Constructing new blocks in Scicos

Scicos Team

March 3, 2009

Contents

1 Introduction 3

2 Scicos data structures (editor level) 3

2.1 Scicos block . . . 4
2.2 Scicos graphics . . . 5
2.3 Scicos model . . . 6
2.4 Utilities Scilab functions . . . 9
  2.4.1 getvalue . . . 9
  2.4.2 set_io . . . 9

3 Scicos data structures (simulator level) 10

3.1 Scicos block type . . . 10
3.2 Scicos block structure of a C computational function (type 4) . . . 11
  3.2.1 Inputs/outputs . . . 12
  3.2.2 Events . . . 16
  3.2.3 Parameters . . . 17
  3.2.4 States and work . . . 22
  3.2.5 Zero crossing surfaces and modes . . . 26
  3.2.6 Miscellaneous . . . 27
3.3 Utilities C macros . . . 28
  3.3.1 Inputs/outputs . . . 28
  3.3.2 Events . . . 29
  3.3.3 Parameters . . . 29
  3.3.4 States and work . . . 30
  3.3.5 Zero crossing surfaces and modes . . . 30
  3.3.6 Miscellaneous . . . 30
3.4 Utilities C functions . . . 31
3.5 Scicos block structure of a Scilab computational function (type 5) . . . 32
  3.5.1 Inputs/outputs . . . 33
  3.5.2 Events . . . 33
  3.5.3 Parameters . . . 34
  3.5.4 States . . . 34
  3.5.5 Zero crossing surfaces and modes . . . 34
  3.5.6 Miscellaneous . . . 35
3.6 Utilities Scicos functions . . . 35
3.7 Use of flags . . . 38

List of Tables

1 Scicos block data structure fields 4
2 Scicos graphics data structure fields 5
3 Scicos model data structure fields 6
4 Scicos block type 10
5 C block structure definition 11
6 Editor/C data type number correspondence table. 20
7 Inputs/outputs C macros 28
8 Events C macros 29
9 Parameters C macros 29
10 States and work C macros 30
11 Zero crossing surfaces and modes C macros 30
12 Miscellaneous C macros 30
13 Scilab block structure definition 32
14 Arguments of the function getscicosvars 37

1 Introduction

Scicos (`www.scicos.org`) is a tool for modeling and simulating dynamical systems. Scicos 4.2.1 is distributed with scilabgtk-4.2 available for download from `www.scicos.org/downloads.html` and `www.scilabgtk.org`.

The graphical editor in Scicos can be used to construct models using blocks available in Scicos palettes. New blocks can be constructed based on existing blocks using the Super Block construction and masking, however, in some situations users may require basic blocks not available in Scicos palettes. Such new blocks can be constructed in Scicos but require a good understanding of the way Scicos works and the data structures used. This document will provide all the information needed for constructing new basic blocks.

A Scicos block is defined via two functions: an interfacing function expressed in Scilab language and a computational function written in C or Scilab. The interfacing function is used during model construction by interacting with the block diagram editor. It contains routines for initializing the block data structure, handling the GUI, etc. The computational function is used during simulation and contains the routines for computing the output, the state, etc. Each function deals with a different block data structure.

2 Scicos data structures (editor level)

Block data structures at the editor level are handled by block interfacing functions in Scicos. Such a function has the following skeleton:

```
function [x,y,typ]=my_interfunc(job,arg1,arg2)
x=[];y=[];typ=[];
select job
case ’plot’ then
  standard_draw(arg1)
case ’getinputs’ then
  [x,y,typ]=standard_inputs(arg1)
case ’getoutputs’ then
  [x,y,typ]=standard_outputs(arg1)
case ’getorigin’ then
  [x,y]=standard_origin(arg1)
case ’set’ then
  x=arg1; //in ’set’ x is the data structure of the block
  graphics=arg1.graphics;
  exprs=graphics.exprs;
  model=arg1.model;

  while %t do
    [ok,...,exprs]=getvalue(’Set block parameters’,....,exprs)
    if ~ok then break,end
    ...
    [model,graphics,ok]=set_io(model,graphics,in,out,...
                             clkin,clkout,in_implicit,out_implicit)
    ...
    if ok then
      graphics.exprs=exprs;
      x.graphics=graphics;
      x.model=model
      break
    end
  end
```

```
case ’define’ then
  model=scicos_model()
  model.sim=list(...)
  model.in=...
  ...

  exprs=string(in)
  gr_i=’xstringb(orig(1),orig(2),’’Block’’,sz(1),sz(2),’’fill’’)’
  x=standard_define([2 2],model,exprs,gr_i)
end
endfunction
```

The input arguments of this function are `arg1`, the data structure of the block, and `job`, which is an input flag. What the output arguments, `x`, `y` and `typ`, return depend on the input flag. In most cases, for `job` ’plot’ (initial draw), ’getinputs’ (position and type of input ports), ’getoutputs’ (position and type of output ports) and ’getorigin’ (initial shape of block) default functions (standard_draw(), standard_inputs(), standard_outputs() and standard_origin()) are used. The main work is done in the ’define’ case where the initial model and the layout of the block are defined, and in the ’set’ case, where the interfacing function handles the model update during used interaction using the functions getvalue() and set_io().

2.1 Scicos block

Fields	**Description**	**Type/size**	**Example**
graphics	Block graphic data structure	mlist	graphics=scicos_graphics()
model	Block model data structure	mlist	model=scicos_model()
gui	Name of the interfacing function	string	gui=’Myinterf’
doc	Block’s documentation.	list of size 2	doc=list(docfun,doc)

Table 1: Scicos block data structure fields

The basic structure that defines a Scicos block is a tlist that includes the fields: graphics, model, gui and doc.

- **graphics**: Mlist including the symbolic parameters, the graphical information concerning the layout of the block, and in general all the information needed at the editor level.
- **model**: Mlist including information needed for the compilation and simulation such as numerical parameters, the name of the computational function, port sizes, etc.
- **gui**: A string containing the name of the interfacing function associated with the block.
- **doc**: Field used for the documentation of the block.

2.2 Scicos graphics

Fields	Description	Type/size	Example
orig	Coordinate of block origin.	vector of size 2	graphics.orig=[1,1]
sz	Size of block.	vector of size 2	graphics.sz=[20,30]
flip	Block orientation.	boolean	graphics.flip=%f
exprs	Formal expression of block parameters.	vector of strings	graphics.exprs=["1.4";... sci2exp([5.1,5])]
pin	Link number connected to the input port.	vector	–
pout	Link number connected to the output port.	vector	–
pein	Link number connected to the event input port.	vector	–
peout	Link number connected to the event output port.	vector	–
gr_i	Graphic instructions.	vector of strings	–
id	Identification label.	string.	graphics.id=["ScopeA"]
in_implicit	Type of input port.	vector of strings	graphics.in_implicit=["E";"I"]
out_implicit	Type of output port.	vector of strings	graphics.out_implicit=["I";"I"]

Table 2: Scicos graphics data structure fields

Object including graphical information about the the layout of the block.

- **orig**: A row vector of double [xo,yo], where xo is the x coordinate of the block origin and yo is the y coordinate of the block origin. [xo,yo] is the coordinate of down-left point of the block shape.
- **sz**: A row vector of double [w,h], where w is the block width and h the block height.
- **flip**: A boolean that sets the block orientation. If true the input ports are on the left of the box and output ports are on the right. If false the input ports are on the right of the box and output ports are on the left.
- **theta**: A double that sets the angle of the Scicos object. This value is in degree and is included in [-360,360].
- **exprs**: A column vector of strings including formal expressions used in the dialog box of the block.
- **pin**: A column vector of integers. pin(i) is the number of the link connected to the ith regular input port (counting from one), or 0 if this port is not connected.
- **pout**: A column vector of integers. pout(i) is the number of the link connected to the ith regular output port (counting from one), or 0 if this port is not connected.
- **pein**: A column vector of integers. pein(i) is the number of the link connected to the ith event input port (counting from one), or 0 if this port is not connected.
- **peout**: A column vector of integers. peout(i) is the number of the link connected to the ith event output port (counting from one), or 0 if this port is not connected.
- **gr_i**: A column vector of strings including graphical expressions to customize the icon block shape. This field may be set with Icon sub_menu.
- **id**: A string including an identification for the block. The string is displayed under the block in the diagram.
- **in_implicit**: A column vector of strings including 'E' or 'I'. 'E' and 'I' stand respectively for explicit and implicit port, and this vector indicates the nature of each input port. For regular blocks (not implicit), this vector is empty or contains only "E".
- **out_implicit**: A column vector of strings including 'E' or 'I'. 'E' and 'I' stand respectively for explicit and implicit port, and this vector indicates the nature of each output port. For regular blocks (not implicit), this vector is empty or contains only "E".

2.3 Scicos model

Fields	Description	Type/size	Example
sim	Name/type of the computational function.	list of size 2	model.sim=list('tows_c',4)
in	First dimensions of regular input ports.	vector of size nin	model.in=[1;2]
in2	Second dimensions of regular input ports.	vector of size nin	model.in2=[3;1]
intyp	Data type of regular input ports.	vector of size nin	model.intyp=[1;8]
out	First dimensions of regular output ports.	vector of size nout	model.out=[1;2]
out2	Second dimensions of regular output ports.	vector of size nout	model.out2=[3;1]
outtyp	Data type of regular output ports.	vector of size nout	model.outtyp=[1;8]
evtin	Size of event input ports.	vector of size nevin	model.evtin=[1;1]
evtout	Size of event output ports.	vector of size nevout	model.evtout=[1;1]
state	Initial condition of continuous state.	vector of size nx	model.state=[0;0.1;-5.1]
dstate	Initial condition of discrete state.	vector of size nz	model.dstate=[0;0;-1]
odstate	Initial condition of object discrete state.	list of size noz	model.odstate=list([0;0;-1],... int32(3))
ipar	Integer parameters.	vector of size nipar	model.ipar=[1;2;-6]
rpar	Real parameters.	vector of size nrpar	model.rpar=[0.8;2.1;-6.55]
opar	Object parameters.	list of size nopar	model.opar=list([0.8;2.1]„... 1+2*%i)
blocktype	Type of the block.	character	model.blocktype='d'
firing	Initial date of output events.	vector of size nevout	model.firing=[-1;0.1]
dep_ut	Scheduling properties.	boolean vector of size 2	model.dep_ut=[%t;%f]
label	Label of the block.	string	model.label=[“My label“]
nzcross	Number of zero crossing.	integer	model.nzcross=1
nmode	Number of modes.	integer	model.nmode=0
equations	Modelica block definition.	list of size 4	model.equations=modelica(); model.equations.model=... 'Capacitor' model.equations.inputs='p' model.equations.outputs='n' model.equations.parameters=... list(['C','v'],list(C,v),[0,1])

Table 3: Scicos model data structure fields

Scicos model is a mlist containing block information used for the compilation and simulation.Scicos model contains the following fields:

- **sim**: A list containing two elements. The first element is a string containing the name of the computational function (C, Fortran,or Scilab). The second element is an integer specifying the type of the computational function. Currently type 4 and 5 are used, but older types continue to work to ensure backward compatibility. For some older case, `sim` can be a single string and that means that the type is supposed to be 0.
- **in**: A column vector of integers specifying the number and sizes of the first dimension (number of rows) of the regular input ports (in most cases ports are numbered sequentially from the top to the bottom on one side of the block). If no input port exists `in==[]`. The size can be negative, equal to zero or positive:
  - If a size is less than zero, the compiler will try to find the appropriate size. If two ports (input or output) have the same negative size (say -1) then the compiler forces them to have the same value. Idem for two sizes associated with the same port. For example to force an input to be a square matrix of arbitrary size, the row and column sizes can be set to -1.
  - If an input row size is equal to zero, the compiler assumes it is equal to the sum of all row sizes of the block outputs. Only used in special blocks such as DEMUX.
  - If a size is greater than zero, then it corresponds to the actual size (number of rows).
- **in2**: A column vector of integers specifying the second dimension (number of columns) of the regular input ports of the block. `in` and `in2` give the row and column dimension of the inputs. For compatibility, `in2` can be empty ([]). This means the dimensions of input ports are `[in,1]` The size can be negative, equal to zero or positive. See the case of `in` for details.

  - If a size is less than zero, the compiler will try to find the appropriate size. See the case of `in` for details.
  - If an input column size is equal to zero, the compiler assumes it is equal to the sum of all column sizes of the block outputs.
  - If a size is greater than zero, then it corresponds to the actual size (number of columns).

- **intyp**: A column vector of integers specifying the types of regular input ports. It has the same size as `in`. The types of regular input ports can be
  - 1: real matrix,
  - 2: complex matrix,
  - 3: int32 matrix,
  - 4: int16 matrix,
  - 5: int8 matrix,
  - 6: uint32 matrix,
  - 7: uint16 matrix,
  - 8: uint8 matrix.

- **out**: A column vector of integers specifying the number and sizes of the first dimension (number of rows) of the regular output ports. If no output port exists `out==[]`. The size can be negative, equal to zero or positive:
  - If a size is less than zero, the compiler will try to find the appropriate size. See the case of `in` for details.
  - If an output row size is equal to zero, the compiler assumes it is equal to the sum of all row sizes of the block inputs. Only used in special blocks such as MUX.

- **out2**: A column vector of integers specifying the second dimension (number of columns) of regular output ports. For compatibility, this dimension can be empty ([]). This means that the dimensions of output ports will be `[out,1]`. A size can be negative, equal to zero or positive:
  - If a size is less than zero, the compiler will try to find the appropriate size. See the case of `in` for details.
  - If a size is equal to zero, the compiler will affect this dimension by added all positive size found in that vector.
  - If a size is greater than zero, then it corresponds to the actual size (number of columns).

- **outtyp**: A column vector of integers specifying the types of regular output ports. Its size is equal to the size of `out`. See the case of `intyp` for types of regular output ports.

- **evtin**: A column vector of integers specifying the number and sizes of activation inputs. Currently activation ports can only be of size one. The size of the vector should be equal to the number of input event ports. If no event input port exists, `evtin` must be equal to [].

- **evtout**: A column vector of integers specifying the number and sizes of activation outputs. Currently activation ports can be only of size one. The size of the vector should be equal to the number of output event ports. If no event output port exists `evtout` must be equal to [].

- **state**: A column vector of doubles containing the initial value of the continuous-time state vector. It must be [] if no continuous state exists.

- **dstate**: A column vector of doubles containing initial values of discrete-time state vector. It must be [] if no discrete state exists.

- **odstate**: List containing initial values of discrete object states. It must be list() if no object states are used. Object states can be of any Scilab variable types. In computational functions of type 4 (C blocks) only list elements containing matrices of real, complex, int32, int16 ,int8 ,uint32, uit16 and uint8 are decoded for reading and writing.

- **rpar**: A column vector of doubles containing floating point block parameters. Must be [] if no floating point parameters.

- **ipar**: A column vector of integers containing integer block parameters. Must be [] if no integer parameters.

- **opar**: List of objects block parameters. Must be list() if no objects parameters. Objects parameters can be any types of Scilab variable. In the computational function case of type 4 (C blocks) only elements containing matrix of real, complex, int32, int16 ,int8 ,uint32, uit16 and uint8 will be correctly provided for reading.

- **blocktype**: A character that can be set to 'c' or 'd' indifferently for standard blocks. 'x' is used if we want to force the computational function to be called during the simulation phase even if the block does not contribute to the computation of the state derivative. 'l', 'm' and 's' are reserved. Not to be used.
- **firing**: A column vector of doubles for initial event firing times of size equal to the number of activation output ports (see evout). It contains output initial event dates (events generated before any input event arises). Negative values stands for no initial event programmed on the corresponding port.
- **dep_ut**: A boolean vector [dep_u, dep_t].
  - **dep_u**: true if block is always active. (output depends continuously of the time)
  - **dep_t**: true if block has direct feed-through, i.e., at least one of the outputs depends directly (not through the states) on one of the inputs. In other words, when the simulation function is called with flag 1, the value of an input is used to compute the output.
- **label**: A string that defines a label. It can be used to identify a block in order to access or modify its parameters during simulation.
- **nzcross**: An integer for the number of zero-crossing surfaces.
- **nmode**: An integer for the length of the mode register. Note that this gives the size of the vector mode and not the total number of modes in which a block can operate in. Suppose a block has 3 modes and each mode can take two values, then the block can have up to 2^3=8 modes.
- **equations**: Used in case of implicit blocks. Data structure of type modelica which contains modelica code description if any. That list contains four entries:
  - **model**: a string given the name of the file that contains the modelica function.
  - **inputs**: a column vector of strings that contains the names of the modelica variables used as inputs.
  - **outputs**: a column vector of strings that contains the names of the modelica variables used as outputs.
  - **parameters**: a list with two entries. The first is a vector of strings for the name of modelica variable names used as parameters and the second entries is a list that contains the value of parameters. Names of modelica states can also be informed with parameters. In that case a third entry is used to do the difference between parameters and states. For i,e: mo.parameters=list(['C','v'],list(C,v),[0,1]) means that 'C' is a parameter(0) of value C, and 'v' is a state(1) with initial value v.

2.4 Utilities Scilab functions

2.4.1 getvalue

- **[ok,x1,..,x14]=getvalue(desc,labels,typ,ini)**
  xwindow dialog for data acquisition.
  - **desc**: column vector of strings, dialog general comment
  - **labels**: n column vector of strings, labels(i) is the label of the ith required value
  - **typ: list(typ_1,dim_1,..,typ_n,dim_n)**:
    - **typ_i**: defines the type of the ith value, may have the following values:
      - **"mat"**: for constant matrix
      - **"col"**: for constant column vector
      - **"row"**: for constant row vector
      - **"vec"**: for constant vector
      - **"str"**: for string
      - **"lis"**: for list
    - **dim_i**: defines the size of the ith value it must be a integer or a 2-vector of integer, -1 stands for arbitrary dimension.
  - **ini**: n column vector of strings, ini(i) gives the suggested response for the ith required value.
  - **ok**: boolean ,%t if ok button pressed, %f if cancel button pressed.
  - **xi**: contains the ith value if ok=%t. If left hand side has one more xi than required, the last xi contains the vector of answered strings.

2.4.2 set_io

- **[model,graphics,ok]=set_io(model,graphics,in,out,clkin,clkout,in_implicit,out_implicit)**
  Checks and sets input/output port sizes.
  - **model**: scicos_model list
  - **graphics**: scicos_graphics list
  - **in**: list of regular input ports description
    list(in,in2,inttyp)
    - **in**: vector of first dimension. Size nin.
    - **in2**: vector of second dimension. Size nin.
    - **intyp**: vector of data type. Size nin.
  - **out**: list of regular output ports description.
    list(out,out2,outtyp)
    - **out**: vector of first dimension. Size nout.
    - **out2**: vector of second dimension. Size nout.
    - **outtyp**: vector of data type. Size nout.
  - **clkin**: vector of size of event input port.
  - **clkout**: vector of size of event output port.
  - **in_implicit**: vector of type of regular input port (”I” or “E“).
  - **out_implicit**: vector of type of regular output port (”I” or “E“).
  - **ok**: boolean ,%t if model has been updated with success, %f if not.

3 Scicos data structures (simulator level)

3.1 Scicos block type

Scicos has the possibility to handle and to call many different sorts of blocks. Some blocks in Scicos palettes are special and are only used internally by Scicos, such as synchro blocks and the Debug block, but most blocks are regular blocks which the user can get inspired by to construct new blocks. The following table gives the known Scicos block types, and is followed by the report of the type of the computational function with its associated calling sequence by block type.

Type	Description	Function type	Simulator call
-2	Event select block (synchro block).	-	Never called.
-1	If Then Else block (synchro block).	-	Never called.
0	C, Fortran or Scilab block. Calling sequence fixed. Obsolete.	Type 0.	Type 0.
1	C or Fortran block. Varying calling sequence. Obsolete.	Type 1.	Type 1.
2	C block. Calling sequence fixed. Obsolete.	Type 2.	Type 2.
3	Scilab block. Calling sequence fixed. Used but obsolete.	Type 3.	Type 2.
4	C block. Calling sequence fixed. In use.	Type 4.	Type 4.
5	Scilab block. Calling sequence fixed. In use.	Type 5.	Type 4.
1001	Fortran block. Dynamically linked. Obsolete.	Type 1.	Type 1.
2001	C block. Dynamically linked. Obsolete.	Type 1.	Type 1.
2004	C block. Dynamically linked. In use.	Type 4.	Type 4.
10001	Implicit C or Fortran block. Obsolete.	Type 10001.	Type 10001.
10002	Implicit C block. Obsolete.	Type 10002.	Type 10002.
10004	Implicit C block. In use.	Type 10004.	Type 4.
10005	Implicit Scilab block. In use.	Type 10005.	Type 4.
30004	Generic Modelica block. Dynamically linked. In use.	Type 10004.	Type 4.
99	Debug block.	Type 5.	Type 4.

Table 4: Scicos block type

Note that even if type 0, 1 and 2 are obsolete, they are still supported in Scicos; some blocks in the standard palettes of Scicos are still of these types. Block type 3 is still used by scifunc block (Scilab block) but users should prefer the type 5 when constructing a computational function in Scilab because it takes full advantage of new data structures. In fact it has all the the functionalities implemented for block type 4.

- **Calling sequence of computational function type 0**
  void myfun(flag,nevrt,t,xd,x,nx,z,nz,tvec,ntvec,rpar,nrpar,ipar,nipar,u,nu,y,ny)
- **Calling sequence of computational function type 1**
  void myfun(flag,nevrt,t,xd,x,nx,z,nz,tvec,ntvec,rpar,nrpar,ipar,nipar,u1,nu1,u2,nu2,...,y1,ny1,y2,ny2,...)
- **Calling sequence of computational function type 10001 (type 1 implicit)**
  void myfun(flag,nevrt,t,res,xd,x,nx,z,nz,tvec,ntvec,rpar,nrpar,ipar,nipar,u1,nu1,u2,nu2,...,y1,ny1,y2,ny2,...)
- **Calling sequence of computational function type 2**
  void myfun(flag,nevrt,t,xd,x,nx,z,nz,tvec,ntvec,rpar,nrpar,ipar,nipar,inptr,insz,nin,outptr,outsz,nout)
- **Calling sequence of computational function type 2 (zero crossing)**
  void myfun(flag,nevrt,t,xd,x,nx,z,nz,tvec,ntvec,rpar,nrpar,ipar,nipar,inptr,insz,nin,outptr,outsz,nout,g,ng)
- **Calling sequence of computational function type 10002 (type 2 implicit)**
  void myfun(flag,nevrt,t,res,xd,x,nx,z,nz,tvec,ntvec,rpar,nrpar,ipar,nipar,inptr,insz,nin,outptr,outsz,nout)
- **Calling sequence of computational function type 10002 (type 2 implicit with zero crossing)**
  void myfun(flag,nevrt,t,res,xd,x,nx,z,nz,tvec,ntvec,rpar,nrpar,ipar,nipar,inptr,insz,nin,outptr,outsz,nout,g,ng)
- **Calling sequence of computational function type 3**
  [x,y,z,tvec,xd]=myfun(flag,nevprt,t,x,z,rpar,ipar,u)
- **Calling sequence of computational function type 4**
  void myfun(scicos_block *block,int flag)
- **Calling sequence of computational function type 5**
  [block]=myfun(block,flag)

3.2 Scicos block structure of a C computational function (type 4)

The fields of the C structure of associated with a Scicos block provides all the necessary information to access block inputs, outputs, parameters, states, etc. This structure is defined in the file scicos_block4.h, and user must include that header in each C computational function:

```
#include <scicos/scicos_block4.h>
...
void mycomputfunc(scicos_block *block,int flag)
{
 ...
}
```

The fields, which can contain either C pointers or the data itself, are then accessible via the `*block` structure with the form `block->field`. These fields can be accessed directly but users should prefer using provided **C_macros** to access them. In the current version of Scicos, the `scicos_block` structure is defined as follows:

Fields	**Description**	**I/O**
int nevprt	Activation input number.	I
voidg funpt	Pointer to the computational function.	I
int type	computational function type.	I
int scsptr	Pointer to a Scilab function.	I
int nz	Length of the discrete state register.	I
double *z	Pointer to the discrete state register.	I/O
int noz	Number of discrete objects state.	I
int *ozsz	Size of discrete objects state.	I
int *oztyp	Type of discrete objects state.	I
void **ozptr	Pointer to discrete objects state.	I/O
int nx	Length of the continuous state register.	I
double *x	Pointer to the continuous state register.	I/O
double *xd	Pointer to the derivative continuous state register.	I/O
double *res	Pointer to the residual continuous state register.	O
int *xprop	Pointer to the continuous state properties register.	O
int nin	Number of regular input ports.	I
int *insz	Size of the regular input ports.	I
void **inptr	Pointer to the regular input ports.	I
int nout	Number of regular output ports.	I
int *outsz	Size of the regular output ports.	I
void **outptr	Pointer to the regular output ports.	O
int nevout	Length of the output event register.	I
double *evout	Pointer to the output event register.	O
int nrpar	Length of the real parameter register.	I
double *rpar	Pointer to the real parameter register.	I
int nipar	Length of the integer parameter register.	I
int *ipar	Pointer to the integer parameter register.	I
int nopar	Number of objects parameters.	I
int *oparsz	Size of object parameters.	I
int *opartyp	Type of object parameters.	I
void **oparptr	Pointer to the object parameters.	I
int ng	Length of the zero crossing register.	I
double *g	Pointer to the zero crossing register.	O
int ztyp	Say if the block use zero crossing register.	I
int *jroot	Pointer to the direction of zero crossing register.	I
char *label	Pointer to the label of the block.	I
void **work	Pointer to the workspace.	I/O
int nmode	Length of the mode register.	I
int *mode	Pointer to the mode register.	I/O

Table 5: C block structure definition

3.2.1 Inputs/outputs

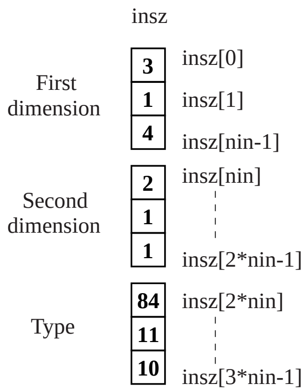
- **block->nin:** Integer that gives the number of regular input ports of the block. One cannot override the index `(3*block->nin)-1` when reading sizes of input ports in the array `insz` and the index `(block->nin)-1` when reading data in the array `inptr` with a C computational function. The number of regular input ports can also be obtained using the C macro `GetNin(block)`.

- **block->insz:** An array of integers of size `[3*nin,1]` that respectively gives the first dimensions, the second dimensions and the type of the data corresponding to the regular input ports. Note that this array of sizes differs from the array `ozsz` and `oparsz`; this is done to provide full compatibility with blocks that only use a single dimension (column vectors).

  Suppose you have a block with three inputs: the first input is an int32 matrix of size `[3,2]`, the second a single complex number (matrix of size `[1,1]`) and the last, a real matrix of size `[4,1]`. In the scicos_model of such a block, the inputs will be defined as follows:

  ```
  model.in = [3;1;4]
  model.in2 = [2;1;1]
  model.intyp = [2;1;3]
  ```

  and the corresponding `block->insz` field at the C computational function level will be coded as follows:

  `block->insz` array

  Note the difference here in the type numbers defined at the **editor level** (2,1,3) and the type numbers defined at the **C level** (84,11,10). The following table gives the correspondence for Scicos types:

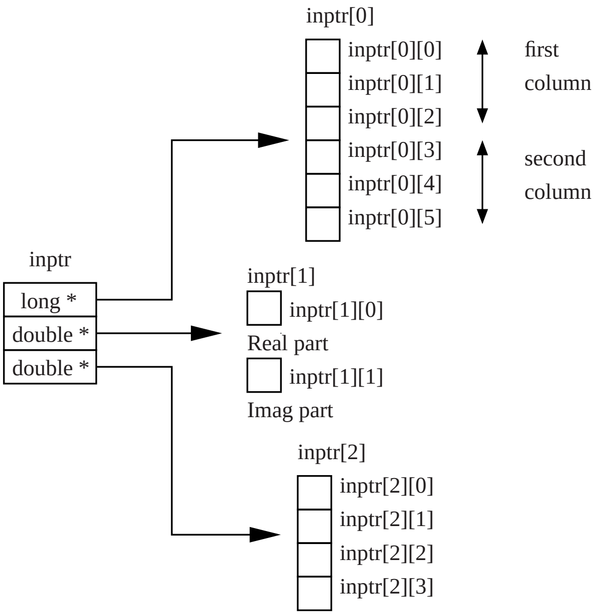
Editor Type	Editor Number	C Type	C Number
real	1	double	10
complex	2	double	11
int32	3	long	84
int16	4	short	82
int8	5	char	81
uint32	6	unsigned long	814
uint16	7	unsigned short	812
uint8	8	unsigned char	811

  Editor/C data type number correspondence table

- **block->inptr:** An array of pointers of size `[nin,1]` that allows direct access to the data contained in the regular input matrices. Consider the previous example (a block with three inputs: an int32 matrix of size `[3,2]`, a complex scalar and a real matrix of size `[4,1]`). `block->inptr` contains three pointers, and should be viewed as arrays containing the data for the int32, the real and the complex matrices as shown in the following figure.

  To directly access the data, the user can use the following instructions:

  ```
  #include <scicos/scicos_block4.h>
  ...
  SCSINT32_COP *ptr_i;
  ```



`block->inptr` array

```
SCSCOMPLEX_COP *ptr_dc;
SCSREAL_COP *ptr_d;
int n1,m1;
SCSINT32_COP cumsum_i=0;
int i;
...
void mycomputfunc(scicos_block *block,int flag)
{
...
/*get the ptrs of the first int32 regular input port*/
ptr_i = (SCSINT32_COP *) block->inptr[0];
/*get the ptrs of the second complex regular input port*/
ptr_dc = (SCSCOMPLEX_COP *) block->inptr[1];
/*get the ptrs of the third real regular input port*/
ptr_d = (SCSREAL_COP *) block->inptr[2];
...
/*get the dimension of the first int32 regular input port*/
n1=block->insz[0];
m1=block->insz[3];
...
/*compute the cumsum of the input int32 matrix*/
for(i=0;i<n1*m1;i++) {
cumsum_i += ptr_i[i];
}
...
}
```

It is highly recommended however that users use provided C macros to access the data:
`GetInPortPtrs(blk,x)`, `GetRealInPortPtrs(block,x)`,
`GetImagInPortPtrs(block,x)`, `Getint8InPortPtrs(block,x)`,
`Getint16InPortPtrs(block,x)`, `Getint32InPortPtrs(block,x)`,
`Getuint8InPortPtrs(block,x)`, `Getuint16InPortPtrs(block,x)`,
`Getuint32InPortPtrs(block,x)`
to have the appropiate pointer of the data to handle and
`GetNin(block)`, `GetInPortRows(block,x)`,
`GetInPortCols(block,x)`, `GetInPortSize(block,x,y)`,
`GetInType(block,x)`, `GetSizeOfIn(block,x)`
to handle number, dimensions and type of regular input ports.

x is numbered from 1 to nin and y numbered from 1 to 2.

For the previous example, this gives:

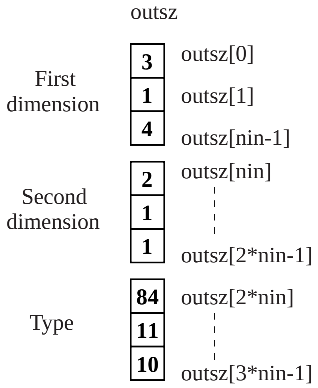
```
#include <scicos/scicos_block4.h>
...
SCSINT32_COP *ptr_i;
SCSCOMPLEX_COP *ptr_dc;
SCSREAL_COP *ptr_d;
int n1,m1;
SCSINT32_COP cumsum_i=0;
int i;
...
void mycomputfunc(scicos_block *block,int flag)
{
...
/*get the ptrs of the first int32 regular input port*/
ptr_i = Getint32InPortPtrs(block,1);
/*get the ptrs of the second complex regular input port*/
ptr_dc = GetRealInPortPtrs(block,2);
/*get the ptrs of the third real regular input port*/
ptr_d = GetRealInPortPtrs(block,3);
...
/*get the dimension of the first int32 regular input port*/
n1=GetInPortRows(block,1);
m1=GetInPortCols(block,1);
...
}
```

Finally note that the regular input port registers are only accessible for reading.

- **block->nout:** Integer that gives the number of regular output ports of the block. One cannot override the index `(3*block->nout)-1` when reading sizes of output ports in the array `outsz` and the index `(block->nout)-1` when reading data in the array `outptr` with a C computational function. The number of regular output ports can also be obtained using the C macro `GetNout(block)`.
- **block->outsz:** An array of integers of size `[3*nout,1]` that gives the first dimensions, the second dimensions and the type of data associated with the regular output ports. Note that this array of sizes differs from the array `ozsz` and `oparsz` to provide full compatibility with blocks that only use a single dimension. Suppose that you have a block with two outputs: the first output is an int32 matrix of size `[3,2]`, the second a single complex number (matrix of size `[1,1]`) and the last a real matrix of size `[4,1]`. In the scicos_model of such a block, the outputs will be defined as follows:

```
model.out = [3;1;4]
model.out2 = [2;1;1]
model.outtyp = [2;1;3]
```

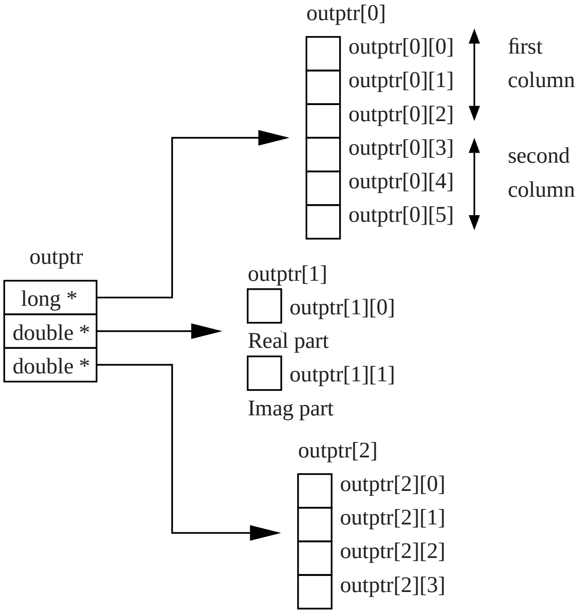
and the corresponding `block->outsz` field at C computational function level will be coded as follows:



`block->outsz` array

Note the difference here in the type numbers defined at the **editor level** (2,1,3) and the type numbers defined at the **C level** (84,11,10); see the previous table to have the correspondence for all Scicos type.

- **block->outptr:** An array of pointers of size `[nout,1]` that allows to directly access the data contained in the regular output matrices. Consider the previous example (block with three outputs: an int32 matrix of size `[3,2]`, a complex scalar and a real matrix of size `[4,1]`). `block->outptr` contains three pointers to the data for the int32, the real and the complex matrices as shown in the following figure.



`block->outptr` array

To directly access the data, the user can use the following instructions:

```
#include <scicos/scicos_block4.h>
...
SCSINT32_COP *ptr_i;
SCSCOMPLEX_COP *ptr_dc;
SCSREAL_COP *ptr_d;
int n1,m1;
SCSINT32_COP cumsum_i=0;
int i;
...
void mycomputfunc(scicos_block *block,int flag)
{
/*get the ptrs of the first int32 regular output port*/
ptr_i = (SCSINT32_COP *) block->outptr[0];
/*get the ptrs of the second complex regular output port*/
ptr_dc = (SCSCOMPLEX_COP *) block->outptr[1];
/*get the ptrs of the third real regular output port*/
ptr_d = (SCSREAL_COP *) block->outptr[2];
...
/*get the dimension of the first int32 regular output port*/
n1=block->outsz[0];
m1=block->outsz[3];
...
/*compute the cumsum of the output int32 matrix*/
for(i=0;i<n1*m1;i++) {
cumsum_i += ptr_i[i];
}
...
}
```

It is however recommended to use the set of C macros provided in Scicos:
`GetOutPortPtrs(block,x)`, `GetRealOutPortPtrs(block,x)`,
`GetImagOutPortPtrs(block,x)`, `Getint8OutPortPtrs(block,x)`,
`Getint16OutPortPtrs(block,x)`, `Getint32OutPortPtrs(block,x)`,
`Getuint8OutPortPtrs(block,x)`, `Getuint16OutPortPtrs(block,x)`,
`Getuint32OutPortPtrs(block,x)`
to have the appropiate pointer of the data to handle and
`GetNout(block)`, `GetOutPortRows(block,x)`,
`GetOutPortCols(block,x)`, `GetOutPortSize(block,x,y)`,
`GetOutType(block,x)`, `GetSizeOfOut(block,x)`
to handle number, dimensions and type of regular output ports.
x is numbered from 1 to nout and y is numbered from 1 to 2.

For the previous example this gives:

```
#include <scicos/scicos_block4.h>
...
SCSINT32_COP *ptr_i;
SCSCOMPLEX_COP *ptr_dc;
SCSREAL_COP *ptr_d;
int n1,m1;
SCSINT32_COP cumsum_i=0;
int i;
...
void mycomputfunc(scicos_block *block,int flag)
{
...
/*get the ptrs of the first int32 regular output port*/
ptr_i = GetOutPortPtrs(block,1);
/*get the ptrs of the second complex regular output port*/
ptr_dc = GetRealOutPortPtrs(block,2);
/*get the ptrs of the third real regular output port*/
ptr_d = GetRealOutPortPtrs(block,3);
...
/*get the dimension of the first int32 regular output port*/
n1=GetOutPortRows(block,1);
m1=GetOutPortCols(block,1);
...
}
```

Finally note that the regular output port registers must only be written into if `flag=1` or `flag=6.`

3.2.2 Events

- **block->nevprt:** Integer that gives the event input port number by which the block has been activated. This number is a binary coding. For example if block has two event inputs ports, `block->nevprt` can take the value 1 if the block has been activated through its first event input port, the value 2 if it has been activated through the second event input port and 3 if it is activated by the same event on both input ports 1 and 2. Note that `block->nevprt` can be -1 if the block is internally activated. One can also retrieve this number by using the C macros `GetNevIn(block)`.
- **block->nevout:** Integer that gives the number of event output ports of the block (also called the length of the output event register). One cannot override the index `(block->nevout)-1` when setting value of events in the output event register `evout`. The number of event output ports can also be obtained by the use of the C macro `GetNevOut(block)`.
- **block->evout:** Array of doubles of size `[nevout,1]` corresponding to the output event register. This register is used to program date of events during the simulation. The values in this array correspond to a delay relative to the current simulation time:

  $$t_{\text{event}} = t_{\text{cur}} + T_{\text{delay}} \tag{1}$$

  where t_{event} is the date of the programmed event, t_{cur} is the current time of simulation and T_{delay} **is the value that must be placed in the output event register.**

For example, suppose you want to generate an event through the first event output port, .1 unit of time after each call to the block, then you should use:

```
#include <scicos/scicos_block4.h>
...
void mycomputfunc(scicos_block *block,int flag)
{
...
if (flag==3) {
block->evout[0]=0.1;
}
...
}
```

Note all output events will be asynchronous with event activating the block even if you set `block->evout[x]=0`. The event output register must be only written into if `flag=3`.

3.2.3 Parameters

- **block->nrpar:** Integer that gives the length of the real parameter register. One cannot override the index `(block->nrpar)-1` when reading the value of real parameters in the register `rpar`. The total number of real parameters can also be obtained by the use of the C macro `GetNrpar(block)`.
- **block->rpar:** Array of double of size `[nrpar,1]` corresponding to the real parameter register. This register is used to pass real parameters coming from the Scicos working environment to your block model. The C type of that array is `double *` (or C scicos type `SCSREAL_COP *`). Suppose you have defined the following real parameters in the scicos_model of a block:

  ```
  model.rpar = [%pi;%pi/2;%pi/4]
  ```
  you can then retrieve it in the C computational function with:

  ```
  #include <scicos/scicos_block4.h>
  ...
  double PI;
  double PI_2;
  double PI_4;
  ...
  void mycomputfunc(scicos_block *block,int flag)
  {
  ...
  /*get the first value of the real param register*/
  PI = block->rpar[0];
  /*get the second value of the real param register*/
  PI_2 = block->rpar[1];
  /*get the third value of the real param register*/
  PI_4 = block->rpar[2];
  ...
  }
  ```

  You can also use the C macro `GetRparPtrs(block)` to get a pointer to the real parameter register. For example, if we define the following scicos_model in an interfacing function of a scicos block:

  ```
  A = [1.3 ; 4.5 ; 7.9 ; 9.8];
  B = [0.1 ; 0.98];
  model.rpar = [A;B]
  ```
  in the corresponding C computational function of that block, we use:

  ```
  #include <scicos/scicos_block4.h>
  ...
  double *rpar;
  double *A;
  double *B;
  ...
  ```

```
void mycomputfunc(scicos_block *block,int flag)
{
...
/*get ptrs of the real param register*/
rpar = GetRparPtrs(block);
/*get the A ptrs array*/
A = rpar;
/*get the B ptrs array*/
B = &rpar[4];
/*or B = rpar + 4;*/
...
}
```

Note that the real parameter register is only accessible for reading.

- **block->nipar:** Integer that gives the length of the integer parameter register. One cannot override the index `(block->nipar)-1` when reading the value of integer parameters in the register `ipar`. The total number of integer parameters can also be obtained by the use of the C macro `GetNipar(block)`.
- **block->ipar:** Array of integers of size `[nipar,1]` corresponding to the integer parameter register. This register is used to pass integer parameters coming from the Scicos working environment to your block model. The C type of that array is `int *` (or C scicos type `SCSINT_COP *`). Suppose you have defined the following integer parameters in the scicos_model of a block:

  ```
  model.ipar = [(1:3)';5]
  ```

  you can the retrieve it in the C computational function with:

  ```
  #include <scicos/scicos_block4.h>
  ...
  int one;
  int two;
  int three;
  int five;
  ...
  void mycomputfunc(scicos_block *block,int flag)
  {
  ...
  /*get the first value of the integer param register*/
  one = block->ipar[0];
  /*get the second value of the integer param register*/
  two = block->ipar[1];
  /*get the third value of the integer param register*/
  three = block->ipar[2];
  /*get the fourth value of the integer param register*/
  five = block->ipar[3];
  ...
  }
  ```

  You can also use the C macro `GetIparPtrs(block)` to get a pointer to the real parameter register. Most of the time in the Scicos C block libraries, the integer register is used to parameterize the length of real parameters. For example if you define the following scicos_model in a block:

  ```
  // set a random size for the first real parameters
  A_sz = int(rand(10)*10);
  // set a random size for the second real parameters
  B_sz = int(rand(10)*10);
  // set the first real parameters
  A = rand(A_sz,1,“uniform”);
  // set the second real parameters
  B = rand(B_sz,1,“normal”);
  // set ipar
  ```

```
model.ipar = [A_sz;B_sz]
// set rpar (length of A_sz+B_sz)
model.rpar = [A;B]
```

the array of real parameters (parameterized by `ipar`) can be retrieved in the corresponding C computational function with:

```
#include <scicos/scicos_block4.h>
...
int A_sz;
int B_sz;
double *rpar;
double *A;
double *B;
double cumsum;
int i;
...
void mycomputfunc(scicos_block *block,int flag)
{
...
/*get ptrs of the real param register*/
rpar = GetRparPtrs(block);
/*get size of the first real param register*/
A_sz = block->ipar[0];
/*get size of the second real param register*/
B_sz = block->ipar[1];
/*get the A ptrs array*/
A = rpar;
/*get the B ptrs array*/
B = &rpar[A_sz];
...
/*compute the cumsum of the first real parameter array*/
cumsum = 0;
for(i=0;i<A_sz;i++) {
cumsum += A[i];
}
...
/*compute the cumsum of the second real parameter array*/
cumsum = 0;
for(i=0;i<B_sz;i++) {
cumsum += B[i];
}
```

Note that integer parameters register is only accessible for reading.

- **block->nopar:** Integer that gives the number of the object parameters. One cannot override the index `(block->nopar)-1` when accessing data in the arrays `oparsz`, `opartyp` and `oparptr` in a C computational function. This value is also accessible via the C macro `GetNopar(block)`.
- **block->oparsz:** Array of integers of size `[nopar,2]` that contains the dimensions of matrices of object parameters. The first column is for the first dimension and the second for the second dimension. For example if we want the dimensions of the previous object parameters, we use the instructions:

```
#include <scicos/scicos_block4.h>
...
int nopar;
int n,m;
...
void mycomputfunc(scicos_block *block,int flag)
{
...
/*get the number of object parameter*/
nopar=block>nopar;
```

```
...
/*get number of row of the last object parameter*/
n=block>oparsz[nopar-1];
/*get number of column of the last object parameter*/
m=block>oparsz[2*nopar-1];
...
}
```

The dimensions of object parameters can be obtained with the following C macros:

```
GetOparSize(block,x,1); /*get first dimension of opar*/
GetOparSize(block,x,2); /*get second dimension of opar*/
```

with `x` an integer that gives the index of the object parameter, **numbered from 1 to nopar**.

- **block->opartyp:** Array of integers of size `[nopar,1]` that contains the type of matrices of object parameters. The following table gives the correspondence for Scicos type expressed in editor number, in C number and also corresponding C pointers and C macros used for `oparptr`:

Editor		**C**		
Type	Number	Number	Type	Macros
real matrix	1	10	double	SCSREAL_COP
complex matrix	2	11	double	SCSCOMPLEX_COP
int32 matrix	3	84	long int	SCSINT32_COP
int16 matrix	4	82	short	SCSINT16_COP
int8 matrix	5	81	char	SCSINT8_COP
uint32 matrix	6	814	unsigned long int	SCSUINT32_COP
uint16 matrix	7	812	unsigned short	SCSUNINT16_COP
uint8 matrix	8	811	unsigned char	SCSUINT8_COP
all others data		-1	double	SCSUNKNOWN_COP

Table 6: Editor/C data type number correspondence table.

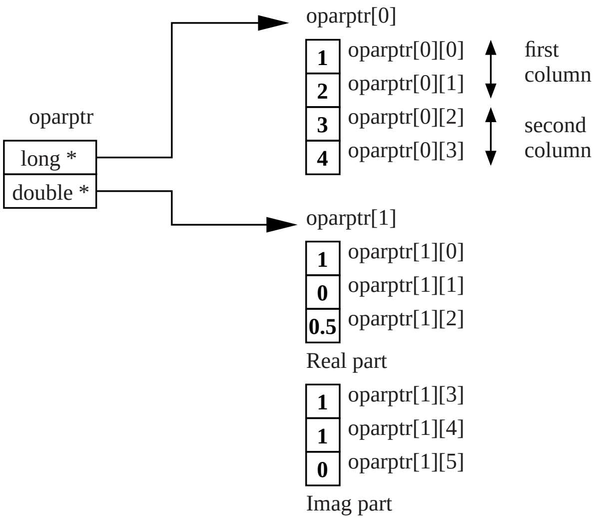
The type of object parameter can also be obtained by the use of the C macro `GetOparType(block,x)`. For example, if we want the C number type of the first object parameter, we use the following C instructions:

```
#include <scicos/scicos_block4.h>
...
int opartyp_1;
...
void mycomputfunc(scicos_block *block,int flag)
{
...
/*get the number type of the first object parameter*/
opartyp_1 = GetOparType(block,1);
...
}
```

- **block->oparptr:** An array of pointers of size `[nopar,1]` that gives direct access to the data contained in the object parameter. Suppose you have a block with the following `opar` field in scicos_model:

```
model.opar=list(int32([1,2;3,4]),[1+%i %i 0.5]);
```

Then we have two object parameters, one is a 32-bit integer matrix with two rows and two columns and the second is a vector of complex numbers that can be seen as a matrix of size `[1,3]`. At the C computational function level, the instructions `block->oparsz[0]`, `block->oparsz[1]`, `block->oparsz[2]`, `block->oparsz[3]` will respectively return the values 2, 1, 2, 3 and the instructions `block->opartyp[0]`, `block->opartyp[1]`, the values 11 and 84. `block->oparptr` will contain then two pointers, and should be viewed as arrays containing data of object parameters as shown in the following figure.



`block->oparptr` array

For example, to directly access the data, the user can use the following instructions:

```
#include <scicos/scicos_block4.h>
...
SCSINT32_COP *ptr_i;
SCSINT32_COP cumsum_i;
SCSCOMPLEX_COP *ptr_d;
SCSREAL_COP cumsum_d;
...
void mycomputfunc(scicos_block *block,int flag)
{
...
/*get the ptrs of an int32 object parameter*/
ptr_i = (SCSINT32_COP *) block->oparptr[0];
/*get the ptrs of a double object parameter*/
ptr_d = (SCSCOMPLEX_COP *) block->oparptr[1];
...
/*compute the cumsum of the int32 matrix*/
cumsum_i = ptr_i[0]+ptr_i[1]+ptr_i[2]+ptr_i[3];
...
/*compute the cumsum of the real part of the complex matrix*/
cumsum_d = ptr_d[0]+ptr_d[1]+ptr_d[2];
...
}
```

One can also use the set of C macros: `GetRealOparPtrs(block,x)`, `GetImagOparPtrs(block,x)`, `Getint8OparPtrs(block,x)`, `Getint16OparPtrs(block,x)`, `Getint32OparPtrs(block,x)`, `Getuint8OparPtrs(block,x)`, `Getuint16OparPtrs(block,x)`, `Getuint32OparPtrs(block,x)` to have the appropiate pointer of the data to handle.

x is numbered from 1 to nopar.

For the previous example that gives:

```
#include <scicos/scicos_block4.h>
...
SCSINT32_COP *ptr_i;
SCSREAL_COP *ptr_dr;
SCSREAL_COP *ptr_di;
...
void mycomputfunc(scicos_block *block,int flag)
{
...
```

```
/*get the ptrs of an int32 object parameter*/
ptr_i = Getint32OparPtrs(block,1);
/*get the ptrs of a double object parameter*/
ptr_dr = GetRealOparPtrs(block,2);
ptr_di = GetImagOparPtrs(block,2);
...
}
```

Note that object parameters register is only accessible for reading.

3.2.4 States and work

- **block->nx:** Integer that gives the length of the continuous state register. One cannot override the index `(block->nx)-1` when reading or writing data in the array `x`, `xd` or `res` with a C computational function.
- **block->x:** Array of doubles of size `[nx,1]` corresponding to the continuous state register. The value of a continuous state, for example the first state, can be obtained with the C instructions:

  ```
  #include <scicos/scicos_block4.h>
  ...
  double x_1;
  ...
  void mycomputfunc(scicos_block *block,int flag)
  {
  ...
  x_1=block->x[0];
  ...
  }
  ```

  Note that on `flag=4`, `6` or `2`, user can (re)initialize this register. The pointer to this array can also be retrieved via the C macro `GetState(block)`.
- **block->xd:** Array of doubles of size `[nx,1]` corresponding to the derivative of the continuous state register. It is an output of the simulation function if the block is an explicit block, i.e. the block models a system of Ordinary Differential Equations (ODE), otherwise, it is an input. In the latter case, the output is the residual vector `res` associated with a system of Differential Algebraic Equations (DAE).

  For example the Lorentz attractor expressed as an ODE system with three state variables:

  $$\dot{x} = f(x,t) \tag{2}$$

  will can be defined as follows:

  ```
  #include <scicos/scicos_block4.h>
  ...
  double *x = block->x;
  double *xd = block->xd;
  ...
  /* define parameters */
  double a = 10;
  double b = 28;
  double c = 8/3;
  ...
  void mycomputfunc(scicos_block *block,int flag)
  {
  ...
  if (flag == 0) {
  xd[0] = a*(x[1]-x[0]);
  xd[1] = x[0]*(b-x[2])-x[1];
  xd[2] = x[0]*x[1]-c*x[2];
  }
  ...
  }
  ```

- **block->res:** Array of doubles of size `[nx,1]` corresponding to Differential Algebraic Equation (DAE) residual. It is used to express block models corresponding to systems that have the following form:

  $$f(\dot{x}, x, t) \quad = \quad 0 \tag{3}$$

  For example the Lorentz attractor written as a DAE system with three state variables will be defined as follows:

  ```
  #include <scicos/scicos_block4.h>
  ...
  double *x = block->x;
  double *xd = block->xd;
  double *res = block->res;
  ...
  /* define parameters */
  double a = 10;
  double b = 28;
  double c = 8/3;
  ...
  void mycomputfunc(scicos_block *block,int flag)
  {
  ...
  if (flag == 0) {
  res[0] = - xd[0] + (a*(x[1]-x[0]));
  res[1] = - xd[1] + (x[0]*(b-x[2])-x[1]);
  res[2] = - xd[2] + (x[0]*x[1]-c*x[2]);
  }
  ...
  }
  ```

- **block->xprop:** Array of integers of size `[nx,1]` corresponding to the properties of the continuous state. That properties are set with `flag=7` when DAE solver is used to perform the simulation. Value of state property can be -1, that means that variable is an algebraic sate or 1 to say that variable is a differential state.

- **block->nz:** Integer that gives the length of the discrete state register. One cannot override the index `(block->nz)-1` when reading data in the array `z` with a C computational function. This value is also accessible via the C macros `GetNdstate(block)`.

- **block->z:** Array of doubles of size `[nz,1]` corresponding to the discrete state register. A value of a discrete state is directly readable (for example the second state) with the C instructions:

  ```
  #include <scicos/scicos_block4.h>
  ...
  double z_2;
  ...
  void mycomputfunc(scicos_block *block,int flag)
  {
  ...
  z_2=block->z[1];
  ...
  }
  ```

  Note that the state register should be only updated for `flag=4`, `6` or `2`. A pointer to this array can also be retrieve via the C macro `GetDstate(block)`.

- **block->noz:** Integer that gives the number of discrete object states. One cannot override the index `(block->noz)-1` when accessing data in the arrays `ozsz`, `oztyp` and `ozptr` in a C computational function. This value is also accessible via the C macro `GetNoz(block)`.

- **block->ozsz:** An array of integer of size `[noz,2]` that contains the dimensions of matrices of discrete object states. The first column is for the first dimension and the second for the second dimension. For example if we want the dimensions of the last object state, we use the instructions:

  ```
  #include <scicos/scicos_block4.h>
  ...
  int noz;
  ```

```
int n,m;
...
/*get the number of object state*/
noz=block>noz;
...
void mycomputfunc(scicos_block *block,int flag)
{
...
/*get number of row of the last object state*/
n=block>ozsz[noz-1];
/*get number of column of the last object state*/
m=block>ozsz[2*noz-1];
...
}
```

The dimensions of object discrete states can be obtained with the following C macro:

```
GetOzSize(block,x,1); /*get first dimension of oz*/
GetOzSize(block,x,2); /*get second dimension of oz*/
```

with `x` an integer that gives the index of the discrete object state, **numbered from 1 to noz**.

- **block->oztyp:** An array of integer of size `[noz,1]` that contains the type of matrices of discrete object states. The following table gives the correspondence table for Scicos type expressed in editor number, in C number and also corresponding C pointers and C macros used for `ozptr`:

Editor		**C**		
Type	Number	Number	Type	Macros
real matrix	1	10	double	SCSREAL_COP
complex matrix	2	11	double	SCSCOMPLEX_COP
int32 matrix	3	84	long int	SCSINT32_COP
int16 matrix	4	82	short	SCSINT16_COP
int8 matrix	5	81	char	SCSINT8_COP
uint32 matrix	6	814	unsigned long int	SCSUINT32_COP
uint16 matrix	7	812	unsigned short	SCSUNINT16_COP
uint8 matrix	8	811	unsigned char	SCSUINT8_COP
all others data		-1	double	SCSUNKNOWN_COP

Editor/C data type number correspondence table

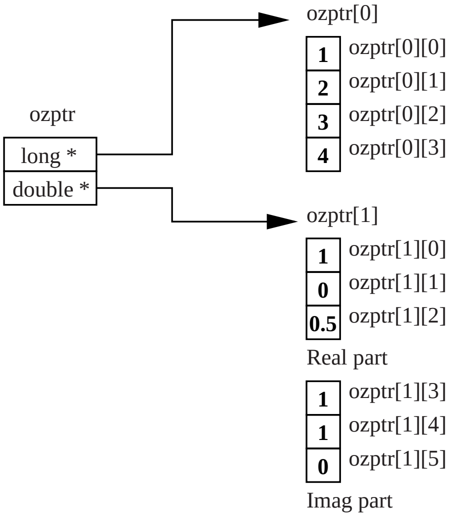
The type of discrete object state can also be obtained by the use of the C macro `GetOzType(block,x)`. For example, if we want the C number type of the first discrete object state, we use the following C instructions:

```
#include <scicos/scicos_block4.h>
...
int oztyp_1;
...
void mycomputfunc(scicos_block *block,int flag)
{
...
/*get the number type of the first object state*/
oztyp_1 = GetOzType(block,1);
...
}
```

- **block->ozptr:** An array of pointers of size `[noz,1]` that allows direct access to the data contained in the discrete object state. Suppose you have defined a block with the following **odstate** field in scicos_model:

```
model.odstate=list(int32([1,2;3,4]),[1+%i %i 0.5]);
```

Then we have two discrete object states, one is a 32-bit integer matrix with two rows and two columns and the second is a vector of complex numbers that can be seen as a matrix of size `[1,3]`. At the C computational function level, the instructions `block->ozsz[0]`, `block->ozsz[1]`, `block->ozsz[2]`, `block->ozsz[3]` will respectively return the values 2, 1, 2, 3 and the instructions `block->oztyp[0]`, `block->oztyp[1]` the values 11 and 84. `block->ozptr` will contain then two pointers, and should be viewed as arrays containing data of discrete object state as shown in the following figure.



`block->ozptr` array

For example, to directly access the data, the user can use the following instructions:

```
#include <scicos/scicos_block4.h>
...
SCSINT32_COP *ptr_i;
SCSINT32_COP cumsum_i;
SCSCOMPLEX_COP *ptr_d;
SCSREAL_COP cumsum_d; ...
void mycomputfunc(scicos_block *block,int flag)
{
...
/*get the ptrs of an int32 discrete object state*/
ptr_i = (SCSINT32_COP *) block->ozptr[0];
/*get the ptrs of a double discrete object state*/
ptr_d = (SCSCOMPLEX_COP *) block->ozptr[1];
...
/*compute the cumsum of the int32 matrix*/
cumsum_i = ptr_i[0]+ptr_i[1]+ptr_i[2]+ptr_i[3];
...
/*compute the cumsum of the real part of the complex matrix*/
cumsum_d = ptr_d[0]+ptr_d[1]+ptr_d[2];
...
}
```

One can also use the following C macros:
`GetRealOzPtrs(block,x)`, `GetImagOzPtrs(block,x)`,
`Getint8OzPtrs(block,x)`, `Getint16OzPtrs(block,x)`,
`Getint32OzPtrs(block,x)`, `Getuint8OzPtrs(block,x)`,
`Getuint16OzPtrs(block,x)`, `Getuint32OzPtrs(block,x)`
to have the appropriate pointer to the data to handle.
x is numbered from 1 to noz.
For the previous example this gives:

```
#include <scicos/scicos_block4.h>
...
SCSINT32_COP *ptr_i;
SCSREAL_COP *ptr_dr;
SCSREAL_COP *ptr_di;
...
```

```
void mycomputfunc(scicos_block *block,int flag)
{
...
/*get the ptrs of an int32 discrete object state*/
ptr_i = Getint32OzPtrs(block,1);
/*get the ptrs of a double discrete object state*/
ptr_dr = GetRealOzPtrs(block,2);
ptr_di = GetImagOzPtrs(block,2);
...
}
```

Finally note that the discrete object states should be only updated if `flag`=4, 6 or 2.

- **block->work:** A free pointer to set a working array for the block. **The work pointer must be first allocated when `flag = 4` and be freed when `flag = 5`.** The life cycle of this pointer in a C computational function should be:

```
#include <scicos/scicos_block4.h>
...
void** work=block->work;
...
void mycomputfunc(scicos_block *block,int flag)
{
...
/*initialization*/
if (flag==4) {
/*allocation of work*/
if (*work=scicos_malloc(sizeof(double))==NULL) {
set_block_error(-16);
return;
}
...
}
...
/*other flag treatment*/
...
/*finish*/
else if (flag==5) {
scicos_free(*work);
}
...
}
```

Note that if a block uses a `work` pointer, it will be called with `flag=2` even if the block does not use discrete states. The pointer of that array can also be retrieved via the C macro `GetWorkPtrs(block)`.

3.2.5 Zero crossing surfaces and modes

- **block->ng:** Integer that gives the number of zero crossing surfaces of the block. One cannot override the index `(block->ng)-1` when reading/writing data in the array `g` with a C computational function. The number of zero crossing surfaces can also be obtained by the use of the C macro `GetNg(block)`.
- **block->g:** Array of doubles of size `[ng,1]` corresponding to the zero crossing surface register. This register is used to detect zero crossings of functions of state variable during simulation. The register is accessible for writing if `flag = 9`. A pointer of this array can also be retrieved via the C macro `GetGPtrs(block)`.
- **block->jroot:** Array of integers of size `[ng,1]` corresponding to the direction of the zero crossing surface register. This register is used to know if a surface is crossed from negative to positive value or from positive to negative value. This register is typicallly used for reading with `flag = 2` or `flag = 3` with `nevprt < 0`. A pointer to this array can also be retrieved via the C macro `GetJrootPtrs(block)`.

- **block->nmode:** Integer that gives the number of modes of the block. One cannot override the index `(block->mode)-1` when reading/writing data in the array `mode` with a C computational function. The number of modes can also be obtained by the use of the C macro `GetNmode(block)`.
- **block->mode:** Array of integers of size `[nmode,1]` corresponding to the mode register. This register is used to set the mode of the zero crossing surfaces during simulation. It is accessible for writing if `flag = 9`. The pointer to this array can also be retrieved via the C macro `GetModePtrs(block)`.

3.2.6 Miscellaneous

- **block->type:** Integer that gives the type of the computational function. For C blocks, this number is equal to 4.
- **block->label:** String array that allows to retrieve the label of the block.

3.3 Utilities C macros

3.3.1 Inputs/outputs

Macro	Type	Description
GetNin(blk)	int	Returns the number of regular input ports.
GetInPortRows(blk,x)	int	Returns the number of rows (first dimension) of the regular input port number x.
GetInPortCols(blk,x)	int	Returns the number of columns (second dimension) of the regular input port number x.
GetInPortSize(blk,x,y)	int	Returns the regular input port size number x. (y=1 for the first dimension, y=2 for the second dimension)
GetInType(blk,x)	int	Returns the type of the regular input port number x.
GetInPortPtrs(blk,x)	void *	Returns the regular input port pointer of the port number x.
GetRealInPortPtrs(blk,x)	double *	Returns the pointer of real part of the regular input port number x.
GetImagInPortPtrs(blk,x)	double *	returns a pointer to the imaginary part of the regular input port number x.
Getint8InPortPtrs(blk,x)	char *	returns a pointer to the int8 typed regular input port number x.
Getint16InPortPtrs(blk,x)	short *	returns a pointer to the int16 typed regular input port number x.
Getint32InPortPtrs(blk,x)	long *	returns a pointer to the int32 typed regular input port number x.
Getuint8InPortPtrs(blk,x)	unsigned char*	returns pointer to the uint8 typed regular input port number x.
Getuint16InPortPtrs(blk,x)	unsigned short *	returns pointer to the uint16 typed regular input port number x.
Getuint32InPortPtrs(blk,x)	unsigned long *	returns a pointer to the uint32 typed regular input port number x.
GetSizeOfIn(blk,x)	int	Returns the size of the type of the regular input port number x in bytes.
GetNout(blk)	int	Returns the number of regular output ports.
GetOutPortRows(blk,x)	int	returns number of rows (first dimension) of the regular output port number x.
GetOutPortCols(blk,x)	int	Returns the number of columns (second dimension) of the regular output port number x.
GetOutPortSize(blk,x,y)	int	Returns the size of the regular output port number x. (y=1 for the first dimension, y=2 for the second dimension)
GetOutType(blk,x)	int	Returns the type of the regular output port number x.
GetOutPortPtrs(blk,x)	void *	Returns a pointer to the regular output port number x.
GetRealOutPortPtrs(blk,x)	double *	Returns a pointer to the real part of the regular output port number x.
GetImagOutPortPtrs(blk,x)	double *	Returns a pointer to the imaginary part of the regular output port number x.
Getint8OutPortPtrs(blk,x)	char *	Returns a pointer to the int8 typed regular output port number x.
Getint16OutPortPtrs(blk,x)	short *	Returns a pointer to the int16 typed regular output port number x.
Getint32OutPortPtrs(blk,x)	long *	Returns a pointer to the int32 typed regular output port number x.
Getuint8OutPortPtrs(blk,x)	unsigned char *	Returns a pointer to the uint8 typed regular output port number x.
Getuint16OutPortPtrs(blk,x)	unsigned short *	Returns a pointer to the uint16 typed regular output port number x.
Getuint32OutPortPtrs(blk,x)	unsigned long *	Returns a pointer to the uint32 typed regular output port number x.
GetSizeOfOut(blk,x)	int	Returns the size of the type of the regular output port number x in bytes.

Table 7: Inputs/outputs C macros

3.3.2 Events

Macro	**Type**	**Description**
GetNevIn(blk)	int	Returns the input event number.
GetNevOut(blk)	int	Returns the number of event output port.
GetNevOutPtrs(blk)	double *	Returns a pointer to the event output register.

Table 8: Events C macros

3.3.3 Parameters

Macro	**Type**	**Description**
GetNipar(blk)	int	Returns the number of integer parameters.
GetIparPtrs(blk)	int *	Returns a pointer to the integer parameters register
GetNrpar(blk)	int	Returns the number of real parameters.
GetRparPtrs(blk)	double *	Returns a pointer to the real parameters register.
GetNopar(blk)	int	Returns the number of object parameters.
GetOparType(blk,x)	int	Returns the type of object parameters number x.
GetOparSize(blk,x,y)	int	Returns the size of object parameters number x. (y=1 for the first dimension, y=2 for the second dimension)
GetOparPtrs(blk,x)	void *	Returns a pointer to the object parameters number x.
GetRealOparPtrs(blk,x)	double *	Returns a pointer to the real object parameters number x.
GetImagOparPtrs(blk,x)	double *	Returns a pointer to the imaginary part of the object parameters number x.
Getint8OparPtrs(blk,x)	char *	Returns a pointer to the int8 typed object parameters number x.
Getint16OparPtrs(blk,x)	short *	Returns a pointer to the int16 typed object parameters number x.
Getint32OparPtrs(blk,x)	long *	Returns a pointer to the int32 typed object parameters number x.
Getuint8OparPtrs(blk,x)	unsigned char *	Returns a pointer to the uint8 typed object parameters number x.
Getuint16OparPtrs(blk,x)	unsigned short *	Returns a pointer to the uint16 typed object parameters number x.
Getuint32OparPtrs(blk,x)	unsigned long *	Returns a pointer to the uint32 typed object parameters number x.
GetSizeOfOpar(blk,x)	int	Returns the size of the object parameters number x.

Table 9: Parameters C macros

3.3.4 States and work

Macro	**Type**	**Description**
GetNstate(blk)	int	Returns the number of continuous state.
GetState(blk)	double *	Returns the pointer of the continuous state register.
GetDerState(blk)	double *	Returns a pointer to the derivative of the continuous state register.
GetResState(blk)	double *	Returns a pointer to the residual of the continuous state register.
GetXpropPtrs(blk)	int *	Returns a pointer to the continuous state properties register.
GetNdstate(blk)	int	Returns the number of discrete states.
GetDstate(blk)	double *	Returns a pointer to the discrete state register.
GetNoz(blk)	int	Returns the number of object states.
GetOzType(blk,x)	int	Returns the type of object state number x.
GetOzSize(blk,x,y)	int	Returns the size of object state number x. (y=1 for the first dimension, y=2 for the second dimension).
GetOzPtrs(blk,x)	void *	Returns a pointer to the object state number x.
GetRealOzPtrs(blk,x)	double *	Returns a pointer to the real object state number x.
GetImagOzPtrs(blk,x)	double *	Returns a pointer to the imaginary part of the object state number x.
Getint8OzPtrs(blk,x)	char *	Returns a pointer to the int8 typed object state number x.
Getint16OzPtrs(blk,x)	short *	Returns a pointer to the int16 typed object state number x.
Getint32OzPtrs(blk,x)	long *	Returns a pointer to the int32 typed object state number x.
Getuint8OzPtrs(blk,x)	unsigned char *	Returns a pointer to the uint8 typed object state number x.
Getuint16OzPtrs(blk,x)	unsigned short *	Returns a pointer to the uint16 typed object state number x.
Getuint32OzPtrs(blk,x)	unsigned long *	Returns a pointer to the uint32 typed object state number x.
GetSizeOfOz(blk,x)	int	Returns the size of the object state number x.
GetWorkPtrs(blk)	void *	Returns a pointer to the Work array.

Table 10: States and work C macros

3.3.5 Zero crossing surfaces and modes

Macro	**Type**	**Description**
GetNg(blk)	int	Returns the number of zero crossing surfaces.
GetGPtrs(blk)	double *	Returns a pointer to the zero crossing register.
GetJrootPtrs(blk)	int *	Returns a pointer to the direction of the zero crossing register.
GetNmode(blk)	int	Returns the number of modes.
GetModePtrs(blk)	int *	Returns a pointer to the mode register.

Table 11: Zero crossing surfaces and modes C macros

3.3.6 Miscellaneous

Macro	**Type**	**Description**
GetLabelPtrs(blk)	char *	Returns the pointer to the label of the block.

Table 12: Miscellaneous C macros

3.4 Utilities C functions

The scicos_block4.h header provides some utility functions to interact with the simulator in the C computational functions.

- **void do_cold_restart();**
  This function forces the solver to do a cold restart. It should be used in situations where the block creates a non smooth signal. Note that in most situations, non smooth situations are detected by zero-crossings and this function is not needed. This block is used in very exceptional situations.
- **int get_phase_simulation();**
  This function returns an integer which indicates whether the simulator is realizing time domain integration. It can return:
  - **1:** The simulator is on a discrete activation time.
  - **2:** The simulator is realizing a continuous time domain integration.
- **double get_scicos_time();**
  This function returns the current time of simulation.
- **int get_block_number();**
  This function returns an integer: the block index in the compiled structure. Each block in the simulated diagram has a single index, and blocks are numbered from 1 to `nblk` (the total number of blocks in the compiled structure).
- **void set_block_error(int);**
  Function to set a specific error number during the simulation for the current block. If used, after the execution of the computational function of the block, the simulator ends and returns an error message associated with the number given as integer argument.
  The following calls are allowed:
  - **set_block_error(-1);:** the block has been called with input out of its domain,
  - **set_block_error(-2);:** singularity in a block,
  - **set_block_error(-3);:** block produces an internal error,
  - **set_block_error(-16);:** cannot allocate memory in block.
- **void Coserror(char *fmt,...);**
  Function to return a specific error message in the Scicos editor. If used, after the execution of the computational function of the block, the simulator will end and will return the error message specified in argument (of type char*).
- **void end_scicos_sim();**
  A very specific function to set the current time of the simulator to the final integration time thus ending the simulation.
  Only expert user should use this function.
- **void set_pointer_xproperty(int* pointer);** (obsolete)
  This function sets a vector of integers to inform the type (algebraic or differential) of the continuous state variables associated with the block. Note that this function is obsolete. User will prefer direct access to the field block->xprop or the macro approach (with `GetXpropPtrs(blk)`) to set the property of continuous state.
- **void * scicos_malloc(size_t);**
  This function must be used to do allocation of Scicos pointers inside a C computational function and in particular when `flag`=4 for the work pointer `*block->work`.
- **void scicos_free(void *p);**
  This function must be used to free Scicos pointers inside a C computational function and in particular when `flag`=5 for the work pointer `*block->work`.

3.5 Scicos block structure of a Scilab computational function (type 5)

A Scicos computational function of type 5 can be realized by the use of a Scilab function. That function doesn't really differs from all other Scilab function: one can use all functions and instructions of the Scilab language inside that function to do the computation.

Such a function must be written in a file with extension .sci, must be loaded inside Scilab by the common loading Scilab function (`exec`, `getf`, `getd`, `genlib`,...) and must have two right hand side arguments and one left hand side argument, as the following calling sequence:

```
function block=myblock(block,flag)
...
//your simulation instructions
...
endfunction
```

When the simulator is calling such a computational function, it build a Scilab structure (in the previous example this is the named `block` rhs/lhs arguments) from his own internal C representation of a block structure (see section 3.2 for more details about the C structure of scicos blocks). That structure is a typed list variable that has the following fields.

Fields	Description	I/O
nevprt	Activation input number.	I
funpt	Pointer to the computational function.	I
type	computational function type.	I
scsptr	Pointer to a scilab function.	I
nz	Length of the discrete state register.	I
z	Discrete state register.	I/O
noz	Number of discrete objects state.	I
ozsz	Vector of size of discrete objects state.	I
oztyp	Vector of type of discrete objects state.	I
oz	List of discrete objects state.	I/O
nx	Length of the continuous state register.	I
x	Continuous state register.	I/O
xd	Derivative continuous state register.	I/O
res	Residual continuous state register.	O
nin	Number of regular input ports.	I
insz	Vector of size of the regular input ports.	I
inptr	List of regular input ports.	I
nout	Number of regular output ports.	I
outsz	Vector of size of the regular output ports.	I
outptr	List of regular output ports.	O
nevout	Length of the output event register.	I
evout	Output event register.	O
nrpar	Length of the real parameter register.	I
rpar	Real parameter register.	I
nipar	Length of the integer parameter register.	I
ipar	Integer parameter register.	I
nopar	Number of objects parameters.	I
oparsz	Vector of size of object parameters.	I
opartyp	Vector of type of object parameters.	I
opar	List of the object parameters.	I
ng	Length of the zero crossing register.	I
g	Zero crossing register.	O
ztyp	Say if the block use zero crossing register.	I
jroot	Vector of direction of zero crossing register.	I
label	String, the label of the block.	I
work	Not Used	
nmode	Length of the mode register.	I
mode	Pointer to the mode register.	I/O

Table 13: Scilab block structure definition

Each fields are then accessible inside the computational function by the use of `block.field`.

3.5.1 Inputs/outputs

- **block.nin:** a scalar that gives the number of regular input ports. This is a read only data.
- **block.insz:** a vector of size `3*nin`, that gives the dimensions and types of the regular input ports.
  - `block.insz(1:nin)`: are the first dimensions.
  - `block.insz(nin+1:2*nin)`: are the second dimensions.
  - `block.insz(2*nin+1:3*nin)`: are the type of data (C coding).

  This is a read only data.
- **block.inptr:** a list of size `nin` that enclosed typed matrices for regular input ports. Each element correspond to only one regular input port. Then i-th matrix of the block.inptr list will have the dimensions [`block.insz(i)`, `block.insz(nin+i)`] and the type `block.insz(2*nin+i)`.
  The data type that can be provided by regular input ports are:
  - **1:** matrix of real numbers,
  - **2:** matrix of complex numbers,
  - **3:** matrix of int32 numbers,
  - **4:** matrix of int16 numbers,
  - **5:** matrix of int8 numbers,
  - **6:** matrix of uint32 numbers,
  - **7:** matrix of uint16 numbers,
  - **8:** matrix of uint8 numbers.

  This is a read only data.
- **block.nout:** a scalar that gives the number of regular output ports. This is a read only data.
- **block.outsz:** a vector of size `3*nout`, that gives the dimensions and types of the regular output ports.
  - `block.outsz(1:nout)`: are the first dimensions.
  - `block.outsz(nout+1:2*nout)`: are the second dimensions.
  - `block.outsz(2*nout+1:3*nout)`: are the type of data (C coding).

  This is a read only data.
- **block.outptr:** a list of size `nout` that enclosed typed matrices for regular output ports. Each element correspond to only one regular output port. Then i-th matrix of the block.outptr list will have the dimensions [`block.outsz(i)`, `block.outsz(nin+i)`] and the type `block.outsz(2*nin+i)`.
  The data type that can be provided by regular output ports are:
  - **1:** matrix of real numbers,
  - **2:** matrix of complex numbers,
  - **3:** matrix of int32 numbers,
  - **4:** matrix of int16 numbers,
  - **5:** matrix of int8 numbers,
  - **6:** matrix of uint32 numbers,
  - **7:** matrix of uint16 numbers,
  - **8:** matrix of uint8 numbers.

  Values of regular output ports will be saved in the C structure of the block only for `flag=6` and `flag=1`.

3.5.2 Events

- **block.nevprt:** a scalar given the event input port number (binary coding) which has activated the block. This is a read only data.
- **block.nevout:** a scalar given the number of output event port of the block. This is a read only data.
- **block.evout:** a vector of size `nevout` corresponding to the register of output event. Values of output event register will be saved in the C structure of the block only for `flag=3`.

3.5.3 Parameters

- **block.nrpar:** a scalar given the number of real parameters. This is a read only data.
- **block.rpar:** a vector of size `nrpar` corresponding to the real parameter register. This is a read only data.
- **block.nipar:** a scalar given the number of integer parameters. This is a read only data.
- **block.ipar:** a vector of size `nipar` corresponding to the integer parameter register. This is a read only data.
- **block.nopar:** a scalar given the number of object parameters. This is a read only data.
- **block.oparsz:** a matrix of size `[nopar,2]`, that respectively gives the first and the second dimension of object parameters. This is a read only data.
- **block.opartyp:** a vector of size `nopar` given the C coding type of data. This is a read only data.
- **block.opar:** a list of size `nopar` given the values of object parameters. Each element of `opar` can be either a typed matrix or a list. Only matrix that encloses numbers of type real, complex, int32, int16, int8, uint32, uint16 and uint8 are allowed, all other types of data will be enclosed in a sub-list. This is a read only data.

3.5.4 States

- **block.nz:** a scalar giving the number of discrete state for the block. This is a read only data.
- **block.z:** a vector of size `nz` corresponding to the discrete state register. Values of discrete state register will be saved in the C structure of the block only for `flag=4`, `flag=6`, `flag=2` and `flag=5`.
- **block.noz:** a scalar that gives the number of discrete object state. This is a read only data.
- **block.ozsz:** a matrix of size `[noz,2]`, that respectively gives the first and the second dimension of discrete object state. This is a read only data.
- **block.oztyp:** a vector of size `noz` giving the C coding type of data.
- **block.oz:** a list of size `noz` giving the values of discrete object states. Each element of `oz` can be either a typed matrix or a list. Only matrix that encloses numbers of type real, complex, int32, int16, int8, uint32, uint16 and uint8 are allowed, all other types of data will be enclosed in a sub-list. Values of discrete object state will be saved in the C structure of the block only for `flag=4`, `flag=6`, `flag=2` and `flag=5`.
- **block.nx:** a scalar giving the number of continuous states for the block. This is a read only data.
- **block.x:** a vector of size `nx` giving the value of the continuous state register. Values of the continuous state register will be saved in the C structure of the block only for `flag=4`, `flag=6` and `flag=2`.
- **block.xd:** a vector of size `nx` giving the value of the derivative continuous state register. Values of the derivative continuous state register will be saved in the C structure of the block only for `flag=4`, `flag=6`, `flag=0` and `flag=2`.
- **block.res:** a vector of size `nx` corresponding to the Differential Algebraic Equation (DAE) residual. Values of that register will be saved in the C structure of the block only for `flag=0`, and `flag=10`.
- **block.xprop:** a vector of size `nx` corresponding to the properties of the continuous state. It used for DAE solver and works with `flag=7`. Values of that register can be -1 or 1 to says respectively that state is an algebraic or a differential state.

3.5.5 Zero crossing surfaces and modes

- **block.ng:** a scalar giving the number of zero crossing surfaces for the block. This is a read only data.
- **block.g:** a vector of size `ng` corresponding to the zero crossing register. Values of that register will be saved in the C structure of the block only for `flag=9`.
- **block.jroot:** a vector of size `ng` corresponding to the direction of the zero crossing register.
- **block.nmode:** a scalar giving the number of mode for the block. This is a read only data.
- **block.mode:** a vector of size `mode` that corresponds to the mode register. Values of that register will be saved in the C structure of the block only for `flag`=9, with `phase_simulation`=1.

3.5.6 Miscellaneous

- **block.type:** a scalar giving the type of the block. This is a read only data.
- **block.label:** a string giving the label of the block. This is a read only data.

3.6 Utilities Scicos functions

- **blk=curblock()**
  Return the current called scicos block during the simulation.
  - blk: the current block number in the compiled structure.
- **[label]=getblocklabel(blk)**
  Returns the label of a scicos block.
  - blk: Integer parameter. Set the index of a block (in the compiled structure).
  - label: String parameter. Gives the string of the label of the block numbered blk.
- **[psim]=phase_simulation()**
  That function says if the Scicos simulator is realizing the time domain integration.
  - psim: get the current phase of the simulation
    1: The simulator is on a discrete activation time.
    2: The simulator is realizing a continuous time domain integration.
- **[xprop]=pointer_xproperty**
  Returns the type of all continuous time state variables. This function returns a vector that informs the type (algebraic or differential) of the continuous state variables of a block.
  - xprop: The value gives the type of the states:
    -1: an algebraic state.
    1: a differential state.
- **t=scicos_time()**
  Returns the current simulation time during simulation.
  - t: that is the current simulation time returned as double.
- **set_xproperty(xprop)** (obsolete)
  Sets the type of a continuous time state variable. This function set a vector to inform the type (algebraic or differential) of the continuous state variables of a block.
  - xprop: The value gives the type of the states:
    -1: an algebraic state.
    1: a differential state.

  Note that this function is obsolete. User will prefer direct access to the field `block.xprop` to set the property of continuous state.
- **set_blockerror(n)**
  Sets the block error number. Function to set a specific error during the simulation for the current block. If used, after the execution of the computational function of the block, the simulation ends and Scicos returns an error message associated with the number given in the argument.
  - n: an error number. The following calls are allowed:
    - set_blockerror(-1)
      the block has been called with input out of its domain
    - set_blockerror(-2)
      singularity in a block
    - set_blockerror(-3)
      block produces an internal error
    - set_blockerror(-16)
      cannot allocate memory in block

- **coserror(str)**
  Abort the simulation to return an error message.
  - str: A string given the error message.

- **[myvar]=getscicosvars([str1;str2;...])**
  Supervisor utility function. That utility function is used to retrieve working arrays of Scicos simulator and compiler during simulation. It can be used inside a Scilab block to get information of all type of blocks. That function is very useful to debug diagrams and to do prototypes of simulations.
  - str,str1,str2,...: That parameter can be a string or a matrix of string. The following entries are allowed:

str	Description
x	returns the continuous state register.
nx	returns the length of the continuous state register.
xptr	returns the pointers register of the continuous state register.
zcptr	returns the pointers register of the zero-crossing surfaces register.
z	returns the discrete state register.
nz	returns the length of the continuous state register.
zptr	returns the pointers register of the discrete state register.
noz	returns the number of elements of the discrete object state list.
oz	returns the discrete object state list.
ozsz	returns the size of the elements of the discrete object state list.
oztyp	returns the type of the elements of the discrete object state list.
ozptr	returns the pointers register of the discrete object state list.
rpar	returns the real parameter register.
rpptr	returns the pointers register of the real parameter register.
ipar	returns the integer parameter register.
ipptr	returns the pointers register of the integer parameter register.
opar	returns the object parameter list.
oparsz	returns the size of the elements of the object parameter list.
opartyp	returns the type of the elements of the object parameter list.
opptr	returns the pointers register of the object parameter list.
outtb	returns the output register.
inpptr	returns the pointers register of the number of regular input ports.
outptr	returns the pointers register of the number of regular output ports.
inplnk	returns the pointers register of the links connected to regular input ports.
outlnk	returns the pointers register of the links connected to regular output ports.
subs	not used
tevts	returns the current date register of the agenda.
evtspt	returns the current event register of the agenda.
pointi	returns the next event to be activated.
iord	returns the vector of blocks activated at the start of the simulation.
oord	returns the vector of blocks whose outputs affects computation of continuous state derivatives.
zord	returns the vector of blocks whose outputs affects computation of zero-crossing surfaces.
funtyp	returns the vector of type of computational functions.
ztyp	returns the pointers vector for blocks which use zero-crossing surfaces.
cord	returns the vector of blocks whose outputs evolve continuously.
ordclk	returns the matrix associated to blocks activated by output activation ports.
clkptr	returns the pointers vector for output activation ports.
ordptr	returns the pointers vector to ordclk designating the part of ordclk corresponding to a given activation.
critev	returns the vector of the critical events.
mod	returns the vector pointers of block modes.
nmod	returns the length of the vector pointers of block modes.
iz	returns the register that store pointers of block->work.
izptr	returns the pointers vector of the register that store C pointers of block->work.
nblk	returns the number of block.
outtbptr	returns the register that store C pointers of outtb.
outtbpsz	returns the register that store the size of the elements of outtb.
outtbtyp	returns the register that store the type of the elements of outtb.
nlnk	returns the number of output.
ncord	returns the number of blocks whose outputs evolve continuously.
nordptr	returns the number of blocks whose outputs evolve by activation.
iwa	n.d.
blocks	returns a list that contains all block structures contains in the diagram.
ng	returns length of the zero-crossing surfaces register.
g	returns the zero-crossing surfaces register.
t0	returns the current time of the simulation.
tf	returns the final time of the simulation.
Atol	returns the integrator absolute tolerance for the numerical solver.
rtol	returns the integrator relative tolerance for the numerical solver.
ttol	returns the tolerance on time of the simulator.
deltat	returns the maximum integration time interval.
hmax	returns the maximum step size for the numerical solver.
nelem	returns the number of elements in outtb.
outtb_elem	returns the vector of the number of elements in outtb.

Table 14: Arguments of the function getscicosvars

- myvar: That output parameter can be an int32 matrix, a double matrix or a Tlist. This is given by the input parameter.

3.7 Use of flags

During the simulation, the computational functions will be called with a given flag that corresponds to the task to be realized and with the event number by which it has been activated.

Flag 4: Initialization

This is done only once in the initialization phase for all blocks. Input event numbers are not used in that case. Outputs and states can be initialized. Some blocks use also this flag to open files, to do allocation and initialization of the field `block->work` or initialize graphic windows.

Flag 6: Initialization, fixed-point computation

Flag 6 is used to set constraints that must be satisfied at the initial time. Scicos uses a fixed point computation scheme to force the constraints so the blocks are called more than once with flag 6 at time 0. This is a special initialization technique for example to find the steady state of a system before running the simulation. Input event numbers are not used in this case.

Flag 1: Output computation

The output computation can be performed many times in one time step of the simulation in particular when the diagram contains blocks that use both discrete and continuous sates and zero crossing surfaces. In the current version of Scicos all blocks are called with flag = 1 at least once in every simulation time step, even if they don't have any outputs.

Flag 2: Discrete state computation

If blocks use states, this flag is when the state registers `block->x`, `block->z`, `block->oz`, `block->work` must be set during discrete activation (with `block->nevptr`$\geq$0) but also to compute `block->x` in the case of activation due to an internal zero crossing, in which case the input event number `block->nevptr` will be -1.

Flag 0: Continuous state derivative computation

This flag is used when the derivative `block->xd` or residual `block->res` of the continuous state needs to be set. Only blocks that use continuous state are called with flag=0.

Flag 3: Output event computation

Output event computation is done for blocks with output event register during discrete activation but also zero crossing activation. Note that in this latter case, the input event number `block->nevptr` will be -1.

Flag 9: Modes and zero crossing computation

Flag 9 is used to evaluate the function of zero crossings `block->g` and to set the modes, `block->mode`.

Flag 5: Ending

All blocks are called with flag = 5 before the end of the simulation or when the simulator aborts the simulation in case an error occurs during the simulation. Input event numbers are not used in that case.

Flag 7: Properties of the continuous state variables

Set the properties of the continuous state variables. Used for the description of DAE system (also internally used and generated by Scicos/Modelica implementation).

Flag 10: Jacobian computation

Computation of Jacobian matrix of the system (internally used and generated by Scicos/Modelica implementation).