Institutt for energiteknikk
OECD Halden Reactor Project

The Software Bus

Communication System

REFERENCE MANUAL

SOFTWARE BUS
DOCUMENTATION

Revision: 4.12.1

Institutt for energiteknikk
OECD Halden Reactor Project

Published by : Institutt for Energiteknikk, OECD Halden Reactor Project
Date : June 2018
Revision : 4.12.1

SOFTWARE BUS DOCUMENTATION

MODULE

SWBus - a function library for interprocess communication.

DESCRIPTION

SWBus contains functions and data structures for interprocess communication.

FILES

The library in binary form consists of the following archive files:

Unix platforms:

libswbus.a	(optimized)
libswbus_debug.a	(non-optimized with debug information)

Windows platforms:

swbusML.lib	(optimized, using LIBC.LIB)
swbusMLd.lib	(non-optimized with debug information, using LIBCD.LIB)
swbusMT.lib	(optimized, using LIBCMT.LIB)
swbusMTd.lib	(non-optimized with debug information, using LIBCMTD.LIB)
swbusMD.lib	(optimized, using MSVCRT.LIB)
swbusMDd.lib	(non-optimized with debug information, using MSVCRTD.LIB)

Within the **SWBus** installation environment these files are located on ***$BUSPATH/lib/$ARCH***

ENVIRONMENT VARIABLES

See the SWBus User's Guide for a complete list of environment variables used by the Software Bus.

COMPILING AND LINKING

The example makefiles serves as templates for how to compile and link SWBus applications. The makefiles are located on ***$BUSPATH/examples***. A ***README*** file is also provided on ***$BUSPATH/examples***.

SEE ALSO

control

AUTHOR

SWBus was developed by Tord Akerbæk, Håkon Jokstad and Michael Louka at IFE.

FUNCTION

SbAdd - add an attribute to a SWBus class.

SYNOPSIS

```
SbTSTI  SbAdd(theClass,theType,name,value)
  SbTSTI  theClass;
  SbTSTI  theType;
  char*   name;
  SbTAddr value;
```

DESCRIPTION

SbAdd extends ***theClass*** with a new attribute called ***name*** of class ***theType***. Existing objects of class ***theClass*** will be resized to give room for the new attribute, using the value pointed to by ***value*** as the initial value for the new attribute for the objects.

RETURN VALUES

theClass contains a list of attribute descriptions. A new description is added to this list, and the symbol table index (STI) of this description is returned. If the function fails, **SbCError** is returned.

EXAMPLE

```
SbTSTI myClass, myObj;
int    level = 3;

myClass = SbSub( SbCCEmpty, "myClass", SbCNull );
myObj   = SbCreate( SbCPLocal, myClass, "myObj", SbCNull, NULL );
SbAdd( myClass, SbCCInteger, "level", &level );
```

SEE ALSO

SbSub, SbCreate

FUNCTION

SbAddr - return a pointer to a data area formatted according to a class description.

SYNOPSIS

```
SbTAddr  SbAddr(theClass,...)
  SbTSTI theClass;
```

DESCRIPTION

Memory corresponding to the class size of ***theClass*** is allocated. The memory is filled with the vararg parameters which are interpreted according to the attribute descriptions of ***theClass***.

RETURN VALUES

The address returned from **SbAddr** is static, so do not try to store it.

EXAMPLE

```
SbPut( object, SbAddr(SbClass(object),3,param,func(3,14),9.34) );
```

FUNCTION

SbCall - call a Software Bus function.

SYNOPSIS

```
SbTSTI  SbCall(func,in,out)
   SbTSTI  func;
   SbTSTI  in;
   SbTSTI  out;
```

DESCRIPTION

SbCall will call the SWBus function ***func*** using ***in*** and ***out*** as actual parameters. The classes of ***in*** and ***out*** should match the classes specified when the function was declared using **SbDeclare**.

func is a function object declared with **SbDeclare** and created with **SbCreate**. The actual code hidden in this function should have the standard SWBus function signature:

```
SbTSTI code( SbTSTI object, SbTSTI function, SbTSTI in, SbTSTI out )
```

When calling a function in a remote process, **SbCall** will set up a timeout and return **SbCError** if no reply has been received within the timeout period. The timeout period is specified in the constructor of **SbCCRemote**, it is stored in an attribute of **SbCCRemote** and might be changed at any moment later on. Default value for the timeout is specified by the environment variable **BUSTIMEOUT** if set, otherwise it is 30.000 (30 seconds).

RETURN VALUES

SbCall returns ***out***, or **SbCError** if the operation failed.

EXAMPLE

```
/* User defined function that returns its input value incremented by 1 */
SbTSTI incFunc( SbTSTI o, SbTSTI f, SbTSTI in, SbTSTI out )
{
   int value;
   SbGet( in, &value );
   value = value + 1;
   SbPut( out, &value );
   return out;
}

/* Declare the function class */
SbTSTI funcClass = SbDeclare( "incFuncClass", SbCCInteger, SbCCInteger,
                              &incFunc, SbCLC );

/* Create a callable function instance */
SbTSTI func = SbCreate( SbCPLocal, funcClass, "incFunc", SbCNull, NULL );

/* Create parameters */
int iValue = 4;
```

```
SbTSTI in  = SbCreate( SbCPLocal, SbCCInteger, "i1", SbCNull, NULL );
SbTSTI out = SbCreate( SbCPLocal, SbCCInteger, "i2", SbCNull, NULL );
SbPut( in, &iValue ); /* inputvalue equals 4 */

/* Call the function */
SbCall( func, in, out );

/* Get the result */
int result;
SbGet( out, &result ); /* result should now equal 5 */
```

SEE ALSO

SbDeclare, SbCreate

FUNCTION

SbCast -cast an object to another class.

SYNOPSIS

```
SbTSTI  SbCast(aClass,anObject)
  SbTSTI  aClass;
  SbTSTI  anObject;
```

DESCRIPTION

SbCast returns the symbol table index (STI) of a static object of class ***aClass*** that shares the properties, including data area, of ***anObject***.

RETURN VALUES

SbCast returns a static STI. Do not expect this index to survive. You can not store it.

FUNCTION

SbCopy - copy data from one object to another.

SYNOPSIS

```
SbTSTI  SbCopy(source,target)
  SbTSTI  source;
  SbTSTI  target;
```

DESCRIPTION

SbCopy copies the value of the object ***source*** into the data area of the object ***target***. If ***target*** is a reference to an object in a remote process, the actual transfer of data does not take place until **SbFlush** is called. If ***target*** has subscribers, the value is distributed to the subscribing objects when the copy is completed.

RETURN VALUES

SbCopy returns ***target***, or **SbCError** if the operation failed.

EXAMPLE

```
/* Set cVar in local process to 314.
  Copy the value to sVar in server process */

SbTSTI server, source, target;
int    value;
server   = SbId( SbCPLocal, "server" );
source   = SbCreate( SbCPLocal, SbCCInteger, "cVar", SbCNull, NULL );
target   = SbCreate( server,    SbCCInteger, "sVar", SbCNull, NULL );
value = 314;
SbPut( source, &value );
SbCopy( source, target );
SbFlush( SbCPLocal, server );
```

SEE ALSO

SbPut, SbGet, SbFlush

FUNCTION

SbCreate - create a new Software Bus object.

SYNOPSIS

```
SbTSTI  SbCreate(p,c,n,cp,d)
   SbTSTI  p;          /* Process containing new object */
   SbTSTI  c;          /* Class of new object */
   char*   n;          /* Name of new object */
   SbTSTI  cp;         /* Parameter to constructor function */
   SbTAddr d;          /* Pointer to fixed data area */
```

DESCRIPTION

SbCreate requests the process ***p*** to instantiate a new SWBus object called ***n*** of class ***c***. ***n*** will be located in the process ***p***. When the new object has been created, its constructor function is called with the parameter object ***cp***. If ***d*** is NULL, memory for the object value is allocated by the SWBus. If ***d*** is not NULL, ***d*** is assumed to point to a user-specified memory location for the object value.

RETURN VALUES

SbCreate returns the symbol table index (STI) of the new object, or **SbCError** if the operation failed.

EXAMPLE

```
SbTSTI myObject, intObject;
int i;

/* Create an integer object in the local process
   with data area at variable i */
intObject = SbCreate( SbCPLocal, SbCCInteger, "intObject", SbCNull, &i );


/* Create a myclass object in remote process theProcess */
myObject = SbCreate( theProcess, myClass, "myObject", SbCNull, NULL );
```

SEE ALSO

SbDelete, SbSub

FUNCTION

SbDataWaiting - test for existence of unsent data

SYNOPSIS

```
int32  SbDataWaiting();
```

DESCRIPTION

For a communication channel that is set nonblocking, the sending of data might be halted if the socket fifo is full. **SbDataWaiting** checks if there exists data that is waiting to be sent to another process.

If unsent data exists, the normal select of the main loop function should be skipped, and **SbDispatch** should be called directly.

RETURN VALUES

In nonblocking mode (when **SbNonBlocking** has been called for one or more communication channels), **SbDataWaiting** returns true (nonzero) if unsent data exists, false (zero) otherwise. In blocking mode (the default) **SbDataWaiting** always returns false.

NOTE

SbDataWaiting is mandatory in user-defined main loop functions. If the default SWBus main loop function is used, **SbDataWaiting** should not be used.

EXAMPLE

See an example under **SbSetLoop**

SEE ALSO

SbNonBlocking, SbSetLoop

FUNCTION

SbDeclare - declare a new function class.

SYNOPSIS

```
typedef SbTSTI (*SbTCode) (SbTSTI o, SbTSTI f, SbTSTI in, SbTSTI out);

SbTSTI  SbDeclare(name,inClass,outClass,code,format)
  char*   name;
  SbTSTI  inClass;
  SbTSTI  outClass;
  SbTCode code;
  SbTType format;
```

DESCRIPTION

SbDeclare declares a function class called ***name***. Instances of this class are callable functions that can be used as arguments to **SbCall**. ***inClass*** and ***outClass*** specifies the classes of the formal input and output parameters respectively. The actual function code is situated at the address ***code***. ***format*** specifies the type of language of the function.

Legal values for ***format*** are:

SbCLC - c-language function

The function situated at address ***code*** must have the standard SWBus function signature defined by **SbTCode**.

When an instance of the function class is called using **SbCall**, the classes of the actual parameters must conform to ***inClass*** and ***outClass*** respectively.

RETURN VALUES

SbDeclare returns the symbol table index (STI) of the new function class, or **SbCError** if the operation failed.

EXAMPLE

See an example at the manual page for **SbCall**.

SEE ALSO

SbCall

FUNCTION

SbDelete - delete a Software Bus object

SYNOPSIS

```
SbTSTI  SbDelete(obj,param)
  SbTSTI obj;          /* The object to be deleted */
  SbTSTI param;        /* Parameter to destructor function */
```

DESCRIPTION

SbDelete deletes a previously created SWBus object. The destructor method (**~destruct**) of the class of ***obj*** will, if present, be called with parameter ***param*** before ***obj*** is deleted.

Deleting a predefined SWBus object is not permitted, neither is deleting a SWBus class that is used by active SWBus objects.

RETURN VALUES

SbDelete returns **SbCOK** if the object was successfully deleted, or **SbCError** if the operation failed.

EXAMPLE

```
/* Create a class and an object */
SbTSTI myClass = SbSub( SbCCEmpty, "myClass", SbCNull );
SbTSTI myObj   = SbCreate( SbCPLocal, myClass, "myObj", SbCNull, NULL );

/* Delete the object and the class */
SbDelete( myObj, SbCNull );
SbDelete( myClass, SbCNull );
```

SEE ALSO

SbCreate

FUNCTION

SbDispatch - call SWBus functions requested from other processes.

SYNOPSIS

```
SbTSTI  SbDispatch(r,w,e)
   fd_set* r;      /* input channel file descriptors */
   fd_set* w;      /* output channel file descriptors */
   fd_set* e;      /* exeption file descriptors */
```

DESCRIPTION

SbDispatch receives input from other SWBus processes and responds correspondingly. Futher, **SbDispatch** checks all active **io** objects (instances of SWBus class **SbCCIo**), to see if any of them has file descriptor sets containing any of the file descriptors in **r**. If so, the call-back function of the **io** object is activated.

RETURN VALUES

SbDispatch returns **SbCOK**, or **SbCError** if the operation failed.

WARNING

Argument ***w*** and ***e*** are currently ignored.

NOTE

SbDispatch is mandatory in user-defined main loopfunctions. If the default SWBus main loop function is used, **SbDispatch** is not needed.

EXAMPLE

See an example at the manual page for **SbSetLoop**.

SEE ALSO

SbLoop, SbSetLoop, SbPeriodic, SbFDGet, SbFDAnd, SbFDOr, SbCCIo

FUNCTION

SbEndLoop - terminate a loop stated by SbLoop.

SYNOPSIS

```
void  SbEndLoop();
```

DESCRIPTION

SbEndLoop terminates the main loop function most recently started by **SbLoop**. **SbEndLoop** deactivates the condition tested for by **SbIsActive**. The next time the main loop function loops, **SbIsActive** returns false and the loop terminates.

SEE ALSO

SbLoop, SbIsActive

FUNCTION

SbExit - leave the Software Bus.

SYNOPSIS

```
SbTSTI  SbExit();
```

DESCRIPTION

SbExit unregisters the process with **control** and leaves the SWBus.

RETURN VALUES

SbExit returns **SbCOK** or **SbCError**.

SEE ALSO

SbInit

FUNCTION

SbFDAnd - logical AND of two file descriptor sets.

SbFDOr - logical OR of two file descriptor sets.

SYNOPSIS

```
int32  SbFDAnd(fds1,fds2,size1,size2)
  fd_set* fds1;
  fd_set* fds2;
  int32   size1;
  int32   size2;

int32  SbFDOr(fds1,fds2,size1,size2)
  fd_set* fds1;
  fd_set* fds2;
  int32   size1;
  int32   size2;
```

DESCRIPTION

SbFDAnd and **SbFDOr** perform a logical AND and OR operation of two file descriptor sets ***fds1*** and ***fds2***. ***fds1*** is modified to contain the result of the operation. ***size1*** and ***size2*** should specify the size (as if calculated by **SbFDSize**) of ***fds1*** and ***fds2*** respectively.

RETURN VALUE

The size of the resulting file descriptor set (as if calculated by **SbFDSize**) is returned.

EXAMPLE

See an example at the manual page for **SbSetLoop**

SEE ALSO

SbFDSize, SbFDGet, SbDispatch, SbSetLoop, SbCCIo

FUNCTION

SbFDGet - get the file descriptor set currently used by the SWBus.

SYNOPSIS

```
int32  SbFDGet(fdc,fds)
  int     fdc;
  fd_set* fds;
```

DESCRIPTION

SbFDGet modifies the file descriptor set pointed to by ***fds*** to contain the file descriptors currently used by the Software Bus. The set includes file descriptors for connections to remote processes, and for **io** objects (instances of class **SbCCIo**).

Legal values for ***fdc*** are **SbCFDREAD**.

RETURN VALUES

SbFDGet returns the size of the file descriptor set (as if calculated by **SbFDSize**).

EXAMPLE

See an example at the manual page for **SbSetLoop**

SEE ALSO

SbDispatch, SbFDAnd, SbFDOr, SbFDSize, SbSetLoop, SbCCIo

FUNCTION

SbFDSize - return the actual size of a file descriptor set.

SYNOPSIS

```
int32  SbFDSize(fds)
  fd_set* fds;
```

DESCRIPTION

SbFDSize calculates the actual size of a file descriptor set. On Unix platforms, the actual size of a file descriptor set is defined as a number greater than the highest bit-number in ***fds*** that is set. On Windows platforms, it is defined as the value ***fds->fd_count***.

RETURN VALUE

The size of the file descriptor set according to the definition above is returned.

EXAMPLE

See an example at the manual page for **SbSetLoop**

SEE ALSO

SbFDGet, SbFDAnd, SbFDOr, SbDispatch, SbSetLoop, SbCCIo

FUNCTION

SbFirst - start an iteration through a set of Software Bus objects.

SbNext - continue an iteration through set of Software Bus objects.

SYNOPSIS

```
SbTSTI  SbFirst(parent,option)
  SbTSTI  parent;
  SbTBP   option;

SbTSTI  SbNext(parent)
  SbTSTI  parent;
```

DESCRIPTION

SbFirst initiates a traversal through a set of objects and returns the first object. **SbNext** iterates over the remaining objects. Different object sets are traversed depending on the values of ***parent*** and ***option***. ***parent*** is a SWBus object. ***option*** is bit pattern.

If ***parent*** is a list, the list is traversed and each element in the list is returned.

If ***parent*** is an array, the array is traversed and each element in the array is returned.

If ***parent*** is a dynamic class and ***option*** is **SbCBPInstance**, the instances of this class and all its subclasses are traversed.

If ***parent*** is an object of a dynamic class and ***option*** is **SbCBPChild**, all attributes of that object are traversed. If ***option*** in addition is **SbCBPDeep**, all attributes within the attributes are traversed as well. If ***option*** is **SbCBPParent**, ***parent*** is regarded as an attribute of some object, and this object is included in the traversal.

The order in which the objects are returned is not defined.

RETURN VALUES

SbFirst and **SbNext** return symbol table indices (STI's). When the traversal is completed, **SbNext** returns **SbCNull**. If there are no objects in the set, **SbFirst** returns **SbCNull**.

EXAMPLE

```
/* Print name of all remote processes */
for ( p = SbFirst(SbCCRemote,SbCBPInstance);
      p != SbCNull;
      p = SbNext(SbCCRemote) )
{
  printf( "%s\n", SbName(p) );
}

/* Print all elements in local process */
for ( p = SbFirst(SbCPLocal,SbCBPChild);
      p != SbCNull;
      p = SbNext(SbCPLocal) )
```

```
{
  printf( "%s\n", SbName(p) );
}
```

SEE ALSO

SbHead, SbTail, SbCCList, SbCCGroup

FUNCTION

SbFlush - transfer buffered data between processes.

SYNOPSIS

```
SbTSTI  SbFlush(source,target)
  SbTSTI source;
  SbTSTI target;
```

DESCRIPTION

SbFlush transfers all data buffered for output in the process ***source*** to the process ***target***. Data is placed in the transfer buffers by calling **SbPut, SbSend**, **SbSendToOne** or **SbCopy**.

If ***source*** is **SbCPLocal**, data are sent from the local process.

If ***source*** is a remote process(an instance of **SbCCRemote**), data are sent from the remote process.

If ***target*** is a remote process (an instance of **SbCCRemote**), buffered data is sent to ***target***.

If ***target*** is a group of processes, buffered data is sent to the processes in ***target***.

If ***target*** is **SbCNull**, buffered data is sent to all processes connected to ***source***.

RETURN VALUES

SbFlush returns ***target***, or **SbCError** if the operation failed.

EXAMPLE

```
int   i = 32;
float f = 3.14;
SbTSTI myProcess = SbId( SbCPLocal, "myProcess" );
SbTSTI myInt     = SbId( myProcess, "myInt" );
SbTSTI myFloat   = SbId( myProcess, "myFloat" );
SbPut( myInt, &i );
SbPut( myFloat, &f );
SbFlush( SbCPLocal, myProcess );     /* int & float are sent */
```

SEE ALSO

SbPut, SbSend, SbSnedToOne, SbCopy, SbLink

FUNCTION

SbHead - get the first object in a list of SWBus objects.

SbTail - get all but the first of a list of SWBus objects.

SbCrown - insert a new element at the beginning of a list.

SbBehead - remove the first element from a list.

SYNOPSIS

```
SbTSTI  SbHead(list)
  SbTSTI  list;

SbTSTI  SbTail(list)
  SbTSTI  list;

SbTSTI  SbCrown(list,object)
  SbTSTI  list;
  SbTSTI  object;

SbTSTI  SbBehead(list)
  SbTSTI  list;
```

DESCRIPTION

SbHead returns the STI of the first element of ***list***, or **SbCNull** if ***list*** is empty.

SbTail returns the STI of a sublist containing all elements but the first element of ***list***.

SbCrown inserts a new element at the beginning of ***list*** and returns ***list***.

SbBehead unlinks the first element from ***list***, and returns ***list***.

EXAMPLE

```
SbTSTI aList, anInt, anOtherInt, anElement, aTail;

/* Create a list and add two elements */
aList = SbCreate( SbCPLocal, SbCCList, "aList", SbCNull, NULL );
SbCrown( aList, anInt);
SbCrown( aList, anOtherInt );

/* Traverse the list */
for ( aTail = aList;
      (anElement=SbHead(aTail)) != SbCNull;
      aTail = SbTail(aTail) )
{
  /* action*/;
}
```

SEE ALSO

SbFirst, SbNext, SbCCList, SbCCGroup

FUNCTION

SbId - get the symbol table index (STI) of a named object.

SYNOPSIS

```
SbTSTI  SbId(parent,name)
  SbTSTI parent;
  char*  name;
```

DESCRIPTION

SbId returns the STI of the object named ***name*** in the scope of the object ***parent***. Using objects of different classes for ***parent*** will return different kinds of objects:

If ***parent*** is **SbCPLocal**, **SbId** returns a top-level object in the local process.

If ***parent*** is a local object, **SbId** returns an attribute of that object.

If ***parent*** is a remote process (i.e. an instance of **SbCCRemote**), **SbId** returns a proxy to a top-level object in the remote process. The proxy is a local instance of the class **SbCCGRef**.

If ***parent*** is an instance of class **SbCCGRef**, **SbId** will return a proxy to an attribute of the remote object represented by ***parent***. The proxy is a local instance of the class **SbCCGRef**.

NOTE

If ***parent*** is structured in several levels, for instance if ***parent*** is a process containing structured objects, or ***parent*** is a nested object, ***name*** may be used to unpack the nesting. This is done with the "." operator. The "." operator have the same functionality as an extra call to **SbId**. In other words, the following lines give the same result:

```
id = SbId( processA, ".objectA.att1" );
id = SbId( SbId(processA,"objectA"), "att1" );
```

WARNING

If ***parent*** is a remote process (i.e. an instance of **SbCCRemote**) or a proxy to a remote object (i.e. an instance of class **SbCCGRef**) **SbId** sends a synchroneous remote function call to obtain the STI in the remote process. Consider using **SbIds** if many remote object ids are to be retrieved simultaneously to reduce the number of remote function calls.

EXAMPLE

```
/* Local object and local attribute */
SbTSTI myObject = SbId( SbCPLocal, "myObject");
SbTSTI myLevel  = SbId( myObject, "level" );

/* Remote process, proxy to remote object
   and proxy to attribute of remote object */
SbTSTI theProcess = SbId( SbCPLocal, "theProcessName" );
SbTSTI theObject  = SbId( theProcess, "myObject");
SbTSTI theLevel   = SbId( theObject, "level" );
```

SEE ALSO

SbIds, SbCCRemote, SbCCGRef, SbCreate, SbAdd

FUNCTION

SbIds - get the symbol table identifiers (STIs) of several named objects.

SYNOPSIS

```
SbTSTI  SbIds(parent,names)
  SbTSTI parent;
  char** names;
```

DESCRIPTION

SbIds works as **SbId**, but takes a NULL-terminated array of character pointers as input. **SbIds** should be used when you need the identifiers of many objects in a remote process.

RETURN VALUES

SbIds returns the symbol table index (STI) of a list of objects corresponding to the input names.

EXAMPLE

```
SbTSTI theGroup, myInt, myFloat, myDouble, myString;
char* names[5];
names[0] = "myInt";
names[1] = "myFloat";
names[2] = "myDouble";
names[3] = "myString";
names[4] = NULL;

theGroup  = SbIds(theProcess,names);
myInt     = SbHead(theGroup); theGroup = SbTail(theGroup);
myFloat   = SbHead(theGroup); theGroup = SbTail(theGroup);
myDouble  = SbHead(theGroup); theGroup = SbTail(theGroup);
myString  = SbHead(theGroup); theGroup = SbTail(theGroup);
```

SEE ALSO

SbId, SbCCList, SbCCGroup, SbHead, SbTail, SbFirst, SbNext

FUNCTION

SbInclude - include elements of one list into another list.

SbWithdraw - remove elements of one list from another list.

SYNOPSIS

```
int32  SbInclude(list,elements)
  SbTSTI list;
  SbTSTI elements;

int32  SbWithdraw(list,elements)
  SbTSTI list;
  SbTSTI elements;
```

DESCRIPTION

SbInclude and **SbWithdraw** both takes the STI's of two lists as parameters.

SbInclude adds each element of ***elements*** to ***list*** if they are not already present in ***list***. **SbWithdraw** removes all occurrences of the elements in ***elements*** from ***list***.

SbInclude and **SbWithdraw** treats lists as sets, they do not appreciate multiple occurrences of an element. These two functions should not be mixed together with **SbCrown** and **SbBehead**.

RETURN VALUES

SbInclude and **SbWithdraw** returns the number of elements added to or removed from list.

EXAMPLE

```
/* Create a group with two integers */
SbTSTI gint1, gint2, myIntGroup;
gInt1      = SbCreate(SbCPLocal,SbCCInteger,"gInt1",SbCNull,NULL);
gInt2      = SbCreate(SbCPLocal,SbCCInteger,"gInt2",SbCNull,NULL);
myIntGroup = SbCreate(SbCPLocal,SbCCGroup,"intGroup",SbCCInteger,NULL);
SbInclude( myIntGroup, SbList(SbCCGroup,gInt1,gInt2,SbCNull) );
```

SEE ALSO

SbCCList, SbCCGroup, SbHead, SbTail, SbFirst, SbNext

FUNCTION

SbIndex - get the symbol table index (STI) of an element of an array.

SYNOPSIS

```
SbTSTI  SbIndex(parent,index)
  SbTSTI  parent;
  int32   index;
```

DESCRIPTION

SbIndex returns the STI of element number ***index*** in the array-object ***parent***. An array object is an instance of a subclass of the SWBus class **SbCCArray**.

SbIndex works simlar to **SbId**.

EXAMPLE

```
/* Local array */
SbTSTI myArray   = SbId( SbCPLocal, "myArray" );
SbTSTI myElement = SbIndex( myArray, 5 );

/* Remote array, retrieve proxy to array and a proxy to an element */
SbTSTI theArray   = SbId( theProcess, "myArray");
SbTSTI theElement = SbIndex( theArray, 7 );
```

SEE ALSO

SbId, SbCCArray

FUNCTION

SbInit - initialize the Software Bus.

SYNOPSIS

```
SbTSTI  SbInit( const char* name, const char* host, const char* pClass)
```

DESCRIPTION

SbInit initializes the Software Bus symbol table, and should be called prior to any other SWBus function. The function connects to **control**, and registers itself as process ***name*** there. ***host*** specifies the host on which control should be running. If ***host*** is NULL, the value of the environment variable **CONTROLHOST** is used. Localhost is assumed if ***host*** is NULL and **CONTROLHOST** is unset. ***pClass*** is process class and can be used to identify different type of processes. The process class is stored in **control** and can be queried using the **controlutil** program or the SWBus function **SbQueryControl**.

SbInit also verifies that the current SWBus version is compatible with the version of **control**. If not so, **SbInit** fails.

RETURN VALUES

SbInit returns **SbCOK** if the initialization was successful, otherwise **SbCError**.

NOTE

SbInit requires a couple of specific entries in the services database. See the "Getting Started" chapter of the SWBus User's Guide for details.

EXAMPLE

```
/* Initialize as a SWBus process */
if ( SbInit( "myProgramName", NULL, "MyProgClass" ) == SbCError )
{
   SbTErrorCode ec = SbGetError();
   printf( "Initialization failed. Error: %s\n", ec.text );
   exit( 1 );
}
:
:
/* Leave the SWBus before terminating */
SbExit();
exit(0);
```

SEE ALSO

control, SbWinInit, SbXtInit

FUNCTION

SbInstances - return all instances of a class.

SYNOPSIS

```
SbTSTI  SbInstances(class)
  SbTSTI class;
```

DESCRIPTION

SbInstances returns all instances of ***class***. The instances has at some time been created by **SbCreate**, and ***class*** must be a dynamic class, a class with a possibly growing number of attributes.

RETURN VALUES

SbInstances returns a static object of class **SbCCGroup**. The group can be traversed using the functions **SbFirst/SbNext** or **SbHead/SbTail**.

EXAMPLE

```
SbTSTI aClass, iGroup, iElement;

for ( iGroup = SbInstances(aClass);
     (iElement=SbHead(iGroup)) != SbCNull;
     iGroup = SbTail(iGroup) )
{
  /* Perform actions on iElement
     iElement is an instance of aClass */
}
```

SEE ALSO

SbSub, SbCreate, SbHead, SbTail, SbCCGroup

FUNCTION

SbIsActive - check an end condition.

SYNOPSIS

```
int32  SbIsActive(s)
  int32 s;
```

DESCRIPTION

SbIsActive checks if a loop should still be executed. ***s*** is the condition, if **SbIsAcive(s)** returns false, the loop should terminate.

RETURN VALUES

SbIsActive returns true (nonzero) if the condition ***s*** is still active, otherwise false (zero).

NOTE

SbIsActive is only used in connection with user defined main loop functions. If the default SWBus main loop function is used, **SbIsActive** is not needed.

EXAMPLE

See an example at the manual page for **SbSetLoop**.

SEE ALSO

SbLoop, SbSetLoop, SbEndLoop

FUNCTION

SbIsConnected - Check whether a remote process is connected

SYNOPSIS

```
int32  SbIsConnected(o)
  SbTSTI o;
```

DESCRIPTION

SbIsConnected checks if its argument is connected.

RETURN VALUES

If *o* is an object of class **SbCCRemote** (or a subclass), **SbIsConnected** returns true if the remote process is connected.

If *o* is an object of class **SbCCGRef**, **SbIsConnected** returns true if the process to which the GRef refers is connected and the GRef itself is valid.

If *o* is an object of any other class, **SbIsConnected** returns false.

SEE ALSO

SbOpen, SbCCRemote, SbId, SbCCGRef

FUNCTION

SbLink - connect two data objects to enable dynamic updates.

SYNOPSIS

```
SbTSTI  SbLink(source,target,mode)
  SbTSTI  source;
  SbTSTI  target;
  SbTBP   mode;
```

DESCRIPTIr ON

SbLink connects the object ***source*** to the object ***target***. ***target*** subscribes to ***source***. When **SbPut, SbSend** or **SbSendToOne** is called for ***source***, the value is transferred to ***target***.

If ***target*** is a proxy to an object in a remote process, the actual transfer of data does not take place until **SbFlush** is called for the processes of ***source*** and ***target***.

The parameter ***mode*** is unused.

RETURN VALUES

SbLink returns ***source***, or **SbCError** if the operation failed.

EXAMPLE

```
/* Set cVar in local process to 314. The value will be
   automatically transferred to sVar in server process */

SbTSTI source, target, server;
int value;

server   = SbId( SbCPLocal, "server" );
source   = SbCreate( SbCPLocal, SbCCInteger, "cVar", SbCNull, NULL );
target   = SbCreate( server,    SbCCInteger, "sVar", SbCNull, NULL );
SbLink( source, target, 0 );

value = 314;
SbPut( source, &value );

SbFlush( SbCPLocal, server );
```

SEE ALSO

SbPut, SbSend, SbSendToOne, SbFlush, SbSubscriptions

FUNCTION

SbList - return a temporary list object.

SYNOPSIS

```
SbTSTI  SbList(theClass,...)
  SbTSTI theClass;
```

DESCRIPTION

SbList creates a temporary object of class ***theClass*** and inserts the listed elements. ***theClass*** must be **SbCCList** or a subclass of **SbCCList**, and remaining parameters must be of type **SbTSTI** terminated by **SbCNull**.

RETURN VALUES

The STI returned from **SbList** is static, so do not try to store it.

EXAMPLE

```
SbTSTI l, int1, int2, int3;
int1 = SbCreate( SbCPLocal, SbCCInteger, "int1", SbCNull, NULL );
int2 = SbCreate( SbCPLocal, SbCCInteger, "int2", SbCNull, NULL );
int3 = SbCreate( SbCPLocal, SbCCInteger, "int3", SbCNull, NULL );

/* Create a temporary group and put a value to all objects in the group */
int32 value = 32;
l = SbList( SbCCGroup, int1, int2, int3, SbCNull );
SbPut( l, &value );
```

SEE ALSO

SbObject

FUNCTION

SbLoop - enter the main loop function.

SYNOPSIS

```
SbTSTI  SbLoop();
```

DESCRIPTION

SbLoop enters the main loop function. A default main loop function is provided by the Software Bus, but a customized main loop function can be set up using **SbSetLoop**. The main loop function executes all pending periodic functions, receives and respond to data from other SWBus processes and executes all io-handlers having pending data.

RETURN VALUES

SbLoop returns **SbCOK**.

SEE ALSO

SbEndLoop, SbSetLoop, SbPeriodic, SbDispatch

FUNCTION

SbName - name of object.

SbClass - class of object.

SbData - data of object.

SbOffset - offset of attribute.

SYNOPSIS

```
char*  SbName(object)
  SbTSTI object;

SbTSTI  SbClass(object)
  SbTSTI object;

SbTAddr  SbData(object)
  SbTSTI object;

int32  SbOffset(theClass,name)
  SbTSTI theClass;
  char*  name;
```

DESCRIPTION

SbName returns the name of ***object***. This may be the class name set by **SbCreate**, the variable name set by **SbSub**, or the attribute name set by **SbAdd**. If ***object*** has no name, the Symbol Table Index (STI) in text form is returned.

SbClass returns the STI of the class of ***object.***

SbData returns a pointer to the data area of ***object***. The size and structure of the data area is defined by the class of ***object***.

SbOffset returns the offset of attribute ***name*** in ***theClass***.

EXAMPLE

```
/* Print properties of attribute "attrib1" of the object "object1" */
SbTSTI parent = SbId( SbCPLocal, "object1" );
SbTSTI child  = SbId( parent, "attrib1" );
printf( "%s is of class %s, its data is at offset %d from address 0x%x\n",
        SbName(child),
        SbName(SbClass(child)),
        SbOffset(parent,"attrib1"),
        SbData(parent) );
```

SEE ALSO

SbCreate, SbAdd, SbSub

FUNCTION

SbNonBlocking - set a communication channel in nonblocking mode.

SYNOPSIS

```
SbTSTI  SbNonBlocking(remProcess,fifoSize)
  SbTSTI remProcess;
  int32  fifoSize;
```

DESCRIPTION

SbNonBlocking sets the communication channel from the local process to ***remProcess*** in nonblocking mode. ***remProcess*** should be an instance of class **SbCCRemote** or a subclass. **SbNonblocking** allocates a buffer of ***fifoSize*** bytes that is used as a fifo circular buffer when sending data.

Using TCP/IP and sockets in blocking mode, which is the default in the SWBus, means that a process will be halted whenever the socket FIFO is full. The sender will have to wait for the receiver to read some bytes from the FIFO before continuing to send. As long as the receiver doesn't read from the FIFO, the sender will be halted. In some situations this can lead to a deadlock. Consider two processes sending data to each other simultaneously and the amount of data exceeding the FIFO size. None of them will ever read, as they are both busy sending, and a deadlock situation has occured.

In nonblocking mode (when **SbNonBlocking** is used), the sender will not be halted. The data to be transmitted will first be appended to the circular buffer, and the SWBus will then send as much data as possible without blocking. If it was unable to send the entire message, the SWBus will leave the rest in the circular buffer and return from send. In future iterations of the main loop, or when sending the next message, the SWBus will try to send more data from the circular buffer. The flow of control is handled entirely by the SWBus, and application developers don't need to know about it.

The probability of entering a deadlock situation is significantly reduced when nonblocking mode is used. However, there is a possibility that the internal circular buffer gets full. If that happens, the last operation that tried to send will fail, but no blocking will occur, and no other data will be lost.

SbNonBlocking should be called from within the **~notify** method of an instance of **SbCCRemote**, as the connection channel should be set nonblocking whenever it is established.

NOTE

SbNonBlocking sets the communication channel from the local process to a remote process nonblocking. If the channel from the remote process to the local process should also be nonblocking, it will be the responsibility of the remote process to call **SbNonBlocking** as well.

Application developers that are customising the main loop using **SbSetLoop**, should carefully study the example at the manual page for **SbSetLoop**. Note the function **SbDataWaiting**.

RETURN VALUES

SbNonBlocking returns **SbCError** if the operation failed.

EXAMPLE

```
/* SbNonBlocking should be called from within the
   ~notify method of your subclass of SbCCRemote */

SbTSTI notify( SbTSTI remProc, SbTSTI f, SbTSTI in, SbTSTI out )
{
   SbTType state = *(SbTType*)SbData(in);

   if ( state & SbCBPSConnected )
   {
   /* Call SbNonBlocking whenever the connection is established */
   SbNonBlocking( remProc, 1048576 );  /* 1MB circular bufer */
   }
   :
   :
   return out;
}
```

SEE ALSO

SbDataWaiting, SbOpen, SbCCRemote

FUNCTION

SbObject - return a temporary object.

SYNOPSIS

```
SbTSTI  SbObject(theClass,...)
  SbTSTI theClass;
```

DESCRIPTION

SbObject returns an object of class ***theClass***. The data area of this object is filled with the vararg parameters which are interpreted according to the attribute descriptions of ***theClass***.

RETURN VALUES

The STI returned from **SbObject** is static, so do not try to store it.

EXAMPLE

```
/* func has input parameter of type objectclass */
SbCall( func,
        SbObject(objectClass,3,4,param,f(3,14),9.34),
        SbCNull );
```

SEE ALSO

SbCreate

FUNCTION

SbOpen - open a connection to a remote process.

SYNOPSIS

```
SbTSTI  SbOpen(procClass,name,classNameRemote,mode)
  SbTSTI      procClass;
  const char* name;
  const char* classNameRemote;
  SbTBP       mode;
```

DESCRIPTION

SbOpen creates a new object of class ***procClass***. ***procClass*** should be **SbCCRemote** or a subclass thereof. The new object is the local representation of the remote process ***name***. ***name*** must correspond to the name used by the remote process when calling **SbInit**.

Similarly, the remote process will create an object to represent the local process. ***classNameRemote*** specifies the name of the class used by the remote process. ***classNameRemote*** should be "**~remote**" (the name of **SbCCRemote**) or the name of a subclass.

As an example, consider two processes P1 and P2, where P1 calls **SbOpen** to connect to P2. **SbOpen** would create an object of class ***procClass*** in P1. The object would be called P2. In P2, an object of class ***classNameRemote*** would be created. This object would be called P1. The ***name*** argument to **SbOpen** in this case should be P2.

mode specifies whether to wait for the connection to be established or to continue and accept the connection at a later stage. Legal values for ***mode*** are **SbCWait** and **SbCContinue**.

To get callbacks when the state of the connection changes, a method called **~notify** can be specified for ***procClass***. This method will be invoked when the connection is established, broken, terminated, etc, and the input parameter of the method is a bitmask describing the current state. The file $**BUSPATH**/examples/connect.c serves as a complete referance on how to set up a **~notify**-method.

If the connection to the remote process is broken, the SWBus will automaticaly start to periodically try to reconnect to the remote process. Note that it is the process that called **SbOpen** the first time (the client) that will try to reconnect, servers just wait for clients to connect.

To disconnect, **SbDelete** should be used for the object returned by **SbOpen**. When a connection is terminated this way, no automatic reconnect will occur. A disconnect can be initiated from the server as well as the client.

RETURN VALUES

SbOpen returns the Symbol Table Index (STI) of the remote process object. This identifier can be used to create objects or do inquiries in the remote process.

EXAMPLE

See the example program **$BUSPATH**/examples/connect.c

SEE ALSO

SbCCRemote, SbCreate, SbNonBlocking

FUNCTION

SbParent - get the parent object of an object

SYNOPSIS

```
SbTSTI SbParent(obj);
SbTSTI obj
```

DESCRIPTION

SbParent returns the parent object of the argument object ***obj***.

NOTE

SbParent only works for local objects.

SEE ALSO

SbCreate

FUNCTION

SbPeriodic - invoke periodic functions.

SbNextTimeout - gives the time to next periodic function.

SYNOPSIS

```
struct timeval*  SbPeriodic();
struct timeval*  SbNextTimeout();
```

DESCRIPTION

SbPeriodic invokes all periodic functions for which the specified time interval has elapsed since the last invocation. A periodic function is set up by creating an instance of class **SbCCPeriodic**. **SbPeriodic** evaluates and returns the time to wait before the next periodic function should be invoked.

If **SbPeriodic** is not able to invoke a periodic function in time, it will shorten down the interval to the next invocation in order to get back in sync. If the delay is larger than the specified interval, the invocation is lost.

SbNextTimeout returns the time to wait before **SbPeriodic** should be called.

RETURN VALUES

SbPeriodic and **SbNextTimeout** return the time that can pass before **SbPeriodic** should be called. If no periodic functions are waiting, **SbPeriodic** and **SbNextTimeout** return NULL. The return value from **SbPeriodic** and **SbNextTimeout** should be used as input value to select(2).

NOTE

SbPeriodic is mandatory in all user-defined main loop functions. If the default SWBus main loop function is used, **SbPeriodic** is not needed.

EXAMPLE

See an example at the manual page for **SbSetLoop**.

SEE ALSO

SbCCPeriodic, SbLoop, SbSetLoop, SbDispatch

FUNCTION

SbPut - place a value in a data object.

SbGet - fetch a value from a data object.

SYNOPSIS

```
SbTSTI  SbPut(object,value)
  SbTSTI  object;
  SbTAddr value;

SbTSTI  SbGet(object,value)
  SbTSTI  object;
  SbTaddr value;
```

DESCRIPTION

SbPut moves a value from the address pointed to by ***value*** into the SWBus object ***object***. If ***object*** is a proxy (an instance of **SbCCGRef**) to an object situated in a remote process the actual transfer of data does not take place before **SbFlush** is called. If any subscriptions are attached to ***object***, the value is also distributed to the subscribing objects.

SbGet moves a value from the SWBus object ***object*** to the address pointed to by ***value***. If ***object*** is a proxy (an instance of **SbCCGRef**) to an object in a remote process, a synchroneous remote function call is sent to obtain the value.

The number of bytes to be moved are determined by the class of ***object***.

RETURN VALUES

SbPut and **SbGet** returns ***object***, or **SbCError** if the operation failed.

EXAMPLE

```
SbTSTI intObj;
int32 val1, val2;

/* Place a value in intObj */
val1 = 45;
SbPut( intObj, &val1 );

/* Retrieve the value from intObj */
SbGet( intObj, &val2 );
```

SEE ALSO

SbCopy, SbLink, SbSend, SbSendToOne, SbFlush, SbCCGRef

FUNCTION

SbPrintDebugInfo - place a value in a data object.

SYNOPSIS

```
typedef uint32 SbTBP;

void  SbPrintDebugInfo( SbTBP mask);
```

DESCRIPTION

SbPrintDebugInfo prints certain information valuable for debugging to its standard output device. The information printed depends on the argument ***mask***. The argument is a bitmask and may be any combination of:

- **SbCBPDbgSTI** Information on number of STIs used
- **SbCBPDbgPool** Information on pools of deleted objects
- **SbCBPDbgRemote** Information on state of all instances of SbCCRemote

See **SbSetMessageOutputCb** for a description on how to redirect the output.

FUNCTION

SbQueryControl - Query contrrol for information about other connected processes.

SbQueryControlEx - Query contrrol for information prior to registering own name

SYNOPSIS

```
struct SbTControlQuery
{
   char   processName[128];
   int32  index;
   int32  isConnected;
   char   processClass[128];
   char   hostName[128];
   int32  pid;
   int32  major;
   int32  minor;
   int32  numberOfserverIDS;
   int32* serverIDS;
   int32  pingTimeout;
   char   latestPingTime[16];
};

SbTControlQuery* SbQueryControl( const char* className )
SbTControlQuery* SbQueryControlEx( const char* controlHost,
                                   const char* className )
```

DESCRIPTION

SbQueryControl extracts information from control about all connected processes of class ***className***. If ***className*** is NULLor an empty string, information about all connected processes is extracted.

The information is returned in an array of type ***SbTControlQuery***. The valid data of the returned array ends when a process with a zero-length name is found.

SbQueryControlEx is similar to ***SbQueryControl***, but the extra argument ***controlHost*** ensures that the function can be used prior to calling ***SbInit***. Thus, ***SbQueryControlEx*** can be used to retrieve information about registered processes prior to registering your own name.

EXAMPLE

```
SbTControlQuery* data = SbQueryControl( "SomeClass" );
while( strlen( data->processName ) > 0 )
{
   if ( data->isConnected )
   {
      // Handle information about connected processes here
   }
   else
   {
      // Handle information about disconnected processes here.
      // Note that only processName and index are
```

```
        // defined attributes for disconnected processes.
    }
    data++; //jump to next item
}
```

SEE ALSO

controlutil

FUNCTION

SbReadScript - create SWBus classes and objects based on a file.

SYNOPSIS

```
int32  SbReadScript( fileName )
  const char* fileName;
```

DESCRIPTION

SbReadScript creates SWBus classes and objects according to the specifications in the file ***fileName***. The syntax of the file is described by the examples below.

SbReadScript creates SWBus classes according to type definitions in ***fileName***. The function **SbSetClassCreatedCb** can be used to register a callback function to be invoked whenever a class has been created.

By default **SbReadScript** creates SWBus objects and allocates memory for their data areas acording to the variable definitions in *fileName*. The default behaviour can be overridden by the use of the SWBus function **SbSetCreateVarCb**. **SbSetCreateVarCb** registers a callback function to be invoked whenever a SWBus object is about to be created. It is then left for the programmer of the callback function to create the object.

The function **SbSetExistingVarReadCb** can be used to get a callback whenever **SbReadScript** reads a variable definition defining an existing variable.

RETURN VALUES

SbReadScript returns the number of syntax errors encountered.

EXAMPLES

Use of comments:

// Comment like in C++

/ * Comment like ordinary C comments */

Use of #include:

#include "dataTypes.pdat" // Include another file. Nested use of include is allowed

Definition and initialization of simple variables:

char c = 'a'; // single char

int16 a = 123; // 16-bit signed integers
short b = 123; // same
short int c = 123; // same

```
uint16              a = 456;      // 16-bit unsigned integers
unsigned short      b = 456;      // same
unsigned short int  c = 456;      // same

int32      a = 789;               // 32-bit signed integers
int        b = 789;               // same
long       c = 789;               // same
long int   d = 789;               // same
int32 anIntVar;                   // same

uint32              a = 789;      // 32-bit unsigned integers
unsigned int        b = 789;      // same
unsigned long       c = 789;      // same
unsigned long int   d = 789;      // same

float32  aFloatVar = 3.14;              // 32-bit float initialized
float      g = 3.14;              // same

float64  f = 6.28;                // 64-bit float initialized
double   g = 6.28;                // same
```

Definition and initialization of simple arrays:

```
char  name[32] = "test";          // character array initialized
int32  xx[5] = 56;                // all entries initialized to 56
int32  yy[5] = { 0, 1, 2, 3, 4 }; // all entries receive different values
int32  zz[5] = { 56, 6 };         // entry 0 and 1 initialized, rest is 0
```

Definition and initialization of simple pointers:

```
int32*     intPtr = &anIntVar;    // int pointer set to point to an integer variable
float32*  fff[5] = &aFloatVar;    // float pointer array, all point to aFloatVar
SbTString bb = "teststring";      // character pointer
char*      charptr;               // char pointer
void*      voidPtr;               // void pointer
```

Struct definitions:

```
struct SimpleStruct               // a simple struct
{
  float32 f = 3.14;               // provide a default value for this attribute
  int32*  i;
  char    name[32];
};

typedef struct                    // a nested struct
{
  int32 i[5];
  SimpleStruct ss;
} NestedStruct;
```

```
struct ListStruct;              // Forward declaration of a struct
struct  ListStruct              // a struct containg a pointer to the same struct
{
  NestedStruct data[3];
  struct ListStruct* next;
};
```

Definition and initialization of struct variables:

```
SimpleStruct ss1;                   // a simple struct variable
SimpleStruct ss2 = ss1;             // ss2 initialized by ss1
NestedStruct*  nsPtr = &ss1;        // pointer to struct
NestedStruct   nsArray[5];          // array of struct
NestedStruct*  nsPtrArray[5];       // array of pointer to struct
ListStruct     lsArray[5];          // array of struct
```

More variable definitions:

```
int32  anIntVar;
int32 *anIntPtr;
int32 $87HY3$;              // odd variable names must begin and end with a '$' character
SbTBuffer  a = { &nsArray, 300 };      // contains an address to a buffer and  size of the buffer
```

More variable initializations:

```
anIntPtr = &anIntVar;
anIntPtr = ss1.i;
anIntPtr = nsArray[2].i;
anFloatVar = nsArray[3].ss.f;
ss1.f = 2.18;
ss1.name = “ss1”;
lsArray[2].data[1].i[0] = 98;
lsArray[2].data[2].i = { 3, 13, 23, 33, 43 };
```

SEE ALSO

SbSetClassCreatedCb, SbSetCreateVarCb, SbSetExistingVarReadCb, SbCreate

FUNCTION

SbSend - distribute the value of a data object.

SbSendToOne - distribute the value of a data object to a specific process

SYNOPSIS

```
SbTSTI  SbSend(object)
  SbTSTI  object;

SbTSTI SbSendToOne( object, process )
  SbTSTI object;
  SbTSTI process;
```

DESCRIPTION

SbSend distributes the value of ***object*** to all objects, local or remote, that subscribes to this value. Subscription has been created by **SbLink**. If the subscribing objects are located in remote processes, the actual transfer of data does not take place until **SbFlush** is called. The functionality of **SbSend** is inherent in **SbPut**, so **SbSend** is only needed when the value of ***object*** has been updated directly.

SbSendToOne is a variant of **SbSend** which distributes the value of ***object*** only to objects in remote process ***process***.

RETURN VALUES

SbSend and **SbSendToOne** return ***object***, or **SbCError** if the operation failed.

EXAMPLE

```
int32  value;
SbTSTI object;

/* Create a SWBus object with data located in variable value */
object = SbCreate(SbCPLocal,SbCCInteger,"intObject",SbCNull,&value);

value = 345;
SbSend( object );
:
: /* Probably update several other objects the same way*/
:
SbFlush( SbCPLocal, SbCNull );
```

SEE ALSO

SbPut, SbLink, SbFlush

FUNCTION

SbSetClassCreatedCb - register a callback function for information about new SWBus classes.

SYNOPSIS

```
typedef void (*SbTClassCreatedCb) ( SbTSTI c );

void  SbSetClassCreatedCb( f )
  SbTClassCreatedCb f;
```

DESCRIPTION

SbSetClassCreatedCb registers a user-defined callback function ***f*** to be called whenever a new SWBus class has been created by **SbReadScript**.

The argument ***c*** of the callback function ***f*** is the Symbol Table Index (STI) of the created class.

RETURN VALUES

None.

EXAMPLE

```
void createClass( SbTSTI c )
{
   printf( "Class \"%s\" with id %d created \n", SbName(c), c );
}


/* Register the callback function */
SbSetClassCreatedCb( createClass );

/* Start reading from file */
SbReadScript( aFileName );
```

SEE ALSO

SbReadScript, SbSetCreateVarCb, SbSetExistingVarReadCb

FUNCTION

SbSetClientOnly -Ensure SWBus process is a pure client.

SYNOPSIS

```
SbTSTI  SbSetClientOnly()
```

DESCRIPTION

SbSetClientOnly should be called prior to SbInit. Invoking the function ensures that the local SWBus process is a pure client that no other SWBus process can connect to. It also prevents wasting system resources as the process will not occupy ports for non-used server sockets.

RETURN VALUES

SbSetClientOnly returns SbCOK on success.

FUNCTION

SbSetCreateVarCb - register a callback function for creating new SWBus objects.

SYNOPSIS

```
typedef SbTSTI (*SbTCreateVarCb) ( SbTSTI c, const char* n, void* d );

void  SbSetCreateVarCb( f )
  SbTCreateVarCb f;
```

DESCRIPTION

SbSetCreateVarCb registers a user-defined callback function ***f*** to be called whenever a new SWBus object is about to be created by **SbReadScript**.

When the callback function ***f*** is invoked, the argument ***c*** is the object's class, ***n*** is the name of the object and ***d*** is a pointer to a pre-allocated data area holding the object's initial value.

f should return the STI of the instansiated object or **SbCError** if the operation failed. If no object should be created, ***f*** should free the data pointed to by ***d*** and return **SbCOK**.

WARNING

If the pre-allocated data area is not used, or only used to copy inital values to another memory location, it must be freed before leaving the callback function.

EXAMPLE

```
SbTSTI myVars[1000]; /* Array of all my variables */
int numVars = 0;     /* Number of variables currently in array */

SbTSTI createVar( SbTSTI c, const char* n, void* d )
{
   SbTSTI o = SbCreate( SbCPLocal, c, n, SbCNull, d );
   if ( SbDBad(o) )
      return SbCError;
   else
   {
      myVars[numVars++] = o;
      return o;
   }
}

/* Register the callback function */
SbSetCreateVarCb( createVar );
/* Start reading from file */
SbReadScritp( fileName );
```

SEE ALSO

SbCreate, SbReadScript, SbSetExistingVarReadCb, SbSetClassCreatedCb

FUNCTION

SbSetError - Sets a value for the SbCError object

SbGetError - Extracts the value of the SbCError object

SYNOPSIS

```
#define SbCErrorTextLen 64
typedef struct
{
   int32        code;
   char         text[SbCErrorTextLen];
} SbTErrorCode;

SbTSTI        SbSetError( SbTErrorCode errorCode );
SbTErrorCode  SbGetError();
```

DESCRIPTION

SbSetError assigns the argument-value to the SWBus error object **SbCError**.

SbGetError extracts the value of the SWBus error object **SbCError**.

EXAMPLE

See an example at the SbCError page

SEE ALSO

SbCError

FUNCTION

SbSetExistingVarReadCb - register a callback function for definition of exising SWBus objects.

SYNOPSIS

```
typedef void (*SbTExistingVarReadCb) ( SbTSTI o );

void  SbSetExistingVarReadCb( f )
  SbTExistingVarReadCb f;
```

DESCRIPTION

SbSetExistingVarReadCb registers a user-defined callback function ***f*** to be called whenever an existing SWBus object is found by **SbReadScript**.

When the callback function ***f*** is invoked, the argument ***o*** is the object found.

EXAMPLE

```
SbTSTI existVar( SbTSTI o )
{
   printf( "SbReadScript found exising variable %s", SbName(o) );
}


/* Register the callback function */
SbSetExistingVarReadCb( existVar );
/* Start reading from file */
SbReadScritp( fileName );
```

SEE ALSO

SbReadScript, SbSetCreateVarCb, SbSetClassCreatedCb

FUNCTION

SbSetFdChangeCb - Register a callback function for changes in the SWBus fd_set

SYNOPSIS

```
typedef void (*SbTFdChangeCb) ( fd_set*, int32 );

void  SbSetFdChangeCb( f )
  SbTFdChangeCb f;
```

DESCRIPTION

SbSetFdChangeCb registers a user-defined callback function ***f*** to be called whenever the file descriptor set used by the SWBus changes. The file descriptor set changes when new connections are established, broken or terminated, when **SbCCIo** objects are created, suspended, restarted or deleted and when the connection to **control** is established, broken or reestablished.

When the callback function ***f*** is invoked, its first argument is a pointer to the current file descriptor set used by the SWBus, and the second argument is the size of that file descriptor set.

SbSetFdChangeCb can be used to avoid calling **SbFDGet** for every iteration of the main loop.

NOTE

SbSetFdChangeCb should be called prior to **SbInit**.

Applications using **SbWinInit** or **SbXtInit** should not call **SbSetFdChangeCb** directly, the functionality is included in the extended initialisation functions.

WARNING

SbSetFdChangeCb will not catch changes in file descriptor sets used by **SbCCIo** objects.

EXAMPLE

```
void fdch( fd_set fds, int32 setSize )
{
   int i;

   printf( "FdSet changed. Size: %d. Contents: {", setSize );
   for ( i = 0; i < setSize; i++ )
   {
      if ( FD_ISSET( i, &fds ) )
        printf( " %d", i );
   }
   printf( " }.\n" );
}
```

```
/* Register the callback function */
SbSetFdChangeCb( fdch );
```

SEE ALSO

SbFDGet, SbFDSize, SbSetLoop

FUNCTION

SbSetLoop - set up a user defined main loop function.

SYNOPSIS

```
void  SbSetLoop( void(*loop)(int32) );
```

DESCRIPTION

SbSetLoop replaces the default SWBus main loop function with the user-specified function ***loop***. The example below gives an outline of the structure of such functions.

WARNING

The functionality of the SWBus classes **SbCCIo** and **SbCCPeriodic** should be investigated before deciding to implement a user-specified main loop function.

EXAMPLE

```
/* The loop function */
void myLoop(sem)
   int32 sem;
{
   struct timeval* interval; /* The time to next loop */
   fd_set          fds;      /* The file descriptor set expected */
   int32           fdsize;   /* The size of this set */

   while ( SbIsActive(sem) )
   {
      interval = SbPeriodic(); /* execute periodic functions */
     if ( SbIsActive(sem) ) /* end condition may have been reached now */
      {
        fdsize = SbFDGet( SbCFDREAD, &fds ); /* Get SWBus file desc set */

       /* Any reevaluation of interval or file descriptor set goes here
           Suitable functions are SbFDOr, SbFDAnd and SbFDSize */

        if ( !SbDataWaiting() )
        {
           if ( select(fdsize,&fds,NULL,NULL,interval) == -1 )
           {
              printf("Error in select \n");
              return;
           }
        }
        SbDispatch( &fds, NULL, NULL );

        /* Put your own dispatch routine here */

      } /* end if SbIsActive */
   } /* end while */
} /* end myLoop */
```

```
/* Replace SWBus loop with user defined loop
   This should be done before calling SbLoop */
SbSetLoop( myLoop );
```

SEE ALSO

SbLoop, SbEndLoop, SbIsActive, SbDispatch, SbPeriodic, SbFDGet, SbFDAnd, SbFDOr, SbFDSize, SbCCPeriodic, SbCCIo

FUNCTION

SbSetMessageOutputCb - Register a callback function to set the output destination

SbSetMessageFilter - Set the wanted output

SbClearMessageFilter - Clear the wanted output

SYNOPSIS

```
typedef void (*SbTMessageOutputCb) (int32 messageClass,const char* str);

void  SbSetMessageOutputCb( SbTMessageOutputCb f )
void  SbSetMessageFilter( int32 messageClass )
void  SbClearMessageFilter( int32 messageClass )
```

DESCRIPTION

SbSetMessageOutputCb registers a user-defined callback function ***f*** to be called whenever a message is sent from the SWBus. **SbSetMessageFilter** sets the desired output message. **SbClearMessageFilter** clears the setting. Legal values for messageClass are:

- `SbCMMessage  /* Message  */`
- `SbCMInfo  /* Information  */`
- `SbCMError  /* Error  */`
- `SbCMWarning  /* Warning  */`
- `SbCMDebug  /* Debug  */`
- `SbCMFlow  /* Flow control  */`
- `SbCMMemory  /* Memory control  */`
- `SbCMCom  /* Comix related stuff */`
- `SbCMData  /* Data structuring  */`

EXAMPLE

```
void output( int msgClass, const char* str )
{
   if( msgClass == SbCMError )
      fprintf( stderr, "%s", str );
   else
      puts( str );
}

main()
{
   :
   SbSetMessageOutput( output );
```

FUNCTION

SbSetPeriodicChangeCb - Register a callback function for changes in periodics

SYNOPSIS

```
typedef void (*SbTPeriodicChangeCb) ();

void  SbSetPeriodicChangeCb( f )
  SbTPeriodicChangeCb f;
```

DESCRIPTION

SbSetPeriodicChangeCb registers a user-defined callback function ***f*** to be called whenever periodics are added or removed. Periodics are added or remove when objects of class **SbCCPeriodic** are created, deleted, suspended or restarted. Further, periodics are added right before a remote synchroneous function call and removed when the reply to the function call appears.

SbSetPeriodicChangeCb can be used to control the timout of non-SWBus loop functions, like the XtAppMainLoop (X11) or the GetMessage (Win32 API).

NOTE

Applications using **SbWinInit** or **SbXtInit** should not call **SbSetPeriodicChangeCb** directly, the functionality is included in the extended init functions.

SEE ALSO

SbWinInit, SbXtInit, SbCCPeriodic

FUNCTION

SbSub- create a new Software Bus class.

SYNOPSIS

```
SbTSTI  SbSub(theClass,name,parameter)
  SbTSTI theClass;
  char*  name;
  SbTSTI parameter;
```

DESCRIPTION

SbSub creates a new SWBus class called ***name***. The new class will be a subclass of ***theClass*** and will inherit all attributes and methods of ***theClass***. New attributes can be added using **SbAdd**.

parameter is only used when creating array classes. ***parameter*** should then be an instance of class **SbCCArrayParameter** and contain the base class and the number of elements.

Three SWBus classes are particularly well suited to subclass:

SbCCEmpty Base class for user-defined SWBus classes corresponding to C/C++ structs or classes.

SbCCArray Base class for user-defined array classes.

SbCCRemote Class representing a remote process. Must be subclassed to overload the default **~notify** method.

RETURN VALUES

SbSub returns the symbol table index (STI) of the new class, or **SbCError** if the operation failed.

EXAMPLE

```
/* Create a class for a user defined structure */
SbTSTI myClass;
myClass = SbSub( SbCCEmpty, "myClass", SbCNull );

/* Create an array class */
SbTArrayParameter paramData;
SbTSTI            param, arrayClass, arrayObj;
paramData.baseClass = SbCCInteger;
paramData.count     = NumberOfElements;
param = SbCreate(SbCPLocal,SbCCArrayParameter,NULL,SbCNull,&paramData);
arrayClass = SbSub( SbCCArray, "arrayClass", param );
SbDelete( param, SbCNull );
arrayObj = SbCreate( SbCPLocal, arrayClass, "theArray", SbCNull, NULL );
```

SEE ALSO

SbAdd, SbCreate, SbCCEmpty, SbCCArray, SbCCRemote

FUNCTION

SbSubscriptions - return all subscriptions for an object.

SYNOPSIS

```
SbTSTI  SbSubscriptions(object)
  SbTSTI object;
```

DESCRIPTION

SbSubscriptions returns a list of all objects subscribing for the the value of ***object***. Subscriptions are created by SbLink.

RETURN VALUES

SbSubscriptions returns a static object of type **SbCCGroup**. The group can be traversed using the functions **SbFirst/SbNext** or **SbHead/SbTail**.

EXAMPLE

```
SbTSTI anObject, sGroup, target;

for ( sGroup = SbSubscriptions( anObject );
     (target=SbHead(sGroup)) != SbCNull;
     sGroup = SbTail( sGroup );
{
  /* target is now an object subscribing for the value of anObject */
}
```

SEE ALSO

SbLink, SbUnlink

FUNCTION

SbUnlink - remove a subscription established by **SbLink**.

SYNOPSIS

```
SbTSTI  SbUnlink(source,target)
  SbTSTI source;
  SbTSTI target;
```

DESCRIPTION

SbUnlink removes subscriptions that has been established with **SbLink**. **SbUnlink** behaves differently depending on the value of ***target***. If ***target*** is

SbCNull	all subscriptions to ***source*** are removed
a process	all subscriptions to ***source*** in that process are removed
an object	the subscription to ***source*** for that specific object is removed.

RETURN VALUES

SbUnlink returns ***source***, or **SbCError** if the operation failed.

SEE ALSO

SbLink SbSubscriptions

FUNCTION

SbWinInit - initialize the SWBus in a Win32-based application

SbXtInit - initialize the SWBus in a Xt-based application

SYNOPSIS

```
SbTSTI  SbWinInit( HINSTANCE hInst, const char* name,
                   const char* host, const char* pClass )

SbTSTI  SbXtInit( XtAppContext appContext, const char* name,
                  const char* host, const char* pClass )
```

DESCRIPTION

SbWinInit is used to initialize the Software Bus symbol table when developing Windows GUI applications, and should be called prior to any other SWBus function. It replaces the SbInit function in Windows GUI applications. The argument hInst is a handle to current application instance. In win32 applications this instance handle is one of the arguments to WinMain. In an MFC application the instance handle can be obtain by calling the AFX function ::AfxGetInstanceHandle().

SbXtInit initializes the Software Bus symbol table when creating X Toolkit Intrinsics applications (Motif, Xt, etc.). The argument appContext is the current application context. The application context is the first output argument from the Xt function XtAppInitialize(3Xt). SbXtInit replaces the function SbInit() in Xt applications.

For both functions the ***name***, ***host*** and ***pClass*** arguments are passed directly on to **SbInit**.

RETURN VALUES

SbWinInit and **SbXtInit** returns **SbCOK** if the initialization was sucessful, otherwise **SbCError**.

SEE ALSO

SbInit

TYPE

SbTSTI - Symbol Table Index (STI).

SYNOPSIS

```
#include <swbus.h>
```

DESCRIPTION

SbTSTI is the generic type of all Software Bus objects. Such objects may be variables, classes, processes, functions, lists, groups, etc. All objects have some properties used to determine their functionality and their position in the SWBus data structure:

name The name of the object. This name can be used in external communication.

parent The parent of the object. For variables, this mean the process where they can be found, for attributes, the object they are a part of.

class The type of the object. The class contains information about the size of the object, and describes all attributes and methods of the object.

data The content of the objects. Some objects, e.g. functions, have no data at all.

Some predefined SWBus constants:

SbCNull A constant used to signal uninitialized STI, end of list, etc.

SbCError A constants signaling that an operation failed.

SbCOK A constant signaling that an operation finished successfully.

Some macros for STI testing:

SbDBad(sti) Returns true if the STI is **SbCError**, **SbCNull** or some less official SWBus constants.

SbDGood(sti) Returns true if the STI might be a valid SWBus object. Note that **SbCOK** is neither “good” nor “bad”.

SbDDead(sti) Returns true if the STI denotes an object that has been deleted.

Some predefined SWBus objects:

SbCPLocal An object representing the local process.

Some predefined SWBus classes:

SbCCChar The class of signed characters. The corresponding C data type is **char**.

SbCCInt8 The class of 8 bit signed integers. The corresponding C data type is **int8**.

SbCCUInt8 The class of 8 bit unsigned integers. The corresponding C data type is **uint8**.

SbCCInt16 The class of 16 bit signed integers. The corresponding C data type is **int16**.

SbCCUInt16 The class of 16 bit unsigned integers. The corresponding C data type is **uint16**.

SbCCInt32 The class of 32 bit signed integers. **SbCCInteger** is an alias for this class. The corresponding C data type is **int32**.

SbCCUInt32 The class of 32 bit unsigned integers. The corresponding C data type is **uint32**.

SbCCFloat32 The class of 32 bit signed floating point numbers. **SbCCFloat** is an alias for this class. The corresponding C data type is **float**.

SbCCFloat64 The class of 64 bit signed floating point numbers. **SbCCDouble** is an alias for this class. The corresponding C data type is **double**.

SbCCAddr The class for general pointers. Objects of this type are not transferred between processes. The corresponding C data type is **SbTAddr**.

SbCCString The class for pointers to NULL-terminated character strings (char*). When objects of this class is transferred between processes, the character string itself is transferred, memory holding any previous contents of the receiving object is freed, and new memory for the contents of the receiving object is allocated. The class has a destructor that frees the memory occupied by the string. The corresponding C data type is **SbTString**.

SbCCChar32Array The class for 32-byte character arrays.

SbCCChar64Array The class for 64-byte character arrays.

SbCCText The class for NULL-terminated arrays of NULL-terminated character strings. When objects of this class is transferred between processes, the character strings are transferred, the memory holding any previous contents of the receiving object is freed, and new memory is allocated for the contents of the receiving object. The corresponding C data type is **SbTText**.

SbCCBuffer The class for variable length byte buffers. The class has two attributes: **~address** (of class **SbCCAddr**) and **~length** (of class **SbCCInt32**). When objects of this class is transferred between processes, the byte buffer itself is transferred, memory holding any previous contents of the receiving object is freed, and new memory for the contents of the receiving object is allocated. The class has a destructor that frees the memory pointed to by the attribute **~address**. Note that the contents of the byte buffer is not marshalled as there is no way for the SWBus to know the structure of the data in the buffer. The corresponding C data type is **SbTBuffer**.

SbCCType The class for 32-bit bitmasks. The corresponding C data type is **int32**.

SbCCList The class for linked lists of SWBus objects. See the manual page for **SbCCList** for more details.

SbCCGroup The class for collection of SWBus objects with common operations. See the manual page for **SbCCGroup** for more details.

SbCCLRef The class for untyped references to other SWBus objects. It is the SWBus equivalent of an untyped pointer. The corresponding C data type is **SbTSTI**.

SbCCTRef The class for typed references to other SWBus objects. The class has two attributes: **~theClass** (of class **SbCCLRef**) and **~object** (of class **SbCCLRef**) The corresponding C data type is **SbTTRef**. **SbCCTRef** is rarely used compared to **SbCCLRef**.

SbCCGRef The class of references to objects in other processes. See the manual page for **SbCCGRef** for more details.

SbCCRemote The class of remote processes. See the manual page for **SbCCRemote** for more details.

SbCCEmpty The base class for all user-defined SWBus classes matching a C/C++ struct.

SbCCArray The base class for all user-defined array classes. See the manual page for **SbCCArray** for more details.

SbCCPeriodic The class for objects capable of calling a function at regular intervals. See the manual page of **SbCCPeriodic** for more details.

SbCCIo The class for objects capable of calling a function whenever some input on a file descriptor occurs. See the manual page for **SbCCIo** for more details.

SbCCKeyboard The class for objects capable of calling a function whenever keystrokes are pressed in a console window. See the manual page for **SbCCKeyboard** for more details.

OBJECT

SbCError - An object signalling that an operation failed

SYNOPSIS

```
extern SbTSTI  SbCError;

#define SbCErrorTextLen 64
typedef struct
{
   int32        code;
   char         text[SbCErrorTextLen];
} SbTErrorCode;
```

DESCRIPTION

SbCError is a SWBus object signalling that an operation failed. The object is an instance of the class **SbCCErrorCode**, and its data is formated according to the struct **SbTErrorCode**.

Error codes with values less than **SbCMinUserErrorCode** are reserved for SWBus internally, application programmers are free to use any other value.

EXAMPLE

```
/* Server side: */
SbTSTI theFunc(SbTSTI o, SbTSTI f, SbTSTI in, SbTSTI out)
{
   : /* Here goes the server function */
   :
   if ( some failure occurred in the function )
   {
      SbTErrorCode ec;
      ec.code = 1000;
      strcpy( ec.text, Description of error );
      return SbSetError( ec );
   }
   else
      return out; /* Normal result */
 }

/* Client side: */
result = SbCall( f, in, out );
if ( result == SbCError )
{
   SbTErrorCode ec = SbGetError();
   printf( Error: %d %s\n, ec.code, ec.text );
}
```

SEE ALSO

SbSetError, SbGetError

CLASS

SbCCArray - the class of continuous tables of one-class objects

SbCCArrayParameter -constructor parameter class for **SbCCArray**.

SYNOPSIS

```
extern SbTSTI SbCCArray;
extern SbTSTI SbCCArrayParameter;
```

DESCRIPTION

The SWBus class **SbCCArray** is a base class for array classes. **SbSub** should be used to create subclasses of **SbCCArray**; instances of these subclasses will be array objects.

A subclass of **SbCCArray** specifies the class of the array elements and the number of elements in the array. This information is specified by the user through the constructor parameter when the subclass is defined. The constructor parameter is an object of class **SbCCArrayParameter**. The C data type corresponding to **SbCCArrayParameter** is **SbTArrayParameter**.

Attributes of SbCCArrayParameter:

~baseClass STI of class of array elements

~count number of elements in the array

SbCCArray is a subclass of **SbCCRoot**. **SbCCArrayParameter** is a subclass of **SbCCEmpty**.

EXAMPLE

```
SbTArrayParameter paramData;
SbTSTI            param, arrayClass, arrayObj;

/* Create an array class */
paramData.baseClass = SbCCInteger;
paramData.count     = aNumber;
param = SbCreate(SbCPLocal,SbCCArrayParameter,NULL,SbCNull,&paramData);
arrayClass = SbSub( SbCCArray, "arrayClass", param );
SbDelete( param, SbCNull );

/* Create an instance of the class, this is the actual array object */
arrayObj = SbCreate( SbCPLocal, arrayClass, "theArray", SbCNull, NULL );
```

SEE ALSO

SbSub, SbCreate, SbCCEmpty, SbTSTI

CLASS

SbCCGRef - the class representing an object in another process.

SYNOPSIS

```
extern SbTSTI SbCCGRef;
```

DESCRIPTION

An object of class **SbCCGRef** is a local representation for a SWBus object in another process. In SWBus documentation, this is also known as a proxy. An object of class **SbCCGRef** is a SWBus object in the local process. One can think of it as a pointer to an object in another process.

Objects of class **SbCCGref** are normally automatically created by the SWBus as an outcome of asking for object references in a remote process. Typically, such objects are created by **SbId**, **SbIds**, or **SbIndex**. One might also use **SbCreate** to create objects in remote processes, and in such a case **SbCreate** returns an instance of **SbCCGRef**.

The corresponding C data type is **SbTGRef**.

Attributes of class SbCCGRef:

~process The STI of the process in which the referred object resides. This is the STI returned by **SbOpen** and it is an instance of class **SbCCRemote** or a subclass.

~theClass The STI of the class of the referred object. Actually this is the STI of a local class identical to the class of the referred object. To enable exchange of complex user-defined data types, the classes must be known (and be identical) at both sides.

~object The STI of the referred object. Note that this is the STI as seen from the process where the object actually resides, it has no valid interpretation in the local process.

SEE ALSO

SbTSTI, SbCCRemote, SbOpen, SbId, SbIds, SbIndex, SbCreate

CLASS

SbCCGroup - the class of collections of objects with common operations.

SYNOPSIS

```
extern SbTSTI SbCCGroup;
```

DESCRIPTION

The SWBus class **SbCCGroup** is a generic class containing collections of SWBus objects that may be accessed in one common operation. Calling a group of functions will result in all of the functions being called. Assigning a value to a group of variables will result in all of the variables getting the same value. A group of functions returning values results in a group of return values.

SbCCGroup is a subclass of **SbCCList**, so list operations for inserting, removing and traversing, can be used for groups as well.

Other SWBus functions that works well for group objects:

SbCall returns a group of return values.

SbId return a group of attributes.

SbPut puts values in each element.

SbFlush flushes values from a group of processes.

SEE ALSO

SbCCList

CLASS

SbCCIo - the class of io-handler objects.

SbCCIoParameter - constructor parameter class for **SbCCIo**.

SYNOPSIS

```
extern SbTSTI SbCCIo;
extern SbTSTI SbCCIoParameter;
```

DESCRIPTION

The SWBus class **SbCCIo** is used for creating functions to be called as a response to activity on an associated file descriptor. The class **SbCCIoParameter** is used to pass initial values to the constructor of **SbCCIo**. The input handler function must be a standard SWBus function.

SbCCIo and **SbCCIoParameter** are both subclasses of **SbCCEmpty**. The corresponding C data types are **SbTIo** and **SbTIoParameter**.

Attributes of class SbCCIoParameter:

~fdc an int32 specifying the file descriptor condition. Legal values for fdc are **SbCFDREAD**.

~fds a pointer to a file descriptor set specifying one or more file descriptors to listen for.

~func the input handler function. The function must have the standard SWBus function signature, **SbTCode**.

~inParam STI of an object used as input parameter to the input handler function.

~outParam STI of an object used as output parameter from the input handler function.

Methods of class SbCCIo:

~suspend Temporarily stops the input handler from being called. The method has no input or output parameters. Returns **SbCError** if invoked for an already suspended object.

~restart Reactivates a suspended input handler. The method has no input or output parameters. Returns **SbCError** if invoked for an object that is not suspended.

EXAMPLE

```
/* This example sets up an input handler for keyboard input.
   Note that this example will not work on Windows platforms,
   as no file descriptor can be obtained for stdin */

/* Define the input handler function */
SbTSTI myHandler( SbTSTI o, SbTSTI f, SbTSTI in, SbTSTI out )
```

```
{
   /* "o" is the STI of the io object */
   /* "f"  is SbCNull */
   /* "in" is the input parameter */
   /* "out" is the output parameter */

   char input[128];
   scanf( "%s", input );
   /* Take some action depending on the input */
   :
   return out;
}


/* Create a file descriptor set for keyboard input */
fd_set*        fds;
fds = malloc( fd_set) ;
FD_ZERO( fds );
FD_SET( 0, fds ); /* 0 is file descriptor for stdin on Unix */

/* Create constructor parameter */
SbTIoParameter cpData;
SbTSTI cp;
cpData.fdc      = SbCFDREAD;
cpData.fds      = fds;
cpData.func     = myHandler;
cpData.inParam  = SbCNull; /* No input paramter */
cpData.outParam = SbCNull; /* No output parameter */
cp = SbCreate( SbCPLocal, SbCCIoParameter, NULL, SbCNull, &cpData );

/* Create io-object and delete constructor parameter */
SbTSTI myIo = SbCreate( SbCPLocal, SbCCIo, "myIo",cp, NULL );
SbDelete( cp, SbCNull );

/* Finally enter the main loop */
SbLoop();
```

SEE ALSO

SbTSTI, SbDispatch

CLASS

SbCCKeyboard - the class of keyboard-handler objects.

SbCCKeyboardParameter - constructor parameter class for **SbCCKeyboard**.

SYNOPSIS

```
extern SbTSTI SbCCKeyboard;
extern SbTSTI SbCCKeyboardParameter;
```

DESCRIPTION

The SWBus class **SbCCKeyboard** is used for creating a keyboard input function to respond to keystrokes in a console window. It can be initialized to respond to single keystrokes and/or input lines. An input line in this context is zero or several keystrokes followed by carriage return. Note that only one instance of a **SbCCKeyboard** class can be active simultaneously in an application and that the class is only available for console applications.

The class **SbCCKeyboardParameter** is used to pass initial values to the constructor of **SbCCKeyboard**. The input handler functions must be standard SWBus functions.

SbCCKeyboard is a subclass of **SbCCIO**. **SbCCKeyboardParameter** is a subclass of **SbCCIOParameter**. The corresponding C data types are **SbTKeyboard** and **SbTKeyboardParameter**.

On Windows platforms the SWBus class **SbCCKeyboard** is only available when compiling the application with multi threaded options (MD or MT).

Attributes of class SbCCKeyboardParameter:

~funcChar the single keystroke input handler function. The function must have the standard SWBus function signature, **SbTCode**. This attribute can be set to **SbCNull**.

~funcLine the input line handler function. This function is called when the carriage return key is pressed in the console window. The function must have the standard SWBus function signature, **SbTCode**. This attribute can be set to **SbCNull**.

~length length of the input buffer used when the attribute **~funcLine** is initialized to a function handler. This attribute specifies the maximum length of a single input line. The **SbCCKeyboard** class will beep if the number of keyboard strokes exceeds this limit.

~inParam STI of an object used as input parameter to the keyboard handler function.

~outParam STI of an object used as output parameter from the keyboard handler function.

Attributes of class SbCCKeyboard:

~character The last character pressed on the keyboard. This attribute is only available in the **~funcChar** keyboard handler.

~buffer A null terminated character array. All keystrokes will be put into this buffer as long as the carriage return is not pressed. It handles backspace and tabulators. This attribute is only available in the **~funcLine** keyboard handler.

Methods of class SbCCKeyboard:

~suspend Temporarily stops the keyboard handler from being called. The method has no input or output parameters. Returns **SbCError** if invoked for an already suspended object. If several instances of the **SbCCKeyboard** class is present in an application the active keyboard instance must be suspended before another instance can be activated.

~restart Reactivates a suspended keyboard handler. The method has no input or output parameters. Returns **SbCError** if invoked for an object that is not suspended or another **SbCCKeyboard** instance is active.

EXAMPLE

```
/* This example sets up a keyboard handler */

/* Define the input line handler function */
SbTSTI lineHandler( SbTSTI o, SbTSTI f, SbTSTI in, SbTSTI out )
{
   /* "o" is the STI of the keyboard object */
   /* "f"  is SbCNull */
   /* "in" is the input parameter */
   /* "out" is the output parameter */
   char*        input;
   SbTKeyboard* kbd = (SbTKeyboard *)SbData( o )
                      /* Take some action depending on the input */
   input = kbd->buffer;
   ...
   return out;
}

/* Create the keyboard input handler */
SbTKeyboardParameter kb;
SbTSTI               kbParam

/* Create constructor parameter */
kb.inParam  = SbCNull;
kb.outParam = SbCNull;
kb.funcChar = SbCNull;
kb.funcLine = lineHandler;     /* Input line keyboard handler */
kb.length   = 1000;            /* Buffer length */

kbParam = SbCreate( SbCPLocal, SbCCKeyboardParameter, 0, SbCNull, &kb );

/* Create keyboard handler and delete the constructor parameter.*/
SbCreate( SbCPLocal, SbCCKeyboard, 0, kbParam, NULL );
SbDelete( kbParam, SbCNull );
```

```
/* Finally enter the main loop */
SbLoop();
```

SEE ALSO

SbTSTI, SbDispatch,SbCCIo

CLASS

SbCCList - class for a linked list of SWBus objects.

SYNOPSIS

```
extern SbTSTI SbCCList
```

DESCRIPTION

SbCCList is a generic class for a linked lists of SWBus objects. **SbCCList** is a subclass of **SbCCRoot**.

SWBus list operation functions:

SbFirst	Start list traversal.
SbNext	Continue a list traversal.
SbHead	Get the first element of a list.
SbTail	Get the sublist starting at the second element of a list.
SbCrown	Insert an element into a list.
SbBehead	Remove the first element of a list.
SbInclude	Include elements of one list into another list
SbWithdraw	Withdraw elements of one list from another list.

SEE ALSO

SbTSTI, SbFirst, SbNext, SbHead, SbTail, SbCrown, SbBehead, SbInclude, SbWithdraw

CLASS

SbCCPeriodic - the class of periodic-handler objects.

SbCCPeriodicParameter - constructor parameter class for **SbCCPeriodic**.

SYNOPSIS

```
extern SbTSTI SbCCPeriodic;
extern SbTSTI SbCCPeriodicParameter;
```

DESCRIPTION

The SWBus class **SbCCPeriodic** is used for creating handler functions to be called regularly from the main loop. The class **SbCCPeriodicParameter** is used to pass initial values to the constructor of **SbCCPeriodic**. The handler function must be a standard SWBus function.

SbCCPeriodic and **SbCCPeriodicParameter** are subclasses of **SbCCEmpty**. The corresponding C data types are **SbTPeriodic** and **SbTPeriodicParameter**.

Attributes for SbCCPeriodicParameter:

~interval An int32 specifying the interval (in milliseconds) between each call to the handler function.

~func The function to be called at the rate specified by **~interval**.

~count An int32 specifying the maximum number of times to call the handler function. If **~count** is set to -1, there is no such limit.

~period An int32 specifying the total time (in milliseconds) to wait before calling the **~term** function. If ~**period** is set to -1, the **~term** function will never be called.

~term Function called when the time specified by **~period** expires. If **~term** is set to **NULL**, the **~suspend** method of the object will be called.

~inParam STI of input parameter to handler function.

~outParam STI of output parameter to handler function.

Methods for class periodic:

~suspend Temporarily stops the handler from being called from the main loop. The method has no input or output parameters. Returns **SbCError** if invoked for an already suspended object.

~restart Reactivates a suspended handler. The method can take an integer object as input parameter and use it's value as a new value for interval. The value is forced to be greater than zero. If no integer object is specified, the old interval value is used. Returns **SbCError** if invoked for an object that is not suspended.

~setInterval Changes the interval between subsequent calls to the handler. The method takes an integer object as input parameter, and sets the interval to the associated value. The value is forced to be greater than zero. The new value will not be used before the timeout of the current interval.

EXAMPLE

```
/* This example sets up a periodic function to
   increment the value of a SWBus object */

/* Define the handler function to be called from the main loop */
SbTSTI myHandler( SbTSTI o, SbTSTI f, SbTSTI in, SbTSTI out )
{
  /* "o"   is the STI of the periodic object */
  /* "f"   is SbCNull */
  /* "in"  is the input parameter */
  /* "out" is the output parameter */

  int value;
  SbGet( in, &value );
  printf( "Incrementing value. New value is %d\n", ++value );
  SbPut( out, &value );
  return out;
}


/* Create an object containg the value to be incremented */
SbTSTI theObject;
theObject = SbCreate(SbCPLocal,SbCCInteger,"theObject",SbCNull,NULL);

/* Create constructor parameter */
SbTPeriodicParameter cpData;
SbTSTI               cp;
cpData.interval = 1000; /* once every second */
cpData.func     = myHandler;
cpData.count    = -1; /* No limit */
cpData.period   = 3600000; /* go for an hour */
cpData.term     = NULL;
cpData.inParam  = theObject;
cpData.outParam = theObject;
cp = SbCreate(SbCPLocal,SbCCPeriodicParameter,NULL,SbCNull,&cpData);

/* Create periodic-object and delete constructor parameter */
SbTSTI myPeriodic;
myPeriodic = SbCreate( SbCPLocal, SbCCPeriodic, "myPeriodic", cp, NULL );
SbDelete( cp, SbCNull );

/* Finally enter the mainloop */
SbLoop();
```

SEE ALSO

SbTSTI, SbPeriodic

CLASS

SbCCRemote - the class of objects representing remote processes.

SbCCRemoteParameter - constructor parameter class for **SbCCRemote**.

SYNOPSIS

```
extern SbTSTI SbCCRemote;
extern SbTSTI SbCCRemoteParameter;
```

DESCRIPTION

An instance of class **SbCCRemote** is the local notion of a remote process. Instances of **SbCCRemote** are created by **SbOpen**, by **SbCreate** or as a response to **SbOpen**/**SbCreate** in a remote process.

SbCCRemote can be subclassed to enable overloading of methods.

SbCCRemote is a subclass of **SbCCProcess**. **SbCCRemoteParameter** is a subclass of **SbCCEmpty**. The corresponding C data types are **SbTRemote** and **SbTRemoteParameter**.

Attributes of SbCCRemoteParameter:

~exchange An int32 describing how to behave when connecting to the other process. Legal values are **SbCWait** and **SbCContinue**. If **SbCWait** is specified, **SbOpen**/**SbCreate** will wait in an internal loop until the connection is established. If **SbCContinue** is specified, **SbOpen**/**SbCreate** will return and the connection may be established at a later time.

~transfer An int32 describing how to behave when flushing data. Legal values are **SbCSync** and **SbCAsync**. If **SbCSync** is specified, the remote process will automatically flush its transferbuffer when receiving a flush. If **SbCAsync** is specified, the remote process will just accept the data. **SbOpen** always uses **SbCAsync**.

~bufSize An int32 specifying the size of the data transfer buffer. **SbOpen** always uses 65536 (64KB).

~className The classname to be used by the remote process when creating the object to represent of the local process.

~inParam STI of an object to be used as constructor parameter for the object created in the remote process.

~hisHost An int32 not used.

~hisPort An int32 not used.

~myHost An int32 not used.

~myPort An int32 not used.

~timeout An int32 specifying the timeout (in milliseconds) to be used for synchroneous function calls to the remote process. **SbOpen** uses the value of **SbVTimeout** for this attribute.

Attributes of SbCCRemote:

~state This attributes contains the current state of the connection to the remote process. Its values are equal to the value sent as input parameter to the **~notify** method. The value can be extracted as follows:

```
SbTType state = ((SbTRemote*)SbData(remProcObj))->state;
```

~timeout This attribute contains the timeout value used when calling remote functions synchronously. The value is initially set to the value of the **~timeout** attribute of the constructor parameter. The value can be modified like this:

```
int32 aValue = 5000; /* 5 seconds */
((SbTRemote*)SbData(remProcObj))->timeout = aValue;
```

~breakConfig The attribute is a bit-mask, and if a certain bit is set, SWBus will break the channel if the corresponding error occurs. By default all the defined bits are set, so the default attribute value is **SbCBPBreakOnComFailure | SbCBPBreakOnNonblockingFull | SbCBPBreakOnTimeout**.

Methods of SbCCRemote:

~construct Is invoked when the object is created. Overload this method if you need some actions when this happens. The input parameter to **~construct** for **SbCCRemote** is an instance of **SbCCRemoteParamter**. If you subclass **SbCCRemote** and want arguments to **~construct**, **SbCCRemoteParamter** should be subclassed as well, and the subclass should be used as input parameter class for **~construct**.

~notify Is invoked whenever the state of the connection to the remote process changes. Overload this method if you need a callback when this happens. The input parameter to **~notify** is an object of class **SbCCType**, an int32 to be interpreted as a bit mask. The example program **$BUSPATH**/examples/connect.c serves as a complete reference on how to set up a **~notify** method. See also the description of the bits in the bitmask below.

~update Is invoked when local variables are updated by the remote process. Overload this method if you need to perform some actions when values are received. **~update** takes an input argument of type **SbCCUpdateParameter** containg a list of objects that have been updated.

Description of the bitmask in ~notify:

The input parameter to **~notify** is an instance of **SbCCType**, an int32 to be interpreted as a bit mask. Here is a description of each bit.

SbCBPSIncompVersion	Indicates that the SWBus version of the remote process is incompatible with the version of the local process. If set together with **SbCBPSDisconnect** and **SbCBPSRequest** it was the remote process that discovered the incompatibility. Otherwise the incompatibility was discovered by the local process.
SbCBPSConnecting	Indicates that the local process has initiated a connection to the remote process. The local proces will periodically try to connect to the remote process at intervals specified by the environment variable **BUSRETRY**. The connection is not yet established.
SbCBPSWaiting	Indicates that the local process waits in an internal loop for the connection to be established. Always used together with **SbCBPSConnecting**. This bit is set if **SbCWait** is used when calling **SbOpen**.
SbCBPSConnected	Indicates that the connection to the remote process is established. If the connection was established on the request of the remote process, **SbCBPSRequest** is set additionally.
SbCBPSRequest	Adds information to **SbCBPSConnected** and **SbCBPSDisconnect**.
SbCBPSBroken	Indicates that the connection is broken. If the local process initiated the connection, it will automatically try to reconnect to the remote process, and in such cases **SbCBPSConnecting** is set additionally.
SbCBPSDisconnect	Indicates that the connection has been terminated gracefully. If the connection was terminated by the remote process, **SbCBPSRequest** is set additionally.
SbCBPSTimeout	Indicates that the local process has given up establishing a connection to the remote process, because the period specified by the environment variable **BUSCONNECT** has elapsed.

Description of the argument to ~update:

The input parameter to **~update** is an instance of class **SbCCUpdateParameter**. The class has three attributes:

~count	An int32 specifying the number of variables updated.
~max	An int32 specifying the maximum number of variables that can be put in the list without reallocating memory. Applications should not use this attribute.
~sti	An array of object ids (SbTSTI). The first **~count** elements in the array contain the ids for the objects updated.

The struct **SbTUpdateParameter** matches the memory layout of the class **SbCCUpdateParameter**.

EXAMPLE

See the example program **$BUSPATH**/examples/connect.c

SEE ALSO

SbOpen, SbTSTI, SbCCGRef, SbNonBlocking

MAIL GROUPS

softbus - The Software Bus Development Team

softbus-ig - The Software Bus Interest Group

SYNOPSIS

```
softbus@hrp.no
softbus-ig@hrp.no
```

DESCRIPTION

softbus-ig@hrp.no is a forum for general discussions about the Software Bus. Messages of general interest to SWBus users will be sent to this group. Everyone that develops software using the Software Bus is encouraged to join the interest group, to keep up-to-date with future plans and new releases, and to discuss possible new features, ideas and general usage. Send a request by email to our support address (**softbus@hrp.no**) to join.

softbus@hrp.no is the Software Bus Development Team and also our support address. Use this address for messages to the development team not concerning the entire interest group. Requests and bug reports should also be mailed to this address.

NAME

control - a server for service management.

SYNOPSIS

```
control [ -? | -p portNumber | -d debugOpts | -f fixedServicesOpts ]
```

DESCRIPTION

control is a server that is used to register and unregister services. A service is a uniquely named port on a host. **Software Bus** processes register themselves with **control** in order to locate other processes. Users do not need to be aware of **control's** existence as it should normally be running as a service. A utility program called **controlutil** can be used to query **control's** state and send administrative messages to **control**.

OPTIONS

-? Writes a summary of the **control** options to ***stdout***.

-p Specifies the port number **control** should use when listening for clients. Using this option will override environment variable CONTROLPORT.

-d Specifies that control should write debug output to ***stdout*** for certain conditions. **<debugOpts>** specifies a combination of conditions. Available conditions are: ALL, NONE, CONNECT, ARCH, REGISTER, COMIX, HOSTACCEPT, PING. Conditions can be combined using a non-whitespace character, for instance CONNECT|PING.

-f **fixedServicesOpts** should be like 10;primServ=1;secServ=2;
First number is number of reserved processes. Then follows a semicolon-separated list of process names and their corresponding process number. All process numbers must be in the range 1..NumberOfReservedProcesses.

NOTE

control requires a specific entry in the services database in order to start properly. See instructions on how to install the **SWBus** in the SWBus User's Guide for details.

ENVIRONMENT

The host on which **control** is running can be specified using the environment variable **CONTROLHOST**.

The port **control** will use when listening for clients can be specified using the environment variable **CONTROLPORT**.

The location of the **control** executable can be specified using the environment variable **CONTROLPATH**, if it is not located in ***$BUSPATH/bin/$ARCH***. This enables processes to start **control** automatically on the local host if one is not already running and **CONTROLHOST** is not set to a remote host.

See the SWBus User's Guide for a complete list of environment variables.

FILES

```
$BUSPATH/bin/$ARCH/control
```

SEE ALSO

controlutil, **SWBus**

NAME

controlutil - a utility program for sending messages to **control**.

SYNOPSIS

controlutil [-? | -d -h ***hostName*** **-D** ***debugOpts*** **-k -q -Q -c** ***className*** **-V -u -i** ***serviceName*** **-t** ***timeout*** **]**

DESCRIPTION

controlutil is used to extract information from **control**, to unregister all services or to tell **control** to stop running.

OPTIONS

-? Write a summary of the **controlutil** options to ***stdout***.

-d Enable debug output from **controlutil** to ***stdout***.

-h Specify the name of the host where **control** is running. If no ***hostname*** is specified, the value of the environment variable **CONTROLHOST** is used if set, otherwise *localhost* is assumed.

-D Set conditions for **control's** output to stdout. Available conditions are listed for command line argument -d to control.

-k Tell **control** to stop running.

-q Write information to ***stdout***. The output includes hostname and version number of **control** and a list of all registered services with service ids. Services that are connected to **control** when **controlutil** extracts its information are specified with processname, process class, hostname and processId. Formerly registered services that has exited without unregistering are listed as disconnected. See below for an example output.

-Q Like the -q option, but more informaiton about each connected process is listed. The additional information includes IP-addresses, SWBus version, and ping information.

-c Like the -q option, but only information about processes of class ***className*** is listed.

-V Write the hostname and version number of **control** to ***stdout***.

-u Unregister all services. Using this option will fail if one or more services are currently connected to **control**.

-i Get the unique identifier of the named service.

-t Timeout in milliseconds used when connecting to control. Default=500ms.

EXAMPLES

```
$ controlutil -h castor -q
```

```
controlutil (SWBus Release 4.1 January 2004)

 Process information from control (4.1) at castor:
   1: performanceServer  disconnected
   2: performanceClient  disconnected
   3: simpleServer  class: Unknown  host: castor,  pid: 11513
   4: simpleClient  class: Unknown  host: castor,  pid: 11520
There are 4 processname(s) registered, 2 of these are currently connected.
```

ENVIRONMENT

The default host for **control** can be set using the environment variable **CONTROLHOST**. See the SWBus User's Guide for a complete list of environment variables.

FILES

```
$BUSPATH/bin/$ARCH/controlutil
```

SEE ALSO

control

AUTHOR

controlutil was developed by Håkon Jokstad and Michael Louka at IFE.

Index