H4.SMR. 403/21

FIFTH COLLEGE ON MICROPROCESSORS: TECHNOLOGY AND APPLICATIONS IN PHYSICS

2 - 27 October 1989

Software Technics

P. BARTHOLDI
Observatory of Geneva, Sauverny, Switzerland

Software technics

Paul Bartholdi
Observatory of Geneva
CH-1290 Sauverny
Switzerland

Preliminary notes - Trieste - october 1989

Fifth College on Microprocessors

Contents

1 Introduction, why Software ?

1.1 Historical perspective

Although history is always more complicated, we can put all computers into 3 groups, according to the technology they are made off :

1. Tubes (valves) (1940-1960) , very bulky, costly, unreliable
2. Transistors (1960-1970) bulky, costly, reliable
3. Integrated Circuits (IC) (1970-...), very small, cheap, very reliable

During the last 45 years, we can see that :

- At constant price, the "capacity" (speed, memory etc) has increased by a factor of $1.4 - 2$ per year
- At constant "capacity", the cost has decreased by ~ 1.5 per year
  or 10'000 in 25 years

Now, the hardware becomes available for all kinds of functions (down to coffee pot) but it is the software which makes this hardware so flexible and performant.

The same hardware will be used for many different applications ($\longrightarrow$ low cost).

For many applications, the special features of the hardware becomes less important.

Since 10 years, the cost of software $\gg$ cost of hardware,

and so **we need tools to make good use of this cheap and powerful hardware !**

1.2 Grosh's law

30 years ago, Herbert Grosh published his "law" about the cost versus performance relation. At that time, the most powerful computers where also the cheapest for a given application. But in 1957, only big main-frame existed. In fact, the Grosh-law is still valid inside a given class if we separate the computers into classes like : super, main-frame, mini and PC.

In cost per instruction, the big supercomputers are not much cheaper than μP which are also a little cheaper than the slide-rules. But the speed ratio is enormous. Typically, a super-computer is 10^9 time faster than a slide-rule (ignoring precision), which is the ratio of five times the speed of light compared to a walking man.

We need tools and techniques to make good use of this speed.

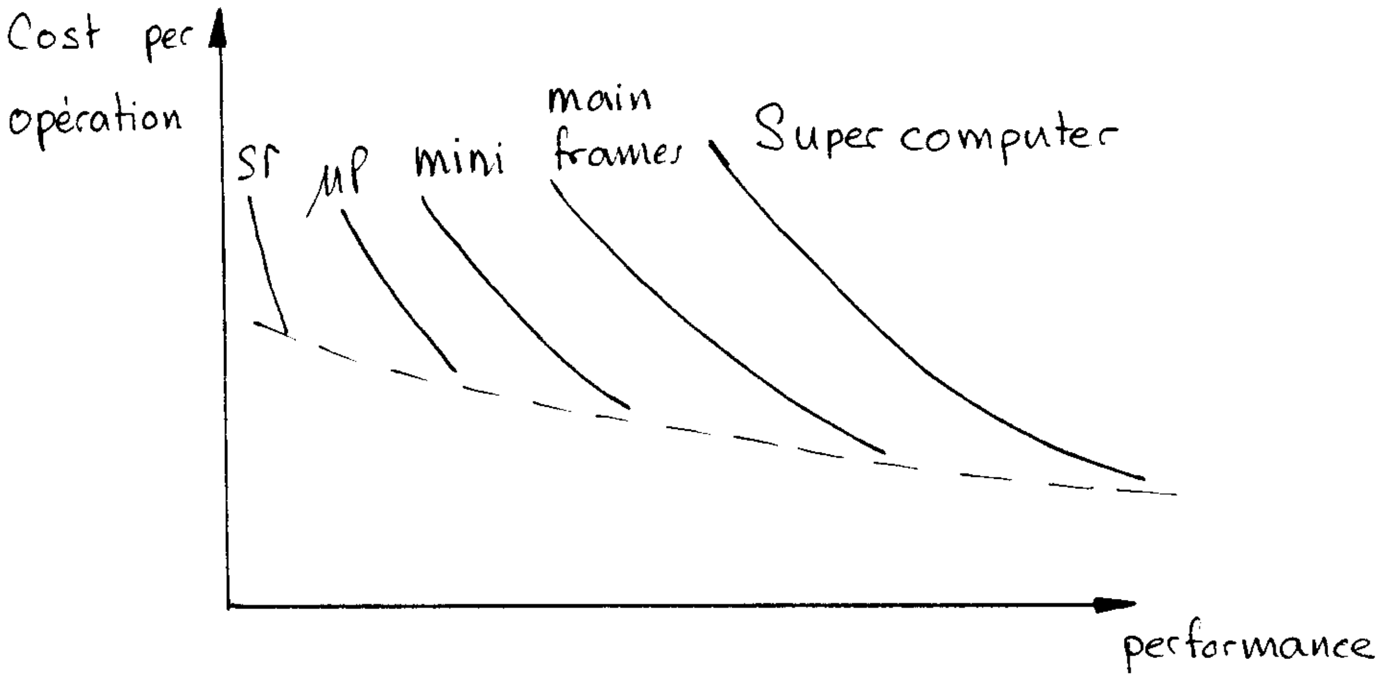


Figure 1: Cost per instruction versus performance

1.3 Trade-off and conflicts

Very often we will face trade-off conflicts between :

- short / long term cost
- safety / flexibility
- speed / memory size
- personal / group advantage

No single rule can be established to resolve these conflicts. They are usually situation dependent, but you must be ready to face them.

2 Structured programming

2.1 Goals

With the hardware improving so rapidly, the programmes became bigger and bigger, more complex, but also containing more errors, being more delayed etc.

At some level, each correction of an error introduced more new errors, and so the situation could only get worse.

So the goals of good programming became first to reduce the **total cost** of software, while producing more and bigger programmes :

- with less errors
- that are easier to correct and modify
- that can be understood by others

As a rule, a good programmer produce 10-20 lines of code per day, each line cost $\sim$ 20 US $.

2.2 Proposed solutions

Around 1970, 6 years after the introduction of the IBM/360 series, many solutions where proposed to solve this problem of software cost :

- goto-less programming - no spaghetti ! (see the article by Dijkstra)
- block programming (1 entry - 1 outcome)
- top-down design (hierarchy)
- stepwise refinement
- data structure as object
- abstraction (ignore the details when not needed)
- information hiding (ignore how the details are done)
- correctness proof

It should be noted that many of these solutions are just good engineering practice, that some are in fact equivalent, and finally that many of them can or even should be combined together.

3 Top-down design

Top-down design is just a good general engineering methodology to :

- guide the designer / programmer during development
- reduce error risks and total cost
- help maintain programme alive for as long as possible

! Cost is not only expressed in $ of £, but also in delays and manpower to produce and later use the product.

3.1 Programme development phases

The development of a programme goes through many phases, and usually cycles through them with testing - correcting - evolution :

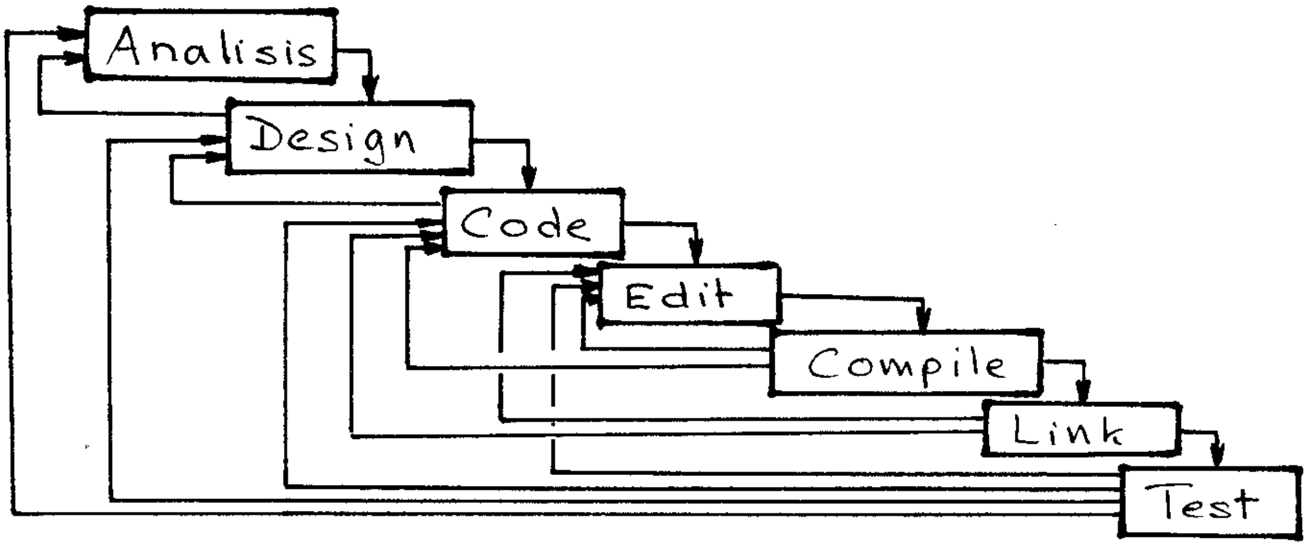


Figure 2: Programme development cycle

The further you have to go back, the more difficult it is !

3.2 Some general rules

- Always start designing at the **TOP**, taking first the general decisions
- Describe **WHAT** you have to do, postpone as late as possible **HOW** you will do it
- Describe carefully :
  - **what** lower levels have to do
  - which **input** do they get
  - which **output** they should produce

  - all **exceptions** , **error handling**
- Hide to higher levels the information on how it is done
- Plan for comprehensive test data set with simulated lower levels
- Take the user interface to fix the highest level

3.3 Programme coding

- Follow the same path for programme coding
  - insufficient or incoherent design are easier and better detected
  - use design immediately for documentation
- Use simulated low levels to start testing higher levels very early
- or start editing - testing at bottom and build programme to the top on certified lower levels
- (Very) large programmes are safer on a pure **top-down** cycle, where **conceptual errors** can have dramatic effects if detected too late
- **Bottom-up** testing is usually faster, specially when **coding errors** dominate

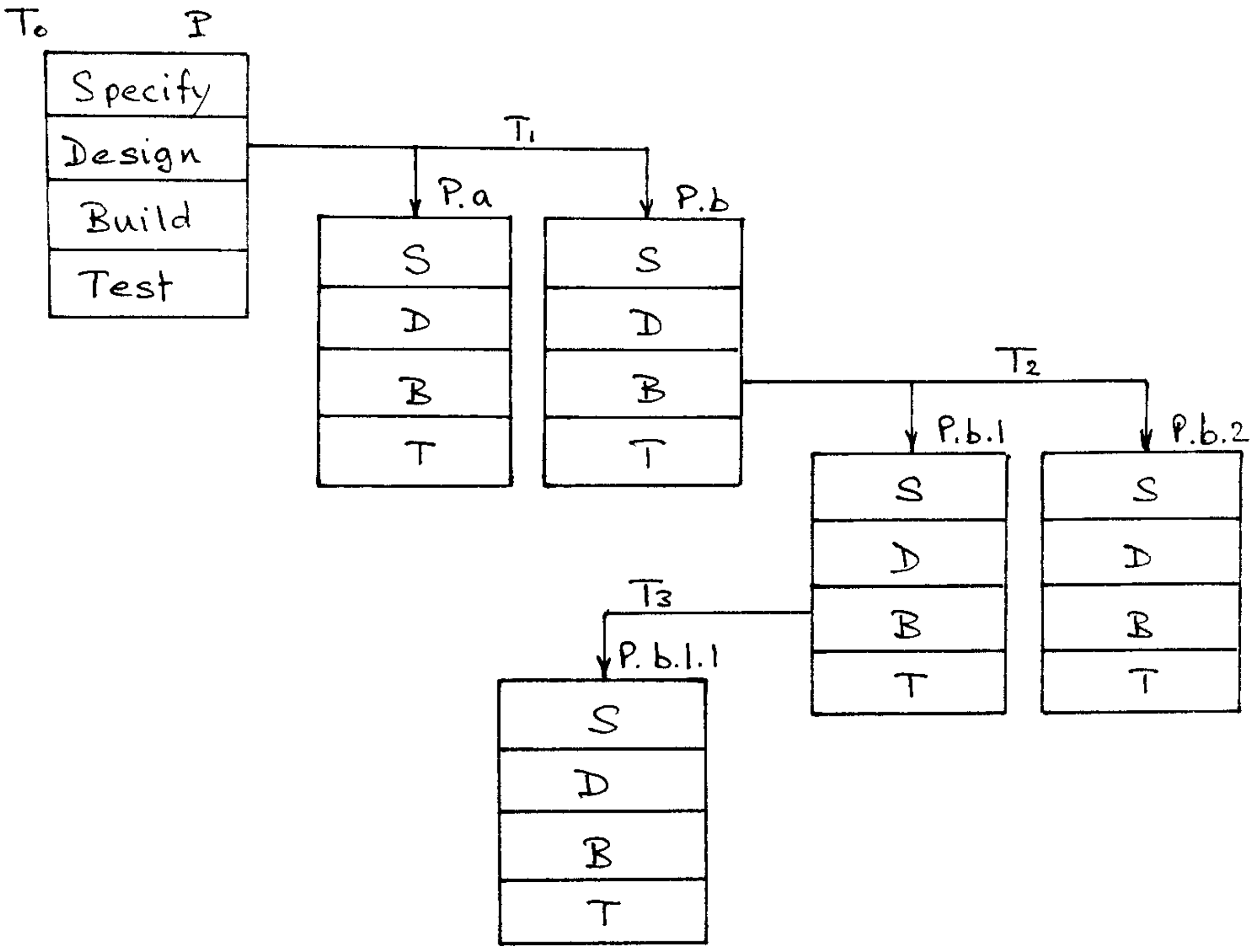


Figure 3: Structured Top-down design time chart

4 Block-structured - Modular programming

- make a new **module** for each identifiable (sub) task
- each module should have
  - a single goal
  - only one entry and one outlet
- each module should be
  - readable as a single object (< 1 page)
  - easily understandable and verifiable (not to many paths)

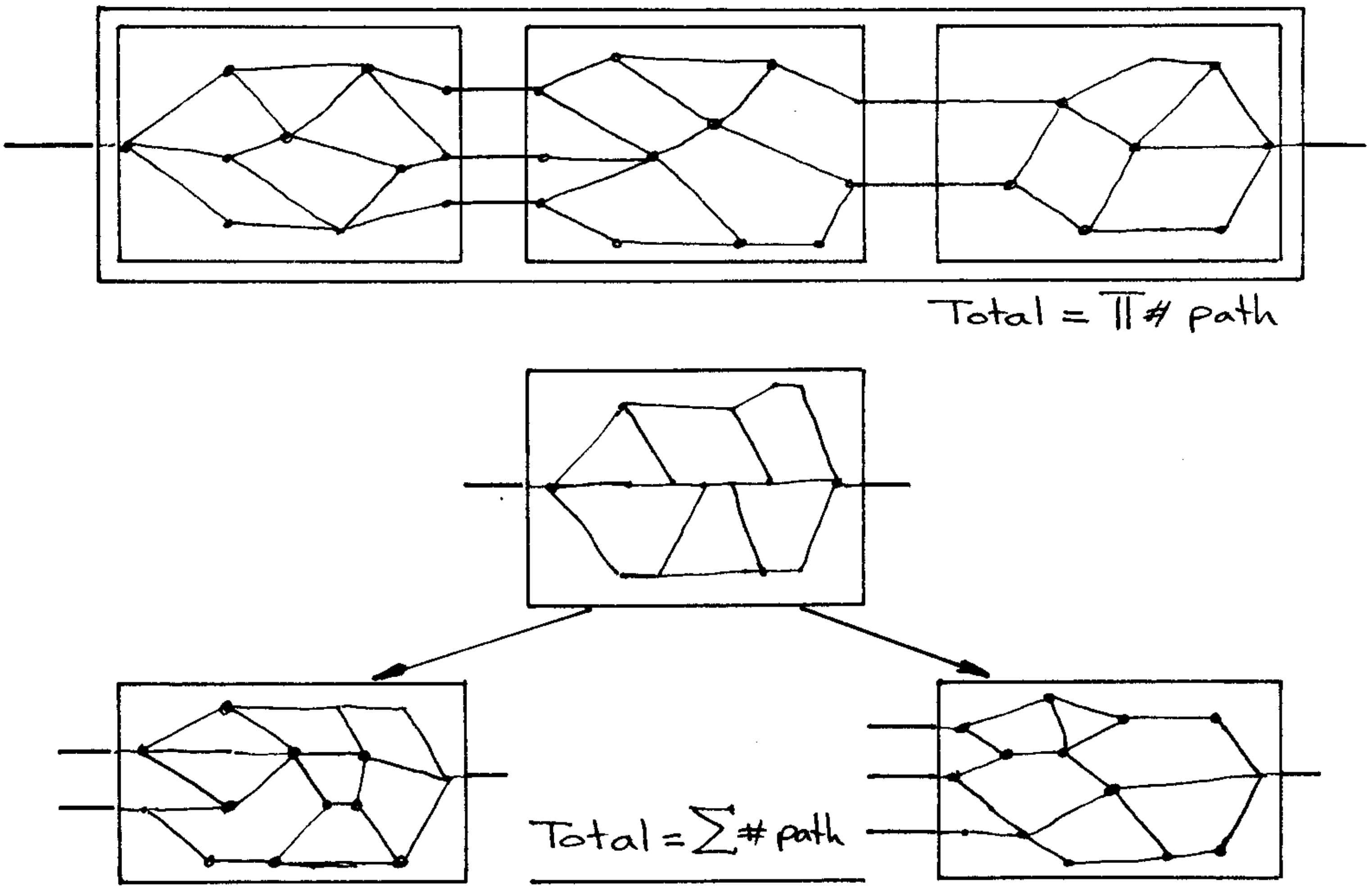


Figure 4: Global and modular testing

5 Documentation

Some programme are used once and never used again.

Most programmes

- will be used many times
- will be changed, up graded
- will go to other users
- will contain undetected errors

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

Good programming + Good documentation = lower future cost

5.1 Internal documentation to the code (for each module)

Header
- name + descriptive title
- programmer 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

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

5.2 Programme logic manual - for maintenance

- programme 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
- debugging aids, how to use them

- interface for new modules

It should complement the Internal documentation (not duplicate it)

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

5.3 User's guide - reference manual

Should help the **user** !

- programme name
- what it does (briefly)
- input expected, controls available
- unusual conditions, errors, limitations
- sample run with input, output and comments
- how to contact author/maintener
- acknowledgements
- references (how it does it)

6 Stepwise refinement

1. describe programme action in a few very general steps
2. take each step, refine it into several smaller ones
3. continue refining each step until you get clear, simple steps

Each atomic step should be :

- written in a few lines ($<$ 50)
- demonstrable to be correct
- detailed information should be handled here, not communicated to higher levels

You can use decimal numeration to help visualize refinement.

6.1 An example

```
        program STATIS
1.      call read_data(#data,data)
2.      call statistics(#data,data,summary)
3.      call print_summary(#data,summary)
        end

1.      proc read_data(#data,data)
1.1     read #data
        do for #data
1.2     call read_unit(data)
1.3     call check_data_unit(data)
        end

1.3     proc read_unit(data)
1.3.1   ...

            etc.
```

7 User interface

The user interface is very important :

- it is the aspect of your programme seen by users
- it conditions strongly its acceptance
- it may play a major role in user's work quality

7.1 Some general rules

- Help him, don't bother him !
- Put yourself in the user position, not him in the programmer position
- put **him** at the commands, not the programme (with some exceptions)
- Let him do what he is best at, let the programme do the rest, what he is bad or slow at
- Consider him as knowledgeable, not as a stupid robot
- Protect him against dangerous actions (double checking)
- Have an **UNDO** command after all dangerous actions
- Give him as much freedom as possible, without imposing a given path
- Do not overflow his vision with unnecessary rubbish
- Let him chose the degree of help he needs, and when he wants it
- Give sensible answers to his questions and calls for help (on-line)
- Avoïd ambiguities
- Build user-friendly error handling, with short but meaningful messages and restart possibilities
- Do not try to interpret what he may wanted ...
- Use a uniform notation
- Use a uniform syntax
- Accept free format values, do not impose artificial or unnecessary rules
- accept many commands on same line, execute them sequentially (left to right, as you read them)

7.2 Some tricks

- have a **HELP** that gives the syntax (parameters) for a given command
- have unfinished command symbol be expanded (as confirmation)
- in case of ambiguities, suggest alternative possibilities
- have a small built-in editor, that works on the last line(s)
- have a short and long version of names, also synonyms
- give the user some indications on how long it will run (if > 5 seconds), show him once in a while that the programme is still alive (% accomplished etc ...)
- record all user inputs, and analyse them to understand :
  - how he works (adapt your programme to it)
  - which kind of errors he makes and why (improve your programme in consequence)
- have some facilities for him to express what he do not like and why (could be included in the previous file)
- have a central, unique, resuable command interpreter, with simple jump table and parameter passing

Good solution : FORTH-like, expandable, interpreter/compiler

7.3 A very safe and stable solution

The following example describes the user interface for an astronomical instrument that is to be used at night, for decades, by untrained layman.

It has no screen, no keyboard, but :

- a set of large push-buttons with green and red lamps
- rotary switches for numbers
- a different task is associated with each button. It can be :

no light	task not running, not runnable
green light	task ready, but not running
red light	task running

  To get some action done, the user has to push one or many of the green lights
- the programme controls the execution by switching on green only permitted tasks
- rotary switches have an extra FF that is set by user when numbers are ready, and reset by programme after read-out
- rotary switches can be set in advance, programme will not wait
- user sees only commands available (green lights)

- push-button can be handled in the dark, even with gloves
- overall, it is very inflexible, what we explicitly wanted, as the instrument should stay unchanged for more than 25 years !

7.4 Common user interface : MENU

Present on the screen a (short) list of possible actions that can be chosen with either a pointer (mouse, cursor, finger etc) or with a single number or letter.

It should be :

- short
- easy to read
- without ambiguity
- not disturbing the rest of the screen

7.4.1 Some design rules

- present most probable choice first
- keep the menu at the top, bottom or on the side of the screen
- avoïd erasing all the screen (you erase some information that could be useful for the choice to be made, and it is rather disturbing for the eyes)
- if the number of choices is too large, build-up **hierarchical menus** (upside-down trees). The **root** menu selects **sub-menus** , **actions** are selected in the **leaves**
- leave the choice to jump directly
  - to the leave you want
  - back to the root menu
- have easy cursor like commands to go back and forth, up and down through the tree
- keep track and show where you come from (use stack)
- build "Jump tables" associated with corresponding choices for each menu/submenu

7.4.2 Why I do not like menu

- at any moment, the choice of actions is limited
- too many decisions are to be made, while usually I know exactly what I want to do
- too much to read, too many changes on the screen
- inflexible flow of actions
- single character answer prone to errors

Addresses	
1	Select file
2	Edit address
3	Select address
4	Print address
5	End

Select file	
1	Create new file
2	Use old file
3	Previous menu
4	
5	

Edit address	
1	New address
2	Old address
3	Previous menu
4	
5	

Select address	
1	by Name
2	by Town
3	by Country
4	by Qualification
5	Previous menu

Figure 5: Menu tree for address programme

7.5 Latest fashion : ICONS

moderatly good graphic screen are getting available at moderatly low price.

Textual symbols + keyboard can be replaced by graphical symbols + pointing device.

Most of the work done on menu, icons etc. have been initiated in the Xerox laboratories in Menlo Park (California).

Icons are heavily used in the MacIntosh (Apple), Apollo, SUN, MG1 etc workstations, as the main interface between the system and the user/programmer.

Advantages :

- image are easier to catch than text
- illiterate can use them
- no need for translation
- easy to pack many on a line
- "an image is worth a thousand numbers" [Chinese proverb]

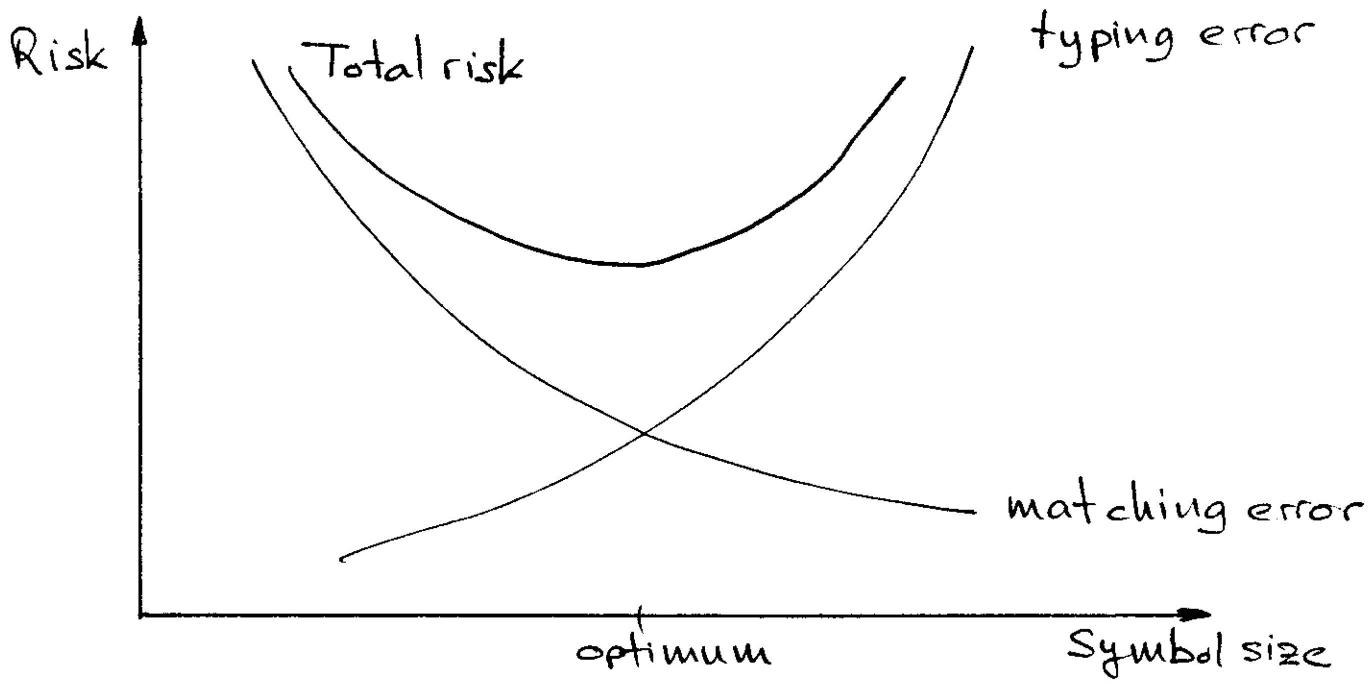


Figure 6: typing/matching error risk versus symbol size

Drawbacks :

- need really a good graphic screen
- need pointing device (slow and error prone)
- difficult to pass parameter(s)
- a "word" is much more precise than an image
- images are not good to express abstractions

The mediterranean civilization came up with the move from the hieroglyphes to the alphabet.

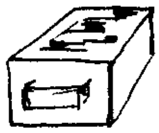
Should we go back to

hieroglyphes ?

ideogrammes ?

The answer is in your hands !

7.5.1 Example of a screen with icons

A	B	C	D	E	F
					
fetch name	print records	print stickers	waste basket delete records files etc.	fetch new file	get current time

The lower part with text is usually not present on the screen, it is here just as comment.

Pointing **E** brings on the screen a list of available files

Pointing one of them, then **D** will delete it

Pointing **C** will print the list of stickers

7.6 Signal generator user interface

Think in terms of Electronic Engineers using generator at the bench.

we have :

1. signal form triangle
   square
   sawtooth
   sinus
2. period (in μs or ms) or frequency (in Hz or KHz ...)

1. and 2. are not necessarily related. You may want to change one or the other, rarely both at the same time.

If it was a non-μP instrument, you would touch only the knobs you wanted, ignoring the other.

The user commands could be in the form :

(n) is any number. Frequencies and Periods are both very useful. The calculations to pass from one to the other are easily done on a μP

- if the command interpreter finds a valid name, it executes it

TR	or	TRIANGLE
SQ	or	SQR or SQUARE
SAW	or	SAWTOOTH
SIN	or	SINUS
(n) HZ	or	(n) KHZ
(n) MS	or	(n) MMS

Figure 7: List of commands for signal generator

- if the name is invalid, it says so, and ask for an other one
- if it finds a number, its value is given to a global variable used by all commands as a parameter

7.7 Main interpreter loop

```
Next_word ;      (loop for ever)
     if line_empty then read_line ;
                        branch next_word ;
     get_a_word ;
     if number then put_into parameter ;
                    branch next_word ;
     if valid_command then execute;
                           branch next_word ;
     if non_above then type "unknown command" ;
                       empty_line ;
                       branch next_word ;
end
```

7.7.1 The symbol table / jump table

STABLE			
2	T	R	I
3	S	Q	U
3	S	Q	R
3	S	A	W
3	S	I	N
2	H	Z	
2	K	H	Z
2	M	S	
3	M	M	S
3	S	E	C
2	E	X	I
2	E	N	D
1	V	O	L

JTABLE
TRIANGLE
SQUARE
SQUARE
SAWTOOTH
SINUS
HZ
KHZ
MSEC
MICROS
SEC
EXIT
EXIT
VOLT

- the first table **STABLE** is 4 bytes wide : 1 byte for the minimum valid length, 3 bytes for the first 3 characters of the symbols
- the first column of **STABLE** gives the minimum valid length for the corresponding symbol
- the next columns of **STABLE** give the valid symbols. Some of them can be shortened to 1 or 2 characters. Some are synonyms like **EXI** and **END**, or **SQU** and **SQR**.
- the second table **JTABLE** is 2-4 bytes wide depending on the processor, and contains the relative address of each subroutine corresponding to the valid symbols in the **STABLE**
- both of these tables should be built using macro.

7.7.2 Signal generator using icons

- variables can be adjusted with up/down "buttons"
- or with pointing device on cursor

It seems very user-friendly, but fine adjustments could be difficult and slow.

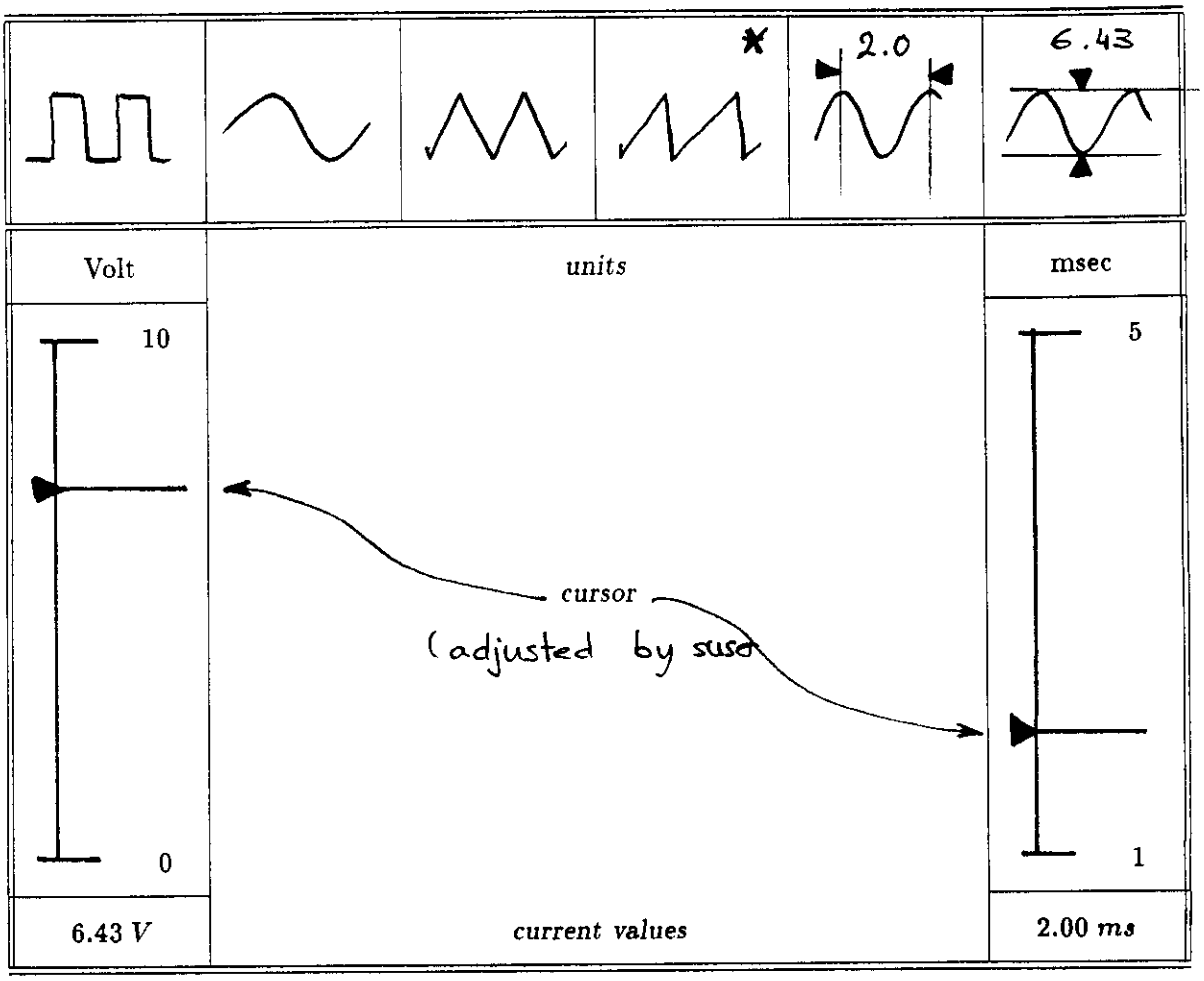


Figure 8: Screen with icons for signal generator

7.7.3 Extensions to signal generator

- Add new commands like (n) `VOLT` or `V` or `MV` to fix the amplitude of the signal
- Put a loudspeaker at the output of the amplifier (with load/impedance adaptor)
- Add (n) `SEC` for the duration of the signal
- Add musical commands like `C D E F`... or `DO RE MI FA`... for sounds, and `LARGO`, `PRESTO` ... for tempi
- What happens if
  - a command needs more than 1 parameter ?
  - we type 2 numbers before a command ?

7.7.4 Address manipulator user interface

User commands :

NAME Vidal	*fetch/create record "VIDAL"*
TOWN Ankara	*set / change element of record "VIDAL"*
SELECT country Nigeria	*extract all addresses in Nigeria*
SORT_BY name	*reorder previous selection*
LIST	*print report, 1 line per record*
ADDRESS	*print stickers*

All lines start with a command (followed by parameter(s))

It has been very well received and used for many years by untrained clerical workers.

The programme looks like :

```
Programme Address_manipulator
     initialise
     do  get_line ; get command ;
         if end then break
                else execute
     enddo
end

proc execute
     case command of
      1 : name ;
      2 : first_name ;
      .
      .
      .
     34 : print_address
     endcase
end

proc name
     get_parameter
     search parameter
     if found then fetch record
              else create record
end

proc print_address
     if no_selection then abort " no selection done yet"
     for i=1 to selection_count
         do print_sticker(name(i))
end

proc print_sticker(name}
     fetch record(name)
     print_line(title,first_name,name)
     print_line(street)
     print_underlined(town)
     print_country
end

...
```

8 Standard

A standard means for you :

- No unnecessary changes
- Transportability
- Reusability
- Stability (good and bad !)
- May not use latest "fashions"
- To be useful, it should be complete and should not contain any extension

There exist now :

standard character set (ascii - ebcdic)

no standard instruction set

no standard assembler

no standard file system

standard language definition (Ada, algol, fortran, pascal etc)

standard hardware interface (GPIB, CAMAC, VME, MULTIBUS)

standard floating point representation (ieee 754)

no standard operating system

Currently, the best defined and enforced standard is for Ada (see the notes on software techniques).

DO NOT USE

EXTENSIONS

TO STANDARD