2" but "w" */

if( (fn=open("/ds", "w")) == NULL )
   { printf("cannot open file /ds \n") ;
     exit (11);
}

if( (fn=open("specific", "r")) == NULL )
if( (fn=open)"default",  "r")) == NULL )
   { printf("neither specific nor default available \n");
     exit (13);
   }
```

Notice in the last example, that the second `if` is skipped if the first succeeds.

8 Data structures

Data structures can be classified into two main categories: linear and non-linear. Linear structures are composed of a sequence of elements and include *arrays*, *linked lists*, *stacks* and *queues*. Non linear structures include *trees* and *graphs*. We will limit our scope to a general introduction to the linear structures, as they are the basis of the structures used in real-time systems.

The operations that can be performed on a linear structure are:

- Traverse the structure and process each element.
- Search a particular element of the structure.
- Add a new element to the structure.
- Remove an element from the structure.
- Rearrange the elements in some order.

The internal representation of a linear structure may take two shapes:

- Array representation, where logically consecutive elements of the structure are represented by *sequential memory locations.*
- Linked list representation, where the relation between the elements are represented by means of *pointers.*

The type of representation one chooses for a particular structure depends on how it will be accessed, and on how many times the different operations will be performed.

8.1 Arrays

Arrays can be linear or multidimensional homogeneous structures. We will limit our scope to linear arrays; the extrapolation of the algorithms to the other cases is relatively easy.

The linear array is a finite list of data elements. The elements are referenced by an *index*, which is the ordering number of the element. The elements are stored in consecutive memory locations. That implies that the index set is composed of consecutive numbers.

The smallest index is called the *lower bound* (LB), and the largest is the *upper bound* (UB). The length of the array is given by the formula

$$L = UB - LB + 1$$

Usually, $LB = 0$ and $L = UB + 1$, or $LB = 1$ and $L = UB$.

The logical representation of an array consist of a series of compartments pictured either vertically or horizontally, depending on the number of elements and on the available space, as shown on the figure 8.

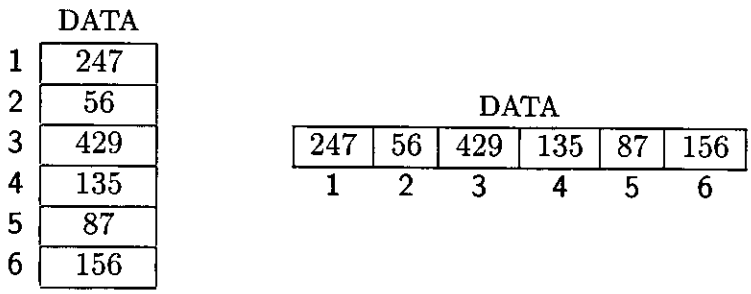


Figure 8: Logical pictures of array DATA.

The computer keeps only track of the *base address* (BA) of the array A, and calculates the position of the kth element by the formula:

$$LOC(A[k]) = BA(A) + w \cdot (k - LB)$$

where w is the number of memory words (bytes for an 8-bit architecture) per element for the array A. The figure 9 shows the internal representation of an array AUTO, with $BA = 200$, $LB = 1932$, and $w = 4$.

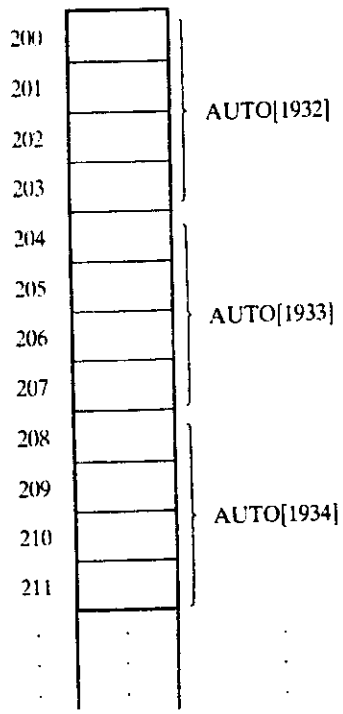


Figure 9: Memory representation of array AUTO.

8.1.1 Operations on linear arrays

Operations on arrays are simple, due to the linear structure of the arrays.

Traversing an array is done by a counting loop (7.6.3), the index of the array being used as the control variable of the loop. The body of the loop defines the operations to do on each element.

Inserting an element at the end of an array is quite simple. Inserting an element in the middle of the array implies moving all the elements located after the insertion point up back a position. This again may be done by using a counting loop initialized at the upper bound, and running down to the insertion point. One has to do it this way, as the higher indexed memory locations may be overwritten without problem.

The figure 10 illustrates this by inserting the value "Ford" in a string array at position 3.

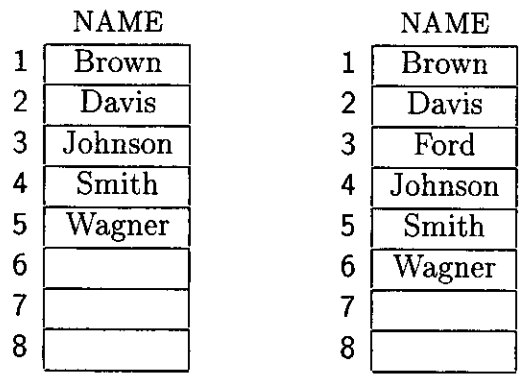


Figure 10: Insertion of an element in an array.

Notice that decreasing index counting loops are not supported by all languages. If not supported, this operation can be simulated by a conditional loop (7.6.4).

Deleting an element of the array is very similar to inserting, at the algorithmic level. A counting loop running upward from the deletion point should be used to move down the succeeding elements.

Searching an element in the array can be done through two algorithms: linear and binary search.

Linear search implies a conditional loop executed at least once. The loop body should check if the element fits the desired item and if the bound of the array is reached. This implies two comparisons at each occurrence of the loop, leading to a possible $2N$ comparisons.

The estimation of the number of basic operations an algorithm needs to be completed is called the complexity of the algorithm. It gives the notion of computation time for the implementation of the algorithm. It is sometimes expressed with the O notation:

$$f(n) \quad \text{is} \quad O(g(n))$$

Where $f(n)$ is the complexity, $g(n)$ is a simple function.

An enhanced algorithm will first write the searched item at the end of the array, in position $N+1$. Then a single comparison is done in the loop, checking for the item, and when successful, a last comparison determines if the item was found in the array or in position $N+1$. The maximum comparisons number is thus $N+1$.

The average number of comparisons, in case of equally probable position of the item, with an absence probability of ε is given by

$$\begin{aligned} 1\cdot\frac{1}{N}+2\cdot\frac{1}{N}+\cdots+N\cdot\frac{1}{N}+(N+1)\varepsilon &= \frac{N(N+1)}{2}\cdot\frac{1}{N}+(N+1)\varepsilon \\ &= (N+1)(\frac{1}{2}+\varepsilon) \end{aligned}$$

If the absence probability is very small, the average number of comparisons will be about half the length of the array.

Binary search is used for maximum efficiency. The array *needs to be somehow sorted.* The comparisons will not be done sequentially, but accessing recursively the middle of the part of the array containing the item to find. At the beginning, the containing part is the whole array.

After M comparisons, the segment containing the item is $\frac{N}{2^M}$ long. Locating the item implies thus a maximum of $M=\log_2(N)+1$ comparisons. This means that a 65000 element array could be searched successfully in 16 comparisons.

So why not use always a so economical algorithm ? Binary search is only possible if the array is sorted, and maintaining a sorted array can be very resource-consuming, for big arrays with a lot of modifications.

Sorting an array is a bit more complicated. There are several algorithms suitable for different data structure. The most simple is called *bubble-sort.*

Let's have a N-element array. The algorithm consists of traversing the array, comparing each element with the element immediately following it and swapping the two elements if necessary. This traverse operation, called a *pass*, enables to put the smallest or the largest element (according to the test) at the upper bound, in element N. This step is repeated $N-1$ times with the sub-arrays upper-bounded by the element indexed $N-1$, $N-2$, etc.

The complete sort is a $N-1$ passes process. The passes involve $N-1$, $N-2$, etc. comparisons, so the entire sort process need, to be complete, a total of

$$(N-1)+(N-2)+(N-3)+\cdots+2+1=\frac{N(N-1)}{2}$$

which is proportional to N^2.

Another well-known sorting algorithm is the *quicksort* algorithm.

In this algorithm, each *step* (fig. 11) is used to find the proper place for one element of the array. Let's take the first number of the array. We compare it with the others, starting backwards from the last. When a smaller number is found, we exchange the two numbers, and start again traversing from left to right the array until we find a larger number. This step stops when the comparison with the element itself. This element is at its correct place in the array.

We then have two sub-arrays which are themselves to be quicksorted.

Comparison 1	[44]	33	11	90	40	22	88	66
Comparison 2	[44]	33	11	90	40	22	88	66
Comparison 3	[44]	33	11	90	40	22	88	66
Swap 1	22	33	11	90	40	[44]	88	66
Comparison 4	22	33	11	90	40	[44]	88	66
Comparison 5	22	33	11	90	40	[44]	88	66
Comparison 6	22	33	11	90	40	[44]	88	66
Swap 2	22	33	11	[44]	40	90	88	66
Comparison 7	22	33	11	[44]	40	90	88	66
Swap 3	22	33	11	40	[44]	90	88	66
	subarray 1					subarray 2		

Figure 11: One step of the quicksort algorithm.

The quicksort algorithm is in the worst case when the array is already sorted. Each step needs N comparisons and produces only one sub-array, of length $N-1$, leading to a total of

$$N+(N-1)+(N-2)+(N-3)+\cdots+2+1=\frac{N^2}{2}$$

comparisons, which is proportional to N^2. The advantage over the bubble-sort appears for the average case. Bubble-sort has a constant number of comparisons. Quicksort, on the other hand, produces 2 sub-arrays in each step, so the successive levels place $1, 2, 4, \ldots, 2^{k-1}$ elements. About $\log_2(N)$ levels will be necessary to sort the array, with a maximum of N comparisons at each level. The average number of comparison for the quicksort is thus proportional to $N \log(N)$.

8.2 Linked lists

As the insertion or deletion of an element in an array is a quite expensive operation, and as arrays are static structures that cannot easily be expanded, it is sometimes necessary to use another type of structure, whose elements contain, in addition to the data, a link to the next element. This way, successive elements need not occupy consecutive memory locations.

This type of structure is called a *linked list*, and is widely used in computer science, due to it's dynamic behavior. A linked list is composed of *nodes*. Each node is divided into two parts: the *information part* and the *link field* or *next pointer field*, which contains the address of the next node in the list.

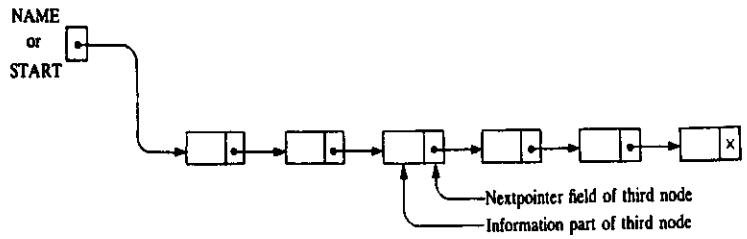


Figure 12: Horizontal representation of a linked list.

A linked list is represented by a series of double boxes linked by vectors, either horizontally or vertically, as shown in figures 12 and 13. The information part may be further subdivided, as seen in figure 13. A separate variable indicates the first element of the list. It is the list pointer variable ($START$). The last element of the list contains a null pointer to indicate the end of the list.

8.2.1 Operations on linked lists

A linked list may be maintained in memory by means of two arrays, one containing the data and the other the links, or by using an array of records

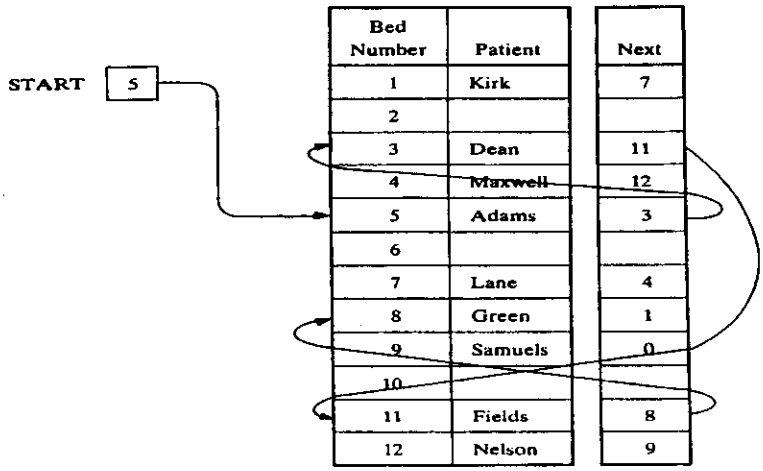


Figure 13: Vertical representation of a linked list.

containing both the data and the links. Let the informative part of element K be $INFO[K]$ and the link field of the same element be $LINK[K]$. Let also $START$ contain the first node address and $NULL$ be the content of the last link.

Traversing a linked list is done by using a variable PTR containing initially the address of the first node ($PTR := START$). After having processed the first node's data, the pointer is updated to point to the next node ($PTR := LINK[PTR]$) and the loop is repeated until $PTR = NULL$.

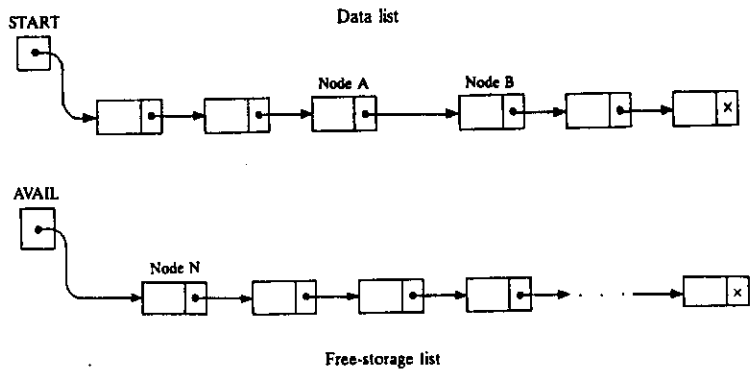


Figure 14: Linked list before an insertion.

Insertion To insert a new node in a list, we need to have some available memory locations, and to be able to allocate them to the list. This is

done by maintaining a parallel list called the *list of available space*, the *free-storage list* or the *free pool*. Let this list be called $AVAIL$.

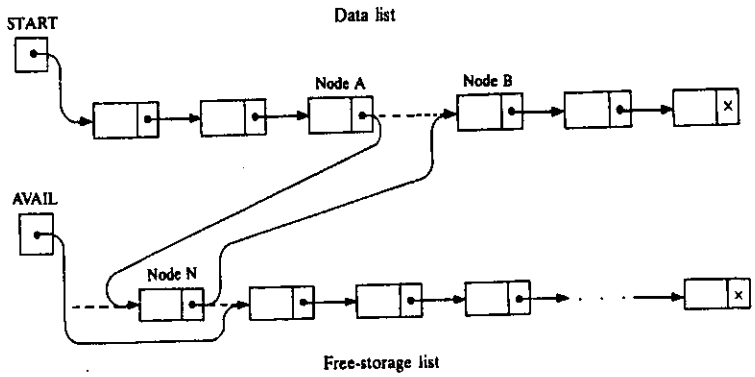


Figure 15: Linked list after an insertion.

The insertion of a node between nodes A and B of a list (fig. 14) is done by removing the first node of $AVAIL$ and storing its address in an auxiliary variable NEW ($NEW := AVAIL$). The $AVAIL$ is updated ($AVAIL := LINK[AVAIL]$); we will then copy the new data in the new node ($INFO[NEW] := ITEM$), and at last we have to insert the new nodes in the list ($LINK[NEW] := LINK[A]$; $LINK[A] := NEW$). The resulting lists are presented on figure 15. Note that were the insertion point be the first node, the two last assignments would have been $LINK[NEW] := START$; $START := NEW$.

Deleting a node of a list seems very simple, as we have only to reassign the pointer of the preceding node to point to the next node. In reality, we can't know the address of the preceding node without traversing the list to compare each node with the deletion point, while remembering the preceding node until the actual node is processed. Another problem is to deallocate the memory we don't use anymore. This task is called *garbage collection* and is done by returning the node to the $AVAIL$ list (fig. 16). Thus, deleting an element of a list is done by traversing the list once, and then returning the node to the free pool, which implies about the same operations as inserting a node. While doing the traversing, we are able to do another task, as searching, for example, a node with specific data, which we want to delete.

Searching a specific item throughout a list implies a loop with an internal concordance test. If the list is sorted, the test may be smarter to check if the item position is already over-passed, which would lead us to stop the loop.

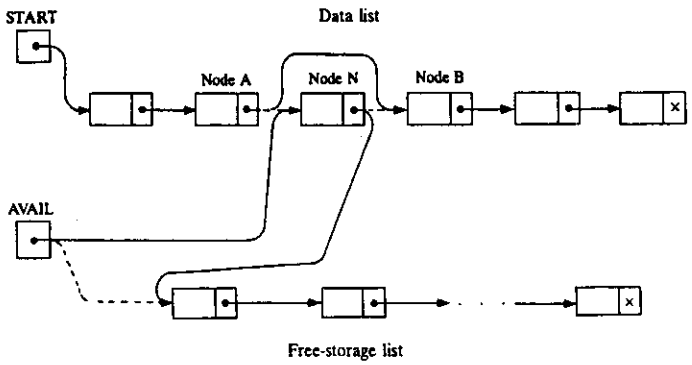


Figure 16: Deletion in a linked list.

Binary search is not possible with linked lists, since there is no way to point to the middle of a list.

Sorting a list may be done by different algorithms. The bubble-sort algorithm (8.1.1) will be suitable for a linked list, but the quicksort algorithm (8.1.1) will need the particular properties of a two-ways list (8.2.2).

Another good way to have a sorted list is to keep it sorted, i.e. insertion is done at the right place (searching).

8.2.2 Particular lists

There are several particular forms of lists that can be used in different situations.

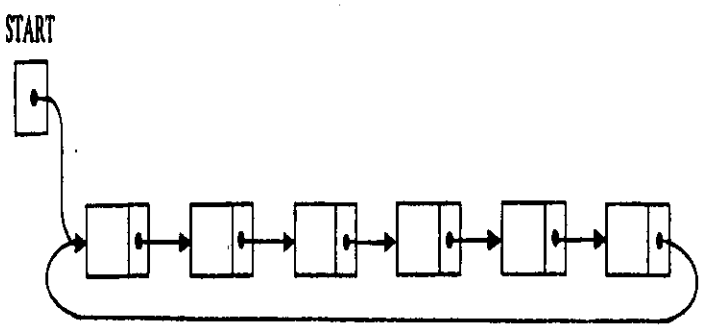


Figure 17: Circular linked list.

A *circular list* (fig. 17) is a linked-list whose last node's link points to the first

node. This kind of list is widely used in computer science, because all the pointers contain valid addresses, and no special treatment is thus required neither for the first node, nor for the last.

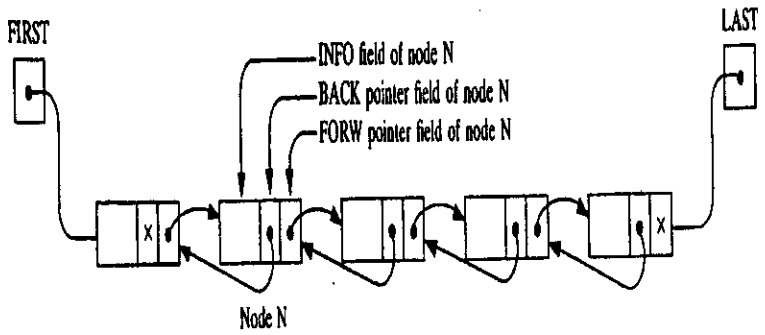


Figure 18: Two-ways linked list.

A *two-ways list* (fig. 18) contains three parts nodes. In addition to the data part and the link field $LINK[K]$ now called $FORW[K]$, there is a second link $BACK[K]$ pointing to the preceding node. The $START$ variable is replaced by two entry point variables $FIRST$ and $LAST$. A two-ways list has the following properties:

- $FORW[A] = B \iff BACK[B] = A$
- Operations can be done in either direction.
- For deletion, the localization of the preceding node is trivial.
- Insertion is a bit more complicated by the presence of the second pointer, i.e. needs two more assignments than insertion in a one-way list.

A *two-ways circular list* mixes the properties of the two previous lists.

8.3 Stacks

A *stack* is a linear structure accessible only by one extremity. This notion is very familiar to us, as we use a lot of stacks in every-day's life, as illustrated in figure 19.

All the operations will be done on a particular point called the *top of the stack*. Adding an element is done by *pushing* it on the stack. Removing an element from the stack is called *popping* (fig. 20). As the top is the only

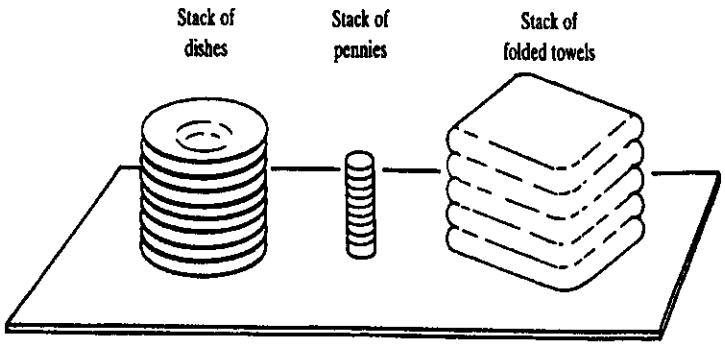


Figure 19: Every-day's life stacks.

access to the stack, the last element pushed in will be the first popped out from the stack. This *last-in, first-out* property has given to the stack its second name: *LIFO*.

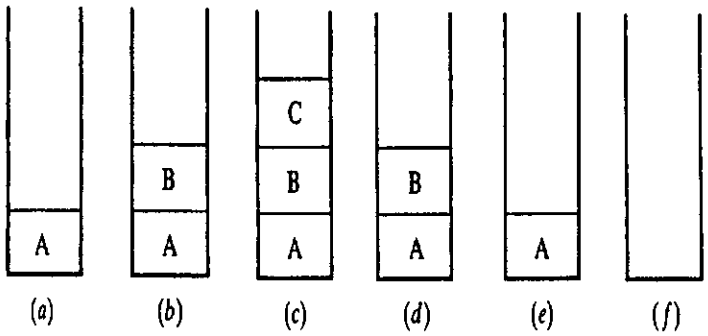


Figure 20: Stack push and pop operations.

Stacks are widely used in computer science. They are the basic structures on which the notion of recursion is implemented, and many well-known algorithms or problems have been implemented and solved through its usage.

Remember the quicksort algorithm (8.1.1). A practical way to keep track of all the sub array bounds while processing one of them is to put them on stacks. The *Towers of Hanoi* problem is implemented recursively (recursion uses stacks), or may be implemented with stacks in an iterative way. *Reverse Polish Notation* (RPN) which writes operations as operands followed by the operator uses stacks: The operands are put on the stack, where each operator pops the number of operands it needs.

8.4 Queues

A *queue* is another familiar concept (fig. 21). In computing, queues are also widely used for bufferizing data arriving from or leaving to a peripheral, or to schedule tasks to a processor. They have a *first-in, first-out* structure, and thus are also called *FIFO*.

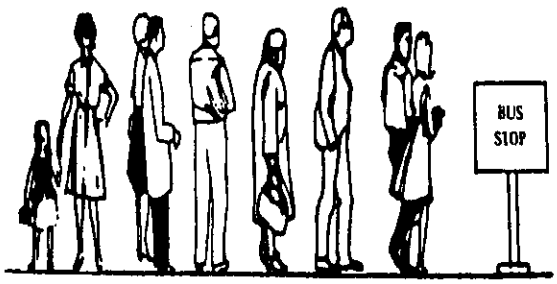


Figure 21: Familiar queue.

Data may be added in a queue only at the end called the *front*, and removed only at the other end, called the *rear*.

Special implementations of queues allow other types of access:

Deques are double ended queues, that can be accessed by either ends, but not in the middle.

Priority queues are queues where the highest priority element is to be processed first. The implementation will determine the ease of inserting or deleting the element in a priority queue. A way to implement a priority queue is to use a linked list with its usual properties for insertion, but where processing and deletion is limited to the first element. In the figure 22, successive deletions will remove AAA, BBB, etc., while insertion of an element XXX is done at a place determined by the algorithm according to its priority (2).

9 Object Oriented Computing

It is highly preferable to group in one unit a logically linked data set. On the other hand, it is not necessary that higher level modules know the internal

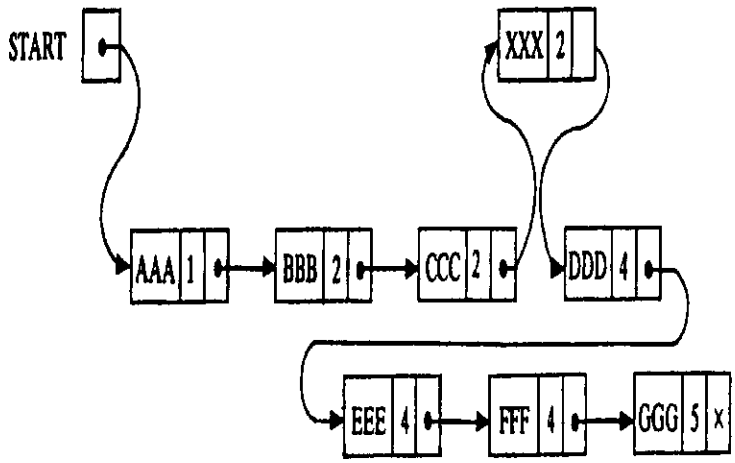


Figure 22: Representation of a priority queue implemented as a list.

functioning of the routines or the structure of a complex data set. An external module should perceive them as a functional black box. This vision is close to the block diagrams used in electronics or automatics.

The interface of the module is its visible and accessible part. It represents the specifications of the module and can be separated from the implementation part, which describes the functionalities of the module.

The Modula-2 language was one of the first languages to comply with the separate compilation of the modules. It addresses the notion of visibility, with library modules consisting of a definition and an implementation parts.

9.1 Objects

The basic concept of object-oriented description is to consider a program as a model for a real world situation. Now, the real world consists of related objects. Objects are thus more stable than relations in the system evolution.

It seems thus natural to decompose this real world situations model in objects models rather than in models of the relations existing between these objects.

From now, we will call objects the models of real world objects. *An object is the whole set of characteristic properties satisfactory to describe the object with regard to the studied model.*

In classical programming, we consider an algorithmic description of the sys-

tem, in which we introduce data. In object-oriented programming (OOP), we consider objects, whose behavior is described by algorithms.

Object = Data structure + Related operations

9.2 Object Oriented Design

Different advantages of the object-oriented approach are examined in the next sections:

9.2.1 Easy Design

Our brain is used to apprehend real objects. The definition of a program's main concepts as objects enables us to better conceive, thus better express the application's goal.

9.2.2 Better Support and Debugging

With the gathering of data structures and related procedures in a single locus (the object), the localization is better, leading to more direct access and easier debugging.

9.2.3 Data Security

An object is a black box. The OOP insists on the separation between the object's properties, described by related operations, and the internal representation of this object. An object provides the handling interface, while hiding the implementation details.

Seen from the outside, an object will be manipulated only on its properties knowledge, without considerations to its realization.

9.2.4 Flexibility

Internal representation of the objects can be modified, adapted to the hardware and so allow performance optimization, without meddling with the application software.

9.2.5 Recycling

An application can be developed from existing objects. This can speed up software production and decrease the development costs.

9.3 Competence Sharing

The overall process of software development involves three aspects:

9.3.1 The Role of the Application's Conceptor

He has to define the objects in three phases:

1. What are the intervening objects of the application ?
2. What are they doing ?
3. How do they interact with each other ?

9.3.2 The Role of the Objects' Programmer

He will create the objects defined by the first above-mentioned point. The answers of the second and third questions will give him the data structure and the associated operations, as illustrated in figure 23.

② ⟶ Data structure

③ ⟶ Associated methods

Figure 23: Concepts – objects relation.

The objects' programmer should be aware of the hardware, to be able to optimize the objects' code, if necessary. No hardware dependent programming should be done at another level, and even at any level, if possible.

9.3.3 The Role of the Application's Programmer

He uses the objects according to the functionalities defined by the conceptor. He is responsible for the application's functioning optimization.

9.4 Object Oriented Programming

Procedural (algorithms based) languages such as Fortran or Pascal associate data to procedures, but OOP associate procedures to data structures to create objects. New languages with some new characteristic are to be used for objects creation and manipulation. As we'll see in the next sections, the object approach is implemented in these languages, as well as some other ideas allowing an easy and complete implementation of the objects.

First, the notions of abstract data types, which defines meta-objects, and of encapsulation is the implementation of the objects themselves. The concepts of inheritance enables the creation of hierarchy of related data types. The polymorphism allows an object to take several shapes, and the dynamic binding dispatches general calls to specific methods adapted to the object type.

In the object-oriented concept, the communication between the objects is done via *messages*, which are used to schedule the methods.

9.4.1 Data Abstraction and Encapsulation

Every data structure should give rise to a control of its manipulation, in order to guarantee the data consistency. One should think of this structure in terms of the actions to be carried out on it, rather than in terms of its representation.

The definition of a type as the whole set of operations linked to a data structure meet this view, provided that one can only manipulate this structure by these operations.

Such a type, whose name is associated with the data structure and whose internal functioning and representation details are hidden by providing the appropriate operations for the variables of this type is called an *abstract data type*.

The gathering of hidden data structures and appropriate operations is called *encapsulation*. The data structures embedded in an abstract type are called *members data*, and the operations making up its interface are called *methods* and form the *specification* of the abstract type.

The specification should be complete in the sense that no access to a variable of the type should neither be necessary nor even possible, without going through the specified operations.

This will increase the data security.

9.4.2 Inheritance

Inheritance or *class derivation* is a mechanism by which OOP languages allow relations between types and sub-types to be defined.

New abstract types can be defined, sharing the properties (including methods) of an already defined abstract type, without having to re-implement these characteristics. The new type inherits all the members data and methods from a defined type, and may modify some of the already defined methods, as well as it may define some new members data and methods. The figure 24 illustrates this concept.

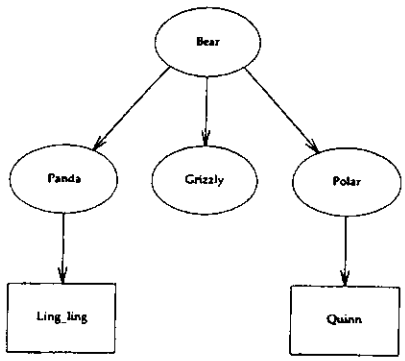


Figure 24: Single inheritance: The bear family class

Some object-oriented languages implement the multiple inheritance concept, allowing a class to be derived from more than one base class, with the aim of inheriting members from different and independent classes. This concept is illustrated in figure 25.

The new type is said to be a *subclass* or *derived class* of the original type, which is a *superclass* or *base class*. We will adopt this terminology from now.

A derived class has obviously to be declared as inheriting, by specifying it's base class.

A derived class can itself be an object of derivation, as seen on the figure 26.

In the base class, the hidden objects have to be declared as accessible to the derived classes: C++ defines three access levels for the members data or functions of a class: public, private and protected.

The *public* declaration in a class enables visibility and access from outside, and usually includes the manipulating functions of the embedded objects.

The *private* declaration is provided for the hidden data structures and functions, that are not accessible, even by a derived class.

The *protected* declared members and methods are only accessible by the derived classes.

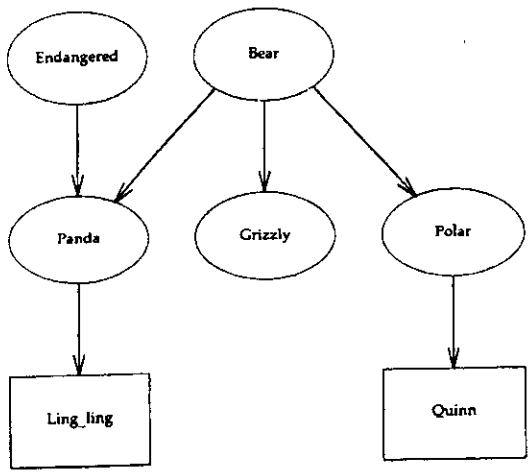


Figure 25: Multiple inheritance: The bear family class, with endangered species indication. Notice that multiple inheritance transform the tree structure of the class hierarchy into a directed graph structure.

The zoo animals fit nicely in an inheritance hierarchy, as already seen in figures 24 and 25. The figure 26 show a three level inheritance hierarchy with multiple base classes and multi-level derivation.

9.4.3 Polymorphism

Derived class variables (objects) can be assigned to its base class variables. Only the inherited methods and data structures will be copied, specific members added after the derivation will be ignored. This rule is very important to guarantee the compatibility between related classes, and implies that an object declared of the base class can take the shape of any object of the derived classes. This peculiarity is known as the *polymorphism* concept.

The special relationship existing between derived classes promotes a *generic* style of programming. The polymorphism mechanism implies the *dynamic*

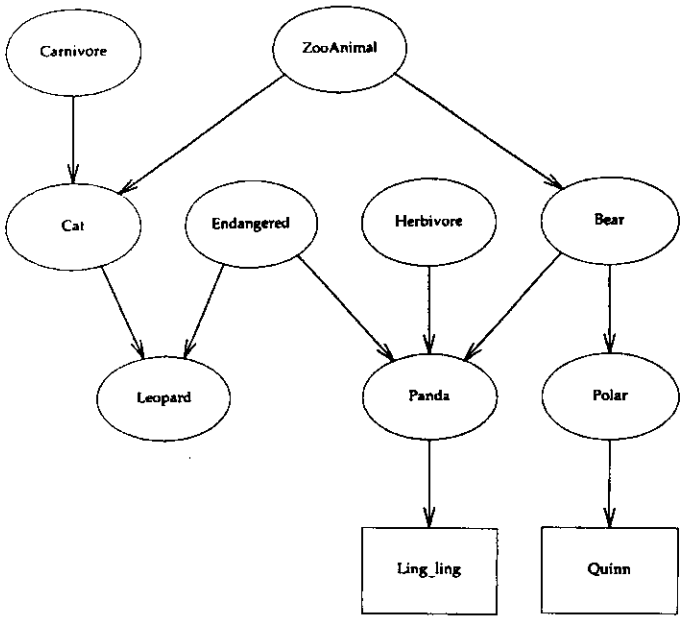


Figure 26: Complete inheritance: A zoo animal inheritance graph.

ently for each derived class, despite it is referred to by the same name for all these classes, and the adequate function can be binded at run-time.

A function should only be declared virtual if the class is supposed to be a base class, the implementation of the function is type-dependent, and it will be called through the base class. In the other cases, the code will be more efficient if the function is declared as a usual member function. The multiple definition process necessary for implementing the class-dependent versions of a function is called *overloading.*

In C++, even basic operators can be overloaded. One may define, for example, a + (plus) operator for adding strings, graphs, or stacks. The multiple declaration of the + operator stands out the necessity to choose from the different functions at some point. If this choice is done at the compilation time, this is called *early-binding* or *static binding.* The *late-binding* approach, where the choice is carried out at run-time, is used with virtual functions (dynamic binding).

9.5 OOP Languages

Languages supporting the different concepts of object-orientation to a certain extent include amongst other Ada, C++, Eiffel, Oberon, Simula, and Smalltalk. We will restrict our view to C++, which is not the most secure and consistent object-oriented language, but which is compatible with its predecessor, the C language, with all the advantages and drawbacks this compatibility involves.

The Eiffel, Simula and Smalltalk are real object-oriented languages, while Ada and Oberon are conventional languages with minimal object-oriented programming support: the concept of object is defined, but neither classes, nor inheritance, even though these concepts may be simulated with some programming effort.

9.5.1 C++

C++ is an extension to C language, for supporting object orientation. The most important extensions from this point of view are

- data abstraction
- operator overloading
- classes with multiple inheritance
- objects with dynamic binding

Abstract data types: Classes

Classes consist of data and functions members, and is divided into a public and a private part. The public part describe the interface, while the private part is inaccessible for heirs and clients. Members may be declared as protected in order to be used by subclasses.

The declarative part of the class is stored in a separate header file, inserted in each file using the class, as well as in the implementation file for the class itself.

Inheritance: Derived classes

Publicly inherited members are public both in the superclass and in the subclass. Privately inherited members cannot be accessed from outside the subclass, and thus inhibit polymorphism.

Multiple inheritance is supported. Virtual derivation allows multiple inheritance while avoiding multiple copy of inherited parts.

Polymorphism:

The assignation of subclass objects to variables of the superclass is allowed, with the previously mentioned exception.

Dynamic binding:

Virtual functions allow to override inherited methods. Functions declared as such in the superclass are dynamically binded.

Objects:

Objects are created either by declaration or by the *new* operator.

An initialization procedure called *constructor* is automatically called by the compiler each time an object is created. This procedure has the same name as the type. Another function of each class is the *destructor* which is used to delete objects, as well as the memory allocated by those objects.

10 Real-Time Systems

Real-time applications are characterized by the strict requirements they impose on the timing behavior of their system. Systems ensuring that those *timing requirements* are met are called *real-time systems.* We will exclude from the beginning the *transactions processing systems* (seat reservations, banking), where the transactions are done in real-time, but without any constraint.

10.1 Concurrent and Real-Time Concepts

A *concurrent program* is a non-sequential program, in the sense that some operations are performed simultaneously. This technique, obviously useful in the case of a multiprocessor system, can even be attractive in a monoprocessor environment, to take full advantage of the independence of the processor and the peripherals.

Consider for example that we want to write characters on a terminal. The figure 27 illustrates the activities of both the processor and the terminal interface.

- The processor has to wait until the terminal is ready to accept a character, it then sends the character to the interface and loops back to its waiting state.
- The interface waits for a character, accepts it, write it to the screen and loops back to its waiting state.

That description shows that both processes are waiting for an information given by the other party, before doing any useful task. This is solved by task or process *synchronization*. In this example, the synchronization is done for one way by an interrupt, and for the other direction by means unspecified at this point. There are several mechanisms able to signal that the character is ready to be processed by the interface process.

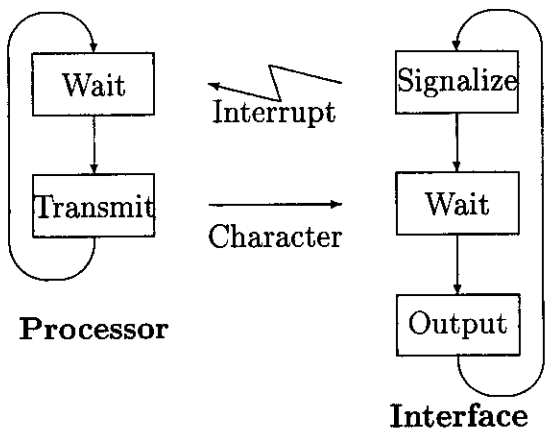


Figure 27: Respective activities of processor and terminal interface for writing a character.

During this time, a concurrent program can perform another task!

Of course, even with the synchronization, one of the two processes will be faster than the other. In our example, the processor will be mostly waiting for the interface to be ready.

Concurrent tasks should avoid accessing shared data simultaneously. This could lead to incoherent informations if two processes write at the same time in a data structure. Concurrent programs always present these two problems:

- Mutual exclusion (Critical resource access).

- Synchronization between processes.

These problems are solved by tools (mechanisms) specific to concurrent programming, called *locks*, *events*, *semaphores*, *monitors*, *mailboxes*, *rendez-vous* or *interrupts*.

A *real-time program* is very much like a concurrent program. It has to manage peripherals, and the mechanisms mentioned above still apply. A real-time program includes a supplementary issue: *timing constraints* imposed by the fact that a real-time program controls an external system.

With the improvement of the performance of the microcomputers, and as their price, size, weight, and power requirements decrease, real-time systems are more and more widespread.

Current fields of applications include scientific instrumentation, medicine, industry, cars and military. For example, a real-time system may drive and monitor an astronomical telescope or an X-ray medical scanner, control an industrial production line or a car motor and navigation system, as well as drive a weapon delivery system or control a entire nuclear power plant.

You have noticed that the word control or a synonym come often in those examples:

Timing and *control* are the master-words in the real-time systems world.

In general, we'll call real-time system any system meeting external timing constraints and able to solve these constraints during its execution; without any specification on the architecture of the system.

A Real-time system can be divided into two groups: The *hard real-time systems*, for which a failure to meeting the timing constraints is considered as a major failure (crash) of the system, and the *soft real-time systems* that will give an error or a warning on such failures, without stopping execution.

10.2 Embedded and Distributed Real-Time Systems

Many complex systems require nowadays an elaborate control system to support their internal functioning. Such systems often use a dedicated computer as controller. Such a computer is called an *embedded computer.*

An embedded computer system has to control the rest of the system. It gets information like data and status from sensors, then issues control commands to actuators.

One feature that distinguishes embedded systems from other real-time sys-

tems is that they are only executing a task relative to a fixed and well-defined workload. They don't provide any development environment.

Study of embedded systems must consider the controlled system as a whole: In particular, mechanical, electro-mechanical parts and electronics should be considered at the specification level of such a real-time embedded system.

The most general way of defining a real-time system is to consider a *multi-machine, distributed computing environment.* The term multi-machine implies that, in addition to the internal timing constraints due to its peripherals, each machine (node) has to deal with timing constraint requests of the other nodes of the system.

10.3 Implementation Issues

Most of the real-time applications cannot be programmed with traditional languages under a traditional operating system, or at least at their standard level, as those languages don't know how to handle the timing constraints imposed by the system. Additional features known as *real-time extensions* are defined for some languages, enabling such systems to be programmed and checked. These extensions often enable the programmed real-time system to override the operating system mechanisms to control directly the hardware.

On the other hand, real-time systems can be programmed with classical languages such as C, if there is a library of functions implementing the real-time mechanisms. In this case, the real-time aspects of the application is shared between the language and the real-time operating system (LynxOS, OS/9).

Another aspect of the implementation of complex, multi-machines real-time applications is the operating system. The traditional approach to multi-tasking operating systems design is to split the time in slices and to attribute those slices to the different computing-resources demanding applications. This kind of management is called *time-sharing.* Time-sharing doesn't address correctly the problems arising in real-time systems.

So, the execution of real-time applications has to be supported by a correct environment, which is obtained through a *real-time operating system.*

These real-time operating systems have to manage timing and interactions problems. Different mechanisms allow them to handle timing constraints correctly, including *interrupts* and *signals.* They also contain mechanisms to solve the processes scheduling problem, that can be quite difficult, with

preemptive tasks and *dynamic priority* setting. Another aspect treats the communications between tasks, with *semaphores* and *shared data* zones.

10.4 Time Handling

Time handling is the most important issue in real-time systems. Time handling includes:

- Knowledge of time
- Time representation concepts
- Time constraints representation

10.4.1 Knowledge of Time

Time is given by *clocks*. In a multi-machine environment, multiple clocks may exist and should be *synchronized*, in order to get a coherence between the different timing constraints and interactions specifying the real-time system.

A clock is characterized by its *correctness*, which defines the quality of the knowledge of time, and by its *accuracy*, which defines the way the clock *drifts*. The accuracy is given by the derivative of the clock signal, as shown by the following definitions:

A *standard or reference clock* is one for which the relation

$$C(t) = t \ , \forall t$$

is confirmed. A clock is *correct* at time t_0, if

$$C(t_0) = t_0$$

A clock is *accurate* at time t_0, if

$$\left.\frac{dC(t)}{dt}\right|_{t_0} = 1$$

10.4.2 Clock Systems

There are different clock systems.

The simplest one consists of one central *clock server*, that should be very accurate and reliable, even though a redundant system can be used. Therefore, this kind of clock system is quite expensive.

Another type of clock system defines a *master clock* polling multiple slave clocks, measuring their differences and sending to them the corrections to do. All the clocks can be of the same accuracy, and if the master fails, another one amongst the other is elected to become the new master. This type of clock system is called *centrally controlled.*

A *distributed clock system* consists of an interlinked network of clocks, which all run the same algorithm, polling the other clocks to get their time, and then estimate their correctness. This type of system can be simple or enhanced, depending on the complexity of the algorithms used at the nodes, and implies a *relatively heavy traffic load* on the communication network.

The graph linking the nodes can be *closely connected*, with any of the clock polling all the others, or *loosely connected* with only a subset of the connections used for time synchronization.

A protocol named *xntp* working through network with the *UDP protocol* is publicly available, and works as a distributed clock system with a hierarchy defining more or less reliable clocks. This hierarchy is organized in levels (strata), a lower level number meaning a more preemptive clock. Each node can be configured to communicate with a certain number of other clocks, either for synchronizing itself (same or lower levels), or to only read the time on higher level clocks.

The Global Positioning System (GPS) is a satellite based navigation system providing precise position, velocity and time information. The heart of the GPS consists of 21 satellites and three spares, that revolve round the earth twice a day, at an altitude of 20000 km. They allow a 24 hours per day worldwide coverage by more than 3 satellites. This system can be used by special hardware to get a good timing information to synchronism clocks. The receivers are cheap (about $ 600-1000).

Other special hardware may take advantage of the time signals broadcasted by radio waves from different standard clock systems in the world, as DCF in Germany, WWV in Boulder, Colorado, WWVH in Hawai or JJY in the Pacific North.

10.4.3 Time Representation

Time representation in real-time systems should be sufficiently well-designed to take into account the properties of the system, and to allow a precise definition of the characteristics of the time constraints.

As a preliminary definition, we should state that the *time granularity* of a system is the clock resolution. This notion is more complex than it seems. Each operating system uses a system clock (fig. 28a) to manage the timing synchronisation between processes. This clock gives interrupts to the system at a certain rate, which can usually be modified, but which should neither be too high, for fear of excessive system overhead, nor too low, because it would penalise the interactive processes by a long response time. This time is usually about some tens of milliseconds. This gives the granularity for scheduling processes, or time-slicing in a classical operating system.

There is another clock used for time measurement (fig. 28b), which can also be used to drive a programmable timer for scheduling events at certain time. This is called the real-time clock, and has a granularity of about microseconds. A real-time operating system will usually use this clock to synchronise the processes or manage timing constraints.

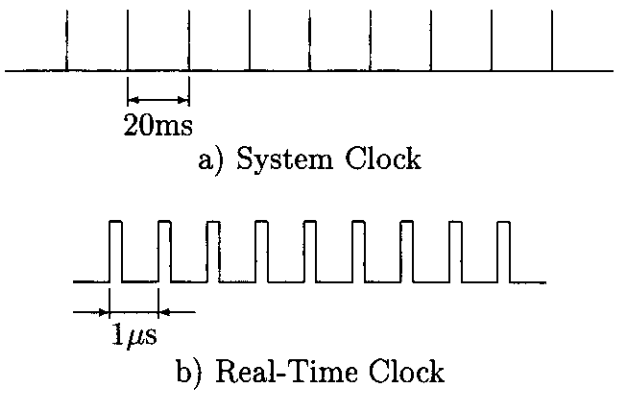


Figure 28: Different clocks are defined in a system.

Point-based representation defines events of zero-length duration, occurring at some time instants in a system, which are responsible for a change in the state of the system.

Interval-based representation defines activities of finite duration, having a start and a stop time. These activities can exist simultaneously.

Both approaches have their drawbacks:

Point-based-representation	Interval-based-representation
Events cannot be decomposed while maintaining an order, as they have no duration.	It is difficult to take into account the time granularity of the system.
Partially overlapping activities cannot be described by this model.	

The best solution is highly dependent of the system, but will often be based on a compromise between both approaches, leading to an *interval based representation, with system's granularity support.*

10.4.4 Timing Constraints Representation

A real-time system has to deal with the arrival of time-constrained requests, i.e. the invocation of processes to be executed in due time.

The system has to allocate the resources to meet the specifications, in order that the process can begin at a specified time, and be completed at another specified time.

The minimal definition of a timing constraint is the triple

$$(Id, T_{begin}(condition1), T_{end}(condition2))$$

where Id is the name or ID-number of the process.
$T_{begin}(condition1)$ is the starting time of the process.
$T_{end}(condition2)$ is the completion time of the process.

Depending on the system and the temporal uncertainties on the allocation time of certain resources, we may need some additional time parameters in the constraint representation.

In particular, the completion time may not be a very severe constraint, and in case of earlier process completion, the resources should be freed for other processes.

On the other hand, a very long process should not monopolize the resources of the system, and the global efficiency of the system would be improved, if time-slices were attributed to this process.

This leads to the more mature definition of a timing constraint as the quintuple

$$(Id, T_{\text{begin}}(\text{condition1}), c_{\text{Id}}, f_{\text{Id}}, T_{\text{end}}(\text{condition2}))$$

where	Id	is the name or ID-number of the process.
	$T_{\text{begin}}(\text{condition1})$	is the starting time of the process.
	c_{Id}	is the computation time of the process, or the time-slice.
	f_{Id}	is the frequency with which the time-slices have to be attributed.
	$T_{\text{end}}(\text{condition2})$	is the completion time of the process.

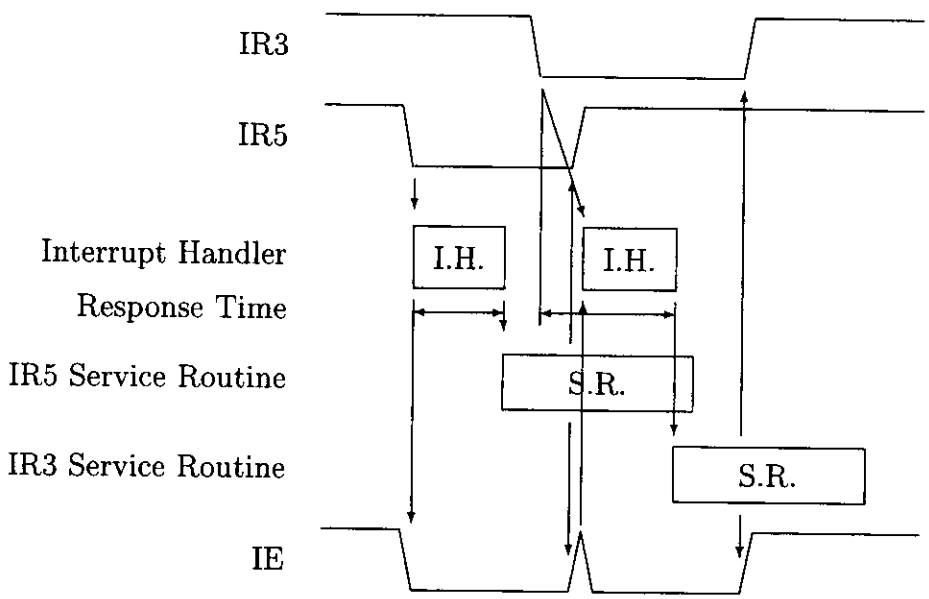


Figure 29: Interrupt Service Scheme

10.4.5 Interrupts driven Systems

Interrupts are often used as a *synchronization mechanism* in real-time systems, particularly in control applications.

An *interrupt* is a signal occurring asynchronously and triggering a *service routine*. This routine is called by the *interrupt handler*, which identifies the

interrupt, locates in a table the appropriate address, and passes it to the program counter (instruction pointer). The handler or the service routine itself has to save the current environment before beginning processing the request, as it could modify this environment.

A signal enabling the interrupt system (IE) is disabled by the acceptance of an interrupt by the handler. It is usually the service routine's responsibility to re-enable it, at some time. In the figure 29, we have a first interrupt arriving (IR5). The interrupt handler accepts it, as there are no other interrupts being processed, and passes control to the IR5 service routine. A second non-preemptive interrupt arrives before the routine has released the IE signal. This interrupt is blocked for a while, until the interrupt handler being re-enabled. Then it is normally processed. This illustrates the fact that response time to interrupt may vary.

The routine has to be carefully designed to meet the time constraints on it's duration, deadline and frequency. Sometimes, the task has also a starting time condition, in which case it can be executed only if both the interrupt has occurred, and the starting condition is met.

10.4.6 Signal Synchronization

Another way to synchronism processes is to signal certain states of the system. Typically, one process needs the system to be in a certain state which it cannot control for continuing it's execution. Arrived at that point, it checks a signal specifying the desired state, and if unsatisfied, waits until the signal arrives, indicating the change in the system state.

On the other hand, another process is responsible of modifying the state of the system, and has to signal it after completion. This method leading to *mailbox* or *rendez-vous synchronization* does not fit well to real-time systems, because it cannot ensure that deadlines are respected, and is mainly used for concurrent processing.

10.5 Real-Time Systems Design

The design of any system should begin by a *requirement specification* phase, followed by the design phase itself. These phases will be followed by the implementation, tests, etc. The design phase can also be decomposed into a preliminary and a detail phase. The different phases and sub-phases may sometimes overlap each other in time.

Take care that a too rigid approach in the design, obtained for example by avoiding any time-overlap between phases, may lead to a very formal and well-documented design, but that will possibly be neither creative nor the best one.

Another aspect is that a project is in itself very much like a "real" real-time system, with timing constraints and deadlines. To achieve a project in the specified delays, one will tend to minimize the specification and design phases to begin as quickly as possible the implementation. This attitude may lead to a badly-designed and possibly fragile system. A better way is to begin the implementation of well-designed parts while refining the design of the rest, ensuring both a good overall design and a quick development of the system.

Let's examine the two phases of the design.

10.5.1 Requirements Specifications

The requirement specification phase is important in real-time systems, because the descriptive aspect of the document enables to easily include the timing constraints.

The requirement specification document should:

- state external behavior of the system.
- avoid specifying any implementation details, but only constraints on the implementation, as the details of the hardware interface.
- state the responses to the exceptions.
- be easily modified.
- be well documented to serve as a reference during all phases of the project.
- specify the timing constraints and deadlines of the project itself.

Some systems may be described in a verbose documentation style only, while others may need some more sophisticated tools as, for example, *state-charts.*

10.5.2 State-Charts

State-charts describe the system as *states* and *transitions* between them, triggered by *events* and *conditions.* States are represented by boxes, transitions by arrows, events and conditions are labels for the arrows (figure 30).

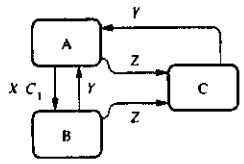


Figure 30: State-chart example.

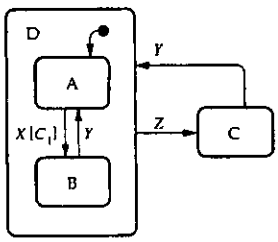


Figure 31: Clustering states in a state-chart.

States can be decomposed to lower level states or combined into a higher level state (figure 31). These operations are called *refinement* and *clustering*.

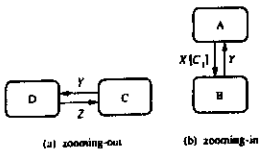


Figure 32: Zooming in and out.

Zooming in and out (figures 32) enables one to have different levels' views of the system.

10.5.3 Petri Nets

The complexity of real-time systems is essentially due to the interactions between tasks, the access conflicts and the temporal evolution of the system. It is necessary to use powerful tools to represent the evolution of such a system at the conception level. The *Petri net representation* is a very powerful tool, which enables to represent the interactions between processes and the evolution of processes.

A Petri net is a quadruple $C = (P, T, I, O)$ including N *places* $p_i \in P$ and

L *transitions* $t_i \in T$. The structure is described by two matrices I and O of dimension $L \times N$ specifying *inputs* and *outputs* viewed by the *transitions.*

The elements of those matrices are integers specifying the weight of the link between a place and transition. The absence of a link is obviously described by a weight $w = 0$.

A Petri net can be represented by a *Petri graph*, with two types of nodes: places and transitions. The directed edges may only link nodes of different type. As an example, a Petri net described by

$$
\begin{aligned}
C &= (P, T, I, O) \\
P &= |p_1, p_2, p_3, p_4, p_5| \\
T &= |t_1, t_2|
\end{aligned}
$$

$$
I = \begin{array}{c|ccccc|c}
 & p_1 & p_2 & p_3 & p_4 & p_5 & \\
\hline
 & 1 & 1 & 2 & 0 & 0 & t_1 \\
 & 0 & 0 & 0 & 0 & 1 & t_2
\end{array}
$$

$$
O = \begin{array}{c|ccccc|c}
 & p_1 & p_2 & p_3 & p_4 & p_5 & \\
\hline
 & 0 & 0 & 0 & 1 & 2 & t_1 \\
 & 0 & 0 & 1 & 0 & 0 & t_2
\end{array}
$$

is represented by the graph of figure 33

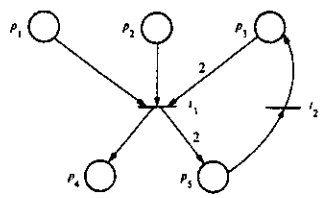


Figure 33: Petri graph with weighted arcs

This definition of a Petri net enables only the static representation of a system. To modelize the temporal evolution, the Petri net is completed by *marking.* A marked Petri net represents a state of the system. Marking *tokens* are represented by dots on the graphs (fig. 34).

A marking is a N-dimensional vector specifying the numbers of tokens in each place. The system becomes dynamic when the tokens travel through the net. The traveling is done through *transition firing.* A transition may be fired only if all the preceding places are marked (active). This transition is said to be enabled.

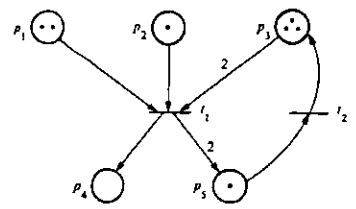


Figure 34: Marked Petri graph

Only one transition is fired at a time, randomly chosen between enabled transitions. A firing has the following effects on the places preceding and succeeding the transition:

- w token is removed from each preceding place.
- w token is put in each following place.

Firing is:

Voluntary An enabled transition may be fired, but it is not mandatory.

Instantaneous All the operations related to a firing occur simultaneously, and take no time.

Complete All the operations related to a firing do occur.

The figure 35 shows the result of firing transition t_1 in figure 34.

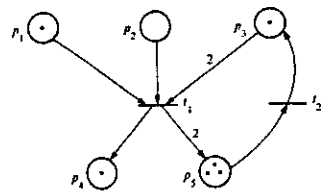


Figure 35: Petri graph after the firing of t_1.

A Petri net may be annotated as shown in the figure 36 illustrating the allocation of a processor: As soon as the processor is idle (p_2 marked) and there is a task waiting in the queue (p_1 marked), the processing may begin (t_1). The task is executed (p_3 marked). At the end (t_2), the task is completed (p_4 marked), and the processor is deallocated (p_2 marked).

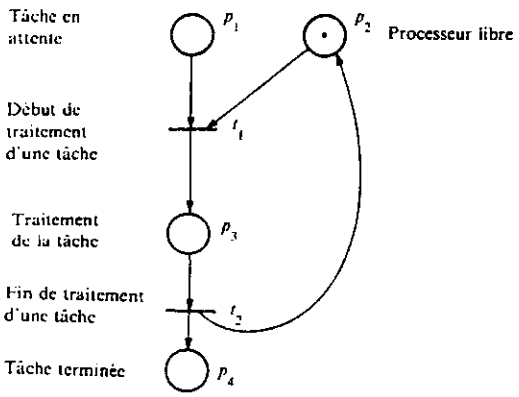


Figure 36: Petri net modelizing a processor allocation.

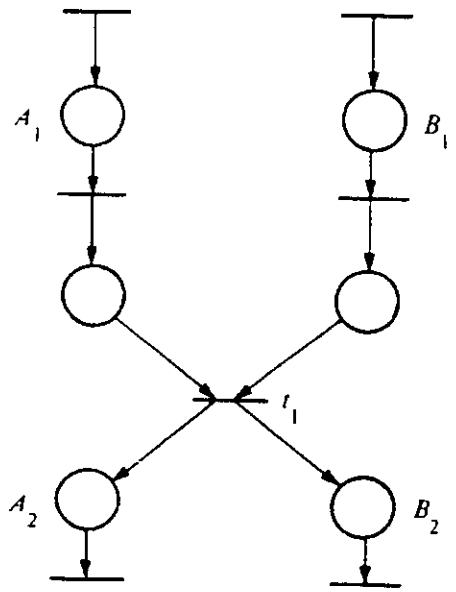


Figure 37: Petri net modelizing a *rendez-vous* type synchronization.

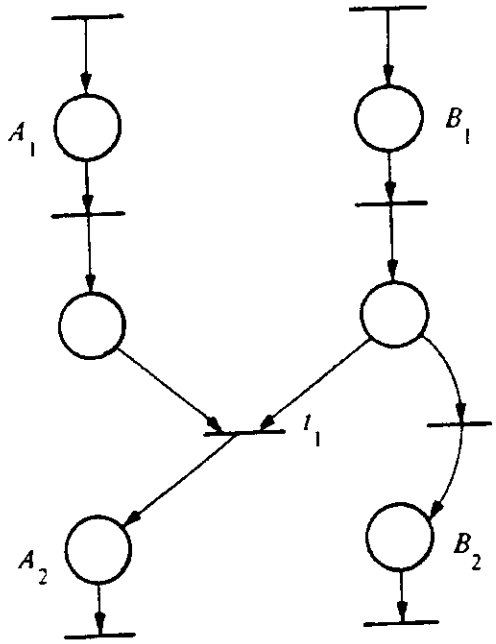


Figure 38: Petri net modelizing a *mailbox* type synchronization.

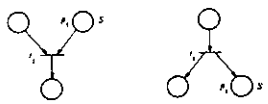


Figure 39: Petri net modelizing the semaphores primitives $P(s)$ (left) and $V(s)$ (right).

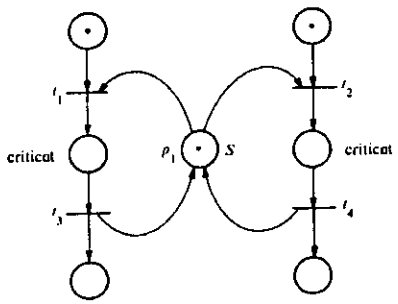


Figure 40: Petri net modelizing mutual exclusion by semaphore.

Without going into the details of the Petri net model, we can say that conditions are associated to places, and events to transitions. The figures 37-40 show Petri nets representing some real-time issues.

The Petri net model may be used by the designer in a kind of top-down structured approach (figs. 41-44) :

- Start with a global Petri net model of the system (fig. 41).
- Stepwise refine it by substituting (fig. 43) the transitions by *well-formed blocks* (fig. 42) . A well-formed block should have only one input and one output (fig. 44).

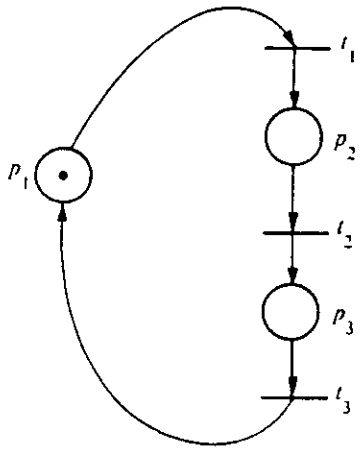


Figure 41: Initial step for structured design.

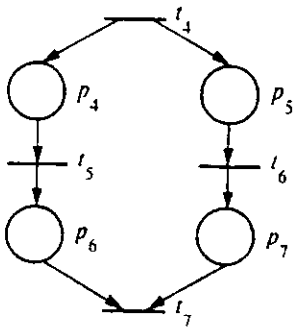


Figure 42: Block example.

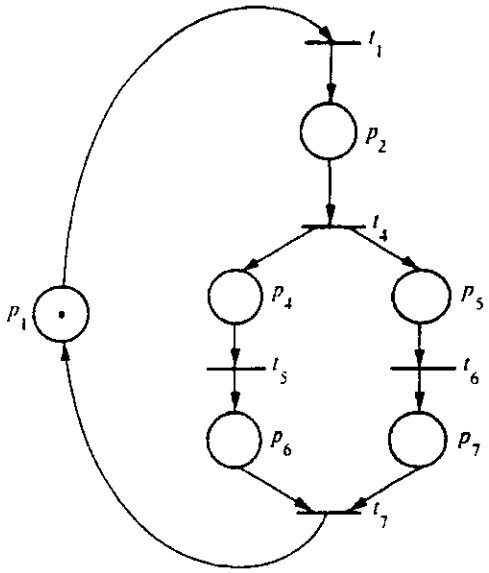


Figure 43: Replacement of t_2, p_3, t_3 in fig. 41 by the block of fig. 42 .

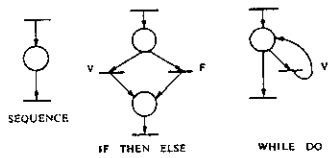


Figure 44: Well-formed blocks.

The Petri nets can be transformed to flowcharts. The nodes of the flowcharts are associated to the Petri net transitions, while the arcs will replace the places (figs. 45 and 46).

10.6 Structured design of Real-Time Systems

In addition to the concepts of structured design, we have to address the notions of timing constraints and interprocess communications. *DARTS* (Design Approach for Real-Time Systems) was developed by General Electric to extend the notion of structured design to include *process decomposition* and *process interfacing.*

First, an analysis of the system has to be done in terms of functions: The system is then viewed as a *data flow* transformed by *functions.*

10.6.1 Process Decomposition

When the functions have been identified and described, they must be assigned to processes. DARTS defines criteria to assign a function to a separate process, or to group it in a process with other functions:

I/O dependency If a slow peripheral dictates the speed of execution of a function, this function should be put in a separate process.

Time-critical functions High priority functions should be kept in a separate process.

Computational requirements Intensive computation functions should receive a separate process.

Functional cohesion Closely related functions should be grouped in a process.

Temporal cohesion Functions triggered by the same stimulus should also be grouped.

Periodic execution Periodically executed functions should be kept in a separate process.

So we see that functional and temporal cohesion are a criterion to group function in a single process, where they can still be separated and distinguished by creating modules inside the process. Timing constraints and special requirements justify on the other hand separate processes.

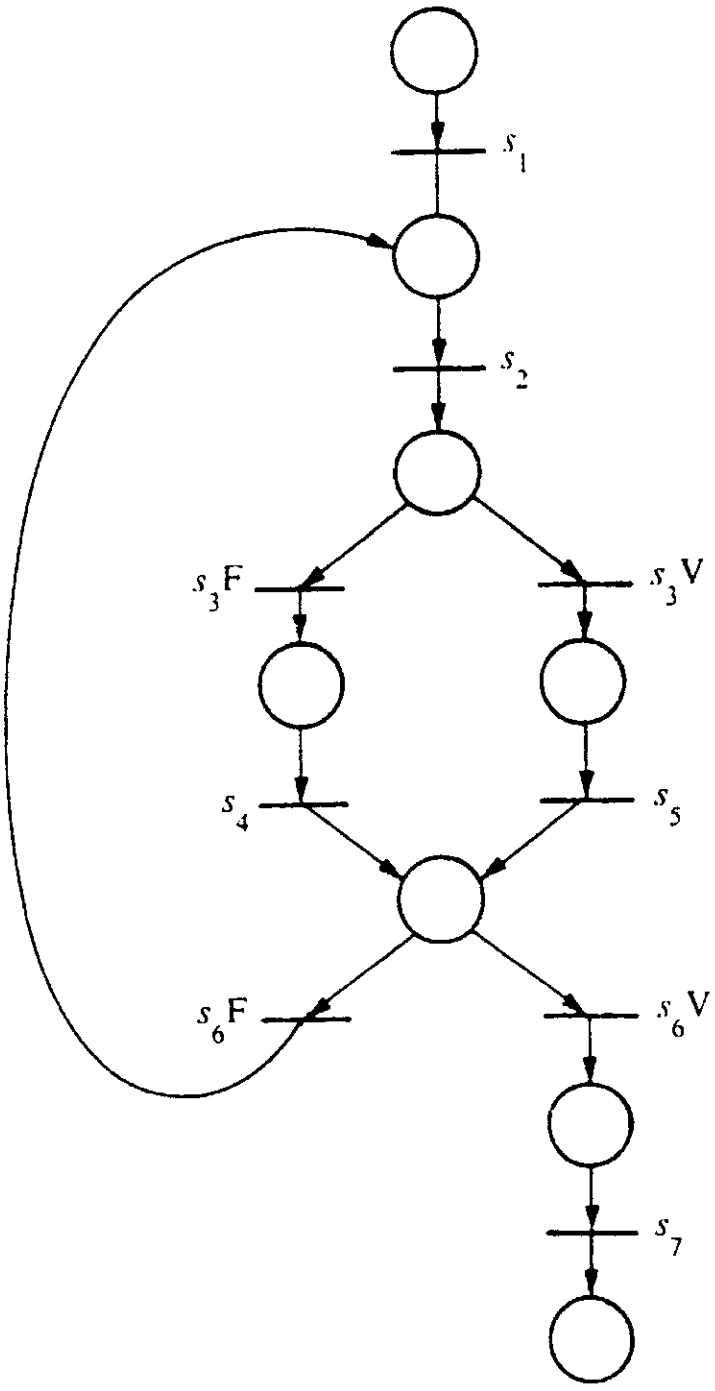


Figure 45: Petri net example.

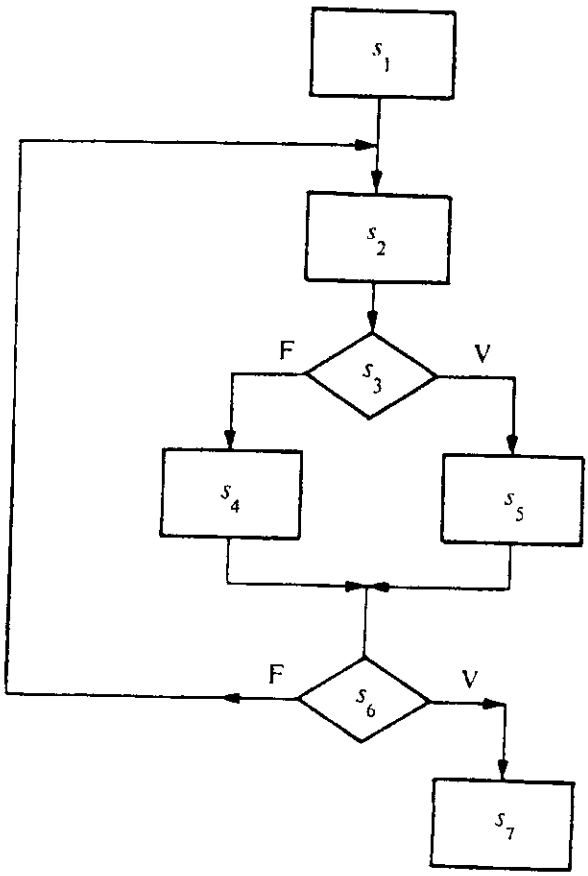


Figure 46: Flowchart for the Petri net of the figure 45.

10.6.2 Interprocess Communication

DARTS provides two types of modules for the communication between processes:

- Message communications modules (MCM).
- Information hiding modules (IHM). It is used mainly in cases of shared data. IHM defines the data structure in a hidden way, with procedures to access it.

The figure 47 shows three processes P_1, P_2 and P_3 communicating through the data they share, and which is defined in the module IHM, with the data hidden in structures B and C, accessed only through the procedure a.

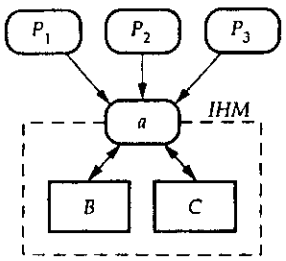


Figure 47: IHM module

Please notice how close this approach is from the object concept.

10.7 Example of a concurrent problem

We want to implement a stop–watch that displays on a terminal screen the times in a format like `00:00:00.0`.

On initialization, the time is `00:00:00.0`. Then, keyboard "one–key" command are driving the instrument:

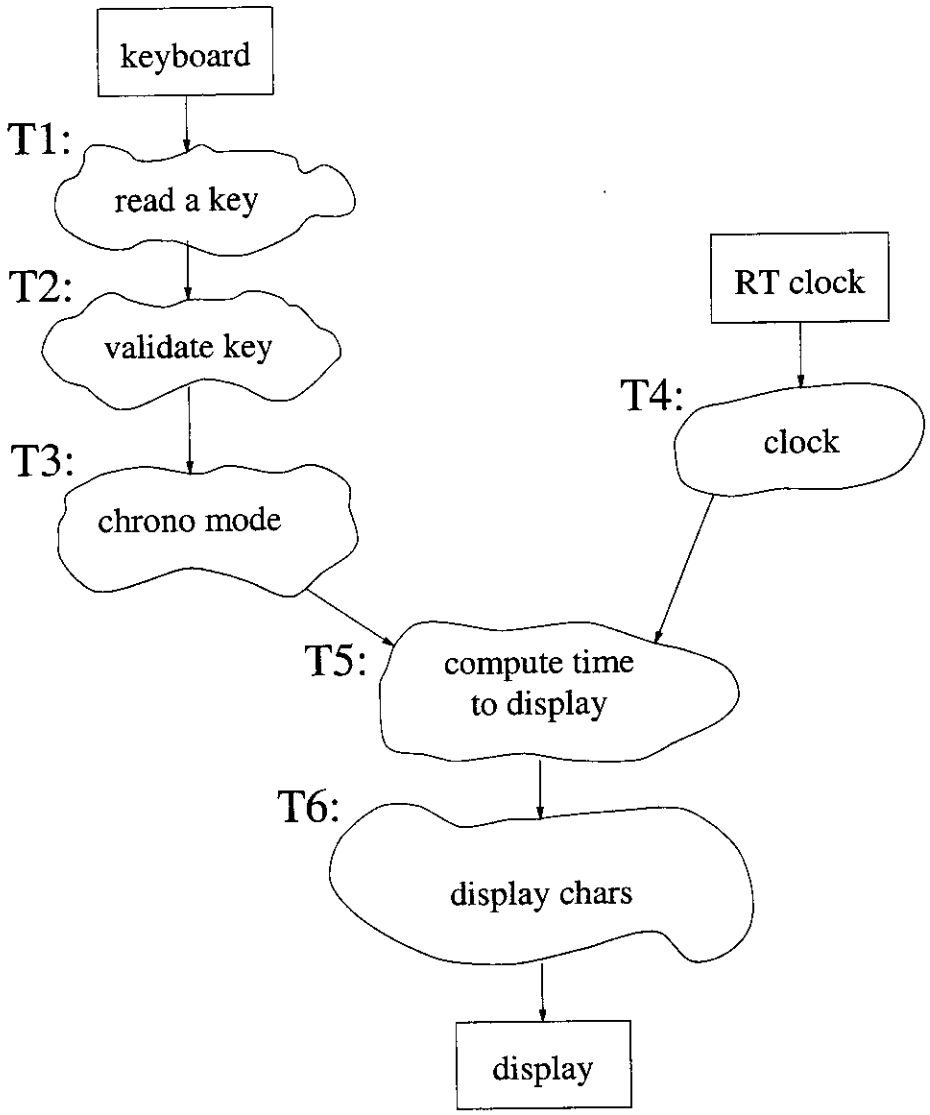
W	: s.chrono→s.watch	mode selection
C	: s.watch →s.chrono	mode selection
H	: s.watch →increment	hours
M	: s.watch →increment	minutes
S	: s.watch →increment seconds	
G	: s.watch →r.watch	
T	: r.watch →s.watch	
R	: r.watch →e.chrono	
I	: r.chrono→i.chrono	
F	: r.chrono→f.chrono	

where:

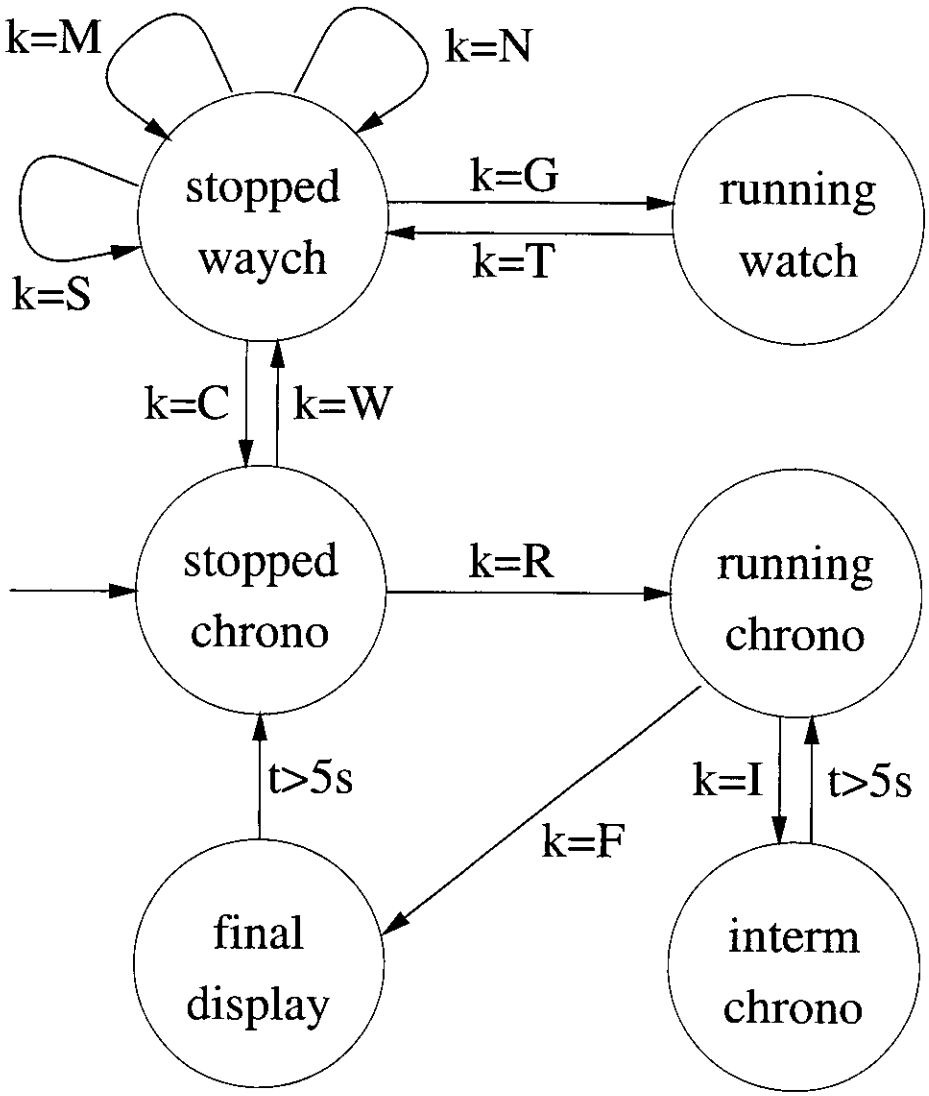
s.watch, s.chrono	: stopped watch and chrono modes
r.watch, r.chrono	: running watch and chrono modes
i.chrono	: intermediate display for 5 seconds
f.chrono	: final display for 5 seconds

In the following pages, we show how this problem can be analyzed using Data flow, State chart, Flow chart and Petri Nets.

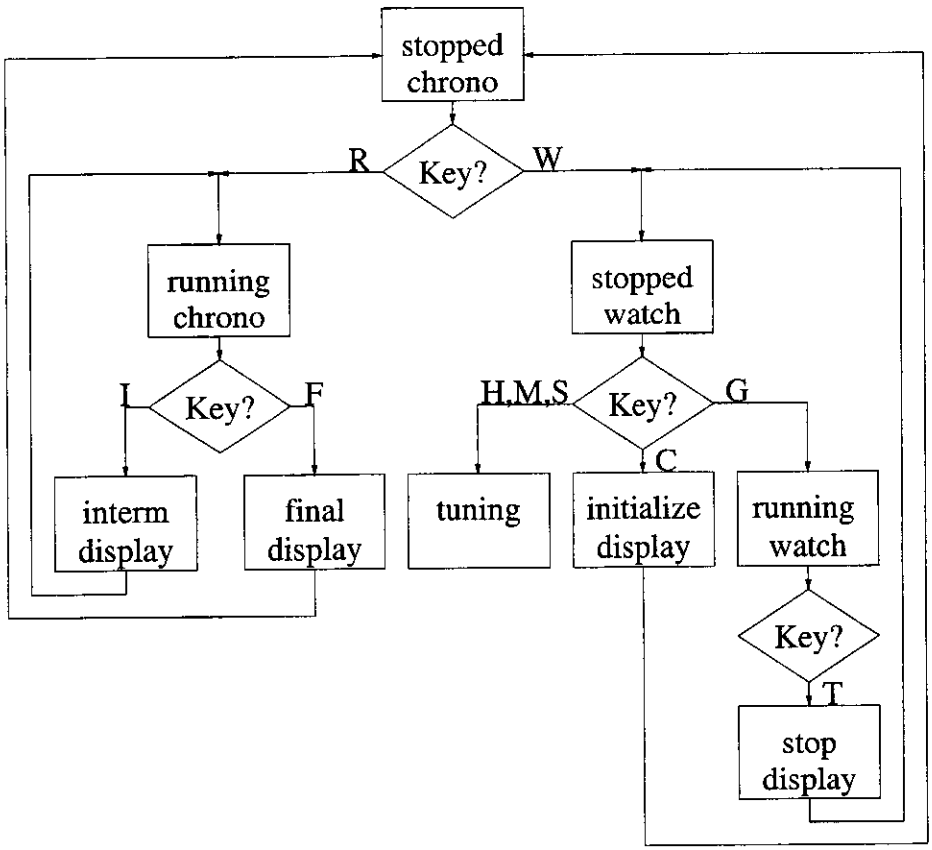
10.7.1 Data flow study



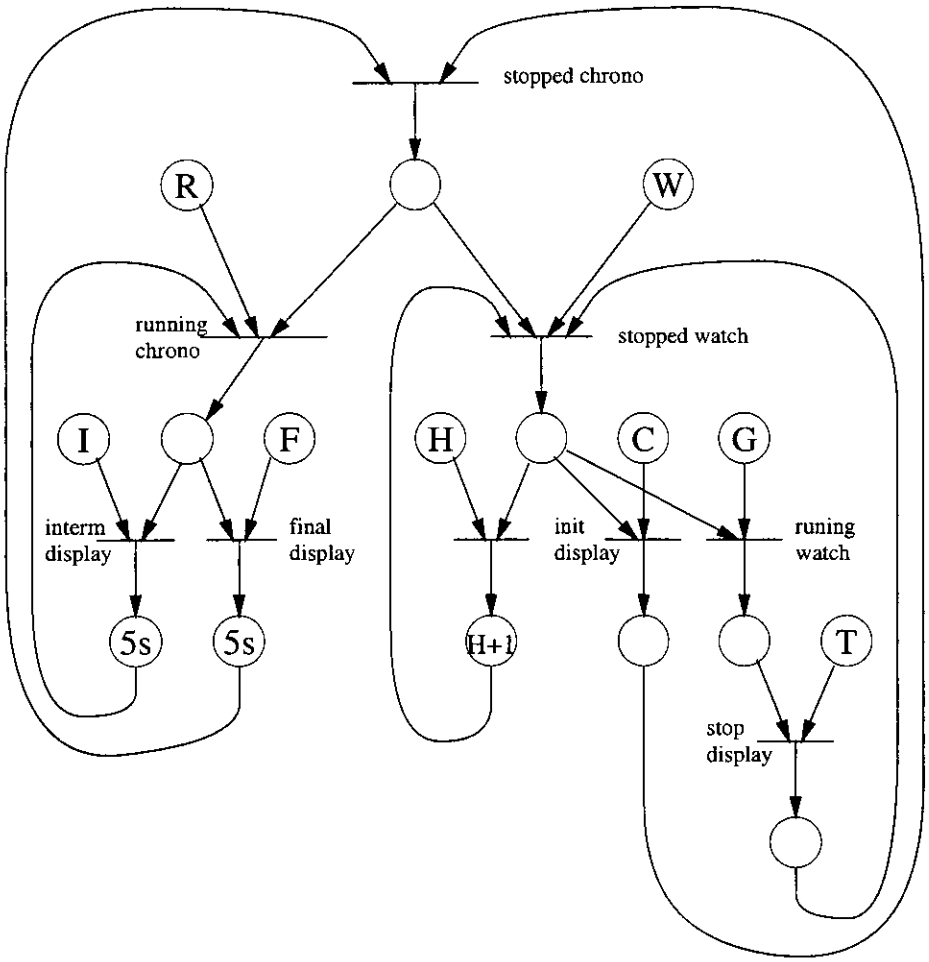
10.7.2 State chart study



10.7.3 Flow chart study



10.7.4 Petri Net study



10.7.5 Process decomposition study

According to DARTS, we have to select the data flow transforms which will receive a separate process.

I/O dependency: A process should be given to transform T1 (keyboard input) and to transform T6 (display output).

Time critical function: none.

Computational requirement: none.

Functional cohesion: same process for T2 (validate key) and T3 (choose mode).

Temporal cohesion: T5 (compute time to display) is to be put in the same process as T2 and T3, or as T4, Mode dependent.

Periodic execution: T4 (clock) is to be executed at each RT clock tick. Should be kept in a separate process.

11 Use of `man` pages, `apropos` and `info`

11.1 `man` and `apropos`

One should not forget all the man pages, either interactively on the screen, or in printed form. The man pages for gcc, in particular, are very detailed.

When printed pages are really needed, they can be produced with

```
man command | lpr
```

or, if troff is installed,

```
man -t command
```

`man -k keyword` and `apropos keyword` can be used to retrieve command names that are related to some keywords.

Here is an example:

```
obssq18:~ 551> apropos administration
admind          admind (1m)      - distributed system administration daemon
admintool       admintool (1m)   - system administration with a graphical user interface
dispadmin       dispadmin (1m)   - process scheduler administration
nis_checkpoint  nis_ping (3n)    - misc NIS+ log administration functions
nis_ping        nis_ping (3n)    - misc NIS+ log administration functions
nisgrpadm       nisgrpadm (1)    - NIS+ group administration command
nistbladm       nistbladm (1)    - NIS+ table administration command
nlsadmin        nlsadmin (1m)    - network listener service administration
pmadm           pmadm (1m)       - port monitor administration
sacadm          sacadm (1m)      - service access controller administration
obssq18:~ 552>
```

11.2 info

info is an interactive hypertext system that is replacing man for documentation, in particular for all recent GNU products. info can be used on any terminal, not necessarily in an X-window.

The information is organized in a tree-like fashion, but can be accessed directly on any leave.

It is invoked as:

```
info keyword
```

where *keyword* is a leave (concept, command or subcommand). If the *keyword* is missing, info starts at the root of the documentation.

Here is for example what info without any parameter returns on my system.

```
File: dir       Node: Top       This is the top of the INFO tree
  This (the Directory node) gives a menu of major topics.
  Typing "d" returns here, "q" exits, "?" lists all INFO commands, "h"
  gives a primer for first-timers, "mEmacs<Return>" visits the Emacs topic,
  etc.
  In Emacs, you can click mouse button 2 on a menu item or cross reference
  to select it.

* Menu: The list of major topics begins on the next line.

* As:   (as).            GNU assembler 'as'.
* Bison: (bison).        GNU version of yacc grammar parser.
* Cfengine: (cfengine). System configuration management.
* Cpio: (cpio).          GNU version of cpio.
* Flex: (flex).          GNU version of lex lexical analyser.
* Gasp: (gasp).          Preprocessor for assembly programs.
* Gdb:  (gdb).           GNU debugger.
...
```

The first lines remind briefly how to use info, and, under the menu, are the sub-top leaves. Putting the cursor on any line starting with an * and pushing the *return* key will show the content of the selected material, which itself may contain other sub-leaves etc.

12 Think

Think !

- think before doing
- think while doing
- think after having done
- your are responsible, you are the master never give it to μP
- μP must obey, not dictate

Think small !

- 'Small is beautiful'
- keep things manageable, under control
- use small modules

Think with others !

- do not reinvent the wheel
- make your work sharable
- build-up libraries
- accept help, call for help
- the others can and must think too

Think on your own !

- do not accept buzz words for granted
- adapt to your own country
- do not destroy your richness
- never accept dogma

13 References and Bibliography

The following bibliography is not necessarily very coherent. It contains old and new books, as well as some reference articles. They are all in my personal library. I have not read all of them, but they all contain something that impressed me and changed my way of using computers.

Many of these books have been reprinted, some re-edited, and the dates given may not be uptodate.

13.1 Structured Programming

- **Dahl O., Dijkstra E.W. and Hoare C.A.R.,** Structured programming, *Academic Press 1972*
- **Dijkstra E.W.,** A discipline of programming, *Prentice-Hall 1976*
- **Kernighan B. W. and Pike, R.** The Practice of Programming. *Addison-Wesley 1999*
- **Kruse R.L.,** Data structures and program design, *Prentice-Hall 1984*
- **Wirth N.,** Program development by stepwise refinement, *CACM* **14**, *221-227 (1971)*
- **Wirth N.,** Systematic programming, *Prentice-Hall 1973*

13.2 Algorithms & Data Structures

- **Bentley Jon** Programming Pearls. *Addison-Wesley 1989*
- **Bentley Jon** More Programming Pearls, Confessions of s Coder. *Addison-Wesley 1988*
- **Knuth D.E.,** The art of computer programming, vol. 1 : Fundamental algorithms, *Addison-Wesley*
- **Knuth D.E.,** The art of computer programming, vol. 2 : Seminumerical algorithms, *Addison-Wesley*
- **Knuth D.E.,** The art of computer programming, vol. 3 : Sorting and searching, *Addison-Wesley*
- **Knuth D.E.,** Literate Programming. *CSLI Lecture Notes No 27, 1992*
- **Krob D.,** Algorithmique et structures de données, Programmation, *Ellipses 1989*
- **Lipschutz S.,** Data Structures, *McGraw-Hill 1986*
- **Sedgewick** Algorithms *Addison-Wesley 1983*
- **Wirth N.,** Algorithms & Data Structures, *Prentice-Hall 1986*

13.3 Object Orientation

- **Aubert J.-P. and Dixneuf P.,** Conception et programmation par objet, *Masson 1991*

- **Blaschek G., Pomberger G. and Strizinger A.,** A comparison of object-oriented programming languages, *Structured programming* **4**, *187-198 (1989)*

- **Booch G.** Object-oriented Analysis and Design with applications, *Addison-Wesley 1994*

- **Quément B.,** Conception objet des structures de données, *Masson 1992*

- **Reiser M.** The Oberon System. *Addison-Wesley 1991*

- **Reiser M. and Wirth N.** Programming in Oberon, Steps beyond Pascal and Modula. *Addison-Wesley, ACM Press 1992*

- **Rubin K.S. and Goldberg A.** Object behavior analysis, *CACM* **9** *(1992)*

- **Voss G.,** Object-oriented programming, *McGraw-Hill 1991*

13.4 Concurrent and Real-Time Programming

- **Levi S.-T. and Agrawala A.K.,** Real-Time system design, *McGraw-Hill 1990*

- **Nichols B., Buttlar D. and Proux Farrel J.,** Pthreads programming, *O'Reilly & Associates 1996*

- **Nussbaumer H.,** Informatique industrielle, vol.2: Introduction à l'informatique du temps réel, *Presses Polytechniques Romandes 1986*

- **Schiper A.,** Programmation concurrente, *Presses Polytechniques Romandes 1986*

13.5 Languages

- **Darnell P.A. and Margolis P.E.** C, A Software Engineering Approach. *Springer-Verlag 1991*
- **Eckel Bruce** Thinking in Java. *Prentice-Hall 1998*
- **Flanagan David** Java in a Nutshell. *O'Reilly & Associates*
- **Hanly J.R. and Koffman E.B.** Problem Solving and Program Design in C. *Addison-Wesley 1996*
- **Hunt John** Java and Object Orientation, An Introduction. *Springer 1999*
- **King K.N.**, Modula-2, *D.C. Heath and Company 1988*
- **Lea Doug** Concurrent Programming in Java, Design Principles and Patterns. *Addison-Wesley 1997*
- **Lemay L. and Casdenhead R.** SAMS Teach Yourself JAVA 2. *Sams 1999*
- **Oualline Steve** Practical C textslO'Reilly & Associates 1993
- **Lippman S.B.**, C++ Primer, *Addison-Wesley 1989*
- Borland C++ Documentation, *Borland International 1989*
- **Thorin M.** Ada, Manuel complet du langage avec exemples, *Eyrolles 1981*
- **Winston P. H. and Narasimhan S.** On to JAVA. *Addison-Wesley 1996*

13.6 UNIX Tools

- **Bolinger D. and Bronson T.** Applying RCS and SCCS. *O'Reilly & Associates 1995*
- **DuBois Paul** Type Less, Accomplish More Using csh & tcsh. *O'Reilly & Associates 1995*
- **Garfinkel S.** PGP, Pretty Good Privacy, *O'Reilly & Associates 1995*

- **Kernighan B.W. and Plauger P.J.** Software Tools. *Addison-Wesley 1976*

- **Miller W.** A Software Tools Sampler. *Prentice-Hall 1987*

- **Quigley E.** UNIX Shells by Examples (Csh, sh, ksh, awk, grep and sed). *Prentice-Hall 1999*

- **Rosenblatt Bill** Korn Shell. *O'Reilly & Associates 1993*

- **Wall L., Christiansen, T, Schwartz, R. L.** Programming Perl *O'Reilly & Associates 1996*

- **Welch B. B.** Practical Programming in Tcl and Tk. *Prentice-Hall 1997*

13.7 RELATIONAL DATABASE

- **Codd E. F.** A Relational Model of Data for large Shared Data Banks. *CACM,* **13, No 6, 377-387, June 1970**

- **Codd E. F.** Relational Database: A Practical Foundation for Productivity. *CACM,* **25, No 2, 109-117, February 1982**

- **Manis R., Schaffer E., Jørgensen** UNIX Relational Database Management, Application Development in the UNIX Environment. *Prentice-Hall 1988*

- **Parsaye K., Chignell, M., Khoshafian, S., Wong, H.** Intelligent Databases. *Wiley 1989*

- **Stonebraker M.** The INGRES Papers: Anatomy of a Relational Database System. *Addison-Wesley 1985*

X Window Programming

Sixth College on Microprocessor-based Real-Time Systems in Physics

Abdus Salam ICTP, Trieste, October 9 — November 3, 2000

Ulrich Raich
CERN — European Organisation for Nuclear Reseach
P.S. Division
CH–1211 Geneva
Switzerland

email: Ulrich.Raich@cern.ch

Abstract

These lecture notes are intendend to give an insight into **G**raphical **U**ser **I**nterface (**GUI**) Programming using **X–Window** system. It explains the different layers of **X11** and gives a short introduction to **Xlib**.

Motif, the widget set supplied by the **O**pen **S**oftware **F**oundation (**OSF**) is used to demonstrate building of more sophisticated **GUI**s. Even though only very few routines of the **Motif** libraries are explained these notes are sufficient to build a **Motif** program driving the *ICTP Colombo Board* which is proposed as an exercise.

1 Introduction to X-Windows

X started its life in 1984 at the Massachusetts Institute of Technology (MIT) with the project **Athena**. At MIT several hundreds of workstations were scattered on the campus. They were intended for the use by students and were rather heterogenous (several different manufacturers, different operating systems). On the other hand most of them had:

- a powerful 32 bit CPU
- large address space
- a high resolution bitmapped display
- a mouse on some of them
- a network connection.

The idea was therefore to provide a window system allowing to write vendor independent applications, that could run on any of these stations. In addition, it should be possible to access applications on any of the workstations from any other workstation using the network. Of course performance was another keyword in the design.

Since the lecture time for **X–Windows** programming is fairly limited (there are some 10 books of 700 pages each explaining the **X–Windows** system!) we prepared this little booklet, which contains both an explanation of some basic features of the system and all the calls you will need for the exercises as well. In the course of the lectures and exercises you will build a little **X–Windows** application simulating the Colombo board on the screen and interact with it.

1.1 Client–Server Model

To write device-independent applications, the details of device access must be hidden in some sort of driver. In **X** this is a program called the *X–Server*. It provides all the basic windowing mechanisms by handling connections from **X**–applications, demultiplexing graphics requests and multiplexing input from keyboard, mouse, *etc* back to the application. This program is usually provided by the hardware vendor.

An application connects to the **X–Server** through an interprocess communication (**IPC**) path either through shared memory or through a network protocol like **TCP**. Such an **IPC** path is called the *X–Client*. Since most applications open only a single connection we often call the application itself the X–Client. However: an application having several **IPC** paths open is considered as several clients. The communication protocol between an *X–Client* and an *X–Server* is called *X–Protocol*.

One of the main design objectives of this protocol was to minimize the network traffic, because the network must be considered the slowest system component. Therefore an asynchronous protocol has been chosen. To bring windows up on the screen the application simply sends off requests without waiting for an acknowledgement. This can be done because of the reliability of the underlying network protocol. The application also does not poll for events like key presses and mouse button presses. It registers interest in certain events with the *X-Server*, which will then send only relevant events back to the application. Both the output requests and input events are buffered.

The *X-Protocol* is the fundamental layer onto which other tools can be built. It is user interface policy free. This means that windows are only represented as simple rectangles on the screen without any semantics. Buttons, pulldown-menus scrollbars don't exist in this layer. These so-called widgets are implemented in a toolkit sitting on top of the protocol.

The *XLib* contains routines that allow access to the *X-Protocol.* It provides the following functionality:

- **display management** (open, close displays)
- **window management** (create/destroy windows and change their visual aspect)
- **two dimensional graphics** (draw lines, circles, rectangles, text)
- **colour management** (colour map and its access routines)
- **event management** (registration of interest in events and event reception)

This gives us an overview over the next few chapters.

In Figure 1 we can see three **X**–Clients running on three different machines (**A**,**B**,**C**) and communicating with a single **X**–Server (on system **C**). For the clients on **A** and **B** the **X**–Protocol runs over the network, while for the text editor a shared memory **IPC** path is used.

1.2 Display Management

In order to create windows on the screen and to receive events a connection to the *display* must be established. In **X** terminology the display consists of:

- one or more screens,
- a single keyboard,
- an (optional) pointing device,
- the **X**–server

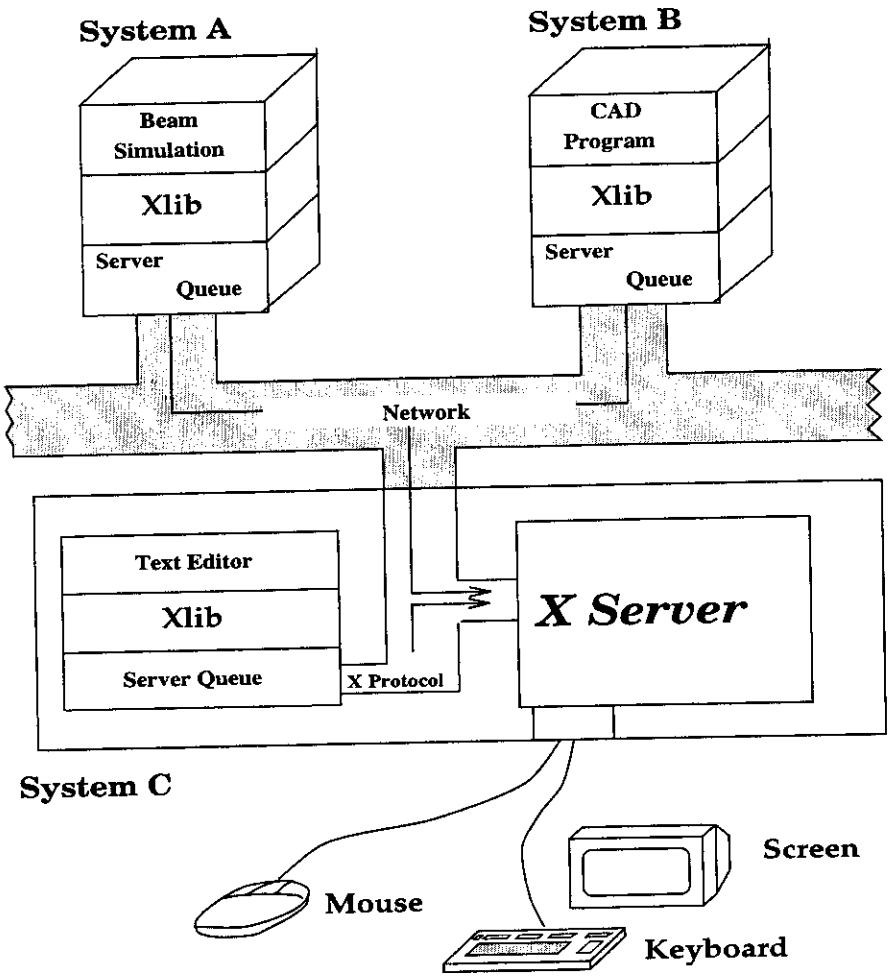


Figure 1: **Client–Server Model**

This connection can be built through the **XLib** call:

```
Display*XOpenDisplay(char *display_name)
```

If `display_name==NULL` the `display_name` defaults to the value stored in the *environment variable* **DISPLAY**. If you want to open the server on your neighbour's workstation, he will first have to allow you access to it:

`xhost` *name_of_your_station*,

then you may define *display_name* as

his_station:server_number.screen_number

usually *server_number.screen_number* is **0.0**.

The return value from this call must be saved, because it will be passed into all subsequent calls. In case of an error a NULL display is returned.

There are several Macros giving information about screen properties. Here are two of them:

- **int DisplayHeight** (Display *display, int screen_number);
- **int DisplayWidth** (Display *display, int screen_number);

giving the width and the height (in pixels) of the screen.

1.3 Windows Hierarchies

Once the Client–Server connection is established, we can generate our first windows:

```
Window XCreateSimpleWindow (
                Display *display,
                Window  parent,
                int     x,y,
                int     width, height, border_width,
                int     x,y, /*  position with  respect  */
                             /* to the upper left corner */
                             /* of  the  parent  window  */
                unsigned int  width,height,border_width,
                unsigned long border_color,background_color)
```

`x, y, width, height` and `border_width` do not need any explanation, `background_color` specifies the background colour. Because colours will only be explained later we will put this to:

`unsigned long` **WhitePixel** `(Display *display,int screen)`

where `WhitePixel` is a Macro returning the pixel value for *white*. X11 is able to support several screens with a single server. The default screen (we only have a single screen which of course is the default one to be used) can be found with

`int` **DefaultScreen** `(Display *display)`

The `border_color` parameter specifies the colour for the window border and is set to:

`unsigned long` **BlackPixel** `(display, int screen);`

The parent parameter will need some more explanation: All windows are inserted into a window hierarchy, where each window has a parent window. The great-grandfather of all windows is the root window who's id can be obtained by:

`Window` **RootWindow** `(Display *display, int screen)`

The root window

- covers the screen completely;
- cannot be moved or resized;
- is the parent of all other windows;
- has all window attributes like background colour, patterns, etc.

You can draw onto the root window as on to all other windows.

At this point an example will probably help more and further explanations. Here is a piece of code demonstrating the above calls and Macros which will create a single window. Please take note that this program is incomplete and will **not** bring the window up onto the screen yet.

```
/*************************************************************/
/* EXAMPLE 1 for XLIB                                        */
/*                                                           */
/* How to open a connection to a display, and create a window.*/
/*                  U. Raich 26-Nov-2000                     */
/*************************************************************/

#include <stdio.h>
#include <X11/Xlib.h>

main(argc,argv) unsigned int argc; char *argv[]; {

Display    *display;
Window     main_window;
int        screen;
int        x,y,width,height,border_width;
unsigned long    background_color,border_color;

/*
      Open the connection to the X-server
      The Null server name defaults to the display name
      defined in the environment variable DISPLAY, which
```

```
      usually is setup to the station running the client
      (client and server on the same machine).
*/

   display = XOpenDisplay(NULL);
   if (display == NULL) {
     fprintf(stderr,"Sorry, I could not open the Display.\n");
     exit(1);
   };

/*
   get color pixel values (see chapter on colors)
   for foreground and background namely black & white
*/
   border_color    =BlackPixel(display,DefaultScreen(display));
   background_color=WhitePixel(display,DefaultScreen(display));
/*
   define position, width and height of the window
*/
   x=50; y=50; width=200; height=100;  /* all this in pixels */
   border_width = 1;
/*
   get the screen number of the default screen (normally zero)
*/
   screen = DefaultScreen(display);
/*
   Create a window (the window does not appear on the screen yet)
   The 'rootwindow' is the parent of all other windows and covers
   the entire screen.
*/
   main_window = XCreateSimpleWindow(
                                  display,
                                  RootWindow(display,screen),
                                  x,y,width,height,border_width,
                                  border_color,
                                  background_color);
}
```

The return value from **XCreateSimpleWindow** is used to build the window hierarchy. In Figure 2 a complete window hierarchy is shown. Since all windows are clipped to the boundaries of their parents some of the windows may be completely invisible.

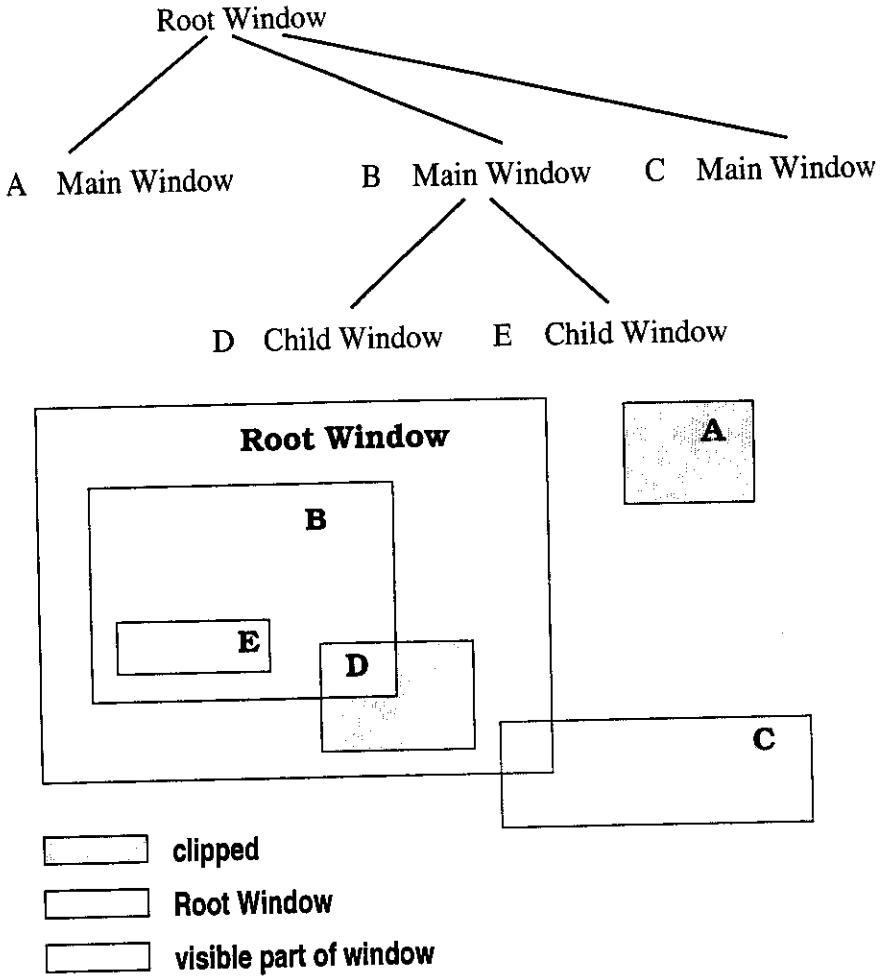


Figure 2: **Window Hierarchies**

After the **XCreateSimpleWindow** all the data structures needed for the management of the window will be created; however the window will still not be visible.

XMapWindow`(Display *display, Window window_id)`
will map the window and all of its subwindows, for which the **XMapWindow** routine has been called. Here is the code that is missing in the above program in order to make it work and bring the window up on the screen.

```
/*
   here is, what was missing in the first program:
*/
/*
  in the declaration part:
*/
XEvent  event;
/*
   at the end of the code:
*/
   XMapWindow(display,main_window);
/*
   don't worry about this part of the code, we will see this in
   quite some detail later
*/
   for(;;)
     XNextEvent(display,&event);
```

As you have hopefully learned in the meantime, Makefiles are a *must* for every serious Unix programmer, therefore we also give the Makefile written for the above program. Notice how simple such a Makefile can be!

```
# Makefile for a simple X-11 program
# U. Raich 26-Nov-2000

CFLAGS = -g -I.
LDFLAGS = -L/usr/X11R6/lib -lX11
CC=gcc

all: example_1 example_1_complete
```

Once the window is mapped, there are several **XLib** calls to change its layout:

- **XMoveWindow** `(Display *display, Window window_id, int x_offset, int y_offset)`
- **XResizeWindow** `(Display *display, Window window_id, int width, int height)`
- **XMoveResizeWindow** `(Display *display, Window window_id, int x_offset, int y_offset, int width, int height)`

- **XSetWindowBorderWidth** (Display *display,
  Window window_id,
  int border_width)
- **XSetWindowBackground** (Display *display,
  Window window_id,
  unsigned long background_pixel)
- **XChangeWindowAttributes** (Display *display,
  Window window_id,
  unsigned long value_mask,
  XSetWindowAttributes
  *attributes)

and many more.

The last call allows to change any of the window **attributes** in a single call. **Attributes** is a **XSetWindowAttributes** structure, having a certain number of fields. The value_mask tells the system, which of the attribute fields are to be taken into account. Only these values will be changed. It is a bitwise inclusive *OR* of the valid attribute mask bits (see Table 1).

Using this mechanism windows may also be created with:

```
Window XCreateWindow (Display *display,
                      Window parent,
                      int x, int y,
                      unsigned int width, unsigned int height,
                      unsigned int border_width,
                      unsigned long valuemask, int depth,
                      unsigned int class, Visual *visual,
                      XSetWindowAttributes *attributes)
```

For depth, class and visual try to specify *CopyFromParent* and have a look at the man page. If in the situation of Figure 2 we would call

XUnmapWindow (display, B_main_window)

the window **B** and all of its subwindows (**D** and **E**) would disappear. Figure 3 shows the results of such a Map call for a single main window.

It has been explained before, that a window simply consists of a rectangle. Here on the contrary many more items like the three buttons on top of the window, the stars, the triangles on each corner, *etc.* can be seen. The layout and the functionality depend on the look and feel (the policy) of the window system. It is another *X-Client*, the window manager, which is responsible

Mask		Structure
0	$\longmapsto$	anything
1	$\longmapsto$	valid
1		valid
0		anything
0	$\longmapsto$	anything
1		valid

Table 1: **Structure and Value Mask**

for the decoration of the main window (child of root window). It allows to change the stacking order of windows, displace and resize windows, iconize them and even killing them (and the application).

The **XLib** provides calls to communicate easily with the window manager to modify the decorations like this one:

```
XStoreName (Display *display,
            Window window_id,
            char   *title_bar_text)
```

This call communicates the title to be put into the title bar by the window manager. The communication is done through the *Inter Client Communication Convention* (**ICCCM**, here M means Manual) using so called *window properties*, which are data that can be attached to a window. We will not be able to go into any detail for lack of time.

1.4 Drawing, the Graphics Context

Let us have a look at the simplest possible drawing instruction: create a line between two points. The line may be done with the call

```
XDrawLine ( Display *display, Drawable drawable,
            GC      graphics_context,
            int     x_start_point,int y_start_point,
            int     x_end_point,  int y_end_point)
```

The meaning of all parameters except *drawable* and *graphics_context* should be obvious. The *drawable* tells the system where to draw. In fact, there are two possibilities. Either we draw into a window on the screen or into

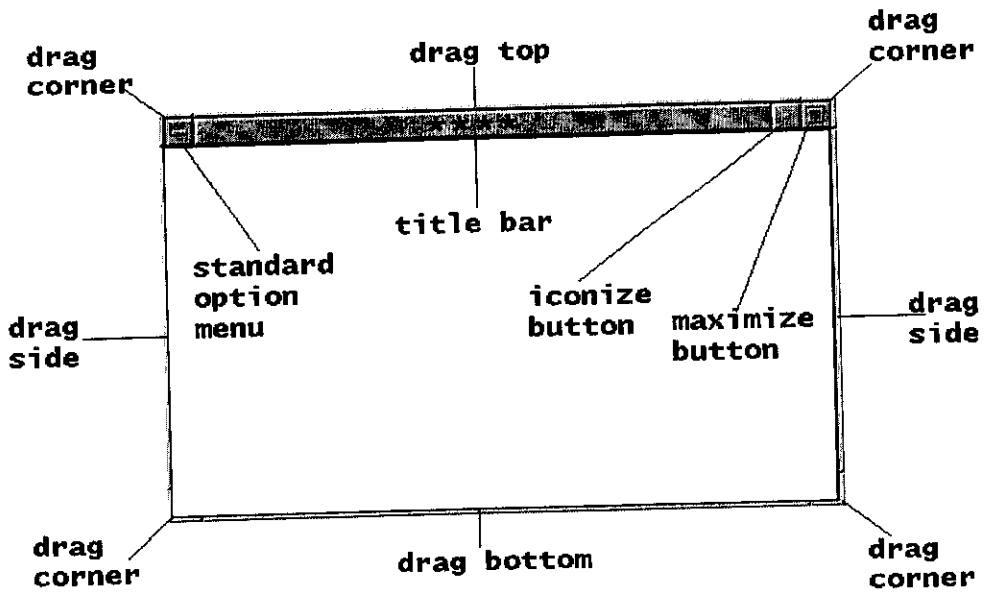


Figure 3: **A Main Window**

a window simulated in memory, called a **pixmap** (Details about pixmaps are found in section 1.5). Remember the *root window* is treated like any other window, thus it is possible to generate background pictures by drawing in the root window.

Coming back to our draw line primitive: when drawing the line several questions remain open:

- what is the line width?
- what colour?
- straight line or dashed, dotted, ..., what are the distances between dashes?
- how to join lines?

and there are many more *drawing attributes.*

Since it is the *X–Client* who generates these graphic requests and it is the *X–Server* who executes them, all attributes must be sent to the server. This could be done on a per primitive basis, however network traffic would be strongly increased and the performance would suffer. For this reason graphics contexts containing all these attributes can be prepared on the server. In the drawing call the identifier of the graphics context resident on the server is specified.

The Call **XCreateGC** creates a graphics context.

```
GC XCreateGC (Display *display,
              Drawable drawable,
              int value_mask, XGCValues *values);
```

The value structure of type **XGCValues** has over 20 entries. In table 2 some of the entries and their corresponding **value_mask** bit names are given. For instance, to create a graphics context that allows drawing of a dashed line with width 4, it is possible to use the following code segment:

```
XGCValues       values;
unsigned long   value_mask;
GC              graphics_context;

/* setup the value mask */
value_mask          = GCLineStyle | GCLineWidth;
/* define the field indicated in mask */
values.line_style   = LineOnOffDash;
values.line_width   = 4; /* width of the line */
graphics_context    = XCreateGC
                      (display, main_window, value_mask, values);
```

All other GC values will be defaulted. Another way is to generate a default **GC** using the call

DefaultGC`(display, screen,_number);`

and then use

XChangeGC `(display, graphics_context, value, value_mask);`

to do the necessary changes.

There are also lots of convenience functions changing a single entry in the value structure:

- **XSetForeground** `(Display *display,`
  `GC graphics_context,`
  `unsigned long foreground);`
- **XSetBackground** `(Display *display,`
  `GC graphics_context,`
  `unsigned long background);`
- **XSetLineAttributes** `(Display *display,`
  `GC graphics_context,`

Entry	value_mask bit	usage and possible values	
values.line_width	GCLineWidth	Type of line to draw	
values.line_style	GCLineStyle	LineSolid	draw full line
		LineDoubleDash	odd lines full differently from even lines
		LineOnOffDash	only even dashes are drawn
values.cap_style	GCCapStyle	*How to draw the end point:*	
		CapButt	line square at the end point
		CapNotLast	as CapButt, but the last point is not drawn
		CapProjecting	as CapButt, but the line is longer half the projection
		CapRound	round and points
values.join_style	GCJoinStyle	*How to join fat lines:*	
		JoinMiter	outer edges extend to meet at an angle
		JoinBevel	corner is cutt off
		JoinRound	round off edges
values.fill_style	GCFillStyle	FillSolid	uses foreground colour for filling
		FillTiled	uses a loloured tile pattern
		FillStippled	same as FillSolid but uses a stipple pattern bitmap as the mask
		FillOpaqueStippled	same as FillTiled but uses a stipple pattern as the mask in addition
values.function	GCFunction	*logical operation for drawing possible values see later*	
values.foreground	GCForeground	*foreground colour*	
values.background	GCBackground	*guess what!*	
values.tile	GCTile	*tile pixmap*	
values.stipple	GCStiple	*stipple bitmap*	
values.clip_mask	GCClipMask	*clip mask*	
values.ts_x_origin	GCTileStipXOrigin	*shifting the tile of stipple pattern origins*	
values.ts_y_origin	GCTileStipYOrigin	*the same for y*	
values.font	GCFont	*font for text drawing*	

Table 2: Some of the X GC Values

```
                    unsigned int line_width,
                    int line_style, int cap_style,
                    int join_style);
```

- **XSetDashes** (Display *display,
  GC graphics_context,
  dash_offset,
  char dash_list[], int n);
- **XSetFillStyle** (Display *display,
  GC graphics_context,
  int fill_style);
- **XSetTile** (Display *display,
  GC graphics_context,
  Pixmap tile);
- **XSetStipple** (Display *display,
  GC graphics_context,
  Pixmap stipple);
- **XSetClipMask** (Display *display,
  graphics_context, Pixmap clip_mask);
- **XSetFont** (Display *display, GC graphics_context, Font font);

and many more. Figure 4 shows the result of these graphics context manipulations.

Last but not least there is an entry **values.function** which sets the binary function that is applied to the existing pixel value when drawing on the screen (src is the pixels to be drawn newly, dest is the actual pixel value)

• **GXClear**	0
• **GXand**	src and dest
• **GXandReverse**	src and (not dest)
• **GXcopy**	src (this is the default of course!)
• **GXnoop**	dest
• **GXxor**	src xor dest
• **GXnor**	(not src) and (not dest)
• **GXequiv**	(not src) xor dst
• **GXinvert**	not dst
• **XorReverse**	src or (not dest)
• **GXcopyInverted**	not src

- **GXorInverted** (not src) or dst
- **GXnand** (not src) or (not dest)
- **GXset** 1

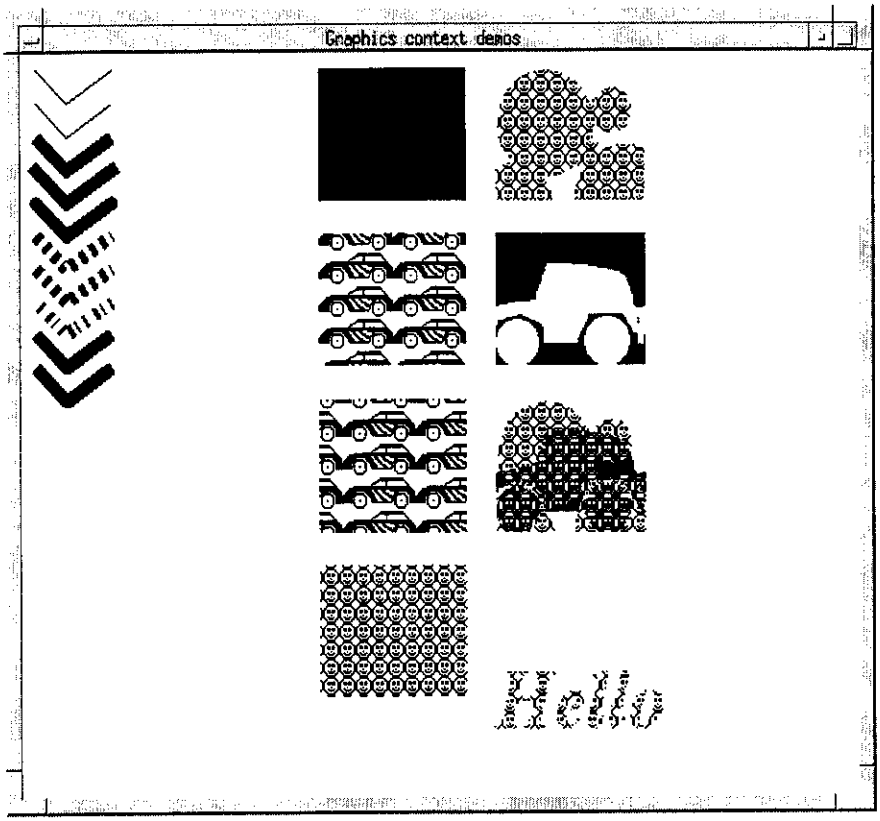


Figure 4: **The Graphics Context**

1.5 Bitmaps and Pixmaps

In the previous section we were talking about pixmaps as drawables for drawing primitives. Therefore the questions:

- What exactly is a pixmap?
- What is the difference between a bitmap and a pixmap?
- How can we generate pixmaps?

As already explained before, a pixmap is a sort of a simulated window in memory. As long as we work on a black and white workstation we need

1 bit for each pixel to be displayed. An array, describing such a pixelplane is called a *bitmap*. Once we use a color display several bitplanes are needed depending on the number of colors, that can be displayed. This collection of bitmaps with a certain *depth* is called a *pixmap*.

An empty pixmap can be allocated with the call:

```
Pixmap XCreatePixmap (Display *display,
Drawable drawable, int width, int height, int depth)
```

The pixmap will be stored on the *X–Server*, which is the reason for the drawable parameter. Just specify the id of your main window. Once you allocate the pixmap you can use it as drawable in any of the drawing primitives. In order to visualize your pixmap you must copy its contents onto a window:

```
XCopyArea (Display *display,
           Drawable source_drawable, Drawable dest_drawable,
           GC gc, int src_x, int src_y,
           unsigned int copy_width, unsigned int copy_height,
           int dest_x, int dest_y);
```

If you have a bitmap which you want to convert to a pixmap or simply visualize on a color display you use:

```
XCopyPlane (Display *display,
            Drawable source_drawable, Drawable dest_drawable,
            GC gc, int src_x, int src_y,
            unsigned int copy_width, unsigned int copy_height,
            int dest_x, int dest_y, unsigned long plane);
```

with plane =1 (bitmap).

Of course it might be rather difficult to build up bitmaps using only drawing primitives. For this reason **X** provides a utility, the bitmap editor.

The command **bitmap** will bring up the application shown in fig. 5.

The result of the editor is a **C** source file which can be included into you application:

```
#define smiley_width 11
#define smiley_height 11
static char smiley_bits[] = {
   0xf8, 0x00, 0x04, 0x01, 0x02, 0x02,
   0xd9, 0x04, 0xd9, 0x04, 0x01, 0x04,
   0x21, 0x04, 0x89, 0x04, 0x72, 0x02,
   0x04, 0x01, 0xf8, 0x00};
```

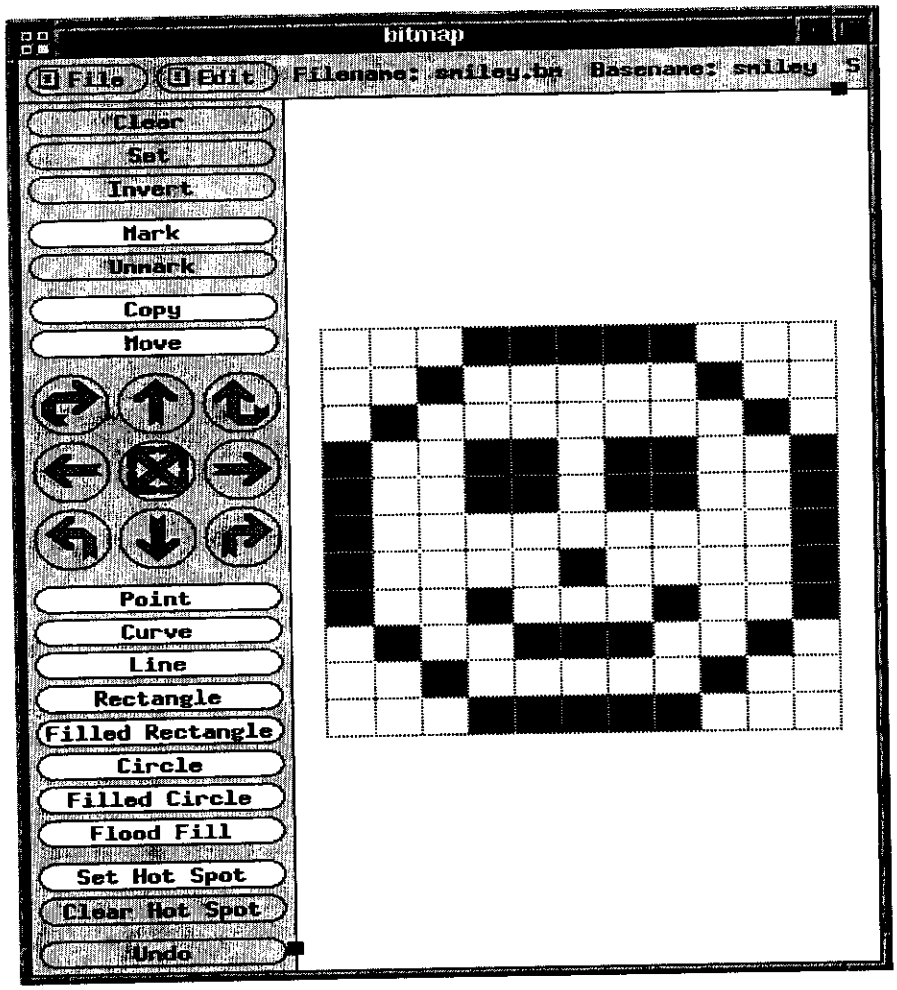


Figure 5: **The Bitmap Editor (bitmap)**

This code can be used to create a pixmap:

```
Pixmap XCreatePixmapFromBitmapData (
        Display *display, Drawable drawable, char *smiley_bits,
        unsigned int smiley_width, unsigned int smiley_height,
        unsigned long foreground, unsigned long background,
        unsigned int depth);
```

In order to find the number of bitplanes used by the *visual* (our display hardware) the macro

DefaultDepth `(Display *display, int screen)`

can be used. The same result can be obtained by reading in the bitmap file directly.

```
int     XReadBitmapFile(Display *display, Drawable drawable,
                        char *bitmap_file_name,
                        unsigned int *width, unsigned int *height,
                        Pixmap *bitmap, int *hot_x, int *hot_y);
```

`hot_x, hot_y` give the coordinates of the *hot spot* used for cursors. Now the bitmap can be converted to a pixmap with the **XCopyPlane** call.

There is also a freely distributable library and a pixmap editor which can be used to generate pixmaps (in colour) directly. (Try **pixmap** on your machine!)

Pixmaps are used for cursors, tiles, stipples, icons etc. They can also be used to restore pictures which have been destroyed by overlapping windows (see section on events).

1.6 Drawing Primitives

X–Window is *NOT* a graphics system! This can be easily seen by the limited number of graphics primitives and by their simplicity:

There are a few functions to clear out an area to be drawn in

- **XClearArea** `(Display *display, Window window_id,`
  `int x, int y,`
  `unsigned int width, unsigned int height,`
  `Bool exposures);`
- **XClearWindow**`(Display *display, Window window_id);`
- **XFillRectangle**`(Display *display, Drawable drawable,`
  `GC graphics_context, int x, int y,`
  `unsigned int width, unsigned int height);`

While most graphics primitives work on a drawable, **XClearWindow** and **XClearArea** work only on windows.
Here are the primitives which actually draw graphic objects:

- **XDrawPoint** `(Display *display, Drawable drawable,`
  `int x, int y);`

- **XDrawPoints**`(Display *display, Drawable drawable,`
  `Point *points, int n_points, int mode);`

where **points** is an array of type

```
typedef struct {
                short x, y;
                } XPoint;
```

n_points means the number of **XPoint** entries in the array *points*, and
mode = **CoordModeOrigin** (x,y is given in absolute pixel coordinates);
or
mode = **CoordModePrevious** (x,y are the relative distances to the last point)

- **XDrawLine** (Display *display, Drawable drawable,
  GC gc, int x1, int y1, int x2, int y2);
- **XDrawLines** (Display *display, Drawable drawable,
  GC gc, Point *points, int npoints, int mode);
- **XDrawRectangle** (Display *display, Drawable drawable,
  GC gc, int x, int y,
  unsigned int width, unsigned int height);
- **XDrawRectangles** (Display *display, Drawable drawable,
  GC gc, Rectangle *rectangles,
  int nrectangles);

where **rectangles** is an array of type

```
typedef struct {
        short           x, y;
        unsigned short  width, height;
        }XRectangle;
```

- **XFillRectangle** (Display *display, Drawable drawable,
  GC gc, int x, int y, unsigned int width,
  unsigned int height);
- **XFillRectangles** (Display *display, Drawable drawable,
  GC gc, Rectangle *rectangles,
  int nrectangles);

and there are some more for drawing arcs, segments,*etc.*

With all this theory let's have a look at a very simple example again: The following program will display a set of 50 random lines in our previously mapped window. The call to XSelectInput and the fact that the drawing is down after the XNextEvent will again be explained later (section on events).

```
#include <stdlib.h> /* needed for the random number generator */
#define  WINDOW_WIDTH  200
#define  WINDOW_HEIGHT 100
```

```
/*
  here we do some drawing, just a series of random lines
*/

   GC  gc;
   int i,x1,x2,y1,y2;
/*
   new program ... new name!
*/
   XStoreName(display,main_window,"Uli's very first X-11 program");

/*
  first define the graphics context
*/
   gc = DefaultGC(display,screen);
/*
   Sorry, that one you will have to believe me for the moment
   again, we will see the call later
*/
   XSelectInput(display,main_window,ExposureMask);

/*
   then do the drawing
*/
   for(;;)
     {
       XNextEvent(display,&event);
       XClearWindow(display,main_window);

       for (i=0;i<50;i++)
       {
           x1 = rand()%WINDOW_WIDTH;
           y1 = rand()%WINDOW_HEIGHT;
           x2 = rand()%WINDOW_WIDTH;
           y2 = rand()%WINDOW_HEIGHT;
           XDrawLine(display,main_window,gc,x1,y1,x2,y2);
        }
     }
```

For text drawing lots of different fonts are available. The command **xlsfont** prints the names of all available fonts. If you want to know how the font looks like, try **xfontsel**. The font names are standardized as follows:

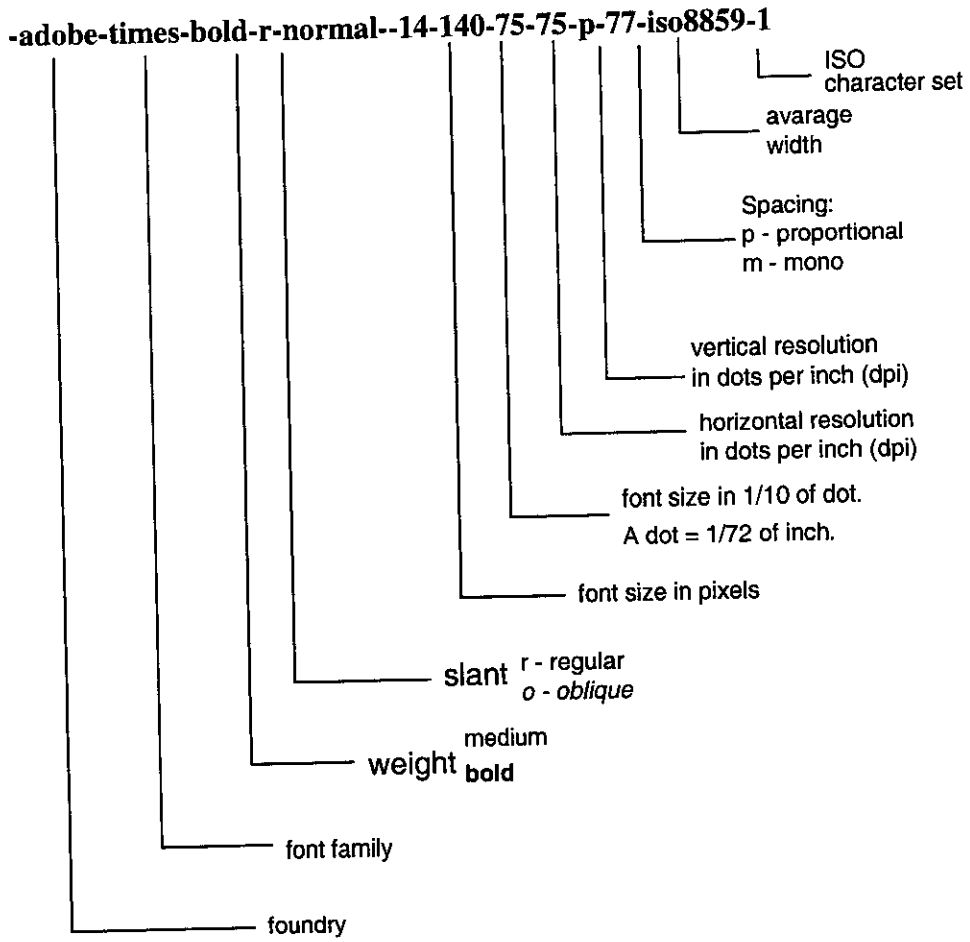


Figure 6: **Naming X fonts**

First the font should be loaded with

```
Font Font XLoadFont (Display *display,char *font_name);
```

then the font must be specified in the graphics context and last but not least we can draw our text using

```
XDrawString (Display *display, Drawable drawable, GC gc,
             int x, int y, char *string, int length);.
```

It is also possible to fill an array of text items:

```
typedef struct {
                char *chars;
                int  nchars;
                int  delta;/* distance between strings, is */
                           /* added  to  horizontal origin */
                Font font;
                }XTextItem
```

and use

```
XDrawText (Display *display, Drawable drawable, GC gc,
           int x, int y, XTextItem *item_array, int nitems);
```

which allows drawing of multiple font text strings.

1.7 Colour Model

The hardware for color displays varies very widely depending on the needed graphics performance of the system. Since X is supposed to be hardware independent, there must be a common color model that can be converted for the different devices. X knows of the following hardware types, referred to as **visuals**:

- **Pseudo colour**
  This used to be the most common type of device. The image is described through a pixel array, where each pixel is interpreted as an index into a colour table, containing the r,g,b values sent to the screen.
- **Static colour**
  Like pseudo colour, except that the values in the colour map are read only
- **Direct colour**
  The pixel is composed of **3** bit fields, each of which is used as an index into one of **3 R**, **G**, **B** colour maps. The values in the three colour maps can be changed
- **True colour**
  As the name proposes, here the color components for each pixel are

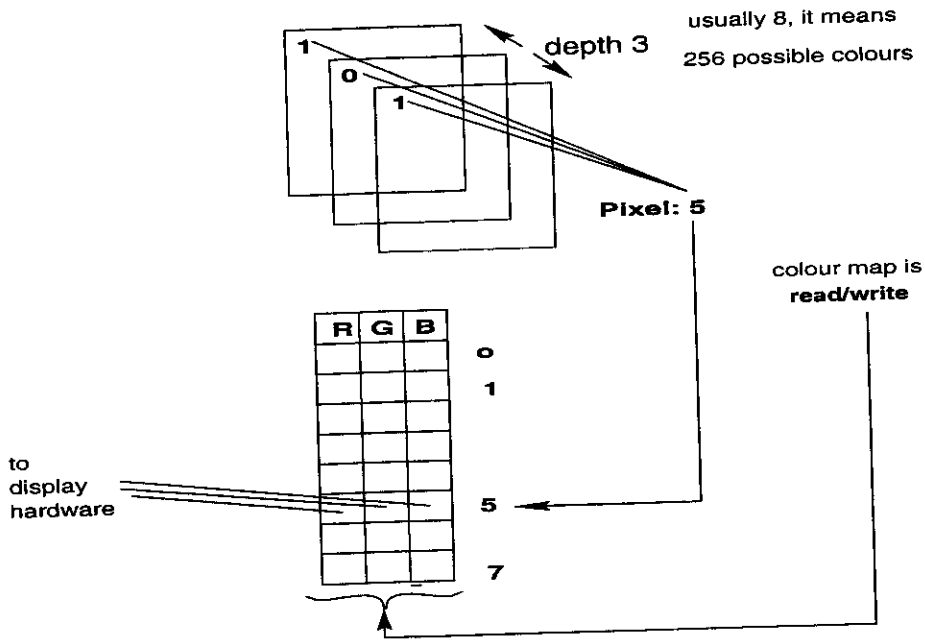


Figure 7: **A Pseudo Color Device**

directly specified. In this case we typically need 3 (components) times 8 bits of video memory for each pixel. For a 1024*1024 screen we therefore need minimum 3 MBytes of memory. While a few years ago, due to cost reasons these types of graphics cards were only used in very costly graphics workstations, it is the most common model nowadays. Since the colors are directly specified 24 bits of the pixel value (pixels are longs!) the indirection through a color map is not needed here.

- **Grey scale**
  As the pseudo colour. There are still three colour maps but only one of them is used

- **Static grey**
  Like grey scale, but colour map is readonly and linear

An **X** application can install its own colour map, but it will then disturb other applications running on the same screen, because their colours will be wrong. The colour maps are switched depending on input focus. So normally a single default colour map is used. We can get an identifier to it by

`Colormap` **DefaultColormap** `(Display display, int screen_number);`

This colourmap (when the visual depth is 8) usually contains **6*6*6** pre-programmed colors and **40** freely programmable colour cells.

To allocate colour cells in the colour map there are two possible methods.

1 allocating readonly colour cells:
These cells are shared between applications and are allocated by colour name

```
XAllocNamedColor (Display *display,
                  Colormap colormap,
                  /* usually DefaultColormap(display,screen)*/
                  char *color_name,
                  XColor *closest_color, XColor *exact_color);
```

2 allocating read/write colour cells (only for Pseudocolor visuals):

```
XAllocColorCells (Display *display,
                  Colormap colormap
                  contig_flag, &plane_masks,
                  n_planes, &pixels, n_pixels);
```

We can then use

```
XStoreColor (display,
             DefaultColormap(display, screen_number),
             my_colour)
```

to store a color into the default colourmap.

While **XAllocNamedColor** can be used for all color visuals XAllocColorCells and XStoreColor will only work on Pseudocolor devices.

The structure

```
typedef struct {
               unsigned long  pixel;
               unsigned short red, green, blue;
               char           flags; /*DoRed | DoGreen | DoBlue */
               char           pad;
               } XColor;
```

describes a colour. The **red**, **green**, **blue** values are always in the range of 0-65535 and are scaled to the number of bits actually in use by the display hardware. Flags allow to use only the **red**, **blue**, or **green** component (or any combination thereof).

Most graphics routines ask for a pixel value. When allocating read-only colour cells, a colour structure is returned (`closest_color`) and the pixel entry in this structure (`closest_color.pixel`) can be used. In **XAllocColorCells**, the pixel values are returned directly. On Truecolor devices we will define the colors directly within 24 bits of the pixel value.

Here is an example that shows a part of the **rgb** database file.

```
255 250 250             snow
248 248 255             ghost white
248 248 255             GhostWhite
245 245 245             white smoke
245 245 245             WhiteSmoke
220 220 220             gainsboro
255 250 240             floral white
255 250 240             FloralWhite
253 245 230             old lace
253 245 230             OldLace
250 240 230             linen
250 235 215             antique white
250 235 215             AntiqueWhite
255 239 213             papaya whip
255 239 213             PapayaWhip
255 235 205             blanched almond
255 235 205             BlanchedAlmond
255 228 196             bisque
255 218 185             peach puff
255 218 185             PeachPuff
255 222 173             navajo white
255 222 173             NavajoWhite
255 228 181             moccasin
255 248 220             cornsilk
255 255 240             ivory
255 250 205             lemon chiffon
255 250 205             LemonChiffon
255 245 238             seashell
240 255 240             honeydew
245 255 250             mint cream
245 255 250             MintCream
```

1.8 Event Handling

Once the client–server connection is opened the *X–Client* sends requests for bringing up windows, changing them, drawing things into them *etc*, but the *X–Server* can also inform the *X–Client* of certain events like exposure of a window, mapping of a window or user initiated, asynchronous events like mouse clicks or keyboard button presses. Due to the enormous amount of possible events (think of mouse movement only!) and the relatively small number of events the *X–Client* is actually interested in, it is much more efficient to filter the events on the server side. Before treating any events the *X–Client* must therefore register interest in a certain type of event on a per window basis.

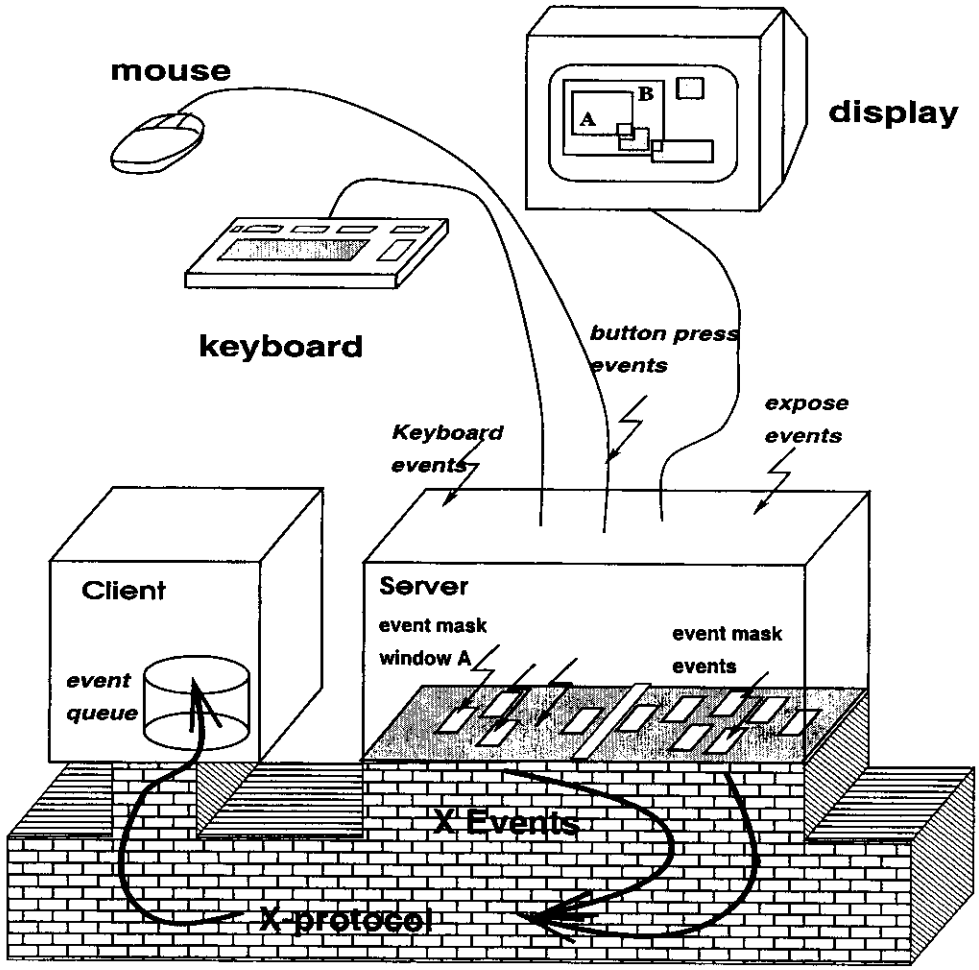
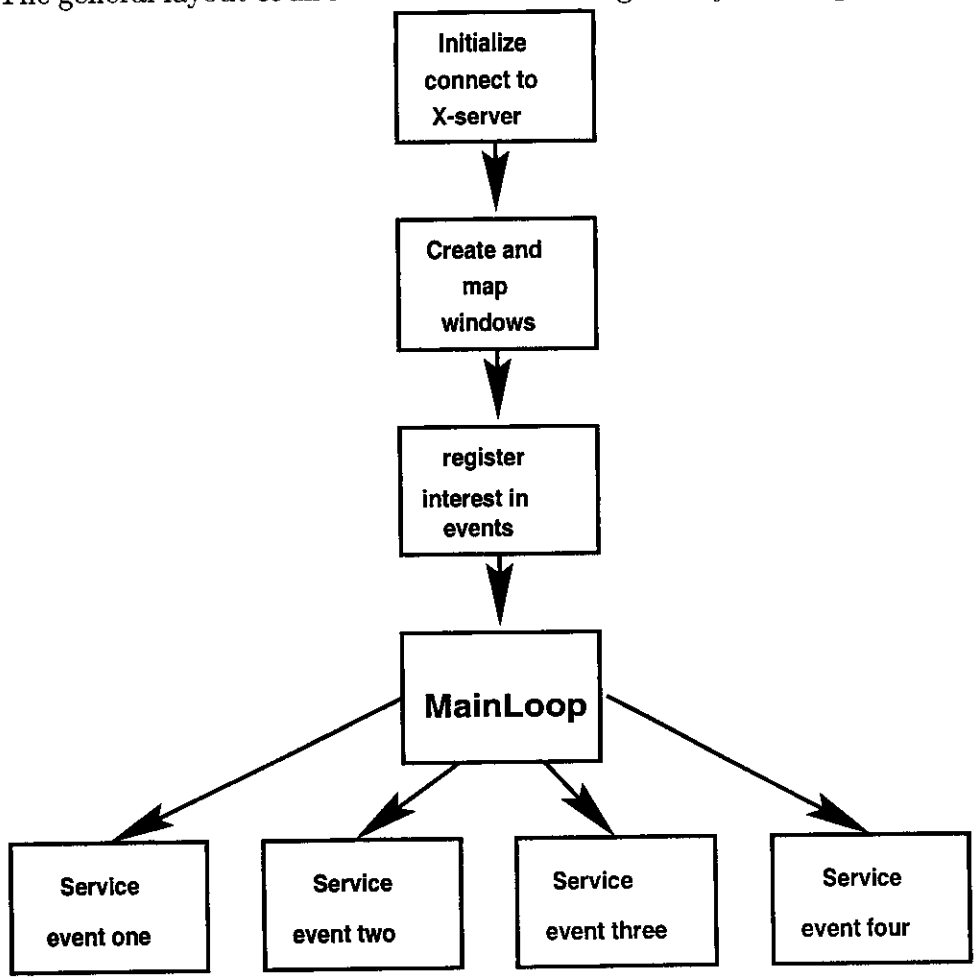


Figure 8: **Events in a Client-Server Model**

The general layout of an *X–Client* is therefore given by the diagram below:



While in the "usual" programming style the program asks for user input at the moment it is needed and convenient (the program controls the user!) in *X–Windows* programs the user can change the flow of control in any manner choosing functions provided by the program in a completely random manner.

There are two possible ways for the *X–Client* to register interest in events:

1 at the moment of window creation we can set the entry **event_mask** and the corresponding bit **CWEVENTMASK** in the value mask to the event types we are interested in;

2 **XSelectInput** (Display display,
`Window window_id, long event_mask);`

If one of the selected events arrives at the *X–Server* (say a mouse click) it sends this event into the *X–Clients* event buffer. There the main loop can pick it up and analyse it.

The call

XNextEvent (Display display, XEvent *event);

retrieves the next event from the event queue and blocks if no events are available.

The **XNextEvent** returns an *XEvent* structure of the following form:

```
typedef struct {
                int             type;
                unsigned long   serial;
                Bool            send_event;
                Display         *display;
                Window          window;
} XAnyEvent;

typedef union { int             type;
                XAnyEvent       xany;
                XButtonEvent    xbutton; ··· many more ···
                XExposeEvent    xexpose; ··· many more ···
                XKeyEvent       xkey;
                XMapEvent       xmap; ··· many more ···
} XEvent;
```

From the *event.type* we can find out which sort of event has happened. The following table gives a few examples. The event mask enabling reception of the event type and the symbol for the event type are given.

Event Mask	Event Type	Event Structure
KeyPressMask	KeyPress	XKeyPressedEvent
KeyReleaseMask	KeyRelease	XKeyReleasedEvent
ButtonPressMask	ButtonPress	XButtonPressedEvent
ButtonReleaseMask	ButtonRelease	XButtonReleasedEvent
PointerMotionMask	MotionNotify	XPointerMovedEvent
LeaveWindowMask	LeaveNotify	XLeaveWindowEvent
EnterWindowMask	EnterNotify	XEnterWindowEvent
ExposureMask	Expose	XExposeEvent

In the above example *Expose* events need to be enabled for each of the sub-windows and ButtonPress events must be enabled for the up/down arrows and the exit button.

When drawing into a window using the **X** drawing primitives, the window must already be mapped onto the screen and all window properties like position, size, id ... must be known. Since the visualization is done by the *X–Server* there is a problem of synchronization. Therefore we usually create and map the windows to be drawn in during the program initialization. Interest in *Expose* events are declared as well. As soon as the *X–Server* has mapped the window (all information on the window is available) an expose event is generated. The drawing then goes to the expose event handler. The **XExposeEvent** structure has the following form:

```
typedef struct {int             type;
                unsigned long   serial;
                Bool            send_event;
                Display         *display;
                Window          window;
                int             x, y;
                int             width, height;
                int             count;
                } XExposeEvent;
```

The **x**, **y**, **width**, **height** parameters in the structure describe a rectangle of pixels which must be redrawn. As can be seen in the Figure 9 below, redrawing of several rectangles may be needed. The count entry indicates how many more such expose events are going to follow. The easiest method to treat these events discards all expose events with nonzero count and redraws the full window on the last *(count=0)* expose event.

If the window size does not change, we can put our drawing into a pixmap of same size as the window and on expose events copy the pixmap onto the window using **XCopyArea.**

This diagram shows the rectangles to be updated if the calculator is brought into foreground

In order to demonstrate the above principles: color and event handling, the drawing example program has been re-written in such a way that firstly the random lines get colored and secondly the lines are drawn only once into a pixmap and then rendered on the window each time an expose event arrives. Note that in the original example the lines change each time we move the window or we cover/uncover it. In the new program this problem is absent. Up to the XSelectInput call the program remains the same except for inclusion of the math library needed, for the calculation of random colors, and the transfer of some local variables into global ones, needed for the subroutine that calculates the lines. Since this is the final version of our X program the full source code is given here. Please note that pressing a mouse

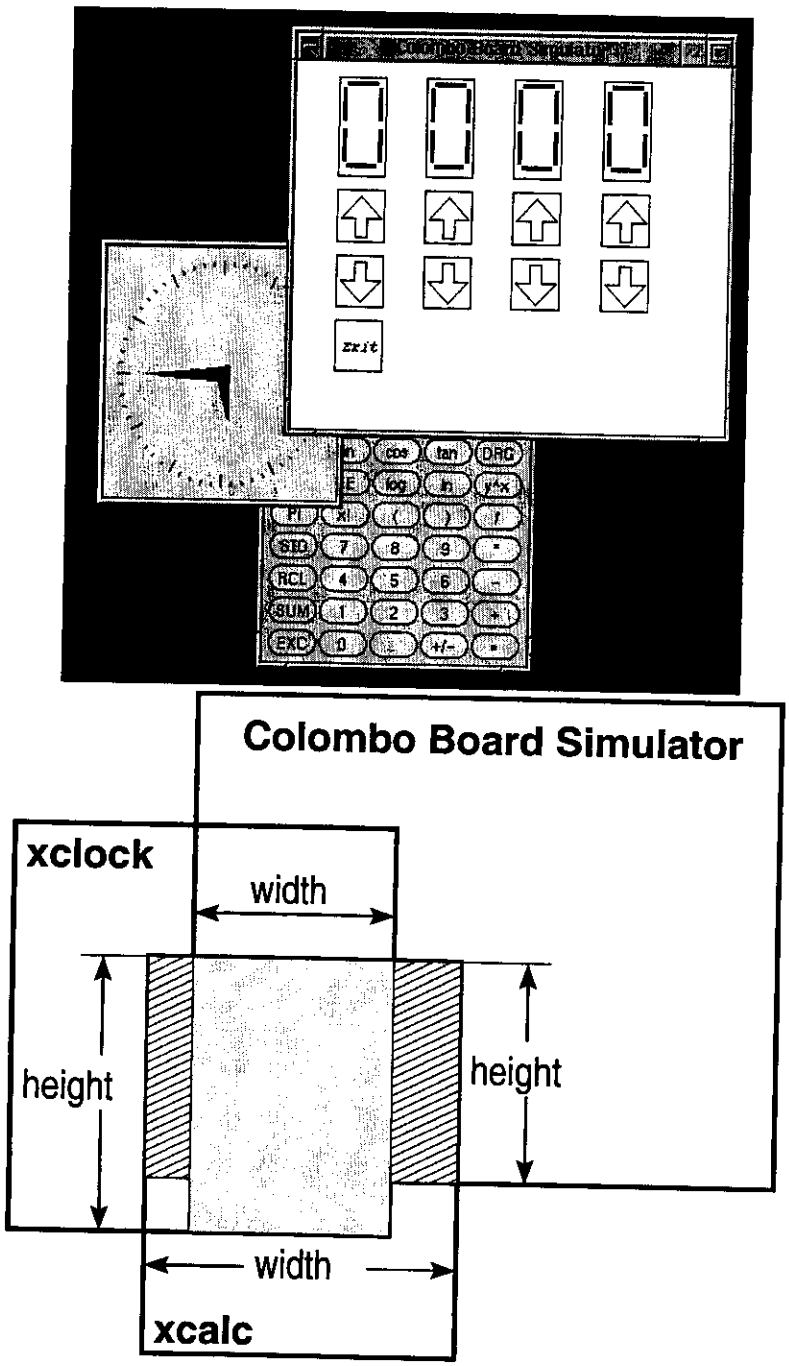


Figure 9: **Expose Events**

button on the window will erase the lines previously drawn and will calculate and display a new set of lines.

```
/*****************************************************************/
/* EXAMPLE 4 for XLIB                                            */
/* How to open a connection to a display, and create a window.   */
/*                                                               */
/*****************************************************************/

#include <stdio.h>
#include <X11/Xlib.h>
#include <stdlib.h>  /* needed for the random number generator */
#include <math.h>

#define  WINDOW_WIDTH  300
#define  WINDOW_HEIGHT 150

Display       *display;
int           screen;
Window        main_window;
Pixmap        backingStore;
int           depth;
int           x,y,width,height;
GC            gc;
void          drawRandomLines();

main(argc,argv) unsigned int argc; char *argv[]; {

unsigned long background_color,border_color,border_width;
XExposeEvent  event;

/*    Open the connection to the X-server
      The Null server name defaults to the display name defined
      in the environment variable DISPLAY, which usually is
      setup to the station running the client (client and server
      on the same machine).
*/

   display = XOpenDisplay(NULL);
   if (display == NULL) {
      fprintf(stderr,"Sorry, I could not open the Display.\n");
      exit(1);
   };
```

```
/*
   get color pixel values
   for foreground and background namely black & white
*/
   border_color     = BlackPixel(display,DefaultScreen(display));
   background_color = WhitePixel(display,DefaultScreen(display));
/*
   define position, width and height of the window
*/
   x=50; y=50;
   width=WINDOW_WIDTH;
   height=WINDOW_HEIGHT;                    /* all this in pixels */
   border_width = 1;
/*
   get the screen number of the default screen (normally zero)
*/
   screen = DefaultScreen(display);
/*
   Create a window (the window does not appear on the screen yet)
   The 'rootwindow' is the parent of all other windows and covers
   the entire screen.
*/
   main_window = XCreateSimpleWindow(
                                    display,
                                    RootWindow(display,screen),
                                    x,y,width,height,border_width,
                                    border_color,
                                    background_color);
/*
   new program ... new name!
*/
   XStoreName(display,main_window,"Demonstrating Expose Events");

   XMapWindow(display,main_window);

/*
  first define the graphics context
*/
   gc = DefaultGC(display,screen);
/*
```

```
   The XSelectInput declares the events we are interested in.
   ButtonPress events as well as Expose will be passed on to
   the client and end up in the event buffer from where
   they recuperated by this program using XNextEvent
*/
   XSelectInput(display,main_window,
                ExposureMask|ButtonPressMask);

   depth = DefaultDepth(display,screen);
   printf("You have a color device with %d color planes\n",
                                                    depth);

   backingStore = XCreatePixmap(display,
                                RootWindow(display,screen),
                                width,height,depth);

   drawRandomLines();
/*
   don't worry about this part of the code, we will see this in
   quite some detail later
*/
   for(;;)
     {
       XNextEvent(display,(XEvent *)&event);
       if (event.type == ButtonPress)
         {
    drawRandomLines();
    XCopyArea(display, backingStore, main_window,
                     gc, 0, 0, width, height,  0, 0);
         }
       else
       XCopyArea(display, backingStore, main_window,
                 gc, event.x, event.y, event.width,
                 event.height,  event.x, event.y);
     }
 }

 void drawRandomLines()
 {
   long color;
   int  i,x1,x2,y1,y2;

    /*
```

```
      clear it out
    */
    XSetForeground(display,gc,WhitePixel(display,screen));
    XFillRectangle(display,backingStore,gc,0,0,width,height);

    /*
      then do the drawing
    */

    for (i=0;i<50;i++)
      {
        x1 = rand()%WINDOW_WIDTH;
        y1 = rand()%WINDOW_HEIGHT;
        x2 = rand()%WINDOW_WIDTH;
        y2 = rand()%WINDOW_HEIGHT;
        /*
           put some color in order to make it nicer
         */
        color = rand()%(int)rint(pow((double)2,(double)depth));
        XSetForeground(display,gc,color);
        XDrawLine(display,backingStore,gc,x1,y1,x2,y2);
      }
}
```

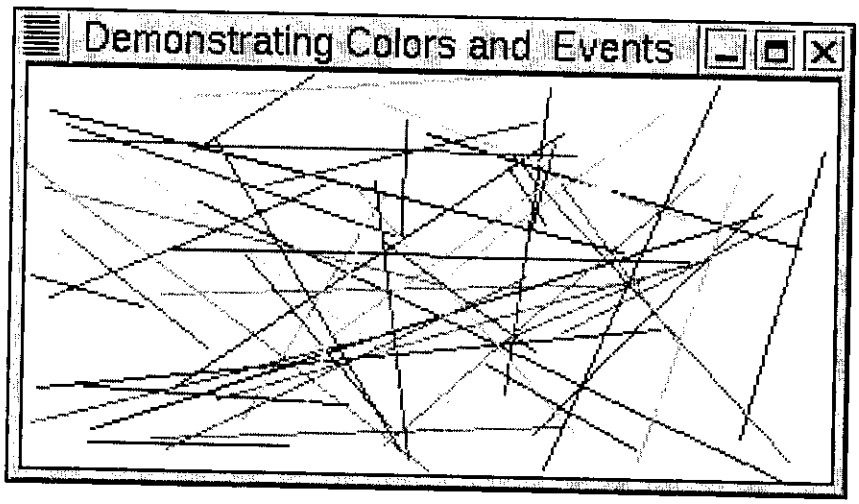


Figure 10: **Color and Events**

2 The Motif Widgets

Up to now we only used **XLib** calls. We managed with some difficulties to implement a *button*, treating mouse button clicks within the *up/down* windows and a sort of label containing the digits. It seems to be a good idea however to standardize on how such a button should react and on how it should look like (contain some text or bitmap, be activated when mouse button1 is pushed and released within its window). This is the so called *look and feel* which is implemented in libraries lying above the **XLib**. Figure 11 shows the different library layers. The lowest layer above the X protocol is **Xlib**, the library we learned in the preceeding sections. Above **Xlib** you find the **Xt** (**X** Toolkit) library onto which the widget sets are built. On top of**Xt** you finally have the widget sets like **Motif**.

A window together with an input/output semantic is called a widget. Typical examples are:

- labels
- push buttons and toggle buttons
- pulldown and popup menus
- boxes and forms containing other widgets
- text input widgets and many more.

For us widgets are simply user interface objects which are the building blocks for our applications. There are several widget sets available on the market. The most common ones are:

- Motif
- OpenLook
- the Athena Widgets
- Gtk
- Qt

The Figure 12 shows some of the Motif widgets.

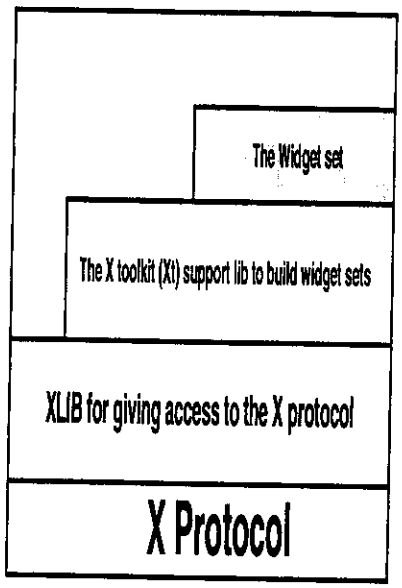


Figure 11: **The structure of Motif and X11 library layers**

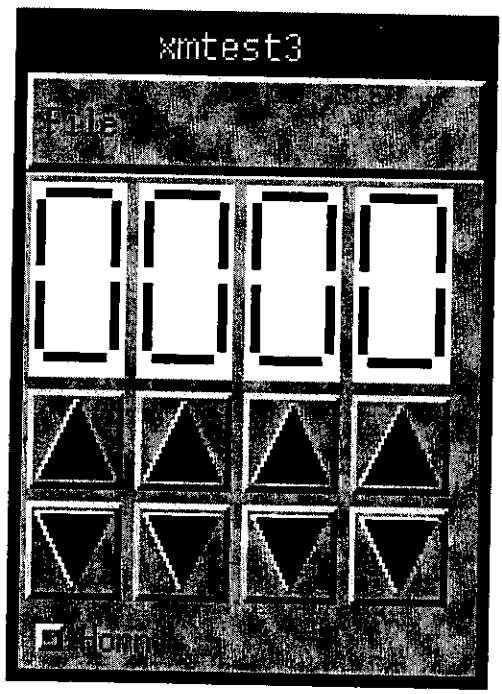


Figure 12: **The Colombo Example showing some Motif Widgets**

During these lectures we will have a closer look at the **Motif** widget set. Why not choose another one?

The Athena widget set is part of X11 and was intended as a testbed for the **X Toolkit** Intrinsics (**Xt**), a library providing routines for building and accessing widgets. It is rather small and simplistic but quite a few applications have been implemented with it. During the first GUI courses we used the **Athena** set because **Motif** was not freely available and **LessTif**, a **Motif** clone, written by the **Free Software Community** was simply too unstable.

During later colleges **Lesstif** was successfully used. Maybe it was due to the pressure from free GUI alternatives like **Lesstif**, **Gtk** and **Qt** that the **O**pen **S**oftware **F**oundation (**OSF**) has finally decided to make **Motif** freely available (well, almost free: it is freely available for free operating systems, not for commercial ones like HP-Unix, Solaris ...). We therefore now have full **Motif-2.1** available.

OpenLook was SUN Microsystems first widget set which they put into the public domain when they switched to Motif for their workstations. It is now deprecated and hardly anyone writes new applications with it.

Qt and **Gtk** are new additions to the widget sets and both are becoming ever more popular. The desktop environment **KDE** is based on **Qt** while **Gnome**, the desktop environment we use, is based on **Gtk**. **Qt**'s access libraries are C++ based and therefore not easily usable during this college. **Gtk** however is a serious candidate for future colleges. This time we are again using **Motif** because now we have free access to the full (former commercial) version, secondly it is still very widely used on all major commercial Unix machines and simply because re-creating a new course is quite a job. Another very serious candidate for future colleges is **Swing**, the widget set for **Java** but again a new language must be learned before the widget set can be employed.

In this section we will learn how to write the *Colombo* example with the help of **Motif** widgets.

Instead of building the window hierarchies the application now builds widget instance hierarchies. Again we have a *root widget* called the **TopLevelShell**. This widget communicates with the window manager to set up the decoration of its window. The child of the **TopLevelShell** is usually a **XmMainWindow** widget, which contains an area for

- Menu bar with pulldown menus
- a container widget used as work area and containing other widgets
- optional scroll bars for the work area
- a command area (which we will not use!)

- and a message area, which may display error messages

As for the windows in **XLib** the windows of the widgets are clipped to their parent widget window boundaries. Even though most applications use a **XmMainWindow** as their base window it is perfectly possible to use a container widget or even a simple widget (see the *hello world* example) for that purpose. In **Motif** we have essentially two types of different container widgets, the **XmBulletinBoard**, where the positions of the children (widgets within the container) are specified as absolute values and the **XmForm**, where all positions are relative to the container or relative to other widgets within the container.

Widgets provide a data structure containing so called resources, which describe them fully. When creating a widget instance all resources are defaulted to reasonable values but they can be changed at creation or later during runtime. The widgets we will be using for our *Colombo Board Simulator* are the following:

- **XmMainWindow**
- **XmForm**
- **XmLabel** (for the digits)
- **XmArrowButton** for the up and down buttons
- **XmCascadeButton** and **XmRowColumn** for the pulldown menu
- **XmToggleButton** for the horn

In additon we will use **XmScale** for a more simple exercise at the beginning.

Before using any widget the **X toolkit** must be initialized and the **TopLevelShell** must be created. This can be done with the call:

```
Widget XtVaAppInitialize (
                          XtAppContext *app_context,
                          String application_class,
                          XrmOptionDescRec *options,
                          Cardinal num_options,
                          int *argc, String *argv,
                          String *fallback_resources,...,NULL);
```

Some comments concerning the parameters to this call:

- **XtAppContext** is an opaque type containg application specific data;
- **application_class**]: this parameter allows to group several applications into *classes* allowing to specify certain properties (resources) for all members of the class. In our case we can create a class **ICTP_examples** and later customize all of its members in a single step;

- you may specify X related options on the command line. The command line parser will pick them out and only leave the non-X ones. Put NULL here.
- **num_options**, since we don't treat special X options put 0 here.
- the ... stand for a NULL terminated list of **resource-name→resource-value** pairs. You will see later what this is good for. For the moment just putting NULL will do the trick.

This initializes the toolkit, opens the display and creates the **TopLevel Shell** who's identifier is returned and which is the great–grandfather of all other widgets. The TopLevelShell communicates with the window manager and gets its decoration. Now we can start to build the widget instance hierarchy. For each type of widget an include file containing widget specific definitions is provided. In order to create a widget, call

```
Widget XtVaCreateManagedWidget
            (String *widget_name,
            WidgetClass widget_class,
            Widget parent,
            ..., NULL)
```

- **widget_name**: each widget gets a name identifying it. These names are used in so-called resource files where the application can be customized by its user without changing a line of source code. We will see this later.
- widget_class: each widget belongs to a class in the *widget class hierarchy*. Here we define the type of widget we want to create.
- **parent**: this is the widget id (note that XtVaCreateManagedWidget has a return value, namely the id of the widget created) of the parent widget. Like in the case of window hierarchies, this allows to create *widget hierarchies*
- ...: NULL terminated list of **resource-name→resource-value** pairs like in the **XtVaAppInitialize** call.

The following table shows the widget names, their class name and the corresponding include file name for the widgets we will be using:

Widget Type	Widget Class Name	include file
XmMainWindow	xmMainWindowWidgetClass	<Xm/MainW.h>
XmBulletinBoard	xmBulletinBoardWidgetClass	<Xm/BulletinB.h>
XmFrame	xmFrameWidgetClass	<Xm/Frame.h>
XmPushButton	xmPushButtonWidgetClass	<Xm/PushB.h>
XmArrowButton	xmArrowButtonWidgetClass	<Xm/ArrowB.h>
XmLabel	xmLabelWidgetClass	<Xm/Label.h>
XmToggleButton	xmTogglwidthonWidgetClass	<Xm/ToggleB.h>
XmCascadeButton	xmCascadeButtonWidgetClass	<Xm/CascadeB.h>
XmRowColumn	xmRowColumnWidgetClass	<Xm/RowColumn.h>

As an example we show how to create an **XmMainWindow** and an **XmForm** widget with default resources apart from width and height, which might be zero otherwise:

```
/*****************************************************************/
/*                                                               */
/* EXAMPLE 1 for MOTIF                                           */
/*    How to initialize the toolkit, create the toplevel shell */
/*    and create a VERY simple widget hierarchy                  */
/*                                                               */
/*****************************************************************/
#include <Xm/XmAll.h>

#define WINDOW_WIDTH  300
#define WINDOW_HEIGHT 150

Widget toplevel, main_window, form;

main(int argc,char *argv[])
{
  XtAppContext theApp;
  Widget       toplevel;
  /*
    initialize the toolkit and create the toplevel shell
  */
  toplevel = XtVaAppInitialize(&theApp,"ICTP_examples",
                               NULL,0,&argc,argv,NULL,NULL);

  /*
```

```
    create the widget instance hierarchy
  */
  main_window = (Widget)XtVaCreateManagedWidget("main_window",
                              xmMainWindowWidgetClass,toplevel,NULL);

  form = (Widget) XtVaCreateManagedWidget("form",
                              xmFormWidgetClass,main_window,
                              XmNwidth,WINDOW_WIDTH,
                              XmNheight,WINDOW_HEIGHT,NULL);

}
```

A widget tree (widget instance hierarchy) can easily be built using several of these **XtVaCreateManagedWidget** calls. Like in the Xlib examples the program still does not put any widgets to the screen. In order to do so the X-toolkit routine corresponding to **XMapWindow** must be called. The function:

XtRealizeWidget `(Widget toplevel);`

does the mapping (and quite a few more things). Instead of calling **XNextEvent** in a loop, the X-toolkit provides a call which already contains an endless loop:

XtAppMainLoop `(XtAppContext theApp);`

Our little example program can therefore be completed by adding:

```
XtRealizeWidget(toplevel);
XtAppMainLoop(theApp);
```

to the end of the program. The result can be seen in Figure 13.

Contrary to our window examples the widgets already contain code for treatment of the **X** events. However, the programmer should be notified of certain sequences of events such as **ButtonPress** followed by **ButtonRelease** within the window of a XmPushButton widget, which corresponds to *pressing the pushbutton* on the screen.

This can be done with **callbacks**: Almost all widgets allow the user to connect callback routines to certain actions. Use the routine:

```
XtAddCallback (Widget widget_id,
               String callback_name,
               XtCallbackProc callback,
               XtPointer client_data);
```

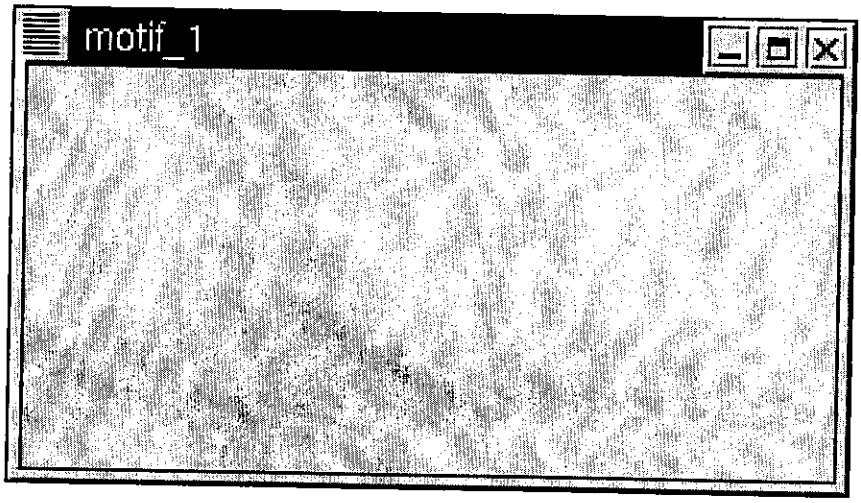


Figure 13: **A first Motif example**

- callback_name is type of callback we are interested in. The most common one is *XmNactivateCallback* used e.g. in pushbutton widgets. Another one: *XmNvalueChangedCallback* is used in Text widgets and indicates that the text widthin the widget has been modified.

- callback is the address of the procedure to be called when the sequence of events has been seen. This is the routine we as programmers have to supply.

- *client_data*: Certain data can be passed as parameters into the callback procedure. *client_data* is a pointer to these data. Most of the time this is simply NULL.

XtCallbackProc is defined as:

```
typedef void (*XtCallbackProc) (Widget widget_id,
                                XtPointer client_data,
                                XtPointer call_data);
```

- call_data is a pointer to widget specific data that might be passed by the widget into the callback procedure.

For the *exit* button in our example programs we will therefore construct a callback procedure:

```
void quitProc(Widget w,
              XtPointer client_data,
              XtPointer call_data)

{
    /* cleanup if needed */
        exit(0);
}
```

After creation of the exit button

```
exit_button = XtVaCreateManagedWidget ("exit_button",
                          xmPushButtonWidgetClass,
                          main_widget, NULL);
```

(creation of a XmPushButton widget) we connect this routine as *activate callback* to the widget:

```
XtAddCallback ( exit_button, XmNactivateCallback,
                quitProc, NULL)
```

Once the widget tree is complete and all callbacks are connected, control is given back to the window system, which will call the registered callback routines as soon as the corresponding event sequence has happened. The call to

```
XtAppMainLoop (theApp);
```

does this.

Here is the improved version of the first **Motif** program including the code activating the pushbutton:

```
#include <Xm/XmAll.h>

#define WINDOW_WIDTH  300
#define WINDOW_HEIGHT 150

Widget toplevel, main_window, form, exit_button;
void quitProc(Widget w, XtPointer client_data, XtPointer call_data);
main(int argc,char *argv[])
{
  XtAppContext theApp;
  /*
```

```
    initialize the toolkit and create the toplevel shell
  */
  toplevel = XtVaAppInitialize(&theApp,"ICTP_examples",
                               NULL,0,&argc,argv,NULL,NULL);

  /*
    create the widget instance hierarchy
  */
  main_window = (Widget)XtVaCreateManagedWidget("main_window",
                           xmMainWindowWidgetClass,toplevel,NULL);

  form = (Widget) XtVaCreateManagedWidget("form",
                                          xmFormWidgetClass,main_window,
                                          XmNwidth,WINDOW_WIDTH,
                                          XmNheight,WINDOW_HEIGHT,NULL);
  exit_button = (Widget) XtVaCreateManagedWidget("exit_button",
                                         xmPushButtonWidgetClass,form,
      XmNlabelString,
                                         XmStringCreateLocalized("exit"),
                                         XmNx,100,XmNy,50,
      NULL);
  XtAddCallback(exit_button,XmNactivateCallback,quitProc,NULL);
  XtRealizeWidget(toplevel);
  XtAppMainLoop(theApp);
}
  /*
    here the callback procedure
  */
  void quitProc(Widget w,
                        XtPointer client_data,
                        XtPointer call_data)
  {
    exit(0);
  }
```

Sometimes it is desired to map non **X** events to callback routines. You may for example want to execute a callback when a certain time has elapsed or when a device driver has data to be read. This can be accomplished with:

```
XtIntervalId XtAppAddTimeOut (XtAppContet app_context,
                              unsigned long interval,
                              timer_proc, client_data);
```

where interval is in msec and **XtIntervalId** is an identifier which allows you to distinguish the timer events in case you use more than one. The corresponding callback routine must be defined as:

```
XtTimerCallbackProc timer_proc (
                              XtPointer interval_id,
                              XtPointer client_data);
```

Similar routines are available for the driver case. Here we would use

```
XtInputId XtAppAddInput ( app_context, source,
                          input_proc, client_data);
```

Again we want to put things together and demonstrate the above principles in a little (and this time even *useful*) demo program.

```
/*************************************************************/
/* A first USABLE Motif program: A very simple digital clock */
/* U. Raich 3-Oct-2000                                       */
/*************************************************************/

#include <stdio.h>
#include <Xm/XmAll.h>
#include <time.h>

#define WINDOW_WIDTH  300
#define WINDOW_HEIGHT 150

Widget       toplevel, main_window, form, time_label;
XtAppContext theApp;
XtIntervalId timer_id;
XtTimerCallbackProc timerProc(XtIntervalId timer_id,
                              XtPointer client_data);
```

```
main(int argc,char *argv[])
{
  /*
    initialize the toolkit and create the toplevel shell
  */
  toplevel = XtVaAppInitialize(&theApp,"ICTP_examples",
                               NULL,0,&argc,argv,NULL,NULL);

  /*
    give it a nice title
  */
  XtVaSetValues(toplevel,XtNtitle,"Uli's Clock",NULL);
  /*
    create the widget instance hierarchy
  */
  main_window = (Widget)XtVaCreateManagedWidget("main_window",
                      xmMainWindowWidgetClass,toplevel,NULL);

  /*
    create a label to display the current time and date
  */

  form = (Widget) XtVaCreateManagedWidget("form",
                          xmFormWidgetClass,main_window,NULL);

  time_label = (Widget) XtVaCreateManagedWidget("timelabel",
      xmLabelWidgetClass,form,
      NULL);
  timerProc((XtIntervalId)NULL,NULL);

  timer_id = (XtIntervalId) XtAppAddTimeOut(theApp,1000,
                                      (void *)timerProc,NULL);

  XtRealizeWidget(toplevel);
  XtAppMainLoop(theApp);
}
  /*
    here the callback procedure
  */
XtTimerCallbackProc timerProc(XtIntervalId timer_id,
```

```
    XtPointer call_data)
{
  /*
     this XmString stuff is explained a little later
     don't worry for the time being
   */
  XmString timeString;
  time_t current_time;

  /*
    get the current date and time
  */
  time(&current_time);
  /*
    convert it first into something readable and then
    into an XmString that can be put into the label widget
  */
  timeString=XmStringCreateLocalized(ctime(&current_time));
  XtVaSetValues(time_label,XmNlabelString,timeString,NULL);
  XmStringFree(timeString);

  timer_id = XtAppAddTimeOut(theApp, 1000,
                                    (void *)timerProc,NULL);
}
```

An this is how it looks like:

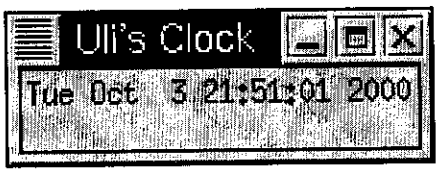


Figure 14: **A useful Motif example: Digital Clock**

The flow of control in an application program using widgets has therefore the following form: (Figure 15)

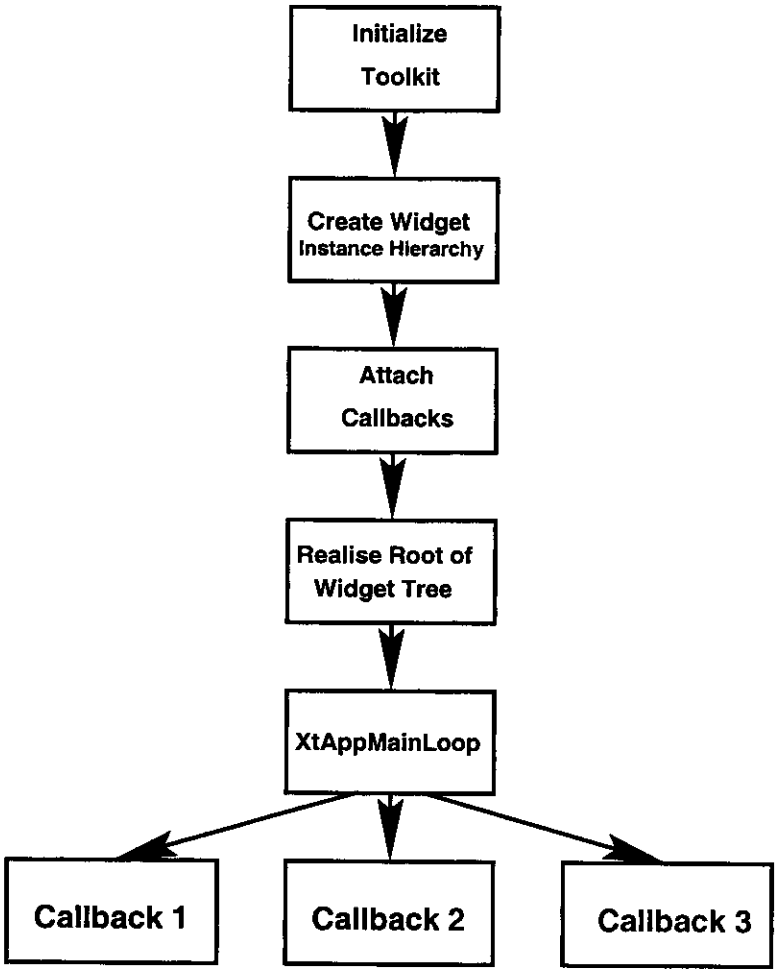


Figure 15: **Flow of Control in a Motif Application**

2.1 The Widget Class Hierarchy

Widgets are built using object oriented programming techniques. This means, that there are some *basic* widgets (basic data structures and access routines, so called methods) which are defined in the Toolkit intrinsics. These basic widgets are the **Core** widget and several **Shell** widgets. A new widget uses part of the data structures and methods defined in these basic widgets and augments them with new data entries and new access routines or modifies some of the properties. Consider a label and a pushbutton widget: The label has some basic resources such as width, height, border width, foreground/background colour, *etc* which it inherits from the **core** widget (all widgets have these properties!). In addition it has a string or a pixmap as-

sociated with it. A command button can be considered to be a label widget having the additional features of being active. In the case of the XmPushButton widget a callback for activation (pushing the button) can be attached.

When a programmer calls **XtVaCreateManagedWidget** a widget instance of the specified class is created. This instance contains the individual values for the label string, the colours, *etc* while the class provides some additional data fields valid for all instances and all the access routines. This is why we always insist on talking about the widget **instance** hierarchy and not just about the widget hierarchy!

To change the default layout of a widget we should access its resources. These changes can be performed during widget creation or using the **Xt** routine **XtVaSetValues** at runtime.

```
XtVaSetValues ( Widget widget_id,
                String name1, XtArgVal value1,
                String name2, XtArgVal value2, ..., NULL);
```

Its counterpart for fixed length argument lists (resource name - resource value pairs) is **XtSetValues**. As you may expect, similar routines for toolkit initialization (**XtAppInitialize**) are also available. Before using these routines we must fill an argument list, where each element is of the following type:

```
typedef struct {
  String     name;     /* name of resource to by modified */
  XtArgVal   value;
               } Arg, *ArgList;
```

A Macro has been defined to accomplish this:

```
XtSetArg (Arg arg, String resource_name, XtArgVal value)
```

This argument list and the number of entries may be specified in the widget instance creation routine or in **XtSetValues**:

```
void XtSetValues (Widhet widget_id, Arg args, int num_args);
```

Imagine we want to set some text like "Quit" in the exit button and set its width and height to fixed values:

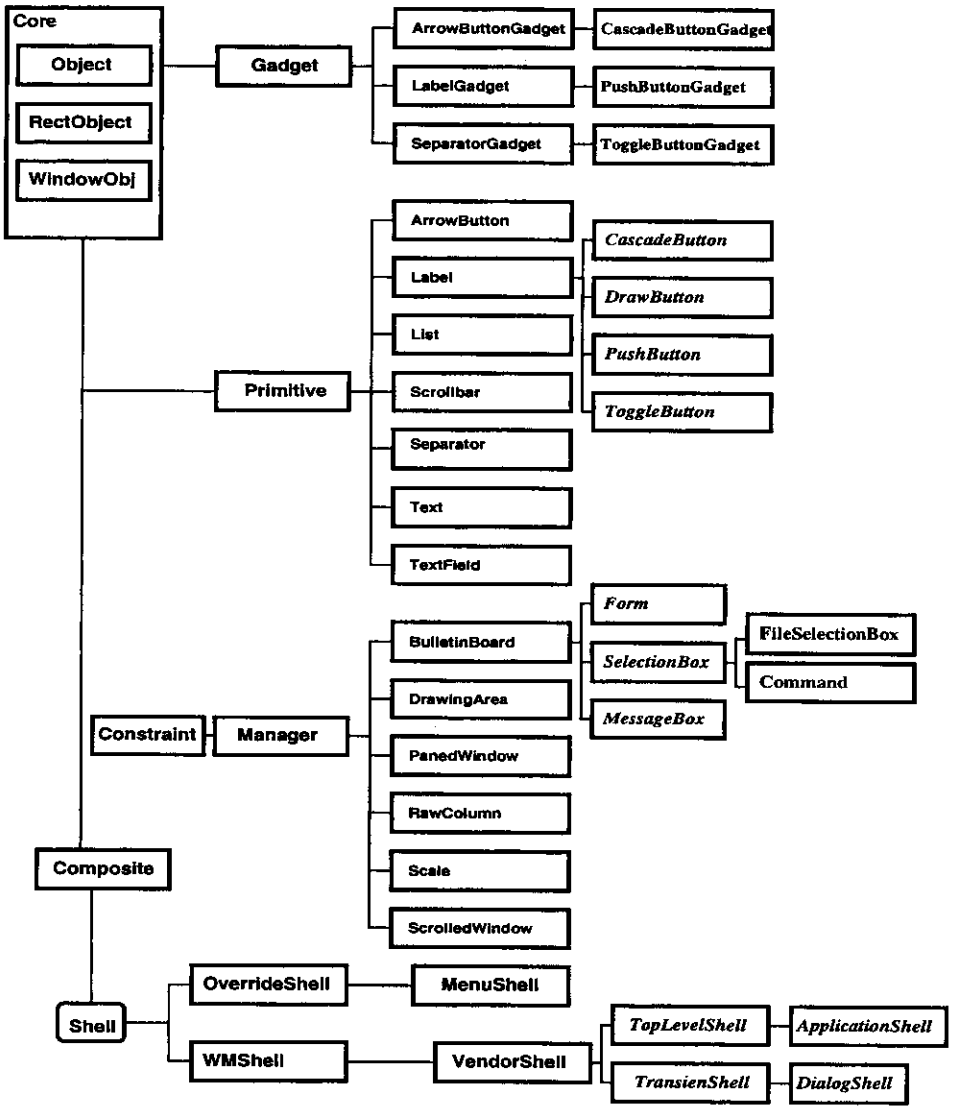


Figure 16: **The Motif Widget Class Hierarchy**

```
Arg          args[5];
XtSetArg    (args[0],XmNlabelString, label_string);
XtSetArg    (args[1],XmNwidth, 100);
XtSetArg    (args[2],XmNheight, 50);
XtSetValues (exit_button, args, 3);  /* will set these three */
                                     /* resources at runtime */
```

(Defining text strings in Motif is a little tricky, see the next section on XmString functions). In the same manner it is possible to read back resources from a widget:

XtSetArgs `(arg[0], XmNlabelString, &return_string);`

XtGetValues `(exit_button, args, 1);`

will return the label string into return_string. Of course the variable length counterpart **XtVaGetValues** is also available.

The next step is to give you a compilation of resources that you will need for development of the exercises on widgets. This list is of course far from being complete. We therefore encourage you to have a look into the **LessTif Manual**, which you have online. You can access it from the root window pulldown menu.

2.2 The compound string (XmString)

You would expect that putting text onto the widgets should be one of the simplest things you can do. Well, you are mistaken! While the **Athena** widget-set uses simple text strings (`char *`) for its labels, titles *etc.* **Motif** goes a step further. Imagine you want to intermix *greek* and *latin* characters, you want to write a text in *German* (having those strange umlauts) or in French, or you want to intermix *Cyrillic* and *latin* text. All this is possible with the **Motif** *compound string conception.* So, you can have labels like $\varepsilon = 25\pi$ *mm mrad* (used very often in accelerator physics!) very easily.

For the course we will restrict ourselves to simple english text but of course you are encouraged to play and try things out. These are the calls that will be enough for you:

- `XmString` **XmStringCreateLtoR** `(char *text, char *tag)`

  In our case text will be the character string we want to convert into a **XmString** and tag will be set to **XmSTRING_DEFAULT_CHARSET.**
  The call will also treat embedded '\n' correctly.
- `XmString` **XmStringCreateLocalized** `(char *text)`

  Here text must be a NULL terminated string without embedded '\n'.

The normal use of **XmStrings** can be demonstrated by setting a new text within a label:

```
XmString   new_string;
Widget     label;
/* some code initializing label...*/
   new_string = XmStringCreateLocalized("My new String");
   XtVaSetValues (label,XmNlabelString,new_string,NULL);
   XmStringFree  (new_string);
```

The last line is needed because the calls creating **XmStrings** allocate the memory space they need for the **XmString**. The deallocation however is left to the user of the call. The same result may be obtained in a more elegant fashion passing through so-called resource converters built into Motif. In order to be able to specify resource values in resource files (see section on resource definitions) routines that convert ASCII strings to internal resource representation, in this case string $\longrightarrow$ XmString are available. These can be called using *XtVaTypedArg*:

```
XtVaSetValues (label,XtVaTypedArg,XmNlabelString,
       XmRString,"Push Me",strlen("Push Me")+1,NULL)
```

2.3 Pixmaps

As you can see from the table of resources later, an **XmLabel** not only can display text strings in all variations, it is also possible to show **Pixmaps**. The easiest way to build bitmaps is the bitmap editor (see section on **X Pixmaps**) which stores bitmaps in an **X** specific way onto disk. **Motif** provides a series of calls allowing someone to read these files and convert them into Pixmap structures that can be used in labels, pushbuttons *etc.* We need this feature when we want to generate labels which ressemble the seven segment displays closely.

```
Pixmap XmGetPixmap (Screen *screen, char *image_name,
       Pixel foreground, Pixel background)
```

- screen is a structure describing the screen you use (not the integer we used to specify in the Xlib examples). You can get a pointer to this structure with the call: Screen ***XtScreen**(Widget w)
- image_name is the filename of a bitmap file

If things go wrong the result will be **XmUNSPECIFIED_PIXMAP**. If the call finds out that the same pixmap had been loaded before, it is not needed to go out to the disk, but it can pick up the pixmap from the *pixmap cache*.

A typical sequence to show Pixmaps is:

```
#define SCREEN 0

Widget toplevel, main_window, form, label;

main(int argc,char *argv[])
{
  XtAppContext theApp;
  Pixmap       smiley_pixmap;
  Screen       *screen;
  Display      *display;
  /*
    initialize the toolkit and create the toplevel shell
  */
  toplevel = XtVaAppInitialize(&theApp,"ICTP_examples",
                               NULL,0,&argc,argv,NULL,NULL);


  /*
    create the widget instance hierarchy
  */
  main_window = (Widget)XtVaCreateManagedWidget("main_window",
                          xmMainWindowWidgetClass,toplevel,NULL);

  form = (Widget) XtVaCreateManagedWidget("form",
                               xmFormWidgetClass,main_window,NULL);

  label = (Widget) XtVaCreateManagedWidget("pixmap_label",
                                           xmLabelWidgetClass,form,
                                           NULL);


  display = XtDisplay(toplevel);
  screen  = XtScreen(toplevel);
  smiley_pixmap = XmGetPixmap(screen,"smiley.bm",
                              BlackPixel(display,SCREEN),
                              WhitePixel(display,SCREEN));

  if (smiley_pixmap == XmUNSPECIFIED_PIXMAP)
    {
      printf ("pixmap no good\n");
      exit(-1);
    }
```

```
  XtVaSetValues(label,XmNlabelType,XmPIXMAP,
                XmNlabelPixmap,smiley_pixmap,NULL);

  XtRealizeWidget(toplevel);
  XtAppMainLoop(theApp);
}
```

2.4 The Core Widget

As we have seen in the *Motif Class hierarchy*, all widgets have **Core** as a superclass. For this reason all widgets inherit the resources defined in **Core**. We only put those resources into the Table 3 that you will definitely need for the solution to the exercises but many more are available. Please have a look at the **Motif** or **Lesstif** docs.

Table 3: Some resources of **Core** widget

Resource Name	Type	Default	Description
XmNx	Position	0	x positon relative to the origin of the parent
XmNy	Position	0	y positon relative to the origin of the parent
XmNwidth	Dimension	dynamic	width of the widget. By default the geometry management decides which width is needed
XmNheight	Dimension	dynamic	height of the widget. The same as in XmNwidth

2.5 The XmMainWindow

Here are a few resources for the *XmMainWindow*. Again the list (Table 4) is far from being exhaustive. So please have a look at the **Motif** documentation. There you will find different sets of resources: Firstly the specific resources for the widget and secondly all the resources of its super classes. You will find that the widget class hierarchy explained previously (The *XmPushButton* will

Table 4: Some resources of **XmMainWindow** widget

Resource Name	Type	Default	Description
XmNworkWindow	Widget	NULL	Container widget that constitues the work area. Most of the different user widgets will be put here
XmNmenuBar	Widget	NULL	The menu bar containg pulldown menus. Usually ther are at least three of them: the *File* menu the *Edit* menu the *Help* menu
XmNshow Separator	Boolean	False	Use **XmSeparators** to separate the different **XmMainWindow** areas.

have its own resources, then the resources of its superclass *XmLabel*, then the resources of *XmPrimitive* and so on (see Figure 16)).

2.6 The XmBulletinBoard

Sorry, there is no description of this widget within this script. For the exercises we only use resources inherited from the Core Widget namely:

- **XmNx**
- **XmNy**
- **XmNwidth**
- **XmNheight**
- **XmNbackground**

2.7 The XmForm

The **XmForm** widget is a container widget performing geometry management on its children. The children of a form may specify their position relative to each other or relative to their parent. When a widget is child of a form it has the following additional resources, some of them shown in the Table 5:

Table 5: Some resources of **XmForm** widget

Resource Name	Type	Default	Desciption
XmNtopAttachment	unsigned char	XmATTACHMENT_NONE	describes where to attach the widget. Some possibilies: XmATTACH_FORM XmATTACH_WIDGET XmATTACH_OPPOSITE_ _WIDGET XmATTACH_POSITION
XmNbottomAttachment	unsigned char	XmATTACHMENT_NONE	see above
XmNleftAttachment	unsigned char	XmATTACHMENT_NONE	see above
XmNrightAttachment	unsigned char	XmATTACHMENT_NONE	see above
XmNtopWidget	Window	NULL	widget onto which we hook on
XmNbottopWidget	— // —	— // —	— // —
XmNleftWidget	— // —	— // —	— // —
XmNrightWidget	— // —	— // —	— // —
XmNtopOffset	int	0	Offset for the attachment
XmNbottopOffset	— // —	— // —	— // —
XmNleftOffset	— // —	— // —	— // —
XmNrightOffset	— // —	— // —	— // —
XmNfractionBase	int	100	used for relative position
and there are many more	see the **LessTif** documentation		

It is also possible to give the position in percentage of the total **XmForm** width and height:

This is done with the resource **XmNfractionBase**

- Set the attachment type to **XmATTACH_POSITION**
- Set **XmNfractionBase**, for example, 100
- Now if you set **XmNtopOffset** to 30 then the widget will be placed at 30% of the **XmForm** height

2.8 The XmScale

The **XmScale** widget is going to be used in an introductory (and therefore simpler) example where we use it as a linear indicator for an analog value. We want the scale to be vertical with the maximum value on top. The actual

value should also be printed as a number. The range of values is defined to be 0 — 5000, which may stand for **0 mV** to **5000 mV**, the digital values we get from *Ang's IO board*. Table 6 shows the nessesary resources.

The **XmScale** widget also has so–called convenience routines which ease the reading and writing of scale values:

- **XmScaleGetValue** `(Widget w, int *value_return)`
- **XmScaleSetValue** `(Widget w, int value);`

do what you would expect.

Table 6: Some **XmScale** resources

Resource Name	Type	Default	Desciption
XmNshowValue	Boolean	False	show not only the analog value by setting the position of the scale, but also its numerical value
XmNtitleString	XmString	NULL	the title
XmNorientation	unsigned char	XmVERTICAL	XmVERTICAL or XmHORIZONTAL
XmNprocessingDirection	unsigned char	dynamic	XmMAX_ON_TOP XmMAX_ON_BOTTOM XmMAX_ON_LEFT XmMAX_ON_RIGHT

2.9 The XmLabel

Labels have two different visual aspects, they may either display text or pictures in form of pixmaps.

Since we will use both in the *Colombo* exercise here are the resources (Table 7) to be changed:

2.10 The XmArrowButton

The **XmPushButton** widget, being a subclass of the **XmLabel** widget, has got all the label widgets resources with the possibility to connect an activation callback in addition.

The **XmArrowButton** reacts as a **XmPushButton**, but already provides arrows as labels. The arrow direction can be specified by means of resources shown in Table 8

Table 7: Some **XmLabel** resources

Resource Name	Type	Default	Desciption
XmNlabelString	XmString	label name	String to be displaying as the label
XmNlabelPixmap	Pixmap	none	Bitmap to be displayed instead of a text string
XmNlabelType	unsigned char	XmSTRING	How to justify the label

Table 8: Some **XmArrowButton** resources

Resource Name	Type	Default	Desciption
XmNarrowDirection	unsigned char	XmARROW_UP	direction of the arrow: XmARROW_UP XmARROW_DOWN XmARROW_LEFT XmARROW_RIGHT

2.11 Pulldown Menus

Still missing is the way to construct menus. As explained above the children in the widget instance hierarchy are always clipped to their parent windows. When creating menus this is not acceptable and we must therefore create another shell widget, which will contain the menu. On the other hand we don't want decoration of the window coming up when we activate the menu. This can be accomplished by creating a *Popup shell.*

Luckily enough Motif provides a *convenience routine* which does all the work for us:

```
file_menu = (Widget) XmCreatePulldownMenu (
            Widget parent_widget,
                                char *widget_name,
                                Arg *args, int no_of_args)
```

creates the pulldown menu. However the menu as yet has no entry in it and is neither hooked onto a button which will pop it up, nor placed into a menu bar. Even worse, it will not appear on the screen because it is not managed. Managing a widget is somehow similar to mapping a window in **XLib**. It can be accomplished by

XtManageChild `(Widget widget_id);`

In order to get rid of the other problems we first create a pushbutton and place it into the pulldown menu:

```
label_string = XmStringCreateLocalized ("Quit");
quit_button  = XtVaCreateManagedWidget ("quit_button",
                        xmPushButtonWidgetClass,
                        file_menu,
                        XmNlabelString, label_string, NULL);
                        XmStringFree(label_string);
```

then we create the menu bar with another convenience routine:

```
menu_bar     = XmCreateMenuBar (
                        main_window,
                        "menu_bar", args, (Cardinal)NULL);
```

and finally the button that pops up the menu:

```
label_string = XmStringCreateLocalized ("File");
file_button  = XtVaCreateManagedWidget ("file_button",
                        xmCascadeButtonWidgetClass,
                        menu_bar,
                        XmNlabelString, label_string,
                        XmNsubMenuId, file_menu,NULL);
XmStringFree (label_string);
```

The resource **XmNsubMenuId** tells the CascadeButton, which menu to pop up once it is activated. Of course the **menu_bar** must be placed into the main window which can be accomplished with

```
XtVaSetValues (main_window, XmNmenuBar, menu_bar, NULL).
```

Now

- **menu_bar** is the standard menu bar of the **XmMainWIndow** that forms the base of our application
- it contains a pulldown menu named *file_menu*
- this *file_menu* contains a single button (that can be activated once we attach a callback procedure to it) namely the **quit_button**
- the file menu is visible and can be popped up through a **XmCascadeButton** named **file_button**

I know this looks pretty complex, but even though pulldown menus are extremely common in **GUI**s they are amongst the most complex structure you can have in user interface programming. Here we put all the menu calls together into a working example: (Put this code into the previous one right after the creation of the main window)

```
  menu_bar     = (Widget) XmCreateMenuBar(main_window,
                                          "menu_bar",NULL,0);
  file_menu    = (Widget) XmCreatePulldownMenu(main_window,"
                                         pulldown_menu",NULL,0);
  exit_button = (Widget) XtVaCreateManagedWidget("exit_button",
xmPushButtonWidgetClass,file_menu,
                                XtVaTypedArg,XmNlabelString,
                                XmRString,"Quit",
                                strlen("Quit")+1,NULL);
  XtAddCallback(exit_button,XmNactivateCallback,quitProc,NULL);

  file_button =  (Widget) XtVaCreateManagedWidget("file_button",
                               xmCascadeButtonWidgetClass,menu_bar,
                               XmNsubMenuId,file_menu,
                               XtVaTypedArg,XmNlabelString,
                               XmRString,"File",
                               strlen("Quit")+1,NULL);
  XtManageChild(menu_bar);
```

2.12 Dialog Boxes

Up to now all widgets came up onto the screen once the toplevel widget has been realised. Very often however we want a box with an error or warning message to pop up only if an error condition has been encountered. This can be done with a **XmMessageBoxDialog** widget.

We create it with

XmCreateErrorBox `("error_box",parent,args,no_of_args);`

and we manage it only once the error has happened. This box contains several buttons, one of which will automatically unmanage the box and thus make it disappear. The used resource is shown in Table 9.

Table 9: A resource for **XmMessageBoxDialog**

Resource Name	Type	Default	Desciption
XmNmessageString	XmString	""	error message

2.13 Connections of widgets to XLib

For several widgets a bitmap id can be used in order to display pictures in buttons, labels etc. When creating a bitmap however we need the identifier of the opened server connection (display variable) or a window id. In order to get this information for a specific widget (which window corresponds to the **main_widget** for example) several calls are available:

```
Display XtDisplay (Widget widget_id) returns the server connection id
Window  XtWindow (Widget widget_id) — // — the widgets window id.
Screen  XtScreen  (Widget widget_id) — // — — // — screen structure
```

Using these calls you may now happily intermix **Xm**, **Xt** and **Xlib** calls.

Now that we know everything needed to build the **GUI** for the *Colombo* example, here is a picture of the widget classes needed for the program:

2.14 Widget Resources

In the previous section we have seen that each widget has associated with it a large number of resources (**XmNwidth**, **XmNlabelString**, **XmNbackground**, **XmN...**) which describe it. These resources can be initialized during the creation procedure of the widget and modified by the running program. Many of the resources need only initialization (or even keep their default values) and are untouched during run time. Think of the label string on a label widget for example.

The **Xt** library allows another very elegant way to modify resources: *the resource file*. This file contains *resourcename-resource* value pairs and it is read during program startup. A typical resource file is `.Xdefaults` in you home directory, which you should have a look at.

Now the question is: How do we specify a widget and its resources. The resource names are the same as the names used within **XtSetValues**, with the leading **XmN** taken away

XmNwidth $\longrightarrow$ width, **XmNlabelString** $\longrightarrow$ labelString, etc.

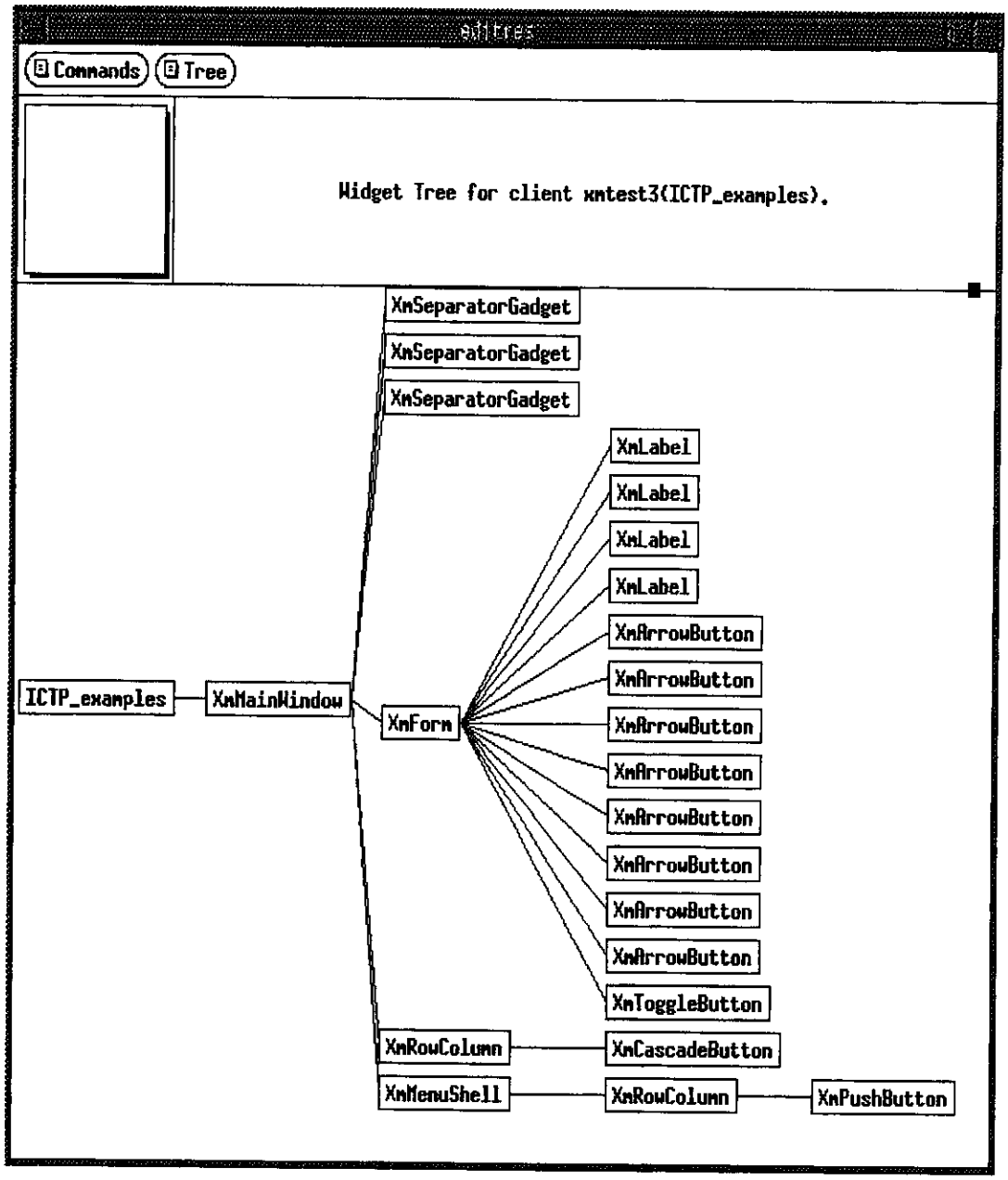


Figure 17: **Widget Instance Hierarchy (classes) as seen by editres**

The widget is specified by giving its path through the widget tree:
digit_label_0 would then be called:
colombo.mainwindow.frame.digit_label_0.width: 45

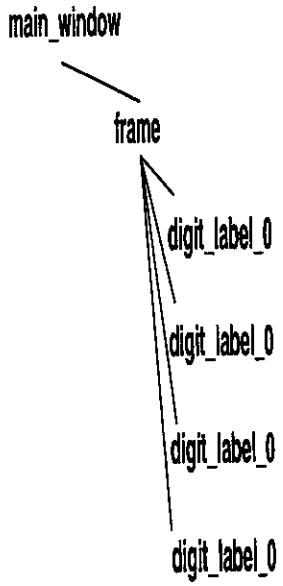


Figure 18: **Specifying Resources in a Resource File**

Like with filenames in Unix wildcards are allowed. The * stands for *any widget* and `*.digit_label_0.width: 50` would most probably have the same effect as the full specification above. Now you also understand why we always give names to the widgets (the string in the widget creation routine **XtCreateManagedWidget**). These names are used for widget identification in the resource file.

To go even one step further we can also specify widget classes instead of widget instances: `*XmLabel` stand for any **XmLabel** widget within any application and there are usually many more **XmLabel** widgets than there are widgets of name `digit_label_0` within an application.

In the toolkit initialization you can also give a classname to your program. With this you could for example. group all editors into a common class Editor or (as we have done) group all solutions to the college exercises into a class `ICTP_examples`.

You may immediately spot interesting possibilities by applying the concept of a resource file:

- Have language dependent resource files. You can then modify the text in an application for a given language by just creating a language dependent resource file. This allows you to change the language for different users without touching a single line of program code.

- Colours are a matter of taste. Is your taste different from a program authors taste? No problem, create a resource file and change the colors.

Here is part of your `.Xdefaults` file:

```
!
! ICTP examples
! ITCP_examples*bitmapFilePath: /usr/local/include/X11/bitmaps
ICTP_examples*font:                                *times-bold-i-*-140-*
ICTP_examples*main_widget*form.background:         dark olive green
ICTP_examples*main_widget*Label.foreground:        red
IOsimulator*background:                            grey75
IOsimulator*XmText*background:                     ivory
IOsimulator*XmText*fontList:\
                                  -adobe-*-*-r-*-*=24-*-*-*-*-*-*
```

For those who want to observe the effect of changing resources before putting them into the resource file a very neat program has been written: **editres**. You find it under *Utilities* on the root window menu. This program shows the complete widget instance hierarchy and gives you access to any resource for any widget within your application. The Figures 19 and 20 show typical screen dumps for our *Colombo* program.

3 Using an Interactive GUI Builder

A very popular design paradigm in GUI programming is the so-called MVC (for Model View Controller) concept. The design is divided into those 3 blocks. The **model** contains all the code needed to modelize the problem, in our case it would contain the data structures and access routines needed for the Colombo simulator like the current values for the digits etc. The **view** is the GUI that is presented to the user, namely the hierarchy of Motif widgets needed to control the model. The **controller** will take the events coming from the View and send messages to the model and view such that model and view updates reflect the actions expected by the user.

The View, namely the static layout of the User Interface can be generated by interactive GUI building tools. The one we provided you is called **vdx**

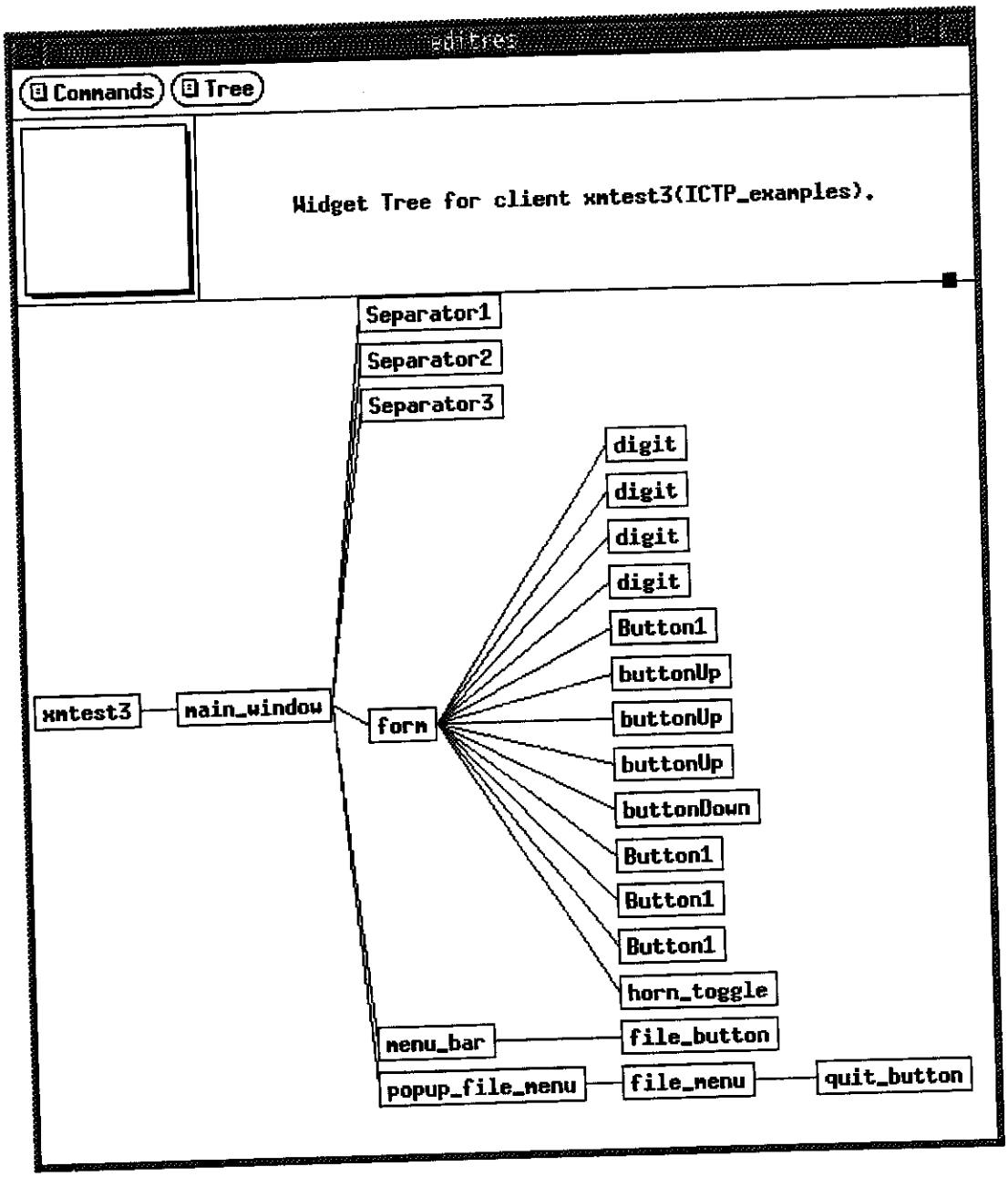


Figure 19: **Widget Instance Hierarchy (Widget Names)**

which stands for Visual Builder for X. You might ask yourself, why we did not present the GUI builder right from the beginning, sparing you the tedious job of building a GUI using direct Motif calls. The answer is rather simple: The GUI builder is a rather complex tool that can only be used to a maximum

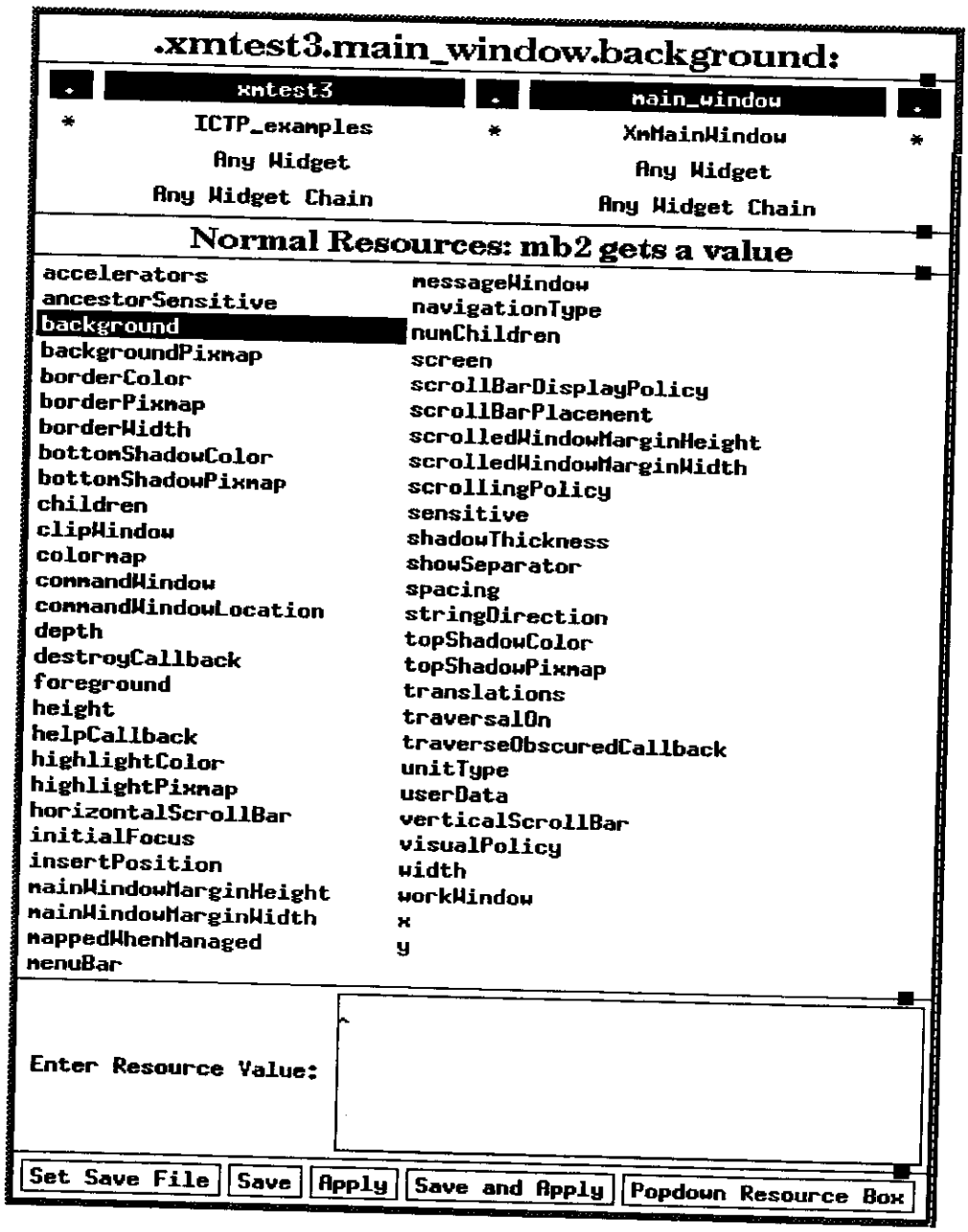


Figure 20: **The Resources**

profit as soon as you understand what is going on behind the scene. As long as you don't understand expressions like *widget class and widget instance hierarchy* as long as you don't know how what *resources* are, it does not make much sense to explain a GUI builder. The **View** of Colombo simulator that

we had built by hand, we will now build using the interactive program. The following Figures all show windows displayed during the building process by vdx. First we must define a *Project*, give it a name and a class (these values will be passed to XtInitialze) and then we can generate the interface with the *widget class palette*. The Figure 21 show the state of the builder during the process of designing the Colombo user interface. The main window as well as the menu bar with the file menu containing the exit button is already done as well as 2 digit labels and and up and a down arrow button.

As you can see the screen looked rather messy when using a GUI builder due the the great number of windows that you need to have in order to access all the tools needed for definition of the widget instance hierarchy. Here is a list of tools that you probably use during the build of a project:

- The main window. This will give you access to all the tools the builder provides and it will usually provide a menu bar with plenty of buttons. Here you can define a new project, save your files and reload them, switch between *design mode* and *test mode* and many things more.
- A window visualizing the GUI. This window (there may be several such windows) will show you how your GUI will look like. Here you will first define a shell widget and then place your widgets inside.
- The Widget Palette. The widget palette offers (in graphical form) a list of all available widgets. You can select any widget from this palette and place the selected widget into you Interface. This allows you to construct your widget instance hierarchy.
- The widget tree. This window shows your widget instance hierarchy in tree form and allows the selection of certain widgets e.g. in order to change their resources or in order to select them as parents for new widgets.
- The resource box. After having selected a widget you may visualize and change any of ist resources, which is done with this box.
- The Generation window. In this window you will generate source code for the constructed user interface. Usually you generate C source files and the corresponding make file, as well as the resource file.
- Help windows. With all this mess it is necessary to have online help which is provided in *help windows*

This is the end of the GUI lectures. Now its is time for you to apply your knowledge in practice. **Good Luck!**

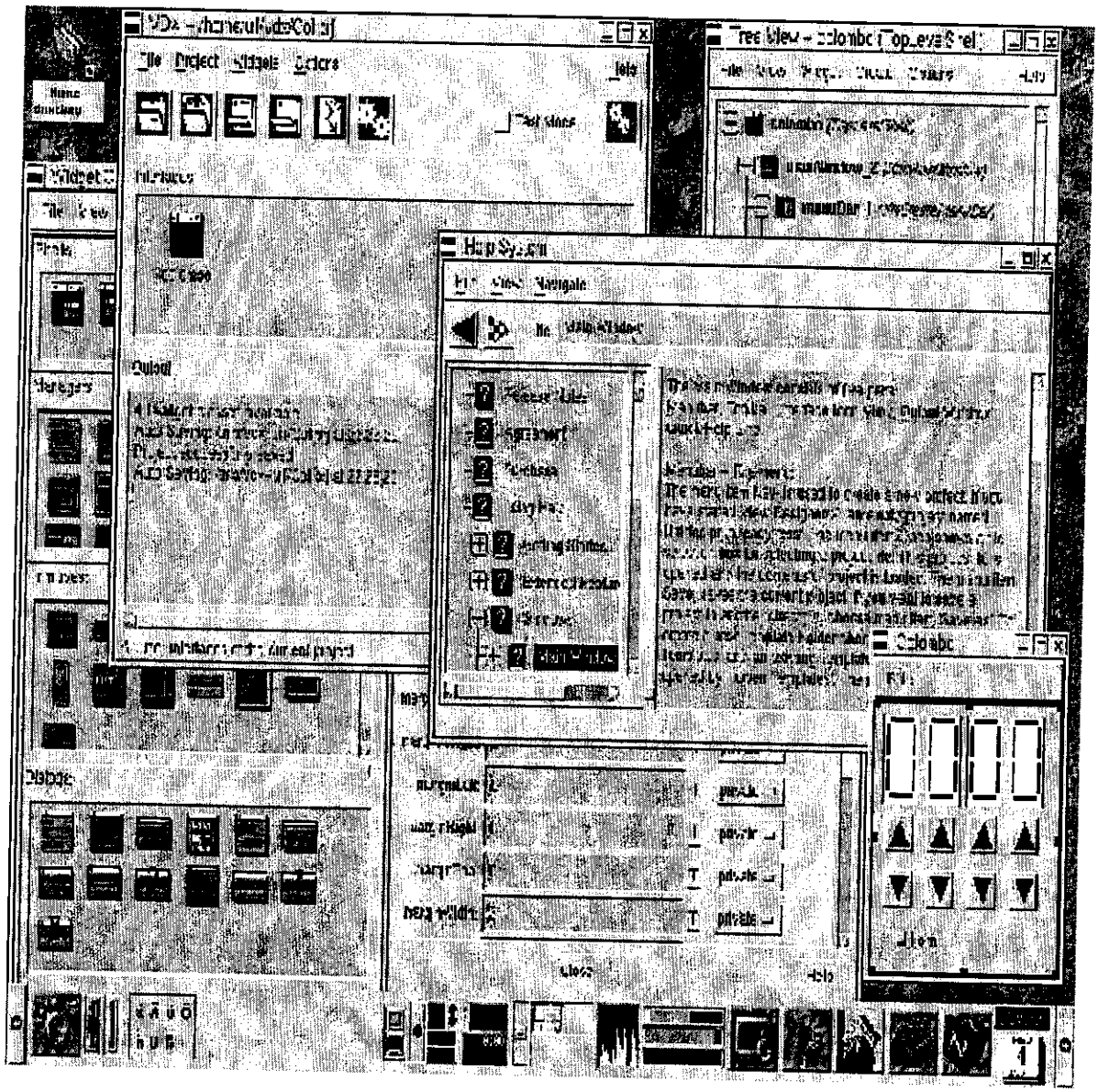


Figure 21: **A Screendump when using the VDX GUI Builder**

Collected Adventures in Linux Device Driver Writing

Sixth College on Microprocessor-based Real-time Systems in Physics

Abdus Salam ICTP, Trieste, October 9–November 3, 2000

Ulrich Raich
CERN - European Organisation for Nuclear Reseach
P.S. Division
CH-1211 Geneva
Switzerland

email: Ulrich.Raich@cern.ch

Abstract

Writing a device driver is one of the most tricky things you can do in programming. The problem is that deep knowledge on the hardware you want to control is needed as much as deep knowledge about the operating system you want to write the driver for. While a software bug in a usual user level program is acquitted with a core dump making a similar fault in system mode when implementing a driver will normally result in a system crash.

This paper gives a step by step introduction on how one should try to tackle the problems. It traces the difficulties the author had himself when trying to provide the *Colombo board* driver. The full driver source code is available for study.

1 Introduction

It may be best to tell you right from the beginning:

Device Driver Writing is a tricky business!

This in fact was the first lecture I learned myself when preparing this series of lectures. I was very proud when I was attributed this course by ICTP because device driver writing has the reputation of being rather difficult. So I was thinking of a course explaining

- all the complicated data structures needed in order to hook up the device driver with the kernel
- the context switch from user to supervisor mode with all its details
- lots of computer science theory of why device drivers are important
- and of course all the details of interrupt and DMA driven device drivers
- connection of file system with block device drivers etc.

I think you see what I mean. *The Theory of Device Driver Writing* might have been the right title. Then I had the splendid idea that it might be good to actually write a driver myself before trying to explain to others how to do it. That was the moment when everything began to go wrong! I started to write the code some 3 months before the course and 2 weeks before I had to give my lectures the driver still did not work! (and of course the transparencies were not prepared either!). The goals of my course became much more modest and I ended up with a course that tells you about all the mischieves I encountered when trying to implement my device driver. No theory! No block device drivers and file systems. Just the story of how I finally managed to get my device driver going. Nevertheless (or perhaps just because the course is now much simpler!) I hope that you will get some insight of

- what a device driver is
- how it works
- and how it is connected to the operating system.

And the best thing of all:

- You get the full source code of the driver
- you can use it
- and you may modify it as ever you like (taking the risk of crashing the system)

2 Generalities

When accessing any type of hardware several problems arise:

- Firstly there are many people who understand often rather complex electronics that make up computer interfaces and there are many people who write splendid software. Finding somebody who understands the operating system writes nicely structured and very robust code (software) and who is capable to read circuit diagrams, understands timing signals and can read datasheets of electronic chips is already a different business. It is therefore reasonable to isolate the code that needs to know all about the intricacies of the hardware and which in addition must be extremely robust (a small bug can bring the whole system down!) into a separate module. Think of a disk driver for example. This module is written and thoroughly tested once and can then be used by everybody.
- In a multi tasking system several concurrent tasks may want to use the same resource, e.g. a line printer and may generate a big mess!
- As already mentionned above, hardware access should only be given to trusted users since an error may easily blow the whole system. This is particularly true for multi user systems.
- Access to fixed memory locations is needed e.g. registers or memory in an interface or even specialized I/O instructions. When a device is capable of generating interrupts or of performing DMA then the story becomes even more complex: For interrupts the program context is changed: A new stack frame is in use and the CPU running mode is changed from user mode to supervisor mode. Therefore device drivers must very tightly cooperate with the operation system kernel.

3 Testing the Hardware

Going through the problems one by one I decided that understanding the hardware was first priority. So, what hardware should I use for my demo device driver? At ICTP the only easily available hardware was the Colombo board, which has been designed for the course 84 in Colombo (Sri Lanka). This board actually consists of 2 parts: a processor/memory/interface part (which in our case will be permanently disabled and which I will not describe any further) and a part simulating some sort of external process to be controlled. This I/O part consists of

- 4 hex seven segment displays (BCD displays in the original version)
- toggle buttons
- 2 pushbutton switches
- a rotary switch with 16 positions
- a voltage to frequency converter, allowing to simulate a simple ADC
- 3 fixed frequencies

The pushbuttons, the voltage to frequency converter and the fixed frequencies may be used to generate interrupts.

This board was designed to be hooked up to a PIA (Motorola M6821, Parallel Interface Adapter) in a M6809 development system with the processor part disabled. Development of programs could then be easily done on the development system using its assembler/linker/loader/debugger. Once everything worked fine a few addresses needed to be changed and the program was blown into EPROM and installed on the Colombo board. Enabling the processor part allowed to run the system in standalone. As I said before, from now on we are only interested in the I/O part of the Colombo board. In order to use it for the device driver I still needed an interface for it on the PC (PIA equivalent). I therefore tried to use the line-printer interface available on most PCs and I manged to get the displays going. However the number of I/O lines especially for input were simply not sufficient and also demonstration of interrupts turned out to be impossible. At the same time I learned that a parallel I/O board had been developped at LIP in Portugal.

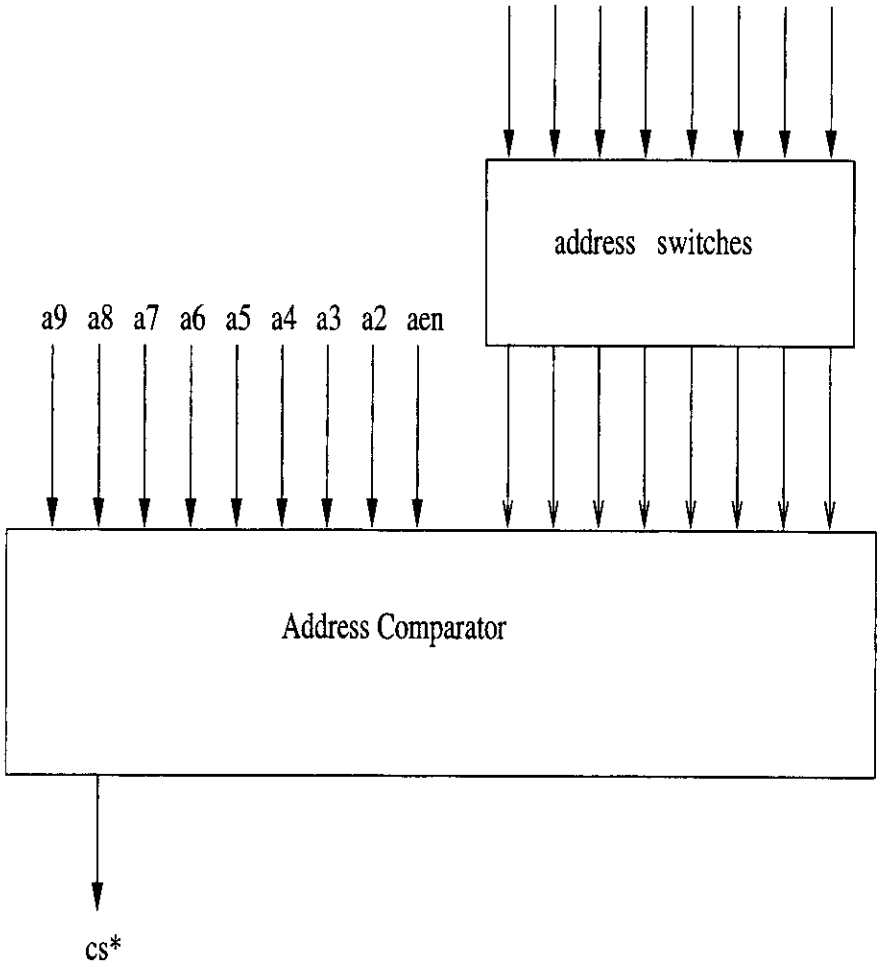


Figure 1: **Address Selection**

The I/O board consists of an Intel 8255 parallel input/output port, known under the name "Programmable Peripheral Interface (PPI)" and some interface logic connecting the 8255 to the PC bus. An 75LS682 comparator chip is used for I/O address selection in conjunction with 8 dual inline switches (see fig. 1, chip U3 in the circuit diagram).

The 8255 has got 2 general purpose 8 bit ports which may be configured as input or output port (Ports A and B). In addition there are 8 more I/O lines which may take over the function of additional input or output bits (4 bit ports) or may be used as handshake lines, depending on the (software) configuration of the chip. Figure 2 shows the chip block diagram.

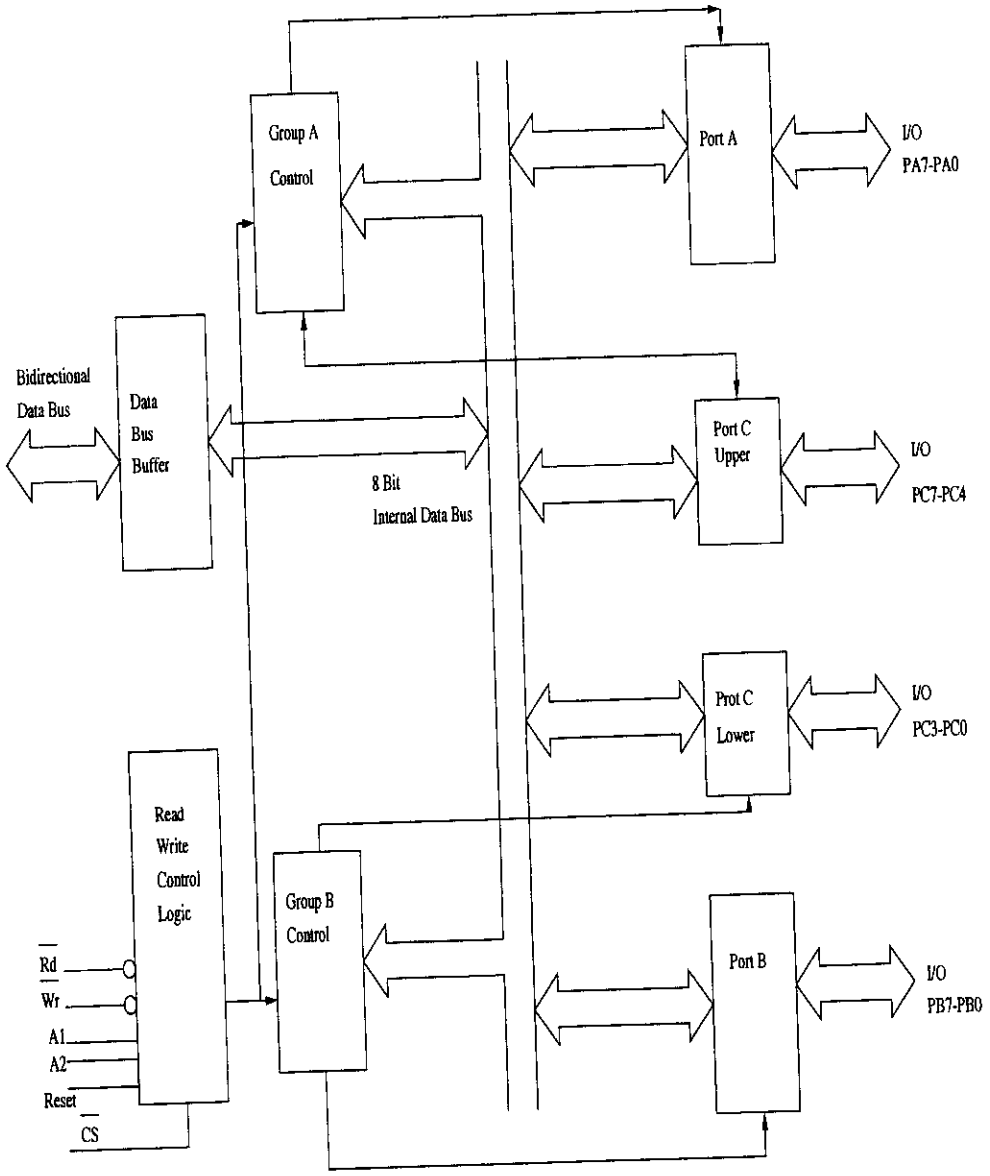


Figure 2: The 8255 Block Diagram

A0	A1	Function
0	0	Port A
0	1	Port B
1	0	Port C
1	1	(write only Control Register

Table 1: Addressing the 8255 chip

From the programmers point of view these ports are presented as 4 registers whose address layout are shown in table 1.

Before communicating with the outside world (the Colombo board in our case) we must configure the 8255 telling it, which port will be used for output and which port for input. In addition we need to specify the type of transfer (latched or not). This is done by programming the control register.

Mode 0: Basic I/O mode (non latched). No interrupts

Mode1: Strobed I/O mode. Here some lines of port C will be used as handshake (strobe) lines and may be used to generate interrupts

Mode 2: Bidirectional mode

How does this map to our equipment hardware? Firstly we need 8 output lines in order to drive the displays. 4 lines are used for the data while the other 4 lines will generate the chip select signals for the registers holding the data of each display.

Then we need 8 input lines for reading of the rotary switch and the toggle and pushbutton switches. Of course the pinout of the PPI board was incompatible with the Colombo board, but a simple passive adapter board did the trick. I finally had the setup shown in fig. 3 .

The hardware was complete and testing could start. I collected all information I was given with my PC and I found a table of hardware addresses, telling me that the address reserved for lpt2: was free.

I therefore set the address switches to 0x13b. (address 0x278 shifted left by 2 and 1 added for aen) I opened my PC (for the first time!) I broke out the metal protection for the I/O slot and I inserted the I/O card into the slot I had selected. I hooked up the cables and the decisive moment has come! I felt quite nervous! Another serious check and...

I switched the PC **on !**

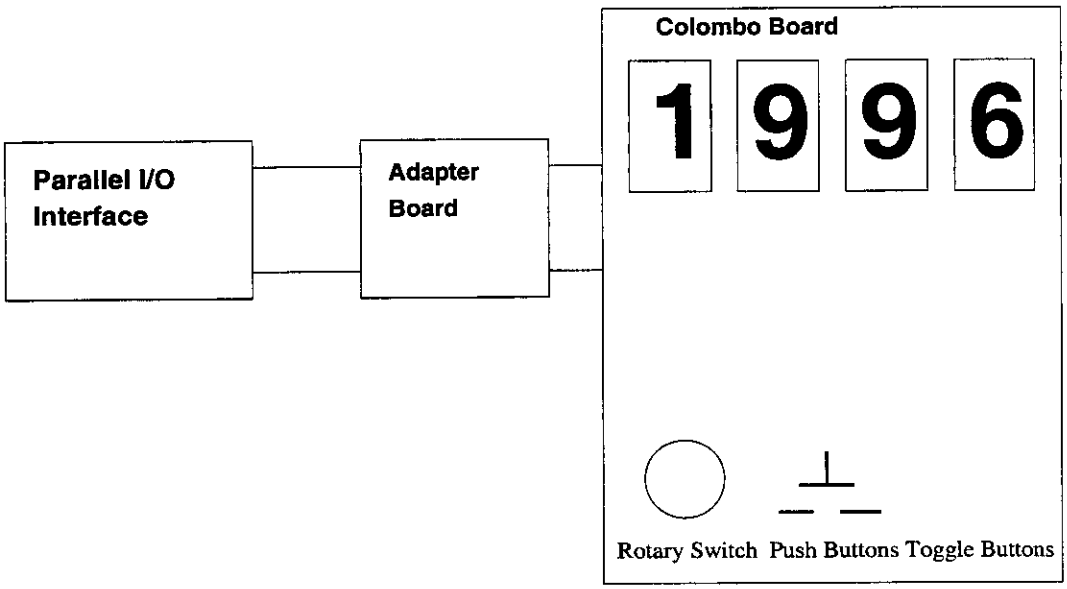


Figure 3: The Hardware Setup

I/O Addresses	Device
000-01F0	DMA Controller 1
020-03F	Interrupt Controller 1
040-05F	Timer
060-06F	Keyboard
070-07F	Realtime Clock
...	some left out
1F0-1F0	Fixed Disk
278-2FF	Parallel Printer Port 2
2F8-2FF	Serial Port 2
300-31F	Protytype Card
378-3FF	Parallel Printer Port 1
3F0-3F7	Floppy Disk Controller

Table 2: I/O Addresses on the PC Bus

Oufff, there was no smoke coming out of the PC and the thing booted normally. However I quickly found out that the floppy did not work any more. Did I finally kill the floppy interface? I re-opened the PC (and decided to leave it open until everything would work or I had to take the PC to the repair shop) took the I/O card out and tried booting again. The floppy worked fine again. So there must have been an address clash between the floppy and the I/O board. (I still don't really know how this comes about!) Looking through the addresses again I selected 0x300 (which turned out to confict with the ethernet card in Trieste, were we finally selected 0x310) which was marked "prototype card". I re-inserted the board and rebooted. This time the machine booted fine again and also the floppy worked normally. Real progress! However I was still unable to talk to the I/O board.

Having a look at the xclock program told me that midnight had passed again! and this was not the first time during the last week. My wife would be angry with me and I tried to find an excuse knowing I would have a hard time with that. The problem was, that she was somehow right but there was so little time left before the course and I absolutely wanted to get the thing going before. Still it was a wise decision to stop at that moment.

The next night (I did all this after working hours) I had another look at the addressing on the I/O board. I controlled the address switches again and they seemed all ok. The only possible source of error was the enable line (en) which was marked without a ~ meaning: active high signal. Mostly enable lines are active low however so I decided that this must have been simply a misprint and I switched the comparator input to active low. It turned out, that I was right. The PC booted, the floppy worked and I was able to talk to the I/O board. This was not the end of the adventure but clearly the end of the first chapter of this novel.

We need to become a little more technical now. You may have asked yourself: What do I mean by talking to the board How do I know if I was able to access the board or not. Which calls are provided within Linux to access external hardware?

The main problem of checking the interface for the first time is to disentangle hardware and software problems. It is therefore important to get an indication that something workes. I tried to find a register that I could just read for the first step e.g. a status register on the I/O card. Finding a reasonable bit combination could be a first indication of successful board access. After that I tried to find a read/write register which I could write with a known bit pattern and read back. Typical bit patterns are : 0x55 and 0xaa (Why?) [1]

[1] Try to write the numbers in binary. You will see that every other bit is set starting

Comparing written to read values gives you a fair idea if things are working, at least if you are successful. Of course checking out the hardware also means writing of very small and simple programs. After having checked read and write access to the interface I wrote a little program that gives a visible indication of something happening on the connected hardware, which was a routine lighting the seven segment displays with known numbers. The program is given below. It consists of two part2: Firstly permission is asked for reading and writing to/from absolute I/O addresses (the addresses of the parallel interface registers). As you might expect in a multitasking and multiuser system access to absolute addresses must be restricted (and they are restricted to the super user only) in order to guarantee system integrity. (A super user is supposed to know what he is doing!) After successful execution of *io_perm*() we have access to our I/O interface registers.

The sequence therefore is:

ioperm(base_address,range,permission);

value = inb(IO_PortAddress);

outb(IO_PortAddress,value);

A similar sequence is available for memory access, e.g. if you want to write into the video memory of a video card directly. Here we would:

open(/dev/mem);

allocate a certain number of memory pages

and map this memory onto the absolute address of the video memory using mmap.

inb, outb etc. are Macros, which are defined in /usr/include/linux/asm/io.h (please have a look at this file !) together with similar ones like inw, inl etc. Since these are "builtin macros" you **must** use the gcc option -O2 in order to get them included into your code. Forgetting -O2 results in unresolved references at link time.

4 Accessing a device driver

You may think, since we now have access to our I/O card, we can read and write data form/to it, well ..., that's it, we have finished! Unfortunately

from bit 0 for 0x55 and bit 1 for 0xaa.

this is not the case. As said before: Any program making use of ioperm, inb, outb or mmap will only run in super user mode. We want to give access to our board to the ordinary user however. In addition there is no resource protection (the board may be written to by several tasks in any wild sequence) and treatment of interrupts or even DMA are excluded. Only the device driver will give you access to these possibilities.

What exactly is a device driver then? and how may an ordinary user access it? We want to slowly approach this question by first looking at the drivers software interface, or said differently: the way a programmer would use the driver.

You have already written programs that make use of files and you have seen the calls:

- open
- close
- read
- write
- lseek etc.

Accessing a device driver is exactly the same. You may think of a device as a **special file** (which is actually the technical term for it). The device is accessed through inodes defined in /dev using the same calls as normal file access. On order to open the device of our Colombo board we would write:

```
fd = open("/dev/ictp0",O_RDWR);
```

In order to write to the board we would fill a buffer and write it to the board:

```
buf[0] = 0x3f; /* fill the buffer */
buf[1] = 0x3e;
buf[2] = 0x3f;
write(fd, buf, 3); /* write to the board */
```

What exactly happens when we do this? The "calls" open, read, write are so-called system-calls and differ from normal subroutine calls. System calls generate software interrupts and doing so change the running mode from (normal) user mode to supervisor mode. After that a subroutine within the system kernel is called and executed in supervisor mode (compare to fig. 4).

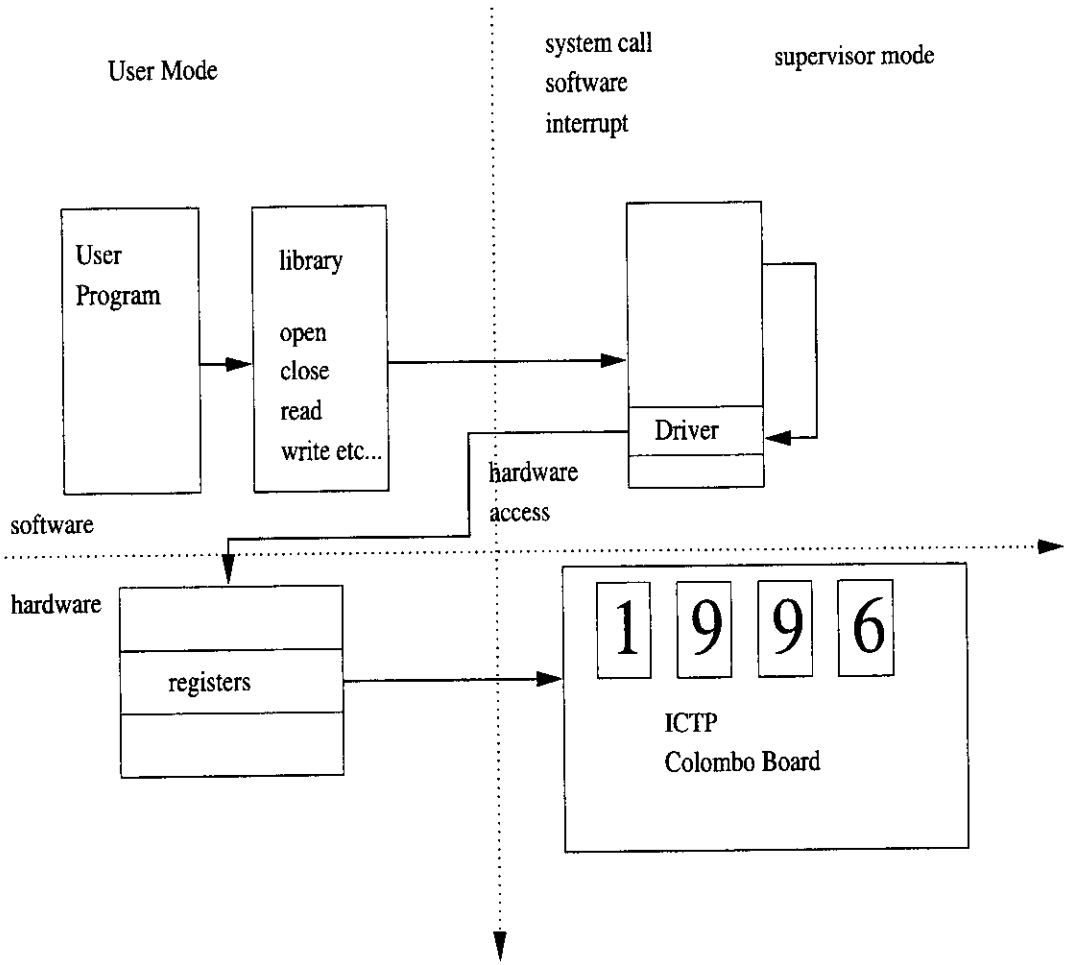


Figure 4: Accessing the Device Driver

You now immediately see that our driver routines are actually executed on the same level as the kernel, they are integral part of the kernel. This also means that errors within the kernel routines are usually unrecoverable (there is no such things as "segmentation fault, core dumped") and will crash the entire system.

The following listing gives an example of access to the ictp driver.

```
/*************************************************/
/* Access the ictp device driver                 */
/* This example writes the displays in RAW mode  */
/* U. Raich 14.3.94                              */
```

```
/*****************************************************/
#include "/usr/include/stdio.h"
#include "/usr/include/fcntl.h"
#include <sys/ioctl.h>
#include "ictp.h"

void main()
{
  int fd,i,ret_code;
  unsigned long mode;
  unsigned char buffer[12];
  short    full_number;

/*
  open the device driver for writing
*/
  fd = open("/dev/ictp0",O_WRONLY);
  if (fd <  0) {
    perror ("Could not open ictp port:");
    exit(-1);
  }
  else
   printf("ictp port successfully opened for writing!\n");


/*
  try out raw mode
  we must code data and chip select signals ourselves

  buf[0] buf[1] buf[2]
  -----|       |------
       --------
*/

  buffer[0]=0x1f;
  buffer[1]=0x17;
  buffer[2]=0x1f;
  buffer[3]=0x9f;
  buffer[4]=0x9b;
  buffer[5]=0x9f;
  buffer[6]=0x9f;
  buffer[7]=0x9d;
  buffer[8]=0x9f;
```

```
  buffer[9]=0x6f;
  buffer[10]=0x6e;
  buffer[11]=0x6f;

  if (write(fd,buffer,12) != 12)
    perror("after write ");

  close(fd);

}
```

5 Representation of the device driver

Having seen how to access the driver from an application we must now figure out how the kernel finds its way into the driver. Trying:

*ls -l /dev/ictp**

will produce the following output:

```
crw-rw-rw-   1 root     root      31,   0 Jan  4 19:07 /dev/ictp0
crw-rw-rw-   1 root     root      31,   1 Jan  4 19:07 /dev/ictp1
crw-rw-rw-   1 root     root      31,   2 Jan  4 19:07 /dev/ictp2
```

where c tells us that the file is actually a character device driver, rw are the usual read and write permission bits and 31 is the major and 0,1,2 the minor device numbers. These numbers are unique in the system. By the way: the device special files are not created by a text editor but by the command:

```
mknod /dev/ictp c 31 0
mknod /dev/drivername, device type, major number, minor number
```

The major number defines the I/O device, the minor number usually indicates a channel number (a serial I/O device may have 4 UARTs representing 4 serial I/O channels, which are driven by a single software module).

As explained before, the driver is an integral part of the operating system and is usually linked into the kernel during system generation. However a software package has been developped for Linux, allowing us to install and deinstall device drivers (or other "modules" like file systems etc.) into a running kernel. This **modules package** provides the following basic programs:

- *insmod*: install a module into the kernel
- *lsmod*: list all installed modules

- *rmmod*: de-install a module from the kernel
- *ksyms*: list exported kernel sysmbols

and the newer versions of the package have in addition:

- *modprobe*: same as insmod but a standard path is searched for the modules while insmod needs the full path name.
- *kerneld*: kernel daemon, started during boot, allows demand loading of modules. The module is simply installed in a standard directory. When an application program calls a device driver routine (ex. "open") and the device driver does not exist in the system, a request to load the corresponding driver is sent to kerneld. kerneld loads the driver via modprobe and the application can continue.

The system you are currently using has all its modules installed in /lib/modules/2.0.33.

6 Implementing the Device Driver, first steps

A device driver always consists of at least 2 files:

- The driver include file (/usr/local/include/ictp.h)
- The driver code itself.

The include file contains the definitions of hardware addresses, register names, and names for each and every bit used within the I/O chip registers. In addition it contains definitions for error codes, names for driver operating modes, ioctl request names and the like. It is used by the driver itself, but it is usually also included by any program using the driver.

The following listing shows the include file provided for the ictp driver:

```
/*****************************************************/
/* Definitions of 8255 addresses and control bits    */
/* U. Raich 31.8.94                                  */
/*****************************************************/

#include <sys/ioctl.h>

#define ICTP_MAJOR          31
#define ICTP_NO              3
```

```
/*
 * defines for 8255 ports
 */

#define ICTP_A base
#define ICTP_B base+1
#define ICTP_C base+2
#define ICTP_S base+3

/*
 * defines ICTP status and control register bits
 */

#define ICTP_MODE_SELECT     0x80
#define ICTP_A_MODE_0        0x00
#define ICTP_A_MODE_1        0x20
#define ICTP_A_MODE_2        0x40

#define ICTP_B_MODE_0        0x00
#define ICTP_B_MODE_1        0x04

#define ICTP_MODE_BLOCKING     0
#define ICTP_MODE_NON_BLOCKING 1

#define ICTP_INPUT_A         0x10
#define ICTP_OUTPUT_A        0x00
#define ICTP_INPUT_B         0x02
#define ICTP_OUTPUT_B        0x00
#define ICTP_INPUT_C_LOW     0x01
#define ICTP_OUTPUT_C_LOW    0x00
#define ICTP_INPUT_C_HIGH    0x08
#define ICTP_OUTPUT_C_HIGH   0x00

#define ICTP_AVAILABLE       1
#define ICTP_NOT_AVAILABLE   0

#define ICTP_SILENCE         0x09
#define ICTP_NOISE           0x08
#define ICTP_BUZZER_BIT      0x10
#define ICTP_BUZZER_ON          1
#define ICTP_BUZZER_OFF         0
```

```
#define ICTP_MODE_RAW          0
#define ICTP_MODE_SINGLE_DIGIT 1
#define ICTP_MODE_FULL_NUMBER  2

#define ICTP_BUSY              1
#define ICTP_FREE              0

#define ICTP_READ_SWITCHES     0
#define ICTP_READ_IRQ5_COUNT   1
#define ICTP_READ_IRQ7_COUNT   2

#define ICTP_ENABLE_IRQ5       0x5
#define ICTP_DISABLE_IRQ5      0x4
#define ICTP_ENABLE_IRQ7       0xd
#define ICTP_DISABLE_IRQ7      0xc

#define ICTP_IBF_B             0x2
#define ICTP_DUMMY             0xff
/*
  the ioctl codes:
*/
#define ICTP_SET_WRITE_MODE    IOC_IN  | 0x0001
#define ICTP_GET_WRITE_MODE    IOC_OUT | 0x0001
#define ICTP_SET_READ_MODE     IOC_IN  | 0x0002
#define ICTP_GET_READ_MODE     IOC_OUT | 0x0002
#define ICTP_SET_BUZZER        IOC_IN  | 0x0003
#define ICTP_GET_BUZZER        IOC_OUT | 0x0004
#define ICTP_ENABLE_INTERRUPT          0x0005
#define ICTP_DISABLE_INTERRUPT         0x0006

/*
  for checking if the board is there
*/
#define ICTP_TSTBIT          0x20
#define ICTP_SET_TSTBIT      0xb
#define ICTP_RESET_TSTBIT    0xa
```

As you can see, all addresses are calculated relative to a single base address *base*. This variable can be set to the 8255 base address using an option (base=0x320) to modprobe. Like this the driver code becomes independant from from the hardware addresses and is configurable for any system.

Before a user program can access the driver it must be included into the system (if kerneld is not running). As already mentioned earlier, this can be

done either by linking it into the kernel at system creation or we register the driver with the operation system once the module containing the driver gets installed with insmod. Therefore we must provide 2 routines:

- *init_module*, which is called by insmod and which will check if the parallel I/O card can be accessed at the base address specified before asking the kernel to register the *ictp* device driver.
- *cleanup_module*, which is called by rmmod and which cleanly removes the module from the system.

Since the driver is an integral part of the operating system and works in supervisor mode, it has no access to the normal C library functions. It cannot be debugged with a normal debugger either (the debugger has no access to supervisory memory!). However a few calls are available to the device driver writer, one of which is *printk*, which is the kernel equivalent to *printf*.

For debugging purposes I therefore put a few printk statements in strategic places in order to be able to follow the execution of my code. When registering the device driver with the system the address of the *fops* table is passed as a parameter. This table contains the entry-points of the driver routines needed for the execution of

- open
- close
- read
- write
- lseek
- ioctl

and a few more system calls. If a call is not implemented the table gets a NULL entry. Here is the *fops* table of our ictp driver:

```
static struct file_operations ictp_fops = {
    NULL,               /* seek       */
    ictp_read,
    ictp_write,
    NULL,/* readdir */
    NULL,/* select  */
    ictp_ioctl,
    NULL,/* mmap */
```

```
    ictp_open,
    ictp_release
};
```

The code for init_module is also given below:

```
/*
 * And now the modules code and kernel interface.
 */

int
init_module( void) {

unsigned char testvalue = 0;

#ifdef ICTP_DEBUG
printk(KERN_DEBUG "ictp:  init_module called\n");
printk(KERN_DEBUG "ictp:  base address %x\n",base);
#endif

/*
   initialize the chip
*/
        ictp_reset();

        testvalue = inb(ICTP_B);
/*
   set bit 5 of port C and read back. This bit is unused
*/

        outb(ICTP_SET_TSTBIT,ICTP_S);
        testvalue = inb(ICTP_C);
#ifdef ICTP_DEBUG
      printk(KERN_DEBUG "ictp: port C after set bit 5 %x\n",testvalue);
#endif
        if ((testvalue & ICTP_TSTBIT) == 0) {
          printk(KERN_ERR "ictp: board not found!\n");
          return -ENODEV;
        }
        outb(ICTP_RESET_TSTBIT,ICTP_S);
        testvalue = inb(ICTP_C);
#ifdef ICTP_DEBUG
      printk(KERN_DEBUG "ictp: port C after reset bit 5 %x\n",testvalue);
#endif
```

```
        if ((testvalue & ICTP_TSTBIT) != 0) {
          printk(KERN_ERR "ictp: board not found!\n");
          return -ENODEV;
        }

/*
  register the device driver with the system
*/
if (register_chrdev(HW_MAJOR, "ictp", &ictp_fops)) {
  printk(KERN_ERR "register_chrdev failed: goodbye world :-(\n");
  return -EIO;
}
#ifdef ICTP_DEBUG
else
  printk(KERN_DEBUG "ictp: driver registered!\n");
#endif
        return 0;
}
```

The first version of the driver registered a fops table with NULL entries only. This version clearly cannot do anything, however it should be possible to test installation and de-installation into the system using *insmod* and *rmmod*. I expected to find the *printk* output on the xconsole and lsmod should allow to check proper installation. What did I find? Well, following Murphies laws, it was the worst possible result: lsmod told me:

```
ictp                    1           0
Module                Pages     Used by
```

but I had no trace whatsoever of my *printk* statements. I was not really sure who was right: *lsmod* or the missing output from *printk*. After several hours of research and some poking around on the internet I found the email address of a *guru* who had written a device driver before. He told me I could check where to find the system console by trying:

```
date > /dev/console
```

When I tried this, I found the output of the date on the xconsole as expected. Still I did not know, where the *printk* messages had gone. The other other test I could find was to recompile the kernel and link my driver into the system. When booting the newly created system I saw the very first *printk* output on the system console but not the following ones.Of course I then checked the system log (/var/log/messages) in order to find out what had

happened and there I found all the *printk* output I had expected on the console. The *printk* output went into the system log! After having add the lines

```
# Send debug messages to the system console
kern.debug                                          /dev/console
```

to /etc/syslog.conf I finally got the debugging messages where I wanted them, namely on the xconsole.

7 The Driver Routines

Since we now

- understand the hardware
- know how to install the device driver into the system
- have a frame of the driver ready
- are able to produce debugging messages

we can actually start to implement the first driver routines that do the real work. The first routines to be implemented are of course the *open* and *close* calls. When opening the device driver we initialize the 8255 chip writing the necessary control code into its command register.

In order to make sure that only one process at a time can access the device a busy flag is set after the first successful open. All subsequent open calls will be refused (giving back the error EBUSY) until the driver is closed again.

```
static int
ictp_open(struct inode * inode, struct file * file)
{
unsigned int      minor = MINOR(inode->i_rdev);
        unsigned char     command;
        int               ret_code;

if (ictp_busy ==  ICTP_BUSY)
return -EBUSY;

        command = ICTP_MODE_SELECT | ICTP_A_MODE_0 | ICTP_B_MODE_0
                                   | ICTP_INPUT_B;
```

```
        outb(command,ICTP_S);              /* setup to non interrupt */

#ifdef ICTP_DEBUG
          printk(KERN_DEBUG "ictp: opened for switch reading\n");
#endif
        ictp_busy         = ICTP_BUSY;
   return 0;
}
```

The above code is a simplified version of the code actually in service for the ictp driver. In the real *open* routine we also switch off the buzzer and, depending on the minor mode (ictp0, ictp1, ictp2) we register interrupt service routines with the system.

In order to implement the write part of the driver we should first have a look at the library routines accessible to the device driver writer. Some of these routines we have already seen before, namely:

- `register_chrdev(unsigned int major,`
  `const char *name,`
  `struct file_operations *fops)`

- `unregister_chrdev(unsigned int major,`
  `const char *name)`

- `printk(fmt)`

There are also 2 Macros that allow us to find out the current major am minor numbers:

- `MAJOR(inode -> i_rdev)` and

- `MINOR(inode -> i_rdev)`

As we have seen in the example code above, inode is a structure that is passed into the driver routines, In order to implement the read and write routines we need additional calls to transfer a data buffer from user space to supervisory space and back. This feature is provided by:

- get_user(char *address)

- put_user(char *address)

Their use is demonstrated by the (incomplete) ictp write routine:

```
/*
 * Write requests on the ictp device.
 */
static int
ictp_write(struct inode * inode, struct file * file,
   const char * buf, int count)
{

        char            c;
const char    *temp=buf;
unsigned char ctemp, digit;

        switch (ictp_write_mode) {
        case ICTP_MODE_RAW:
             temp = buf;
     while (count > 0) {
       c = get_user(temp);
       outb(c,ICTP_A);
       count--;
       temp++;
     }

     return temp-buf;
     break;
```

We have now seen the *open, close, write* routines (the *read* is very similar to the *write* once we replace get_user by put_user). The only missing code is *ioctl.*

As a typical example we will have a look at the code that drives the buzzer. The buzzer is connected to the PC-4 line of the 8255 and can be programmed by specifying the bit number in the bits 1-3 of the 8255 command register with biy 7 set to 0 and bit 0 defining on (bit=1) or off (bit=0). The ioctl call as seen from the driver user's point of view has got 3 parameters:

- the file descriptor
- a command code
- and a parameter

Command codes and symbolic names for possible parameters are described in the ictp.h file

```
#define ICTP_SET_BUZZER IOC_OUT | 0x 0004
```

```
ioctl(ictp\_fd,ICTP\_SET\_BUZZER,ICTP\_NOISE)
ioctl(ictp\_fd,ICTP\_SET\_BUZZER,ICTP\_SILENCE)
```

will switch the buzzer on and off.

The command parameter in the driver code receives the ioctl command code while the arg is passed the corresponding argument (buzzer on or off).

```
/*
 * Handle ioctl calls
 */

static int
ictp_ioctl(struct inode * inode, struct file * file,
   unsigned int cmd, unsigned long arg)
{
   unsigned int minor = MINOR(inode->i_rdev);
   unsigned char port_C_status;
   unsigned short dummy;

   case ICTP_SET_BUZZER:
#ifdef ICTP_DEBUG
     printk(KERN_DEBUG "ictp: ioctl set buzzer function entered!\n");
#endif
     if (arg == ICTP_BUZZER_ON) {
       outb(ICTP_NOISE,ICTP_S);
       return 0;
     }
     else if (arg == ICTP_BUZZER_OFF) {
       outb(ICTP_SILENCE,ICTP_S);
       return 0;
     }
     else
       return -EINVAL;
     break;

   case ICTP_GET_BUZZER:
#ifdef ICTP_DEBUG
     printk(KERN_DEBUG "ictp: ioctl get buzzer function entered!\n");
#endif
     port_C_status = inb(ICTP_C);
     if (port_C_status & ICTP_BUZZER_BIT)
       return ICTP_BUZZER_OFF;
     else
       return ICTP_BUZZER_ON;
```

```
      break;

   default: return -EINVAL;
      }
}
```

The driver allows users to choose a write mode forcing subsequent write calls to be interpreted in a different way:

- **ICTP_MODE_RAW** will send the data transfered in the write data buffer as is to the hardware. In this mode the driver user is responsable to set up the data and chip selects correctly. This mode has been provided in order give you trouble... well, in order to teach you the hardware.
- **ICTP_MODE_SINGLE_DIGIT** will write a single digit. Here a single byte containing the digit number (0-3) in the high nibble and the data value in the low nibble must be given in the databuffer.
- **ICTP_MODE_FULL_NUMBER** takes a short containing the number to be displayed on all for digits. This is of course the simplest way to use the driver. (Remember to specify a size of 2 (bytes) in the write call).

The full ioctl code not only permits the user to set the write mode using the ICTP_SET_WRITE_MODE ioctl command, several other commands are implemented for:

- Enabling/Disabling interrupts
- Setting the interrupt type (read becomes blocking or non blocking)
- Reading/Writing the buzzer state
- Reading/Writing the write mode

8 Appendix A: The ICTP device driver user's manual

A sample device driver for the ICTP board has been developed. The following gives a summary of its functions.

The ictp driver expects an I/O board using an Intel 8255 chip at an I/O adress that can be specified at module load time giving the option "base=0x320" to insmod (default 0x300). The connections to the ICTP board must be made as follows:

- Port A: ICTP displays
  Port A is therefore programmed as **output** port.
- Port B: ICTP switches
  Port B is therefore programmed as **input** port.
- If you open minor device 0, the port B of the 8255 will be set to mode 0 (non strobed input, allowing to read directly the state of the switches. Port A is set to mode 1 (strobed I/O).
- When opening minor device 1 the 8255 chip is initialized such that both, port A and port B are set to mode 1 (strobed I/O). This allows interrupts for both ports.
- In mode 1, with port A output, the bits 4 and 5 of port C may be used as normal I/O pins, while the other bits are used as handshake signals or interrupt lines.
  Bit 4 of port C must be connected to CA2 (the ICTP buzzer).
- Bit 2 and Bit 6 of port C are strobe lines which must be connected to one of the interrupt generating line CA1,CA2 or CB1.

The driver functions:

Read calls:

The driver uses major number 31 and 3 minor numbers:

- read on minor number 0: read the switches
  read on minor number 1: returns the number of interrupts arrived since the last read call.
- read on minor number 2: same as above for interrupt 2

Reads for interrupts exist in 2 flavors:

- **non blocking**: The number of interrupts since the last read is immediately returned, even if it is zero.

- **blocking**: If the number of interrupts is zero, it blocks the calling process until the next interrupt (or other signal like `^C`) arrives.

Write calls:

Writing works on any of the three minor devices. There are 3 different write modes which may be set up by *ioctl* calls (see later).

- **ICTP_MODE_RAW**: in this mode the data coming from the user are sent untreated to the I/O port. In order to make the displays work correctly, the user must select the suitable data/chipselect sequences cs high + data, cs low + data, cs high + data for all digits. In each byte the high nibble specifies the data and the low nibble the chip selects. 12 data bytes are expected and the driver will return **EINVAL** if the count is wrong

- **ICTP_MODE_SINGLE_DIGIT:** a single data byte is accepted. The high nibble contains the digit number (0-3) and the low nibble contains the data.

- **ICTP_MODE_FULL_NUMBER**: a short is expected. This number will be put onto the digits.

ioctl calls:

• ICTP_SET_WRITE_MODE:	sets up the writing mode. The following values are accepted:
— ICTP_MODE_RAW	12 data bytes expected but allowed anything
— ICTP_MODE_SINGLE_DIGIT	only 1 data byte allowed
— ICTP_MODE_FULL_NUMBER	a short needed;
• ICTP_SET_READ_MODE	set the read mode
— ICTP_MODE_BLOCKING	if count = zero, block process until interrupt arrives
— ICTP_MODE_NON_BLOCKING	return current count immediately
• ICTP_GET_WRITE_MODE	return the current write mode
• ICTP_GET_READ_MODE	return the current read mode
• ICTP_SET_BUZZER	controls the buzzer. Valid args are:
— ICTP_BUZZER_ON	
— ICTP_BUZZER_OFF	guess, what they are doing!
• ICTP_GET_BUZZER	read the current buzzer state.

9 Appendix B: The full Driver Code

```
/*
 * Implements the ICTP character device driver.
 * Create the device with:
 *
 * mknod /dev/ictp c 31 0
 *
 * - U. Raich
 * 13.3.94 : First version working with PC parallel printer port
 *
 * Modifications:
 * 30.8.94 : U.R. complete rewrite for Manuel's board
 */


/* Kernel includes */

#include <linux/module.h>

#include <linux/mm.h>
#include <linux/errno.h>
#include <linux/kernel.h>
#include <linux/major.h>
#include <linux/sched.h>
#include <linux/malloc.h>
#include <linux/ioport.h>
#include <linux/fcntl.h>
#include <linux/delay.h>

#include <asm/io.h>
#include <asm/segment.h>
#include <asm/system.h>

#include "ictp.h"

#define HW_MAJOR    31 /* nice and high */
#define ICTP_DEBUG   1

/*
   some globals:
*/
int                base=0x300;
```

```
unsigned long     ictp_write_mode  = ICTP_MODE_RAW;
unsigned long     ictp_read_mode   = ICTP_MODE_NON_BLOCKING;
int               ictp_busy        = ICTP_FREE;
int               irq5 = 5, irq7 = 7;
unsigned char     irq5_count = 0, irq7_count = 0;
struct wait_queue *ictp_wait_q;

/*
 * The driver.
 */

static void
out_digit(unsigned char digit, unsigned char number)

{
  unsigned char mask,c;

  mask = 1 << digit;
  mask = ~mask;
#ifdef ICTP_DEBUG
  printk("ictp: mask %x\n",mask);
#endif
  c = (number << 4) | 0xf;
  outb(c,ICTP_A);
  c &= mask;
  outb(c,ICTP_A);
  c |= 0xf;
  outb(c,ICTP_A);
}


/*
  first the tough part: the interrupt code
*/

static void
ictp_irq7_interrupt( int irq)
{

  outb(ICTP_DUMMY,ICTP_A);          /* this just clears the interrupt */

  irq7_count++;
  if (ictp_read_mode == ICTP_MODE_BLOCKING)
```

```
    wake_up(&ictp_wait_q);
}

static void
ictp_irq5_interrupt(int irq)
{
  unsigned char dummy;
  dummy = inb(ICTP_B);          /* this just clears the interrupt */

  if (ictp_read_mode == ICTP_MODE_BLOCKING)
    wake_up(&ictp_wait_q);
  irq5_count++;

}

static void
ictp_reset(void)
/*=======================*/

/*
  initializes the 8255 chip
*/

{
unsigned char command;

/*
  sets port A to output
       port A is connected to the ICTP module displays
       high order nibble: data
       low  order nibble: chip selects
  mode 1: stobed I/O
       allows use of interrupts
       CA1: on interrups
       CA2: (Buzzer) on PC4

*/
        command = ICTP_MODE_SELECT | ICTP_A_MODE_0 | ICTP_B_MODE_0
                                   | ICTP_INPUT_B;

outb(command, ICTP_S);
/*
  kill the buzzer
```

```
  first setup port C to bit set (bit set/reset mode with set bit on! )
*/
        outb(ICTP_SILENCE,ICTP_S);
return ;
}

/*
 * Handle ioctl calls
 */

static int
ictp_ioctl(struct inode * inode, struct file * file,
   unsigned int cmd, unsigned long arg)
{
   unsigned int minor = MINOR(inode->i_rdev);
   unsigned char port_C_status;
   unsigned short dummy;
   switch (cmd) {
   case ICTP_SET_WRITE_MODE:
#ifdef ICTP_DEBUG
    printk(KERN_DEBUG
           "ictp: ioctl write function entered! cmd: %x, arg: %lx\n",
   cmd, arg);
#endif
    if (arg > ICTP_MODE_FULL_NUMBER)
 return -EINVAL;
     else {
         ictp_write_mode = arg;
 return 0;
       }
     break;
   case ICTP_GET_WRITE_MODE:
#ifdef ICTP_DEBUG
     printk(KERN_DEBUG "ictp: ioctl read function entered! cmd: %x\n",cmd);
#endif
     return ictp_write_mode;
     break;

   case ICTP_SET_READ_MODE:
#ifdef ICTP_DEBUG
    printk(KERN_DEBUG
           "ictp: ioctl write function entered! cmd: %x, arg: %lx\n",
   cmd, arg);
```

```
#endif
    if (arg > ICTP_MODE_NON_BLOCKING)
 return -EINVAL;
     else {
         ictp_read_mode = arg;
 return 0;
       }
     break;
   case ICTP_GET_READ_MODE:
#ifdef ICTP_DEBUG
     printk(KERN_DEBUG "ictp: ioctl read function entered! cmd: %x\n",cmd);
#endif
     return ictp_read_mode;
     break;

   case ICTP_SET_BUZZER:
#ifdef ICTP_DEBUG
     printk(KERN_DEBUG "ictp: ioctl set buzzer function entered!\n");
#endif
     if (arg == ICTP_BUZZER_ON) {
       outb(ICTP_NOISE,ICTP_S);
       return 0;
     }
     else if (arg == ICTP_BUZZER_OFF) {
       outb(ICTP_SILENCE,ICTP_S);
       return 0;
     }
     else
       return -EINVAL;
     break;

    case ICTP_GET_BUZZER:
 #ifdef ICTP_DEBUG
     printk(KERN_DEBUG "ictp: ioctl get buzzer function entered!\n");
 #endif
     port_C_status = inb(ICTP_C);
     if (port_C_status & ICTP_BUZZER_BIT)
       return ICTP_BUZZER_OFF;
     else
       return ICTP_BUZZER_ON;
     break;

    case ICTP_ENABLE_INTERRUPT:
```

```
#ifdef ICTP_DEBUG
     printk(KERN_DEBUG "ictp: enabling interrupts on 8255\n");
#endif
     if (minor == ICTP_READ_IRQ7_COUNT) {
       dummy = 0xff;
       outb(dummy,ICTP_A);              /* reset int flag */
       outb(ICTP_ENABLE_IRQ7,ICTP_S);

       irq7_count = 0;
       dummy = inb(ICTP_C);
#ifdef ICTP_DEBUG
       printk(KERN_DEBUG "ictp: port C after enable int 7: %x\n",dummy);
#endif
       return 0;
     }
     else if (minor == ICTP_READ_IRQ5_COUNT) {
       outb(ICTP_ENABLE_IRQ5,ICTP_S);
       dummy = inb(ICTP_B);                /* now interrupts should come in */

       irq5_count = 0;
       dummy = inb(ICTP_C);
#ifdef ICTP_DEBUG
       printk(KERN_DEBUG "ictp: port C after enable int 5: %x\n",dummy);
#endif
       return 0;
     }
     else
       return -EINVAL;
     break;

   case ICTP_DISABLE_INTERRUPT:
#ifdef ICTP_DEBUG
     printk(KERN_DEBUG "ictp: disabling interrupts on 8255\n");
#endif
     if (minor == ICTP_READ_IRQ7_COUNT) {
       outb(ICTP_ENABLE_IRQ7,ICTP_S);
       return 0;
     }
     else if (minor == ICTP_READ_IRQ5_COUNT) {
       outb(ICTP_ENABLE_IRQ5,ICTP_S);
       return 0;
     }
     else
```

```
      return -EINVAL;
    break;

  default: return -EINVAL;
      }
}
/*
 * Read the status of the ICTP board switches
 */

static int
ictp_read(struct inode * inode, struct file * file,
  char * buf, int count)
{

unsigned int  minor = MINOR(inode->i_rdev);
        unsigned char testvalue;

        if (count != 1) return -EINVAL;

        switch (minor) {
case ICTP_READ_SWITCHES:
  testvalue = inb(ICTP_B);        /* read the switches */
#ifdef ICTP_DEBUG
          printk(KERN_DEBUG
 "ictp: switch value read from port: %x\n",testvalue);
#endif
          put_user(testvalue,buf);
  return 1;
          break;
case ICTP_READ_IRQ7_COUNT:
#ifdef ICTP_DEBUG
          printk(KERN_DEBUG "ictp: irq7_count: %d\n",irq7_count);
#endif
          if (ictp_read_mode == ICTP_MODE_BLOCKING){
            if (irq7_count == 0) {
#ifdef ICTP_DEBUG
      printk(KERN_DEBUG "ictp: Going to sleep ...\n");
#endif
              interruptible_sleep_on(&ictp_wait_q);
    }
#ifdef ICTP_DEBUG
            printk(KERN_DEBUG "ictp: returned from sleep\n");
```

```
#endif
            put_user(irq7_count,buf);
            irq7_count = 0;
          }
          else {
            put_user(irq7_count,buf);
            irq7_count = 0;
          }
          testvalue = inb(ICTP_C);
#ifdef ICTP_DEBUG
          printk(KERN_DEBUG "ictp: Port C data: %x\n",testvalue);
#endif
          return 1;
case ICTP_READ_IRQ5_COUNT:
#ifdef ICTP_DEBUG
          printk(KERN_DEBUG "ictp: irq5_count: %d\n",irq5_count);
#endif
          if (ictp_read_mode == ICTP_MODE_BLOCKING){
            if (irq5_count == 0) {
#ifdef ICTP_DEBUG
      printk(KERN_DEBUG "ictp: Going to sleep ...\n");
#endif
              interruptible_sleep_on(&ictp_wait_q);
    }
#ifdef ICTP_DEBUG
            printk(KERN_DEBUG "ictp: returned from sleep\n");
#endif
            put_user(irq5_count,buf);
            irq5_count = 0;
          }
          else {
            put_user(irq5_count,buf);
            irq5_count = 0;
          }
          testvalue = inb(ICTP_C);
#ifdef ICTP_DEBUG
          printk(KERN_DEBUG "ictp: Port C data: %x\n",testvalue);
#endif
          return 1;
default:
  return -EINVAL;
}
}
```

```
/*
 * Write requests on the ictp device.
 */
static int
ictp_write(struct inode * inode, struct file * file,
   const char * buf, int count)
{

        char          c;
const char    *temp=buf;
unsigned char ctemp, digit;

        switch (ictp_write_mode) {
        case ICTP_MODE_RAW:
             temp = buf;
     while (count > 0) {
       c = get_user(temp);
       outb(c,ICTP_A);
       count--;
       temp++;
     }

     return temp-buf;
     break;
        case ICTP_MODE_SINGLE_DIGIT:
                     if (count != 1)
                       return -EINVAL;
                     c = get_user(temp);
                     digit = c >> 4;
                     ctemp = c & 0xf;
                     out_digit(digit,ctemp);
                     return 1;
     break;
case ICTP_MODE_FULL_NUMBER:
                     if (count != 2)
                       return -EINVAL;
                     temp = buf;
                     c = get_user(temp);
#ifdef ICTP_DEBUG
                     printk(KERN_DEBUG "write, mode 2, first byte: %x\n",c);
#endif
                     ctemp = c & 0xf;
```

```
                        out_digit(0,ctemp);
                        ctemp = c >> 4;
                        out_digit(1,ctemp);

                        c = get_user(temp+1);
#ifdef ICTP_DEBUG
                        printk("write, mode 2, second byte: %x\n",c);
#endif
                        ctemp = c & 0xf;
                        out_digit(2,ctemp);
                        ctemp = c >> 4;
                        out_digit(3,ctemp);
/*
  get first nibble
*/

     return 2;
     break;
default:
     return 1;
                        break;
   }

}


static int
ictp_open(struct inode * inode, struct file * file)
{
unsigned int      minor = MINOR(inode->i_rdev);
        unsigned char     command;
        int               ret_code;

if (minor >= ICTP_NO)
return -ENODEV;
if (ictp_busy ==  ICTP_BUSY)
return -EBUSY;

        ictp_write_mode  = ICTP_MODE_RAW;

        switch (minor) {
/*
  this allows interrupts on the push button
```

```
*/
        case ICTP_READ_IRQ7_COUNT:
          ret_code = request_irq(irq7,(void *)ictp_irq7_interrupt,
                                 SA_INTERRUPT,"ictp",NULL);
          if (ret_code) {
              printk(KERN_WARNING "ictp: unable to use interupt 7\n");
            return ret_code;
  }
          else {
#ifdef ICTP_DEBUG
            printk(KERN_DEBUG "ictp: irq7 registered\n");
#endif
            command = ICTP_MODE_SELECT | ICTP_A_MODE_1 | ICTP_B_MODE_0
                                       | ICTP_INPUT_B;
            outb(command,ICTP_S);           /* strobed output */
/*
  kill the buzzer
  first setup port C to bit set (bit set/reset mode with set bit on! )
*/
            outb(ICTP_SILENCE,ICTP_S);
  }
  break;

case ICTP_READ_IRQ5_COUNT:

          ret_code = request_irq(irq5,(void *)ictp_irq5_interrupt,
                                 SA_INTERRUPT,"ictp",NULL);
          if (ret_code) {
              printk(KERN_WARNING "ictp: unable to use interupt 5\n");
              return ret_code;
  }
  else {
          command = ICTP_MODE_SELECT | ICTP_A_MODE_0 | ICTP_B_MODE_1
                                     | ICTP_INPUT_B;
          outb(command,ICTP_S);           /* strobed input */
/*
  kill the buzzer
  first setup port C to bit set (bit set/reset mode with set bit on! )
*/
            outb(ICTP_SILENCE,ICTP_S);
#ifdef ICTP_DEBUG
            printk(KERN_DEBUG "ictp: interrupt 5 registered\n");
#endif
```

```
    }
  break;

case ICTP_READ_SWITCHES:
          command = ICTP_MODE_SELECT | ICTP_A_MODE_0 | ICTP_B_MODE_0
                                    | ICTP_INPUT_B;
          outb(command,ICTP_S);           /* setup to non interrupt */
/*
  kill the buzzer
  first setup port C to bit set (bit set/reset mode with set bit on! )
*/
          outb(ICTP_SILENCE,ICTP_S);
#ifdef ICTP_DEBUG
          printk(KERN_DEBUG "ictp: opened for switch reading\n");
#endif
          break;

default:
  return -EINVAL;
}
        ictp_busy        = ICTP_BUSY;
MOD_INC_USE_COUNT;
   return 0;
}

static void
ictp_release(struct inode * inode, struct file * file)
{
unsigned int minor = MINOR(inode->i_rdev);

/*
  free the interrupt
*/
        switch (minor) {
case ICTP_READ_IRQ7_COUNT:
  free_irq(irq7,NULL);
#ifdef ICTP_DEBUG
          printk(KERN_DEBUG "ictp: interrupt 7 free'd\n");
#endif
  break;
case ICTP_READ_IRQ5_COUNT:
  free_irq(irq5,NULL);
#ifdef ICTP_DEBUG
```

```
        printk(KERN_DEBUG "ictp: interrupt 5 free'd\n");
#endif
  break;
       default: ;
       }
ictp_busy  = ICTP_FREE;
MOD_DEC_USE_COUNT;
}

static struct file_operations ictp_fops = {
    NULL,               /* seek       */
    ictp_read,
    ictp_write,
    NULL,/* readdir */
    NULL,/* select  */
    ictp_ioctl,
    NULL,/* mmap */
    ictp_open,
    ictp_release
};

/*
 * And now the modules code and kernel interface.
 */

int
init_module( void) {

unsigned char testvalue = 0;

#ifdef ICTP_DEBUG
printk(KERN_DEBUG "ictp:  init_module called\n");
printk(KERN_DEBUG "ictp:  base address %x\n",base);
#endif

/*
   initialize the chip
*/
        ictp_reset();

        testvalue = inb(ICTP_B);
/*
   set bit 5 of port C and read back. This bit is unused
```

```
*/

        outb(ICTP_SET_TSTBIT,ICTP_S);
        testvalue = inb(ICTP_C);
#ifdef ICTP_DEBUG
        printk(KERN_DEBUG "ictp: port C after set bit 5 %x\n",testvalue);
#endif
        if ((testvalue & ICTP_TSTBIT) == 0) {
          printk(KERN_ERR "ictp: board not found!\n");
          return -ENODEV;
        }
        outb(ICTP_RESET_TSTBIT,ICTP_S);
        testvalue = inb(ICTP_C);
#ifdef ICTP_DEBUG
        printk(KERN_DEBUG "ictp: port C after reset bit 5 %x\n",testvalue);
#endif
        if ((testvalue & ICTP_TSTBIT) != 0) {
          printk(KERN_ERR "ictp: board not found!\n");
          return -ENODEV;
        }

/*
  register the device driver with the system
*/
if (register_chrdev(HW_MAJOR, "ictp", &ictp_fops)) {
  printk(KERN_ERR "register_chrdev failed: goodbye world :-(\n");
  return -EIO;
}
#ifdef ICTP_DEBUG
else
  printk(KERN_DEBUG "ictp: driver registered!\n");
#endif
        return 0;
}

void
cleanup_module( void) {

#ifdef ICTP_DEBUG
  printk(KERN_DEBUG "ictp:  cleanup_module called\n");
#endif
  if (ictp_busy)
    printk(KERN_WARNING "ictp: device busy, remove delayed\n");
```

```
  if (unregister_chrdev(HW_MAJOR, "ictp") != 0) {
    printk("cleanup_module failed\n");
  }
#ifdef ICTP_DEBUG
  else
    printk(KERN_DEBUG "cleanup_module succeeded\n");
#endif
}
```

Towards Real Time Data Communications

Sixth College on Microprocessor-based Real-time Systems in Physics

Abdus Salam ICTP, Trieste. October 9–November 3, 2000

Abhaya S Induruwa
Technical Advisory Unit
University of Kent
Canterbury CT2 7NF
England.

email: abhaya@cse.mrt.ac.lk
asi1@ukc.ac.uk

Abstract

These lectures are intended to help understand the computer network architecture comprising network protocols, standards, hardware and supporting technologies needed to perform real time data transfer. The IP architecture and its components used in real time data communication are discussed. IP/TV is illustrated as a real life example.

Also discussed are High Speed LANs which, along with ATM, play a vital role in delivering real time data to the desk top. In this respect the new variants of Ethernet are playing an increasingly important role.

More recent attempts based on WAP to deliver Internet services to handheld mobile devices and on ADSL to deliver Internet to the home user are introduced.

1 Introduction

Computer networks are increasingly becoming an integral and indispensable part of scientific as well as public life. Over the last couple of decades data networks have changed their character from a slow speed point to point connection to a high speed data communication backbone supporting full multimedia information transfer. More recently the handheld wireless phone based on **WAP** (Wireless Application Protocol) and **ADSL** (Asynchronous Digital Subscriber Line) technology supporting high speed data transfer rates over twisted pair subscriber lines have revolutionised the provision of public access to the Internet and its information services.

Today the technology offers the possibility of merging Real Time applications such as voice and data acquisition services which are time sensitive, with time insensitive non Real Time services on a single network infrastructure. The largest Real Time Network in the world (also the oldest!) is the telephone network which provides only 4 kHz analog bandwidth per voice channel. The newer architectures such as **ISDN** (Integrated Services Digital Network) and Broadband ISDN offer channel bandwidths of 64 Kbps and above. These higher bandwidths are suitable to carry either a number of basic voice channels or a single application requiring a larger bandwidth. **ATM** Asynchronous Transfer Mode, a cell based transport technique has been developed to support the **B-ISDN** services.

RTP (Real time Transport Protocol), together with **RTCP** (Real time Transport Control Protocol), have been devised to facilitate the communication of Real Time data over computer networks, which have been designed and built to guarantee the delivery of time insensitive bursty data. **IPv6** is emerging as the next generation IP with all the features necessary to support the delivery of multimedia and other real time data services at high speed.

At the Local Area Network level, the popular **Ethernet** running at 10Mbps is enhanced to operate at 100Mbps (**Fast Ethernet**) and 1000Mbps (**Gigabit Ethernet**) thus making it suitable as a delivery mechanism of Real Time traffic to the desktop.

2 Network Classification

A computer network is a collection of computers interconnected by one or more transmission paths for the purpose of transfer and exchange of data between the computers. Today these networks span the entire globe and belong to many different nations and network operators. Such networks can be classified in many ways depending on the switching mechanism, transmission

speed, etc [Black 93], [Stallings 94a], [Stallings 94b].

For the purpose of this discussion it is appropriate to classify them based on their:

a. geographical coverage

b. network topology.

2.1 Geographical Coverage

Networks can be classified into 4 categories depending on their geographical coverage (from the smallest to the largest) as follows:

1. Desktop Area Networks (DANs) [1]
2. Local Area Networks (LANs)
3. Metropolitan Area Networks (MANs)
4. Wide Area Networks (WANs).

There is a significant level of deployment of all or some of the above even in developing countries (most notably LANs and WANs with DANs just appearing) and hence should be of interest to research scientists.

2.2 Network Topology

Networks can be classified according to their topology in the following manner [Stallings 93], [Stallings 94b].

1. Bus Topology (eg. CSMA/CD; Ethernet)
2. Ring Topology (eg. Token Ring, FDDI)
3. Star Topology (eg. ArcNet, Switched networks such as Fast Ethernet)
4. Mesh Topology (eg. Telephone network).

The bus topology has been used initially in the Ethernet (broadcast) and later in Token Bus. Today it is important in the DQDB (Distributed Queue Dual Bus).

The token passing mechanism shown in Figure 1 has originally been used in the Token Ring and later in FDDI.

If however networks are categorised according to their transfer mechanism, then the following classification results.

[1] a recent classification arising out of delivering ATM to the desktop at 25 Mbps. DANs are used to interconnect devices such as camera, telephone and workstations.

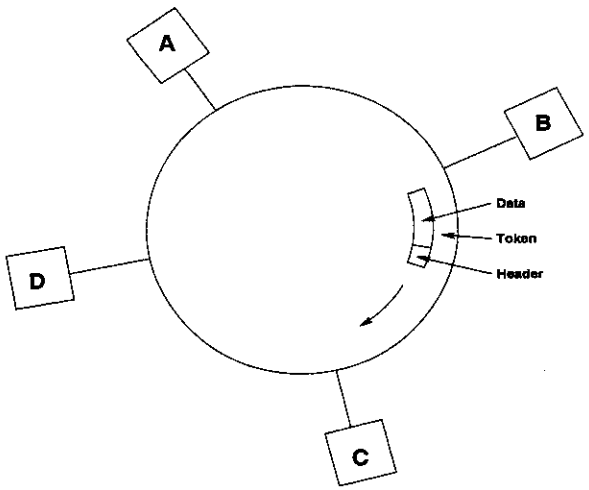


Figure 1: Token Passing Ring

1. Broadcast Networks

   Figure 2 shows a broadcast network which is used to broadcast from 1 to many. CSMA/CD (Carrier Sense Multiple Access/Collision Detect) is a medium access protocol which is used to broadcast on a bus topology.

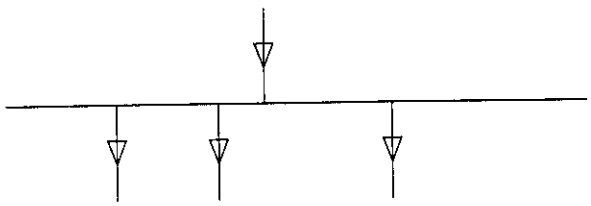


Figure 2: Broadcast Network

2. Switched Networks

   In Figure 3 is shown a switched network which is used to switch data from 1 to 1, 1 to many, or many to many. The telephone network is an example of a switched network. It can be used to support applications such as tele conferencing which involves the switching of many to many.

   Although initially switched networks were mainly used in telecommunication networks, today because of its superior performance, switched technologies are used in LANs (for example switched Ethernet) and ATM networks.

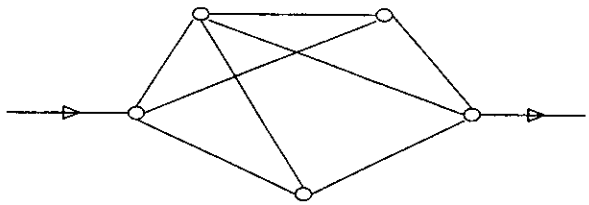


Figure 3: Switched Network

3. Hybrid Networks

   Figure 4 shows a hybrid network which consists of a switched part and a broadcast part.

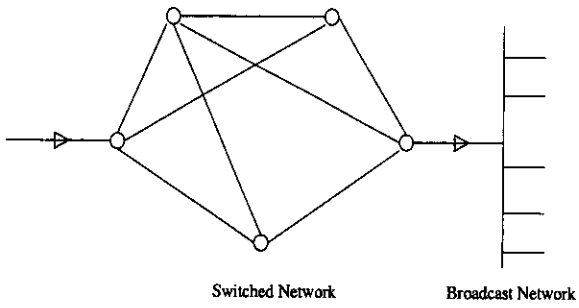


Figure 4: Hybrid Network

3 Network Architecture

The topology, transmission mechanism, and a protocol which manages the transmission mechanism together define a network architecture.

3.1 What is a Network Protocol?

A network protocol is used to facilitate the transmission of data between a sender (transmitter) and a recipient (receiver) across a data communication network in an agreed manner. Over the years several different network protocol architectures have evolved, the most notable being:

i. ISO - OSI (Open Systems Interconnect) Reference Model (OSI-RM)

ii. IP (Internet Protocol)

iii. ISO 8802.X (for LANs and MANs - same as IEEE 802.X)

In addition to the above there are hundreds of vendor specific protocols such as:

a. IBM's SNA (Systems Network Architecture)
b. DEC's DNA (DEC Network Architecture),

which are proprietary and hence are not truly interoperable.

Because todays networks span the entire globe, it is important to utilise standard protocols to facilitate seamless data communication over the networks belonging to different network operators to ensure interoperability. This has been the major objective of the bodies involved in preparing standards, such as the International Standards Organisation (ISO), Internet Engineering Task Force (IETF), International Telecommunication Union (ITU), Institute of Electrical and Electronic Engineers (IEEE) [2], American National Standards Institute (ANSI) and the ATM Forum.

Figure 5 shows the layered architectures of the protocols developed by the above standards bodies.

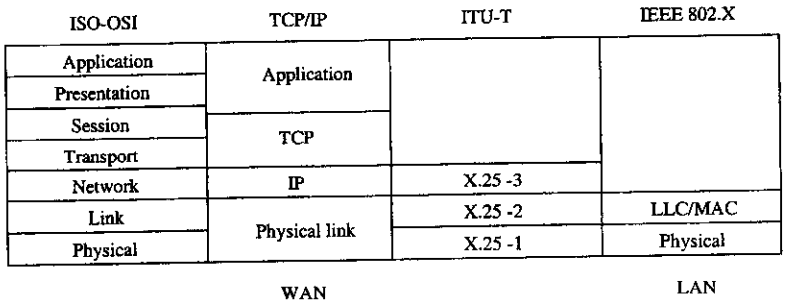


Figure 5: Layered Architectures of different Protocols

In Figure 6 is shown the architecture of the IEEE 802 family of standards for LANs and MANs. FDDI is a standard developed by ANSI-ASC X3T9 (Accredited Standards Committee) and provides services specified by the ISO Data link and Physical layers (ISO 9314).

The key features of FDDI are 100 Mbps data rate, use of optical fibre (*multi mode fibre, single mode fibre, low cost fibre*), the token ring style MAC protocol and the reconfiguration concept (automatic healing property in case of faults). The FDDI standard however allows the FDDI to be carried on other physical media such as twisted pair copper (CDDI). FDDI-II supports protocols for transmission of real time data [Jain 94].

IEEE 802.6 MAN standard specifies a DQDB (Distributed Queue Dual Bus) protocol which can support data, voice and video traffic. It can also

[2] http://grouper.ieee.org/groups/802/index.html

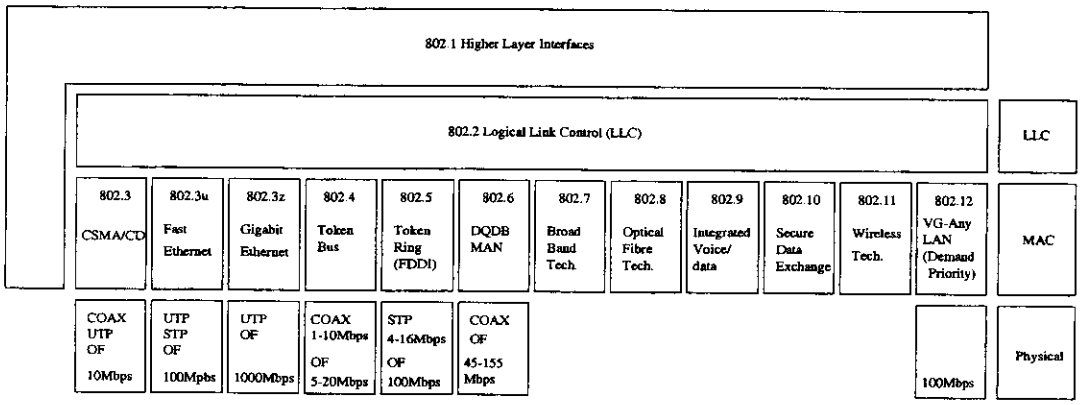


Figure 6: The Architecture of IEEE 802 LAN/MAN Standards

serve as a LAN. DQDB MAN operates on a shared medium with two unidirectional buses that flow in opposite directions. Two methods of gaining access to the medium depending on the type of traffic have been specified. In the first method a node on the DQDB subnetwork can queue to gain access to the medium by using a distributed queue or by requesting a fixed bandwidth through a prearbitrated access method. Data is transmitted in fixed size units called slots of length 53 bytes (52 bytes data + 1 byte access control field).

DQDB private networks are connected to the public network by point to point links. A DQDB MAN can typically range up to more than 50 km in diameter and can operate at a variety of speeds ranging from 34 Mbps to 150 Mbps.

DQDB has been conceived to integrate data and voice over a common set of equipment, thereby reducing maintenance and administrative costs. B-ISDN is an attempt to provide universal and seamless connectivity for multimedia services. IEEE 802.6 MAN has been designed to provide an interim solution and to act as a migration path to B-ISDN.

Nodes in a DQDB subnetwork are connected to a pair of buses flowing in opposite directions and can operate in one of two topologies, namely; *open bus* (Figure 7) or *looped bus* (Figure 8) (open bus topology is similar to Ethernet and looped bus topology is similar to token ring).

Although the DQDB access layer is independent of the physical medium, the speeds at which DQDB MANs operate demand the use of fibre or coaxial cables. For example, ANSI-DS3 operates at 44.736 Mbps over 75 Ω coax or fibre and ANSI SONET STS3 operates at 155.52 Mbps over single mode fibre. The ITU-T G.703 operates at 34.368 Mbps and 139.264 Mbps over a metallic medium.

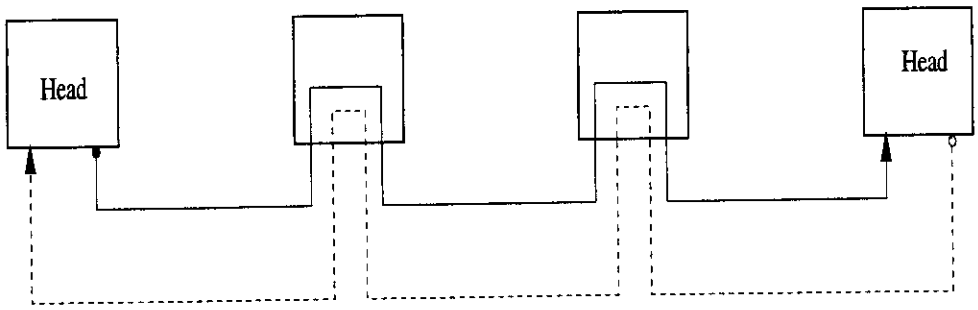


Figure 7: Open Bus DQDB Network

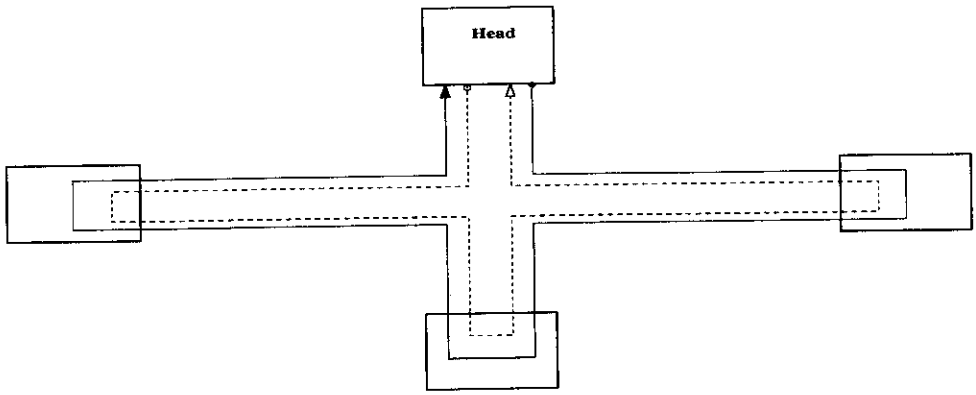


Figure 8: Looped Bus DQDB Network

A complete treatment of both the FDDI (Fibre Distributed Data Interface) and the DQDB (Distributed Queue Dual Bus) along with LANs and MANs is found in [Stallings 93], [Stallings 94b].

3.2 Transmission Mechanism

From the beginning, the voice data in a telephone network had been transmitted in real time using circuit switching techniques. Since circuit switching is not an efficient transmission mechanism for the communication of non-voice data, packet switching techniques have been devised.

3.2.1 Packet Switching

Transmission mechanisms based on packet switching allow the multiplexing of data packets from different sources on the same transmission path thereby making use of the channel bandwidth more efficiently. It also allows the transmission of packets from one source along different paths thus taking care of line congestion and availability problems. When the packets reach

the intended destination in whatever path they may have taken, they are reassembled and presented to the user application.

The most widely used protocols which manage the packet transfer across a data network belong to the family of standards conceived by the ITU-T (X.25) and Internet Architecture Board (IP). However **the variable packet lengths and the multiplexing technique introduce jitter making their Quality of Service (QOS) unacceptable for real time applications**. The inherent limitation of X.25 in high speed data transfer has been removed to some extent in the Frame Relay [Smith 93] transfer mechanism.

3.2.2 Frame Relay

Frame Relay offers a high speed version of packet switching and has the potential of operating effectively at much higher speeds compared to X.25, reaching speeds of 45 Mbps. Frame relay is well suited to high speed data applications, but not suited for delay sensitive applications such as voice and video because of the variable length of frames [Smith 93], [Stallings 95].

3.2.3 Cell Switching

Cell Relay is a transmission mechanism that combines the benefits of time division multiplexing with packet switching. It operates on the packet switching principle of statistically interleaving cells on a link, on an 'as required basis', rather than on a permanently allocated time slot basis. The fixed cell size used enables a reasonably deterministic delay to be achieved across a network. This deterministic nature of cell relay guarantees Quality Of Service (QOS) and hence makes it suitable for all traffic types including real time traffic, within a single network.

ATM (Asynchronous Transfer Mode) is fast becoming the dominant form of cell relay. It uses a cell of 53 bytes long (5 bytes header and 48 bytes data) and typically operates at speeds of 155 and 622 Mbps. ATM is delivered to the desktop at 25 Mbps and Gbps platforms are being tested [Partridge 94]. ITU-T has selected ATM as the transport technique for B-ISDN (Broadband Integrated Services Digital Network) [De Prycker 95], [Handel et al 94], [Stallings 95].

3.3 Physical Media

The most popularly used physical media are:

- Optical fibre cables
- Coaxial cables
- Unshielded Twisted Pair (UTP) cables
- Shielded Twisted Pair (STP) cables

Today all of the above media are used to carry data in excess of 100 Mbps speeds. Only the distances they cover are different (for example, optical fiber can operate at gigabit speeds for a few km whereas UTP can operate at 100 Mbps over a distance of 100 m without repeaters).

4 Internetworking

By internetworking is meant the process of interconnection of computers and their networks to form a single internet. In other words to make two or more networks logically to look and work like one. The Internet (note the uppercase 'I') is such an internet and uses TCP/IP (Transmission Control Protocol /Internet Protocol) protocol suite [Wilder 93].

TCP is connection oriented and provides a reliable stream transport. Although TCP is commonly associated with IP (as its underlying protocol), it is an independent, general purpose protocol that can be adapted to use with other delivery systems. The popularity of TCP has resulted in ISO – TP4, which has been derived from TCP. TCP, together with IP, provides a reliable stream delivery for data traffic.

TCP/IP protocol suite has become the *de facto* standard for open system interconnection in the computer industry. It is used world wide in academic, government, private and public institutions. Some of the reasons for its wide acceptance can be attributed to the following:

- It provides the highest degree of interoperability.
- It encompasses the widest set of vendors' systems.
- It runs over more network technologies than any other protocol suite.

Over the years Unix (and hence Linux) and TCP/IP have become almost synonymous. Today it has become part of the operating system kernel. The OS runs a separate process for IP, TCP input/output and UDP input/output.

In building internets, following hardware devices used to interconnect networks are of interest [Black 93], [Comer 88].

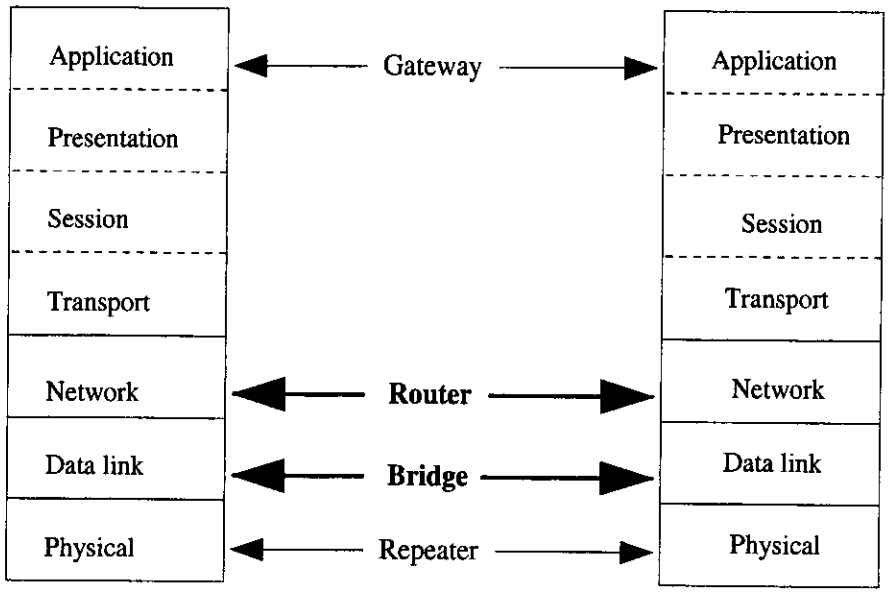


Figure 9: Interconnection Components

1. Repeaters
2. Bridges and Switches
3. Routers and Brouters
4. Gateways.

Figure 9 shows the use of these components in relation to the ISO-OSI Reference Model.

4.1 Repeaters

When two networks are to be connected at the lowest level, ie. the physical level, an interconnecting device known as a *repeater* is used. A repeater simply takes bits arriving from one network and just forwards (repeats) them on to the other. In some cases a repeater might have to translate between two different physical layer formats, for example from optical fiber to UTP (Unshielded Twisted Pair) cable. This may involve some processing of the received signal such as signal regeneration for noise elimination. However, repeaters pass on the data received without paying attention to the address information.

4.2 Bridges and Switches

Bridges and switches are used to interconnect networks at the medium access control (MAC) layer. Typically this requires the interconnected networks to have identical MAC layers although networks with different but related MAC layers can be interconnected. Since Bridges operate at the MAC layer, they can be used to effectively segment the traffic by filtering the traffic entering one segment from another thereby reducing the unwanted traffic flow on network segments.

Unlike a repeater which replicates electrical signals, bridges replicate packets. They are superior in their function, because they do not replicate noise and errors or malformed frames. Moreover, bridges implement CSMA/CD rules and hence collisions and propagation delays remain isolated without affecting the other segments. In other words, every port on an Ethernet bridge or a switch is on a separate *Collision Domain* or physical network. As a result an almost arbitrary number of Ethernet segments can be connected together with bridges whereas with repeaters the maximum number of segments is five giving a total length of 2.5 km. Since bridges hide details of interconnection, a set of bridged segments acts like a single Ethernet.

A bridge can be used to make a decision on which frames to forward from one segment to another. Such bridges are called *adaptive* or *learning* bridges. They learn over time, which hosts are connected to which segment. Thus an adaptive bridge builds up the address table automatically without human intervention.

Bridges are often used to improve the performance of an overloaded network by effectively partitioning the network into segments.

The type of bridges used in CSMA/CD LANs are known as *transparent bridges* (IEEE 802.1D) since their presence is not visible to the stations. The type of bridges used to interconnect token rings are called *source routing bridges* (IEEE 802.5) and the routing information is provided by the source station.

4.3 Routers

Routers interconnect networks at the network layer (level 3 in the OSI-RM and the IP layer of the TCP/IP suite) and perform routing functions. This is the main building block in internetworking using IP. Most routers now support at least one of the multicast routing protocols, which is an essential functionality to support the delivery of Real Time data over IP.

For performance reasons the functionality of a bridge and a router is

combined to form a *brouter*. Whenever possible a brouter bridges data for better efficiency rather than routing.

4.4 Gateways

Gateways are used when networks based on completely different network architectures have to be interconnected at layers higher than the network layer and when protocol translation is necessary. A typical example is interconnecting two networks based on TCP/IP suite and OSI suite of protocols. An application level gateway is required to support FTP on TCP/IP and FTAM on OSI. From this it should be clear that a different application level gateway is required for every application supported across the interconnected networks.

4.5 Multiport-Multiprotocol Devices

The multiplicity of transmission media and protocols used in today's networks require the use of multiport repeaters and bridges, as well as multiprotocol routers which support more than one protocol stack.

5 A word about the Internet

The one and only global network of interest to the whole scientific community of the world today is the **Internet**, which is based on the **Internet Protocol (IP)**.

Internet Protocol is a truly scalable protocol that is used to connect computers to small LANs and to WANs forming a global internetwork. IP is available for almost all computing platforms ranging from the smallest palmtop to the largest ultrasuper computer. Nowadays IP is increasingly becoming available in handhled mobile phones and personal data assistants (PDAs) and also on standalone micro-controller boards and embedded systems.

The Internet is an ever expanding network of networks. As of this writing (September 2000), it interconnects an estimated 100 million computers and 1,000,000 computer networks in 176 countries. It experiences a staggering growth rate of 100% per year. Figure 10 shows the Internet connectivity in the world in 1997. Today, as a result of the emergence of various transmission technologies including optical fibre, satellite and enhanced communication technologies based on robust high speed modems and DSL (Digital Subscriber Line), almost all the countries in the world have some Internet connectivity. Although it is difficult to keep track of the growth of the Internet in real time

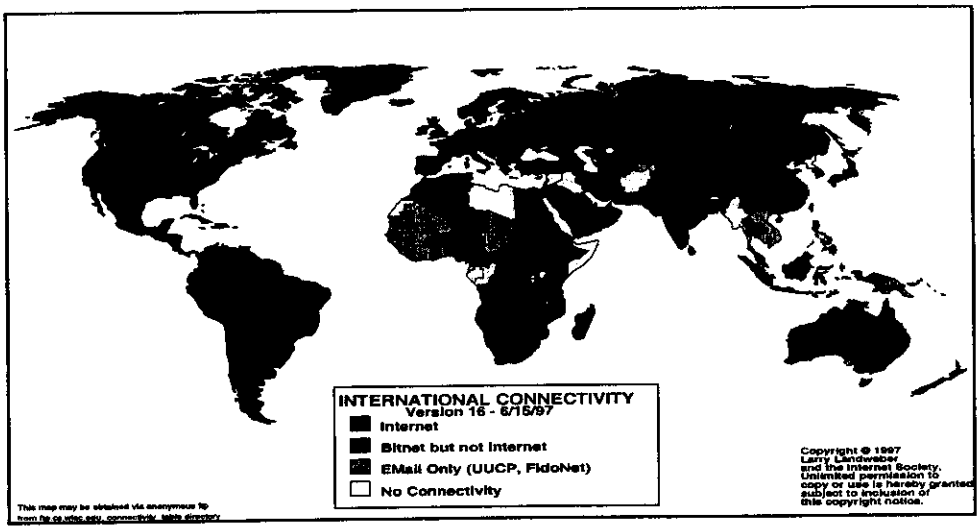


Figure 10: Worldwide Internet Connectivity

any more, Figures 11 and 12 serve to show the scale of world wide Internet deployment.

The DSL, specially ADSL (Asynchronous Digital Subscriber Line) is an innovative modem based communication technology that allows the use of the ordinary telephone line to enjoy the full multimedia capability of the Internet at home delivered at a reasonably high speed.

The world wide popularity of the Internet is largely due to the World Wide Web. This is the most pervasive application on the Internet today. Figure 11 shows the staggering growth in the number of web hosts in the world. Over a period of 4 years this number has risen from near zero to over 17.5 million (in June 2000). Also this number has more than doubled in a year!

Figure 12 shows the explosive growth of the Internet measured in terms of the number of Internet hosts connected to the Internet.

Both these show evidence of exponential growth in the usage of the Internet. This has required some rethinking of the Internet strategy, specially in relation to security, address space, quality of service, etc, in order to make it sustainable in the face of this rapid growth.

Due to this unprecedented popularity and the growth of usage of IP, today, IP suite is included as a standard component of all UNIX and UNIX like (and hence Linux) distributions, making the networking of any computer

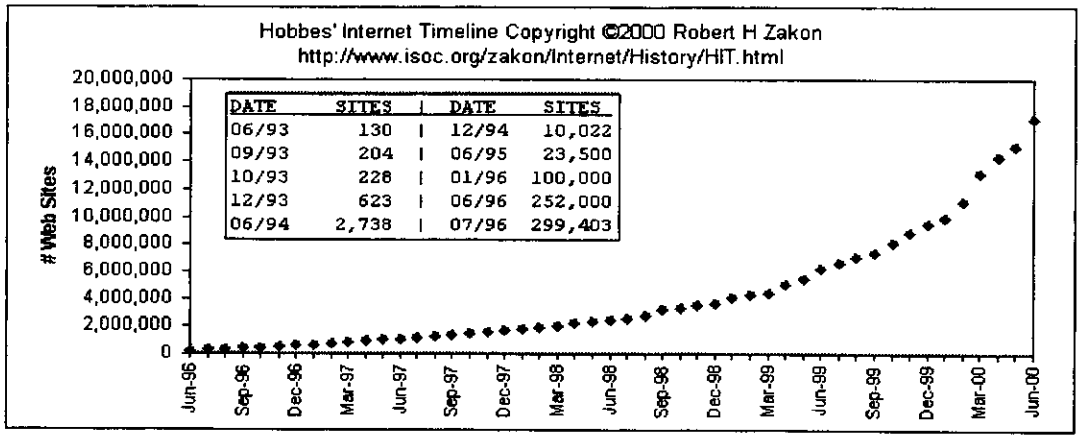


Figure 11: Growth of the number of Web hosts on the Internet

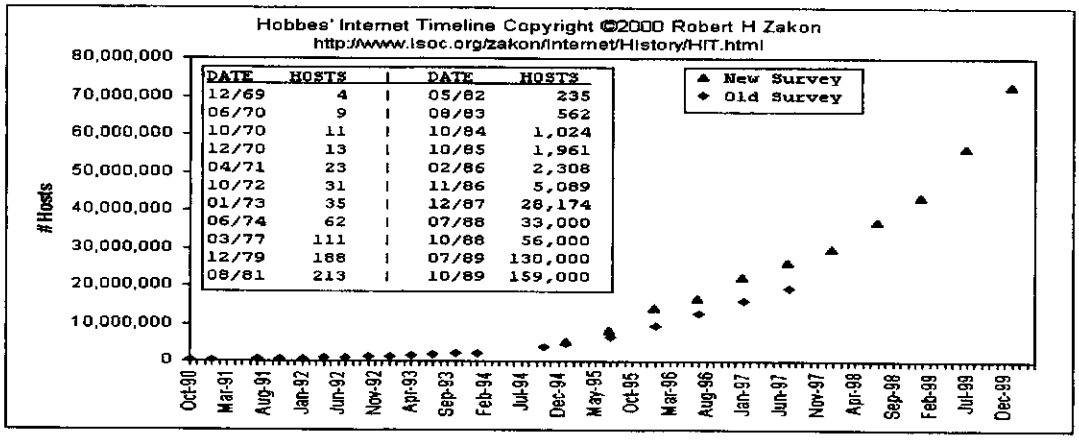


Figure 12: Growth of the number of Internet hosts

running Linux much easier. IP is also incorporated into the OSI-RM thus ensuring compliance with ISO standards.

6 Internet Protocol Architecture

By far the most successful and the most widely used protocol architecture for LANs and WANs alike is the Internet Protocol (IP) [Black 93], [Comer 88], [Stallings 89].

IP is a layered architecture (see Figure 5) consisting of only 4 layers (ISO-OSI RM has 7 !). TCP (Transmission Control Protocol) and UDP (User Datagram Protocol) are the two transport protocols supported over IP. Both the TCP and UDP make the assumption that the link is of acceptable reliability (measured in terms of its BER - Bit Error Rate) and is capable of delivering a packet to the intended destination without the intervention of the upper layers. Some of the services supported in TCP/IP, including RTP are shown in Figure 13.

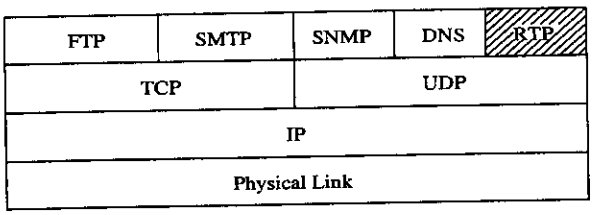


Figure 13: Services Supported over IP

6.1 IP Addressing

IPv4 uses 5 classes of addresses as shown in Figure 14.

An example of IP address allocation in a typical network is shown in Figure 15.

Many hosts have a host name associated with the IP address which can be used to address them. However it must be understood that an IP address does not identify a host. It identifies a network connection to a host because an IP address encodes both a network and a host on that network. Hence if the host is moved to another network, its IP address has to be changed.

No two devices on the global Internet should be allocated the same IP address, although on a private network with no connection to the outside world, arbitrary addresses can be allocated. The address allocation has been initially handled by the Network Information Centre (NIC). Now it is handled

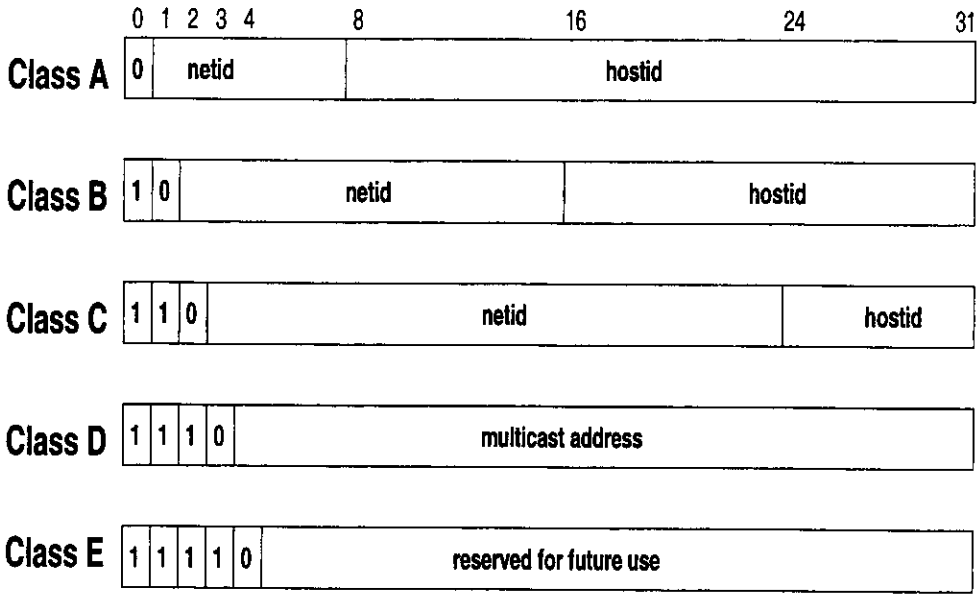


Figure 14: IP Addresses

by the NIC as well as RARE in Europe and APNIC in the Asia- Pacific region. The addresses must be officially obtained from one of the above before using them on your network.

Nowadays there are Internet Service Providers (ISPs) in many countries (most of the time more than one !) who are allocated blocks of IP addresses by the relevant NIC. These ISPs in turn allocate the IP addresses to their customers.

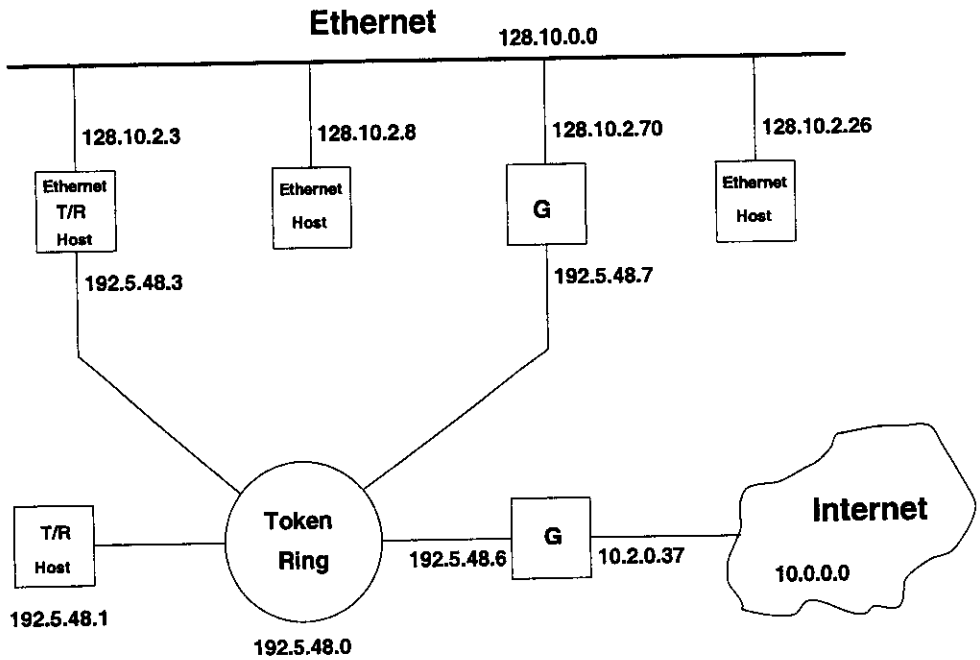
The explosive growth of the Internet has resulted in the exhaustion of the address space provided in IPv4 which uses a 32 bit (4 byte) address. IPng (IP new generation), also known as IPv6, is designed to provide among other things an enhanced address space using 16 bytes (128 bits) [RFC 1883] [3], [Huitema 96], [Goncalves 98].

6.2 The Internet Protocol

The Internet Protocol is a connectionless network layer protocol. TCP is a higher layer protocol which sits on top of IP. IP provides a connectionless (or datagram type) service with *Best Effort Delivery* to its user. In other words data given to IP (by the higher layer) is not guaranteed to be delivered.

The IP datagrams are the encapsulation of data packets passed from the higher layer with an IP header as shown in Figure 16.

[3]RFCs are available from ftp://rs.internic.net/rfc



One address for each network / segment

Internet	**10.0.0.0**	**(Class A)**
Ethernet	**128.10.0.0**	**(Class B)**
Token Ring	**192.5.48.0**	**(Class C)**
Network Hosts		
	128.10.2.8	
	192.5.48.1	
	128.10.2.3	
	192.5.48.3	

Two addresses for each gateway.

Figure 15: Allocation of IP Addresses on an internet: an Example

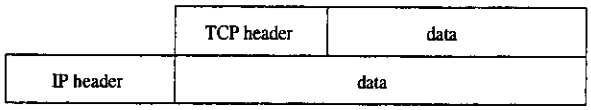


Figure 16: IP Datagram

6.3 Internetworking with IP

Figure 17 shows how IP is used to internetwork two networks running different medium access control mechanisms namely, CSMA/CD and Token Ring.

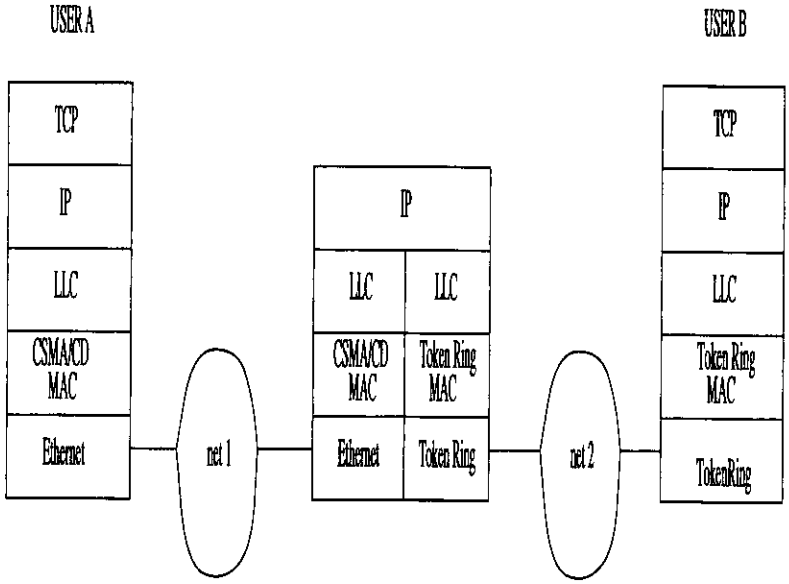


Figure 17: Internetworking Using IP

6.4 IP Datagram Format

The IP datagram format which has a preamble of ten 16 bit words and an option field of variable length is shown in Figure 18.

6.5 Brief Description of TCP and UDP

TCP and UDP are the two transport protocols used in the IP architecture [Comer 88], [Stallings 89].

TCP is a connection oriented transport protocol designed to work in conjunction with IP. TCP provides its user (application layer) with the ability

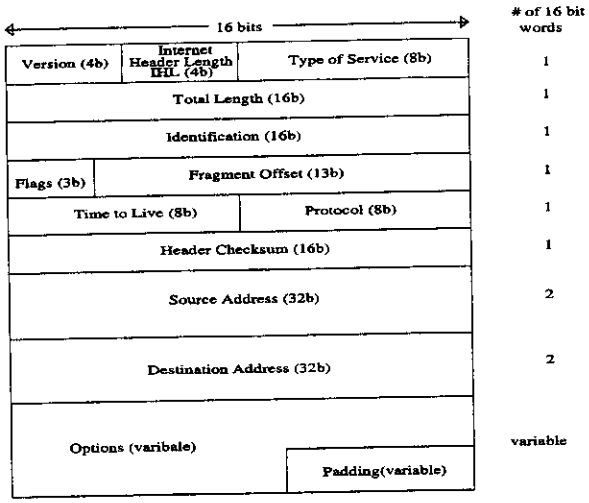


Figure 18: IP Datagram Format

to transmit reliably a byte stream to a destination and allows for multiplexing multiple TCP connections within a transmitting or receiving host computer.

Being connection oriented, TCP requires a connection establishment phase (like dialing a number to make a phone call) which is followed by the data transmission phase. A connection is terminated when it is no longer in use. TCP/IP is ideal for the transmission of bursty data. It works on the principle of retransmission of dropped packets which is one of the major contributors to delays in transmission. However, since voice and video data are time sensitive, packet technologies such as TCP/IP cannot guarantee the proper delivery of such data. Figure 19 shows the TCP header format.

UDP, on the other hand, is a connectionless transport protocol designed to operate over IP. Its primary functions are error detection and multiplexing. UDP does not guarantee the delivery of packets (compare with the ordinary postal service) but guarantees that if a packet is ever delivered in error, such error will be detected (use of checksum). It also allows for communicating with multiple processes residing on the same host computer.

UDP packet format is simple (see Figure 20). It is also fast compared to the use of TCP, since there is no connection establishment phase. Moreover, UDP is important since RTP (Real time Transport Protocol) is supported over UDP.

6.6 IP Multicasting

Multicasting is important in allowing a stream of data to be sent efficiently to many receivers. In multicasting, rather than sending a separate stream

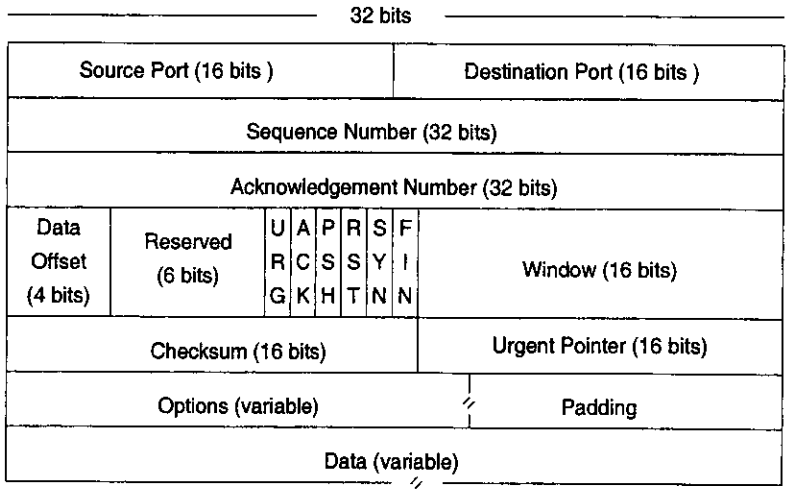


Figure 19: TCP Packet Format

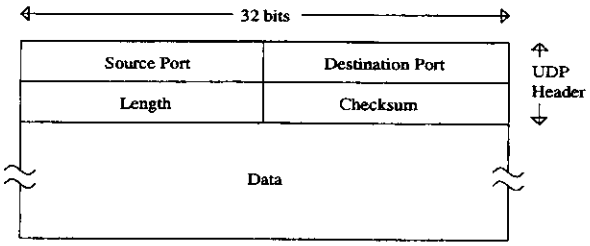


Figure 20: UDP Packet Format

of data packets to each intended user (unicasting) or transmitting all packets to everyone (broadcasting), a stream is transmitted simultaneously to a designated subset of network users. The concept of multicasting is shown in Figure 21.

IP Multicasting is defined in [RFC 1112] of 1989. A key to IP multicasting is the Internet Group Management Protocol (IGMP). IGMP enables users to sign up for multicast sessions and allows these multicast groups to be managed dynamically, in a distributed fashion. Enhancements have been made to existing protocols to direct the traffic to the members of the group. These include Distance Vector Multicasting Routing Protocol (DVMRP) and Multicast Open Shortest Path First (MOSPF) protocol. An entirely new protocol developed specifically for multicasting is the Protocol Independent Multicast (PIM).

At the start of a multicast session group addresses are allocated which are relinquished at the end of that session and reused later.

Internet MBONE is Internet's multicast backbone which is a collection of

multicast routers that can distribute multicast traffic. MBONE participants use class D Internet addresses which identify a group of hosts rather than individual hosts.

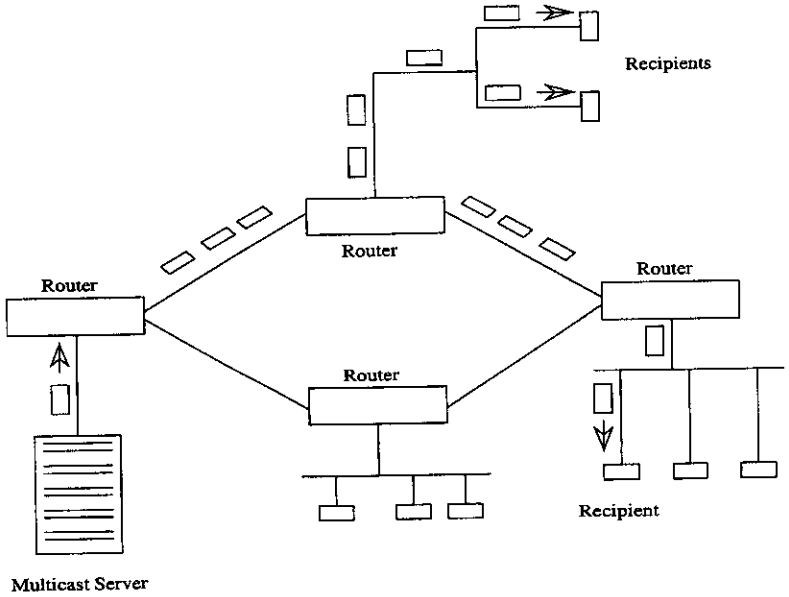


Figure 21: Multicasting

6.7 Resource Reservation Protocol (RSVP)

A key factor in achieving real time quality of service is a reservation set up protocol, a mechanism for creating and maintaining flow specific state information in the end point hosts and in routers along the data flow path. The IETF has developed its Resource Reservation Protocol (RSVP) specifically for the packet switched multicast environment.

RSVP has been designed to meet a number of requirements:

1. support for heterogeneous service needs;
2. flexible control over the way reservations are shared along branches of multicast delivery trees;
3. scalability to large multicast groups;
4. and the ability to preempt resources to accommodate advance reservations.

The RSVP protocol basically acts according to its name. An RSVP request specifies the level of resources to be reserved for some or all of the packets in a particular session. An application requests resources by specifying a flow specification, which describes the type of traffic anticipated (for example, average and peak bandwidths and level of burstiness), and a resource class specifying the type of service required (such as guaranteed delay). A filter specification is also specified, which determines the sources to which a given reservation applies.

RSVP mandates that a resource reservation be initiated by the receiver rather than the sender. While the sender knows the properties of the traffic stream it is transmitting, it has been found that the sender initiated reservation scales poorly for large, dynamic multicast delivery trees. Receiver initiated reservation deals with this by having each receiver request a reservation appropriate to itself; differences among heterogeneous receivers are resolved within the network by RSVP. After learning sender's flow specification via a higher level "out of band mechanism", the receiver generates its own desired flow specification and propagates it to senders, making reservations in each router along the way.

RSVP itself uses a connectionless approach to multicast distribution. The reservation state is cached in the router and periodically refreshed by the end station. If the route changes, these refresh messages automatically install the necessary state along the new route.

RTP and RTCP information is simply data from the point of view of routers that move the packets to their destination. RSVP prioritises multimedia traffic and provides a guaranteed quality of service. Routers that have been upgraded to support RSVP can reserve carrying capacity for video and audio streams and prevent unpredictable delays that would interfere with their transmission.

7 IPv6 – The New Generation Internet Protocol

7.1 The Design of IPv6

IPv4 is a very robust design. If it had a major design flaw the Internet could not have been so successful. However IPv6 is not a simple derivative of IPv4, but a definitive improvement [Huitema 96], [Goncalves 98].

7.2 IPv6 Header Format

Compared to IPv4 header format (Figure 18) which has 10 fixed header fields, 2 addresses and some options, the IPv6 header format shown in Figure 22 with only 6 fields and 2 addresses is in fact much simpler. Consequently the header processing is expected to be much more efficient in IPv6, thereby cutting down the processing and reducing the transit delays in internetworking devices.

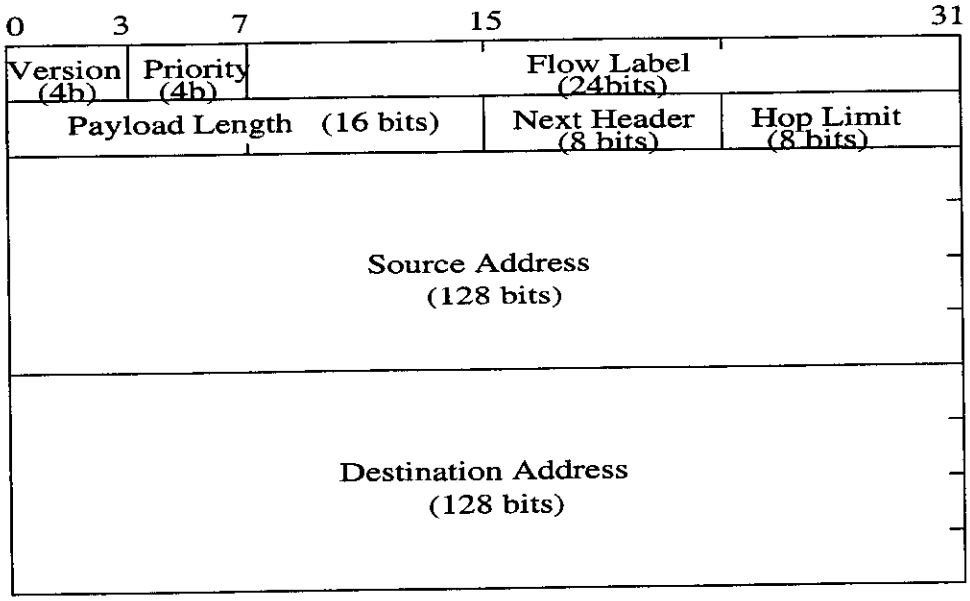


Figure 22: The IPv6 Header Format

7.3 Simplifications

The design of IPv6 has included three major simplifications (based on the experience gained in operating IPv4 for over 20 years !):

a. assign a fixed format to all headers

b. remove the header checksum

c. remove the hop by hop segmentation procedure.

IPv6 handles options in a different way (see Figure 23), in the form of extension headers and hence there is no need for header length field in the IPv6 header.

IPv6 currently defines six extension headers:

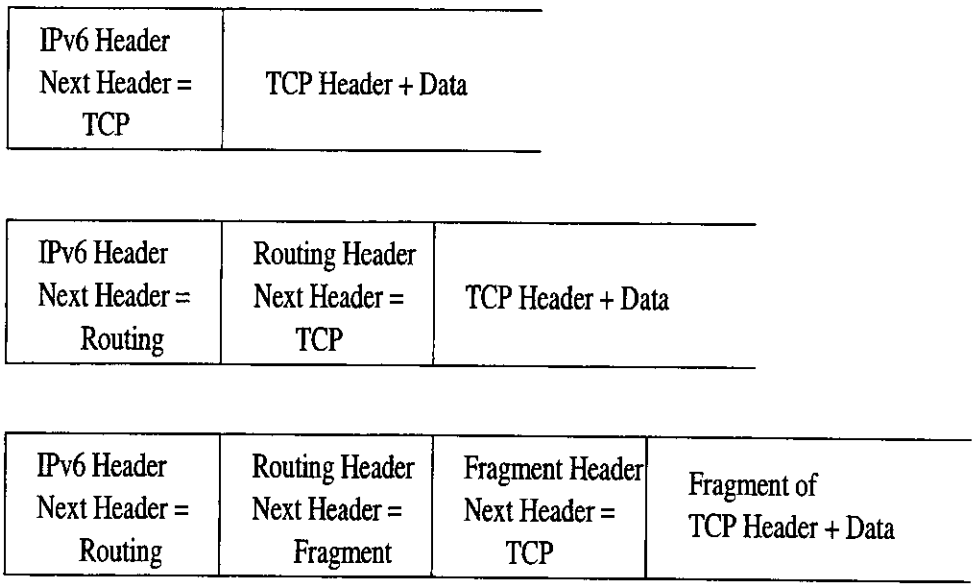


Figure 23: Chaining of IPv6 Headers

1. Hop by hop options header – special options requiring hop by hop processing
2. Routing header
3. Fragment header – fragmentation and reassembly
4. Authentication header – integrity and authentication
5. Encrypted security payload header – confidentiality
6. Destination options header – optional information to be examined by the destination node.

In order to improve the performance when handling subsequent option headers and the transport protocol which follows, IPv6 options are always an integer multiple of 8 octets long. This also helps to retain the alignment for subsequent headers.

The main advantage of removing header checksum is to diminish the cost of header processing by removing the need to check and update the checksum at each intermediate relay. This can however result in misrouted packets. Experience has shown that the risk is minimal since most encapsulation procedures include a packet checksum (eg. MAC procedure of IEEE 802.X networks, in the adaptation layers for ATM and in the framing procedure of the Point to Point Protocol for serial lines).

The last simplification is the removal of the Type of Service (TOS) field. It was found that although IPv4 provides TOS, this filed was hardly ever set by applications.

7.4 New Fields

The *Flow Label* and *Priority* have been included mostly to facilitate the handling of Real Time Traffic so as to ensure the proper treatment of high quality multimedia communications in the new Internet. Flow labels will allow the stipulation of severe real time constraints, for example.

7.5 Special Services

In traditional packet switching, a packet is queued at a switch (Figure 24) until a line is available to carry it thereby giving rise to congestion and delay.

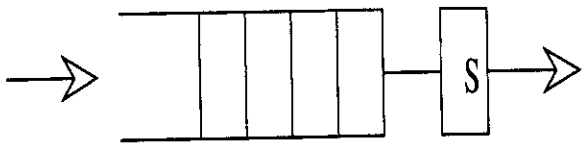


Figure 24: Traditional Packet Switching – One Queue Per Line

Assigning higher priority to the queue of real time packets over that of data packets is not enough. It is necessary to ensure that the *service rate* is equal or higher than the *arrival rate* of packets. This can be achieved as follows:

If each real time queue is serviced at a rate compatible with its requirements then it will never suffer from unpredictable queueing delays (see Figure 25). The data queue, however, will only be serviced on a *best effort basis.*

The proposers of IPv6 suggest that the *New Internet* is capable of providing all the services required by its users, including *Real Time Audio and Video.* It is their opinion that ATM is **"Another Terrible Mistake"**.

7.6 IPv6 Address Space

The address length of 128 bits gives rise to a total of $256x10^{36}$ different addresses in its address space. However due to inefficiencies in address allocation and administration (expressed by H Factor [4] which has found generally

[4]H Factor is defined by [Huitema 96] as the ratio between log(number of addresses) and the number of bits in the address

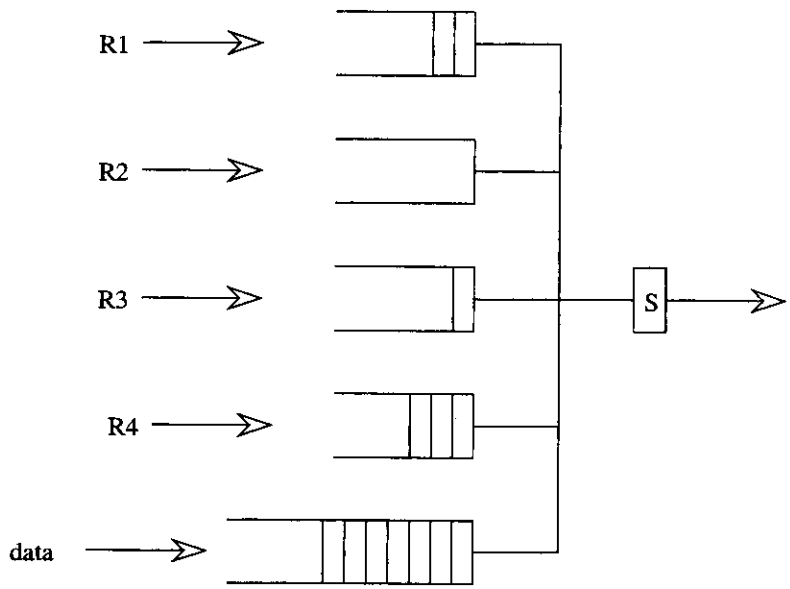


Figure 25: Real Time Streams and Data in Separate Queues

to lie in the range 0.22 to 0.26), the new Internet is expected to support 10^{15} (quadrillion) hosts and 10^{12} (trillion) networks.

The notation for writing IPv6 addresses is (remember the dot notation in IPv4!):

FEDC:BA98:7654:3210:FEDC:BA98:7654:3210

The notation allows to skip leading zeros; for instance

0000 can be written as just 0 and 0032 can be written as 32. Further the notation also allows removing a 0 leaving the colons. Thus an address such as 1030:0:0:0:80:3210:2c15:417a would become 1030::80:3210:2c15:417a. The double colon notation can be used at the beginning or at the end of an address but only once.

In the interim period an IPv4 address will be written as an IPv6 address by prepending 12 octets of zeros giving 0:0:0:0:0:0:0:0:0:0:0:0:128.145.48.12. It is also allowed to write this as ::128.145.48.12.

7.7 Making IPv6 Compliant

IPv6 will coexist with IPv4 for a number of years after which all computers will run only IPv6. During this period it is suggested to have a dual protocol stack consisting of IPv6 and IPv4 (Figure 26).

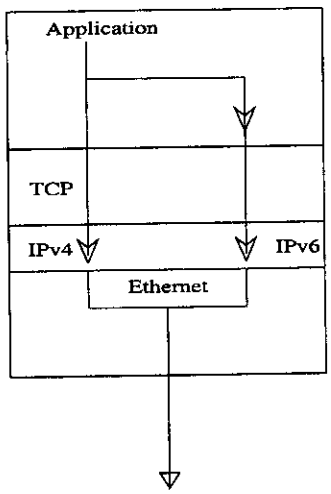


Figure 26: Typical Dual Stack Configuration

7.7.1 IPv6 Tunneling

Ipv6/IPv4 dual protocol stack will also be implemented in routers thus facilitating communication between two IPv6 compliant computers at the two ends of the IPv4 based Internet. This is known as IPv6 tunneling (see Figure 27).

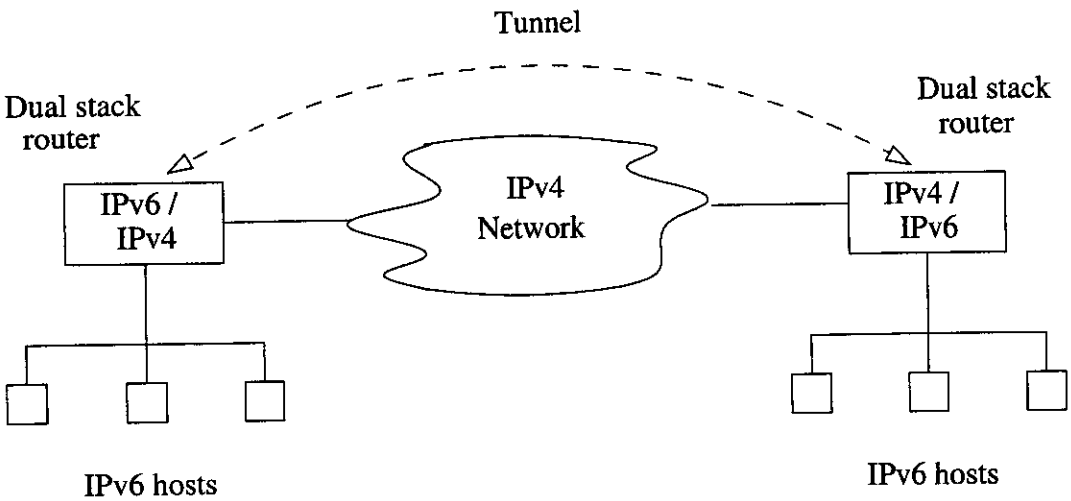


Figure 27: IPv6 Tunneling over a IPv4 Network

8 Data Communication in Real Time

Packet switching techniques based on ITU-T X.25 and IP have been traditionally used for non Real Time data transfer. Since they do not guarantee packet sequence integrity and consistent latency times in delivery, they are inherently unsuitable for Real Time applications.

The following are required to carry out Real Time data transfer on existing networks.

1. Enough bandwidth for extremely dense audio and video traffic.
2. A transport protocol appropriate for the streaming requirements of real time data (RTP).
3. A protocol to reserve network bandwidth and assigning priorities for various types of traffic (RSVP).

8.1 RTP Data Transfer Protocol

A Real time Transport Protocol is therefore needed to provide end to end network transport functions suitable for applications communicating in real time. Such applications include transmission of interactive audio and video data or real time simulation data over multicast or unicast network services.

The largest (and the oldest) network which supports real time data communication is the telephone network which falls in to the category of a circuit switched network. However in terms of a network protocol there is not much. Once the network connection is established the communication process is largely in the hands of the two persons communicating with each other.

In view of this a Real time Transport Protocol (RTP) along with a profile for carrying audio and video over RTP were defined by the IETF in January 1996 [RFC 1889], [RFC 1890].

8.1.1 Characteristics of RTP

The Realtime Transport Protocol has the following characteristics.

i. Payload type identification

ii. Sequence numbering

iii. Time stamping

iv. Delivery monitoring.

Real Time applications typically run RTP on top of UDP to make use of its multiplexing and checksum services (see Figure 28). Tailoring RTP to the application is accomplished through auxiliary profile and payload format specifications. A payload format defines the manner in which a particular payload, such as an audio or video encoding, is to be carried in RTP. A profile assigns payload type numbers for the set of payload formats that may be used in the application.

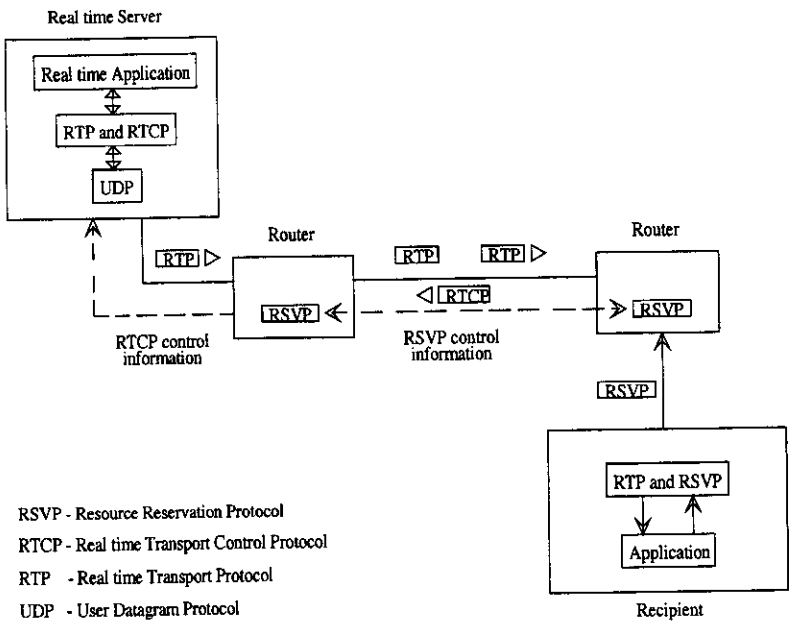


Figure 28: Real Time Application running RTP on top of UDP

However RTP is not limited to be used with UDP/IP. It can be used equally with other underlying network or transport protocols such as ATM or IPX. Moreover, RTP supports data transfer to multiple destinations using multicast distribution if provided by the underlying network, a feature which makes RTP ideal for multi party multimedia conferencing.

RTP is designed to work in conjunction with RTCP (Real time Transport Control Protocol) to monitor the quality of service. RTP delivers real time traffic with timing information for reconstruction as well as feedback on reception quality. The Resource Reservation Protocol (RSVP) is used to reserve network bandwidth and assign priority for various traffic types.

8.1.2 Definitions in RTP

The following definitions are extracts from [RFC 1889, RFC 1890].

- RTP Payload

  The data transported by RTP in a packet, for example audio samples or compressed video data.

- RTP Packet

  A data packet consisting of the fixed RTP header, a possibly empty list of contributing sources, and the payload data. Typically one packet of the underlying protocol contains a single RTP packet, but several RTP packets may be contained if permitted by the encapsulation method.

- RTCP Packet

  A control packet consisting of a fixed header part similar to that of RTP data packets, followed by structured elements that vary depending upon the RTCP packet type. Typically multiple RTCP packets are sent together as a compound RTCP packet in a single packet of the underlying protocol. This is enabled by the length field of the fixed header of each RTPC packet.

- Port

  The "Abstraction" that transport protocols use to distinguish among multiple destinations within a given host computer (TCP/IP protocols identify ports using small positive integers and the transport selectors (TSEL) used by the OSI Transport layer are equivalent to ports). RTP depends on the lower layer protocol to provide some mechanism such as ports to multiplex the RTP and RTCP packets of a session.

- Transport Address

  The combination of a network address and port that identifies a transport level end point, for example an IP address and a UDP port. Packets are transported from a source transport address to a destination transport address.

- RTP Session

  The association among a set of participants communicating with RTP. For each participant, the session is defined by a particular pair of destination transport addresses consisting of one network address and a port pair for RTP and RTCP. The destination transport address pair may be common for all participants, as in the case of IP multicast, or may be different for each, as in the case of individual unicast network addresses plus a common port pair. In a multimedia session,

each medium is carried in a separate RTP session with its own RTCP packets. The multiple RTP sessions are distinguished by different port number pairs and/or different multicast addresses.

- Synchronisation Source (SSRC)

  The source of a stream of RTP packets, identified by a 32 bit numeric SSRC identifier carried in the RTP header so as not to be dependent upon the network address. Examples of synchronisation sources include the sender of a stream of packets derived from a signal source such as a microphone, a camera or an RTP mixer. If a participant generates multiple streams in one RTP session, for example from separate video cameras, each must be identified as a different SSRC.

- Contributing Source (CSRC)

  A source of a stream of RTP packets that has contributed to the combined stream produced by an RTP mixer. The mixer inserts a list of the SSRC identifiers of the sources that contribute to the generation of a particular packet into the RTP header of that packet. This list is called the CSRC list. An example application is audio conferencing where a mixer indicates all the talkers whose speech was combined to produce the outgoing packet, allowing the receiver to indicate the current talker, even though all the audio packets contain the same SSRC identifier (that of the mixer).

- End System

  An application that generates the content to be sent in RTP packets and/or consumes the content of received RTP packets.

- Mixer

  An intermediate system that receives RTP packets from one or more sources, possibly changes the data format, combines the packets in some manner and then forwards a new RTP packet. Since the timing among multiple input sources will not generally be synchronised, the mixer will make timing adjustments among the streams and generate its own timing for the combined stream. Thus all data packets originating from a mixer will be identified as having the mixer as their synchronisation source.

- Monitor

  An application that receives RTCP packets sent by participants in an RTP session, in particular the reception reports, and estimates the current quality of service, fault diagnosis and long term statistics.

8.1.3 RTP Fixed Header Fields

The RTP header format is shown in Figure 29. The first twelve octets are present in every RTP packet, while the list of CSRC identifiers is present only when inserted by a mixer.

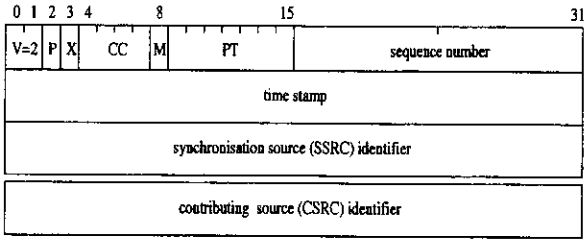


Figure 29: RTP Header Format

The RTP header provides the timing information necessary to synchronise and display audio and video data and to determine whether packets have been lost or arrive out of order. In addition, the header specifies the payload type, thus allowing multiple data and compression types. This is a key advantage over most proprietary solutions, which specify a particular type of compression and thus limit users' choice of compression schemes.

8.1.4 Multiplexing RTP Sessions

In RTP, multiplexing is provided by the destination transport address (network address and a port number) which define an RTP session.

8.1.5 Real time Transport Control Protocol (RTCP)

The RTCP is based on the periodic transmission of control packets to all participants in the session, using the same distribution mechanism as the data packets. The underlying protocol must provide multiplexing of the data and control packets, for example using separate port numbers with UDP.

The primary function of RTCP is to furnish information on the quality of data distribution. This feedback is a critical part of RTP's use as a transport protocol, since applications can use it to control how they behave. The feedback is also important for diagnosing distribution faults. For instance, by

monitoring reports from all data recipients, network managers can determine the spread of a problem. When used in conjunction with IP multicast, RTCP enables the remote monitoring and diagnosis.

In addition RTCP controls the rate at which participants in an RTP session transmit RTCP packets. In a session with a few participants, RTCP packets are sent at the maximum rate of one every five seconds whereas for a larger group, RTCP packets may be sent only once every 30 seconds. In other words, the more participants there are in a conference, the less frequently each participant sends RTCP packets. This makes RTCP scalable to accommodate tens of thousands of users.

8.2 Real Time Data Transfer using ATM

Audio and video applications generate lots of bits, and the traffic has to be streamed or transmitted continuously rather than in bursts. This is in contrast to conventional data types such as text, files and graphics, which are able to withstand short and inconsistent periods of delay between packet transmissions. What is needed then is a network capable of transporting both streaming and bursty data. ATM (Asynchronous Transfer Mode) is a technique which just does this [De Prycker 95], [Stallings 95].

ATM is a connection oriented protocol designed to support high bandwidth, low delay (even services with predictable delay), packet like switching and multiplexing. The design of ATM ensures the capability to carry both stream traffic (such as voice and video) and bursty traffic (such as interactive data) with guaranteed QOS. It uses a fixed cell size for all types of traffic. In the case of stream traffic ATM guarantees the integrity of cell sequence which is essential for the successful delivery of such traffic.

ATM has grown out of the need for a worldwide standard to allow interchange of information regardless of the "end system" or type of information. Historically there have been separate methods used for the transmission of information among users on LANs and the users on WANs. This situation has been made more complex by the user's need for connectivity expanding from the LAN to MAN to WAN. ATM is a method to unify the communication of information on LANs and WANs.

ATM is the only technology based on standards, and has been designed from the beginning to accommodate the simultaneous transmission of data, voice and video. It is an easily scalable backbone which can be upgraded merely by adding more switches or links. ATM is switched instead of routed and therefore it is faster since not every IP packet at every node is examined to determine its destination. ATM LAN Emulation (LANE) allows transparent interconnection of "legacy" LANs based on Ethernet or FDDI

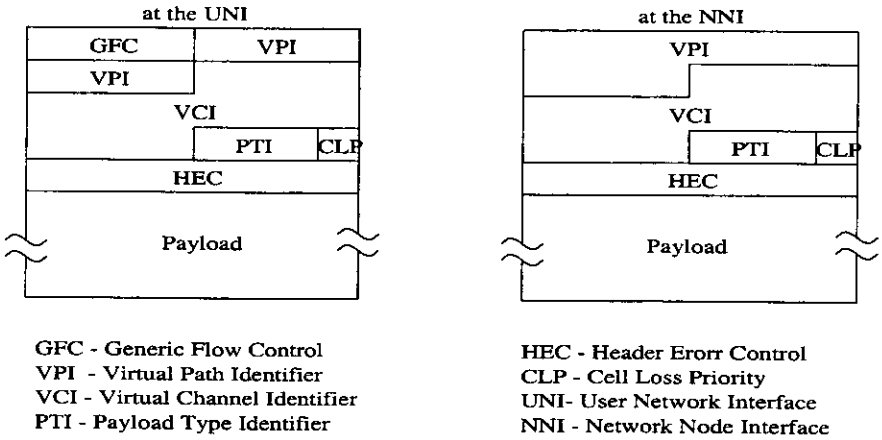


Figure 30: ATM Cell Formats

technology, making the ATM backbone look like a fast Ethernet or FDDI to workstation applications. As more and more ATM nodes are deployed, the differences between local and wide area networks will disappear to form a seamless network based on one standard.

To use the limited bandwidth more efficiently, ATM uses circuit switching principles to give the users a full channel to themselves. Since these users do not use the full channel all the time, ATM uses statistical analysis to time division multiplex several users onto the same line. This allows each user to have all of the channel's bandwidth for the period of time in which it is needed.

To achieve this ATM uses two connection concepts; the *Virtual Channel (VC)* and the *Virtual Path (VP)*.

A virtual channel (also known as a virtual circuit) provides a logical connection between end users and is identified by a VCI (Virtual Channel Identifier) in the ATM header (see Figure 30). A virtual path defines a collection of virtual circuits traversing the same path in the network and is identified by a VPI (Virtual Path Identifier). The VPI emulates the functions of the trunk concept in circuit switching. Thus virtual paths define the cross connection functions across the network, whereas virtual channels are concerned with switching and connection establishment functions. Virtual paths are statistically multiplexed on the physical link on a cell multiplexing basis.

GFC (Generic Flow Control) is used to control the amount of traffic entering the network.

VPI and VCI are used for routing. VPI will change from one node to the next when it travels through the ATM layer. VCI is predefined and usually

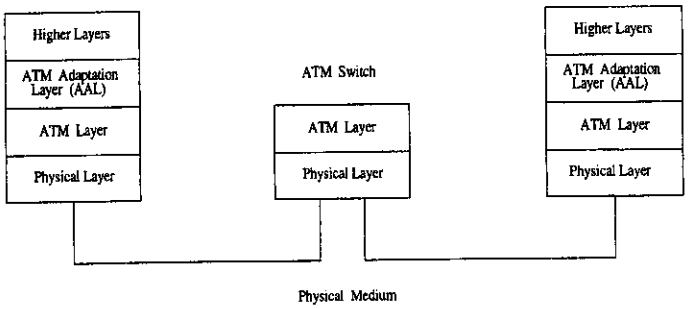


Figure 31: ATM Protocol Architecture

remains the same throughout the duration of the transmission. PTI (Payload Type Identifier) is used to distinguish between cells that are carrying user data and those carrying control information. CLP is a single control bit which provides selective discard during network congestion and HEC is used to check header errors.

The ATM cell formats used at the UNI (User-Network Interface) and NNI (Network-Node Interface) are shown in Figure 30.

8.2.1 ATM Protocol Structure

Figure 31 shows the ATM layered architecture as described in ITU-T recommendation I.321 (1992). This is the basis on which the B-ISDN Protocol Reference Model has been defined.

- ATM Physical Layer

  The physical layer accepts or delivers payload cells at its point of access to the ATM layer. It provides for cell delineation which enables the receiver to recover cell boundaries. It generates and verifies the HEC field. If the HEC cannot be verified or corrected, then the physical layer will discard the errored cell. Idle cells are inserted in the transmit direction and removed in the receiving direction.

  For the physical transmission of bits, 5 types of transmission frame adaptations are specified (by the ITU and the ATM Forum). Each one of them has its own lower bound or upper bound for the amount of bits it can carry (from 12.5 Mbps to 10 Gbps so far).

  1. Synchronous Digital Hierarchy (SDH) $\geq$ 155 Mbps;
  2. Plesiochronous Digital Hierarchy (PDH) $\leq$ 34 Mbps;
  3. Cell Based $\geq$ 155 Mbps;

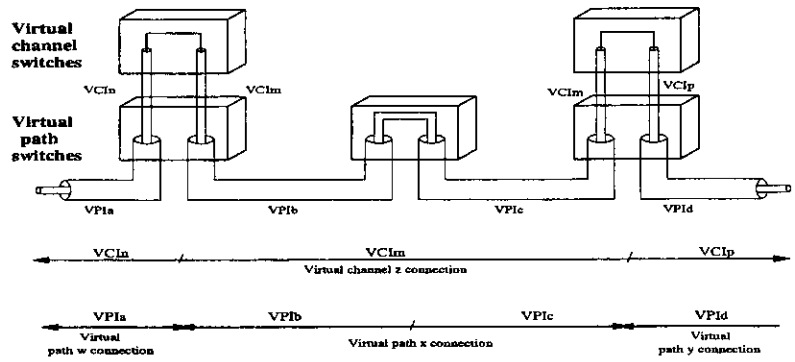


Figure 32: VC/VP Switching in ATM

4. Fibre Distributed Data Interface (FDDI) = 100 Mbps;
5. Synchronous Optical Network (SONET) $\geq$ 51 Mbps.

The actual physical link could be either optical or coaxial with the possibility of Unshielded Twisted Pair (UTP Category 3/5) and Shielded Twisted Pair (STP Category 5) in the mid range (12.5 to 51 Mbps).

- ATM Layer

  ATM layer mainly performs switching, routing and multiplexing. The characteristic features of the ATM layer are independent of the physical medium. Four functions of this layer have been identified.

  1. cell multiplexing (in the transmit direction)
  2. cell demultiplexing (at the receiving end)
  3. VPI/VCI translation
  4. cell header generation/extraction.

  This layer accepts or delivers cell payloads. It adds appropriate ATM cell headers when transmitting and removes cell headers in the receiving direction so that only the cell information field is delivered to the ATM Adaptation Layer.

  At the ATM switching/cross connect nodes VPI and VCI translation occurs. At a VC switch new values of VPI and VCI are obtained whereas at a VP switch only new values for the VPI field are obtained (see Figure 32). Depending on the direction, either the individual VPs and VCs are multiplexed into a single cell or the single cell is demultiplexed to get the individual VPs and VCs.

- ATM Adaptation Layer (AAL)

  The ATM Adaptation Layer (AAL) is between ATM layer and the higher layers. Its basic function is the enhanced adaptation of services provided by the ATM layer to the requirements of the higher layers.

  This layer accepts and delivers data streams that are structured for use with user's own communication protocol. It changes these protocol data structures into ATM cell payloads when receiving and does the reverse when transmitting. It inserts timing information required by users into cell payloads or extracts from them. This is done in accordance with five AAL service classes defined as follows.

  1. AAL1 - Adaptation for Constant Bit Rate (CBR) services (connection oriented, 47 byte payload);
  2. AAL2 - Adaptation for Variable Bit Rate (VBR) services (connection oriented, 45 byte payload);
  3. AAL3 - Adaptation for Variable Bit Rate data services (connection oriented, 44 byte payload);
  4. AAL4 - Adaptation for Variable Bit Rate data services (connection less, 44 byte payload);
  5. AAL5 - Adaptation for signalling and data services (48 byte payload).

  In the case of transfer of information in real time, AAL1 and AAL2 which support connection oriented services are important. AAL4 which supports a connection less service was originally meant for data which is sensitive to loss but not to delay. However, the introduction of AAL5 which uses a 48 byte payload with no overheads, has made AAL3/4 redundant. Frame Relay and MPEG -2 (Moving Pictures Expert Group) video are two services which will specifically use AAL5.

8.2.2 ATM Services

- CBR Service

  This supports the transfer of information between the source and destination at a constant bit rate. CBR service uses AAL1. A typical example is the transfer of voice at 64 Kbps over ATM. Another usage is for the transport of fixed rate video.

  This type of service over an ATM network is sometimes called circuit emulation (similar to a voice circuit on a telephone network).

- VBR Service

  This service is useful for sources with variable bit rates. Typical examples are variable bit rate audio and video.

- ABR and UBR Services

  The definition of CBR and VBR has resulted in two other service types called Available Bit Rate (ABR) services and Unspecified Bit Rate (UBR) services.

  ABR services use the instantaneous bandwidth available after allocating bandwidths for CBR and VBR services. This makes the bandwidth of the ABR service to be variable. Although there is no guaranteed time of delivery for the data transported using ABR services, the integrity of data is guaranteed. This is ideal to carry time insensitive (but loss sensitive) data such as in LAN-LAN interconnect and IP over ATM.

  UBR service, as the name implies, has an unspecified bit rate which the network can use to transport information relating to network management, monitoring, etc.

8.2.3 IP over ATM

The transmission of classical IP traffic over ATM can be accomplished as shown in figure 33.

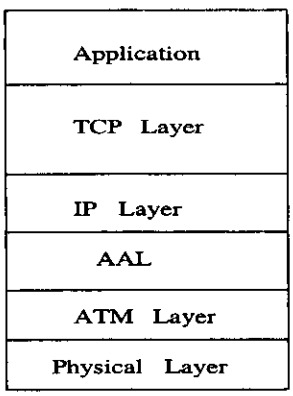


Figure 33: Transmission of IP over ATM

8.3 IP/TV - A Real life Example

IP/TV^{TM} is a client server application that multicasts live or prerecorded digital video and audio streams in real time to an unlimited number of users

over any IP based local or wide area network including the global Internet, using fully compliant TCP/IP protocol stacks supporting real time protocol components. It uses state-of-the-art Internet standards such as IP multicasting, RTP, RTCP and RSVP in its *Flashware*^{TM} software suite to provide high quality, synchronised audio/video information over existing packet switched networks simultaneous with current network data traffic.

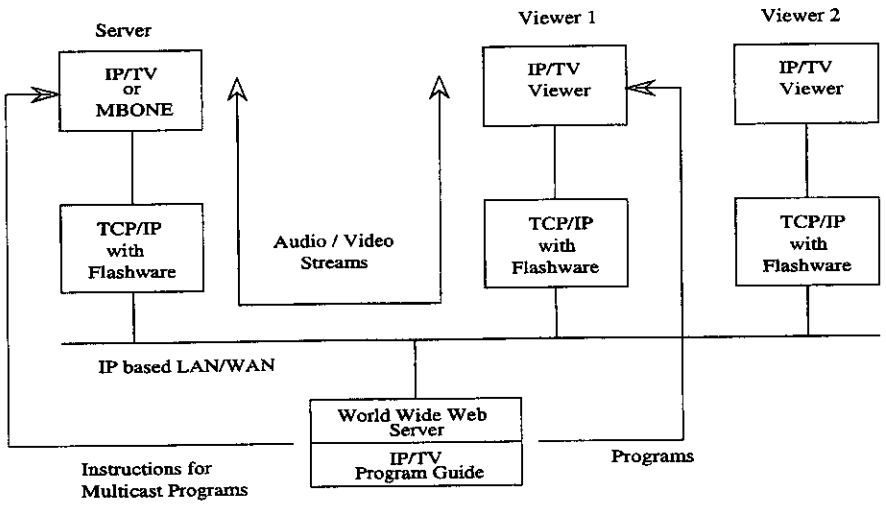


Figure 34: Real Time Audio/Video Over IP based LAN/WANs

IP/TV [5] consists of a Viewer, a Program Guide and a Server (Figure 34). The program guide shows a schedule of multicasts and can be accessed via Web browser with HTTP (Hyper Text Transport Protocol). MBONE session information can be accessed with the Program Guide which controls the number of streams allowed on the network and the format of those streams, ie. audio only, audio and video or some other combination. The server delivers prerecorded or live multimedia streams based on the Program Guide schedule and parameters such as start time and file name.

The IP/TV viewer, a tool for signing up for scheduled multicasts, is designed to provide VCR like controls as well as "channel changing" controls. With a software based codec (compliant with ITU video conferencing standard H.320/H.261) colour video running at a rate of 30 frames per second uses about 500 Kbps bandwidth. The use of IP multicasting helps to conserve network bandwidth by transmitting over the network a single data stream that can be picked up by any interested user. The use of RSVP provides the ability to reserve bandwidth on RSVP compliant routers, thereby giving priority to time dependent audio/video streams over less critical network traffic

[5] available with Flashware from Precept Software Inc.

thus ensuring the desired Quality of Service (QOS).

8.4 Delivering Real Time Data to the Desk Top

8.4.1 Ethernet Based Solutions

Ethernet was invented by Dr Robert M Metcalfe in the 1970s at the Xerox PARC (Palo Alto Research Centre). This was at a time when a network bandwidth of 3Mbps was found to be adequate to serve the needs of the users whose computers were connected to such networks. Formal specifications for Ethernet were published in 1980 by a multi vendor consortium that created the DEC-Intel-Xerox (DIX) standard and turned the 3Mbps Ethernet into an open production quality Ethernet operating at 10Mbps. The popularity of Ethernet made the LAN Standards Committee of the Institute of Electrical and Electronic Engineers (IEEE 802) to adopt and publish this as an IEEE standard in 1985. The IEEE standard which is based on the original DIX technology and provides an "Ethernet like" system, is called the IEEE 802.3 Carrier Sense Multiple Access/ Collision Detect (CSMA/CD).

The original specification of 1985 specifies copper based transmission media, specially, thick (10base5) and thin (10base2) coaxial cables. Since then it has grown to include twisted pair (10/100baseT) and optical fibre (10/100baseFL). It is estimated that there are about 400 million Ethernet connections in the world today, making it the most widely deployed LAN technology.

However, the majority of today's computing is based on *client/server technology.* As more applications are moved form desktops to severs, and the number of desktop clients increases, the demand placed on a typical Ethernet operating at 10Mbps tends to affect the application response time and impair the users' ability to effectively access, manipulate and transmit information. Moreover the high performance servers of today need maximum bandwidth and highly reliable network connections to make the most of their powerful processing capabilities.

Clients, on the other hand, require high speed, low cost connections to the high bandwidth networks. This asymmetric approach to client/server network design has led to the development of Fast Ethernet (IEEE 802.3u - 100base..) and Gigabit Ethernet (IEEE 802.3z - 1000base..) specifications giving a speed advantage of 10 and 100 times respectively, compared to the original Ethernet. Associated with the high bandwidth is also the switching technique, which makes the Ethernet and its variants to be appropriate for the delivery of real time data.

8.4.2 Signal Topology and Timing

The *signal topology* of the Ethernet is also known as the *logical topology* to distinguish it from the actual physical layout of the media cabling. The logical topology of an Ethernet provides a single channel (bus) that carries signals to all stations attached to it. Repeaters which amplify and re-time the signals can be used to link multiple Ethernet segments together to form large Ethernets.

The signal timing is based on the amount of time it takes to traverse the complete media system from one end to the other and back. This is also called the round trip time, the maximum of which is strictly limited to ensure that every interface can hear all network signals within the specified amount of time provided in the Ethernet Medium Access Control (MAC) mechanism.

The longer a given network segment is the more time it takes for a signal to traverse it. Therefore configuration guidelines (rules) are provided to make sure that the round trip timing limits are met, no matter what combination of media segments are used in the network configuration.

However the expansion of Ethernet using repeaters is limited because of the signal timing restrictions. In order to meet the expansion needs of today, two kinds of hubs known as *repeater hubs* and *packet switching hubs* are available. The advantage of using a switching hub is that each port of the hub provides a connection to an Ethernet media system that operates as a separate Ethernet LAN, and the round trip timing rules for each LAN stop at the switching hub port. This in effect allows several individual LANs each supporting hundreds of computers to be linked together forming a much larger LAN, but still meeting the requirements of signal timing.

The use of switching hubs allows, in addition, to mix Ethernet links operating at 10Mbps with links operating at higher link speeds and to operate either some or all of the links at higher bandwidths, namely 100Mbps or 1000Mbps.

8.4.3 High Performance Ethernets

Two approaches have been considered in making the Ethernet to operate at higher bandwidths. In the first approach, the original Ethernet system with the CSMA/CD MAC mechanism has been speeded up to 100Mbps. This approach is therefore called 100baseT Fast Ethernet (IEEE802.3u).

The second approach has been to create an entirely new MAC mechanism using hubs that control access to the medium using a **demand priority** mechanism. The design of the medium access control based on demand

priority mechanism has allowed the transportation of token ring frames in addition to the standard Ethernet frames. Therefore it is called 100VG-AnyLAN (IEEE802.12).

Switching invariably becomes central to the operation of high performance Ethernets. The backplane speed (in packets per second (PPS)) describes the capacity of the switch to move the data from the incoming ports to the outgoing ports and is an important parameter. Another factor which is important for the efficient functioning of a switch is the amount of buffer memory provided. In general, higher the number of ports the more buffer memory is needed.

8.4.4 Switched Ethernets

An Ethernet switch at its basic level can be thought of as a bridge with many ports and low latency. A network segment connected to each of the ports physically represent a separate Ethernet with its own repeater count and timing restrictions. Switches are useful in segmenting large networks for improved performance and/or easy manageability. However the many segments of the network operates as one single logical network.

The design of Ethernet switches gives rise to the following classification based on the type of packet forwarding technique used in the switch architecture. The packet switching type employed has an effect on latency which describes the delay in the switch. The latency has the greatest impact in environments where real time video and audio applications are supported.

- Store and Forward Switches

A store and forward switch stores each incoming frame in a buffer, checks for errors, and if the frame is good then forwards the frame to its destination port. The store and forward technique has the advantage that it prevents wasting network bandwidth by effectively blocking the damaged frames. However the disadvantage is that it increases the latency and therefore results in lower throughput in networks with few errors. Store and forward switches are useful in networks which may experience high error rates.

In general Store and Forward switching is likely to be the best choice when a network needs efficiency and stability.

- Cut Through Switches

In a cut through switch a frame is forwarded immediately upon receiving its destination address thus resulting in a very low latency in the switch. The disadvantage is that it propagates errors and therefore is only suitable for networks which experience a few occasional errors.

- Hybrid Switches

Hybrid switch is the result of an attempt to achieve the best of both the Store & Forward and Cut Through techniques. In its normal operating mode a hybrid switch operates as a Cut Through switch constantly monitoring the rate at which damaged or invalid frames are forwarded. When the error rate is above a certain threshold then the switch reverts to Store and Forward mode and continues to operate in that mode until the error rate has fallen to an acceptable level before reverting to Cut Through mode.

The Hybrid switch has the performance advantage of a Cut Through switch when the error rate is low, and the error trapping ability of a Store and Forward switch when the error rates are high.

The above types are only applicable when the source and destination ports are all running at the same speed. If the switch has to perform a speed conversion, which is the case in many new network installations especially if a standard Ethernet is migrated to high performance Ethernet, then the switch must operate in the Store and Forward mode to cater to the differing link speeds.

8.4.5 Fast Ethernet

Fast Ethernet (IEEE 802.3u) operating at 100Mbps is designed as the most direct and simple extension of 10baseT Ethernet operating at 10Mbps. The Medium Access Control mechanism used in the Fast Ethernet is simply a scaled up version of the MAC technique used by 10baseT Ethernet. This makes 100baseT to be similar to the conventional 10baseT, only faster. It uses the same reliable, robust and economical technology of 10baseT. The seamless compatibility between 10baseT and 100baseT allows easy migration to high speed network connections. The Fast Ethernet specification includes a mechanism for *auto negotiation* of the media speed. This allows the installation of a dual speed Ethernet interface which automatically detects and sets its link speed.

Although Fast Ethernet preserves the critical 100m maximum UTP cable length from the hub to the desktop, the rules applicable to the 100Mbps technology are different because of the scaling of the MAC interface.

100baseT Fast Ethernet is a natural evolution from the standard 10baseT Ethernet. Figure 35 shows the two in comparison.

Fast Ethernet also supports multiple media types. For 100baseT the same cabling installed for a 10baseT network can be used. Figure 36 shows the three media specifications supported by Fast Ethernet, namely 100baseT4, 100baseTX and 100baseFX.

Feature	10baseT Ethernet	100baseT Fast Ethernet
Speed	10Mbps	100Mbps
IEEE Standard	802.3	802.3
Media Access Protocol	CSMA/CD	CSMA/CD
Topology	Bus or Star	Star
Media support	Coax, UTP, Optical Fiber	UTP, STP, Optical Fiber
Hub to node distance (Maximum)	100 meters	100 meters
Media Interface	AUI	MII

Figure 35: Comparison between Ethernet and Fast Ethernet

- 100baseTX

The 100baseTX specification supports 100Mbps transmission speed over two pairs of UTP Category 5 or STP Category 5. The RJ45 connector used for 100baseTX UTP is exactly the same as that used by 10baseT UTP. For STP wiring 100baseTX also specifies the traditional DB-9 connector.

- 100baseT4

100baseT4 media uses four pairs of Category 3, 4 or 5 UTP wiring to carry data at 100Mbps. The signalling scheme in 100baseT4 uses three pairs of wires for data and the fourth for collision detection. Because 100baseT4 can use Category 3 it enables migration to 100baseT4 without having to rewire.

100baseT4 also uses RJ45 connector.

- 100baseFX

100baseFX media specification defines 100Mbps operation over two strands of 62.5/125 micron fibre and allows transmission over greater distances than UTP. The fibre optic connectors are the same as those defined for 10baseFX networks.

Fast Ethernet specification includes a Media Independent Interface (MII) which defines a standard interface between the CSMA/CD MAC layer and

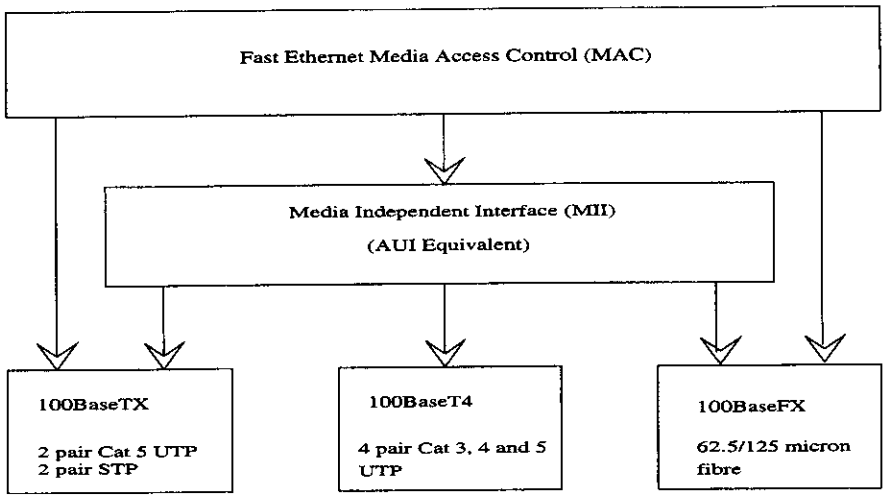


Figure 36: Fast Ethernet Media Support

any of the three media specifications supported. This is much like the AUI connector for standard Ethernet. The MII defines a 40 pin connector to connect to the external transceivers.

The Fast Ethernet however does not support coaxial cabling largely due to its inability to support high data rates over the distances of interest. The ISO 11801 cabling standard is applicable to Fast Ethernet implementation.

8.4.6 Gigabit Ethernet

The ever increasing processing power in the computers and the deployment of power hungry applications on the networks must be matched by developing high speed network connections to reduce traffic bottlenecks, improve overall performance and ultimately enhance the productivity of the users who use the network. The development of Gigabit Ethernet is clearly a solution to match the network infrastructure with the desktop capability.

Gigabit Ethernet (IEEE802.3z) is an extension of the highly successful 10Mbps Ethernet and 100Mbps Fast Ethernet standards and therefore is fully compatible with the huge installed Ethernet base. Gigabit Ethernet employs all the features of Ethernet specification including frame format, support for CSMA/CD protocol, full duplex transmission, flow control and management objects. This compatibility preserves investment in network administrator expertise and support staff training.

The enhancements in Gigabit Ethernet includes the support of fast optical fiber connections at the physical layer of the network and a MAC layer specification which sustains a tenfold increase in the MAC layer data rates.

This is a key to its ability to support data intensive applications such as imaging and video conferencing.

The proliferation of Gigabit Ethernet will take place in phases. Initially Gigabit Ethernet will be used as backbone switch-to-switch connections. The next phase will be to deploy switch-to-server connectivity to boost access to critical server resources. This evolution will be driven by the increasing installation of PCs with 100Mbps network interfaces. Finally as the desk top costs come down and user network demand increases, Gigabit Ethernet switches will enter the backbone and will take over the switch fabric.

At this stage, Gigabit Ethernet will probably use links that are compliant with the installed base of UTP Category 5 cabling up to 100 m distances.

8.5 ADSL – delivering RT multimedia to the home and small business

ADSL technology is the result of the recent advances in modem technology that converts existing twisted pair telephone subscriber lines into access paths for the transfer of multimedia and high speed digital data. Present day ADSL can sustain downstream transmission speeds up to 6 Mbps to a subscriber and in excess of 800 Kbps in both directions (duplex).

These transmission rates expand the existing access capacities by a factor of 100 or more (V.90 modem supports 56 Kbps nominally) without new cabling, thus transforming the existing public telephone infrastructure which is only capable of delivering voice, text and low resolution graphics to a powerful system capable of bringing multimedia, including full motion video, in quality assured real time, to the home and small business.

ADSL is seen as an interim solution which will play a crucial role in the next decade for the delivery of information in video and other multimedia formats. This is expected to pave the way for a full broad band service, which will take decades to reach all prospective customers. In this interim period ADSL will bring movies, television, video catalogues, remote CDROMs, corporate LANs and the global Internet into homes and small businesses.

An ADSL circuit consists of an ADSL modem at each end of a twisted pair telephone subscriber line creating three information channels as follows (see Figure 37):

- high speed downstream channel (up to 6 Mbps)
- medium speed duplex channel (800 Kbps)
- POTS (Plain Old Telephone Service) or an ISDN channel (64 – 128 Kbps)

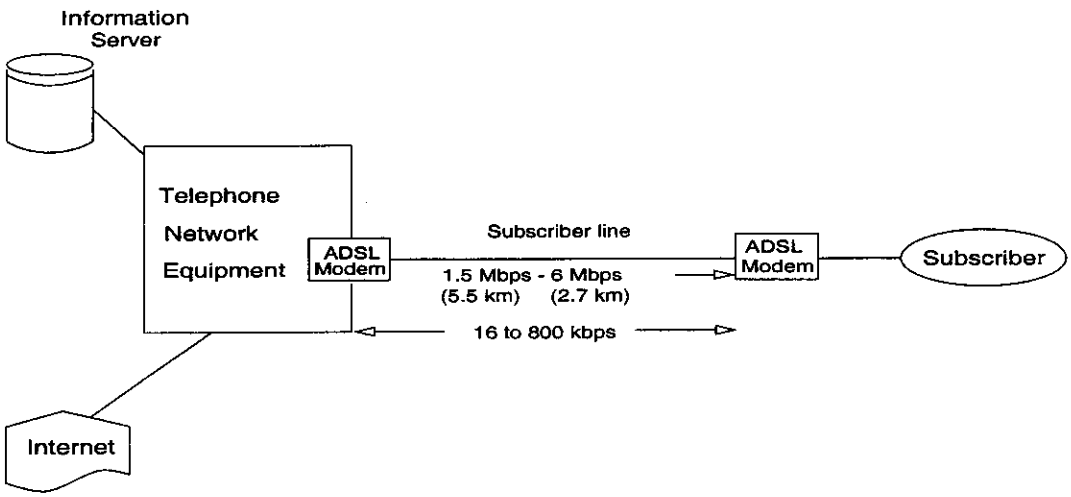


Figure 37: ADSL Channels

It is this asymmetric nature in the data transmission rates in the channels which gives rise to its name Asymmetric Digital Subscriber Line.

The downstream data rates of an ADSL channel depends on:

- length of copper line
- wire gauge
- presence of bridged taps
- cross coupled interference

In transmitting digital compressed video as a real time signal, ADSL cannot use link level or network level error control procedures commonly found in other data communication systems due to the fact that they are generally based on error recovery by re–transmission. ADLS modems therefore incorporates forward error correction.

Multiple channels are created by the ADSL modem by dividing available bandwidth of a telephone line in one of two ways (see Figure 38):

- FDM – Frequency Division Multiplexing
- Echo Cancellation

In FDM one frequency band is assigned for upstream data and another for downstream data. The downstream path is then divided by FDM into one or more channels at the desired data rates.

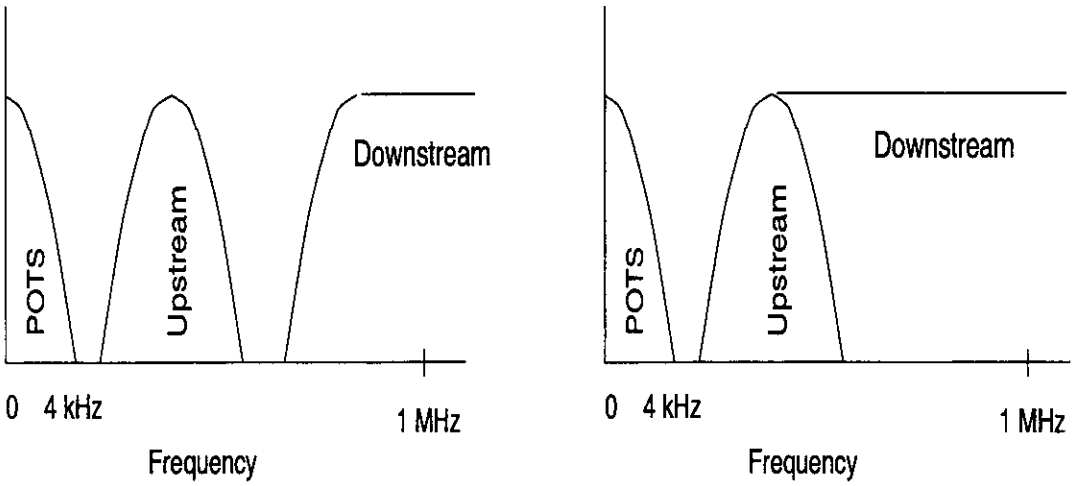


Figure 38: Creating ADSL Channels by (i) FDM and (ii) Echo Cancellation

Echo cancellation, a technology used in V.32 and V.34 modems, assigns the upstream band such that it overlaps the downstream band and separates the two by local echo cancellation.

With either technique ADSL splits a 4 KHz region for POTS at the DC end of the frequency spectrum and guarantees the availability of telephone channel even if ADSL fails.

8.5.1 ADSL Standardisation

At the present moment the following standards are applicable to ADSL:

1. ANSI T1.413 Issue I (1995) up to 6.1 Mbps
2. ETSI Annex to T1.413 to reflect European requirements
3. ANSI T1.413i2 (Issue II 1998) includes multiplexed interfaces at user end and protocols for configuration and network management.
4. ITU–T G.dmt (G.992.1) and G.lite (G.992.2) (1999) the latter allows the splitting to be done at the telephone network equipment end, at the expense of line speed.

The final step has helped moving towards vendor interoperability and service provider acceptance, further increasing ADSL deployment.

ATM Forum has recognised ADSL as a physical layer transmission protocol for UTP medium.

9 WAP - Wireless Application Protocol

WAP is designed as an application environment based on a set of communication protocols for wireless handheld devices to enable manufacturer, vendor and technology independent access to the Internet and advanced telephony services. It bridges the gap between the mobile world and the Internet as well as corporate intranets and offers the ability to deliver an unlimited range of services to subscribers independent of their network service provider or type of terminal device. This enables the mobile user to access the same wealth of information from a wireless device as they can from a desktop.

WAP is based on a secure specification that allows users to access information instantly via handheld wireless devices such as:

- mobile phones
- pagers
- two way communication radios
- smart phones
- communicators, etc.

WAP supports most wireless network technologies in use today, most notable being CDMA, GSM (Global System for Mobile communications), TDMA, CSD (Circuit Switched cellular Data), CDPD (Cellular Digital Packet Data), etc.

The operating systems that support WAP are many. Ones specifically engineered for handheld devices include:

- PalmOS
- Windows CE
- JavaOS, etc.

Although WAP supports HTML and XML, the WML (Wireless Mark-up Language) is specifically designed for small screens and one hand navigation without a keyboard.

WAP supports WMLScript (similar to JavaScript) but makes minimal demands on limited memory and CPU power of handheld devices. Browsers that are specially designed to run on WAP devices and called micro-browsers are of small file size.

9.1 WAP Specification

The WAP specifications define a set of protocols to be used in application, session, transaction, security and transport layers. These are designed to enable operators, manufacturers, and application and content providers to meet the challenges in advanced wireless service provision.

WAP utilises standard Internet components such as XML, UDP (User Datagram Protocol), TLS (Transport Layer Security) and IP. These are optimised for the mobile handheld device environment with its own unique constraints; low bandwidth, high latency and less connection stability. WAP utilises binary transmission for greater compression of data and WAP sessions are designed to cope with intermittent coverage.

The WAP content is produced using WML and WMLScript and is scaleable from two line display on a basic handheld device to a full graphic screen on smart phones and communicators.

As WAP is based on a scaleable layered architecture each layer can develop independently of others. This will also enable content providers to customise content to match client expectations and differentiate themselves from their competitors with new, enhanced information services.

While WAP users benefit from easy, secure access to Internet based information such as unified messaging, banking, and entertainment, they will also enjoy considerable freedom of choice when selecting mobile devices and network operators.

9.2 Architecture of the WAP Gateway

Figure 39 shows the layered architecture of the WAP Gateway.

- WDP – The WAP Datagram Protocol is for the transport layer that sends and receives messages via any available bearer network.
- WTLS – Wireless Transport Layer Security has encryption facilities to provide secure transport service required by secure applications such as e-commerce, e-banking, etc.
- WTP – The WAP Transport Protocol layer provides transaction support adding reliability to the datagram service provided by WDP.
- WSP – The WAP Session Protocol layer provides a light weight session layer to allow efficient exchange of data between applications.
- HTTP Interface – This serves to retrieve the WAP content from the Internet requested by the mobile device. WAP content (WML WMLScript)

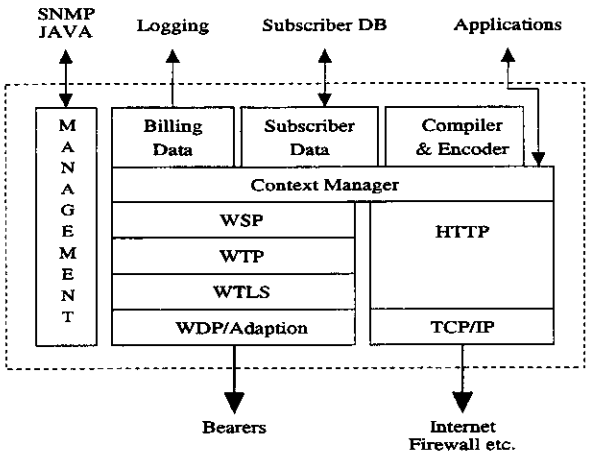


Figure 39: Layered Architecture of WAP Gateway

is then converted into a compact binary byte code for efficient transmission over the air.

The WAP microbrowser software resident within the mobile device interprets this byte code and displays the interactive WAP content.

10 Summary

Real time Transport Protocol (RTP), together with a host of other protocols facilitate the transfer of real time data streams over existing LANs and WANs based on the Internet Protocol (IP) technology. The efforts in the commercial sector had been focussed mostly towards the support of multimedia audio and video streams on PCs running Windows environments (such as Windows 3.11, Windows 95 and Windows NT). IP/TV is a strong case in point which demonstrates how fast commercial products, adhering fully to international standards, appear in the market place (RTP/RTCP on which IP/TV is based were proposed only in January 1996 and products started to appear later in the same year).

The overall success and acceptability of Ethernet as a LAN technology has led to its enhancement to high performance platforms operating at 100Mbps (Fast Ethernet) and 1000Mbps (Gigabit Ethernet). These two offer the users the familiar Ethernet operating environment with the ability to deliver high bandwidth real time data such as video and imaging to the desktop.

The technology and the tools developed are available for other real time data transfer applications, such as data acquisition, which are of interest to research scientists.

11 Bibliography

[**Black 93**] Black U, *Computer Networks, Protocols, Standards and Interface (2nd Edition)*, Prentice Hall, 1993.

[**Black 95**] Black U, *TCP/IP and Related Protocols (2nd Edition)*, McGraw Hill, 1995.

[**Comer 88**] Comer D, *Internetworking with TCP/IP: Principles, Protocols and Architecture*, Prentice Hall, 1988.

[**Comer 91**] Comer D, *Internetworking with TCP/IP - Vol I: Principles, Protocols and Architecture (2nd Edition)*, Prentice Hall, 1991.

[**Comer *et al* 91**] Comer D and Stevens D L, *Internetworking with TCP/IP - Vol II: Design, Implementation, and Internals*, Prentice Hall, 1991.

[**De Prycker 95**] De Prycker M, *ATM Solutions for Broadband ISDN (3rd Edition)*, Prentice Hall, 1995.

[**Goncalves *et al* 98**] Goncalves M and Niles K, *IPv6 Networks*, McGraw-Hill, 1998.

[**Handel *et al* 94**] Handel R, Huber M N, and Schroder S, *ATM Networks - Concepts, Protocols, Applications*, Addison-Wesley, 1994.

[**Huitema 96**] Huitema C, *IPv6: The New Internet Protocol*, Prentice Hall, 1996.

[**Jain 94**] Jain R, *FDDI Handbook - High Speed Networking using Fibre and Other Media*, Addison-Wesley, 1994.

[**Keshav 97**] Keshav S, *An Engineering Approach to Computer Networking - ATM Networks, the Internet, and the Telephone Network*, Addison-Wesley, 1997.

[**Partridge 94**] Partridge C, *Gigabit Networking*, Addison Wesley, 1994.

[**RFC 1112**] *Host Extensions for IP Multicasting*, August 1989.

[**RFC 1883**] *Internet Protocol, Version 6 (IPv6) Specification*, April 1996.

[**RFC 1889**] *RTP: A Transport Protocol for Real Time Applications*, January 1996.

[**RFC 1890**] *RTP Profile for Audio and Video Conferences with Minimal Control*, January 1996.

[**Smith 93**] Smith P, *Frame Relay - Principles and Applications*, Addison Wesley, 1993.

[**Stallings 89**] Stallings W, *Handbook of Computer-Communications Standards Vol 3: The TCP/IP Protocol Suite (2nd Edition)*, Howard W. Sams, 1989.

[**Stallings 93**] Stallings W, *Local and Metropolitan Area Networks (4th Edition)*, Macmillan, 1993.

[**Stallings 94a**] Stallings W, *Data and Computer Communications (4th Edition)*, Macmillan, 1994.

[**Stallings 94b**] Stallings W, *Advances in Local and Metropolitan Area Networks*, IEEE Computer Society Press, 1994.

[**Stallings 95**] Stallings W, *ISDN and Broadband ISDN with Frame Relay and ATM (3rd Edition)*, Prentice Hall, 1995.

[**Tanenbaum 96**] Tanenbaum A S, *Computer Networks (3rd Edition)*, Prentice Hall, 1996.

[**Wilder 93**] Wilder F, *A Guide to the TCP/IP Protocol Suite*, Artech House, 1993.