Seventh College on Microprocessor-Based Real-Time Systems in Physics

Abdus Salam ICTP, Trieste.

October 28 – November 22, 2002

Lecture Notes

Contents

Chapter 1

Software Design
by Paul Bartholdi

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.

1.1 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**.

$\boxed{\textbf{Good programming} + \textbf{Good documentation} = \textbf{lower total cost}}$

1.1.1 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.

1.1.2 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

1.1.3 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.

1.1.4 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

1.1.5 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

1.1.6 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.

1.1.7 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

1.1.8 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.

1.1.9 General Index

Goal: Find information anywhere in the documentation.

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

1.1.10 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

1.1.11 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}
\index{example!literate programming}

@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;
```

1.2 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?*

1.2.1 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.

1.2.2 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?

1.2.3 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 . . .

1.2.4 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

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?

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?

1.2.5 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` ...

1.2.6 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 !

1.2.7 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!"

1.3 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.

1.3.1 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.

1.3.2 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
```

or

```
prog < test.data | tee test.results | less
```

While every module has only one *standard input*, it has effectively two output ones: the *standard output* and the *standard error output*. The output redirection sign (>) followed by an ampersend (&) merge the two output strams into the given file. Similarely, the pipe sign (|) followed by an ampersend merge the two output streams into a single one as input to the following programme. Finaly, if the variable `noclobber` is defined, then the redirections will not overwrite old existing files, except if the redirection operator is followed with a ! sign,

Here is a summary of all redirections and pipes:

<	file	redirect *file* into standard input
<<	"Word"	redirect the following lines, until the one starting with "Word"
>	file	redirect standard output to *file*
>>	file	append standard output to the end of *file*
>&	file	redirect standard output and standard error to *file*
>>&	file	append standard output and standard error to the end of *file*
>!	file	same as above, but ignore *noclobber*
>&!	file	same as above, but ignore *noclobber*

1.3.3 Five ways to input data into a programme

a) type data by hand
b) `prog < data-file`
c) `cat data-file | prog`
d) `echo "data" | prog`
e)
```
cat << FOF | prog
...
   data
...
FDF
```

1.3.4 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. If it is not yet the case, put this alias in your `.tcshrc` or `.bashrc` file. `ls` will then automaticaly append a "`*`" at the end of executable files, a "`/`" at the end of directories and a "" at the end of symbolics links.

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 xx.

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.

1.3.5 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 (*tcsh*) or functions (*bash, ksh*):

```
        alias    help  "egrep -ih \!* ~/.help ./.help"
        function help(){egrep -ih $1  ~/.help ./.help}

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

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

`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'
```

1.3.6 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).

1.3.7 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
```

1.3.8 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
```

1.3.9 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

1.3.10 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.

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*>

1.3.11 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 1.8.4.

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).

1.3.12 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.

`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.

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.

1.4 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).

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*.

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
```

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` .

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"
```

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*.

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
```

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 1.4.1 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.

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
```

1.4.1 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`

ksh	csh
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

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.

Here is a full list of signals as used in LINUX. It is extracted from the file /usr/src/linux/include/asm/signal.h

```
#include <linux/types.h>

#define SIGHUP  1
#define SIGINT  2
#define SIGQUIT  3
#define SIGILL  4
#define SIGTRAP  5
#define SIGABRT  6
#define SIGIOT  6
#define SIGBUS  7
#define SIGFPE  8
#define SIGKILL  9
#define SIGUSR1 10
#define SIGSEGV 11
#define SIGUSR2 12
#define SIGPIPE 13
#define SIGALRM 14
```

```
#define SIGTERM 15
#define SIGSTKFLT 16
#define SIGCHLD 17
#define SIGCONT 18
#define SIGSTOP 19
#define SIGTSTP 20
#define SIGTTIN 21
#define SIGTTOU 22
#define SIGURG 23
#define SIGXCPU 24
#define SIGXFSZ 25
#define SIGVTALRM 26
#define SIGPROF 27
#define SIGWINCH 28
#define SIGIO 29
#define SIGPOLL SIGIO
/*
#define SIGLOST 29
*/
#define SIGPWR 30
#define SIGSYS 31
#define SIGUNUSED 31

/* These should not be considered constants from userland.  */
#define SIGRTMIN 32
#define SIGRTMAX (_NSIG-1)

```

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. They are written in `ksh` or `bash`, but are easily converted to `csh`.

arithmetic	calc calc2 guess1 guess2 minutes
awk	KillKillMeAfter
loops	convert convert2 flist tolower toupper
select	term1 term2
case	convert minutes term2
if	KillKillMeAfter KillMeAfter 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 ˆC

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
```

1.5 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.

1.5.1 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
. . .		

Thousend 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`

1.5.2 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 xli).

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.

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.

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 xli 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`

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".

1.6 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.

1.6.1 File transfer

File transfer between two computers can be done with the program `ftp` (`f`*ile* `t`*ransfer* `p`*rotocol*)

`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.

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.

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.

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.

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.

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 . . .).

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.

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.

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`.

1.6.2 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.

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.

1.6.3 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"
```

1.6.4 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.

1.6.5 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.

1.6.6 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.

1.7 Data structures

Data structures can be classified into two main categories: linear and nonlinear. 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.

1.7.1 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 1.1.

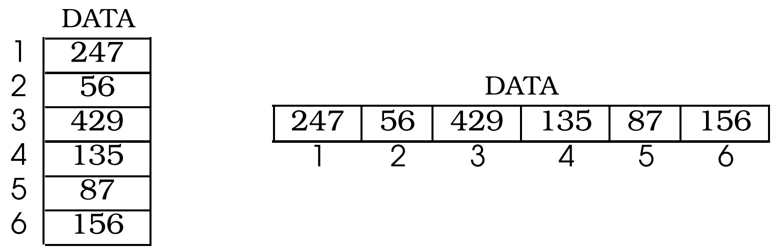


Figure 1.1: Logical pictures of array DATA.

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 1.2 on page l shows the internal representation of an array AUTO, with *BA=200*, *LB=1932*, and *w=4*.

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, 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 1.3 illustrates this by inserting the value “Ford” in a string array at position 3.

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

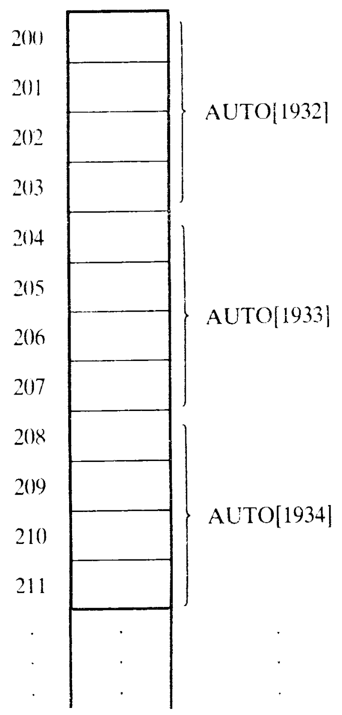


Figure 1.2: Memory representation of array AUTO.

	NAME		NAME
1	Brown	1	Brown
2	Davis	2	Davis
3	Johnson	3	Ford
4	Smith	4	Johnson
5	Wagner	5	Smith
6		6	Wagner
7		7	
8		8	

Figure 1.3: Insertion of an element in an array.

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:

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

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. 1.4 on page lii) 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	$\boxed{44}$	33	11	90	40	22	88	$\underline{66}$
Comparison 2	$\boxed{44}$	33	11	90	40	22	$\underline{88}$	66
Comparison 3	$\boxed{44}$	33	11	90	40	$\underline{22}$	88	66
Swap 1	$\underline{22}$	33	11	90	40	$\boxed{44}$	88	66
Comparison 4	22	$\underline{33}$	11	90	40	$\boxed{44}$	88	66
Comparison 5	22	33	$\underline{11}$	90	40	$\boxed{44}$	88	66
Comparison 6	22	33	11	$\underline{90}$	40	$\boxed{44}$	88	66
Swap 2	22	33	11	$\boxed{44}$	40	$\underline{90}$	88	66
Comparison 7	22	33	11	$\boxed{44}$	$\underline{40}$	90	88	66
Swap 3	22	33	11	$\underline{40}$	$\boxed{44}$	90	88	66
		subarray 1					subarray 2	

Figure 1.4: 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 $N\log(N)$.

1.7.2 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.

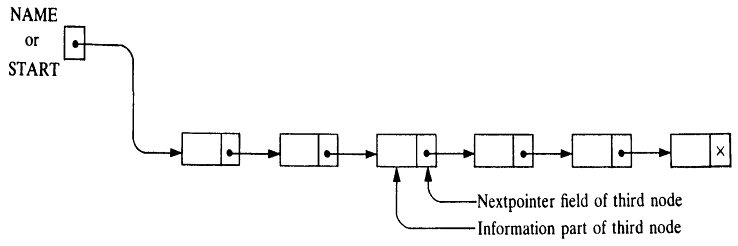

NAME
or
START
Nextpointer field of third node
Information part of third node

Figure 1.5: 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 1.5 and 1.6. The information part may be further subdivided, as seen in figure 1.6 (page liv). 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.

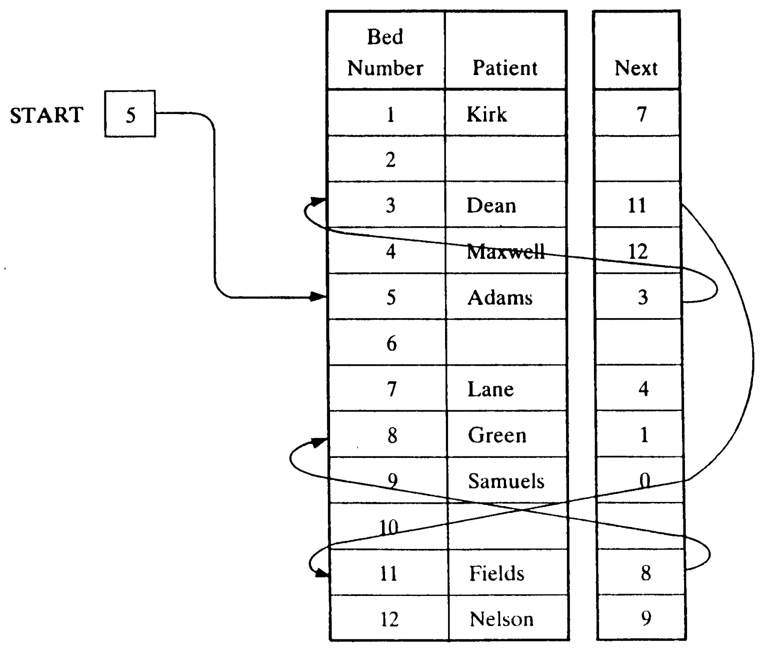
Operations on linked lists

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 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*.

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*.

The insertion of a node between nodes *A* and *B* of a list (fig. 1.7) is done by removing the first node of *AVAIL* and storing its address in an auxil-



Bed Number	Patient	Next
1	Kirk	7
2		
3	Dean	11
4	~~Maxwell~~	12
5	Adams	3
6		
7	Lane	4
8	Green	1
9	Samuels	0
10		
11	Fields	8
12	Nelson	9

Figure 1.6: Vertical representation of a linked list.

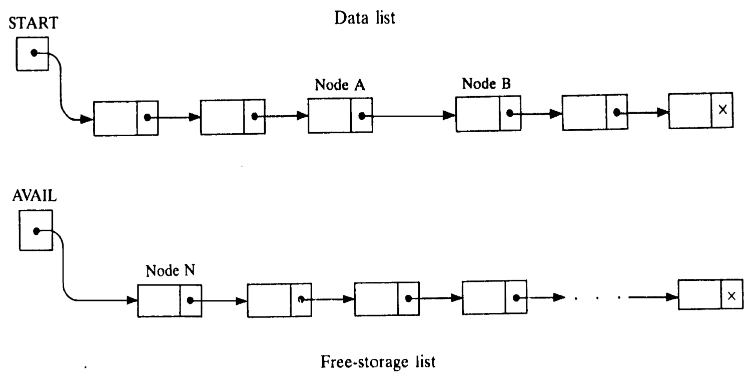


Figure 1.7: Linked list before an insertion.

iary 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 1.8. 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

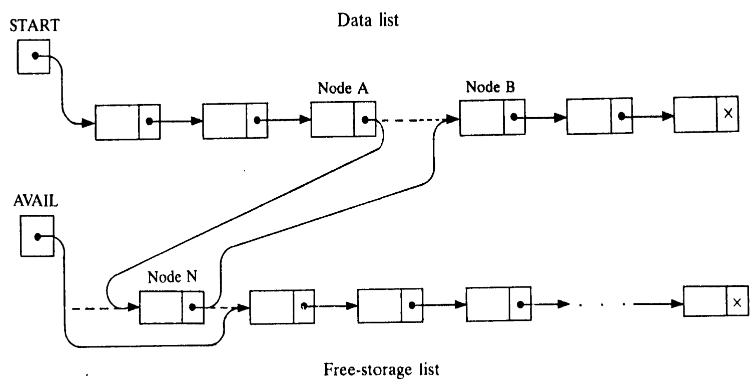


Figure 1.8: Linked list after an insertion.

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. 1.9). 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.

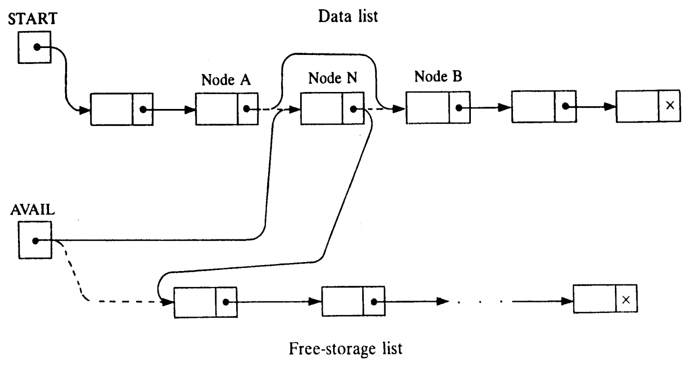


Figure 1.9: Deletion in a linked list.

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.

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 (1.7.1) will be suitable for a linked list, but the quicksort algorithm (1.7.1) will need the particular properties of a two-ways list (1.7.2).

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

Particular lists

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

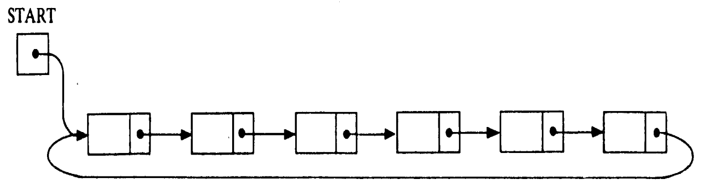


Figure 1.10: Circular linked list.

A *circular list* (fig. 1.10) 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.

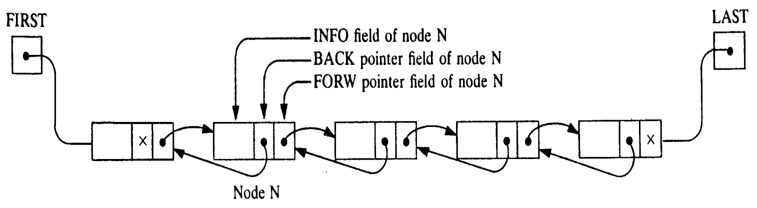


Figure 1.11: Two-ways linked list.

A *two-ways list* (fig. 1.11) 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.

1.7.3 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 1.12.

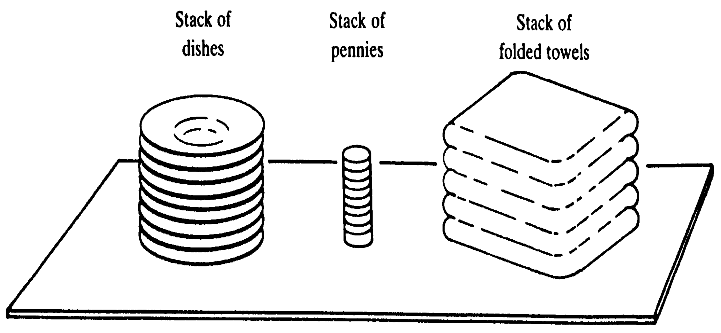


Figure 1.12: Every-day's life stacks.

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. 1.13). As the top is the only 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*.

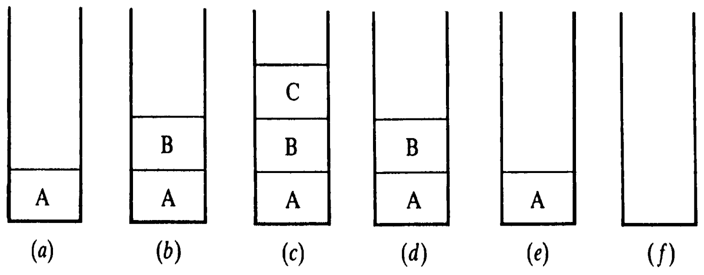


Figure 1.13: Stack push and pop operations.

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 (1.7.1). 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.

1.7.4 Queues

A *queue* is another familiar concept (fig. 1.14). 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*.

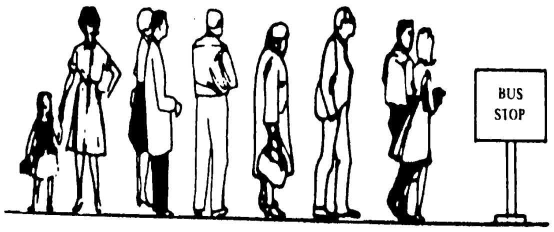


Figure 1.14: Familiar queue.

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.

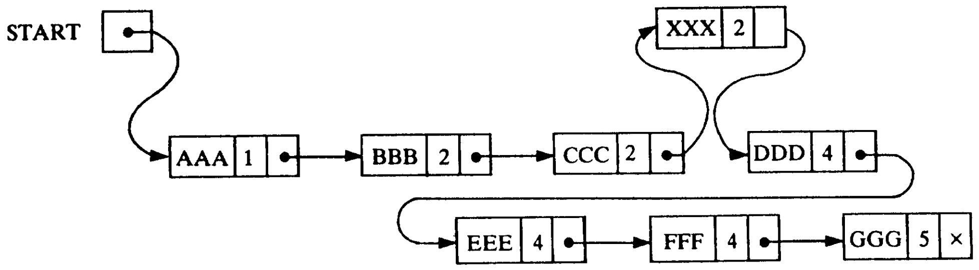


Figure 1.15: Representation of a priority queue implemented as a list.

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 1.15, 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).

1.8 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.

1.8.1 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 mono-processor 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 1.16 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.

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.

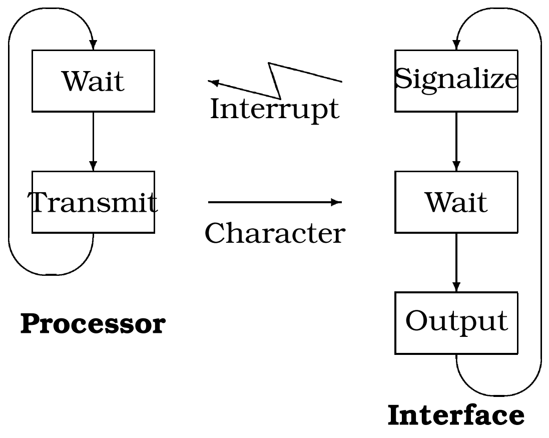


Figure 1.16: Respective activities of processor and terminal interface for writing a character.

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.

1.8.2 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 systems 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.

1.8.3 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 multitasking 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.

1.8.4 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

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

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

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

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

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.

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. 1.17a) 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. 1.17b), 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.

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.

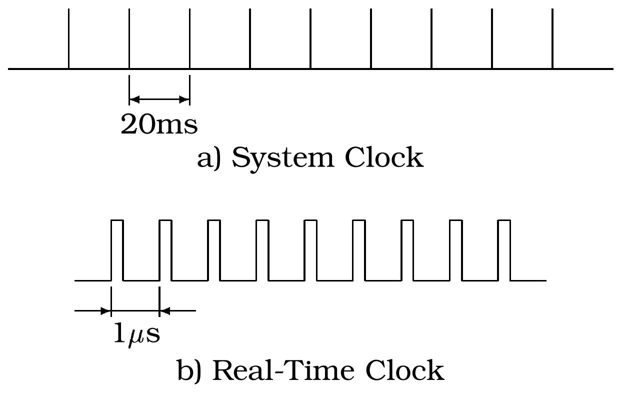


Figure 1.17: Different clocks are defined in a system.

Both approaches have their drawbacks:

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

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.*

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}(condition 1)	is the starting time of the process.
	T_{end}(condition 2)	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_{begin}(condition 1)	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_{end}(condition 2)	is the completion time of the process.ls -l

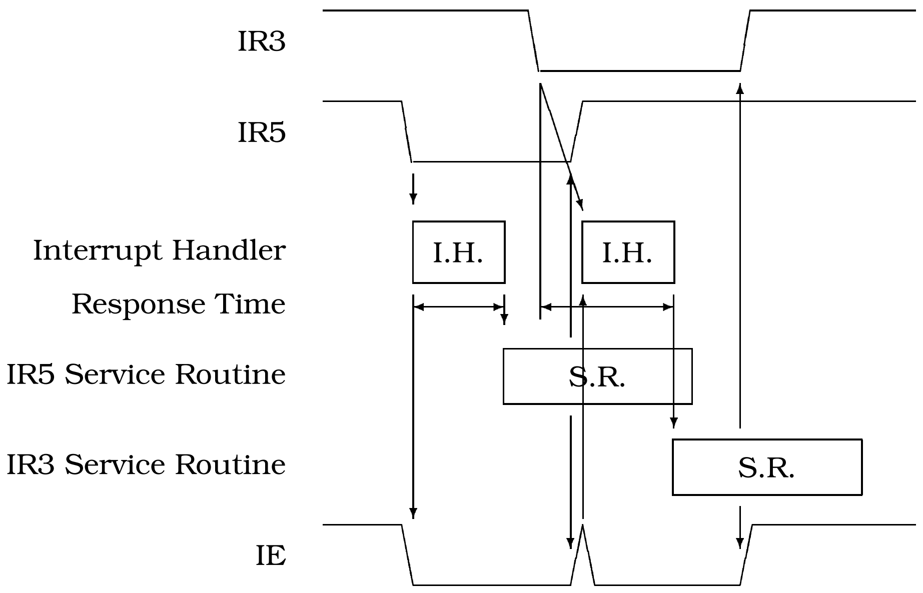


Figure 1.18: Interrupt Service Scheme

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 1.18, 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.

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.

1.8.5 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.

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*.

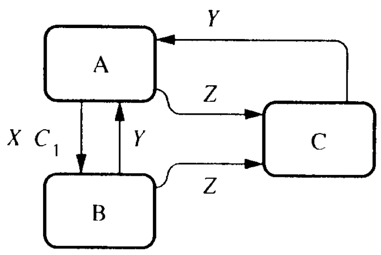


Figure 1.19: State-chart example.

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 1.19).

States can be decomposed to lower level states or combined into a higher level state (figure 1.20). These operations are called *refinement* and *clustering*. Zooming in and out (figures 1.21) enables one to have different levels' views of the system.

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.

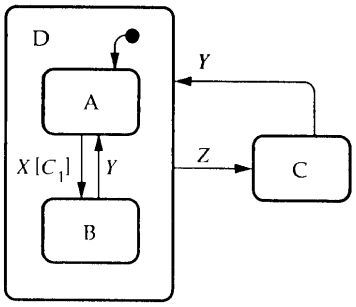


Figure 1.20: Clustering states in a state-chart.

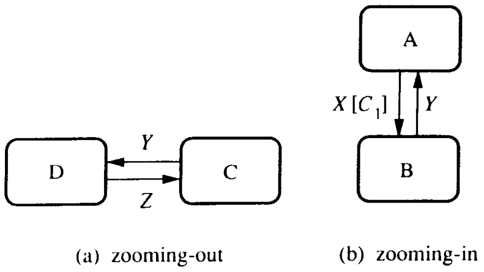


Figure 1.21: Zooming in and out.

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 x 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

$$
\begin{aligned}
C &= (P,T,I,O)\\
P &= |p_1,p_2,p_3,p_4,p_5|\\
T &= |t_1,t_2|\\
I &= \begin{array}{|ccccc|c} \multicolumn{1}{c}{p_1} & p_2 & p_3 & p_4 & \multicolumn{1}{c}{p_5} & \\ 1 & 1 & 2 & 0 & 0 & t_1\\ 0 & 0 & 0 & 0 & 1 & t_2 \end{array}\\
O &= \begin{array}{|ccccc|c} \multicolumn{1}{c}{p_1} & p_2 & p_3 & p_4 & \multicolumn{1}{c}{p_5} & \\ 0 & 0 & 0 & 1 & 2 & t_1\\ 0 & 0 & 1 & 0 & 0 & t_2 \end{array}
\end{aligned}
$$

is represented by the graph of figure 1.22

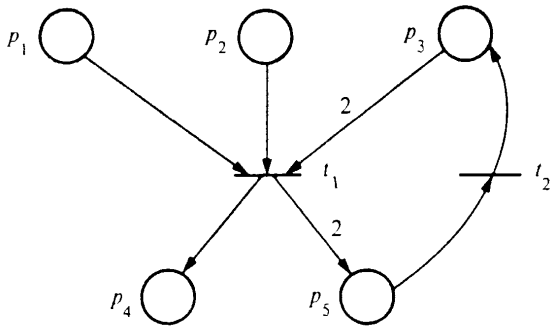


Figure 1.22: Petri graph with weighted arcs

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. 1.23).

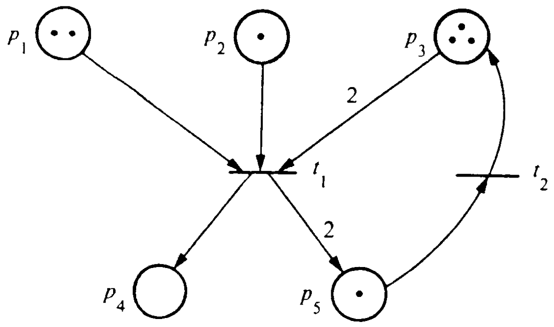


Figure 1.23: Marked Petri graph

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.

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 1.24 shows the result of firing transition t_1 in figure 1.23.

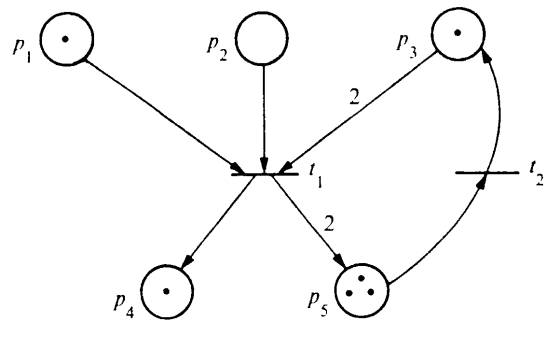


Figure 1.24: Petri graph after the firing of t_1.

A Petri net may be annotated as shown in the figure 1.25 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).

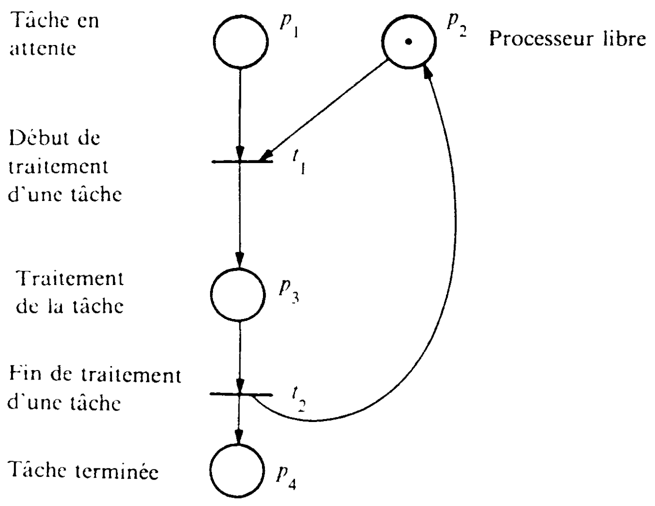


Figure 1.25: Petri net modelizing a processor allocation.

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 1.26-1.29 show Petri nets representing some real-time issues.

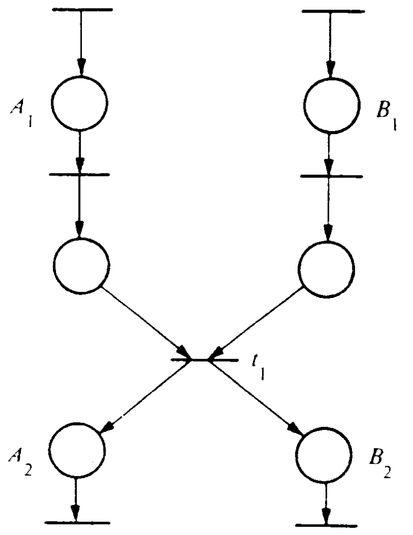


Figure 1.26: Petri net modelizing a *rendez-vous* type synchronization.

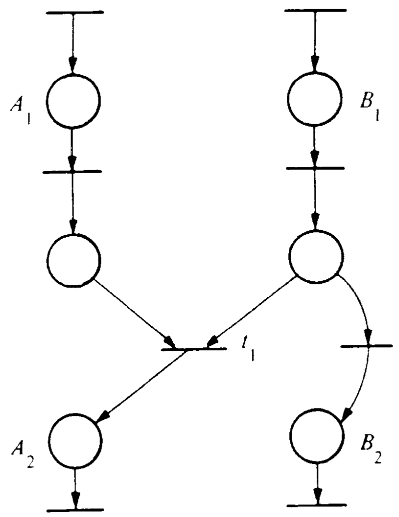


Figure 1.27: Petri net modelizing a *mailbox* type synchronization.

The Petri net model may be used by the designer in a kind of top-down structured approach (figs. 1.30-1.33):

- Start with a global Petri net model of the system (fig. 1.30).
- Stepwise refine it by substituting (fig. 1.32) the transitions by *well-formed blocks* (fig. 1.31). A well-formed block should have only one input and one output (fig. 1.33).

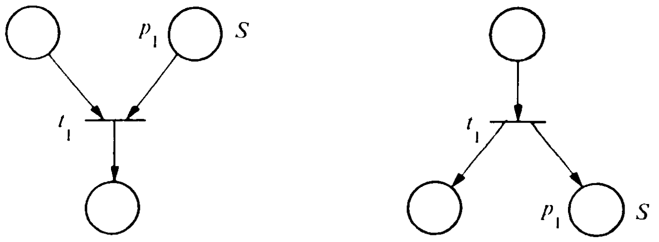


Figure 1.28: Petri net modelizing the semaphores primitives *P(s)* (left) and *V(s)* (right).

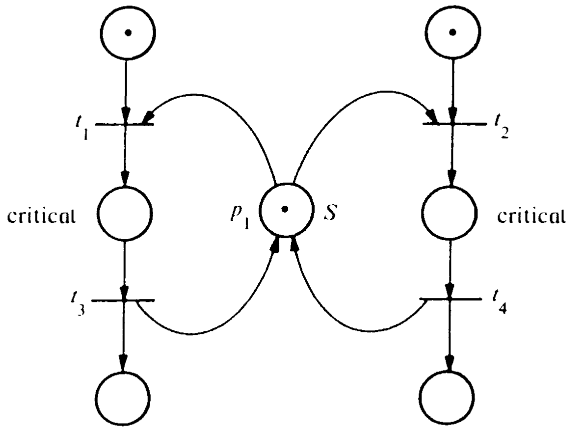


Figure 1.29: Petri net modelizing mutual exclusion by semaphore.

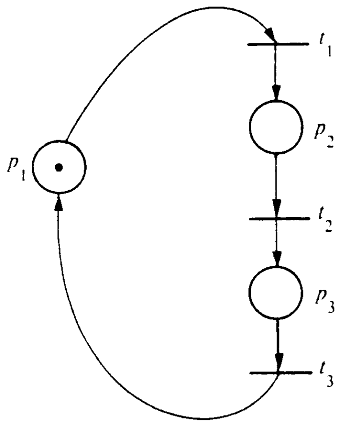


Figure 1.30: Initial step for structured design.

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. 1.34 and 1.35).

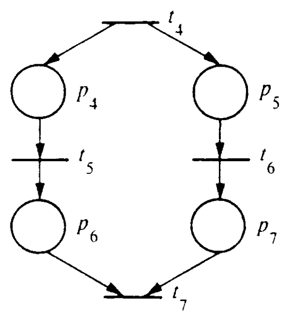


Figure 1.31: Block example.

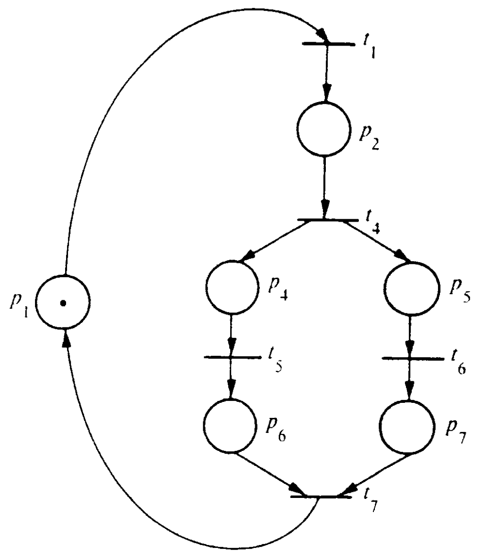


Figure 1.32: Replacement of t_2, p_3, t_3 in fig. 1.30 by the block of fig. 1.31 .

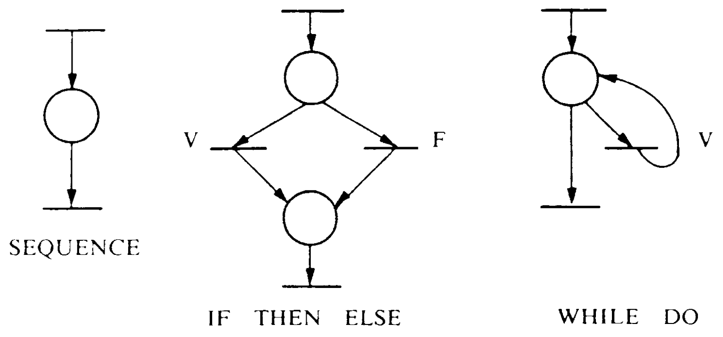


Figure 1.33: Well-formed blocks.

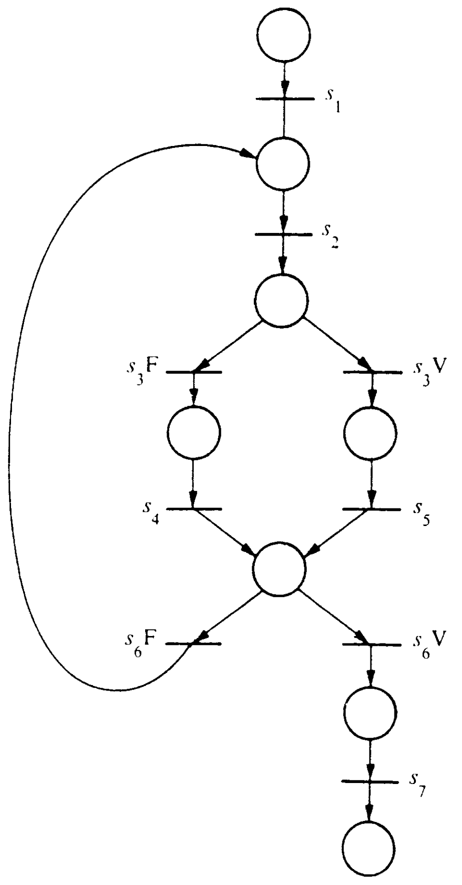


Figure 1.34: Petri net example.

1.8.6 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.*

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.

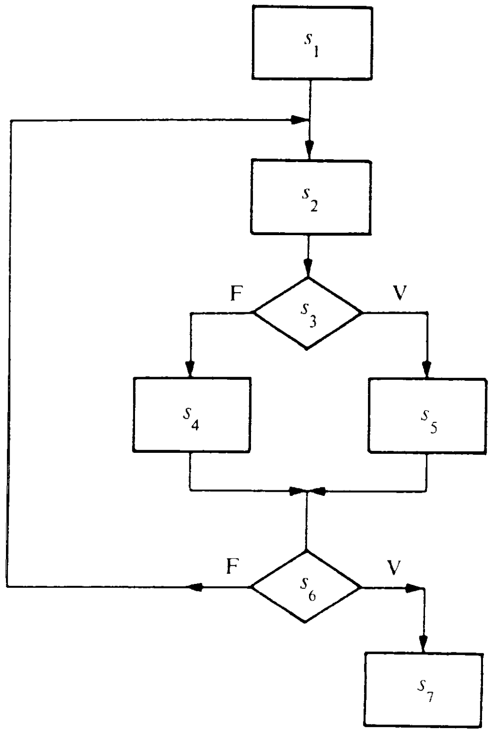


Figure 1.35: Flowchart for the Petri net of the figure 1.34.

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.

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 1.36 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*.

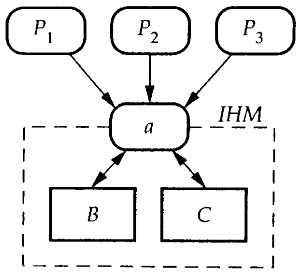


Figure 1.36: IHM module

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

1.8.7 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:

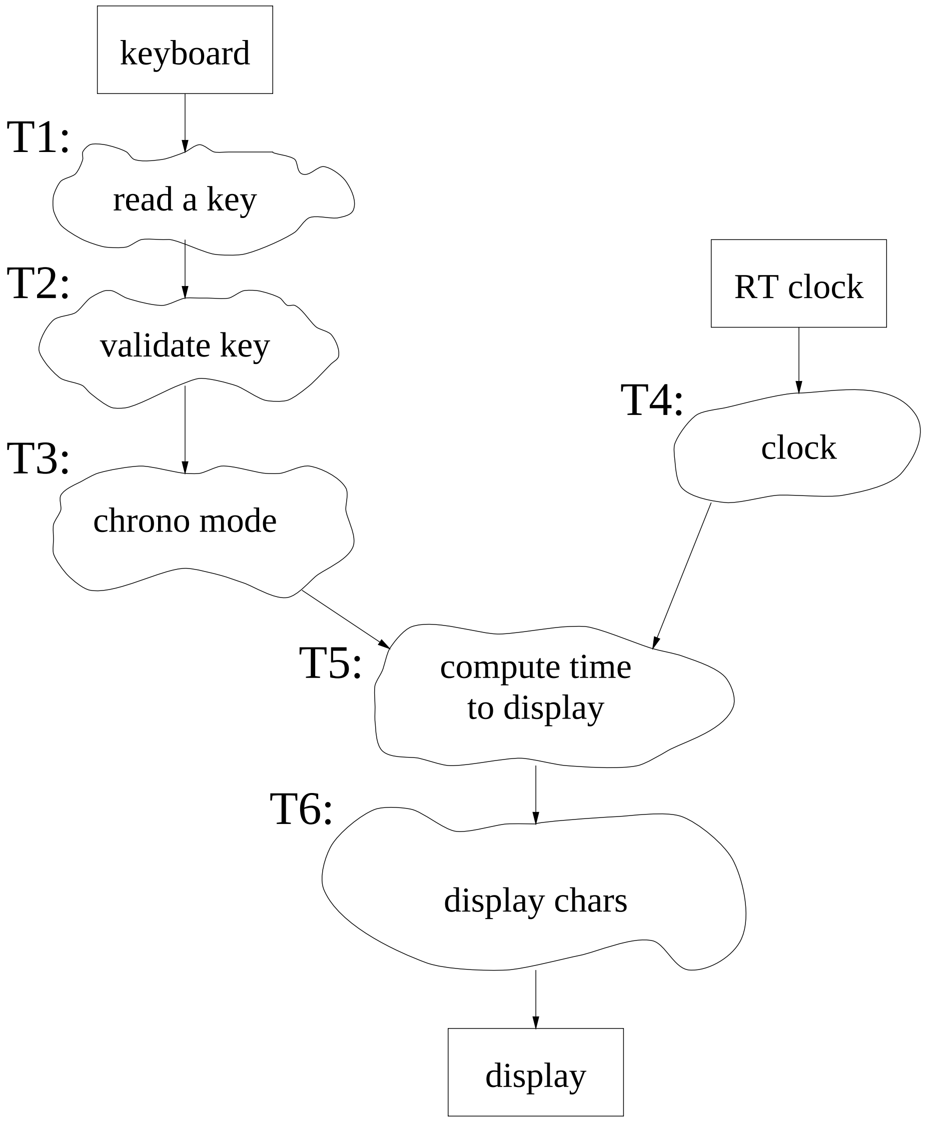
W	: s.chrono→s.watch	mode selection
C	: s.watch →s.chrono	mode selection
H	: s.watch →increment	hours
M	: s.watch →increment	minutes
S	: s.watch →increment seconds	
G	: s.watch →r.watch	
T	: r.watch →s.watch	
R	: r.watch →e.chrono	
I	: r.chrono→i.chrono	
F	: r.chrono→f.chrono	

where:

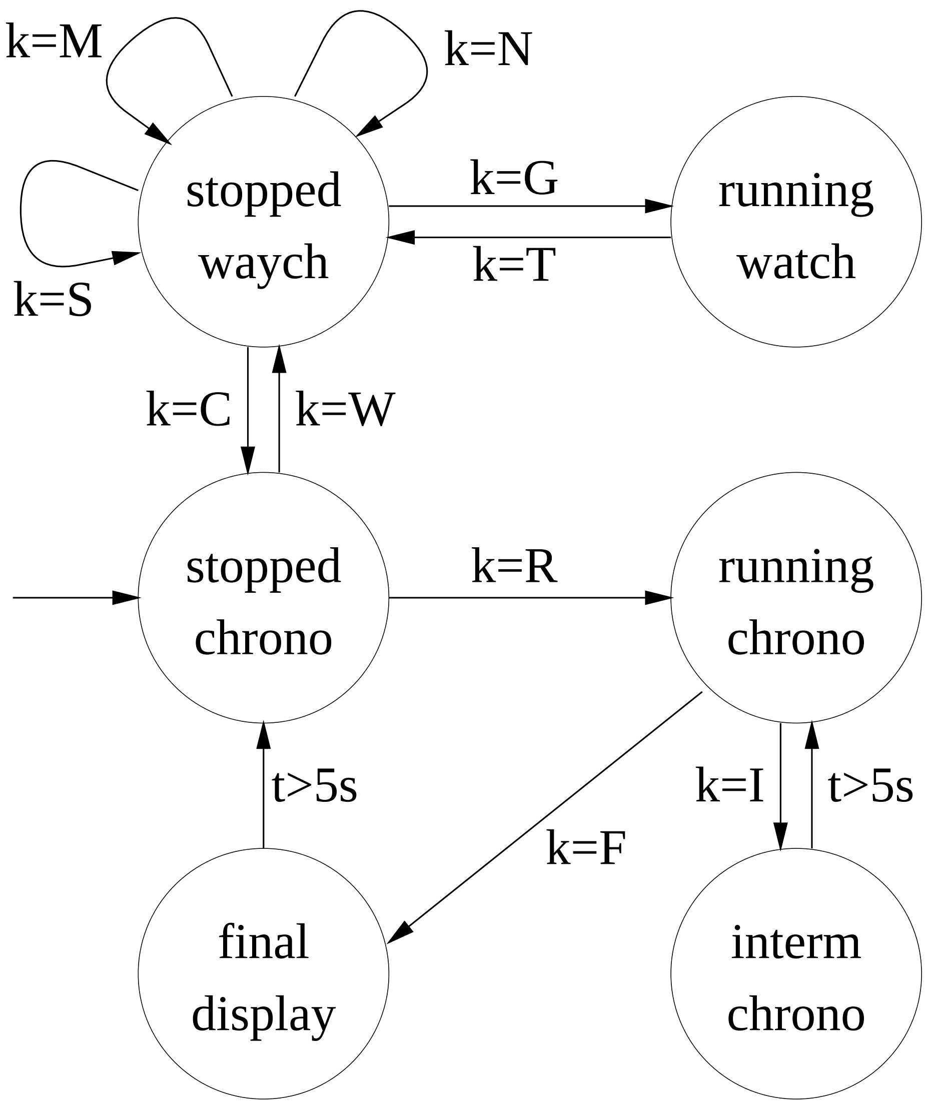
s.watch, s.chrono	: stopped watch and chrono modes
r.watch, r.chrono	: running watch and chrono modes
i.chrono	: intermediate display for 5 seconds
f.chrono	: final display for 5 seconds

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

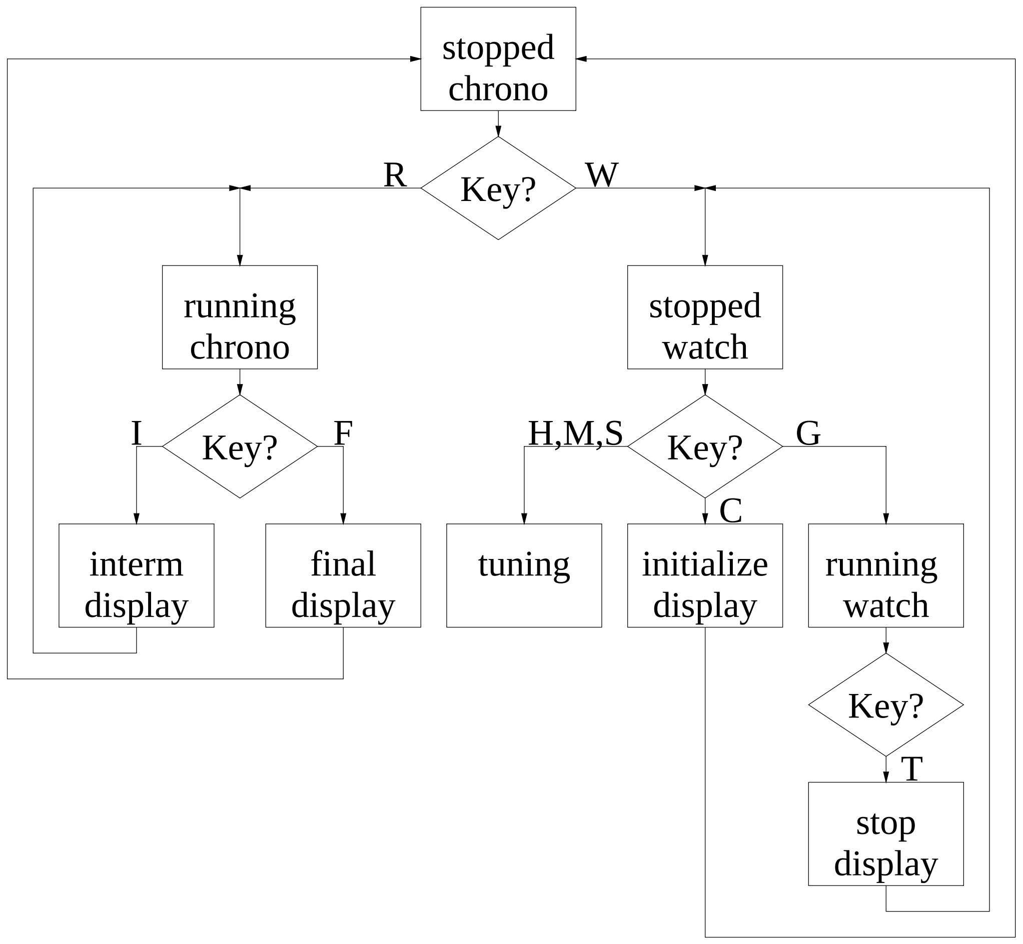
Data flow study



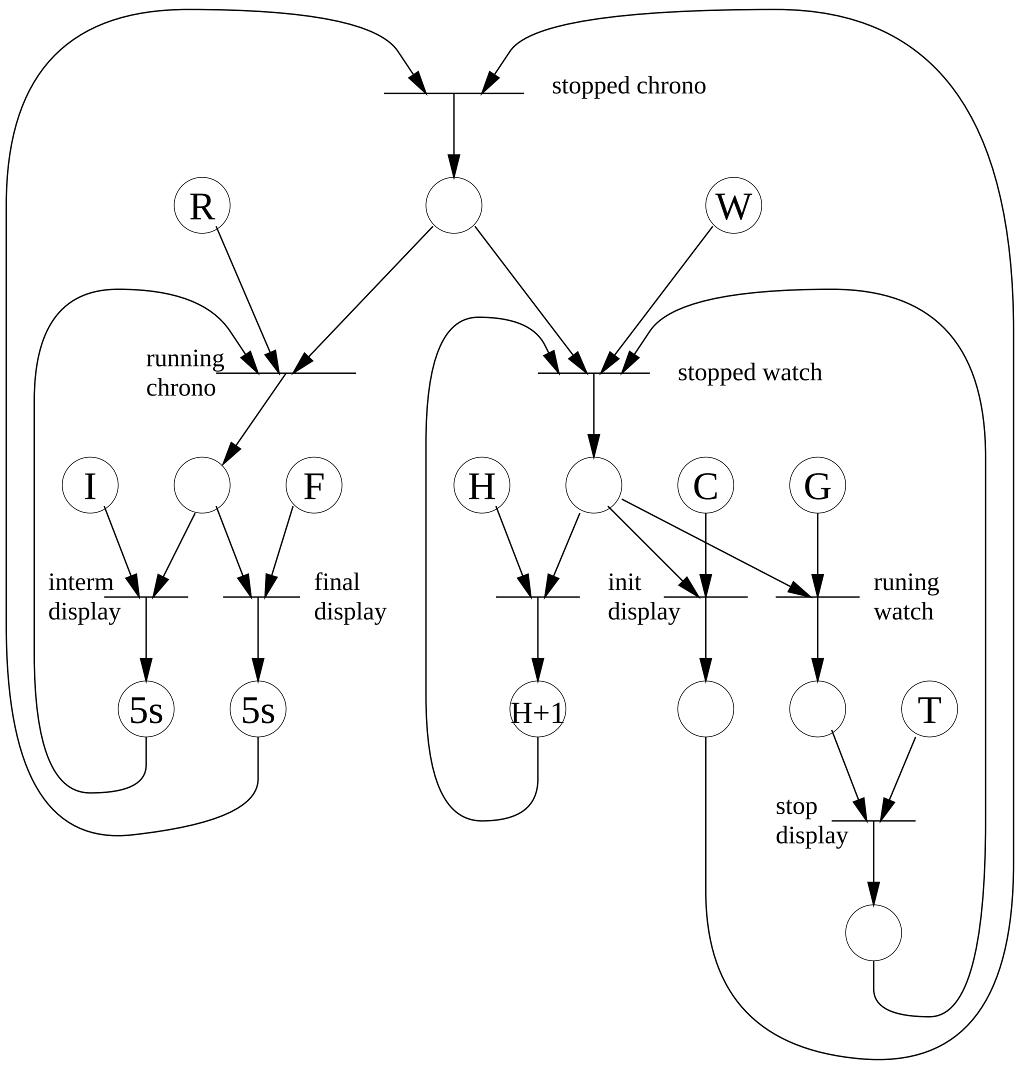
State chart study



Flow chart study



Petri Net study



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.

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

1.9.1 `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>
```

1.9.2 `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.

1.10 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

1.11 Introduction

In general, we can consider a computer as a box with some input channels, some output channels, probably local storages devices and links to other computers.

Nowadays, operations are going very fast, even on samll computers, and data storages (disk, cd, dvd) have huge capacities in the order of 10^{10-11} Bytes. If we assume 10^4 characters on an A4 piece of paper (a rather full page), then a small 10 GB disk corresponds to 10^6 pages, or two metric tons of paper.

The organizations of data in such a way that they can be classified and retrieved rapidity is not new. For centuries, librarians have been very good for this job. More recently, the punched cards were invented at the turn of last century to deal with the American census data. Pencil and paper were not sufficient anymore.

But this simple system of book catalogs or punched card sets is not sufficient, neither well adapted as data storage devices in computers.

From the early fifties until now, a lot of work on this has been done, many clever data organization systems developed and made available, even for small computers.

It is interesting to remember that IBM thought they would eventually sell 10 copies of their first electronic computer. This number would be sufficient for decades. . . It had a few KB of memory, no disk, but tapes with a capacity of a few MB.

1.11.1 Why Data Base Systems ?

A fundamental aspect of computers is their neutrality. Without changing anything in the hardware, they can be used for many different types of work, sometimes even at the same time. Data can be flags, numbers, words, phrases, codes, images, sounds etc. These data are not independent. Relations exist between many of them.

Think of music sound files. They are related to the composer, the CD editor, the interpreter, the place on your hard disk and in you book shelves, the date of purchase etc. What about finding the address of the interpreter, or a list of composers born in the same year as the one in your file?

What we want is a system that is sufficiently general to deal with all kinds of data, that is easy, fast and secure to use for storing new data, keeping them uptodate and retrieving them with their relations.

1.11.2 What is a Data Base System ?

A Data Base System is a set of programs that can be used according to what is described in the previous paragraph. In many cases, they interact with the users through a single interface hiding their complexity effectively. Some even go as to replace entirely the operating system (The PIC relational computer system or the R38 computer of IBM for example). At the other extreme, the system appears effectively as a large set of programs/commands that can be combined in the normal operating system (/rdb for example). We will see later (Section 1.13.4, page xciv) the relative advantages of both approaches.

Data Base Systems can be local on a single machine, or distributed on many different ones, either in a hierarchical or egalitarian way. They can interact in real time or in pseudo batch mode (night update for example).

The fundamental aspects of the Database system are that it hides completely from the users the way the data are organized physically in the computers, and its ability to furnish a single and uniform way to interact with the data.

1.11.3 What is a Data Base ?

A Data Base is the set of all data that are somehow related.

A single Data Base system can contain many Databases that are not related. For example, my collection of books, with their authors, title, editors, date of publishing etc, has probably nothing to do with the meteorological data I keep for many years. They are two distinct Data Bases in my single Data Base System.

1.11.4 Database Models

As we have seen, a Database is not just data, but also the relations that exist between them.

In the world, we can recognize three or four types of relations:

hierarchical A company is made of divisions, divisions of workshop etc.

network A company has contractors and sales agents who are themselves. . .

relational A book is defined by its title, author, editor etc, the author by its names, age, location etc.

object A melody is defined as a sheet of paper, its composer, author etc, and by all what can be done with it: singing, recording, printing. . .

They are all part of the reality, and Data Base Systems have been built in the past modeling more or less all four types. But today most work is done with relational model, Object (Relational) Databases being still in their infancy.

1.12 The Relational Model

1.12.1 Historical background

Historically, the Relational Model came up relatively late, mainly with the two marking papers of Codd (see 1.14.7, page xcix). Codd was working as a mathematician in IBM research laboratories. His goal was to setup a system based on simple and mathematically coherent rules. It should avoid all the pitfalls of the hierarchical and network data bases. In particular, the system would be very easy to interrogate and resilient to all kinds of software and hardware problems.

The implementation of the concepts proposed by Codd was rather slow. They were too far advanced for the hardware available at that time. They were also given in a mathematically oriented vocabulary that discouraged many potential users or developers.

Among the cornerstones in the development of usable relational data bases, we should notice the definition of a single "structured query language" (SQL) by IBM, the hardware implementation of the concepts in the system R38 of IBM, the PIC operating system based entirely on the relational model and the work of Stonebraker ((see section 1.14.7, page xcix)) to develop a full blown relational system on a mini computer. The last one became the INGRES system, soon joined by ORACLE on the same ground.

Around 1985, a few people (see 1.14.7, page xcix) recognised that the relational model and the UNIX pipes and redirections fitted very well together, permitting the elaboration of a completely open relational data base system. Sadly, this was never very popular outside a small circle of users, enthusiastic by its simplicity of usage.

Many, commercial or not, Relational Data Bases are now available, for example IBM DB2, Informix, Ingres, Oracle, Postgres, Sybase, /rdb, mSQL, MySQL, PostgreSQL . . .

1.12.2 The relational model

Vocabulary

The work of Codd is fundamental, but it uses a vocabulary more oriented toward mathematician than database users. So let us start with a small table of equivalent words :

Formal relational term	Informal equivalent
relation	table
tuple	record, row
attribute	field, column
primary key	unique identifier

1.12.3 The relational system

C. J. Date (see section 1.14.7, page xcix) characterises a relational system by :

1. The data are perceived by the user as tables, and nothing but tables. The user should not necessarily be aware of the physical representation of the data.
2. The operators at the user's disposal are operations that generate new tables from old ones. The three main operators are **selection**, **projection** and **join**.

Codd himself goes into more details :

- represent all information as tables;
- keep the logical representation of data independent from its physical storage characteristics;
- use a high-level language for structuring, querying and changing the information in the data base;
- support the main relational operations (selection, projection and join), and set operations as union, intersection, difference and division;
- support views, which allow the user to specify alternative ways of looking at data in tables;
- differentiate unknown and zero or blank data;
- support mechanisms for security and authorization;
- protect data integrity through (atomic) transactions and recovery procedures.

1.12.4 Data entities and relations

Relation are contained in a homogeneous table, and each line conveys the information for one entity. As an example, take a table with all participants. It contains a set of columns like PIN, Name, FirstName, Address etc. For each participant, each entity, there is a line with his PIN, his name, first name, address etc. A data element, or value, is at the intersection of a row and a column. It can be "valid data", or "Unknown" or "Empty" (`NULL` value).

Each row must be identified by a **unique** identifier, called the **primary key**. In our case most probably the PIN, though on many occasions the Name or a combination of Name-FirstName could be easier to work with.

A database is made of a set of related tables. Most will contain data prepared by the user, but the system will also keep its information concerning the database in its own tables that can be accessed in the same way by the user, all information are treated as tables, as Codd says.

We will see later in this section how to organize the data into tables.

One more thing: Codd and Date insist rightfully on the independence of the physical implementation of the tables. We said that they can be considered as files, but in most relational systems the user will never be able to see them as such. The only exception is the UNIX flat tables described in the next section. The logical design of tables should also be independent of the user's view of them.

1.12.5 The Data model

Here we will elaborate a little further on the organization of data into tables.

1. make a general overview of all the data you want to put into the database. Look for what is related though not originally intended to be part of the database.
2. make a list of entities with their properties or attributes; mark those properties that belongs to more than one entity, those that are effectively transformations of another property.
3. make sure that each entity has an attribute (or a group of attributes) that you can use to uniquely identify any row in the future table. If the *logical* primary key (Name above for example) is possibly not unique, then look for a different attribute (PIN, ISBN number etc.) or create an artificial one (record number for example) if necessary.
4. consider the relationships between the entities. They can be of three forms:

   one-to-one: for each entry in a table there is a corresponding one in another table, and the converse is true;

   one-to-many: for each entry in a table there is a corresponding one in another table, but there are possibly many entries in the first table corresponding to one entry in the second one;

   many-to-many: for each entry in a table there are many entries in another table and the converse is also true.
5. apply the normalisation rules to eliminate all one-to-one and many-to-many relationships.
6. verify the coherence of the system and its adequacy to the available data and foreseen queries.

1.12.6 Entity-Relationship diagrams

A convenient way to look at table organization is the entity-relationship (E-R) diagrams. The usual convention is to display each table as a box with columns listed inside. This could be done with paper and pencil during the early development phase, then automatically produced by the system.

Participants	**Countries**	**Cities**
Name	Country_Name	City_Name
FirstName	Capital	Country
Country	Male	Altitude
Year	Female	Longitude
Phone	Ratio	Latitude
Email	Groth	Population
	Population	

The second step is to mark (underline, using bold) the primary key for each table. On many occasions, it will consist of effectively adding a new column that is unique (The “Name” above is surely not a unique identifier, so we need to add a PIN column).

The third step is to mark with arrows the relationships between entities in different tables, without arrow head in the case of *one-to-one*, single arrow head for *one-to-many* and double arrow heads for the *many-to-many* case.

The fourth step consists of resolving the *one-to-one* and *many-to-many* relationships (see also the normalisation rules below). Both are unacceptable, and are an indication of something wrong in the design, at least in the view of the relational model.

The *one-to-one* is easy to resolve. It is probably sufficient to merge the two tables into one as they relate to the same entities.

The *many-to-many* is a little more complex. Usually, it is necessary to create an extra table with only the two related columns. This new table is usually called *connecting* or *association* table.

In the case of the previous tables, since there are no *one-to-one* relations, there is no need to apply the merging steps above. If we assume that every participant can come only from one country, then we have a *one-to-many* relation (one country per participant, but many participants from each country). But if double nationalities are permitted, then we need to take out “Country” from the Participants table and crate a new table “Nationalities”, with the two columns “Name” and “Country” only. Then, every multi-nationality will have only one entry in in the Participant table, but as many entries as nationalities in the third table. All in all, only *one-to-many* relations are left.

1.12.7 Normal forms

Codd himself suggested a set of rules or forms that must be verified by the table design. Four or five are usually recognised as mandatory, while the next ones are more often ignored. Each form imply that the requirements of the previous ones have been met. Following these normalisation guidelines, we will often decrease the number of columns and add new tables as we did just before.

First normal form

It requires that at each row-column intersection there must be one and only one atomic value. There must be no repeating group in a table. If participants were allowed to come back for many colleges, then the "Participant" table above would not be acceptable. It could be solved by creating a fourth table with "Name" (or "PIN") and "Year", dropping "Year" from "Participants".

Second normal form

This form concerns only tables where the primary key consists of many columns. Then every *non-key* column must depend on the entire primary key. A table must not contain a non-key column that pertains to only part of the composite primary key. In other words, no non-key column shall be a fact about a subset of the primary key.

Third normal form

This is a generalisation of the second form. It requires that no non-key column depends on another non-key column. Each non-key column must be a fact about the primary key only. In the "Countries" table above, the column "Ratio" depends numerically on the "Male" and "Female" columns, both non-key columns. In our case, this column (Ratio) should be dropped, as it can be recomputed easily at any time.

Fourth normal form

This form forbids independent one-to-many relationship between primary key columns and non-key columns. This situation is relatively rare. It is an indication that we are mixing together in the same relation things that are effectively independent and should appear in different tables.

Remarks

These rules or forms are rather abstract and the user may not see immediately why he should apply them.

If you are a stranger to their formal beauty, then there are very good reasons to apply them nevertheless. The keywords here are coherence and (no) redundancy. If the data base is to survive system crashes as well as operator errors, and stay coherent at the end, then the application of the four normalisation steps is necessary. First, there will be no redundancy; all information will appear only once. Then, because of this, all data will stay coherent, they can not appear with different values at different places. The "Ratio" column discussed in the third normal form description, is a good example. If any value in the "Male" or "Female" column is changed, but the "Ratio" left unchanged, then the table is incoherent, and there is no way to force them to be updated at the same time.

This was the main problem encountered with hierarchical and network data bases. They used either multiple copies of some informations, or had pointers all over the place to some informations. Any incident during an update was catastrophic, as different values for the same information were left behind, or pointers were pointing to data non existing anymore. Relational database systems can be built without using a single pointer.

Formal rules are very useful, but not always sufficient.

1.13 UNIX flat tables

1.13.1 Using pipes and redirections

Among others, UNIX is based on two things : 1) all files are unstructured strings of characters; 2) a large number of small programs, dedicated to a well defined task, that can be linked together in chains, each one getting data from its predecessor and giving results to its successor. Real files appear at both end of the chain, while pipes, virtual files (memory buffers ?), link the various modules (programs).

As a reminder, `prog < file1 > file2` means that `file1` is redirected to the standard input of `prog`, and the standard output is redirected to `file2`. Similarly, `prog1 | prog2` means that the standard output of `prog1` is redirected as standard input for `prog2`. These redirections and pipes can be combined as desired : `prog1 < file1 | prog2 | prog3 ... progN > file2`

This model is implicitly present in most, if not all, data base manipulation or interrogation if it is assumed that relations are represented by individual files (see the seminal work of Manis et al. 1.14.7, page xcix).

As an example, we have a file containing the Name, FirstName, Country, Telephone, Email, No, Flag and Year for all participants of ICTP Colleges. Let us call this file `Participants`. We want to extract the Name and Email addresses of all participants of the colleges between 1997 and 1999. The result should be ordered alphabetically by Name. Here is a solution :

```
cat Participants | \
select ' Year > 1996 && Year < 2000 ' | \
column Name Email | \
sorttable Name > P1997-1999
```

1.13.2 Building Databases

Tables can be created with any editor (vi, crisp, emacs etc.), or from within any program in C, Fortran or Java, or from a shell script.

Here is a small example of a script that runs forever and records every 60 seconds the local time, the number of users and the three “loads” as measured with the `uptime` command.

```
#!/usr/local/bin/ksh
```

```
# Create and load a table with time, no of users and loads from uptime
echo "time\tusers\tload1\tload2\tload3"           > Load.rdb
echo "----\t-----\t-----\t-----\t-----"          >> Load.rdb
while true ; do
  uptime | tr -s " ,\t" "   " > /tmp/uptime
  exec 0< /tmp/uptime
  read Time dum dum Users dum dum dum Load1 Load2 Load3
  echo "$Time\t$Users\t$Load1\t$Load2\t$Load3"   >> Load.rdb
  sleep 60
done
rm /tmp/uptime
exit 0
```

The following is the beginning of the resulting file (the <TAB>s have been expanded as spaces) :

```
time    users   load1   load2   load3
----    -----   -----   -----   -----
10:22pm 0       0.17    0.15    0.10
10:23pm 0       0.12    0.14    0.09
10:24pm 0       0.09    0.12    0.09
```

Finally, here is a larger script that builds up a dictionary of all fields from the files in the current directory.

```
#!/usr/local/bin/ksh
# Build a dictionary of all fields from the /rdb files
# in the current directory
#
# First create a small dictionary for every table (*.rdb file)
for f in *rdb ; do
  dict=${f}_dict
  echo "field\t$f"    > $dict
  echo "-----\t----" >> $dict
  head -1q $f | \
    tr "\t" "\n" | \
    sort | \
    awk '{ print $1 "\tX" }' >> $dict
done
# Then join together all individual dictionary
First=1
for f in *_dict ; do
  if [[ $First -eq 1 ]] ; then
    First=0
    cp $f Dictionary
  else
    jointbl -c < Dictionary field $f > /tmp/Dico
    mv /tmp/Dico ./Dictionary
  fi
done
```

```
# Add a comment field
awk ' NR==1 { print $0 "\tDescription" }
      NR==2 { print $0 "\t-----------" }
      NR>2  { print $0 "\t"        } ' < ./Dictionary \
                                       > /tmp/Dico
mv /tmp/Dico ./Dictionary
# Do some cleanup and exit
for f in *rdb ; do
  rm ${f}_dict
done
exit 0
```

and the resulting `Dictionary` :

```
field       College.rdb  Countries.rdb  Load.rdb  Description
----------  -----------  -------------  --------  --------------------
Capital                  X
Country     X            X
Email       X
Female                   X
FirstName   X
Flag        X
Groth                    X
Male                     X
Name        X
Nb          X
Population               X
Ratio                    X
Telephone   X
Year        X
load1                                   X
load2                                   X
load3                                   X
time                                    X
users                                   X
```

The *Description* field can now be edited by hand.

1.13.3 UNIX commands

There are at least three or four different systems based on the previous ideas. The simplest (*RDB*) is entirely written in *Perl*. Because of this, the user can read the code and get an unusual feeling of what is done. The entire code, is 8100 lines, for 20 commands. At the other end, *starbase* and */rdb*, are written in C with access to *awk* and *sed*. */rdb* contains about 130 different programs, including for commercial applications. *starbase* is almost as rich, but was developed with astronomical users in mind. Both are much faster than *RDB*.

Here is a commented list of the main programs of *RDB*.

indextbl builds an index for one or more columns from a table for fast searching (see `search` bellow);

jointbl joins two tables on a common column;

mergetbl adds the contents of a table to the end of another table (they must have exactly the same columns);

ptbl pretty prints the contents of a table;

rdbedit a window data entry/editing facility for tables;

reporttbl formats and prints an arbitrary style report, as specified in another file;

row extracts lines from a table according to some given criteria;

search fast searching using index file built with `indextbl`;

sorttbl sorts a table according to one or more columns in numerical or lexicographical order;

subtotal calculates subtotals for one or more numerical columns;

summ summarizes information from a table, as number of rows, number of unique values in some given columns as well as minimum, average, maximum and total.

uniqtbl removes adjacent rows that are identical in some columns;

valid check that a table is in “good” format, meaning all the rows correspond to the header of the table.

These programs have many options, including `-h` that will give a good description for each of them.

As said before, these UNIX tables have nothing special and can be used or manipulated by any other program which is not part of the original Database System.

1.13.4 Advantages and limitations

The advantages of this approach are numerous, but its limitations are also important.

Advantages

- it is extremely simple and fast to create new tables, often *on the fly*, work with them, and destroy them when the job is finished;
- it is an open system, and so very easy to interface to other programs, data acquisition, backup, text processing, graphics, statistics etc.

- the files are very compact; no extra space is reserved in advance;
- the user has full control of what is happening; he can optimize complex operations using his knowledge of the data and query.
- considering the last rules of Codd (see section 1.12.3, page lxxxvii), the UNIX system itself provides confortable access security, though at the file level more than at the record level, and many solutions for recovery procedures. `rcs`, or better `cvs`, can be used to keep track of all table modifications, including facilities for back and forward rolling. Atomic transactions can be done using lock files or semaphores.

Limitations

- its simplicity encourage going to fast development, bypassing the careful analysis needed for a good Data Base design;
- it is not standardised, neither in the exact format for the tables, nor in the command names;
- it is much better at data query than update due to the simple table format.
- there is no global view or description of the Data Base as such. All tables rest independent, only under the control of the user who has to keep them coherent.
- it is usually slower during execution, though the factor is smaller than could be expected for an interpreted system.

1.13.5 SQL equivalence

SQL is a de facto standard, in particular outside the UNIX environment. It is thus useful to look at the equivalent commands on both side.

SQL	/rdb
`SELECT col1 col2 FROM table`	`column col1 col2 < table`
`WHERE column = expression`	`row 'column == expression'`
`COMPUTE column = expression`	`compute 'column = expression'`
`GROUP BY`	`subtotal`
`HAVING`	`row`
`ORDER BY column`	`sorttable col`
`UNIQUE`	`uniq`
`COUNT`	`wc -l`
`NESTING`	“pipes” —
`INSERT, UPDATE, DELETE`	editors, form softwares etc.

1.14 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.

1.14.1 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*

1.14.2 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 : Semi-numerical 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*

1.14.3 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*

1.14.4 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*

1.14.5 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*

1.14.6 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*
- **Newbam C. and Rosenblatt B.** Learning bash shell, *O'Reilly & Associates 2000*
- **Quigley E.** UNIX Shells by Examples (Csh, sh, ksh, awk, grep and sed). *Prentice-Hall 1999*
- **Rosenblatt Bill** Korn Shell. *O'Reilly & Associates 1993*
- **Scott, G., Gundavaram, S. and Birznieks, G.** CGI Programming with Perl. *O'Reilly & Associates 2000*
- **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*

1.14.7 Relational Database

- **Bowman J. S., Emerson S. L. and Darnovsky M.** The Practical SQL. *Addison-Wesley 2001*
- **Codd E. F.** Relational Database: A Practical Foundation for Productivity. *CACM,* **25, No 2, 109-117, February 1982**
- **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**
- **Date C. J.** Relational Data Bases: Selected Writing. *Addison-Wesley 1986*
- **Date C. J.** A Guide to INGRES. *Addison-Wesley 1987*
- **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*
- **Yarger R. J., Reese G. and King T.** MySQL & mSQL. *O'Reilly 1999*

Index