SCOPE /SDK

Version 4.0

Chapter 1: *SCOPE Paradigm*

CreamWare Datentechnik GmbH
Wilhelm-Ostwald-Strasse 0/K2
53721 Siegburg
Germany

Tel.: (+49) 2241-5958-0

Fax: (+49) 2241-5958-5

Hotline: (+49) 2241-5958-12

Main Table of Contents

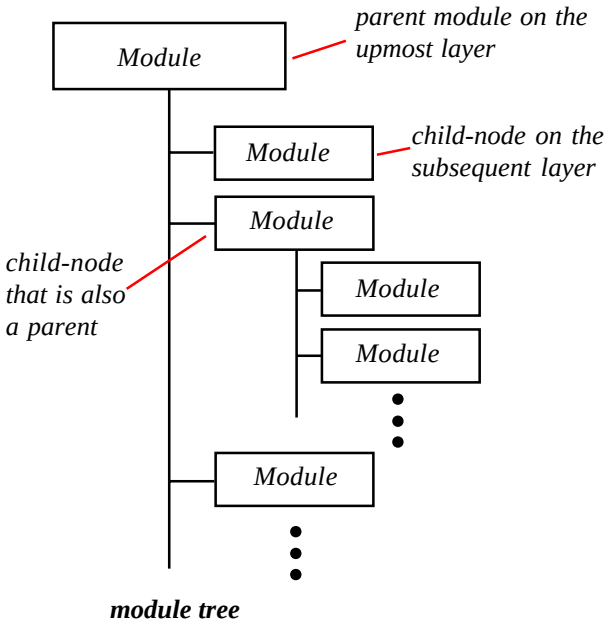
Contents

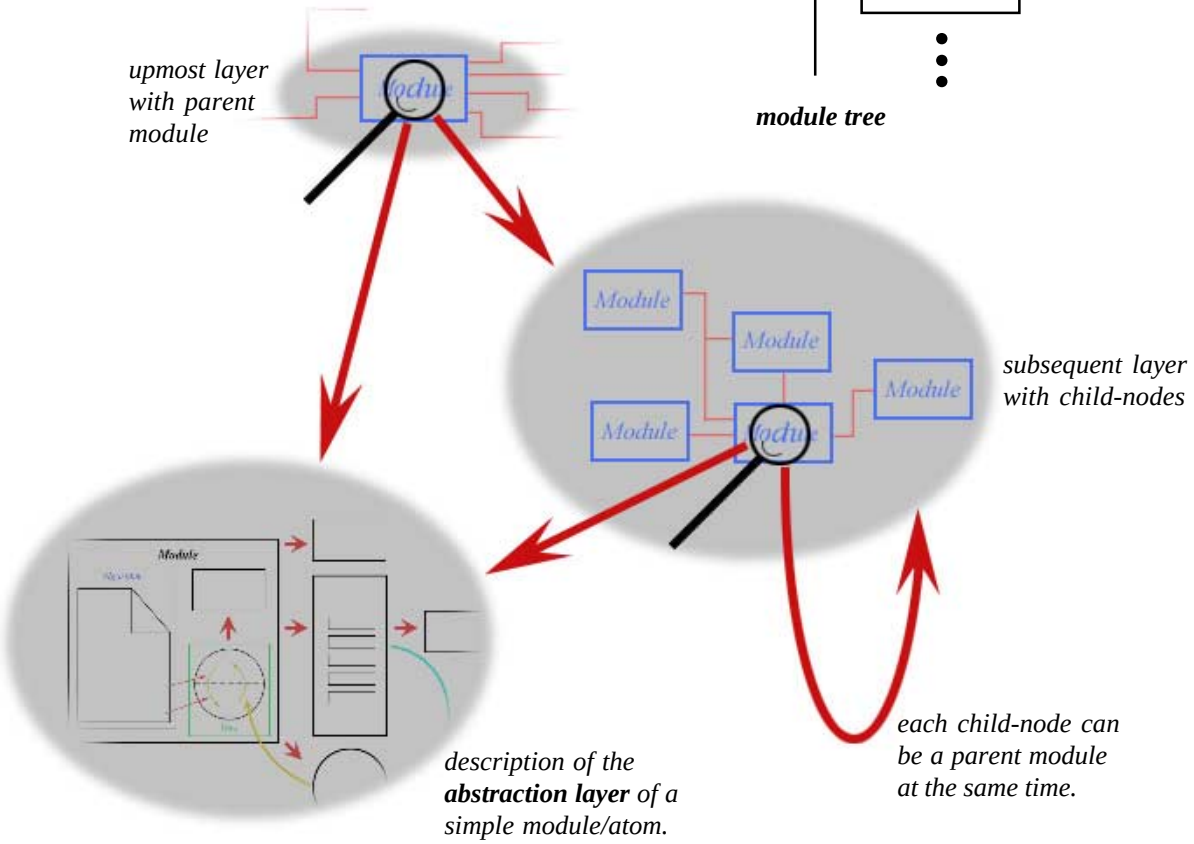
The SCOPE paradigm

This chapter discusses the basic concepts of SCOPE, the elementary entities and their relation to each other.

SCOPE has a component oriented approach. By interconnecting modules so that they become a circuit you build more complex modules. By folding groups of modules you can structure the processing network and build up **hierarchies**. By that you generate processing units that can be saved and re-used in a larger context.
Folding results in a hierarchy that shows the behavior of typical parent-child-relations. The children inherit attributes from the parent module.
This garanties high modularity and re-usibility throughout the environment.

Each layer of the hierarchy remains accessible so that each node and each connection can be edited at any time.
As there is no restriction in the number of possible layers a module can have, each child-node can act as a parent-module itself.
The resulting structure is that of a tree - in SCOPE it is also referenced as the ***module tree***.





Abstraction layer

Modules are the fundamental elements in SCOPE - you could even say that everything inside SCOPE is a module. Therefore it is important to have a closer look to the object called module.

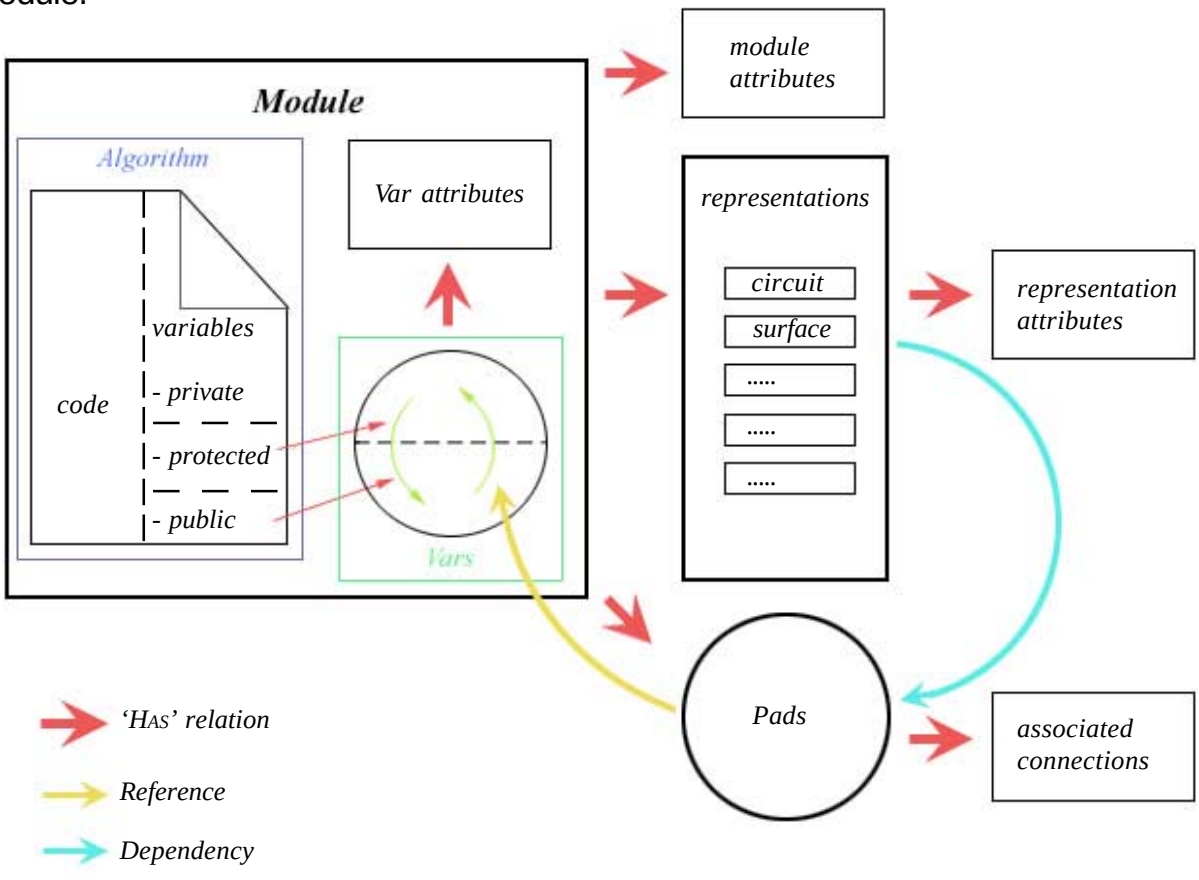
Speaking about a module is more like speaking of a context - appearance, complexity, functionality or structure do not concern. A module is an entity that is fully operational on its own.
Whereas 'module' is a more abstract definition for an entity, the term atom always refers to the smallest entity holding executable code.

The most simple structured module does not necessarily differ much from an atom - neither in functionality, nor in appearance.

To really understand what a module is it is best to have a close up on it. You could call it the *abstraction layer* of a simple module.

A simple module loads an ***algorithm*** that determines its functionality. Basically an algorithm is *executable code*. The algorithm **describes the default state** of the module. The default state is overloaded when customizing the module.

Vars are derived from the protected variables and from the public variables of the algorithm. They pass values from the modules to the corresponding variables. *Vars* are defined by their *attributes* - i.e. their values and their properties.
Pads are automatically generated for *Vars* from public variables. By default *Vars* from protected variables do not have *Pads*.
Pads are references to *Vars*; therefore a *Var* can have multiple *Pads*. *Pads* can be connected to *Pads* of other modules whereas *Vars* cannot be accessed externally. However, you can access and change their value but you cannot connect them to other *Pads* or *Vars*.



The attributes of the *Vars* also affect the character of the referencing *Pads*. The latters themselves have *associated connections*.

Although *Pads* are a sort of interface to access variables of the algorithm they are not always visible throughout the SCOPE environment. As for the module itself this depends on the ***representation***. Currently there are two representations in use - ***circuit*** and ***surface***. Circuit representation is suitable for nodes which generate and process sound. Surface representation is ideal for control elements like faders, potentiometers and so on, which are most likely to be used on sufaces.

A module can have multiple representations. It does not impose any problems if a module has a circuit representation as well as a surface representation. The representations have a set of attributes to manage and optimize them.
Finally a module itself has attributes. Mostly they set processing preferences for the module. For more complex modules these preferences affect the underlying hierarchy as well - i.e. the child-nodes of the module. They are inherited within the *module tree*.

To understand each of these items properly they are discussed one by one in the ongoing consideration.

Algorithm

The algorithm contains the code for the module which is actually executed. In the case of a simple module the algorithm is the object and the module itself is an instance of this object. It only describes the default state and is overloaded by the module itself. This can be down by customizing the module attributes and the settings of the *Pads*.

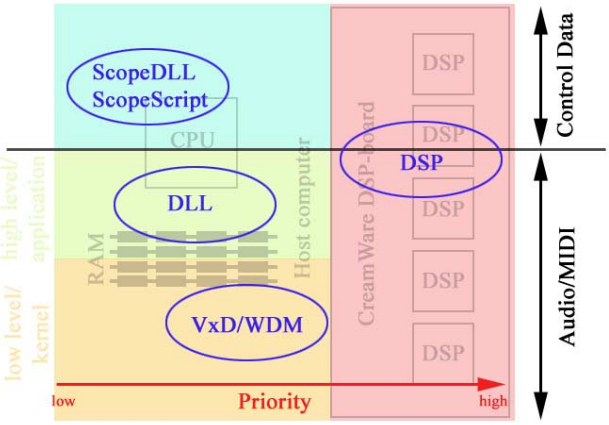
Most often algorithms generate and process signals. There are lots of different signals - audio signals, controller data, MIDI messages, etc. serving specific tasks. One of the most important distinctions between the tasks is their accuracy.
Accordingly it is possible to distinguish between two groups of processes on which tasks rely - **synchronous** and **asynchronous**:
Synchronous processes are *updated on every word-clock impulse*. This is useful for time-critical tasks.
In contrast to this, on **asynchronous** processes *changes occur only occasionally* and *need not to be handled immediately*, but merely ‘very soon’, e.g., within the next millisecond or so.

Thus synchronous processes generate signals that are more timing accurate and their impulses have a higher density. This results in a smoother interpolation of the signal. Therefore their are synchronous signals are peferrably used for audio signals as well as for signals as well as for signals which modulate audible audio signals.
Asynchronous signals mainly generate and process control signals that are generated on user input. In most cases their resolution is to coarse for dynamic modulations. This would result in a stepwise interpolation which leads to audible artefacts, so called zippering.

Within these two groups of tasks further distinctions can be made. Most often timing is a concern in a signal processing environment. So a signal with a high priority has to be processed from an algorithm that does not introduce any

latency. So you can form groups of tasks according to the priority in timing. This is also related to the type of the algorithm which is used to perform them.

Basically there are four different types of algorithms: DSP-, Scopefx-, ScopeDLL- and ScopeScript-algorithms. The Scopefx-algorithms can be differentiated into VxDs on Win 9x/Me or WDMs on Win NT/2000/XP respectively and into DLLs.



SCOPE is a DSP-based development platform. Basically you can say that the more timing is a concern the more likely the algorithm is performed on the DSPs of the CreamWare hardware.
However digital signal processors (DSPs) are not always the best choice for specific processing tasks. Thus there is the possibility to process synchronous and asynchronous signals on the CPU of the host computer. Nevertheless timing should have a lower priority for such tasks.
Finally there are surfaces and control elements. Those only process asynchronous data. It is sufficient to update them on request - i.e. on user input.

Generally speaking you can conclude that most of the time-critical tasks are carried out by the DSPs on the CreamWare hardware. However a series of control signals (like control elements on surfaces, the signal routing on the DSPs) are generated and partially processed by the host.

The following list gives a brief overview of the different types of algorithms with short explainations:

DSP
executed on the CreamWare hardware; for processing both, synchronous and asynchronous signals with the highest priority.

Scopefx
VxD/WDM
executed by the host; for processing both, synchronous and asynchronous signals with medium priority.

DLL
executed by the host; for processing both, synchronous and asynchronous signals with low priority.

ScopeDLL and **ScopeScript**
executed by the host; for processing asynchronous control data and for managing routings on the DSPs. Surfaces and control elements are based on these types.
They are very similar, you can derive classes from one another. The main difference is how they are implemented. ScopeScripts are always written in the Scope scripting language.
The Scope scripting language is very similar to Java and must be interpreted during runtime - in contrast to the C++ implemented ScopeDLLs which may be used similarly.
ScopeDLLs enables you to combine the funcitonality of ScopeScript with the features of C++. Additionally ScopeDLLs provide access to the API of the operating system.

For further informations on the different types of algorithms, please consult the related manuals.

Although DSP-algorithms provide the most timing accurate solutions it is not always necessary to implement the functionality on DSPs. In such cases the other options help to avoid unnecessary load on the DSPs.
Additionally the allocation of the DSP-algorithms on the processors can be customized. Thereby you can optimize your project.This ability is accessible via the *module attributes*.

Module Attributes

A module has a set of attributes. These can be divided into two sections:

- The first one provides information on the selected node and on the underlying algorithm.

- The second one enables you to influence the execution of the DSP-algorithms.

Especially the second ones are often set for the module as a whole, and not for each child-node. Indeed this means using the module tree with its parent-child-relation. Thus the settings of the parent module are inherited by the children.

The given information on the selected nodes include:

- **Name**
  name of the module. The module does not always have the same name as the algorithm or the atom from which it is derived.

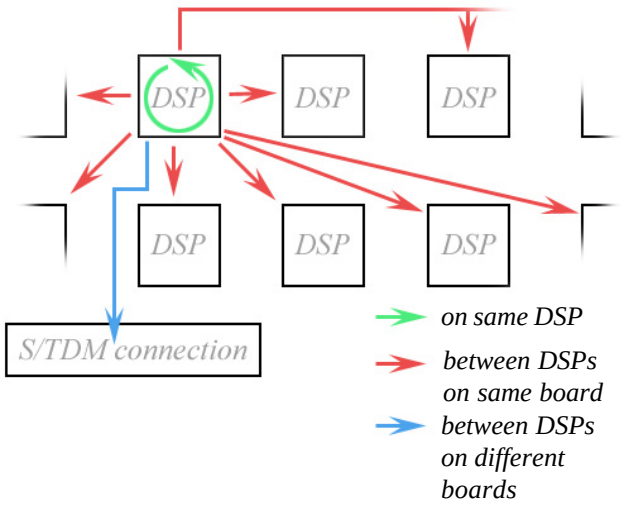
- **Class**
  indicates the name of the class of ScopeScripts and ScopeDLLs. Regarding these algorithms a module is an instance of its class.

  For ScopeScripts there are dependencies regarding the name of the class and the name of the file. The file has to have the same name as the class. Additionally the class name has to reflect the relative path of the file. The root directory for ScopeScripts is always the 'Script' folder inside your SCOPE Installation folder.
  If the file of a script class is in a sub-folder the name of the class - not that of the file - has to take this into account: The relative path is added to the class name and the path separator is replaced by a '@'. If you look at the class 'Surfaces@BasicSurface' the name of the script-file is 'BasicSurface.pep' and it is located in the 'Surfaces' folder inside the 'Script' folder. Therefore the relative path would be 'Surfaces/BasicSurface'.

- **File**
  filename and the path of the algorithm.
  For DSP-, DLL- and VxD-algorithms the absolute path and the filename with extension is displayed.
  For ScopeScripts and ScopeDLLs the relative path as outlined above is displayed.

The second section of the module attributes influences the placement of the DSP-algorithms on the digital signal processors. To understand these attributes it may be best to first have a closer look on how the DSP-algorithms are processed and how they transfer data.

SCOPE is based on a dedicated DSP hardware. Not only is one digital signal processor capable of handling multiple threads simultaneously but also the environment is able of using multiple processors on different boards at the same time.
This induces different types of connections and varying transfer times (latencies) according to the involved DSPs.The following scheme depicts this:



A processor has a specific amount of cycles within which he can perform tasks. As it can also perform different tasks simultaneously you can load a specific number of modules onto a single DSP. The transfer between two modules on the same DSP happens immediately.
Transfers between any two DSPs on the same board take two samples to complete. As all the DSPs on a single board are connected to each other, it is of no importance which processors are are involved.

Transfers between DSPs on different boards happen within two to six samples. This depends on the involved DSPs and the available S/TDM connections.
Although these are all rather short times, for some application this might be important. These transfers affect the phase correlation of multi-channel audio signals if their routings introduce different amounts of latency.

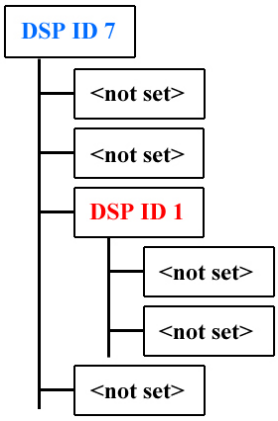
You can assure that the phases stay the same by watching the routings. The second part of the module attributes serves this purpose.

Module attributes are normally made for the parent and are inherited by the children.
Although these settings only impact DSP-atoms each module has them. This is a logical consequence of the parent-child-relation within the *module tree.* Taking into account that every module can be a paren, on the upmost layer or on a subsequent layer, this appears to be very useful.
Therefore **every** module has to be able to pass and/or inherit these options.

It is important to emphasize that settings made for a module on a lower layer of the hierarchy overwrite settings that are inherited from the parent. The child-nodes of this lower layer module inherit the settings of their direct parent module.
Therefore it is possible to specify different settings for a sub-tree of the hierarchy.

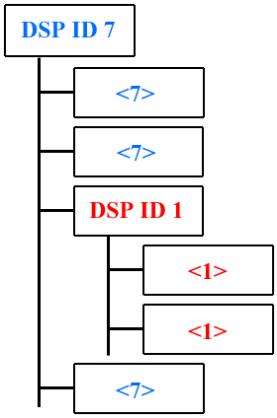
Most of the attributes can be inherited. You can set different values for each option. Those which can be inherited have a value <not set>. If it gets overwritten by a parent it changes to <inherit *value*>.

To clarify this consider an examplary case:
The DSP ID allows you to specify on which digital signal processor the code of a module is loaded and executed. For the exemplary module this flag is set for the upmost module and for one of its children.



This flag is only set for two of the modules of the module tree. For the others the flag is not set.

Scope will distribute the modules according to their parent-child-relation in the module tree. It is important that the value set for the child-node overwrites the value inherited from the parent!



The individual attributes are:

Single Load
values: yes, no
determines if the DSP/SCOPEfx-atom is loaded only once. 'yes' means it is monophonic, 'no' means that it is polyphonic with the number of voices set for Voices.
Single Load can be set directly in the code of the underlying algorithm. In such a case this flag is has no affect.

! *Single Load can not be inherited*

Voices
values: <not set>, any number between 0 and x
determines the number of voices a module uses. Does only take effect if **Single Load** is set to 'no'.
'0' removes the module from the DSPs (not from the **SCOP**E environment) to reduce the load on the processors.
'-1' is equivalent to <not set> and allows the module to inherit the value from the parent.

! *The difference between Single Load and Voices set to '1' might not be obvious. Single Load tells the module if it is monophonic or not. If it is monophonic it only has one output.*
If it is set to polyphonic (Single Load = no) it has as many active outputs as it has voices.
So, in a polyphonic circuit a module that is only loaded once (Single Load = yes) the output is fed to all active inputs of the connected polyphonic modules. In contrast to that if the module has only one active voice (Single Load = no; Voices = 1) the modules sends its single active output to the first active input of connected polyphonic modules. The other active inputs of the connected polyphonic modules do not receive a signal.

Board ID
values: <not set>, any number between 0 and x
determines the DSP-board on which the module is loaded.
The highest applicable value depends on the number of DSP boards you have installed with the convention that x='amount' - 1.
'-1' is equivalent to <not set> and allows the module to inherit the value from the parent.

DSP ID
values: <not set>, any number between 0 and x
specifies on which DSP a module is loaded. Does not necessarily load the whole module onto the same DSP (see **On Same DSP**)! Use it in conjunction with **Board ID** and **On Same DSP**.
The highest applicable value depends on the number of DSPs on the selected board.
'-1' is equivalent to <not set> and allows the module to inherit the value from the parent.

On Same DSP
values: <not set>, yes, no
determines if a entire module should be executed on a single DSP or not. A module can only be executed on a DSP if the processor is capable of computing all necessary cycles simultaneously.
'-1' is equivalent to <not set> and allows the module to inherit the value from the parent.

Hint
values: <not set>, yes, no
specifies if SCOPE should try to load an entire module on the same DSP board. This is more like a recommendation to the environment. Whereas the former discussed options **Board ID**, **DSP ID** and **On Same DSP** are obligatory this option is a hint.
Please note that hint should be only set for the upmost node of a module/device. If you set this flag again on a lower level you get a new, independent hint-group.

DSP Placement
values: unassigned, tuples
provides information on the placement of a module on the DSPs. The information is provided in tuples of the **Board ID** and the **DSP ID** for each voice of the module.
unassigned indicates that the current node either not loaded onto a DSP (Voice setting is '0') or it is a ScopeScript/ScopeDLL and therefore not executed on a DSP.
Additionally there are two special tuples: (0/15) is used for Scopefx nodes which are also not placed on a DSP. (-1/-1) indicates that the node could not be loaded onto a DSP. This is especially important for DSP developers.
For modules with children DSP Placement information on its *module tree* is also available.

The module attributes enable you to customize the distribution of the DSP-algorithms on the digital signal processors and therefore handle the latencies of your multi-channel audio signals.
Normally modules are distributed intelligently on the DSP by a smart algorithm. So the module attributes should only be set manually, if you encounter any problems or if you have special requirements.
You can say that the *module attributes* influence the execution of the module in the environment whereas *Vars* affect the processing directly.

Vars

Vars are derived from public and protected variables of the algorithm. The *Vars* of the module form the interface towards the algorithm. The value of the Var is passed to the corresponding variable of the algorithm.
There are three different types of these interfaces - the so called I/O-types: inputs (InPads), outputs (OutPads) and combined inputs/outputs connectors (IOPads).
Furthermore a *Var* has a specific type and a specific range for which the received values are valid. *Vars* can be of any of these types: int, float, short, byte, double, char, string, boolean, object or MIDI.

For a more detailed description of the Var types consult the programming tutorials.

Most often ints are used. If the range of a Var has only positive scale then it is **unipolar**, if it has both, positive and negative scale, it is **bipolar**. The currently highest value for int is 2147483647 (2^{31}-1; referenced as +Max) and the lowest -2147483647 (-Max) respectively.
There are a couple of *Vars* for which the range is not sufficient to operate properly - they also necessitate a *unit* argument. This is true for frequency, time and gain *Vars*. The argument tells the Scope environment how to interpret the given value.

Like processes *Vars* can be **asynchronous** or **synchronous**. Sychronous *Vars* are updated on every word-clock whereas asynchronous ones are updated 'very soon' after changes did occurred.
All changes made to *Vars* are passed to the variables and therefore are reflected in the output of the process.

Each *Var* has a set of attributes:

Name
name of the *Var*

I/O type
InPad, OutPad, IOPad

Variable Type
the variable type, e.g. int, string, etc.

Size
The size in bytes that is reserved for the *Var*. This is not applicable to all *Vars*, as strings, for example, are dynamic.

Range
maximum and *minimum* values which define the valid range and if the *Var* is unipolar or bipolar

Processing mode
which tells if it is *synchronous* or *asynchronous*

Unit
Information needed to interpret the *Vars* values, only for specific *Vars*

*Normally it is **not useful or even advisable** to change the arguments of these attributes, as this may result in unpredictable behaviour. The Var is derived from the algorithm and is the interface for the variables. Changing the name or even the I/O type of the Var can render the Var inoperable.*

According to these attributes you can determine which values are valid arguments for this *Var*.
They do not only concern the *Var* itself but also for all ***Pads*** that reference this *Var*.

Pads

Vars themselves cannot directly exchange data with other *Vars* as they have a strong relation to the variables of the algorithm. Changing the attributes of a *Var* means to directly alter this link - in most cases the link will break.This results in an inoperable module.
This is the reason why *Vars* would be too restricted when being used as external connectors for modules. As a consequence the value of a *Var* can only be accessed manually.
Therefore *Pads* serve as external connectors to modules. They reference *Vars*. Thus *Var* attributes also apply to them.

Pads are automatically generated for *Vars* from public variables. You can generate multiple *Pads* for **every** *Var*. You can as well create *Pads and associate them with Vars* from protected variables.
This is like overwriting the qualifier (public and protected) of a variable during run-time, as it will get accessible by creating a *Pad*. Analogous *Pads* can be deleted.

Pads can be connected to *Pads* on the same module or on others on the same or on a different layer. They can receive and transmit values depending on the I/O type of their *Var*.
Sharing the attributes with the *Vars*, *Pads* have rather associated connections than attributes. Their names and their I/O types can be modified freely. Pads are external connectors and their names and their I/O types are nothing but representations of these connectors.
This is a very flexible architecture and pretty convenient. It only affects their appearance and not their behavior.

The *type of the algorithm, the module attributes* and the *Vars* determine how the module behaves in the SCOPE environment. They specify and influence how the module is processed and transmit values from or to the modules. *Pads* however add a new dimension - the graphical *representation* in the environment.
Also modules themselves have graphical *representations*. They augment the ease of use and indicate the scheduled area of application.

Representations

It has already been explained that there are synchronous and asynchronous processes and signals, as well as that processes are executed with varying latency either on the host computer or on the DSP hardware. These considerations are related to how a task is handled rather than what it is for.
However, when speaking about the representations of modules the designation of the tasks is more important than the handling.

There are tasks that generate signals and others that manipulate them. Certain modules generate signals dynamically, others from user input. The first ones are typically use in circuits, the latters on the module's user interface. Although both generate signals the representations of these modules are be different.
Indeed, for circuit components the representation is rather simple. You have a box that symbolizes the processing entity. It has to show the name and the *Pads* of the module. The box itself does not necessarily need a specific size, it should even resize dynamically

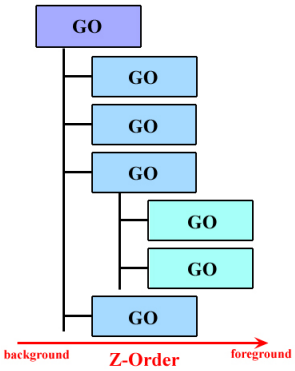
according to the number of *Pads* and the length of the module's name.
As opposed to circuit components, surface elements are used as user interface. Normally you want the module's surface to appear always like you have designed it. So surface elements have fixed dimensions. As a surface element often mimics a real world element, like a fader or a potentiometer, its representation is more complex, too.
Furthermore the representations of the surface elements check the computer screen for user input.

Additionally the SCOPE environment provides the ability to work with different views. There is a *Surface* view and a *Circuit* view. In *Surface* view you can edit surfaces and use modules that have a surface representation. In *Circuit* view you edit circuits and use modules with a circuit representation.
This is very handy because while in *Circuit* view you do not have to deal with surface elements (i.e. modules with a surface representation) and while in *Surface* view not with circuit components (i.e. modules with a circuit representation) respectively. Instead you can concentrate on the principal tasks.

As outlined before, the way a module is displayed does not depend on the algorithm or the assigned processes but on its representation. It can even have a representation for each view.
However in most cases a module needs only one representation. Most likely a module is mainly used in one view. Only in some situation it might be reasonable to have more than one representation.
Nevertheless SCOPE gives you the freedom to fit it to your requirements.

*Currently there are two views implement - **Surface** view and **Circuit** view. Further views can be implemented if needed.*

Representations use bitmaps or animations to visualize the module graphically. The *representation attributes* enable you to optimize the performance and the graphical layout for each module or fot the whole surfaces.
A representation of a single node normally consists of more then one graphical object - a so called ***GO***. Indeed you can speak of a hierarchical structure of graphical objects - the ***GO tree***. The lower the *GO* is in the hierarchy the higher is its ***Z-Order***. This relation is also called the *Z-Depth*.



GO tree
Graphical Objects with the same Z-Order are displayed with the same color.

GOs on the same hierarchy layer have the same *Z-Order*. If *GOs* with the same *Z-Order* overlap the recently selected *GO* will display on top.
The elements of a *GO tree* also have a parent-child-relation to each other. Most attributes set for the parent are inherited by the children as long as they are not set manually for them. This is analogous to the inheritance of module attributes in the *module tree*.

Representations for single modules are made of *GOs* and *GO trees*. A single modules, however, is a member of a *module tree*. For circuit components the *module tree* is sufficient to manage their representations as well. For surface

elements with their more complex *GO trees*, it is not.
The user interface of a module is a composition of complex *GO trees*. The *Go trees* of the individual surface elements have to be organised in a hierarchy that corresponds to the layout of the user panel.
Surface elements are modules as well and are therefore part of the *module tree*. Although this is a hierarchical structure it is not sufficient for this purpose.
The primary concern of the *module tree* is to generate a hierarchy that organizes the circuit into a collection of larger processing entities. This is mainly done by folding or using other modules as containers. To be more accurate, the *module tree* more or less affects and takes care of the processes.
So, this is more about how processes are performed and which ones are linked.

Although the modules themselves are already organized in the *module tree*, it is also desirable to organize their graphical representations into larger containers which structure the appearance of the surface.
So there is the need of an independent hierarchical structure with objects that combine single *GOs* or even complete *GO trees* into groups. As the *module tree* handles processes this hierarchy would handle the visualization.

In SCOPE this hierarchy is called ***ViewTree***. It organizes the *GOs* and *GO trees* of surface elements in a new hierarchical structure. The generated composition is directly related to how these modules are organized in the *module tree*. However it is not identical, it rather depends on the involved *GOs* and *GO trees*.

The *ViewTree* is not a classical parent-child-relationship. There is **no inheritance** of attributes within the *ViewTree*. Additionally, only *GO Groups* can act as parents. Like folders in the *module tree* their main concern is to group individual *GOs* and *GO trees* and to manage the graphical visualization of their members.

For the moment it is not really necessary that you understand the concept of the ViewTree completely. The SCOPE application computes it in real-time according to specific rules. Indeed it is not that important do understand what the ViewTree is but what it does.
In the following chapters it will be discussed in more detail what it does.

It is important to understand thoroughly how the individual attributes affect the way the representations are displayed.

View ID
Specifies the type of the representation. ‘Surface’ means that it is a only visible in *Surface* view. ‘Circuit’ means it is only visible in *Circuit* view. If ‘None’ it is not visible at all.

Z-Order
Sets the *GO* layer for the representation. *GOs* with a higher Z-Order are positioned on top. An argument ‘Normal’ displays the object according to its position in the *GO tree* or *ViewTree* respectively. ‘Bottom’ will move it to the background. Numbers specify higher layers.

Visible
Controls the visibility of the *GO*/ *GO tree*. This attribute is inherited and cannot be overwritten. A child-*GO* cannot be visible if its parent is not.

Selectable
Specifies if the selected *GO* can be selected.

Fixed Z-Order
This flag forces all *GOs* into a fixed hierarchy. The hierarchy is determined by the *ViewTree* and then followed by the *GO tree*. Generally speaking this flag affects *GOs* that are located on the same level in the *GO tree* as well as *GO trees* that are on the same level in the *module tree*.

Draw ClipChildren
This flag affects how a parent *GO* (in the *GO tree* as well as in the *ViewTree*) will display its children. If it is set it forces the parent to only draw/redraw *GOs* that are inside its dimensions. Everything outside is clipped.
If this flag is not set the parent looks for its children on the whole surface to (re-)draw them. This can be time consuming.

Select ClipChildren
This flag prevents children from being selected if they are not positioned within the dimensions of the *GO*. Thus the parent does not need to check inputs from them.

! *For performance issues these flags (**Draw ClipChildren**/**Select ClipChildren**) should be set almost every parent.*

! *The last four flags only work for **GO Groups** and are related to the **ViewTree** rather than to the **GO tree**. Go Groups are surface equivalents to the folders in the circuit. They also serve as parents in the **ViewTree**.*

Custom horizontal size
Set this to resize a *GO Group* manually in horizontal direction. If this flag is unchecked and the '**ViewTree Group**' flag is set the parent will resize so that it includes all its children.

Custom vertical size
Set this flag to resize a *GO Group* manually in vertical direction. If this flag is unchecked and the **ViewTree Group** flag is set the parent will resize so that it includes all its children.

ViewTree Group
This flag specifies that the *GO Group* resizes to include all its children from the *ViewTree*. For performance issues the two custom size flags should be set and this flag should be turned off.
This flag should be **only** used to calculate the custom size of the selected parent. After having set the '**Custom horizontal size**' flag and the '**Custom vertical size**' flag it should be turned off again!

Behave like view
This flag determines that the children are positioned at the upper left border of the parent.

These options customize the representation of modules. One of the most important attributes is the *View ID*. It affects the visibility and also the usability. For surface elements (*View ID* set to *Surface*) *Pads* are not displayed as they are not supposed to be visible on surfaces.
Nevertheless they still exist and can be connected.

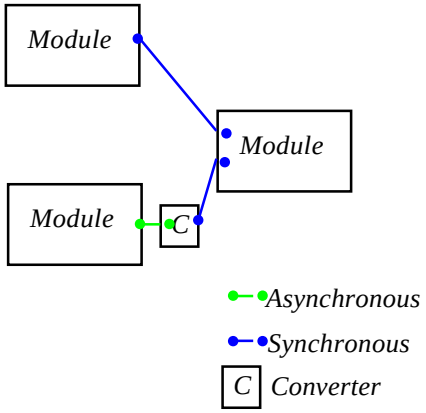
Signal conversion

As pointed out before SCOPE follows a component oriented approach. The module is the basic element throughout this approach. After this brief overview you should have a basic unterstanding of a module.
If you recall the structure of a module (on page 5), until now this discussion mainly considered the abstraction layer. However most of the time you actually work on a 'normal' layer.

On a layer you combine multiple modules to form a pocessing network. This is done by making connections between the *Pads* of the modules.
The connections serve to transmit signals. The different signals can be divided into three main groups - synchronous signals, asynchronous signals and MIDI messages.
According to these groups the related *Pads* have different update rates and different ranges. This means that the algorithms expect a specific kind of data.

> ***Please keep in minde that** a **Pad** is a reference to a **Var** which is transfering the values to and from the variables of the algorithm. As a reference a **Pad** shares the attributes of the **Var**.*

As you can imagine a unipolar *Pad* does not understand a bipolar signal without any kind of conversion. A conversion between an asynchronous and a synchronous signal is other example.



As a consequence there have to be different converters that handle the signal conversion between different Pads. Indeed the only conversion that is not possible is from and to MIDI messages.

Associating *Pads*

In a consecutive step a processing network can become a sub-network by packing the nodes of the circuit into a new module. This generates a sub-layer to which the specified modules are moved. The relation between modules and layers is iterative as the scheme on page 5 indicates.
Nevertheless the signals have to be routed to the *Pads* of the modules which are now on a lower layer. To enable these connections you can generate new *Pads* and associate them with *Pads* of modules on a lower layer. This is possible because a *Pad* is a reference to a *Var*.

Parameters and Presets

There are a few more attributes that are especially interesting for more complex modules or for devices.

A **device** is the third file type that you encounter while working with SCOPE - besides atoms and modules. Keep in mind that a module is an entity for which appearance, complexity, functionality and structure are not characterizing. More likely it is a composition of one or more atoms and most often it has a control surface. Typically it can be re-used in a larger context.
An atom by itself is simply the smallest entity hoding executable code.
Devices are complex modules which have more or less one specific

functionality. Additionally they have a surface and presets. So a device can be considered to be the final state of your circuit.

If the basic design for a device is finished you might want to test it with different controller settings. These can be saved in **presets**. Other then for a module which current state is captured by saving the module, presets allow you to switch easily between different settings.

Actually there are two different kinds of presets in SCOPE - the first ones are *Var* oriented and the second ones are *parameter* oriented.
Presets save the state of a *Var* - or in other word they save the value of the *Var*. Doing this for all significant *Vars* captures the current state of the device. However, you might not want to save all the *Vars* in presets. The number of voices, the state of the device's surface (open-closed), the position of the surface on the computer screen. Normally these states are not saved in presets, because you do not want them to change each time you load a new preset.

The *Var* oriented presets are faster to set up and might therefore be prefered as long as the device is not completed yet. Later on, the *parameter presets* are more powerful and handier at the same time.
Parameters in this context are a collection of *Vars* that have been defined as parameters.

Restore levels
The faster method for prototyping is the *Var* oriented implementation. Each *Var* has a **restore level** that enables you to customize the restore behavior of the *Var.* By that you can specifiy for each *Var* whether it should be restored in presets or not. For devices you can even define that a *Var* is not restore after loading.
The default *restore level* of a *Var* is that it restores in presets as well as after loading. As a consequence a lot of data has to be saved with the presets and has to be restored after loading. This results in hugh preset files and long loading times.

To avoid this you would have to customize the *restore level* of the *Vars*. Having to set them manually for each *Var* would not be very convenient. Normally only the values of a minority of the *Vars* are of interest for presets. This means the *restore level* of the majority of them needs to be adjusted to avoid unnecessary overhead.

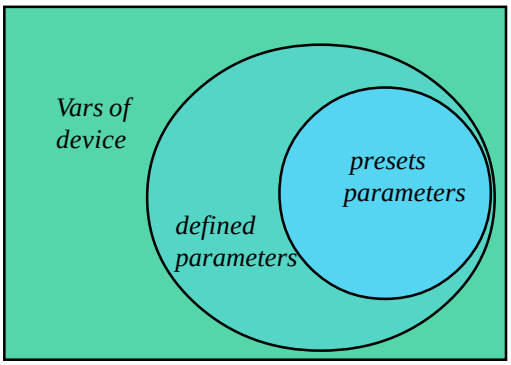
Parameter presets
Parameter presets work the other way round. You have to select the *Vars* you want to use as parameters. In a second step you define the *preset parameters* from these parameters.

Parameters are saved with the project files, preset parameters in the presets.

Generally you want to save the current state of your device with the project. In preset however you only save a subset of those parameters - like you would like to save with the project if a surface of your device is open or closed which you do not always want to save with every preset.
There are a couple of other *Var* values which you only want to save with the project but not in presets.

As the listings of the *preset parameters* are a subset of the parameters, you first define the latters and in a subsequent step you create a *preset parameter* list

from the parameter list. The *preset parameter* list determines which *Vars* are captured in presets.

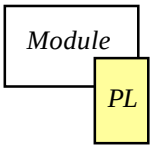


Although *preset parameters* seem to be more time demanding they allow a much greater control over the resource usage and memory consumption of the device. Compared to the constraints it would take to achieve the same results with the *Var* presets it is still the faster and the more proper solution.
It also provides a couple of other serious advantages. Some are listed below:

- It becomes possible to exchange modules within the device without losing the presets.
- Presets can be applied to similar devices or groups (like channels for mixer console).
- It gets much easier to make preset lists for individual sections of a device.
- You can easily create preset lists containing different sets of preset parameters for one device.
- The new automation concept is based upon the parameter concept.

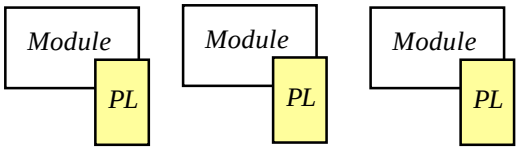
Parameter context

A parameterized module or device also has a ***parameter context*** which is another *module attribute*. A *parameter context* contains all the parameters of a paramterized module or device.
So what is it good for? It may occur frequently that parts of your circuit are used more than once in a complex circuit. It was mentioned that a module can be re-used in a greater context - and of course it can be re-used more than once in a single device.
It was also pointed out that only a minority of the *Vars* of the device get paramterized. Unfortunately, a minority can still be a huge number. Therefore, it is a good idea to parameterize the sub-module before you duplicate it to re-used it several times.

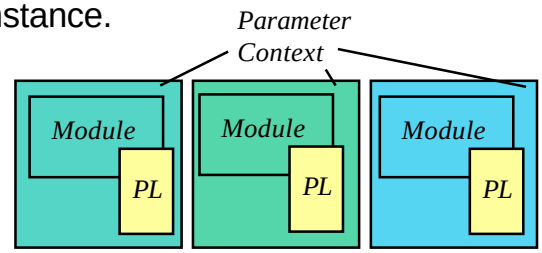


*Module with **P**arameter **L**ist*

Having done it this way, you have a couple of modules having the same parameters, now. To be able to distinguish between them SCOPE needs some kind of additional information. This could be an attribute that links each the parameter to the appropriate instance of the module.



Exactly this is the functionality of the *parameter context*. It puts the parameters into the context of the instance. So the parameters can be addressed independently for each instance.



This was the theory of how the SCOPE environment works. By now, you should know have obtained a brief overview what the individual parts and options are and how they interact and affect each other.
In the next chapters you will learn more about the SCOPE application and the work-flow. Whilst reading these chapters you will associate concrete windows, commands and operations with the points that were discussed theoretically in this chapter.

Index