A description of TSC V1.2

Toni Soto
Sebas Vila

Report LSI–94–17–T

A description of TSC V1.2

A tool conceived to build user interfaces
to distributed applications

ABSTRACT

TSC is a package conceived to build on it any kind of application that requires an user interface mainly command oriented. The package has a distributed philosophy thus having a high modularity and portability. Although TSC was designed with graphical applications in mind, its use is not restricted to them and big profit can be achieved by using it in other kind of applications. This report describes the functional capabilities of the product and it shows several examples of use.

Table of Contents

1 Introduction

TSC is a package conceived to build on it any kind of application that requires an user interface mainly command oriented. The package has a distributed philosophy thus having a high modularity and portability. Although **TSC** was designed with graphical applications in mind, its use is not restricted to them and big profit can be achieved by using it in other kind of applications.

For the kind of programs that people would try to build on top of this user interface, several requirements were identified:

- A flexible command language. The full power of an application becomes more tame when one can easily construct his own commands based on the basic repertoire. Moreover, it is desirable to be able to reprogram the commands on-line, that is without quitting and reentering the application. This favors the fast and easy modification of the interface needed to hasten the iterative cycle of design and evaluation of the interface.
- Command interruptibility. During the execution of a command, the user often needs to execute another command in order to be able to answer the prompts issued by the running command. Therefore we require the command interpreter to have the ability to freeze the execution of the command it was performing and execute a new one instead, and later return to the frozen command in the very same state. This feature adds a lot of power and flexibility to the resulting application with no work by the programmer (other than following good programming practice).
- Command substitution. For instance there may be several ways of entering a point. A command will include any of them as a default, but the user may want to use another in a particular execution of the command. Dynamic substitution —in execution time— of the input function used is thus necessary. Again we're aiming at flexibility with no programming "tax". This also simplifies the interaction while executing the application. The operator need not be prompted for a choice if he can freely change the default "on the fly" whenever he wants to.
- State sensitivity. The application may have several distinct states, such that the commands that are available in each of those states may differ. For example, at points one may be working in a 2-dimensional subsystem, and at times in a 3-dimensional subsystem, which may share some commands (like refresh screen) but not others (like geometric transformations, which are intrinsically 2D or 3D).

We decided to move most of the user interface to a separate process, which would thus become a closed package interacting with the user application through message interchange. This separate process would then handle all input from and text output to the user, and also the command language compiler and the virtual machine running the output from such compiler. This implements

dialogue independence through compliance with the calling model provided. The result is that the programmer writes fully parameterized and encapsulated procedures that are then easily maintained and reused. The interaction with the user application(s) is through a message-passing mechanism. To further save the user from the complications of handling this message interchange mechanism, a compiler was built that, given information on the routines that make up the users' application, automatically produces an ANSI-C main program including the message-passing routines and the appropriate calls to each routine on behalf of each possible incoming message. The user has only got to build a set of C-callable routines, and a file giving a high level description of them.

2 Architecture overview

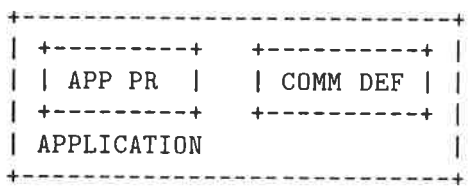
The TSC architecture will not be fully explained in this manual. Only the necessary concepts and terminology that an user needs are shown.

The first concept to be dealt with is the three types of TSC existing users. TSC is a tool to build applications. The person who use this tool to build a new application is named a *programmer*. The programmer usually builds an application offering some degree of configurability. That is, the application can be configured in order to become more useful to do a given task. The person who configures an application is named *customizer*. Once the application has been built and configured it can be used. The person who uses it is named *user*.

Usually, a programmer becomes customizer and user to test the product. It is also current for a user and a customizer to be the same person. Other combinations are also possible. In fact, the three categories of persons related to TSC are logical but physical ones. Nonetheless, it is a good matter to know at any time which kind of person you are. This categories are named *user categories*.

Everyone of the user categories should have a distinct view of TSC. In particular, TSC must be transparent to an user. Thus this manual is not intended for it. The customizer should know little about how to build applications, therefore he should be mainly involved with writing new commands see Chapter 4 [Command Language], page 17. Nonetheless, the customizer and the user should know about the particular application. Thus, it is recommended to include the needed information into the application documentation itself.

A TSC application should be seen, from the architectural standpoint, as distinct things depending on user categories. From the user view point, a closed application is seen and therefore without any particular architecture due to TSC. From the customizer standpoint, the application is a set of application primitives that can be used to define commands using a command language. This can be shown in a scheme like the following:



The command primitives seen depend on the particular application. Thus, a exact view of the particular architecture should be given to the customizer by the programmer.

The third standpoint is referred to programmer. In fact a programmer sees two distinct aspects of a TSC application: the architecture of TSC as a tool and the architecture of a TSC built application. Let us see both of them.

TSC as a tool is a set of

- Tools to help the programmer to construct applications.
- Executable programs needed to run a TSC built application.
- Auxiliary systems offered by other people (contributions).

The first item refers to *generators*. That is, tools that produce source code that will be part of the application. Currently only one of this tools is needed.

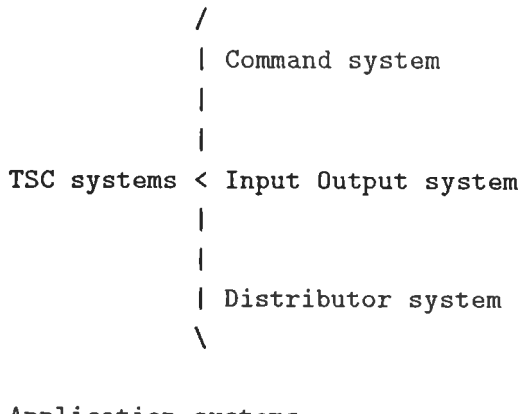
The second item refers to *TSC systems*. That is, fixed parts of a TSC application that are provided in a ready-to-use form. You should keep in mind that a TSC application is always a multiprocess application. Therefore, some of this needed processes are TSC systems.

The third item refers to *auxiliary systems*. That is, to systems that offer new application primitives to the customizer at no cost for the programmer. In fact, these are systems whose differences with the systems that a programmer should implement are none but their ready-to-use form. These systems are not covered by this manual.

If TSC is correctly installed in your site the location of these systems should be transparent to programmer. Ask to your system manager if you have trouble with it or read Section C.2 [Installing TSC], page 80.

All these tools are used by the programmer to build an application. A TSC built application is always a set of distinct processes communicating between them. The processes can be classified into two classes: TSC systems as seen before, and *application systems*. Application systems are processes built by the programmer that construct the application. These processes implement the application primitives that make an application distinct from the other ones.

Therefore, a TSC application is, during run time, composed of the following processes:



The *command system* is the kernel of the system. See Chapter 7 [Command System], page 41. Its main function is to catch user commands and eject the necessary application primitives calls to do the task. It is a configurable system and the commands definition is loaded during start up from a file. See Chapter 4 [Command Language], page 17. The *Input Output System* plays the role of a system console. All the system i/o is done by default using this system. Currently this system opens a window on the screen that lets the user communicate with it. See Chapter 8 [Input Output System], page 47. The *distributor system* is a message dispatcher. It only exists by technical reasons and can be seen as part of the command system. Although part of the TSC set of systems, it must be generated for every application because of its dependence on the last one. See Chapter 12 [Building Distributor], page 61.

3 Building a toy application

In this chapter a brief example will be shown. All the necessary steps to build a toy application inside the POSIX environment are going to be explained. Thus, the reader will acquire a global knowledge of the overall system. More precise definitions should be found in the next chapters. If you are working in a DEC/VMS environment, you should take into account the slight existing differences respect to TSC. These differences are to be explained in the following chapters. To read a brief enumeration of these please refer to Appendix B [DEC/VMS vs. POSIX environment], page 77.

3.1 Sketch of the necessary steps

Below, a list of the steps necessary to build an application are sketched. The same pattern should be followed to construct any application:

1. Define the set of application primitives that the system will offer and establish what process is going to execute everyone of them.
2. Define how many contexts will exist and the set of operations that will be allowed to execute in each context.
3. Establish the mapping between the contexts and the processes that will execute the incoming requests from every context.
4. Edit the process-context mapping file. This file reflects the mapping between the contexts and the processes. The following topics should be accomplished:
   - There was no context mapped onto more than one process.
   - Every context is mapped onto one process.
   - The process names couldn't clash with any other executable name in the system.
5. Edit the application description file. This file contains the definition of the system that the user will see from the command language. This includes the contexts definition, the types definition and the operations definition.
6. Using the gu utility, generate the main body of every process. From this activity, we obtain the files '*process*.c' and '*process*.h'.
7. Implement the application primitives of each process. The following topics need to be considered:
   - The ANSI-C prototypes of the functions that should be implemented are contained in the file '*process*.h'.

- It is mandatory to implement the function `void CanviContext ( int ca, int cf )`. This function is a hook called when a context switch is done.
- It is mandatory to implement the function `void IniUsuari ( int argc, char **argv )`. This function is a hook automatically called before the process is used the first time.
- It is mandatory to implement the function `void FiUsuari ( void )`. This function is a hook. It is the last function called before the process exits.

8. Generate the distributor system using the gu utility. This utility builds the program named `dist.c`.
9. Compile and link the systems built by the user. Every object should be linked with the automatically generated main body. The executable file has to be named *process* without exception. This assures the required consistency with data included in the process-context mapping file.
10. Generate the application loader. This is a shell script that loads all the processes required to run the application. This shell script is generated using the gu utility. Several topics should be considered related to it:
    - The generated script assume that will be executed in the directory where the executable files corresponding to the processes live.
    - This script redirects the output of every running process to the files named '*process*`.log`' when invoked using the `-l` flag. Usually these files contain the trace log of the applications.
11. Edit into a file the definition of the commands needed to work with the application.

3.2 Define the application

The application that we will build is a two dimensional square editor. Therefore, we should define a two-dimensional space delimited by the coordinates [0,0] and [100,100]. In this space, the following operations will be defined:

- An operation to create a square given its size and color at a fixed position.
- An operation to delete a square.
- An operation to inquire the data associated to a square.

With this requisites, the operations should be formally defined. We design a set of operations around the data type `QUADRAT`. Thus, we are going to define an abstract data type. In order to do that, the `PUNT` data type will be useful. Let us define the operations:

`FUNCTION CreaQuadrat (PUNT` *p*`, REAL` *c*`, REAL` *color* `) RETURN QUADRAT`

This function builds an square centered in *p*. The square size is *c* and the color *color*. If the size of the square is bigger than the whole two dimensional world, the square is not created and an `QuadratNul` is returned. The *color* range is [0,5].

`PROCEDURE BorraQuadrat (QUADRAT` *q*`)`

The square *q* is deleted.

`FUNCTION EsQuadratNul (QUADRAT` *q*`) RETURN INTEGER`

The function returns 0 if *q* is a valid square or 1 if it is a `QuadratNul`.

`FUNCTION QuinQuadrat (PUNT` *p*`) RETURN QUADRAT`

The function returns the square that is centered at the point *p*. If there are two or more of these squares, which of them is returned is implementation dependent.

`FUNCTION ColorQuadrat (QUADRAT` *q*`) RETURN INTEGER`

This function returns the color of the square *q*.

`FUNCTION PosQuadrat (QUADRAT` *q*`) RETURN PUNT`

The function returns the position of the square *q*.

`FUNCTION LongQuadrat (QUADRAT` *q*`) RETURN REAL`

The function return the edge length of the square *q*.

`FUNCTION CoorX (PUNT` *p*`) RETURN REAL`
`FUNCTION CoorY (PUNT` *p*`) RETURN REAL`

The functions return one of the components of the two dimensional point *p*.

`FUNCTION CreaPunt ( REAL` *x*`, REAL` *y* `) RETURN PUNT`

This is the funtion that builds a point given both components.

`FUNCTION FisicPunt ( PUNTF` *p* `) RETURN PUNT`

This function returns a point given a physical point. That is, given a point in device coordinates, compute the world coordinates point applying the device transformation and return it.

Being defined the operations, the application data types used should be implemented. Only records are offered as type constructors. Therefore, to define the required application data types we do:

```
TYPE QUADRAT IS INTEGER
TYPE PUNT IS REAL, REAL
```

Following, the number of contexts and how the context are related to the operations should be decided. However, in this example only one context is needed, let us call it `QUA`. Now, the application description file can be written. It will be called '`initial.sc`'. See Appendix A [initial.sc listing], page 65.

To finish, the process-context mapping file should be defined. In this case, there is only a context, so, only one process is needed. The former, named 'proc.sc', will resemble:

```
quadrador QUA
```

All the necessary data to build the application has been completed. The next step is to use the **TSC** utilities to make them effective.

3.3 Implementing the application

Once the files 'initial.sc' and 'proc.sc' are written, the implementation process can be started. Let's continue with our example.

1. Build the main loop of the 'quadrador' process. The command that should be ejected is

   ```
   gu -main quadrador -userp quadrador initial.sc proc.sc
   ```

   This command produces the files

   'quadrador.c'
   : The main loop of the application process. This module should be compiled and linked with the other modules of the process, including the application primitives implemented by this process.

   'quadrador.h'
   : A header file containing all the ANSI-C prototypes of the user-provided subprograms. Every implementation of an user-provided subprogram must include this file. See Appendix A [The file quadrador.h], page 65

2. The application programmer should implement the application primitives. Everyone of the prototypes existing in the file 'quadrador.h' should be implemented. In this toy application, only one file is needed to contain the implementation. Let's name this file 'quadre.c'. See Appendix A [The file quadre.c], page 65.

   This file must be compiled following the procedures required by the supporting operating system. In a typical UNIX installation, the command should be:

   ```
   cc -c quadre.c
   ```

3. The application system can be built. To obtain application systems, the files 'quadre.o' and 'quadrador.o' must be linked together. Because of the way the application subprograms are implemented, also the library 'libtscg' should be linked to them. This library is located at

the directory '`/usr/local/lib`' in a typical installation[1]. Thus, in this installation we can obtain the user process executable file by issuing

```
cc -o -L/usr/local/lib -o quadrador quadrador.o quadre.o -ltscg -lX11
```

Care should be taken because the name of the resulting file, '`quadrador`', must agree with the name given to the process in the file '`proc.sc`'.

4. Now, all the application systems are built. It is time to make the distributor system. This can be achieved by

   ```
   gu -dist dist initial.sc proc.sc
   ```

   This utility generates the file '`dist.c`' which must be compiled and linked to obtain the distributor process.

5. The last step is to obtain the application loader. As the application is built by several processes, a tool is provided for making an application loader. Therefore, using gu, it is as easy as

   ```
   gu -loader requadre initial.sc proc.sc
   ```

   We obtain a shell script named '`requadre`'. It's execution loads and starts all the required processes.

The process that executes the user commands has two main parts. The first, which is named *main loop*, is a dispatcher whose principal task is to call application primitives depending on the received requests from the command system. The main loop is automatically generated by a generator. The second, the set of *application primitives* , is a collection of subprograms and data structures. These are, as his own name suggests, the user supplied procedures and functions that carry out the core work done in an application system.

3.4 Write the commands definition

The application built has no command defined. The applications built under the toolkit **TSC** should define the commands offered to the end user via a command language. The application, during load time, automatically reads a command definition file named '`.tsccomrc`'. All the command defined in this file are to be incorporated into the command processor and, therefore, available to the user. To see a complete description of this language, you should see Chapter 4 [Command Language], page 17.

The command language allows the definition of a commands set laying on the application primitives specified in the file '`initial.sc`'. These commands are accesible to the user and they are the only media available to use the application. In our case, the defined commands are:

[1] This library is not part of TSC but it is supplied with no warranty in the same distribution package

`CREA` Create a fixed square in a fixed location. The square has a edge length of 10 units and a position of the center located at the point (50.0, 50.0).

`Quadrat` Create a new square with a given center and edge length.

`Consulta` Given a square pointed by the user, the command writes the square characteristics.

`Esborra` Delete the user pointed square.

`Escala` Given a square pointed by the user, scales it by a factor.

`Transllada`
Given a square pointed by the user, moves it to a new position.

`Color` Given a square pointed by the user, changes its color.

The complete listing of the command definition can be seen in Appendix A [`'.tsccomrc' listing`], page 65.

3.5 Run it

The application is now completely built. Therefore, it can be installed in the definitive place and any end user can use it. Following we will briefly describe the main salient characteristics of the application from a end user standpoint.

The application is constituted by a set of files. We call this set the *run-time set of systems* or, shortly, the run-time set. In our case, the run-time set is composed by the files:

`'sc'` The command system process. It is a **TSC** system.

`'se'` The input-output system process. It is a **TSC** system.

`'quadrador'`
The application system. It belongs to the set of application systems.

`'dist'` The distributor system. It is a TSC system.

`'initial.sc'`
The file containing all the primitive operations supplied by the application systems.

`'.tsccomrc'`
The file containing all the command definitions that the application is going to know from starting time.

`'requadre'`
The file used to start the application by the user.

All the executable files should be located in a directory known to the path. In addition, the **TSC** systems are the same for all the applications built under the toolkit. Thus, it is recommended to place them in a common use directory. The files '`initial.sc`' and '`.tsccomrc`' should reside in the working directory.

The application starts by issuing the command

```
requadre
```

This action loads all the application processes. Because of this, appear in the screen two new windows:

- The user system window. This the application main window. This window is going to support the graphical interaction with the end user. The manipulation of squares will be done via this window.
- The input-output system window. This window is the media used by the command system to talk with the end user. The end user can issue commands and the command system can show alphanumerical data to the end user. This window has two main subwindows:
  - The bottom window. This is where the user wrote the commands and requested data.
  - The upper window. This is where the command system does the output of textual data.

Now, the input system is implemented using a Motif widget. Therefore, all the Motif resources associated to it will apply. The later are documented in *Motif User Manual.*

At this moment, the user can type any command. Type for instance, the following:

```
CREA
```

and a square is created in the graphic window. Any other command that is defined can be typed. Try, for instance, to type the command:

```
Quadrat
```

the command execution prompts you with the following question:

```
Quadrat, centre
```

point anywhere in the graphic window and answer the other questions. As a result, a square is drawn in the given point. Now, try again with the same command in order to create another square. When the command asks you with

```
Quadrat, centre
```

type the following answer

```
@RealPunt
```

suddenly, the command systems ejects some incredible messages and prompts again asking

```
Punt X
```

A mechanism named *substitution* has been used. Because of that, the way you give the point to the command system has changed. Before the substitution, the point should be given by pointing on the graphic window. After that, the point should be given as a pair of real coordinates. More about how this mechanism works can be read in Section 7.3.2 [Substitution Mechanism], page 43.

The commands can also be interrupted. This means that while a command is running, another one can be executed freezing the later command execution. Try to execute the command

```
Quadrat
```

When this command asks you the point where to put the square, answer with the name of another command, for instance:

```
Consulta
```

The command `Consulta` begins its execution. Answer yourself the asked questions. After the `Consulta` command execution, the command `Quadrat` execution is resumed at the same place where it has been stopped. More about this mechanism can be read in Section 7.3.1 [Interrupt Mechanism], page 43.

Another useful utility is given by the completion mechanism. This mechanism allow to complete the name of an incompletely typed command. Try to execute

```
Col
```

The input system shows you the complete name of the command

```
Color
```

Now you can select the complete name from the input system history box. If the typed prefix is shared by more than one command name, the answer obtained contains all the possible commands. Try with the prefix `Co`. More about the mechanism can be read in Section 7.3.3 [Completion Mechanism], page 46.

This introduction cannot be finished without introducing the *metacommands* concept. The metacommands are a set of predefined commands that any application built on **TSC** has. These are mainly involved with governing the own command system. The currently most used metacommand, is `Compile`. This metacommand reads a text file containing a number of command definitions and incorporates them to the running system. Let's try it.

Write a new command into a file. Say, for instance, a command that translates a given square to a new position. Name the file as '`trans.sc`'. See Appendix A [The file trans.sc], page 65, by example. Now, type the metacommand

```
Compile
```

and answer to the file name question with '`trans.sc`'. The command is now compiled and incorporated to the set of available commands. Verify it yourself by issuing the command

```
Transllada
```

There are several metacommands more. For a complete reference see Section 7.2 [Metacommands], page 42.

4 The command language

The command system is a specialized process whose main function is to interpret the commands sent by the end user and, in response to that, eject the corresponding requests to the application systems. The commands available to the user and their corresponding meaning are described by a programming language. We call this programming language the *command language*.

A definition written in this language is always translated to a kind of 'assembler language' and stored in a *repository*. The repository plays the role of the RAM memory where command definitions are stored inside the command system. See Chapter 7 [Command System], page 41. The later process is named *command compilation*.

See Section 7.2.1 [Command Loading Commands], page 42, for more information about how to compile a command definition. Following, the command definition language is explained and his most salient characteristics sketched.

4.1 Identifiers

All the identifiers required by the language should have the same syntax. An identifier is defined, using UNIX-like regular expressions as:

```
<identifier> ::= [a-zA-Z][a-zA-Z0-9_]*
```

Classical examples should be:

```
Function1
tmp_2
DELETE_OBJECT
```

It is important to note that the language is case sensitive. Thus, the variables `FirstIndex` and `FIRSTINDEX` are distinct each other. In fact, it is a common agreement to write composed word identifiers by gluing together all the words and writing in capitals every first letter of each word. See for instance: `DeleteAll`, `CloseConnection1` or `RollBackTransaction`.

4.2 TSC data types

The CL is a hard typed language. That is, types cannot be mixed in expressions except using conversion operators. The CL defines four distinct predefined data types which are named *TSC data types*. Of course, everyone of these data types has its associate set of primitives. These primitives are *core primitives*. The core data types are:

`INTEGER`

- This type can represent any integer value that can be represented with an `int` data type of the `C` language. Typically this corresponds to the hardware CPU word.
- The allowed arithmetic operations on this data type are the following

  `+` Integer addition

  `-` Integer subtraction

  `*` Integer product

  `/` Integer quotient

  `%` Integer remainder. The operator semantic is the same that the corresponding in the `C` programming language.
- The allowed boolean operations on this data type are the following

  `>` Greater than

  `>=` Greater than or equal

  `<` Lesser than

  `<=` Lesser than or equal

  `=` Equal

  `!=` Not equal

  with the classical meaning.
- The applicable core primitives are

  `ReadInt` It is a function with the following prototype

  ```
  FUNCTION ReadInt (STRING mess) RETURN INTEGER
  ```

  The function, when executed, prompts the end user with the message *mess* and waits the user to introduce an integer. The introduced integer is returned.

  `WriteInt` It is a procedure with the following prototype

```
PROCEDURE WriteInt (INTEGER val)
```

The procedure, when executed, writes the value of the integer *val* to the input system history window. The output is always followed by a carriage return. The value is always written in decimal.

- Integer constants take the classical form, this is, for instance
  `0, -123, +45, 98`

`REAL`

- This type can represent the same range of values that can be represented with a `float` data type of the `C` programming language. Typically this is a single precision floating point real.
- The allowed operations on the type are

  `+` Real addition

  `-` Real subtraction

  `*` Real product

  `/` Real quotient

- The allowed boolean operations on this data type are the following

  `>` Greater than

  `>=` Greater than or equal

  `<` Lesser than

  `<=` Lesser than or equal

  `=` Equal

  `!=` Not equal

  with the classical meaning.

- The applicable TSC primitives are

  `ReadReal` It is a function with the following prototype

  ```
  FUNCTION ReadReal (STRING mess) RETURN REAL
  ```

  The function, when executed, prompts the end user with the message *mess* and waits the user to introduce a real value. The introduced real is returned.

  `WriteReal`

  It is a procedure with the following prototype

  ```
  PROCEDURE WriteReal (REAL val)
  ```

  The procedure, when executed, writes the value of the real *val* to the input system history window. The output is always followed by a carriage return. The value is always written in decimal.

- Constants follow the typical syntax. Despite of that, integer part as well as mantissa part are forced to exist. Thus, they are correct constants

```
0.0, -13.1, 12.001, 0.003, +9.0
```

`STRING`

- This type can represent any sequence of alphanumeric characters belonging to the ASCII set. The sequence has no length limitation.
- The applicable TSC primitives are

`ReadString`

It is a function with the following prototype

```
FUNCTION ReadString (STRING mess) RETURN STRING
```

The function, when executed, prompts the end user with the message *mess* and waits the user to introduce a string. The introduced string is returned.

`WriteString`

It is a procedure with the following prototype

```
PROCEDURE WriteString(INTEGER val)
```

The procedure, when executed, writes the value of the string *val* to the input system history window. The output is always followed by a carriage return.

`StringCompare`

It is a function with the following prototype

```
FUNCTION StringCompare( STRING s1, STRING s2 )
    RETURN INTEGER
```

The function returns an integer code depending on the lexicographical ordering between the two given strings. The returned values are, therefore

`x<0` if s1 < s2

`0` if s1 = s2

`x>0` if s1 > s2

`StringConcat`

It is a function with the following prototype

```
FUNCTION StringConcat (STRING s1, STRING s2)
    RETURN STRING
```

The function returns a string obtained by concatenating the strings *s1* and *s2*.

- String constants follow the classical syntax.

```
"This is a string constant"
```

`PUNTF`

- This type is used to represent a physical point related to a given window and also certain mouse special keys.
- The applicable TSC primitives are

  `ReadPhPoint`

  It is a function with the following prototype

  ```
  FUNCTION ReadPhPoint (STRING mess) RETURN POINTF
  ```

  The function, when executed, prompts the end user with the message *mess* and waits the user to click a point in a window. The picked point is returned.

  `ValidPhPoint`

  It is a function with the following prototype

  ```
  FUNCTION ValidPhPoint( PUNTF p ) RETURN INTEGER
  ```

  The function returns something distinct from `0` if `p` is a valid point or `0` if the point is a end of sequence. This operation is intended to help on working with point sequences. With his help, a iteration can be built over a points sequence. At moment, the user should invariably press an `ESC` key in order to obtain such an end of sequence.

4.3 Application data types

In addition to the TSC data types, each application can provide a set of specific ones. These are named *application data types*. This facility is intended to allow the command language to be tailored to a given application. The set of data types defined by the application is established by the application designer by means of the application description file see Chapter 5 [Application Description File], page 35 and cannot be changed by the user.

The user can declare as many objects of these types as he wants and, of course, he can use any type in a parameter list or as a returning type of a function.

Not only data types are provided but also a set of operations, functions and procedures, over them. The reader should remember that the command definition language is strict-typed, therefore the defined operations are the only mean to work with the data types. Usually, the data types and the operations have an ADT-like organization. Nonetheless, this is enforced by no rule and it is the application designer who chooses the best organization depending on the application goals.

4.4 Variable declarations

Variable objects can only be declared as local objects inside a subprogram see Section 4.8 [Subprograms], page 26. Any number of objects of any existing type, being it predefined or applicati is specific, can be declared. The syntax of a declaration is given by:

```
<type-identifier> <variable-identifier>;
```

It should be noted that only one variable can be declared by declaration. Therefore, to decl e two o more variables, the declarations should be repeated as in the following example:

```
REAL a;
REAL b;
OBJECT obj3;
```

4.5 Expressions

Expressions are objects that can be built by operating other objects. They are defined by t e following rules:

1. Any variable `V` is an expression and its type is the type of the variable.
2. Any constant `K` is an expression and its type is the type of the constant. For instance:
   2 is an expression of `INTEGER` type
3. If `f` is a function and `E` an expression, then `f(E)` is an expression and its type is given by t e returning type of `f`.
4. If `op` is a binary operator and `E` an expression then `E op E` is an expression and the type is t e returning type of `op`.
5. If `op` is an unary operator and `E` an expression then `op E` is an expression and the type is t e returning type of `op`.
6. If `E` is an expression then `(E)` is also an expression and its type is that of `E`.

The evaluation order of an expression is given by the priority of the operators as shown in the following table from more to less priority:

```
()
f()
```

```
- (unary)
* / %
+ -
NOT
= != > >= < <=
AND
OR
```

When conflict arises between two operators of the same priority, then the associativity rule is used. In this language, associativity goes from left to right.

It is interesting to note that there exist expressions of boolean type in spite of existing no `BOOLEAN` type. These expressions are used for sequence control constructions see Section 4.7 [Sequence Control], page 24 like the conditional sentence. These expressions can be obtained by using the relational operators associated to TSC data types see Section 4.2 [TSC Data Types], page 18 and also using the following boolean operators:

`OR`	Boolean or. Whether it is a conditional-or or not is undefined.
`AND`	Boolean and. Whether it is a conditional-and or not is undefined.
`NOT`	Boolean negation.

At present, this characteristic, what can be explained in terms of original design goals, has become obsolescent and will be corrected in future versions of TSC. Therefore a `BOOLEAN` type will be created belonging to the TSC data types and boolean operations will be the type operations.

4.6 Assignment

This is the fundamental statement of the language and is written `:=`. The use is as in any programming language:

```
<variable> := <expression>
```

Where the type of both sides must be the same. Assignment is defined for any kind of existing type.

4.7 Sequence control

Usually is called sequence control to the set of structures that allow the programmer to define the sequence of statements executed. The command language provide a reduced but complete set of structures like most of the imperative programming languages. Let's see every structure.

4.7.1 Sequential composition

A sequence of statements can be written as follows:

```
<statement-1>; <statement-2>; ... <statement-i>;
```

An example of composition can be:

```
i := 3; j:= 4; RedrawObject(q);
```

4.7.2 Conditional sentence

The conditional sentence has the following syntax:

```
IF <boolean-expression> THEN
        <sentence>
ELSE
        <sentence>
ENDIF
```

as usual, the ELSE part can be omitted if it isn't necessary. An example can be:

```
IF NumberOfObjects(scene) > 100 THEN
        WriteString("Too many objects!!");
ENDIF
```

4.7.3 Generalized conditional sentence

There exists a restricted generalized conditional sentence that works only for INTEGER type discriminants. The syntax is the following:

```
CASE <discriminant> IS
        WHEN <integer-const> DO
                <sentence>
        WHEN <integer-const> DO
                <sentence>
        ...
        OTHERWISE
                <sentence>
ENDCASE
```

Given an integer value for the discriminant, the `WHEN` whose integer constant matches the discriminant is executed. Only one `WHEN` clause can match the discriminant at every time. If no `WHEN` clause is matched, then execution of the sentence raises a **run-time error** and process is stopped.

The `OTHERWISE` clause is optional. When given, it catches the discriminant values for which no `WHEN` clause exists.

4.7.4 Iterative sentence

Only one iterative sentence is provided. Its syntax is:

```
WHILE <boolen-expression> DO
      <sentence>
ENDWHILE
```

The corresponding semantic is that of equivalent sentence in the `C` programming language.

4.7.5 Context change sentence

This sentence is intended to execute a list of sentences in a specific context see Section 4.9 [Managing Contexts], page 30. The syntax of the sentence is easy:

```
CONTEXT <context> DO
        <sentence>
ENDCONTEXT
```

The execution of a command is always done within a context. Nonetheless, sometimes it is desirable to execute some sentences within another context. In this case the given sentence should be used. The result of using it is that `<sentence>` will be executed under the `<context>` instead of

being executed in the actual context. Thus, before executing the sentences, context is changed to the new context and after the sentences have been executed, context is changed back to the original context.

As you will notice in Chapter 5 [Application Description File], page 35, a graph of allowed transitions from one context to another exists for every application. Therefore, the context switch correctness is defined in terms of the later. Because of command language structure, this correctness can be verified during compilation time. See Section 4.8 [Subprograms], page 26. Thus, the following rules are applied:

1. Before executing a `CONTEXT` sentence, the set of possible contexts in which command system can be are known to the compiler. Let's name it BC. You can understand why the compiler knows this set by remembering that the `CONTEXT` sentence can be in one of this two situations:
   - It is inside another `CONTEXT` sentence like

     ```
     CONTEXT c1 DO
             CONTEXT c2 DO
                     ...
     ```

     In this case, compiler knows that BC={c1}
   - It is inside a subprogram. In this case BC is the set of contexts for which the subprogram is allowed to be executed.
2. The `CONTEXT c1 ...` sentence is correct iff there exists an allowed transition from each context belonging to BC to the context `c1` in the application graph.

Note that this rule has an important defect: it should be also necessary, to be consistent, to force the existence of an allowed transition from context `c1` to every context in BC. At moment, this is considered an error and can change in future versions of TSC.

For knowing more about contexts and its use, please refer to Section 4.9 [Managing Contexts], page 30.

4.8 Subprograms

Three kind of subprograms are provided by the language. While procedures and functions have a similar meaning and utilization that in any programming language, commands have a special role in the command definition language. Let's see all of them.

4.8.1 Commands

The command language has as a main goal the definition of commands that an user can execute. These commands are built relaying on the primitives offered by the programmer. Command subprograms are the given tool to define this user executable commands. In fact, command subprograms are a special kind of procedures without parameters that **can be executed on user demand**. Thus, the set of defined command subprograms for an application automatically define the set of commands that the user can type.

The syntax of a command is given by the following rule:

```
COMMAND <identifier> WITHIN <context-1>, ... ,<context-i> IS
<list-of-variable-declarations>
BEGIN
        <sentences>
ENDCOMMAND
```

This defines a command named `<identifier>` that can be executed while the command system is within any of the `<context-1>...<context-i>` contexts. For executing the command a set of variables can be optionally declared. These variables are local to the lexical scope of the command and they will exist only during the command execution time interval. Obviously, execution of the command implies sequential execution of its defining sentences. Examples of typical commands can be:

```
COMMAND ERASE WITHIN TwoDEditor IS
        WINDOW w;
BEGIN
        w := FirstAppWindow();
        WHILE NotLast(w) DO
                EraseWindow(w);
                w := NextWindow();
        ENDWHILE;
ENDCOMMAND
```

In this case, it should be understood that

```
WINDOW type,
FirstAppWindow,
NotLast,
EraseWindow and
NextWindow
```

are data types and operations defined by the programmer. Thus, they can be considered as primitive tools for the given application.

4.8.2 Procedures

As in most programming languages, command definition language support the use of procedures. These subprograms have the following syntax:

```
PROCEDURE <identifier> (<par-list>) WITHIN <context-1>, ... ,<context-i> IS
<list-of-variable-declarations>
BEGIN
        <sentences>
ENDPROCEDURE
```

where

```
<par-list>::= <type> <par-identifier>, ... ,<type> <par-identifier>
```

The following topics should be given about procedures:

1. Procedures can only be called from other procedures, from function or from commands but they cannot be called directly by the user.
2. Procedures can be recursive.
3. A procedure can be called iff the actual context of the execution belongs to the context list of the procedure. The compiler checks it during compilation time. Therefore, a given procedure cannot be called from another subprogram if the subprogram context characteristic set isn't contained in the procedure characteristic set.
4. Parameters of a procedure can be only input parameters.
5. A procedure is called giving the actual parameters as expressions of the same type. See for instance:

```
PROCEDURE DrawPoly(POLYLINE p) WITHIN TwoDEditor IS ...

...
p := CanonicalPolyline();
DrawPoly(p);
...
```

4.8.3 Functions

Functions are also supported like procedures. Their syntax is defined by the following pattern:

```
FUNCTION <identifier> (<par-list>) RETURN <type-identifier>
         WITHIN <context-1>, ... ,<context-i> IS
<list-of-variable-declarations>
BEGIN
        <sentences>
        RETURN <expression>;
ENDFUNCTION
```

where

```
<par-list>::= <type> <par-identifier>, ... ,<type> <par-identifier>
```

The following topics apply to functions:

1. Functions can only be called from other procedures, from functions or from commands but they cannot be called directly by the user.
2. Functions can be recursive.
3. A function can be called iff the actual context of the execution belongs to the context list of the function. The compiler checks it during compilation time. Therefore, a given function cannot be called from another subprogram if the subprogram context characteristic set isn't contained in the function characteristic set.
4. Function parameters might be of input type.
5. A function can return a value of any type.
6. A function implementation can only contain one `RETURN` statement.
7. A function is called giving the actual parameters as expressions of the same type. See for instance:

```
FUNCTION ColorOfObject (OBJECT o) RETURN Color WITHIN TDE IS ...

...
COLOR a;
a := ColorOfObject(SelectObject());
...
```

4.9 Managing the contexts

The context mechanism is one of the most salient characteristics included in command language. The main purpose of this mechanism is to help the application programmer to build a state sensitive application.

Command language is the description tool used to define the commands understood by the command system and thus user available. The command system can be modeled by a virtual programmable computer. Its 'assembler language' is the command language. This virtual computer has a state register. The register value is named the *current context* and the set of allowed valued is the *set of contexts* The user can change this state see Section 7.2.3 [Context Change Command], page 42 following certain rules. The actual context, define the set of subprograms (including commands) that the command system can execute at a given time. Therefore, from a user standpoint, the set of commands isn't always the same: it changes depending on the actual context.

The set of supported contexts is defined in the application description file. Therefore it is established by the programmer during design time. Every context has an unique context-identifier. There is an authorization graph that describes which context changes can be made. See Chapter 5 [Application Description File], page 35.

Every subprogram, including commands, is tagged with a set of contexts in which the subprogram is allowed to execute. This is done using the `WITHIN` clause. Thus, the set of subprograms available within a context A are those the tag of which contains the A context. This subprograms set is named the *subprograms set* of a context. From the command language, every subprograms set becomes a referencing environment inside of which, the name space is unique. Therefore, only one subprogram with a given name can exist within a subprogram set. Of course, it is possible that two subprograms with the same name exist in two distinct subprograms set.

Because of expression economy, it is possible for two given subprograms belonging to distinct subprograms sets, A and B, and having the same name, say C, to share the same implementation too. This can be achieved by declaring the subprogram implementation available to both contexts via the `WITHIN` clause. In fact, this is somewhat tricky: all the subprograms called from C should be able to run on both the A and B contexts. It can be seen as having two identical implementations for the subprogram C one within A subprograms set and another within B subprograms set.

The context change statement allows a given subprogram to execute part of its code within a fixed context. Thus, the effect of using it is to change the virtual computer current context. This context change has the same restrictions that the others see Chapter 5 [Application Description File], page 35. Most of the applications use this sentence to simulate the invocation of a shell running

in another context than the current. This can be achieved by using the command interruption mechanism. See Section 7.3.1 [Interrupt Mechanism], page 43 for more information about it. A sketch of this usage is given below:

```
COMMAND TWODIM WITHIN TREEDIM IS
STRING s;
...
CONTEXT TWODIM DO
        s := ReadString("GiveMe TwoDim Commands");
        WHILE StringCompare(s,"EXIT")=0 DO
                s := ReadString("Give me TwoDim Commands");
        ENDWHILE
ENDCONTEXT
...
```

This code iterates continuously within the `TWODIM` context until the user types the keyword `EXIT`. While `ReadString` is waiting, the user can supply any `TWODIM` command name and the command will be executed by means of interruption mechanism. Therefore a new command shell has been created.

4.10 Miscellaneous TSC primitives

`Abort` This is a predefined procedure with the following prototype:

```
PROCEDURE Abort()
```

When called from a subprogram causes the command to abort its execution. (Veure si aborta la comanda o tots els processos actius i congelats.

4.11 Compilation units

This language doesn't look like the conventional ones in that the former is used to build a virtual machine while the later do not. Therefore, if we think in the time the virtual machine is alive as an infinite time, every compilation unit satisfactorily compiled increases the virtual machine set of operations. It is in this context that it can be established what the compilation units are:

- Individual subprograms: that is procedures, functions or commands.
- Prototypes: that is subprogram definitions without implementation.

The compiler always take files to compile see Section 7.2.1 [Command Loading Commands], page 42. These files can contain any number of compilation units everyone of which is independently processed.

4.11.1 Referencing environment and prototypes

Also in this context the referencing environment can be defined. At some given time instant, the referencing environment is the set of subprograms that the virtual machine knows. Of course, these subprograms are classified into subprograms sets see Section 4.9 [Managing Contexts], page 30. This referencing environment let's us to state an important rule:

Given a subprogram A, all the subprograms called by A must be known inside the referencing environment during compilation time. If it is impossible due to recursive calls, for instance, then prototypes must be used to insert a new operation into the referencing environment without implementation. It is important to remember that the implementation must be compiled before running the virtual machine. Prototypes are built by only taking the header of a subprogram. See for instance:

```
PROCEDURE Test(INTEGER a, OBJECT c) WITHIN C22;
COMMAND OPEN WITHIN C22, C45;
```

If at run time a subprogram is called with unknown implementation because only its prototype was given, the execution aborts raising a run time error.

4.11.2 Replacement rules

When a subprogram is compiled that exists before in the referencing environment, a set of rules is applied to know what is the final state. These rules are the following:

1. The compiled subprogram, as stated in Section 4.9 [Managing Contexts], page 30, is considered as a set of subprograms with identical implementations within several, or one, subprograms set. Let's name these subprograms as S1,S2,..,Sn and the corresponding contexts C1,C2,...,Cn. The existing subprogram is also considered as a set of subprograms O1,O2,...,Om within the contexts CO1,CO2,...,COm.
2. If Oj is an older implementation of Si for a context Ci=COj then Oj is replaced by Si.
3. If for a given Si doesn't exist any Oj such that COj=Ci, then Si is incorporated to the virtual machine operations set.

4. If for a given Oi doesn't exist any Sj such that COi=Cj, then Oi remains into the virtual machine operations set.

5 The application description file

This file is intended to be the complete description of the bare virtual machine. See Chapter 7 [Command System], page 41. It contains a description of the application defined data types, contexts and operations. The syntax looks like the command language but a more rigid approach is taken. File is divided into five sections enclosed by the construction `DEFINE...END DEFINE`. All the sections are mandatory although some of them can be empty. This will be explained later. The order between the sections is fixed and it is given below:

```
DEFINE CONTEXT ... END DEFINE
DEFINE INITIAL CONTEXT ... END DEFINE
DEFINE VALID CONTEXT CHANGE ... END DEFINE
DEFINE TYPE ... END DEFINE
DEFINE USER ... END DEFINE
```

Below, each section and its contains is explained.

5.1 Defining the contexts

To define the contexts related data it is necessary to establish the following topics:

- The set of existing contexts (context identifiers).
- The current context at virtual machine initialization time (usually named *initial context*).
- The graph of allowed transitions from one context to the rest of them. This graph sets a security policy about the allowed context changes.

Everyone of the topics is set by a `DEFINE` sentence. The set of context is established by the sentence

```
DEFINE CONTEXT <context-id-1>, <context-id-2>, ... END CONTEXT
```

where the context identifiers follow the same rule that the command language identifiers. See Section 4.1 [Identifiers], page 17.

The initial context is established by means of the following sentence:

```
DEFINE INITIAL CONTEXT <context-id> END DEFINE
```

And, finally, The graph of allowed context changes is defined by the sentence:

```
DEFINE VALID CONTEXT CHANGE
<context-id>, <context-id>, ..., <context-id> TO <context-id>;
....
END DEFINE
```

The meaning is obvious: it is allowed to change from any of the context in the left list to the right context.

5.2 Application data types

Any number of new types can be predefined by aggregation of predefined data types. Every new data type (usually named user defined data type) has its own identifier following the same rules that those of command language. To define new data types, the sentence shown below should be used:

```
DEFINE TYPE
<type-id> IS <list-of-predefined-types>;
...
END DEFINE
```

where `<list-of-predefined-types>` is a comma separated list of TSC data types. See, for instance:

```
DEFINE TYPE
POINT3 IS REAL, REAL, REAL;
PointSym IS INTEGER;
END DEFINE
```

5.3 Specifying the operations

Operations of the bare virtual machine are defined using the prototype syntax as explained in Section 4.11.1 [Referencing Environment], page 32. Thus the section has the following aspect:

```
DEFINE USER
<prototype-1>;
<prototype-2>;
...
```

```
END DEFINE
```

There are two restrictions to the prototypes:

1. Prototypes must be of procedures or functions but not of commands.
2. Every prototype must contain a `WITHIN` clause with one and only one context identifier.

The last conditions should be understood by examining how these operations are bound to the process that execute them. See Chapter 2 [Architecture], page 3 and Chapter 6 [Process-Context Mapping File], page 39.

6 The process-context mapping file

This file reflects the binding between contexts and the processes. You should remember that the every primitive operation of the bare virtual machine must be executed by a fixed user process Section 3.2 [Define Application], page 8. By design decision, every primitive belonging to a given context should be executed by the same process. Thus there is a binding that relates a context with a process. In fact, a process can attend more that one context but a given context cannot be attended by more than one process.

This file states which contexts are attended by every application process in the application. Its syntax is extremely simple:

```
<process-name> <context-id-1> <context-id-2> ...
<process-name> <context-id> ...
...
```

It should be remarked that the process name must coincide with the real executable name of the process. See Section 3.3 [Implement Application], page 10.

7 The Command System: sc

This system is the kernel of TSC. Its main responsibility concerns interpreting the commands issued by the user. The system hasn't any mechanism in order to maintain a communication with the user: this task is assigned to the input system. See Chapter 8 [Input Output System], page 47.

7.1 Starting-up an application

Command system is an universal process: it is the same for all applications built. Every application defines the adequate bare command system by means of the application description file. See Chapter 5 [Application Description File], page 35. To achieve this result, the application description file name should be given on the command line when running command system. In fact, this is not usually important because the task is transparently done by the loader generator see [Source Generator], page 49.

After command system is started and initialized for a given application, default commands for this one are read-in if needed. This process takes place automatically by checking the existence of several predefined files in standard places. These files should contain correct command definitions as explained in Chapter 4 [Command Language], page 17. The algorithm describing this loading process is the following:

```
IF exists($PWD/.tsccomrc) THEN
        load-commands-from($PWD/.tsccomrc)
ELSEIF exists($HOME/.tsccomrc) THEN
        load-commands-from($HOME/.tsccomrc)
ELSE
        load-no-commands
ENDIF
```

Note that, in DEC/VMS environment, the file '`.tsccomrc`' is named '`TSCCOMRC.`'.

If no commands are loaded, application has no utility. Therefore, commands must be loaded using a metacommand (see explanation below).

Care should be taken because incorrect definitions of commands will cause the commands not to be loaded. Thus, user can perceive an incorrect application because of this. This fact can be detected by looking into the command system log file.

7.2 Controlling the CS: the metacommands

Metacommands are, from an user point of view, nothing but predefined commands. Nonetheless, there is a big difference between both: metacommands are special commands to control command system while regular commands control the application. Following there is a table with all the existing metacommands:

7.2.1 Command loading metacommands

`Compile` This metacommand is used to load new commands. The commands must be defined in a text file. The file name is asked by the metacommand. If errors are detected during load process, they are written to the command system log file.

7.2.2 Statistical and informative metacommands

`RepositoryStatistics`
The command asks for a file name and dumps into it a lot of statistical data about the subprograms repository usage. This command is mainly used to verify the repository performance.

`ListSymbolTable`
The command asks for a file name and dumps into it the contents of the virtual machine symbol table. Mainly used for debugging purposes.

7.2.3 Context change metacommand

`ChangeContext`
This metacommand followed by a context identifier changes the current context of command system (if allowed). See Section 4.9 [Managing Contexts], page 30.

7.2.4 Journaling metacommands

`OpenJournal`
The command prompts for a file name and puts it as the current output journal. This doesn't mean that the logging process begins but that the log file is established.

`CloseJournal`
: Closes the current output journal file and ceases the logging activity (if existing).

`ActivateRecording`
: This command can only be issued if a current output journal file exists. When executed, it activates the logging mechanism on the current journal file.

`DeactivateRecording`
: This command can only be used when the logging mechanism is activated. When issued, the logging ceases.

`ReadJournal`
: The command prompts for a journal file name and reproduces it.

`ReadStepJournal`
: The command prompts for a journal file name and reproduces it step by step.

7.3 User oriented facilities

The command system offers a set of generic user oriented facilities. These facilities are provided to allow the user to achieve an higher degree of freedom while dialoguing with the application. Although these facilities are offered by the command system, in fact the user will be using it through the input-output system. Please read Chapter 8 [Input Output System], page 47 for more information.

7.3.1 The command interrupt mechanism

This mechanism lets the user to issue a new command, say A, while another one, say B, is being executed. The B command is frozen and the A command is executed. When the A command finish, the B command is waked up and its execution continue at the place where it has been frozen.

A command can be only frozen while it is waiting for a user input. This should easily be done by issuing the name of the new command.

7.3.2 The function call substitution mechanism

This mechanism is provided to allow for a flexible set of input techniques to be easily used and implemented. The mechanism can be used while the command system is waiting for a input data

to came from the user. At this moment, there is a set of subprograms called ending up with the subprogram asking for input. Let's represent the stack of called subprograms by:

```
Sk        <---- Input asking subprogram
Sk-1
...
S3
S2
S1        <---- Command issued by user
```

The user can dynamically substitute the Si by another equivalent subprogram Sj. To do this the following conditions should be stated:

1. Si must be a function subprogram not predefined or belonging to the bare virtual machine.
2. Si and Sj must be *equivalent functions* . That is, both should have the same return type and the same type for all its parameters. For instance,

   ```
   FUNCTION PickPoint(STRING mess) RETURN Point WITHIN CTD;
   FUNCTION OriginPoint(STRING mess) RETURN Point WITHIN CTD;
   ```

   are equivalent functions.

When a substitution by Ss is requested, the stack of subprogram calls is scanned from top to bottom and the first function Sm found that is equivalent to Ss is substituted by Ss. All the stack frames from Ss to the top of stack are discarded and execution is restarted. If no function id found equivalent to Ss, an error warning is issued and execution continues. Therefore, in the example before, if we substitute S3 by Ss, execution is restarted with the following state:

```
Ss
S2
S1      <----- Command issued by the user
```

This mechanism allows the user to change the method (function) he is using for doing a certain input request by another method giving the same information. For instance, suppose the two functions given before: the first one returning a point by the user picking it using the mouse; the second one returning a point by reading the two coordinates from the keyboard. Its implementation should like

```
FUNCTION PickPoint(STRING mess) RETURN Point WITHIN CTD IS
BEGIN
        RETURN ConvertPh(ReadPhPoint(mess));
END FUNCTION

FUNCTION OriginPoint(STRING mess) RETURN Point WITHIN CTD IS
```

```
BEGIN
        RETURN ConvertRe(ReadReal(mess),
                         ReadReal(mess),
                         ReadReal(mess));
END FUNCTION
```

A given command to trace a line should like

```
COMMAND LINE WITHIN CTD IS
Point p1;
Point p2;
BEGIN
        p1 := PickPoint("First point");
        p2 := PickPoint("SecondPoint");
        Line(p1, p2);
END COMMAND
```

When this command is issued by the user, the stack looks like the following:

```
ReadPhPoint      <--- Function asking user
ConvertPh
PickPoint
LINE             <--- Command issued
```

Therefore, user is prompted to input a point by picking it on the screen. If the user prefers to input it by giving the coordinates, a substitution must be issued changing `PickPoint` by `OriginPoint`. After that, stack will look like

```
OriginPoint
LINE             <--- Command issued
```

and when restarted stack will grow to

```
ReadReal      <--- Function asking user
ConvertRe
OriginPoint
LINE             <--- Command issued
```

that is the desired result.

As a consequence of this mechanism, care shoul be taken when designing functions using the command definition language. A correct set of function definitions will establish a correct set of

equivalent functions being able to be substituted. In fact, sets of equivalent functions play the role of being set of alternative input methods from a user standpoint.

7.3.3 The command completion mechanism

This mechanism allows a user to ask the system for command names beginning by a given prefix. The system will answer with the command names of all the matching commands, thus allowing the user to select the correct one.

8 The Input-Output System: se

At the moment, this system don't offers any kind of special services other that offered by the command Motif widget. Thus please refer to the Motif documentation to achieve more information.

9 The source code generator: gu

This utility command is the only tool needed by an application builder to construct a new application. Its main objective is to automatically write distinct pieces of code needed for building the application. See Chapter 3 [Toy Application], page 7. At the moment, the code produced in POSIX environment is ANSI-C and the only system resources needed are POSIX.1 compliant, therefore a POSIX.1 environment is needed to take advantage of it. There is an exception to this rule that needs a POSIX.2 compliant shell (see option `-loader` below). In the other hand, the code produced in the DEC/VMS environment is DEC-C compliant and includes VMS specific library calls. For obtaining information about how to use this tool, please refer to Chapter 10 [Building User Process], page 51, Chapter 11 [Building Input User Process], page 53 and Section 3.1 [Building Steps], page 7.

Command syntax

```
gu [parameters] <resource-description-file> <process-description-file>
```

Command parameters

`-h` Prints on the standard output a usage message showing all the command options.

`-V` Prints on the standard output a message informing about the version of the utility itself.

`-loader <file>`
Generate an executable file named `<file>` that, when executed starts the application. The generated file is a shell script, thus a Bourne compatible shell is needed in order to be executed. This includes the POSIX.2 compliant shell and the Korn shell. If you are working in the DEC/VMS environment, then the option produces a command procedure file fully compatible with VMS JCL. See Section 3.1 [Building Steps], page 7 to know how this executable should be used.

`-dist <file>`
Generate a self-contained source code file named `<file>.c` that, after being compiled and linked, became the distributor process. In order to be compatible with the generated loader, this process must always be named '`dist`'. See Chapter 3 [Toy Application], page 7.

`-userp <process>`

Generate a file named '`<process>.h`' containing the prototypes of the functions that an application builder should provide in order to compile and link the process '`<process>`'. See Chapter 10 [Building User Process], page 51.

`-main <process>`

Generate a source file named '`<process>.c`' that, when compiled, became the `main` function to be linked with the user provided functions in order to obtain a regular user process. See Chapter 10 [Building User Process], page 51.

`-inputp <file>`

Generate a source file named '`<file>.h`' that contain the prototypes of the functions that can be used and the functions that must be used to built an input providing process. See Chapter 11 [Building Input User Process], page 53.

`-input <process>`

Generate a source file named '`<process>.c`' that, when compiled, must be part of the input providing process. See Chapter 11 [Building Input User Process], page 53.

10 Building a standard application process

- A TSC system implements a subset of the primitives defined by the programmer in the application description file. An application system does not need any specific input function.
- There is a tool in TSC that generates the main program of the application system. Thus, the user of TSC will only have to implement the primitives of the system.
- The main program is generated by typing

  `gu -main` *app_system_name app_desc_file proc_cont_file*
- where *user_system_name* is the name of the application system for which we want to generate the main program. *app_system_name* must be the name of one of the processes described in the process-context mapping file '`proc_cont_file`'.
- The former tool generates the main program in the file '`app_system_name.c`'.
- The main program

  manages the communication with the command system

  calls the primitive functions implemented by the application system when requested by the command system.
- Once obtained, the user of TSC only have to compile the main program and link it with the other modules of the application system.
- The user of TSC can get the ANSI-C prototypes of the primitive functions that the application system implements typing

  `gu -userp` *app_system_name app_des_file proc_cont_file*

  where *app_system_name* is the name of the application system for which we want to generate the primitive function prototypes. *app_system_name* must be the name of one of the processes described in the process-context mapping file '`proc_cont_file`'.
- The former tool generates the primitive function prototypes in file '`app_system_name.h`'

11 Building an input-providing application process

- An input-providing application system implements a subset of the primitives defined by the programmer in the application description file. In addition, it implements user specific input functions.
- An input-providing application system deals with two different tasks:
  1. calling the primitive functions implemented by the input-providing user system when requested by the Command System.
  2. sending the tokens generated by the user specific input functions to the Command System.
- The main program of the input-providing user process must deal, at least, with two input sources: the communication channel with the command system and an specific input channel, used by the application specific input functions. Thus, the TSC cannot offer a tool to generate a main program (because it depends on the number and type of the specific input channels used). What can be generated is a communications module that offers functions to manage the call-to-primitive requests of the Command System and to send tokens to the Command System. Actually, these functions aren't the same for a DEC/VMS environment that for a POSIX environment due to deep differences in the operating system resources. See Section 11.1.3 [Input Providing Examples], page 58 to appreciate the differences.
- The set of functions provided by this module is immutable. The ANSI-C prototypes of this function can be obtained by typing:

  `gu -inputp` *file_name app_desc_file proc_cont_file*
- This tool generates the communications module prototypes to file '`file_name.h`'. It contains the prototypes of the following functions:
  1. Initializes the communications. It should be called before any other function of the module.

     ```
     void TscIni(int argc, char **argv);
     ```
  2. Inquire the file descriptor bound to the communication channel and returns it. It only exists in POSIX environment.

     ```
     int TscConsFileDesc(void);
     ```
  3. Inquire the EFN bound to the communication channel and returns it. It only exists in DEC/VMS environment.

     ```
     int TscConsEFN(void);
     ```
  4. Inquire the address of the IOSB bound to the communication channel and returns it. It only exists in DEC/VMS environment.

     ```
     const io_statblk *const TscConsIOSB(void);
     ```
  5. Starts the mechanism that raises the EFN every time that the communication channel should be read. This function should be called when the process is ready to receive data from the channel. If it is not called, EF is never raised. It only exists in DEC/VMS

environment. See Section 11.1.3 [Input Providing Examples], page 58, for an example on how to use it.

```
void TscArrenca(void);
```

6. Receives communication from the command system and dispatches the call-to-primitive requests to the corresponding primitive. This function should be called whenever input is detected in the file descriptor returned by TscIni.

```
void TscReb(void);
```

7. Sends a token to the Command System.

```
void TscEnvToken(const char *token);
```

The token can be of type integer, real or string, but with its textual representation. The syntax of textual representations of values of the former types follows:

```
integer ::= [-]<digit><digit>
real ::= [-]<digit><digit>.<digit><digit>[(e|E)[-]<digit><digit>]
string ::= ["]<character>["]
```

The token can also be a function substitution or a command interruption with the following syntax:

```
function substitution

        @<identifier>

command interruption

        <identifier>
```

8. Sends a token of type PUNTF to the command system.

```
void TscEnvPunt(int winddn, float x, float y);
```

9. Sends an special value of type PUNTF denoting end of sequence.

```
void TscEnvPuntFinal(void);
```

10. Sends a command abort notification to the command system. The user wants to abort the execution of the current command.

```
void TscEnvAbort(void);
```

11. Sends an end-of-application notification to the command system. The user wants to finish the execution of the application.

```
void TscEnvFi(void);
```

- There are applications that use the context change information sent by the Command System. The communication module also offers function to get a context name from a context identifier and vice versa.

```
const char *TscNomContext(int context_id);
int TscContextId(const char *context_name);
int TscDefaultContext(void);

#define TscNullContext          -1
```

- The communications module can be generated by typing

  `gu -input` *input_system_name app_desc_file proc_cont_file*

  where *input_system_name* is the name of the input providing application system and must be the name of one of the processes described in the application description file *proc_cont_file*.
- Once compiled, the communications module must be linked with the other modules of the input-providing user system, included the main program (that must be written by the TSC user).

11.1 Examples of main programs of input-providing user systems

11.1.1 Working with Motif (POSIX environment)

This is not intended to be an tutorial to Motif, thus Motif and Xt functions are not explained. See *OSF/Motif Programmer's Guide* and *X Toolkit Intrinsics Programming Manual* for more information.

```
...

#include <Xm/Xm.h>

/*
* Prototypes of the input-providing user system.
* The file menus.h has been generated with
*      gu -userp menus fr fp
* where menus is the name of the input-providing user process,
* fr is the User Systems Resource Description File and
* fp is the Processes Description File
*/
#include "menus.h"
/*
* Prototypes of the functions of the communications module generated with
*      gu -inputp tscenv fr fp
*/
#include "tscenv.h"

...

/*
* Xt alternative source of input function.
* This function is called when the data is available in the file
* descriptor fd.
```

```
*/

static void connexio_sc(XtPointer client_data, int *fd, XtInputId *id)
{
   TscReb();
}

/*
* Xt Callback function.
* This function will be called when the user has clicked on a button.
* The button is associated with a command. The command which is
* associated with corresponds to the parameter client_data.
*/
void ButtonCB(Widget w, XtPointer client_data, XtPointer call_data)
{
   TscEnvToken((char *)client_data );
}

void main( int argc, char **argv )
{
       ...

       /*
        * Initialize communications.
        */
       TscIni(argc, argv);

       /*
        * Create widgets, add callbacks, ...
        */

       ...

       option1 = XmCreatePushButtonGadget(parent, "Option 1", NULL, 0);
       XtAddCallback(option1, XmNactivateCallback, ButtonCB, "Command1");

       ...

       /*
        * Add a new input source corresponding to the communication
        * channel to the Command System.
        */
       XtAppAddInput(MnAppContext,
                     TscConsFileDesc(),
                     (XtPointer)XtInputReadMask,
                     connexio_sc,
                     NULL);
       XtAppMainLoop(MnAppContext);
}
```

11.1.2 Working with Motif (DEC/VMS environment)

This is not intended to be an tutorial to Motif, thus Motif and Xt functions are not explained. See *OSF/Motif Programmer's Guide* and *X Toolkit Intrinsics Programming Manual* for more information.

```
...

#include <Xm/Xm.h>

/*
* Prototypes of the input-providing user system.
* The file menus.h has been generated with
*      gu -userp menus fr fp
* where menus is the name of the input-providing user process,
* fr is the User Systems Resource Description File and
* fp is the Processes Description File
*/
#include "menus.h"
/*
* Prototypes of the functions of the communications module generated with
*      gu -inputp tscenv fr fp
*/
#include "tscenv.h"

...

/*
* Xt alternative source of input function.
* This function is called when the data is available in the
* communication channel. This is known via a EFN+IOSB
*/

static void connexio_sc(XtPointer client_data, int *fd, XtInputId *id)
{
   TscReb();
}

/*
* Xt Callback function.
* This function will be called when the user has clicked on a button.
* The button is associated with a command. The command which is
* associated with corresponds to the parameter client_data.
*/
void ButtonCB(Widget w, XtPointer client_data, XtPointer call_data)
{
   TscEnvToken((char *)client_data );
}
```

```
void main( int argc, char **argv )
{
        ...

        /*
         * Initialize communications.
         */
        TscIni(argc, argv);

        /*
         * Create widgets, add callbacks, ...
         */

        ...

        option1 = XmCreatePushButtonGadget(parent, "Option 1", NULL, 0);
        XtAddCallback(option1, XmNactivateCallback, ButtonCB, "Command1");

        ...

        /*
         * Add a new input source corresponding to the communication
         * channel to the Command System.
         */
        XtAppAddInput(MnAppContext,
                      TscConsEFN(),
                      TscConsIOSB(),
                      connexio_sc,
                      NULL);
        /*
         * Activate EFN+IOSB mechanism
         */
        TscArrenca();

        XtAppMainLoop(MnAppContext);
}
```

11.1.3 Working with Xlib (POSIX environment)

```
...

#include <X11/Xlib.h>
#include <X11/Xutil.h>

...
```

```
void main( void )
{
   Display     *display;
   XEvent      event;
   int         fd, xcn;
   ...

   /*
    * Initialize communications.
    */
   TscIni(argc, argv);

   /*
    * Initialize connection with default display.
    */
   display = XOpenDisplay(NULL);
   assert(display != NULL);
   /*
    * Get the file descriptor assotiated with display
    */
   xcn = XConnectionNumber(display);
   fd = TscConsFileDesc();

   ...

   for (;;) {
       /*
        * Select the input source.
        * The UNIX select system call can be used.
        */

       /*
        * Call the apropiate function depending on the file descriptor
        * that has data available.
        *      if data on xcn call XNextEvent
        *      if data on fd call TscReb()
        */
   }
}
```

12 Building the distributor system

- The distributor system is the process that distributes call-to-primitive requests issued by the command system to the application process (or input-providing application process) that implements the requested primitive.
- The distributor system depends on which are the systems that form the whole application. Thus, an specific distributor system must be generated for each application.
- The distributor system is generated with the utility

  ```
  gu -dist dist resources processes
  ```
- This tool generates the distributor process to file

  ```
  dist.c
  ```
- The file name must be '`dist`' if the tool that builds the loader shell script is used.
- The file '`dist.c`' must be compiled and linked to get the distributor process.

13 Building the application loader shell script

- A TSC application is composed of a set of systems. Three of them are provided by the TSC itself (TSC systems). These are

  `'sc'` Command System

  `'se'` Input System

  `'dist'` Distributor System. See Chapter 12 [Building Distributor], page 61, for details.
- The rest of the systems are built by the programmer (with the aid of the gu tool) and are called application systems.
- The systems that compose the applications are implemented as separate processes that communicates between them.
- A tool that loads all processes of an application is needed. These task will be performed by a shell script called application loader shell script.
- The application loader shell script is generated by typing

  `gu -loader` *shell_script_name app_desc_file proc_desc_file*

  where *shell_script_name* is the name of the file thaw will contain the generated shell script.
- Running the application is achieved running the application loader shell script. In POSIX environment this is

  ```
  shell_script_name
  ```

  in DEC/VMS environment this is

  ```
  @shell_script_name
  ```
- The application loader shell script understands the following options

  `-u` Usage message

  `-v` Print TSC version and generation date

  `-e` Verbose mode

  `-l` Processes stdout and stderr redirected to log files

Appendix A Toy application listings

A.1 The file '.tsccomrc'

```
{
       Funcions d'utilitat.
}

FUNCTION LlegirSReal( STRING missatge ) RETURN REAL WITHIN QUA IS
BEGIN
      RETURN ReadReal( missatge );
ENDFUNCTION

FUNCTION LlegirSEnter( STRING missatge ) RETURN INTEGER WITHIN QUA IS
BEGIN
       RETURN ReadInt( missatge );
ENDFUNCTION

FUNCTION LlegirQuadrat( STRING missatge ) RETURN QUADRAT WITHIN QUA;

FUNCTION ColorQ ( STRING missatge ) RETURN INTEGER WITHIN QUA IS
BEGIN
       RETURN ColorQuadrat( LlegirQuadrat( missatge ) );
ENDFUNCTION

FUNCTION PosQ ( STRING missatge ) RETURN PUNT WITHIN QUA IS
BEGIN
       RETURN PosQuadrat( LlegirQuadrat( missatge ) );
ENDFUNCTION

FUNCTION LongQ ( STRING missatge ) RETURN REAL WITHIN QUA IS
BEGIN
       RETURN LongQuadrat( LlegirQuadrat( missatge ) );
ENDFUNCTION


FUNCTION LlegirPunt( STRING missatge ) RETURN PUNT WITHIN QUA IS
{
       Llegeix un punt.
}
BEGIN
       RETURN FisicPunt( ReadPhPoint( missatge ) );
ENDFUNCTION


FUNCTION RealPunt( STRING missatge ) RETURN PUNT WITHIN QUA IS
```

```
{
        Llegeix les coordenades numeriques.
}
BEGIN
        RETURN CreaPunt( ReadReal( "Punt X" ), ReadReal( "PuntY") );
ENDFUNCTION


FUNCTION LlegirQuadrat( STRING missatge ) RETURN QUADRAT WITHIN QUA IS
{
        Llegeix punts fins que un es correspon amb un quadrat.
}
        QUADRAT q;
BEGIN
        q := QuinQuadrat( LlegirPunt( missatge ) );
        WHILE EsQuadratNul( q ) != 0 DO
                WriteString( "ERROR: El punt no correspon a cap quadrat" );
                q := QuinQuadrat( LlegirPunt( missatge ) );
        ENDWHILE;
        RETURN q;
ENDFUNCTION



{
        Accions d'utilitat.
}

PROCEDURE ErrorQuadrat( QUADRAT q, STRING missatge ) WITHIN QUA IS
{
        Si q es el QuadratNul escriu el missatge. Altrament no fa res.
}
BEGIN
        IF EsQuadratNul( q ) != 0 THEN
                WriteString( missatge );
        ENDIF;
ENDPROCEDURE



{
        Comandes.
}

COMMAND CREA WITHIN QUA IS
PUNT p;
QUADRAT q;
```

```
BEGIN
        p := CreaPunt(50.0, 50.0);
        q := CreaQuadrat(p, 10.0, 3);
ENDCOMMAND


COMMAND Quadrat WITHIN QUA IS
        QUADRAT q;
BEGIN
        q := CreaQuadrat( LlegirPunt( "Quadrat, centre" ),
                          LlegirSReal( "Quadrat, costat" ),
                          LlegirSEnter( "Quadrat, color" ) );
        ErrorQuadrat( q, "ERROR: Quadrat Invalid" );
ENDCOMMAND


COMMAND Consulta WITHIN QUA IS
        QUADRAT q;
BEGIN
        q := LlegirQuadrat( "Consulta, Quin quadrat?" );
        WriteString( "Centre, Longitud, Color" );
        WriteReal( CoorX( PosQuadrat( q ) ) );
        WriteReal( CoorY( PosQuadrat( q ) ) );
        WriteReal( LongQuadrat( q ) );
        WriteInt( ColorQuadrat( q ) );
ENDCOMMAND


COMMAND Esborra WITHIN QUA IS
        QUADRAT q;
BEGIN
        q := LlegirQuadrat( "Esborra, Quin quadrat?" );
        BorraQuadrat( q );
ENDCOMMAND


COMMAND Escala WITHIN QUA IS
        QUADRAT q;
        QUADRAT p;
BEGIN
        p := LlegirQuadrat( "Escala, Quin quadrat?" );
        q := CreaQuadrat( PosQuadrat( p ),
                          LongQuadrat(p)*LlegirSReal("Escala, Factor escalat"),
                          ColorQuadrat( p ) );
        IF EsQuadratNul( q ) != 0 THEN
                WriteString( "ERROR: Quadrat Invalid" );
        ELSE
                BorraQuadrat( p );
        ENDIF;
ENDCOMMAND
```

```
COMMAND Color WITHIN QUA IS
        QUADRAT q;
        QUADRAT p;
        PUNT    a;
        REAL    l;
        INTEGER c;
BEGIN
        p := LlegirQuadrat( "Canvi Color, Quin quadrat?" );
        a := PosQuadrat( p );
        l := LongQuadrat( p );
        c := LlegirSEnter( "Canvi Color, Nou color" );
        BorraQuadrat( p );
        q := CreaQuadrat( a, l, c );
        ErrorQuadrat( q, "ERROR: Quadrat Invalid" );
ENDCOMMAND
```

A.2 The file 'context.h'

```
#define CONTEXT_NUL             0
#define CONTEXT_QUA             1

#define DEFAULT_CONTEXT         1
```

A.3 The file 'initial.sc'

```
DEFINE CONTEXT QUA END DEFINE

DEFINE INITIAL CONTEXT QUA END DEFINE

DEFINE VALID CONTEXT CHANGE END DEFINE

DEFINE TYPE
  QUADRAT IS INTEGER;
  PUNT IS REAL, REAL;
END DEFINE
```

```
DEFINE USER
  { Square related operations }
  FUNCTION CreaQuadrat( PUNT p, REAL c, INTEGER color) RETURN QUADRAT
     WITHIN QUA;
  PROCEDURE BorraQuadrat( QUADRAT q )
     WITHIN QUA;
  FUNCTION QuinQuadrat( PUNT p ) RETURN QUADRAT
     WITHIN QUA;
  FUNCTION ColorQuadrat( QUADRAT q ) RETURN INTEGER
     WITHIN QUA;
  FUNCTION PosQuadrat( QUADRAT q ) RETURN PUNT
     WITHIN QUA;
  FUNCTION LongQuadrat( QUADRAT q ) RETURN REAL
     WITHIN QUA;
  FUNCTION EsQuadratNul( QUADRAT q ) RETURN INTEGER
     WITHIN QUA;

  { Point related operations }
  FUNCTION CoorX( PUNT p ) RETURN REAL
     WITHIN QUA;
  FUNCTION CoorY( PUNT p ) RETURN REAL
     WITHIN QUA;
  FUNCTION CreaPunt( REAL x, REAL y ) RETURN PUNT
     WITHIN QUA;
  FUNCTION FisicPunt( PUNTF p ) RETURN PUNT
     WITHIN QUA;
END DEFINE
```

A.4 The file 'quadrador.h'

```
void    IniUsuari( int argc, char **argv );
void    FiUsuari( void );
void    CanviContext( int ContextPrevi, int ContextNou );
void    CreaQuadrat( /* PUNT */ float p0, float p1 /* Fi PUNT */,
                     float c, int color,
                     /* QUADRAT */ int *RetVal0 /* Fi QUADRAT */ );
void    BorraQuadrat( /* QUADRAT */ int q0 /* Fi QUADRAT */ );
void    QuinQuadrat( /* PUNT */ float p0, float p1 /* Fi PUNT */,
                     /* QUADRAT */ int *RetVal0 /* Fi QUADRAT */ );
void    ColorQuadrat( /* QUADRAT */ int q0 /* Fi QUADRAT */,
                      int *RetVal );
void    PosQuadrat( /* QUADRAT */ int q0 /* Fi QUADRAT */,
                    /* PUNT */ float *RetVal0, float *RetVal1 /* Fi PUNT */ );
void    LongQuadrat( /* QUADRAT */ int q0 /* Fi QUADRAT */,
                     float *RetVal );
void    EsQuadratNul( /* QUADRAT */ int q0 /* Fi QUADRAT */,
                      int *RetVal );
```

```
void    CoorX( /* PUNT */ float p0, float p1 /* Fi PUNT */,
               float *RetVal );
void    CoorY( /* PUNT */ float p0, float p1 /* Fi PUNT */,
               float *RetVal );
void    CreaPunt( float x, float y,
                  /* PUNT */ float *RetVal0, float *RetVal1 /* Fi PUNT */ );
void    FisicPunt( /* PUNTF */ int p0, float p1, float p2 /* Fi PUNTF */,
                   /* PUNT */ float *RetVal0, float *RetVal1 /* Fi PUNT */ );
```

A.5 The file 'quadre.c'

```
#include <math.h>
#include <float.h>
#include <limits.h>
#include "quadrador.h"
#include "context.h"
#include "tscg.h"


/*
* Constants del modul
*/
#define MAX 200
#define NUL -1
#define TRUE 1
#define FALSE 0
#define EPSILON 2.0


/*
* Estructures de dades
*/
struct quadre {
  float x,y;
  float lon;
  int color;
  int ident;
};

static struct quadre tq[MAX];
static int pbuit;
static int identificador;



/*
*      Funcions de control
*/
```

```
void CanviContext(int a, int b)
{}

void FiUsuari(void)
{}

void IniQuadre(void);

void IniUsuari( int argc, char **arcv )
{
        float r[256], g[256], b[256];
        float zero = 0.0;
        float cent = 100.0;
        float setcents = 700.0;
        int un = 1;
        int i;

        FORTRAN(g_ini)();

        /* carreguem paleta */
        r[0] = 1.0; g[0] = 0.0; b[0] = 0.0;
        r[1] = 0.0; g[1] = 1.0; b[1] = 0.0;
        r[2] = 0.0; g[2] = 0.0; b[2] = 1.0;
        r[3] = 1.0; g[3] = 1.0; b[3] = 0.0;
        r[4] = 1.0; g[4] = 0.0; b[4] = 1.0;
        r[5] = 0.0; g[5] = 1.0; b[5] = 1.0;
        r[6] = 1.0; g[6] = 0.0; b[6] = 0.0;
        for (i=7; i<256; i++)
                r[i] = g[i] = b[i] = 0.0;
        FORTRAN(g_pal)(r,g,b);

        /* creem finestra */
        FORTRAN(g_cregra)(&un,
                 &zero, &zero, &setcents, &setcents,
                 &zero, &zero, &cent, &cent);
        FORTRAN(g_switch)(&un);
        FORTRAN(g_syncro)();
}



/*
*       Funcions privades del modul
*/
static int cerca(int q)
{
         int i;
```

```
        i = 0;
        while ( i < pbuit && tq[i].ident != q ) i++;
        return i;
}


void pinta( float x, float y, float l, int c)
{
  float dx, dy;
  int un = 1;

  /* pinta */
  FORTRAN(g_lcol)( &c );
  dx = x - l/2;
  dy = y - l/2;
  FORTRAN(g_move)( &dx, &dy );
  dy += l;
  FORTRAN(g_draw)( &dx, &dy );
  dx += l;
  FORTRAN(g_draw)( &dx, &dy );
  dy -= l;
  FORTRAN(g_draw)( &dx, &dy );
  dx -= l;
  FORTRAN(g_draw)( &dx, &dy );

  FORTRAN(g_marker)(&un, &x, &y);
  FORTRAN(g_syncro)();
}

/*
* Primitives del modul
*/

void IniQuadre(void)
{
  int i;

  identificador = pbuit = 0;
}


void CreaQuadrat(float x, float y, float l, int col, int *ret)
{
   int i;
   float dx, dy;
```

```
    /* verifiquem que el quadre cap */
    if ( (x - l/2.0) < 0.0   ||
         (x + l/2.0) > 100.0 ||
         (y - l/2.0) < 0.0   ||
         (y + l/2.0) > 100.0 ) {
       *ret = NUL;
       return;
    }

    /* cerquem lloc buit */
    if (pbuit >= MAX) {
       *ret = NUL;
       return;
    } else {
       /* reservem el buit */
       i = pbuit++;
    }

    /* assignem valors */
    tq[i].x = x;
    tq[i].y = y;
    tq[i].lon = l;
    tq[i].color = abs(col % 6);
    tq[i].ident = identificador++;

    pinta(x,y,l,tq[i].color);

    /* retornem el valor del quadre */
    *ret = tq[i].ident;

}


void BorraQuadrat(int q)
{
        int i;

        q = cerca(q);
        /* verifiquem quadre existent */
        if ( q < pbuit ) {
                pinta(tq[q].x, tq[q].y, tq[q].lon, 7);
                for(i=q; i < pbuit-1; i++)
                        tq[q] = tq[q+1];
                pbuit--;
        }
}
```

```
void EsQuadratNul( int q , int *res )
{
        *res = (q == NUL) ? (1) : (0);
}



void QuinQuadrat( float x, float y, int *res )
{
        int i;
        float d;

        i = 0; d = FLT_MAX;
        while (d > EPSILON  && i < pbuit) {
                d = (tq[i].x - x) * (tq[i].x - x) +
                    (tq[i].y - y) * (tq[i].y - y);
                i++;
        }

        if (d <= EPSILON)
                *res = tq[i-1].ident;
        else
                *res = NUL;
}



void ColorQuadrat( int q, int *res )
{
        *res = tq[ cerca(q) ].color;
}



void PosQuadrat( int q, float *x, float *y )
{
        int i;

        i = cerca(q);
        *x = tq[i].x;
        *y = tq[i].y;
}



void LongQuadrat( int q, float *res )
{
        *res = tq[ cerca(q) ].lon;
}
```

```
void CoorX( float x, float y, float *res )
{
        *res = x;
}


void CoorY( float x, float y, float *res )
{
        *res = y;
}


void CreaPunt( float x, float y, float *rx, float *ry )
{
        *rx = x;
        *ry = y;
}


void FisicPunt( int f, float x, float y, float *rx, float *ry )
{
        int un = 1;

        FORTRAN(g_inqreal)( &un, &x, &y, rx, ry );
}
```

A.6 The file 'trans.sc'

```
COMMAND Transllada WITHIN QUA IS
        QUADRAT q;
        QUADRAT p;
BEGIN
        p := LlegirQuadrat( "Transllada, Quin quadrat?" );
        q := CreaQuadrat( LlegirPunt( "Transllada, Nou centre" ),
                          LongQuadrat( p ),
                          ColorQuadrat( p ) );
        IF EsQuadratNul( q ) != 0 THEN
                WriteString( "ERROR: Quadrat Invalid" );
        ELSE
                BorraQuadrat( p );
        ENDIF;
ENDCOMMAND
```

Appendix B DEC/VMS vs. POSIX environment

Working in the DEC/VMS environment or in the POSIX environment is acomplished in a very similar way. Nonetheless, due to deep diferences between both environments, TSC programmer and customizer will observe some differences. These differences are sketched below:

1. The set of operations available to build input-providing application systems is distinct when working on DEC/VMS. In fact, while POSIX operations relay on the file descriptor concept, DEC/VMS equivalent operations relay on the event flag mechanism. See Chapter 11 [Building Input User Process], page 53.
2. The application loader generated by gu is a Bourne compatible shell script in POSIX environment. In DEC/VMS environment it is a proper command procedure. Because of this, the POSIX invocation is done bye typing the filename while in DEC/VMS environment the @ symbol should precede the command procedure file name. See Chapter 13 [Building Loader], page 63.
3. There may exist differences in the TSC installation on your site. Ask your system manager about.

Appendix C Installing TSC

C.1 POSIX installation

TSC is distributed in source form. All the files are packed in a tar file usually named '`TSCv*.tar`' where the star means the version number. Suppose that the file is named '`TSCv1-2.tar`', then it should be expanded doing

```
$ tar xvf TSCv1-2.tar
```

after that, a directory named '`tsc`' is created which contains all the source files. The directory structure resembles the following:

```
tsc +--> Makefile
    +--> Make.sun4
    +--> Make.hp
    +--> Make.hp_cc
    +--> comp.awk
    +--> comandes +--> Makefile
    |              +--> ...
    +--> contrib +--> Makefile
    |            +--> menus +--> Makefile
    |            |          +--> ...
    |            +--> ...
    +--> doc +--> Makefile
    |        +--> README
    |        +--> ...
    +--> drivers +--> Makefile
    |            +--> ...
    +--> entrada +--> Makefile
    |            +--> ...
    +--> gentsc +--> Makefile
    |           +--> ...
    +--> tads +--> Makefile
              +--> ...
```

Once the is tar file is extracted, installation process[1] can begin. Please follow the points below:

[1] Installation process requires a GNU `make` compatible `make` and a `flex` compatible utility. Thus if your installation doesn't have this tool, please get GNU `make` and `flex` from *Free Software Foundation* and install it.

1. Change the current directory to the root directory of the extracted tree:
   ```
   $ cd tsc
   ```
2. Set the host type where you want to compile. Nowadays, allowed host types are:

   `sun4` Sun workstations under Solaris 1.x and running GNU `gcc`.

   `hp` HP workstations running `hp-ux` and using the native C compiler.

   `hp_cc` HP workstations running `hp-ux` and using the GNU `gcc`.

   To set the correct host define the `HOSTTYPE` environment variable to the host type identifier. For instance, using Bourne shell the `sun4` host type can be set doing
   ```
   $ HOSTTYPE=sun4; export HOSTTYPE
   ```
   If your host type isn't defined, you should establish a new identifier (say `sg`) and copy any one of the files named '`Make.*`' to '`Make.sg`'. For instance,
   ```
   $ cp Make.sun4 Make.sg
   ```
   now, edit the '`Make.sg`' file and try to define the correct environment following the comments in it. When you have finished, define the new `HOSTTYPE` value.
3. The installator must decide where TSC applications should live. This should be done before TSC is generated because this information is needed during compilation time. By default TSC is installed in the '`/usr/local/tsc`' directory and this is the recommended place. If you choose the default, nothing special should be done when using `make`. Otherwise, you must select two directories (that can be the same) where TSC will be installed. The first one containing the executables and the second one containing other files. Let's say, for instance, that the first is '`/usr/local/bin`' and the second '`/usr/local/etc`'. Then every invocation of `make` from now to the very end of the installation process should be done as in the example:
   ```
   $ make \
   INSTALL_TSC_EXE=/usr/local/bin \
   INSTALL_TSC_AUX=/usr/local/etc \
   chkvars
   ```
4. Configuration variables can be verified by typing the command:
   ```
   $ make chkvars
   ```
5. If variable contents are correct start TSC generation by typing
   ```
   $ make all
   ```
6. TSC generation should run fine. After that, installation process can be done by typing
   ```
   $ make install
   ```
   Remember that you need permission to create files in the directories where TSC is being installed.

C.2 DEC/VMS installation

1. The DEC/VMS version of TSC is automatically built from the POSIX one. To get a tarfile containing the DEC/VMS version, you should first get a copy of the TSC distribution directory tree on a Sun host[2] as done in the POSIX installation. Then, you must do the following steps:
   1. Change the working directory to 'tsc'.
   2. Execute the following GNU make target:
      ```
      $ make vms
      ```
   3. The result must be a tarfile named 'TSCvms.tar'
2. Now, the file 'TSCvms.tar' must be copied to the DEC/VMS host where you want to install it. Use any kind procedure (ftp for instance).
3. Now, in the DEC/VMS host, change the default directory to the place where 'TSCvms.tar' has been copied. Be sure that this directory doesn't contain anything else.
4. Using the tar command in VMS/POSIX environment, extract the 'TSCvms.tar' files doing:
   ```
   VMS> posix
   POSIX> tar xvf TSCvms.tar
   ...
   POSIX> exit
   ```
   After that, the directory must contain the extracted files.
5. Start building and installing the product by typing:
   ```
   VMS> @compile
   ```
   This command procedure will ask for the directory where TSC should be installed. Please answer an absolute path name for the directory NOT A RELATIVE ONE. Be careful with the directory protections: you must have permission to create files in the directory. For instance you can answer:
   ```
   Installation directory for TSC: DISK$A:[TOOLS.TSC]
   ```
   Compilation and installation process begins automatically.
6. After that, installation is complete. If an user wants to use TSC, he must include in his login file a call to execute the command procedure 'INITSCENV'. This file is located in the TSC installation directory. Thus a line like the following must be added to the 'LOGIN.COM':
   ```
   $ @ DISK$A:[TOOLS.TSC]INITSCENV
   ```

[2] In future versions any POSIX host will be sufficient

Appendix D Formal syntax descriptions

D.1 Command definition language syntax

```
command         ::= COMMAND command_name WITHIN context_list IS
                        var_decl
                    BEGIN
                        sentences
                    ENDCOMMAND

proc_proto      ::= PROCEDURE proc_name "(" param_list ")"
                    WITHIN context_list ";"

procedure       ::= PROCEDURE proc_name "(" param_list ")"
                    WITHIN context_list IS
                        var_decl
                    BEGIN
                        sentences
                    ENDPROCEDURE

func_proto      ::= FUNCTION func_name "(" param_list ")" RETURN type_name
                    WITHIN context_list ";"

function        ::= FUNCTION func_name "(" param_list ")" RETURN type_name
                    WITHIN context_list IS
                        var_decl
                    BEGIN
                        sentences
                        RETURN expression ";"
                    ENDFUNCTION

var_decl        ::= { type_name variable ";" }

param_list      ::= [ param { "," param } ]

param           ::= type_name param_name

context_list    ::= context_name { "," context_name }

sentences       ::= { sentence ";" }
```

```
sentence        ::= assignement
                  | condicional
                  | case
                  | iteration
                  | context_chng
                  | procedure_call

assignement     ::= variable ":=" expression

condicional     ::= IF expression THEN
                        sentences
                    ENDIF
                  | IF expression THEN
                        sentences
                    ELSE
                        sentences
                    ENDIF

case            ::= CASE expression IS
                        { a_case }
                        [ otherwise ]
                    ENDCASE

a_case          ::= WHEN constant DO
                        sentences

otherwise       ::= OTHERWISE sentences

iteration       ::= WHILE expression DO
                        sentences
                    ENDWHILE

context_chng    ::= CONTEXT context DO
                        sentences
                    ENDCONTEXT

procedure_call  ::= proc_name "(" expr_list ")"

expr_list       ::= [ expression { "," expressio } ]
```

```
expression      ::= "(" expression ")"
                  | expression OR expression
                  | expression AND expression
                  | expression "<" expression
                  | expression "<=" expression
                  | expression ">" expression
                  | expression ">=" expression
                  | expression "=" expression
                  | expression "!=" expression
                  | NOT expression
                  | expression "+" expression
                  | expression "-" expression
                  | expression "*" expression
                  | expression "/" expression
                  | expression "%" expression
                  | "-" expression
                  | variable
                  | constant
                  | function_call

constant        ::= integer | real | string

integer         ::= [ "-" ] digit { digit }

real            ::= [ "-" ] digit { digit } "." digit { digit }

string          ::= """ { character } """

function_call   ::= func_name "(" expr_list ")"

variable        ::= identifier

param_name      ::= identifier

command_name    ::= identifier

proc_name       ::= identifier

func_name       ::= identifier

context_name    ::= identifier

type_name       ::= INTEGER | REAL | STRING | identifier

identifier      ::= letter { letter | digit | "_" }
```

D.2 Resource description file syntax

```
file            ::= def_context
                    def_ini_context
                    def_cha_context
                    def_type
                    def_user

def_context     ::= DEFINE CONTEXT
                            context_list
                    END DEFINE

context_list    ::= context_name { "," context_name }

context_name    ::= identifier

def_ini_context ::= DEFINE INITIAL CONTEXT context_name END DEFINE

def_cha_context ::= DEFINE VALID CONTEXT CHANGE
                            { context_list TO context_name ";" }
                    END DEFINE

def_type        ::= DEFINE TYPE
                            { type_name IS elem_type_list ";" }
                    END DEFINE

type_name       ::= identifier

elem_type_list  ::= elem_type { "," elem_type }

elem_type       ::= INTEGER | REAL | STRING | PUNTF

def_user        ::= DEFINE USER
                            { subprog }
                    END DEFINE

subprog         ::= procedure | function

procedure       ::= PROCEDURE proc_name "(" param_list ")"
                    WITHIN context_list ";"

proc_name       ::= identifier

function        ::= FUNCTION funct_name "(" param_list ")"
                    RETURN type_name WITHIN context_list ';'

funct_name      ::= identifier

param_list      ::= [ parameter { "," parameter } ]

parameter       ::= type_name parameter_name
```

```
parameter_name  ::= identifier

identifier      ::= letter { letter | digit | "_" }
```

D.3 Proces-Context Mapping File syntax

```
file            ::= line { NEWLINE line }

line            ::= process_name context_id { context_id }

process_name    ::= identifier

context_id      ::= identifier

identifier      ::= letter { letter | digit | "_" }
```

Appendix E Known errors

1. When somebody aborts a virtual machine process. The virtual machine doesn't recover gracefully the correct state.
2. The journaling system is hardware is hardware-dependent. Therefore, the journal files cannot be re-executed at distinct sites. This is only true for the applications using graphic input.
3. The journaling system doesn't correctly support a command abort.
4. When somebody has defined a subprogram prototype but the implementation of the former hasn't been given in a command definition file, the compiler only emits a warning about. An error should be raised instead.
5. Sometimes, the command compiler produces wrong results when a function is defined and a syntactical error exists in the parameter list. The bug is related to the mechanism used to insert the function being defined into the correct family list.
6. When a subprogram without known implementation is called —at run time— it simply returns. It should abort the current command and print an error message.
7. The command completion mechanism has an undesirable side effect: it is impossible to execute a command with a name that is a prefix of another command.

Appendix F Wish list

1. It can be useful to allow the boolean data type as a predefined type.
2. Review the command completion mechanism:
   - Allow functions to be included in the completion mechanism. Study a clever mechanism allowing only the subset of correct functions to be completed.
   - Modify the mechanism to achieve something similar to the `tcsh` completion mechanism.
3. Study the case-sensitivity of the commands. Is it possible to give a good solution for all users?
4. Provide the applications builder with tools that allow him to know accounting data for the application: statistics about the commands and primitives issued by the user, statistics about the response time of the primitives and commands, statistics about memory usage, etc.
5. Generate code for languages other than ANSI-C (C++, F...)
6. Change the prompt for input system depending of context.
7. Add some metacommands to let the user know space left in the repository, symbol table and other fixed size structures.
8. Reduce fixed size data structures to the minimum.
9. Study the problem of multilingual sets of commands. Study what about the standards.
10. Improve the treatment of messages (errors, warnings, ...) in the compilers used by the system.
11. Add a user-friendly feature to allow the user to define commands interactively.
12. Make the systems communicate via network.
13. Let the command interpreter to parse and understand expressions. How to allow the user defined functions to participate in this expressions?
14. Make processes and contexts and independent matter as it really is.
15. Study the complex problem of the input functions. Don't forget the relationship with the interruptibility.
16. Study a new communication protocol solving some of the main problems: crash recovery and others.
17. Add the ability to generate `const` modifiers for the automatically generated ANSI-C code.
18. Study how the command syntax that the user should follow can be flexibilized. I'm talking about what the user should type to run a command. This is the dialog syntax.
19. Think about adding a window showing a selectable set of replaceable functions. This would allow the user to easily perform a substitution.
20. Wrote a set of man pages for the TSC.

Appendix G Version to version change log

G.1 From V1.1 to V1.2

1. Communication mechanism has been rewritten layering on pipes.
2. Implementation of the context mechanism has been partially redesigned and reimplemented in order to achieve extensibility on the number of allowable contexts. Now, increasing the number of allowable context should take little work.
3. The number of allowable contexts has been raised to 32 (on most of the architectures).
4. The set of tools allowing the generation of part of the application have been completely redesigned and reimplemented. All the old tools (`gu`, `gc`, `gd`, `gm`) have been substituted by an integrated tool (`gu`) written in C that can do the same work. The main goal achieved with this change is the elimination of a lot of operating-system dependent scripts thus making the new product more transportable, fast and reliable.
5. The generated scripts that load the application have been enhanced. Now, several options are ofered about the verbosity of the application and the creation of log files is optional. Care has been taken in making the script robust an the use of `kill...` has been replaced by the less awkful `^C`. The new generated script assures that any number of applications can be executed simultaneously in distinct working directories. If an application is working already in the same directory, the starting application detects it and aborts gracefully.
6. Now, generation tools don't produce object files or executable ones. This policy don't enforces the application's designer to be chained to a given compiler as in previous version. In addition, a greatest independence of the tools from the operating system is achieved.
7. Because of the change in the communication mechanism, now time consuming processes have been eliminated. Thus the consume of CPU time has drastically decreased.
8. A lot of work has been spent in achieving a higher degree of portability. Now all the TSC set falls into the POSIX.1 and X11R4 standard thus allowing to be ported easily to a great number of architectures. Modules have been classified into categories depending on the standard that they require. This policy assures an easy maintenance without breaking the portability rules.
9. Installation policy has been redesigned. Now public executables of TSC can be installed into an path contained directory (say for instance '`/usr/local/bin`'). This makes the TSC user not to worry about where the utilities are. Other useful files and internal executables are located in a private directory (say for instance '`/usr/local/tsc`').
10. Work has been spent in offering a common aspect to the commands of TSC. Now most of the comma's accept minimum set of flags and all of them can show a little help on error or user demand.

11. Project organization has been notably improved. Now it's easy to maintain a multi-architecture set of executables and installation procedures have been automatized.
12. An extensive user manual has been written. The manual contains all the information needed by an application designer that wants to use TSC. The manual is intended to be useful to the novice user as well as to the expert one.
13. TSC has been ported to DEC/VMS. Communication drivers have been rewritten using mailboxes and a slightly distinct set of primitives. Therefore, the Event Flag synchronization mechanism is now supported in the DEC/VMS configuration. Given that improvement, it is possible to write portable applications spending a small effort.

Index

F

G

I

L

M

N

O

P

R

S

T

U

V

W