



# **SIMIOBOARD SC-PASCAL**

## **Programming user manual**

Doc. Ver.: SIMIOB-HD V.1.2 EN

Date: 18/06/2015

URL: [www.simioboard.com](http://www.simioboard.com)



# TABLE OF CONTENT

1. What SC-PASCAL is? .....	6
2. SC-PASCAL syntax.....	6
3. Type of data.....	9
3.1. Integers.....	9
3.2. Reals .....	9
3.3. Character (Char) .....	10
3.4. Logic (Boolean).....	10
3.5. String .....	10
4. Basic structure of a SC-PASCAL program.....	11
5. Alternative control structures .....	11
5.1. IF structure .....	11
5.2. CASE structure.....	12
6. Repetitive control structures .....	13
6.1. WHILE structure .....	13
6.2. REPEAT structure.....	14
6.3. FOR structure .....	15
6.4. When to use WHILE/REPEAT/FOR structures .....	16
7. Procedures and functions .....	16
7.1 Debug .....	16
7.2 Writelocp.....	17
7.3 RegIOCPOffset .....	18
7.4 InitOffsetFSUIPC .....	18
7.5 WriteFSUIPC .....	18
7.6 ReadFSUIPC .....	19
7.7 RegEncoder .....	19
7.8 WriteDisplays .....	20
7.9 SetDisplayIntensity.....	21
7.10 WriteOut .....	21
7.11 ReadIOCP .....	22
7.12 TestBit .....	22



7.13 SetBit .....	23
7.14 ChangeBit .....	23
7.15 WriteIOCPKey .....	24
7.16. CreateTimer.....	24
7.17. SetTimer .....	25
7.18. GetTimer .....	25
7.19. GetEncoder.....	26
7.20. SetEncValue.....	26
7.21. WriteDisplaysSwap.....	27
7.22. RegIFLYOffset .....	27
7.23. WriteIFLY .....	28
7.24. ReadIFLY .....	28
7.25. EnableADCChannel.....	28
7.26. WritelocpServer .....	29
7.27. ReadIniFile .....	29
7.28. WriteIniFile .....	30
7.29. SendKey .....	31
7.30. ReadAdc .....	32
7.31. EnableServo.....	33
7.32. WriteServo .....	33
8. Events .....	35
8.1 OnInputChange .....	35
8.2 OnInputChangeAll .....	35
8.3 OnADCChange .....	36
8.4 OnEncoderChange.....	36
8.5 OnIOCPConnect.....	37
8.6 OnIOCPConnectClosed .....	37
8.7 OnIOCPConnectFail .....	37
8.8 OnIOCPChange .....	38
8.9 OnFSUIPCError .....	38
8.10 OnFSUIPConnect .....	39
8.11 OnFSUIPCDisconnect .....	39



8.12 OnFSUIPCChange.....	39
8.13. OnTimern.....	40
8.14. OnIFLYConnect .....	40
8.15. OnIFLYChange .....	41
8.16. OnIFLYDisconnect .....	41
8.17. OnIOCPServerChange.....	41
9. User Interface (UI) .....	43
9.1 Menu bar .....	43
9.1.1 Exit.....	43
9.1.2 Load Project.....	44
9.1.3 New Project .....	44
9.1.4 Close Project.....	44
9.1.5 Save Project.....	44
9.1.6 Save File.....	44
9.1.7 Save All Files .....	44
9.1.8 Update Hardware .....	45
9.1.9 RUN PROJECT .....	45
9.1.10 STOP PROJECT .....	45
9.1.11 COMPILE PROJECT .....	45
9.2 Projects.....	46
9.3 Connections.....	48
9.3.1 Connection with FSUIPC.....	48
9.3.2 Connection with IOCP .....	49
9.3.3 Handling connections.....	50
9.4 Codes.....	50
9.4.1 New Code .....	51
9.4.2 Add Code .....	51
9.5 Hardware.....	52
9.5.1 Test Main.....	53
9.5.2 Test EXP 64 INPUTS .....	53
9.5.3 Test EXP 32 INPUTS 32 OUPUTS.....	54
9.5.4 Test EXP 64 OUPUTS.....	54



9.5.5 Test EXP 32 DISPLAYS .....	55
9.5.6. Test USB-JOY .....	55
9.5.7. Test 8Servo-8Adc.....	55



## 1. What SC-PASCAL is?

SC-PASCAL is a language to develop scripts. It's based 100% in Pascal language. That's why the syntax is very similar.

The goal of SC-PASCAL is let us design scripts that defines a common interface between different hardware devices and the simulation software. SC-Pascal includes specific functions to make this easy.

## 2. SC-PASCAL syntax

The syntax of SC-PASCAL is an extract of the functions available at PASCAL, so any programmer can conclude that SC-PASCAL is the same as PASCAL but with some limited capabilities and with other adapted to achieve an optimal communication between hardware and software.

Here is the syntax for SC-PASCAL:

```
Program -> [ PROGRAM Ident ';' ]  
          [ UsesClause ]  
          Block '
```

```
UsesClause -> USES (String/,)... '
```

```
Block -> [DeclSection]...  
        CompoundStmt
```

```
DeclSection -> ConstSection  
              -> VarSection  
              -> ProcedureDeclSection
```

```
ConstSection -> CONST (ConstantDecl)...
```

```
ConstantDecl -> Ident '=' Expression '
```

```
VarSection -> VAR (VarList ';' )...
```

```
VarList -> Ident / ',' ... ':' TypeIdent [InitValue]
```

```
TypeIdent -> Ident  
           -> Array
```

```
Array -> ARRAY '[' ArrayDim / ',' ... ']' OF Ident
```

```
ArrayDim -> Expression..Expression  
          -> Expression
```

```
InitValue -> '=' Expression
```

```
Expression -> SimpleExpression [RelOp SimpleExpression]...
```

```
SimpleExpression -> ['-'] Term [AddOp Term]...
```



Term -> Factor [MulOp Factor]...

Factor -> Designator

- > UnsignedNumber
- > String
- > '(' Expression ')'
- > NOT Factor
- > '[' SetConstructor ']'

SetConstructor -> SetNode/','...

SetNode -> Expression ['..' Expression]

RelOp -> '>'

- > '<'
- > '<='
- > '>='
- > '<>'
- > '=='
- > IN
- > IS

AddOp -> '+'

- > '-'
- > OR
- > XOR

MulOp -> '\*'

- > '/'
- > DIV
- > MOD
- > AND
- > SHL
- > SHR

Designator -> ['@'] Ident ['.' Ident | '[' ExprList ']' | '(' ExprList ')']...

ExprList -> Expression/','...

Statement -> [SimpleStatement | StructStmt]

StmtList -> Statement/';'...

SimpleStatement -> Designator

- > Designator ':=' Expression
- > BREAK | CONTINUE | EXIT

StructStmt -> CompoundStmt

- > ConditionalStmt
- > LoopStmt
- > TryStmt
- > WithStmt

CompoundStmt -> BEGIN StmtList END



ConditionalStmt -> IfStmt  
                  -> CaseStmt

IfStmt -> IF Expression THEN Statement [ELSE Statement]

CaseStmt -> CASE Expression OF CaseSelector/';'... [ELSE Statement][';'] END

CaseSelector -> SetConstructor ':' Statement

LoopStmt -> RepeatStmt  
             -> WhileStmt  
             -> ForStmt

RepeatStmt -> REPEAT StmtList UNTIL Expression

WhileStmt -> WHILE Expression DO Statement

ForStmt -> FOR Ident ':=' Expression ToDownto Expression DO Statement

ToDownto -> (TO | DOWNTO)

TryStmt -> TRY StmtList (FINALLY | EXCEPT) StmtList END

WithStmt -> WITH (Designator/,...) DO Statement

ProcedureDeclSection -> ProcedureDecl  
                         -> FunctionDecl

ProcedureDecl -> ProcedureHeading ';' Block ';'

ProcedureHeading -> PROCEDURE Ident [FormalParameters]

FunctionDecl -> FunctionHeading ';' Block ';'

FunctionHeading -> FUNCTION Ident [FormalParameters] ':' Ident

FormalParameters -> '(' FormalParam/';'...' )'

FormalParam -> [VAR | CONST] VarList



### 3. Type of data

Like any other programming language, SC-PASCAL has its own predefined data. All of them are defined in the following sections.

#### 3.1. Integers

An integer variable is a number with no decimals. Depending on the range, it can be negative or not, as shown in the following table:

Name	Range (from...to)	Size (bytes)	Format
Integer	-32768 a 32767	2	Entero con signo
Word	0 a 65535	2	Entero sin signo
Byte	0 a 255	1	Entero corto sin signo
Longint	-2147483648 a 2147483647	4	Entero largo con signo

Example:

```
VAR
    i : integer;

//Main
BEGIN
    i := 3;
END.
```

#### 3.2. Reals

A real variable is the one that contains decimals. Again, depending on the range, it can be negative or not, as shown in the following table:

Name	Range (from...to)	Size (bytes)	Number of figures
Real	2,9E-39 a 1,7E38	6	11-12
Single	1,5E-45 a 3,4e38	4	6-7
Double	5,0E-307 a 1,7E307	8	15-16
Extended	1,9E-4932 a 1,1E493210	10	19-20

Example:

```
VAR
    i : real;

//Main
BEGIN
    i := 3.1;
END.
```



### 3.3. Character (Char)

This type of variable depends on the characters code set on pur computer. The most popular one is ASCII code. The Char type variable can only store one single character between quotation marks.

Example:

```
VAR
    c : char;

//Main
BEGIN
    c := 'M';
END.
```

### 3.4. Logic (Boolean)

The Boolean variable con only store two values: True or False.

Example:

```
VAR
    b : boolean;

//Main
BEGIN
    b := false;
END.
```

### 3.5. String

The string variable can store a sequence of characters, up to 255. One string is defined between quotation marks.

Example:

```
VAR
    s : string;

//Main
BEGIN
    s := 'abcd';
END.
```



## 4. Basic structure of a SC-PASCAL program

A SC-PASCAL program consists on an optional area where the name of the program is set, another area for declarations and the main body, as shown in the following schema:

<b>PROGRAM</b> ident;	(Optional. The name of the program is set here)
<b>USES</b> libs;	(Optional. Use of libraries)
<b>CONST</b> const	(Optional. Constants declarations)
<b>VAR</b> vars	(Optional. Variable declarations)
<b>//Funciones o procedimientos</b>	(Optional. Functions declarations)
<b>BEGIN</b>	(The main program starts here...)
<b>END.</b>	(...and ends here)

where

PROGRAM, USES, CONST, VAR, BEGIN, END are reserved words.

ident: string that indicates the name of the program.

libs: list of libraries to be used.

consts: list of declared constants.

vars: list of declared variables.

## 5. Alternative control structures

These are the ones that redirect the execution of the program to one group of sentences or other, depending on the result of a condition. The available alternative control structures are explained in the following sections.

### 5.1. IF structure

An IF structure has a mandatory first part and an optional second one. One or other will be executed depending on the result of the condition. If it is True, then the program will run the first part, but if it's false, the second part will be the one executed. The structure is:

<b>IF</b> (condition) <b>THEN BEGIN</b>	(1st Part)
body	
<b>END ELSE BEGIN</b>	(2nd Part)
body	
<b>END;</b>	

where

IF, THEN, ELSE, BEGIN, END: are reserved words.

condition: any expression that can be answered with True or False.



body: set of functions, expressions or structures.

As commented before, the second part is optional. If it doesn't exist, a semicolon ';' has to be written just after the end of the first part:

```
IF (expr_logica) THEN BEGIN  
    cuerpo  
END;
```

Here is an example of using the IF structure:

```
CONST  
    max = 10;  
  
VAR  
    i : integer;  
  
BEGIN  
    i := 3;  
  
    IF (i > max) THEN BEGIN  
        Debug('The value ' + inttostr(i) + ' is bigger than ' + inttostr(max));  
    END ELSE BEGIN  
        Debug('The value ' + inttostr(i) + ' is lower than ' + inttostr(max));  
    END;  
END.
```

Result: **The value 3 is lower than 10**

## 5.2. CASE structure

The CASE structure has a mandatory first part and an optional second one. If the second part exists, it's only executed if the evaluated value doesn't comply with the conditions in the first part. CASE structure follows this format:

```
CASE expression OF                                (1st Part)  
    val1: BEGIN body END;  
    val2: BEGIN body END;  
    .  
    .  
    .  
    valN: BEGIN body END;  
ELSE                                (2nd Part)  
    BEGIN body END;  
END;
```

where

CASE, OF, ELSE, BEGIN, END: are reserved words.

expression: any function with a numeric, Boolean, character or string result

body: SC-PASCAL expressions, function and/or structures.



If the second part is not necessary, we have to finish the first part with the word **END** as shown:

```
CASE expression OF
  val1: BEGIN cuerpo END;
  val2: BEGIN cuerpo END;
  .
  .
  valN: BEGIN cuerpo END;
END;
```

This is an easy example of the CASE structure:

```
VAR
  c : char;

BEGIN
  c := 'M';

  CASE c OF
    'M': BEGIN debug('Value for c is M') END;
    'A': BEGIN debug('Value for c is A') END;
    'I' : BEGIN debug('Value for c is I') END;
    'N': BEGIN debug('Value for c is N') END;
  ELSE
    BEGIN
      Debug('Value ' + inttostr(i) + ' is lower than ' inttostr(max));
    END;
  END;
END.
```

Result: **Value for c is M**

## 6. Repetitive control structures

Repetitive control structures create a loop. The way to control this loop is defined by three SC-PASCAL structures as follows.

### 6.1. WHILE structure

The WHILE structure executes sentences inside its body while the condition is true. The format is:

```
WHILE condition DO
  BEGIN
    body
  END;
```

where



WHILE, DO, BEGIN, END: are reserved words.  
condition: any expression with a TRUE or FALSE answer.  
body: SC-PASCAL expressions, functions and/or structures.

If the condition is FALSE, instructions inside body are not executed and the program continues with instructions after the WHILE structure. If the condition is TRUE, instructions inside body will be executed until the condition changes to FALSE.

Note: if the condition is always TRUE, the loop will be infinite, and the rest of the program after WHILE structure will never be executed.

An easy example of this structure is:

```
CONST
  max = 5;

VAR
  i : integer;

BEGIN
  i := 1;

  WHILE i < max DO
    BEGIN
      Debug('The value for i is: ' + inttostr(i));
      i := i+1;
    END;
  END.
```

Result: **The value for i is: 1**  
**The value for i is: 2**  
**The value for i is: 3**  
**The value for i is: 4**

## 6.2. REPEAT structure

REPEAT is very similar to WHILE, but the condition is checked after executing the instructions inside the body. This means that the body part is executed at least one time. The format is:

```
REPEAT
  BEGIN
    body
  END;
UNTIL condition;
```

where

REPEAT, BEGIN, END, UNTIL: are reserved words.  
condition: any expression with a TRUE or FALSE answer.



body: SC-PASCAL expressions, functions and/or structures.

Note: if the condition is always TRUE, the loop will be infinite, and the rest of the program after REPEAT structure will never be executed.

This is an example that uses the REPEAT structure.

```
CONST
    max = 5;

VAR
    i: integer;

BEGIN
    i := 1;

    REPEAT
        BEGIN
            Debug('The value for i is: ' + inttostr(i));
            i := i+1;
        END;
    UNTIL i < max;
END.
```

Result: **The value for i is: 1**  
**The value for i is: 2**  
**The value for i is: 3**  
**The value for i is: 4**

### 6.3. FOR structure

FOR is used to run the instructions in the body section a determined number of times.

The format is:

```
FOR var := expression1 TO|DOWNTO expression2 DO
    BEGIN
        body
    END;
```

where

FOR, TO, DOWNTO, DO, BEGIN, END: are reserved words.

var: integer variable.

expression: any expression with a integer result.

body: SC-PASCAL expressions, functions and/or structures.

In this case var is set with the initial value (result of expression1). Var will increase (TO) or decrease (DOWNTO) until reaching the final value (given by expression2). For each step the instructions inside the body will be executed.



If  $\text{expression1} > \text{expression2}$  when you are using TO (or  $<$  when you use DOWNTO), the instructions inside body will never be executed.

An example of the FOR structure.

```
CONST
  max = 5;

VAR
  i : integer;

BEGIN
  FOR i:= 1 TO max DO
    BEGIN
      Debug('The value for i is: ' + inttostr(i));
    END;
  END.
```

Result: **The value for i is: 1**  
**The value for i is: 2**  
**The value for i is: 3**  
**The value for i is: 4**  
**The value for i is: 5**

## 6.4. When to use WHILE/REPEAT/FOR structures

- Use FOR when the number of loops is known and the variable that controls this number is an integer.
- Use REPEAT when the instructions in the body must be executed at least one time.
- Use WHILE for the rest of cases.

## 7. Procedures and functions

These procedures are not part of the standard PASCAL language. They have been designed for simulation projects.

### 7.1 Debug

This function shows a message (red marked in the picture below) in the debug section.

SC-PASCAL syntax:

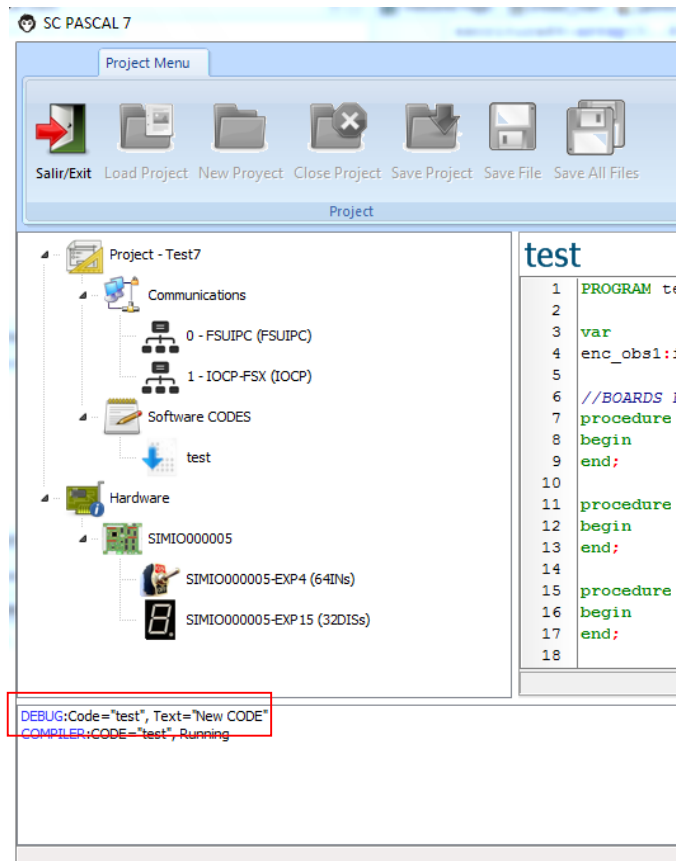
```
procedure Debug(Text: string);
```

Example:

```
Debug('New CODE');
```

Parameters:

Text: it's a string that will be shown.



## 7.2 Writelocp

Procedure to send a value through an IOCP connection previously established.

Parameters are connection, offset and value.

SC-PASCAL syntax:

*procedure* **Writelocp**(*id\_conn*,*offset*,*value*: integer);

Example:

**WriteIOCP**(1,43,180); //Send through connection no. 1 the value 180 for offset 43 (OBS NAV1).

Parameters:

**Id\_conn**: connection identifier. From 1 to 10.



**Offset:** Integer value for IOCP offsets. Check IOCP offsets list.

**Value:** Integer with the value.

### 7.3 RegIOCPOffset

This tells the IOCP server which offsets must to be sent. If the value is changed in the server, this value must be registered. To do so, we use this procedure setting the connection and offset we want to register.

SC-PASCAL syntax:

procedure **RegIOCPOffset**(id\_conn,offset: integer);

Example:

**RegIOCPOffset**(1,43); //We register offset no. 43 (OBS NAV1) in connection no. 1

Parameters:

**Id\_conn:** connection identifier (from 1 to 10)

**Offset:** Integer value for IOCP offset. Check IOCP offsets list.

### 7.4 InitOffsetFSUIPC

Like with IOCP, in the case of FSUIPC its offsets must be registered too. FSUIPC sends the value when it changes. In order to this registration, we must define the offset number and its size.

SC-PASCAL syntax:

procedure **InitOffsetFSUIPC**(offset,nbytes: integer);

Example:

**InitOffsetFSUIPC**(\$0C4E,2); //We register offset \$0C4E (OBS NAV1) with length 2 bytes.

Parameters:

**Offset:** Integer value for IOCP offset. Check IOCP offsets list.

**Nbytes:** Offset number of bytes. Check FSUIPC list of offsets.

### 7.5 WriteFSUIPC

This function lets us write a specific value of a FSUIPC offset. We must define offset number, its size and the value to be stored.

SC-PASCAL syntax:

procedure **WriteFSUIPC**(offset,nbytes: integer;value: variant);

Example:

**WriteFSUIPC**(\$0C4E,2,180); //We write the value 180 in offset \$0C4E (OBS NAV1) with length 2 bytes.

Parameters:

**Offset:** Integer value for IOCP offset. Check IOCP offsets list.

**Nbytes:** Number of bytes of offset (check FSUIPC list of offsets).

**Value:** Value to be stored in the offset.

## 7.6 ReadFSUIPC

We can read the value stored in a FSUIPC offset. Parameters for this function are the offset and its size. The function returns the value stored in the offset.

SC-PASCAL syntax:

function **ReadFSUIPC**(offset, nbytes: integer): variant;

Example:

**Obs\_nav1:=ReadFSUIPC**(\$0C4E,2); //We read offset \$0C4E (OBS NAV1) which is two bytes length. The value is assigned to our variable **Obs\_nav1**.

Parameters:

**Offset:** Integer value for IOCP offset. Check IOCP offsets list.

**Nbytes:** Number of bytes of offset (check FSUIPC list of offsets).

**Return:** The function returns the value stored in the offset.

## 7.7 RegEncoder

One of the most used components for our cockpits is the encoder. It consists on two inputs and some additional options, like recursive option, dual pulse per detent, etc. Encoders are defined by this function, and will generate an event each time the encoder changes its value.

SC-PASCAL syntax:



*function RegEncoder(sn:string; in1,in2,val\_min,val\_max:integer; recursive:byte; val\_init:integer; half:Boolean; vel:single): integer;*

Example:

*Id\_enc\_obs\_nav1:= **RegEncoder**('SIMIO000005',17,18,0,360,1,0,false,100);*

Parameters:

**sn:** Serial Number of the card where the encoder is connected

**In1:** input no. 1 where the encoder is connected.

**In2:** input no. 2 where the encoder is connected.

**Val\_min:** Minimum value that the encoder can have.

**Val\_max:** Maximum value that the encoders can have.

**Recursive:** when the encoder reaches its maximum value and we continue turning in the same direction, if this option is set to '1' the encoder value changes to the minimum value. If it's set to '0', the encoder value doesn't change.

**Val\_init:** we can set an initial value when the system is started. This value must be between val\_min and val\_max.

**Half:** Normally, encoders makes one state change with each detent. On the other hand, there are encoders with more than one change per detent. If we have encoders with two changes per detent, set to 'true' this parameter.

**Vel:** normally, when we turn the encoder quickly, we have to add 10 units for each detent. This value lets set the time (in milliseconds) between changes that makes add 10 units instead 1. This value depends on our computer so we have to set it in a test-and-decide process.

**Return:** The result is the identifier value of the encoder.

## 7.8 WriteDisplays

With this function we send the value to be shown by a group of 7-segments displays. We have to define the serial number of the displays card, the initial figure, the value to be shown, the filling character, the number of figures and if we are using the decimal point or not.

SC-PASCAL syntax:

*procedure **WriteDisplays**(sn:string,dig\_init:byte;value,extra\_char:string;ndigits,dcon:byte)*

Example:

***WriteDisplays**('SIMIO000005-EXT15',11,'360','0',3,0);*



Parameters:

**sn:** serial number of displays card.

**Dig\_init:** initial figure. Number of the initial display in a group of displays.

**Value:** value to be shown by the group of displays.

**Extra\_char:** is the filling character in the case that the value to be shown is smaller than the number of figures in the group. For example, if we have to show '36' for course, we have to set the extra char as '0' so the shown course is '036'.

**Ndigits:** number of figures. 3 for course for example.

**Dcon:** we can enable (1) or not (0) the use of decimal point.

## 7.9 SetDisplayIntensity

Each displays card consists on four groups of 8 displays each. We can set the light intensity of each group separately.

SC-PASCAL syntax:

procedure **SetDisplayIntensity**(sn:string;block,value:byte)

Example:

**SetDisplayIntensity**('SIMIO000005-EXT15',1,15);

Parameters:

**sn:** Serial Number of displays card.

**Block:** Block number (1 to 4).

**Value:** intensity value (from 1 to 15).

## 7.10 WriteOut

With this function we can set on and off any output in the system. Parameters are the serial number of the card, the number of output and if it has to be set to on (1) or off (0).

SC-PASCAL syntax:

procedure **WriteOut**(sn:string;noutput,value:byte)



Example:

**WriteOut** ('SIMIO000005',15,1);

Parameters:

**sn:** Serial Number of outputs card.

**Noutput:** number of the output to be set. 1 to 16 for Main USB card; 1 to 32 for 32in32out cards; 1 to 64 for 64 outputs cards.

**Value:** 0 or 1 to turn off and on the output.

### 7.11 ReadIOCP

With this function we can request the value of any IOCP offset without waiting for it event. We have to define the identifier of the IOCP connection and the offset we want to read.

SC-PASCAL syntax:

function **ReadIOCP**(id\_conn:byte;offset:integer)

Example:

Offset\_val:=**ReadIOCP**(1,43);

Parameters:

**Id\_conn:** IOCP connection identifier, from 1 to 10.

**Offset:** Integer value of the IOCP offset. See IOCP offset list.

**Return:** the function returns the value of the IOCP offset.

### 7.12 TestBit

We can check the value of a determined bit. Many times, we have to do a certain operation depending on the value of a determined bit. We have to define the numeric variable, the bit position and the size of the variable. We will get 0 or 1 depending on the bit value.

SC-PASCAL syntax:

Function **TestBit**(number:variant;position,length:byte)



Example:

```
Val_test:=TestBit(2,2,8);
```

Parameters:

**number:** Variable or numeric value where to check the bit.

**position:** Bit position.

**Length:** Size of the number.

**Return:** the function returns 0 or 1.

### 7.13 SetBit

On the other hand, this function writes a specific value 0 or 1 on a specific bit.

SC-PASCAL syntax:

Function **SetBit**(number:variant;position,value,length:byte)

Example:

```
Int_val:=2;
```

```
Int_val:=SetBit(Int_val,2,1,8);
```

Parameters:

**number:** Variable or numeric value where to write the bit.

**position:** Bit position.

**Value:** Value to be written, 0 or 1.

**Length:** Size of the number.

**Return:** the function returns the number value with the new bit value.

### 7.14 ChangeBit

If we only want to change the value of a bit (from 0 to 1 or from 1 to 0) we can use this function. We have to define the numeric value, the bit position and the size of the value.

SC-PASCAL syntax:



Function **ChangeBit**(number:variant;position,length:byte)

Example:

```
Int_val:=2;
```

```
Int_val:= ChangeBit (Int_val,2,8);
```

Parameters:

**number:** Variable or numeric value where to change the bit.

**position:** Bit position.

**Length:** Size of the number.

**Return:** the function returns the value of the number with the changed bit.

## 7.15 WriteIOCPKey

Procedure to send an IOCP offsetsKey value through an active IOCP connection. We have to define the connection id, the offset and its value.

SC-PASCAL syntax:

```
procedure WriteIocpKey(id_conn,offsetKey,value: integer);
```

Example:

```
WriteIOCPKey(1,43,180); //Send through connection 1 the value 180 to offset 43.
```

Parameters:

**Id\_conn:** Connection id, from 1 to 10.

**OffsetKey:** Integer value for IOCP offsetsKeys. See list of IOCP offsetsKeys.

**Value:** Integer with the value to be sent.

## 7.16. CreateTimer

This procedure lets us to create a “timer”. We have to define an unique identifier for it and the interval (in milliseconds). When the timer is enabled, it will start counting the defined number of milliseconds. When it finishes, the event defined on “OnTimer<n>” (see 8.13) will be started.

SC-PASCAL syntax:

```
procedure CreateTimer(idTimer:cardinal;intervalMs:cardinal)
```

Example:

```
CreateTimer(1,1000);
```

Parameters:

**idTimer**: variable or numeric value that identifies the timer. Must be unique.

**intervalMs**: Value in milliseconds.

### 7.17. SetTimer

This function enables or disables the timer created with the previous procedure. We have to define the timer ID and the value that defines its state (true or false).

SC-PASCAL syntax:

```
procedure SetTimer(idTimer:cardinal;status:boolean)
```

Example:

```
CreateTimer(1,1000);
```

```
SetTimer(1,true); //in this case, after creating the timer "1", it is enabled.
```

Parameters:

**idTimer**: Variable or numeric value that defines the timer. Must be unique.

**status**: value that defines the timer state: "true" or "false".

### 7.18. GetTimer

This function gives us the state of a timer, to know if it is enabled or not.

SC-PASCAL syntax:

```
Function GetTimer(idTimer:cardinal):Boolean;
```

Example:

```
CreateTimer(1,1000);
```

```
Timer_status:=GetTimer(1); //in this example, after creating timer "1", we get its state.
```



Parameters:

**idTimer:** Variable or numeric value that defines the timer. Must be unique.

**Return:** The result of this function gives us the boolean value of the timer state, 1 for enabled, 0 for disabled.

### 7.19. GetEncoder

This function will give us the calculated value of a previously registered encoder. Normally, we get the value from an encoder during its event, but if we need to know this value at any other moment, we can use this function.

SC-PASCAL syntax:

Function **GetEncoder**(idEncoder:integer):variant;

Example:

```
idEnc:= RegEncoder('SIMIO000005',17,18,0,360,1,0,false,100);
```

```
valEncoder:=GetEncoder(idEnc);// idEnc is the identifier for the registered encoder
```

Parameters:

**idEncoder:** Variable or numeric value that indicates the encoder ID. Must be unique.

**Return:** This function returns the integer value of the encoder.

### 7.20. SetEncValue

An encoder has a calculated value set during its event. We can change the value of an encoder at any time using this function. If we have the heading encoder with 180, we can initialize it to 0 using this function.

SC-PASCAL syntax:

```
procedure SetEncValue(idEncoder:integer;newValue:integer);
```

Example:

```
idEnc:= RegEncoder('SIMIO000005',17,18,0,360,1,0,false,100);
```

```
valEncoder:=GetEncoder(idEnc);// idEnc is the registered encoder ID.
```

```
SetEncValue(idEnc,100);// Initializes encoder value to 100
```

Parameters:

**idEncoder:** Variable or numeric value that indicates the encoder ID. Must be unique.

**newValue:** New integer value for encoder.

### 7.21. WriteDisplaysSwap

This function Works like the original WriteDisplays, but it handle the figures in a reversed way. This is useful when we have installed the displays in the wrong order.

SC-PASCAL syntax:

```
procedure WriteDisplaysSwap(sn:string,dig_init:byte;value,extra_char:string;ndigits,dcon:byte)
```

Example:

```
WriteDisplaysSwap('SIMIO000005-EXT15',11,'360','0',3,0);
```

Parameters:

**sn:** serial number of displays card.

**Dig\_init:** initial figure. Number of the initial display in a group of displays.

**Value:** value to be shown by the group of displays.

**Extra\_char:** is the filling character in the case that the value to be shown is smaller than the number of figures in the group. For example, if we have to show '36' for course, we have to set the extra char as '0' so the shown course is '036'.

**Ndigits:** number of figures. 3 for course for example.

**Dcon:** we can enable (1) or not (0) the use of decimal point.

### 7.22. RegIFLYOffset

If we are using this add-on for FSX, we can read and write its variables. We need an active connection with iFly first. Then we have to register the offset we are interested in. When this offset changes, the event OnIFLYChange (see 8.15) will inform us.

SC-PASCAL syntax:

```
procedure RegIFLYOffset(offset: integer);
```

Example:

```
RegIFLYOffset (43); //we register offset 43;
```



Parameters:

**Offset:** Integer value for iFly offset. See iFly list of offsets.

### 7.23. WriteIFLY

With this function we can write the value of an iFly offset. Is not necessary to register it.

SC-PASCAL syntax:

```
procedure WriteIFLY(offset,value: integer);
```

Example:

```
WriteIFLY(43,180); //Send the value 180 to offset 43.
```

Parameters:

**Offset:** Integer value for iFly offset. See iFly list of offsets.

**Value:** integer with the value to be written.

### 7.24. ReadIFLY

Like with IOCP offsets, if we want to read the value of a previously registered iFly offset but not using the corresponding event, we can use this function.

SC-PASCAL syntax:

```
function ReadIFLY(offset:integer):integer;
```

Example:

```
Offset_val:=ReadIFLY(43);
```

Parameters:

**Offset:** Integer value for iFly offset. See iFly list of offsets.

**Return:** the function returns the value for the iFly offset.

### 7.25. EnableADCChannel

If we want to use the ADC channels on USB Main, USB Joystick and USB 8ADC/8Servos cards, we have to enable it in advance. The activation must be done once only.

SC-PASCAL syntax:

```
procedure EnableADCChannel(boardSN:string;naxis:byte;valOnOff:byte;roundVal:integer);
```

Example:

**EnableADCChannel**('SIMIO000001',1,1,10); //activates channel 1 on USB Main card, and rounds the result into tenths.

Parameters:

**BoardSN:** alphanumeric value for card serial number.

**naxis:** number of ADC channel to enable or disable.

**valOnOff:** 0 or 1 to enable or disable.

**roundVal:** round value to mute noise on the potentiometers values.

## 7.26. WriteIocpServer

SC-PASCAL lets us to create an IOCP connection as a server. If we need to write an offset value in the server, we will use this function.

SC-PASCAL syntax:

```
procedure WriteIocpServer(offSet:integer;value:integer);
```

Example:

**WriteIocpServer**(45,10); //Write the value 10 for IOCP server number 45.

Parameters:

**Offset:** Integer value for IOCP offset id. See list of IOCP offsets.

**value:** Integer value for IOCP offset.

## 7.27. ReadIniFile

Configuration INI text files that can be read by SC-PASCAL must have the following structure:

```
[<section1>]
```

```
<key1>=<value1>
```

```
<key2>=<value2>
```



```
.  
.  
[<section2>]  
  
<key1>=<value1>  
  
<key2>=<value2>  
  
.  
.
```

In order to read these INI files, we use the function `ReadIniFile`.

SC-PASCAL syntax:

```
function ReadIniFile(filename:string;section:string;key:string):variant;
```

Example:

```
Offset_HDG:= strtoint(ReadIniFile ('config.ini','offsets','hdg'));
```

```
"Config.ini
```

```
[offsets]
```

```
Hdg=185"
```

Parameters:

**Filename:** File name with the complete route for INI file.

**Section:** Section inside INI file where we want to read a key value.

**Key:** Key that we want to read.

## 7.28. WriteIniFile

With this function we can write in an INI file.

SC-PASCAL syntax:

```
procedure WriteIniFile(filename:string;section:string;key:string;value:string);
```

Example:

```
WriteIniFile ('config.ini','offsets','hdg','186');
```



Parameters:

**Filename:** File name with the complete route for INI file.

**Section:** Section inside INI file where we want to write a key value.

**Key:** Key that we want to write.

**Value:** Value that we want to write.

## 7.29. SendKey

Sometimes we have to send keystrokes to an application, for example to Flight Simulator. These are the rules to follow in order to send a keystroke to an application.

“SendKeys supports the Visual Basic SendKeys syntax, as documented below.

Supported modifiers:

+ = Shift

^ = Control

% = Alt

Surround sequences of characters or key names with parentheses in order to modify them as a group. For example, '+abc' shifts only 'a', while '+(abc)' shifts all three characters.

Supported special characters

~ = Enter

( = Begin modifier group (see above)

) = End modifier group (see above)

{ = Begin key name text (see below)

} = End key name text (see below)

Supported characters:

Any character that can be typed is supported. Surround the modifier keys listed above with braces in order to send as normal text.



Supported key names (surround these with braces):

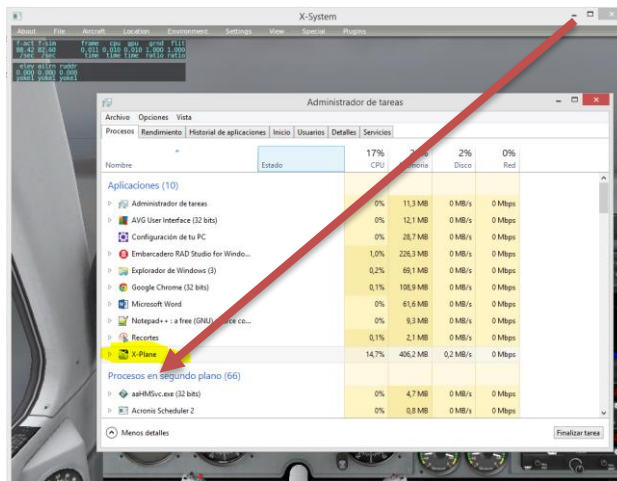
BKSP, BS, BACKSPACE, BREAK, CAPSLOCK, CLEAR, DEL, DELETE

DOWN, END, ENTER, ESC, ESCAPE, F1, F2, .. F16

HELP, HOME, INS, LEFT, NUMLOCK, PGDN, PGUP

PRTSC, RIGHT, SCROLLLOCK, TAB, UP"

We need to know the name of the window where the application is running. This info can be found in the tasks administrator, in this example the name is "X-Plane".



SC-PASCAL syntax:

```
procedure SendKey(windowsName:string;keys:string);
```

Example:

```
SendKey ('X-Plane','F1'); //We send keystroke F1 to X-Plane window.
```

Parameters:

**WindowsName:** Name of the window.

**Keys:** string that defines the keystroke to be sent.

### 7.30. ReadAdc

Cards with analogic-digital converters (ADC) send the result of the conversion using an event. But we can read the value from the ADC using this function too.

SC-PASCAL syntax:

Function **ReadAdc**(boardSN:string;naxis:byte):integer;

Example:

Value\_pot1:=**ReadAdc** ('SIMIO000001',1); //we read the value of ADC number 1 from Main card 'SIMIO000001'.

Parameters:

**boardSN**: alphanumeric value with the serial number of the card.

**naxis**: number of channel that we want to read.

### 7.31. EnableServo

If we want to use a servo motor, we have to enable it first by using this function. We can disable servo motors too.

SC-PASCAL syntax:

procedure **EnableServo**(boardSN:string;nservo:byte;valOnOff:byte;iniValue:integer);

Example:

**EnableServo** ('SIMIO300001',1,1,127); //we enable servo number 1 on card SIMIO300001, setting the initial value to 127.

Parameters:

**boardSN**: alphanumeric value for card serial number.

**nservo**: number of servo motor to be enabled (or disabled).

**valOnOff**: we send 0 to disable or 1 to enable the servo motor.

**iniValue**: initial value for servo motor. From 0 to 255.

### 7.32. WriteServo

With this function we send a specific value to one servo motor.

SC-PASCAL syntax:

procedure **WriteServo**(boardSN:string;nservo:byte;value:integer);



Example:

**WriteServo** ('SIMIO300001',1,137); //we set the value 137 for servo number 1 on card SIMIO300001.

Parameters:

**boardSN:** alphanumeric value for card serial number.

**nservo:** servo motor number.

**Value:** value for servo motor. From 0 to 255.



## 8. Events

Events make possible not to be continuously asking for the value of an offset, state of a connection, etc. The events are run when a change on the associated offsets, connection... happens.

### 8.1 OnInputChange

This event is launched when an input changes its state. The event informs us about the card serial number, the input number and the value (0 or 1).

SC-PASCAL syntax:

```
procedure OnInputChange(sn:string;input,value:byte);  
  
begin  
  
    {Your Code here}  
  
end;
```

Parameters:

**sn:** Serial Number of the card where the input has changed.

**Input:** the number of input that has changed.

**Value:** value of the input (0 or 1).

### 8.2 OnInputChangeAll

Like in the previous event, this one gives a string with every input in binary code. The size of the string will depend on the card type.

SC-PASCAL syntax:

```
procedure OnInputChangeAll(sn,inputs:string);  
  
begin  
  
    {Your Code here}  
  
end;
```

Parameters:

**sn:** Serial Number of the card where the inputs have changed.

**Inputs:** binary string with all the values for the card inputs.



### 8.3 OnADCChange

This event works with cards equipped with analogue to digital converters (ADCs). When any ADC changes on Main USB card, servo motors card or Joystick card, this event send the number of card, the channel and the value.

SC-PASCAL syntax:

```
procedure OnADCChange(sn:string;nadc:byte;value:integer);  
  
begin  
  
    {Your Code here}  
  
end;
```

Parameters:

**sn:** Serial Number of the card where the ADC has changed.

**nadc:** number of the channel (1 to 5 for Main USB; 1 to 8 for USB 8Servo/8ADCs, 1 to 8 for USB Joystick).

**Value:** value for ADC.

### 8.4 OnEncoderChange

After registering an encoder (see point 7.7), when the value of an encoder changes, this event sends the value of this encoder, giving the encoder id and the value.

SC-PASCAL syntax:

```
procedure OnEncoderChange(id_encoder,value:integer);  
  
begin  
  
    {Your Code here}  
  
end;
```

Parameters:

**Id\_encoder:** encoder identifier (see point 7.7)

**Value:** value of the encoder that has changed.



## 8.5 OnIOCPConnect

When IOCP connects to a valid server, the server generates an event to check that the connection is enabled, giving us the socket or connection identification. We have sockets from 1 to 10.

SC-PASCAL syntax:

```
procedure OnIOCPConnect(socket:byte);  
  
begin  
  
    {Your Code here}  
  
end;
```

Parameters:

**Socket:** IOCP connection identifier.

## 8.6 OnIOCPConnectClosed

This event is launched when an IOCP socket or connection is finished.

SC-PASCAL syntax:

```
procedure OnIOCPConnectClosed(socket:byte);  
  
begin  
  
    {Your Code here}  
  
end;
```

Parameters:

**Socket:** IOCP connection identifier

## 8.7 OnIOCPConnectFail

This event is launched when there is any error in the IOCP connection. Then, in the Debug window the specific error is shown. The event gives us the connection identifier.

SC-PASCAL syntax:

```
procedure OnIOCPConnectFail(socket:byte);  
  
begin
```



*{Your Code here}*

*end;*

Parameters:

**Socket:** *IOCP connection identifier*

## 8.8 OnIOCPChange

This event is launched when the value of an registered IOCP offset has changed. See point 7.3 to see how to register an IOCP offset.

SC-PASCAL syntax:

```
procedure OnIOCPChange(socket,offset:byte;value:integer);
```

```
begin
```

```
    {Your Code here}
```

```
end;
```

Parameters:

**Socket:** *IOCP connection identifier.*

**Offset:** *IOCP offset identifier.*

**Value:** *Value of the registered offset that has changed in IOCP server.*

## 8.9 OnFSUIPCError

When an error happens in a FSUIPC connection, this event is launched, giving us a message with information about the error.

SC-PASCAL syntax:

```
procedure OnFSUIPCError(msg:string);
```

```
begin
```

```
    {Your Code here}
```

```
end;
```

Parameters:

**msg:** *Message with the error in the FSUIPC connection.*



### 8.10 OnFSUIPCConnect

This event is launched when a correct connection with FSUIPC has happened, giving us a message with the answer info from the FSUIPC server.

SC-PASCAL syntax:

```
procedure OnFSUIPCConnect(msg:string);  
  
begin  
  
    {Your Code here}  
  
end;
```

Parameters:

**msg:** Message with the answer from FSUIPC server.

### 8.11 OnFSUIPCDisconnect

This event is launched when FSUIPC connection finishes, giving us a message with the reason for disconnection.

SC-PASCAL syntax:

```
procedure OnFSUIPCDisconnect(msg:string);  
  
begin  
  
    {Your Code here}  
  
end;
```

Parameters:

**msg:** Message with the reason for FSUIPC disconnection.

### 8.12 OnFSUIPCChange

This event is launched when there a change in any registered FSUIPC offset, giving us the offset id and the new value. To register an offset, see point 7.4.

SC-PASCAL syntax:

```
procedure OnFSUIPCChange(offset:integer;value:variant);
```



*begin*

*{Your Code here}*

*end;*

Parameters:

**Offset:** Integer value for FSUIPC offset (Check FSUIPC list of offsets).

**Value:** Value of the FSUIPC offset that has changed.

### 8.13. OnTimern

This event is launched when a defined “timer” has ended. It doesn’t return any value. “n” means the timer number identification. You can create a timer using the function “CreateTimer” (see 7.16).

SC-PASCAL syntax:

*procedure OnTimer1();*

*begin*

*{Your Code here}*

*end;*

Parameters:    **NONE**

### 8.14. OnIFLYConnect

This event is launched when SC-PASCAL has connected correctly with IFLY correctamente.

SC-PASCAL syntax:

*procedure OnIFLYConnect();*

*begin*

*{Your Code here}*

*end;*

Parameters:    **NONE**



### 8.15. OnIFLYChange

When we register an IFLY offset, if this offset changes its value, SC-PASCAL will launch this event.

SC-PASCAL syntax:

```
procedure OnIFLYChange (offset:integer;value:variant);
```

```
begin
```

```
    {Your Code here}
```

```
end;
```

Parameters:

**Offset:** integer value for IFLY offset. See list of IFLY offsets.

**Value:** value of the IFLY offset that has changed.

### 8.16. OnIFLYDisconnect

This event is launched when the IFLY connection closes.

SC-PASCAL syntax:

```
procedure OnIFLYDisconnect();
```

```
begin
```

```
    {Your Code here}
```

```
end;
```

Parameters:    **NONE**

### 8.17. OnIOCPServerChange

When an IOCP connection is configured as a server, if any IOCP server offset changes then SC-PASCAL will launch this event.

SC-PASCAL syntax:

```
procedure OnIOCPServerChange (offset:integer;value:integer);
```

```
begin
```



*{Your Code here}*

*end;*

*Parameters:*

***Offset:*** integer value for IOCP offset. See list of IOCP offsets.

***Value:*** value of the offset that has changed in the IOCP server.



## 9. User Interface (UI)

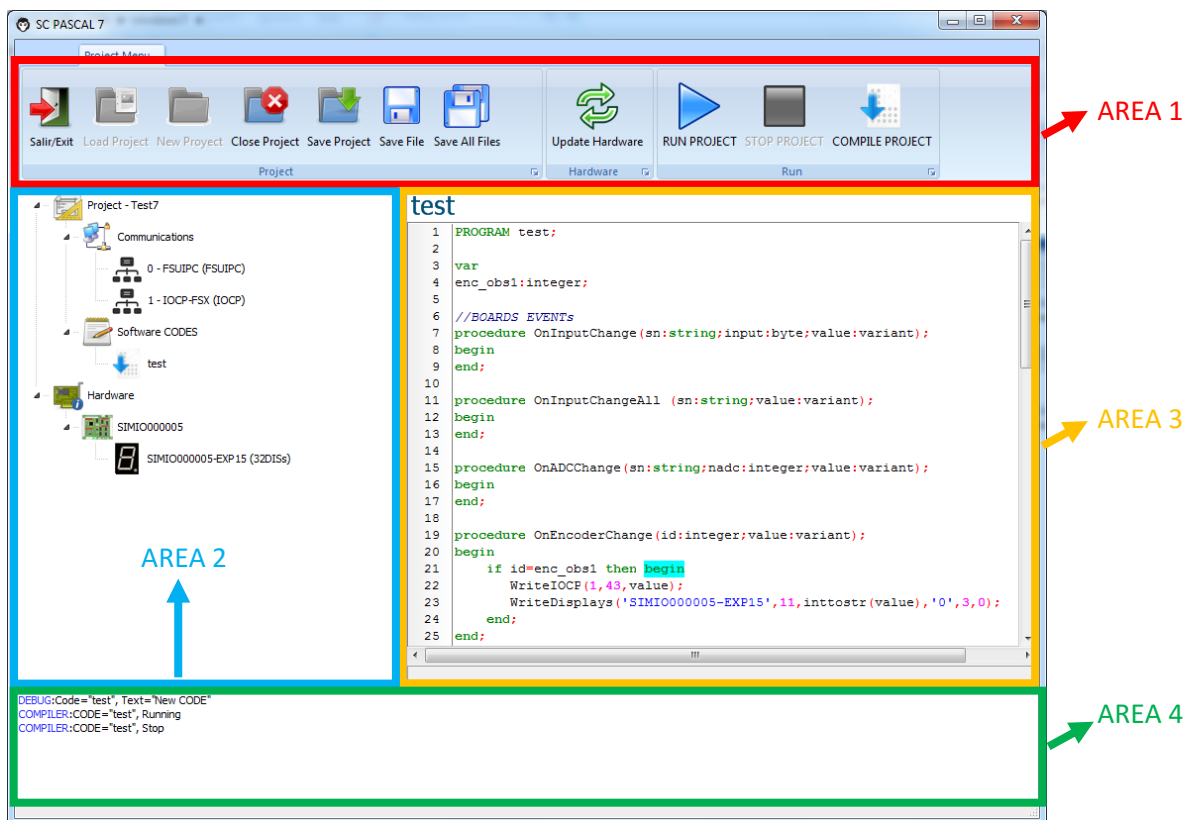
The UI of SC-PASCAL 7E1 IDE de SC-PACAL7 is divided into four áreas.

Area 1: Menu bar: with different buttons for different functions.

Area 2: Objects of software and hardware. Every objects available for development and working of our simulation system.

Area 3: Working window. Shows different functionalities, depending on the selected object under objects área.

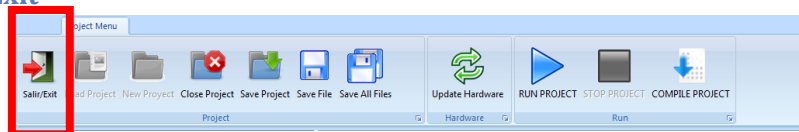
Area 4: window that shows errors, information and messages from the debugger (see point 7.1).



### 9.1 Menu bar

The menú bar has the following elements:

#### 9.1.1 Exit



To exit from the program.



### 9.1.2 Load Project



We can load an existing Project. It's disabled while another project is opened.

### 9.1.3 New Project



To create a new project.

### 9.1.4 Close Project



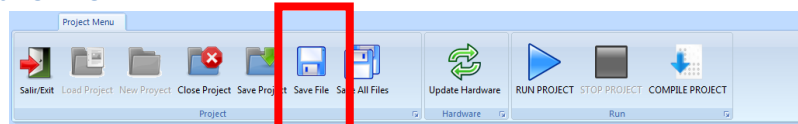
Once a project is opened, we have to close it if we want to open a different one.

### 9.1.5 Save Project



To save the opened project.

### 9.1.6 Save File



To save the modification made into the code in the working area. Only saves this area, not the rest of the project.

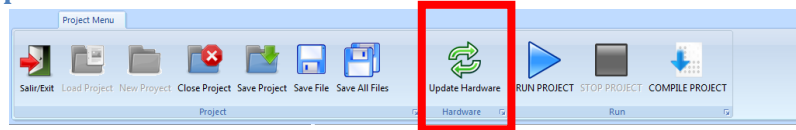
### 9.1.7 Save All Files



Like the previous one, but saves every code file in the project.

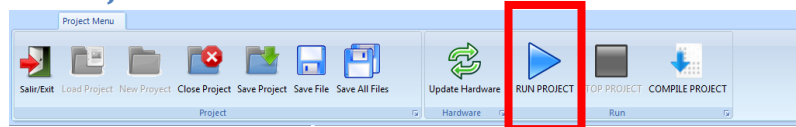


### 9.1.8 Update Hardware



When any change in our hardware happens, we can force the update with this button. Normally, the update is automatic, but if we change one extension we have to click in the button to refresh the info. This action doesn't affect the execution time.

### 9.1.9 RUN PROJECT



We run our project by clicking on this button. It starts the connections with IOCP and/or FSUIPC and executes the code inside the project.

### 9.1.10 STOP PROJECT



On the opposite, this buttons stop the project and finishes the connections.

### 9.1.11 COMPILE PROJECT

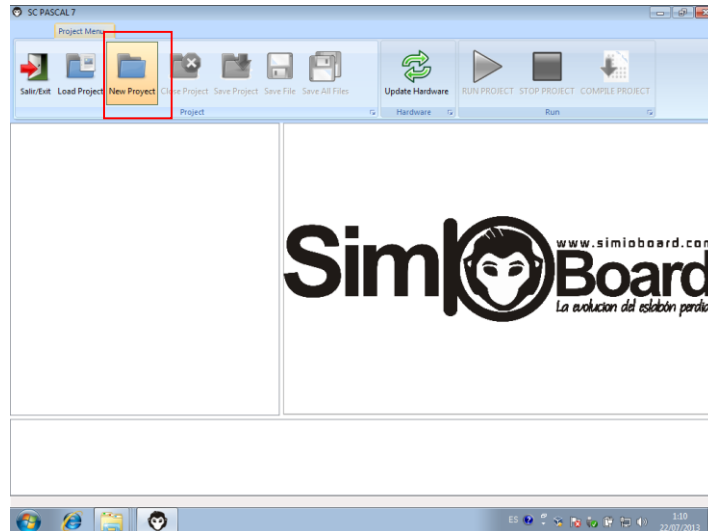


When we are developing a Project, we can check if there is any mistake in the code. In the debug window, the error and location will be shown. This action doesn't make any connection to IOCP or FSUIPC.

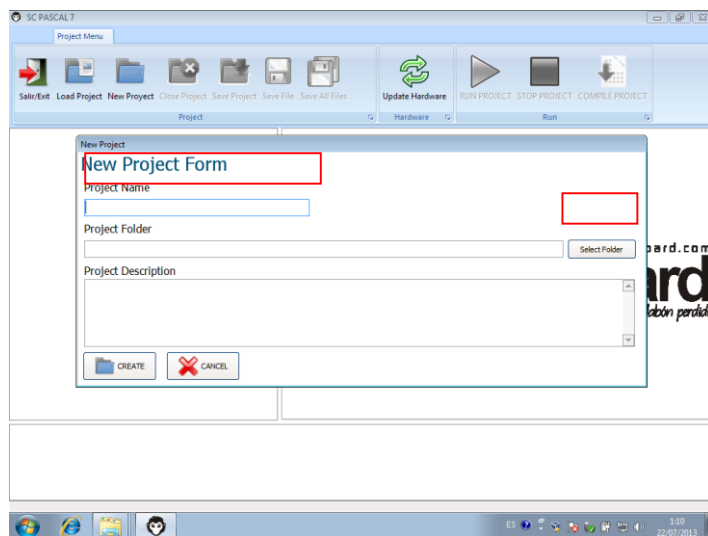


## 9.2 Projects

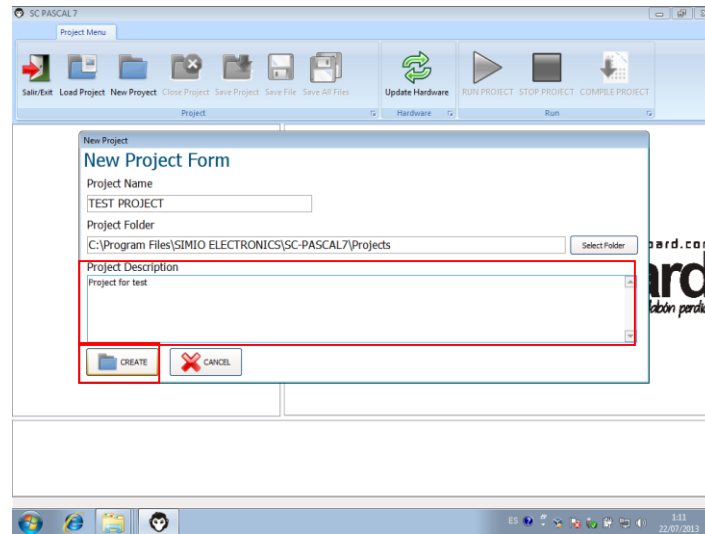
SC-PASCAL7 works with projects. Each project defines the required connections and code. Once SC-PASCAL7 is started, we click on “New Project”.



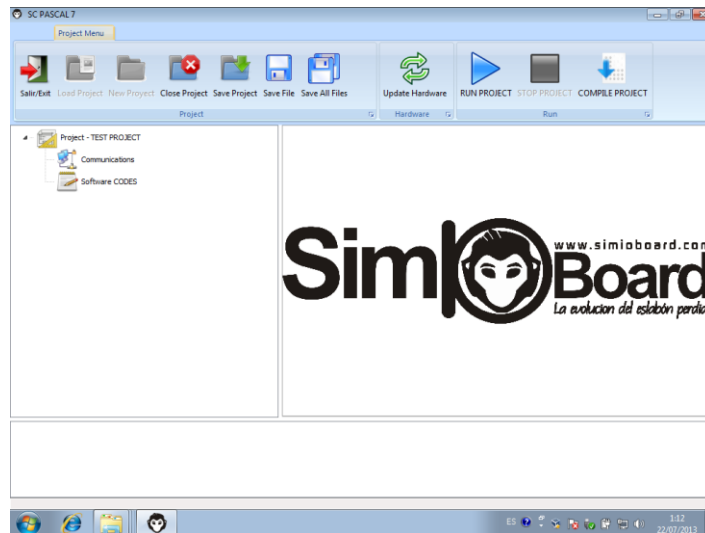
We write the name of the project and the desired folder by clicking on “Select folder”.



We can also include a text with a brief description of the project. Then we click on “CREATE”.



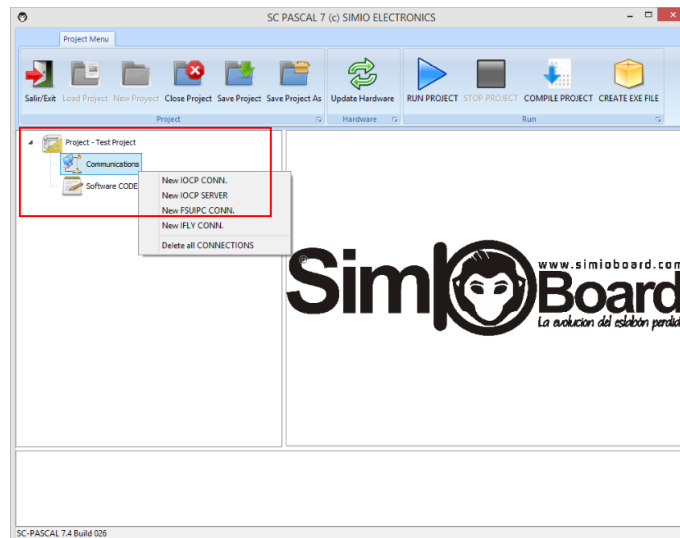
Once the project is created, we are ready to create the connections and code.





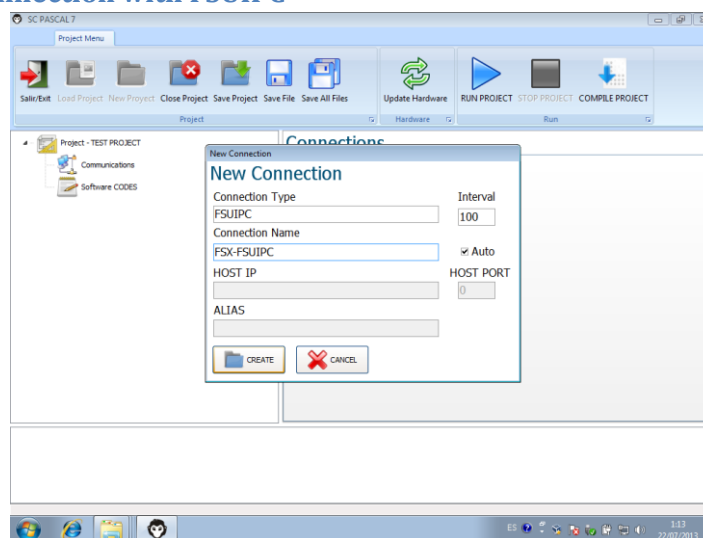
## 9.3 Connections

In the objects area we can configure a new connection with FSUIPC and 10 with IOCP. To create a new connection, we right-click and we will get this menu



We will click on “New IOCP CONN.” to create a new IOCP connection, or “New FSUIPC CONN.” to create a FSUIPC one. A new window for connection configuration will pop-up.

### 9.3.1 Connection with FSUIPC



In this window we set the data for the FSUIPC connection:

“Connection Type”: Fixed value, can’t been changed

“Connection Name”: We can set a name for this connection.

“Interval”: FSUIPC is a pool type connection, so we have to ask for the data continuously, even if the data hasn’t changed. We have to set a time interval that depends on

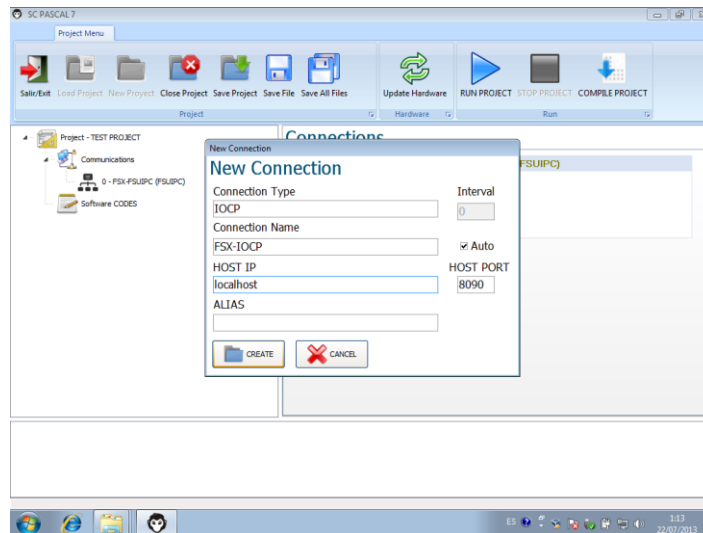


each computer. The default value is 100 milliseconds. For powerful computers this value can be reduced.

“Auto”: If the flag is checked, if there’s a connection error, the system will try to connect again.

By clicking on “CREATE” the connection with FSUIPC server is created.

### 9.3.2 Connection with IOCP



IOCP connection is different because we can connect with different IOCP servers. For each connection we have to know the IP address (or server alias) and the associated port. In the example, we have configured a connection with a server in our local machine, so no need to set an IP. Only the DNS of our machine (localhost).

“Connection Type”: Fixed value; can’t be modified.

“Connection Name”: We can set a name for this connection.

“Interval”: it’s not needed, because the IOCP protocol works with events when something changes.

“Auto”: If the flag is checked, if there’s a connection error, the system will try to connect again.

“HOST IP”: we have to define the IP or DNS where the IOCP server is located. If it’s the own machine, we will write 0localhost’. Si es el mismo PC donde esta SC-PASCAL7 podemos usar “localhost”.

“HOST PORT”: listening port of IOCP server. Normally 8090.

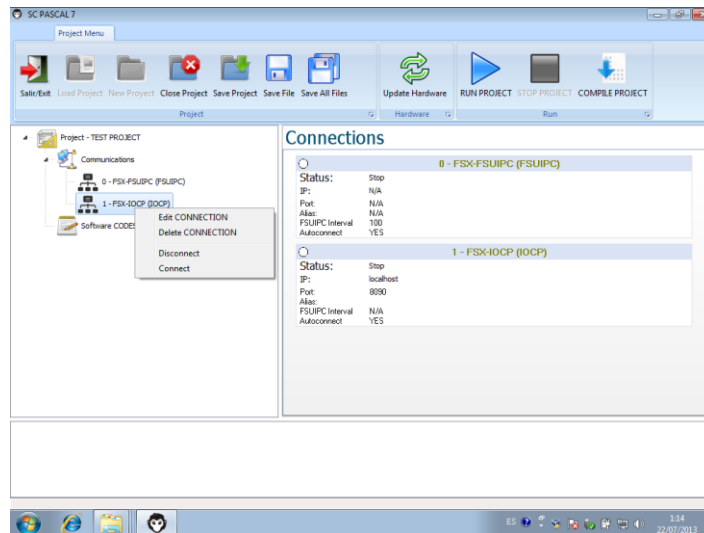
“ALIAS”: IOCP lets make connections through a protocol automatic finding system. We have only to give the server name and the IOCP connection is automatically configured. (Not yet available in this versión).



By clicking on “CREATE” the connection with IOCP server is created.

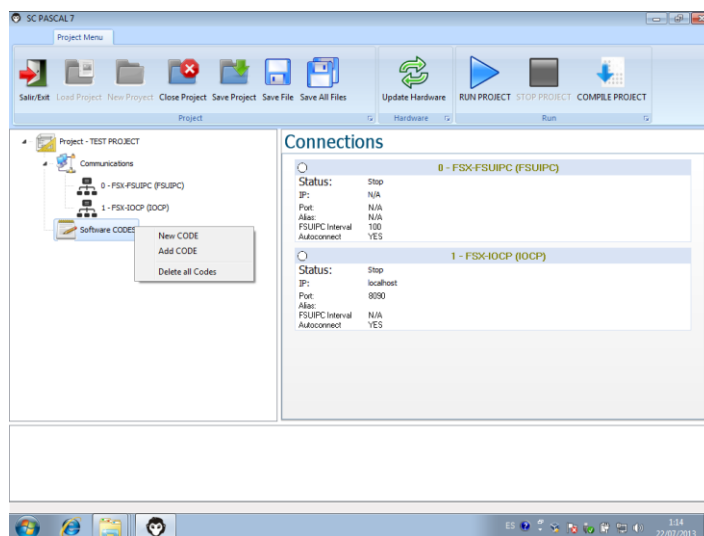
### 9.3.3 Handling connections

We can handle every available connection with SC-PASCAL7. We can delete, edit, connect or disconnect them. When we click on the “Communications” icon (objects window), a list of connection with their properties and state will be shown on the working area.



## 9.4 Codes

Codes contain the programs that link our hardware with the flight simulation software. We can have a big number of codes running at the same time. We have the option to create new files for codes or to load existing files. If we right-click on the label “Software CODES” (objects window), a menu will pop-up:

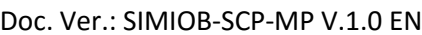




With this option we add a new code to the project. We have to define a name for it and the base code and every available events will be created for this file:



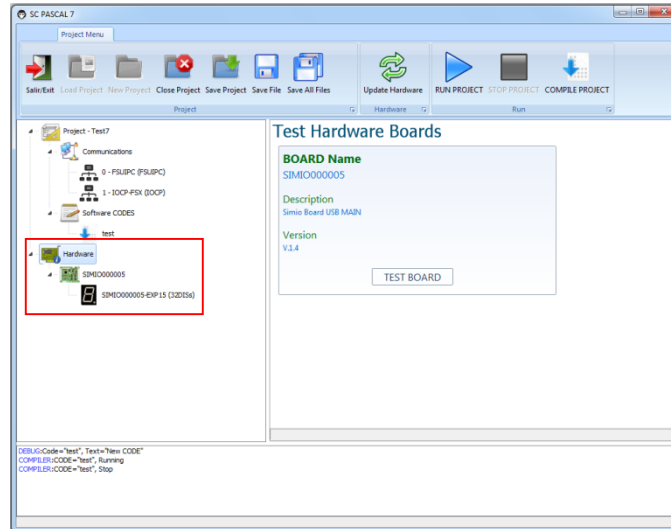
We can add an existing file to our project. This file can have been designed by another user.



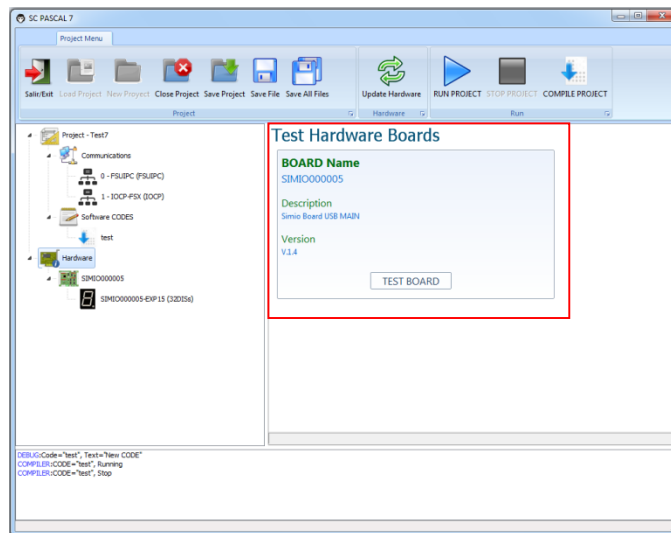


## 9.5 Hardware

When a SimIO card is connected to the computer running SC-PASCAL7, every card (Main USB and extensions cards) are shown in the objects area.



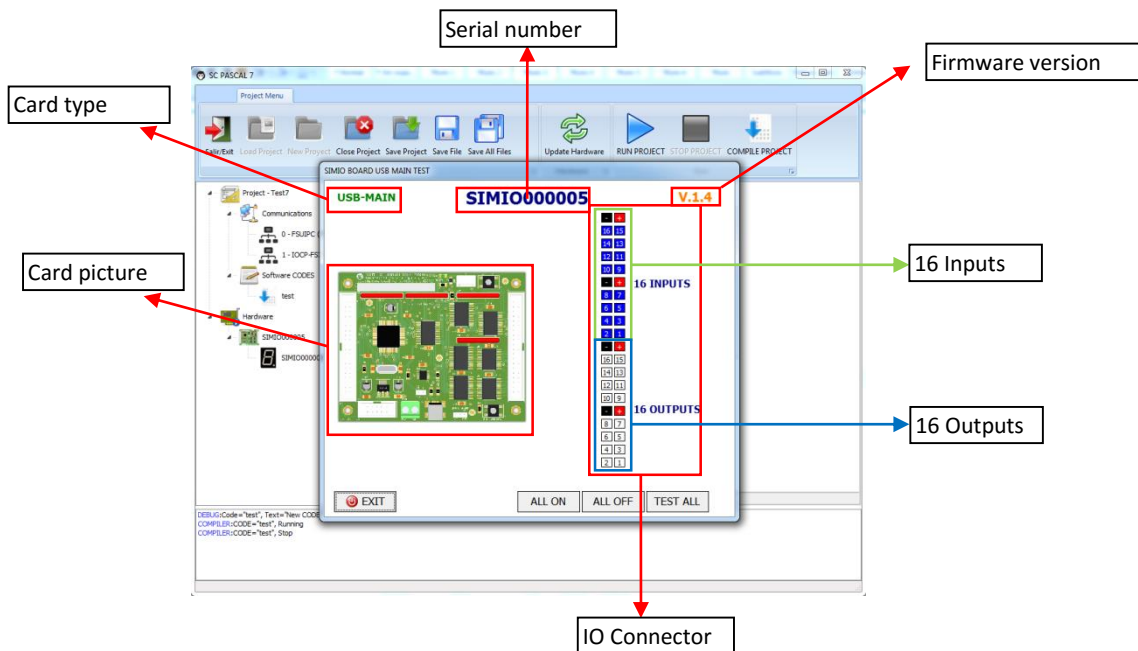
When we click on a card label, the properties are shown in the working area. By clicking on the “TEST BOARD” button, we can check if the card Works properly.





### 9.5.1 Test Main

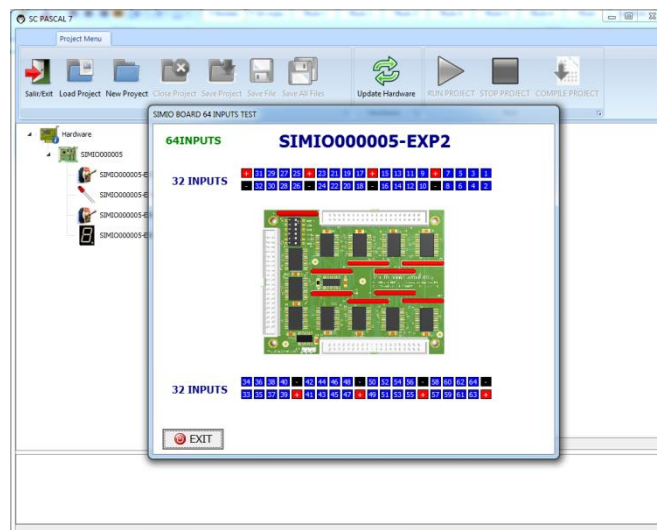
We can test and check inputs and outputs of Main USB card.



When any input changes, the box corresponding to that input will change its color. If we want to change the value of an output, we can click on the corresponding box. We can change all the outputs at the same time with the buttons “ALL ON”, “ALL OFF” or “TEST ALL”.

### 9.5.2 Test EXP 64 INPUTS

We can test and check inputs of 64inputs cards.

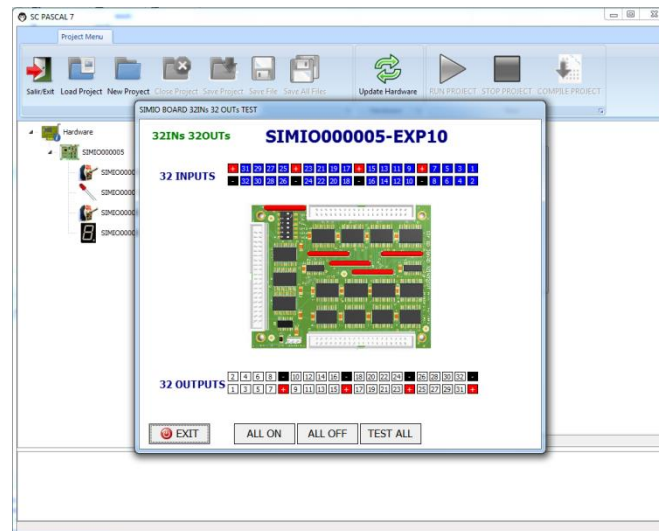


When any input changes, the box corresponding to that input will change its color.



### 9.5.3 Test EXP 32 INPUTS 32 OUPUTS

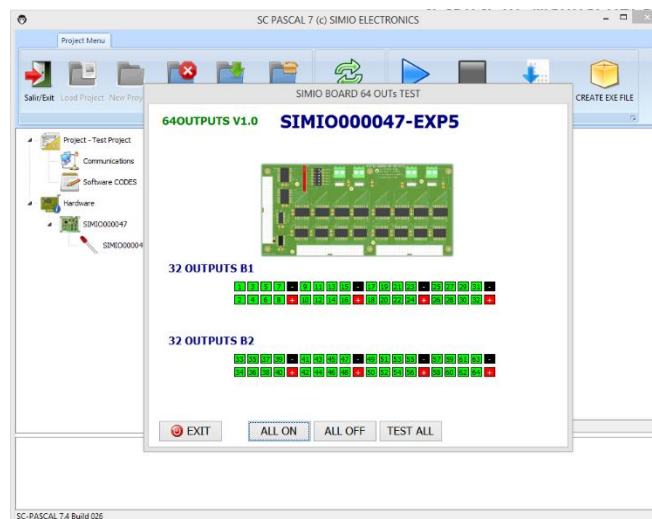
We can test and check inputs and outputs of 32inputs32outputs cards.



When any input changes, the box corresponding to that input will change its color. If we want to change the value of an output, we can click on the corresponding box. We can change all the outputs at the same time with the buttons “ALL ON”, “ALL OFF” or “TEST ALL”.

### 9.5.4 Test EXP 64 OUPUTS

We can test and check outputs of 64outputs cards.

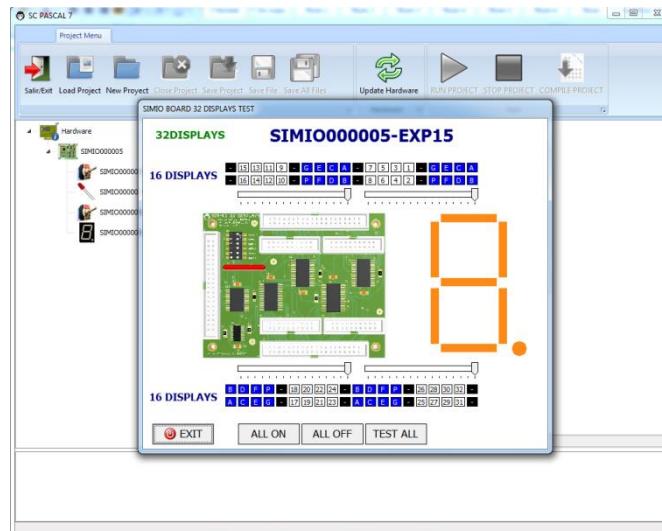


If we want to change the value of an output, we can click on the corresponding box. We can change all the outputs at the same time with the buttons “ALL ON”, “ALL OFF” or “TEST ALL”.



### 9.5.5 Test EXP 32 DISPLAYS

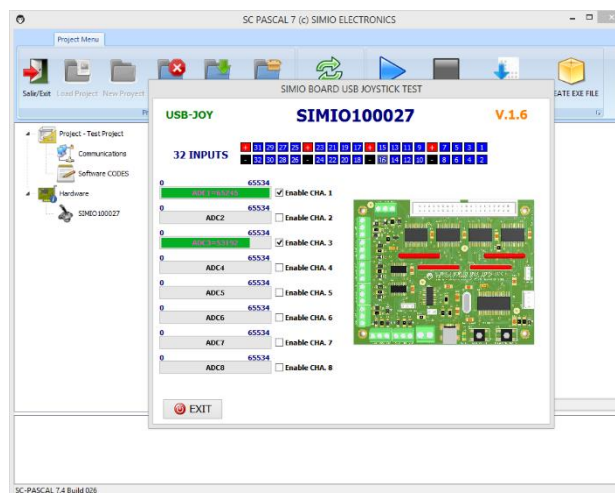
We can test and check displays of 32displays cards.



We can test each segment by clicking on the corresponding box. Sliding the control, we can change the leds intensity of each block too.

### 9.5.6. Test USB-JOY

We can test and check the SIMIO USB JOYSTICK card. This card includes 32 digital inputs and 8 ADC channels.



### 9.5.7. Test 8Servo-8Adc

We can test and check the SIMIO USB 8Servo-8Adc card. This card can control up to 8 servo motors and 8 ADC channels.

