

VAB™

Language Reference

Visual Application Builder

Worldwide Technical Support and Product Information

ni.com

National Instruments Corporate Headquarters

11500 North Mopac Expressway Austin, Texas 78759-3504 USA Tel: 512 683 0100

Worldwide Offices

Australia 1800 300 800, Austria 43 0 662 45 79 90 0, Belgium 32 0 2 757 00 20, Brazil 55 11 3262 3599, Canada 800 433 3488, China 86 21 6555 7838, Czech Republic 420 224 235 774, Denmark 45 45 76 26 00, Finland 385 0 9 725 725 11, France 33 0 1 48 14 24 24, Germany 49 0 89 741 31 30, India 91 80 51190000, Israel 972 0 3 6393737, Italy 39 02 413091, Japan 81 3 5472 2970, Korea 82 02 3451 3400, Lebanon 961 0 1 33 28 28, Malaysia 1800 887710, Mexico 01 800 010 0793, Netherlands 31 0 348 433 466, New Zealand 0800 553 322, Norway 47 0 66 90 76 60, Poland 48 22 3390150, Portugal 351 210 311 210, Russia 7 095 783 68 51, Singapore 1800 226 5886, Slovenia 386 3 425 4200, South Africa 27 0 11 805 8197, Spain 34 91 640 0085, Sweden 46 0 8 587 895 00, Switzerland 41 56 200 51 51, Taiwan 886 02 2377 2222, Thailand 662 278 6777, United Kingdom 44 0 1635 523545

For further support information, refer to the *Technical Support and Professional Services* appendix. To comment on National Instruments documentation, refer to the National Instruments Web site at ni.com/info and enter the info code `feedback`.

Important Information

Warranty

The media on which you receive National Instruments software are warranted not to fail to execute programming instructions, due to defects in materials and workmanship, for a period of 90 days from date of shipment, as evidenced by receipts or other documentation. National Instruments will, at its option, repair or replace software media that do not execute programming instructions if National Instruments receives notice of such defects during the warranty period. National Instruments does not warrant that the operation of the software shall be uninterrupted or error free.

A Return Material Authorization (RMA) number must be obtained from the factory and clearly marked on the outside of the package before any equipment will be accepted for warranty work. National Instruments will pay the shipping costs of returning to the owner parts which are covered by warranty.

National Instruments believes that the information in this document is accurate. The document has been carefully reviewed for technical accuracy. In the event that technical or typographical errors exist, National Instruments reserves the right to make changes to subsequent editions of this document without prior notice to holders of this edition. The reader should consult National Instruments if errors are suspected. In no event shall National Instruments be liable for any damages arising out of or related to this document or the information contained in it.

EXCEPT AS SPECIFIED HEREIN, NATIONAL INSTRUMENTS MAKES NO WARRANTIES, EXPRESS OR IMPLIED, AND SPECIFICALLY DISCLAIMS ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. CUSTOMER'S RIGHT TO RECOVER DAMAGES CAUSED BY FAULT OR NEGLIGENCE ON THE PART OF NATIONAL INSTRUMENTS SHALL BE LIMITED TO THE AMOUNT THEREFORE PAID BY THE CUSTOMER. NATIONAL INSTRUMENTS WILL NOT BE LIABLE FOR DAMAGES RESULTING FROM LOSS OF DATA, PROFITS, USE OF PRODUCTS, OR INCIDENTAL OR CONSEQUENTIAL DAMAGES, EVEN IF ADVISED OF THE POSSIBILITY THEREOF. This limitation of the liability of National Instruments will apply regardless of the form of action, whether in contract or tort, including negligence. Any action against National Instruments must be brought within one year after the cause of action accrues. National Instruments shall not be liable for any delay in performance due to causes beyond its reasonable control. The warranty provided herein does not cover damages, defects, malfunctions, or service failures caused by owner's failure to follow the National Instruments installation, operation, or maintenance instructions; owner's modification of the product; owner's abuse, misuse, or negligent acts; and power failure or surges, fire, flood, accident, actions of third parties, or other events outside reasonable control.

Copyright

Under the copyright laws, this publication may not be reproduced or transmitted in any form, electronic or mechanical, including photocopying, recording, storing in an information retrieval system, or translating, in whole or in part, without the prior written consent of National Instruments Corporation.

Trademarks

National Instruments, NI, ni.com, and LabVIEW are trademarks of National Instruments Corporation. Refer to the *Terms of Use* section on ni.com/legal for more information about National Instruments trademarks.

Other product and company names mentioned herein are trademarks or trade names of their respective companies.

Members of the National Instruments Alliance Partner Program are business entities independent from National Instruments and have no agency, partnership, or joint-venture relationship with National Instruments.

Patents

For patents covering National Instruments products, refer to the appropriate location: **Help»Patents** in your software, the `patents.txt` file on your CD, or ni.com/patents.

WARNING REGARDING USE OF NATIONAL INSTRUMENTS PRODUCTS

(1) NATIONAL INSTRUMENTS PRODUCTS ARE NOT DESIGNED WITH COMPONENTS AND TESTING FOR A LEVEL OF RELIABILITY SUITABLE FOR USE IN OR IN CONNECTION WITH SURGICAL IMPLANTS OR AS CRITICAL COMPONENTS IN ANY LIFE SUPPORT SYSTEMS WHOSE FAILURE TO PERFORM CAN REASONABLY BE EXPECTED TO CAUSE SIGNIFICANT INJURY TO A HUMAN.

(2) IN ANY APPLICATION, INCLUDING THE ABOVE, RELIABILITY OF OPERATION OF THE SOFTWARE PRODUCTS CAN BE IMPAIRED BY ADVERSE FACTORS, INCLUDING BUT NOT LIMITED TO FLUCTUATIONS IN ELECTRICAL POWER SUPPLY, COMPUTER HARDWARE MALFUNCTIONS, COMPUTER OPERATING SYSTEM SOFTWARE FITNESS, FITNESS OF COMPILERS AND DEVELOPMENT SOFTWARE USED TO DEVELOP AN APPLICATION, INSTALLATION ERRORS, SOFTWARE AND HARDWARE COMPATIBILITY PROBLEMS, MALFUNCTIONS OR FAILURES OF ELECTRONIC MONITORING OR CONTROL DEVICES, TRANSIENT FAILURES OF ELECTRONIC SYSTEMS (HARDWARE AND/OR SOFTWARE), UNANTICIPATED USES OR MISUSES, OR ERRORS ON THE PART OF THE USER OR APPLICATIONS DESIGNER (ADVERSE FACTORS SUCH AS THESE ARE HEREAFTER COLLECTIVELY TERMED "SYSTEM FAILURES"). ANY APPLICATION WHERE A SYSTEM FAILURE WOULD CREATE A RISK OF HARM TO PROPERTY OR PERSONS (INCLUDING THE RISK OF BODILY INJURY AND DEATH) SHOULD NOT BE RELIANT SOLELY UPON ONE FORM OF ELECTRONIC SYSTEM DUE TO THE RISK OF SYSTEM FAILURE. TO AVOID DAMAGE, INJURY, OR DEATH, THE USER OR APPLICATION DESIGNER MUST TAKE REASONABLY PRUDENT STEPS TO PROTECT AGAINST SYSTEM FAILURES, INCLUDING BUT NOT LIMITED TO BACK-UP OR SHUT DOWN MECHANISMS. BECAUSE EACH END-USER SYSTEM IS CUSTOMIZED AND DIFFERS FROM NATIONAL INSTRUMENTS' TESTING PLATFORMS AND BECAUSE A USER OR APPLICATION DESIGNER MAY USE NATIONAL INSTRUMENTS PRODUCTS IN COMBINATION WITH OTHER PRODUCTS IN A MANNER NOT EVALUATED OR CONTEMPLATED BY NATIONAL INSTRUMENTS, THE USER OR APPLICATION DESIGNER IS ULTIMATELY RESPONSIBLE FOR VERIFYING AND VALIDATING THE SUITABILITY OF NATIONAL INSTRUMENTS PRODUCTS WHENEVER NATIONAL INSTRUMENTS PRODUCTS ARE INCORPORATED IN A SYSTEM OR APPLICATION, INCLUDING, WITHOUT LIMITATION, THE APPROPRIATE DESIGN, PROCESS AND SAFETY LEVEL OF SUCH SYSTEM OR APPLICATION.

TABLE OF CONTENTS

Background and Disclaimer	4
Introduction to OORVL Programming	4
Before you begin	5
General Coding Strategy	6
Language Elements	7
Native Components	7
Vector Components	7
Getting Started With OORVL	7
Variables and Arithmetic Expressions	8
The If Statement	10
The For Statement	11
The While Statement	13
Variable Names	15
Data Types	15
Constants	15
Arrays	15
Union of Variables	16
Functions	17
Types, Operators, and Expressions	17
Arithmetic Operators	17
Relational and Logical Operators	18
Bitwise Operators	19

Type Conversions	19
Assignment Operators and Expressions	20
Order of Operations	20
Control Flow	21
Functions and Program Structure	21
OORVL Hierarchical Components	21
Native and Vector Components	22
Pointers and Arrays	22
Read Element At	22
Variable Read	22
Variable Read with Offset	23
Variable Write	23
Variable Write with Offset	23
Write Element At	23
Summary	24

Background and Disclaimer

National Instruments produces several graphical-oriented component software design products for engineering and scientific work, including the RIDE[®] (Real-time Integrated Development Environment) and VAB[™] (Visual Application Builder) software products. Over the years, NI has noticed a tendency for users to refer to their algorithm development using our products as “writing their application in RIDE” – as if RIDE were a language itself, which is not really the case, RIDE is a product. With an eye to software architecture and hoping to improve the capabilities of our graphical-oriented component DSP algorithm development products, NI studied its products and compared them to typical high-level textual languages (like C, etc.) to see how it matched up.

After studying what was thought to be an abstraction of the requirements for a high-level language, RIDE was fortified with certain new capabilities. Resulting, this document was created to do an internal crosscheck of how well RIDE could perform when thinking of it as a language albeit a graphical language. An acronym was created, OORVL (Object Oriented Real-time Visual Language) to refer to the “language” used within the products (such as RIDE or VAB). NI is not putting effort into introducing yet another ‘new’ language, but rather has provided this document to aid the user in some of the new ways RIDE and VAB related products may be used. Also we wanted to give a name, or handle (OORVL), to the ‘writing an application in RIDE’ misnomer. It does, however, allow for some interesting new ways to create algorithms, especially for real-time DSP applications!

Introduction to OORVL

Programming

An overview of the OORVL Programming Language

OORVL, **Object Oriented Real-time Visual Language**, is a graphical language used for designing algorithms for target processors (typically DSP processors). The uniqueness of this graphical language is seen in that OORVL supports target devices (or processors) in a device-independent fashion, which translates into a broad-based method of algorithm development. Similar to a C cross compiler, algorithms may be developed for a number of targets, including industry standard DSP boards which may already be in the marketplace or one-of-a-kind custom target hardware developed by the end-user. In fact, just as DSP chip simulators are used in conjunction with textual-based code development, they also may be used along with code developed using OORVL. This language can be used together with conventional standard DSP C compilers, assemblers, and linkers, or may be used as a stand-alone platform for development of algorithms. This documentation is designed to present an introduction to OORVL application development filled with examples of the most common language elements and coding constructs that software engineers working with OORVL are likely to find useful.

With OORVL, the user is able to create a DSP algorithm from a graphical design, or block diagram approach. Using component based functions, this graphical design is then used to produce DSP object code directly within the environment – no C compiler or assembler required. Since OORVL is truly device independent, the user must select the appropriate target (or driver if real-time hardware is used) to ensure that the correct machine code and interface is used when ‘compiling’ the source code (block diagram). Remember, the OORVL environment **is** the compiler/linker, capable of producing an executable object file in machine code directly.

Note: If the target is actually a physical DSP board within a PC (and supported by appropriate software application, driver), it is possible to actually run directly on the target hardware. In this case, components (PC-based components) are typically available which run on the host platform (PC) and interoperate with target side components (DSP-based components). To support this, the data must be moved between the Host and the Target via special upload/download block components. Typical examples of these types of components would be displays, file read/write components, and those components which literally run on the PC (PC does all processing for that function).

Before you begin

Prior to designing with OORVL, it is important to remember that OORVL **represents a completely new level of algorithm development**; as such it affords a tremendous benefit to users who embrace the new technology. But beware – just as many engineers are hesitant to change even their text editors to migrate to advanced textual code development tools (i.e., using a DOS-based text editor to develop Windows applications instead of an IDE such as Microsoft’s Visual Studio), there may be a tendency to start using OORVL without giving consideration to the fact that the end-user has never written code of this nature before! Just as an engineer would likely need to learn a new textual language (i.e., Assembler, C, Pascal, Modula, etc.) prior to writing code with that language, there is some rethinking and learning involved in becoming proficient using OORVL for algorithm development.

Note: Although someone experienced in algorithm development using textual based languages (i.e., C or Assembly) might easily comment that a particular algorithm is “much easier done just writing some C”, keep in mind that one of the advantages of a graphical language is that someone NOT experienced in textual languages could actually perform the design, or pick up a design which was developed by another user.

Additionally, for certain applications (such as fixed-point), a standard textual language such as C, may not have the direct support required for the application to be accomplished efficiently enough to run real-time (i.e., there is no standard C variable and associated operations for direct fixed-point arithmetic); in this case, a lower-level textual language (assembly) is typically employed, with an associated increase in learning curve and a reduction in readability and maintainability.

The good news is that it is relatively easy to learn to use OORVL and that using it has a great number of advantages before, during, and after the project is completed. The fact that this language is **self-documenting** (as a block diagram) is only one of the important features of OORVL. The true ‘**Black Box**’ **design approach** using **standard components** which may be exchanged, etc., is also of great importance in ensuring **modular programming practices** of the person designing algorithms. Additionally, the **open software architecture** is important for extensibility by the user. A good first step is to read through this section of this manual noting the manner in which code is developed using OORVL.

Some of the sections that will be discussed include the general coding strategy, fundamental language elements, control-flow constructions, variable declaration, and other information required for well-structured algorithm development using OORVL.

General Coding Strategy

Development of OORVL algorithms is similar to C in that variables, operators, expressions, and functions are used to create other functions, and/or the overall program. The difference is that each of these constructs is expressed graphically as opposed to textually. Although initially quite different than software coding textually, the advantage of using a language which is similar in nature to how an engineer typically constructs and conceptualizes (i.e., a block diagram) has definite advantages during the course of a project’s development.

This graphical ‘block diagram’ language is **especially productive for those with limited experience and expertise in assembly, C and other textual languages** (it is interesting to note, however, that there is even a tool available for producing standard ANSI C source code from OORVL designs!). In addition, **the self-documenting** nature of the language is an important consideration, and allows for a **quick learning curve** by other engineers, or by the same engineer at a later date. The ability of OORVL to support a number of **different DSP targets** allows for **relatively painless migration** from one processor family to another.

With its **Open Software Architecture**, language extensibility and creation of new components is very easily accomplished. And with a small amount of initial learning, well-written, easy-to-follow, **modular algorithms** can be designed very quickly by the user.

Language Elements

Designing with OORVL involves selecting components from a set of graphical components, or function blocks, and then graphically connecting these components to show the data flow relationships among them. The execution sequence, or control flow, of these components is then assigned (typically automatically, but this may also be accomplished manually). OORVL's base language elements consist of two main classes of blocks, or components – Native Components, and Vector Components.

Native Components

Native Components are very efficient language elements which operate on data values which are inherently single point data (not frame-based, or vector-based). One may think of these language elements as being an 'inline' type of language element. The intent of the Native Component language elements are to provide the necessary fundamental functions, operators, etc. needed for language extension using hierarchy.

Vector Components

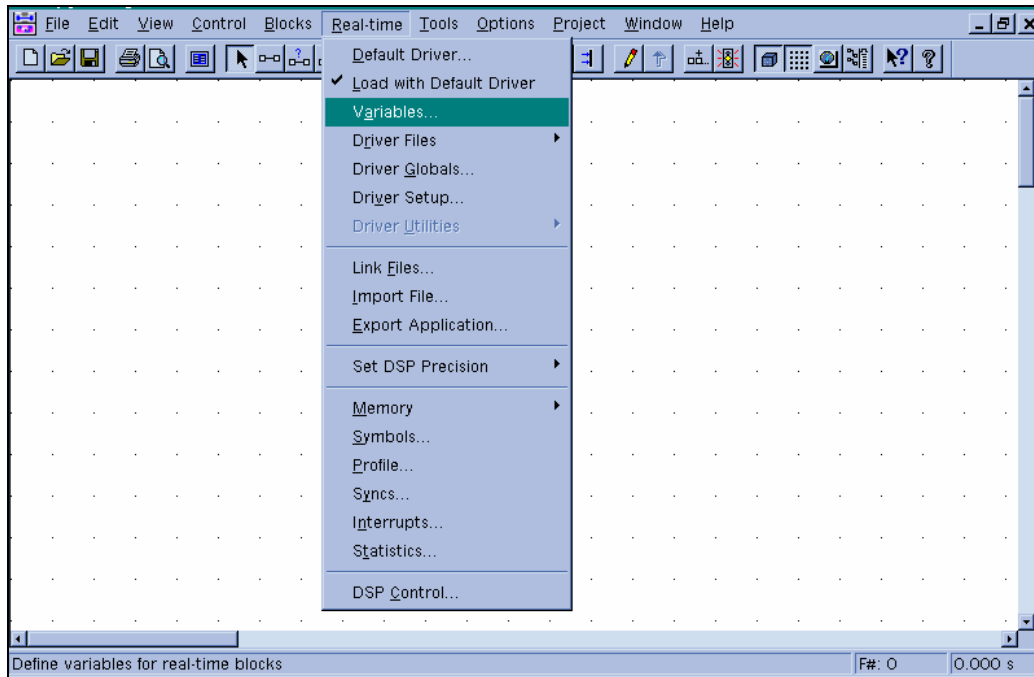
Vector Components are language elements which are typically of a larger grain size and can inherently process frame-based, or vector-based data. These Vector Components may also be used on single point data, but the efficiency of Native Components is better with this type of data. One may think of these language elements as being a 'called subroutine' type of language element. With the **Open Software Architecture**, Vector Components allow for easy expansion using classic textual language software tools, since the user may create their own Vector Component using standard C Compilers (Assemblers).

Getting Started With OORVL

It is probably a good idea to study and work with some examples before beginning your own algorithm development with OORVL. In the next section, we will attempt to introduce several basic examples that demonstrate some of the language constructs of OORVL. Once you understand these examples, you should begin to start trying some of your own graphical software algorithm designs.

Variables and Arithmetic Expressions

OORVL allows for variable declaration and for arithmetic expressions involving variables. In order for variables to be used, they must first be declared and initialized. This is accomplished with the 'Variable' menu option under the Real-time Menu as shown in the figure below.



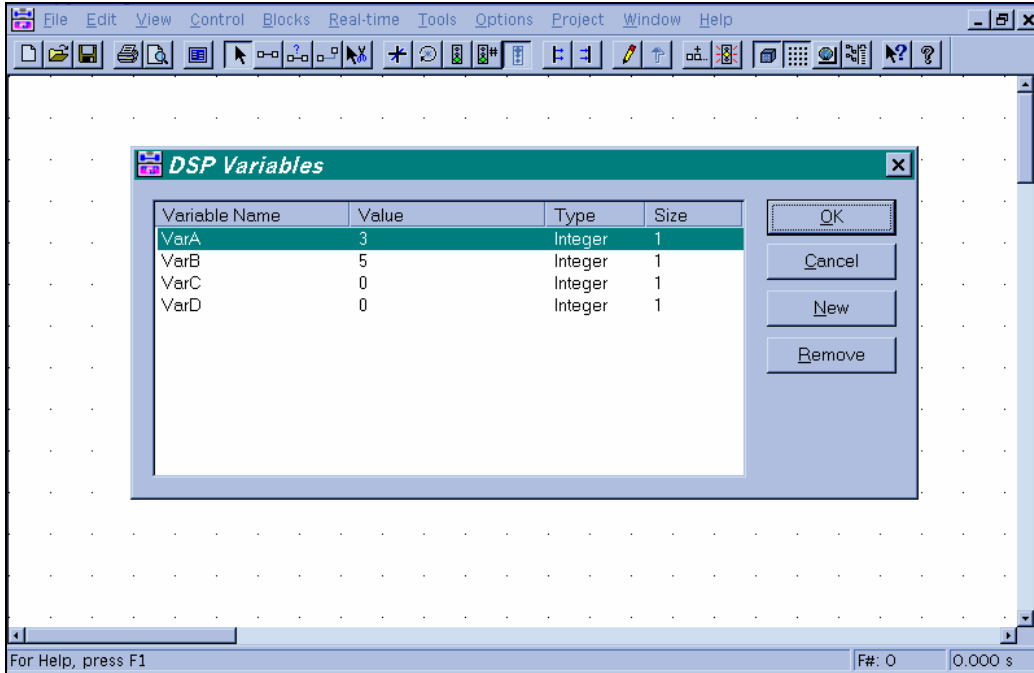
Variable Selection Menu Choice

At the time a new variable is declared, a default initialization value is given. This step accomplishes both the declaration and the initialization of the variable. Note that the variable is always initialized to something, thereby reducing (or eliminating entirely) the possibility of errors due to uninitialized variables (historically, a very large problem with some textual languages). In addition, the data type for the variable (if multiple data types are allowed by the current target) is selected at this time.

The following is a variable declaration and initialization which is given in a textual format; it accomplishes the variable data typing (int), the variable naming (VarA, etc.), and the initialization of the variables.

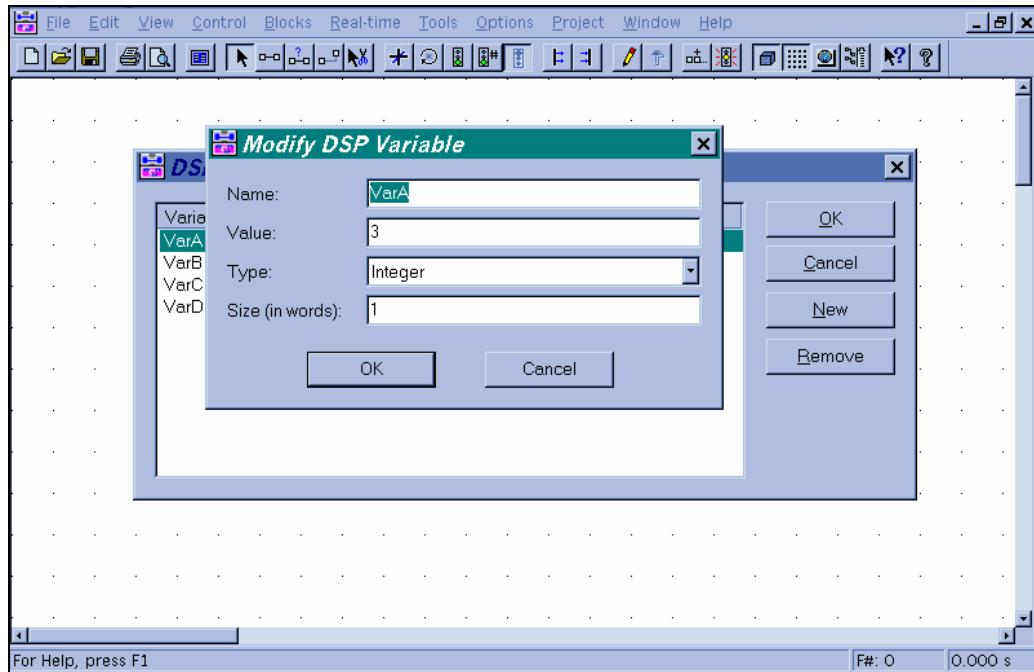
```
int VarA = 3;
int VarB = 5;
int VarC = 0;
int VarD = 0;
```

This same variable declaration, data typing, and subsequent initialization is accomplished in OORVL as follows:



Variable Declaration

The attributes of a variable may be modified (by double-clicking on the variable) as shown in the following figure:



Variable Attributes

The If Statement

The output of a block component may be used to conditionally control whether another component (or group of components) is executed. Very similar to a C expression 'if(Value)' if the output value of a component (Value) is TRUE (non-zero), the component (or group of components) which has been conditionally connected will be allowed to execute. Otherwise, the component (or group of components) will not be allowed to execute.

In order to achieve this conditional connection, the conditional connect mode must be used; when a connection of this nature is made, the connection will show up on the top side of the connected component (or components), and will typically be displayed using a red line. If the component being connected to is a hierarchical component, the conditional connection may be selected to be something other than the standard 'IF' (IF, WHILE, or FOR). For connections made to non-hierarchical block components, the conditional connection is always an 'IF' type.

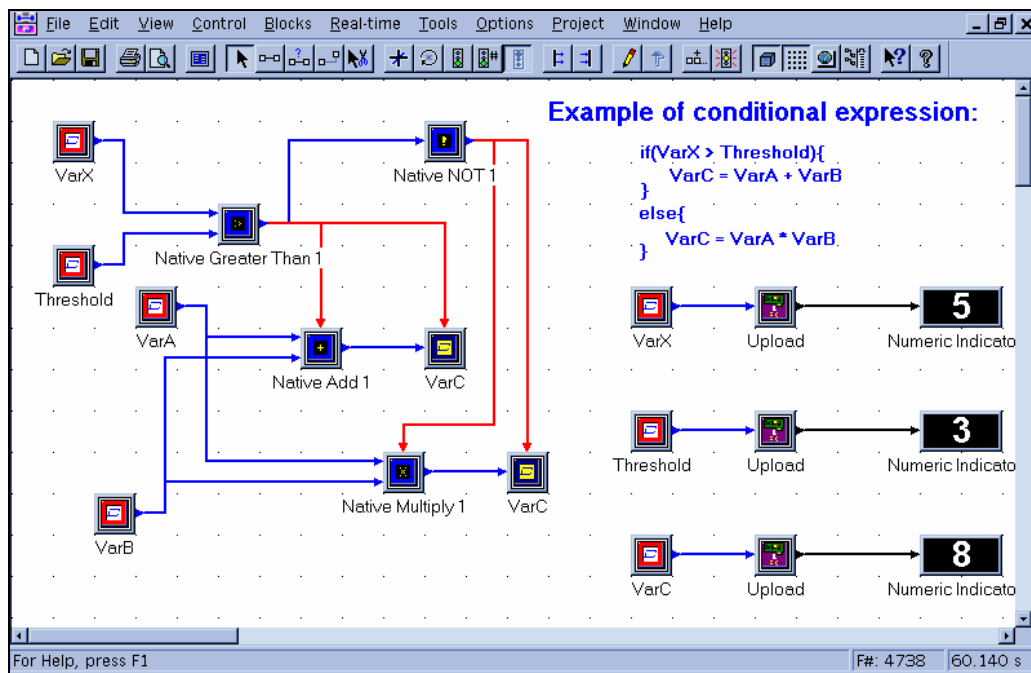
The following example is of a typical 'IF' construct using a textual language:

```

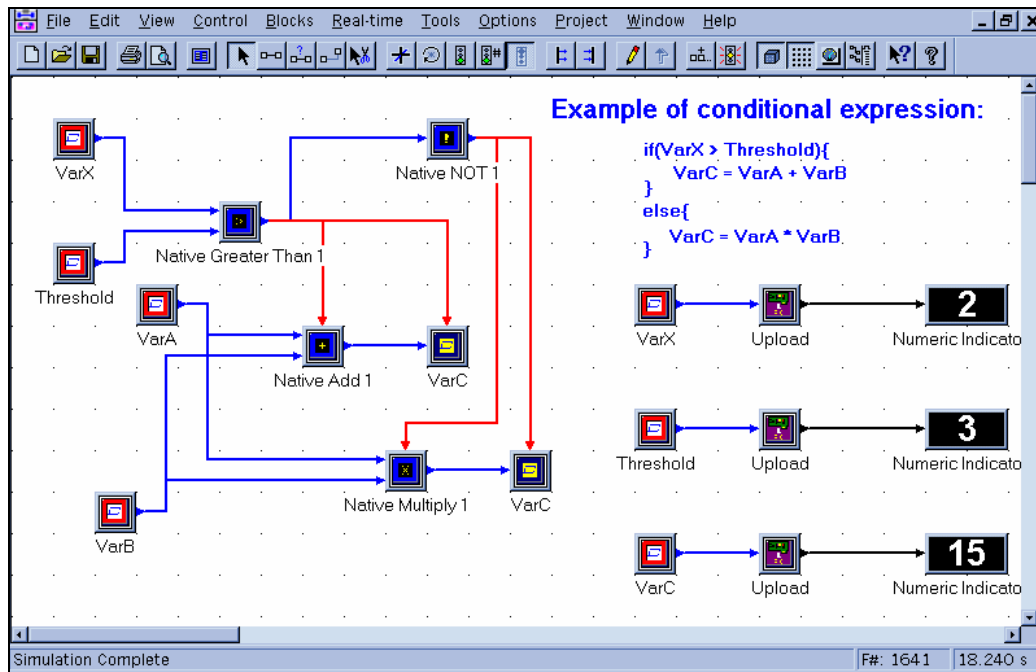
If (VarX > Threshold) {
  VarC = VarA + VarB;
}
else{
  VarC = VarA * VarB;
}

```

This same logic is now shown using OORVL:



'IF' construct where VarX > Threshold



'IF' construct where VarX < Threshold

The For Statement

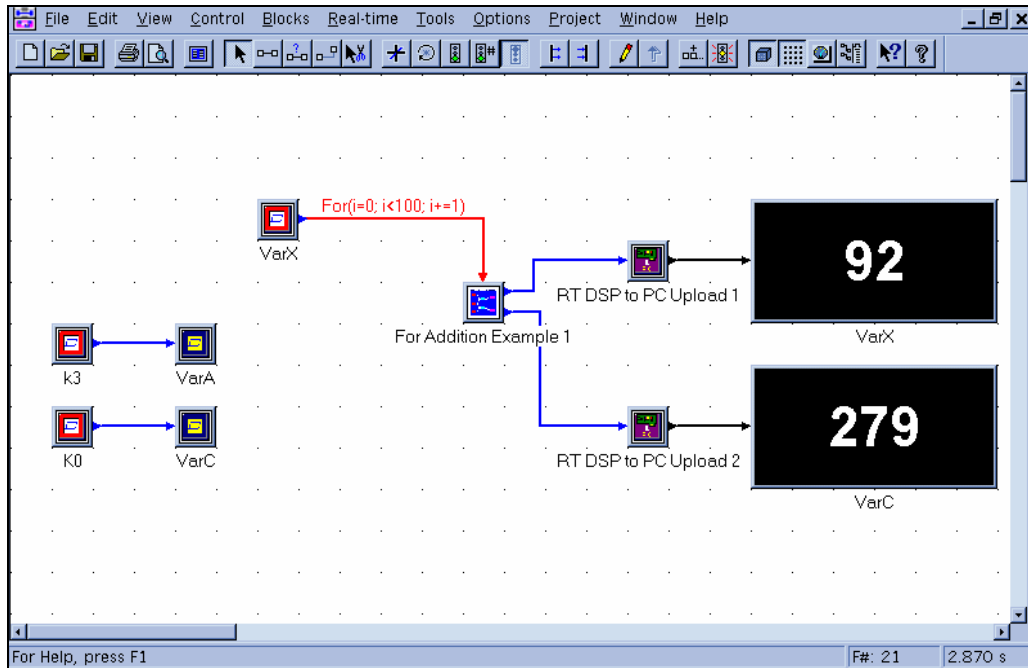
The common 'FOR' structure is convenient for many algorithm applications and allows for direct loop constructs, using a governing loop variable that is implicitly controlled by the loop construct itself. Using a standard textual approach, a typical 'FOR' structure is given as follows:

```

VarA = 3;
VarC = 0;
For(VarX=0; VarX<100; VarX++){
  VarC = VarC + VarA;
}

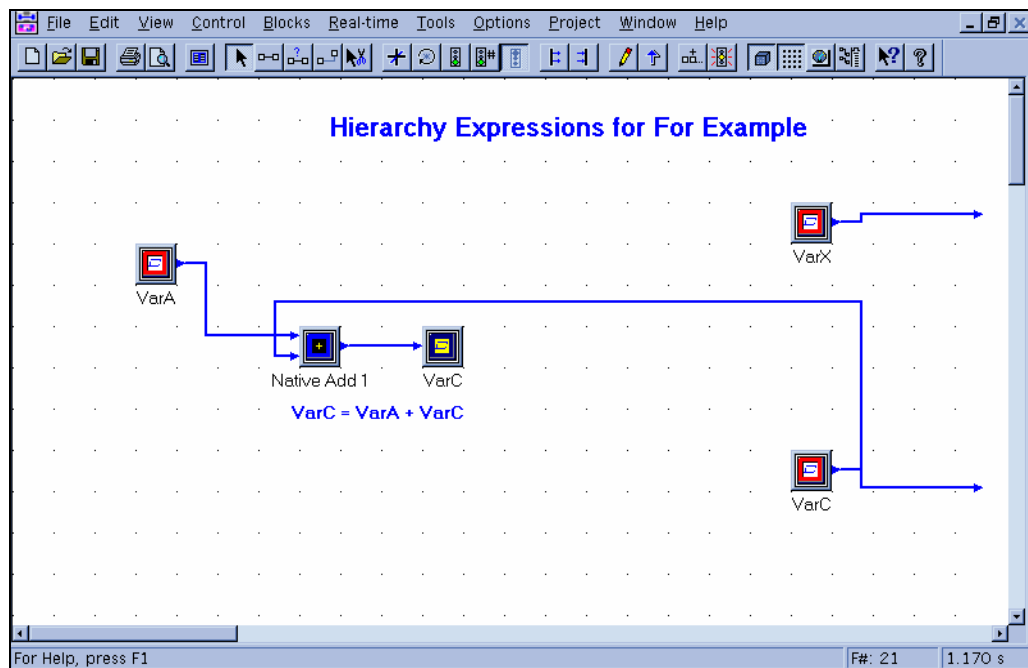
```

Using hierarchy to contain the logic which is to be performed within the 'FOR' loop, the same looped expression above would be achieved as shown below:



Example 'FOR' construct (top level)

Notice how the logic which is to be executed a certain number of times is located within a single hierarchy component connected at its top with a conditional connection (red line), thus enabling a quick identification (and isolation) of the looped code. The code which is being executed within the hierarchy component which is responsible for actually 'doing the work' is as follows:



Example 'FOR' construct (hierarchy component)

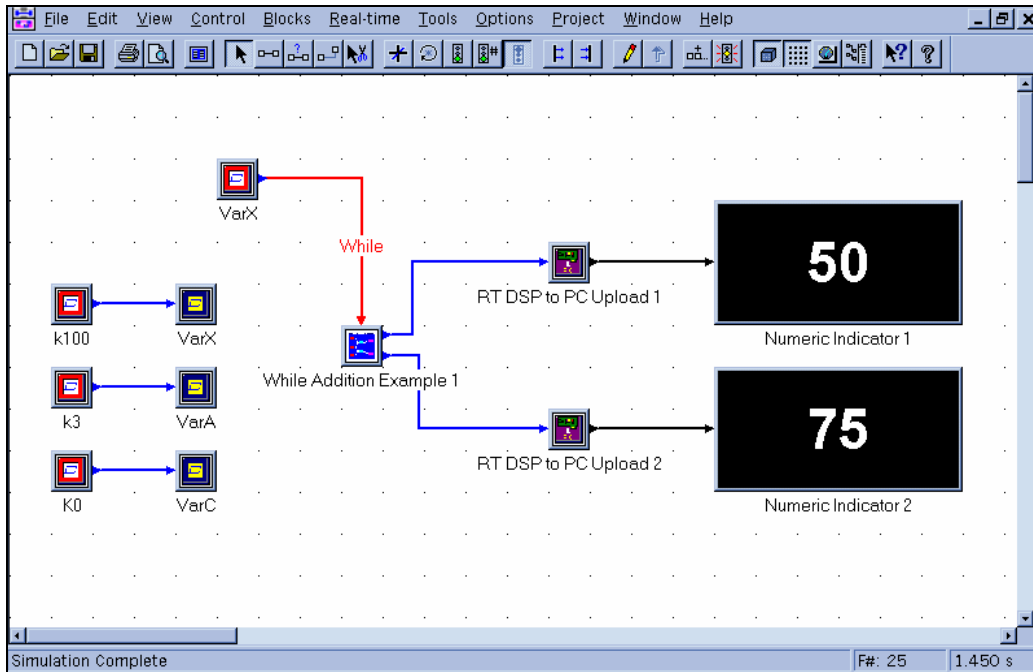
Note that the governing variable, VarX, is 'passed' to the conditionally controlled hierarchy component, where it may be used within any expression, etc. In this example, it is simply 'passed' back out the top output channel. Control of the entire algorithm may be thought of as being 'passed' to the hierarchy component, and maintained there while the 'FOR' controlling variable, VarX, cycles from its initial condition to its terminal condition using the given indexing; once terminal condition has been reached, control is 'passed' out of the hierarchy component to the next component on the above level (top level, in this example).

The While Statement

In some cases, a loop construct of a different nature than the 'FOR' loop construct is called for; in certain applications, a loop construct which is governed by a loop variable **not controlled** by the loop itself is required. This construct in textual software coding is typically called a 'WHILE' loop construct. Note that the looping control variable is contained within the executed segment of code within the loop itself. This is sometimes advantageous, but may also easily lead to 'endless loops' if programmed incorrectly (loop control variable not ever reaching terminal condition!). A typical example of a textual-based 'WHILE' example is something like this:

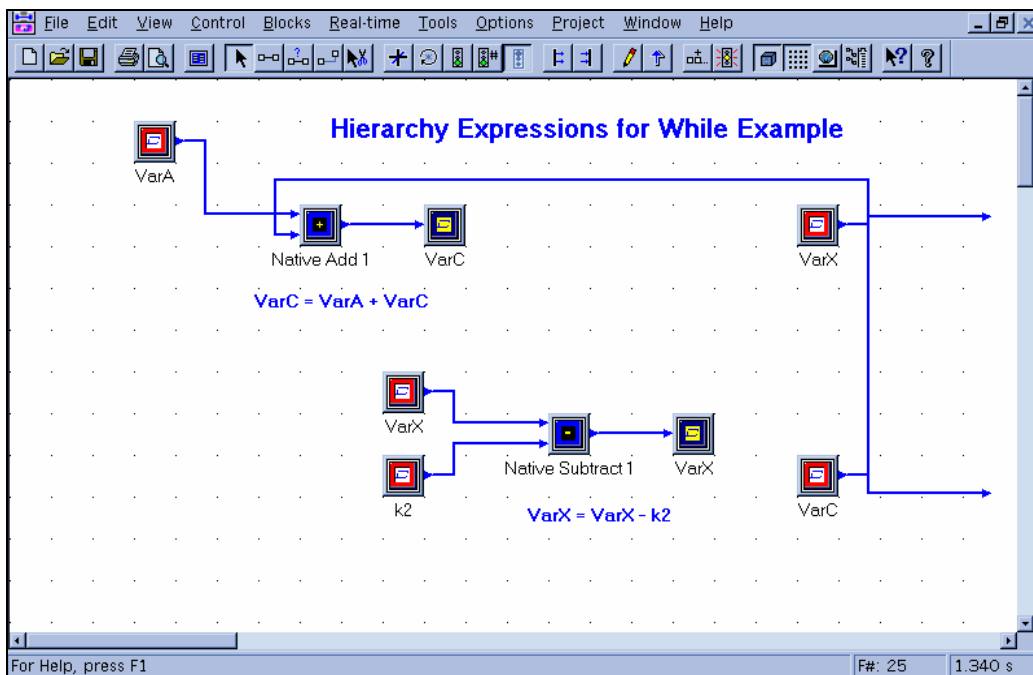
```
VarX = 100;  
VarA = 3;  
VarC = 0;  
  
While(VarX) {  
    VarC = VarC + VarA;  
    VarX = VarX - 2;  
}
```

The loop control variable in this example is VarX; note how it is decremented from 100 down to 0 (the terminal condition). With OORVL, the same example would look like this:



Example 'WHILE' construct (top level)

Notice again how the logic which is to be executed a certain number of times is located within a single hierarchy component connected at its top with a conditional connection (red line), again enabling a quick identification (and isolation) of the looped code. The code which is being executed within the hierarchy component which is responsible for actually 'doing the work' in this 'WHILE' example is as follows:



Example 'WHILE' construct (hierarchy component)

Note that the governing variable, VarX, is modified by its own expression to ensure it reaches a terminal condition; if this were not implemented, control would be passed to this hierarchy component and stay there indefinitely in an infinite loop (just as in a textual case). If the loop control variable is modified correctly, control of the entire algorithm may be thought of as being 'passed' to the hierarchy component, and maintained there until the 'WHILE' controlling variable, VarX, cycles from its initial condition to its terminal condition using the given expression (here, $\text{VarX} = \text{VarX} - 2$); once terminal condition has been reached, control is 'passed' out of the hierarchy component to the next component on the above level (top level, in this example).

Variable Names

OORVL allows very free-form expression when naming variables. Variable names may include any combination of characters (both upper and lower case), numbers, and even spaces. Suggested variable naming would be of generalized Title Case for readability.

Data Types

OORVL allows for a variety of data types for variables, depending upon the target processor. Fixed point processor targets normally allow a fixed-point variable type (Q0 to QN), and sometimes a floating point type as well, while floating point processor targets (those capable of hardware floating point operations) are generally limited to integer and floating point data types. The data type is selected at the time of variable declaration.

Constants

In OORVL, a constant is simple a variable initialized to a particular constant value. Though not necessary, quite often the variable nomenclature of 'kx' is used, where x is the constant being set. For example, a variable of k3 would typically imply a constant of k=3. If the constant is a floating point number, the nomenclature 'kfx' would typically be used.

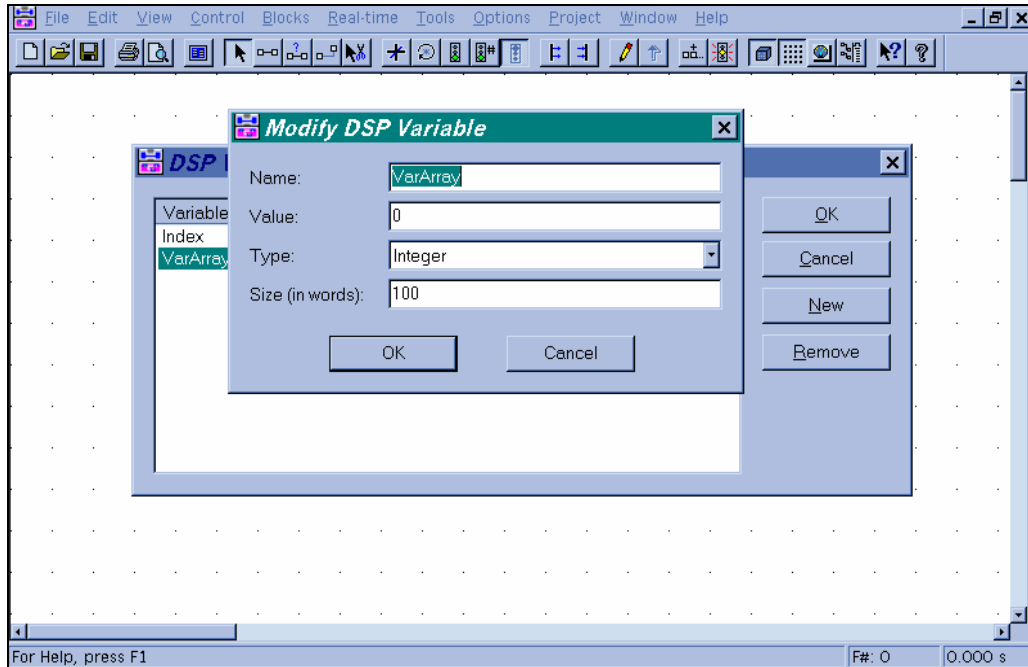
Arrays

Arrays are variables containing a number of values and are allowed within OORVL. The manner of creating, or declaring arrays is the same as that for declaring a single element variable. The only difference is that the 'size' attribute for the variable is defined as something greater than 1. Obviously, the amount of data memory required for the variable is equal to the number of words of memory required for a single variable (of that particular data type) multiplied by the size (or number of) of the array. In order to access the array, the use of a 'pointer' concept is required; in OORVL, the components in the 'Variable Operators' group are used for this purpose.

An example of a textual based variable array declaration might be something like this:

```
int VarArray[100]={0..};
```

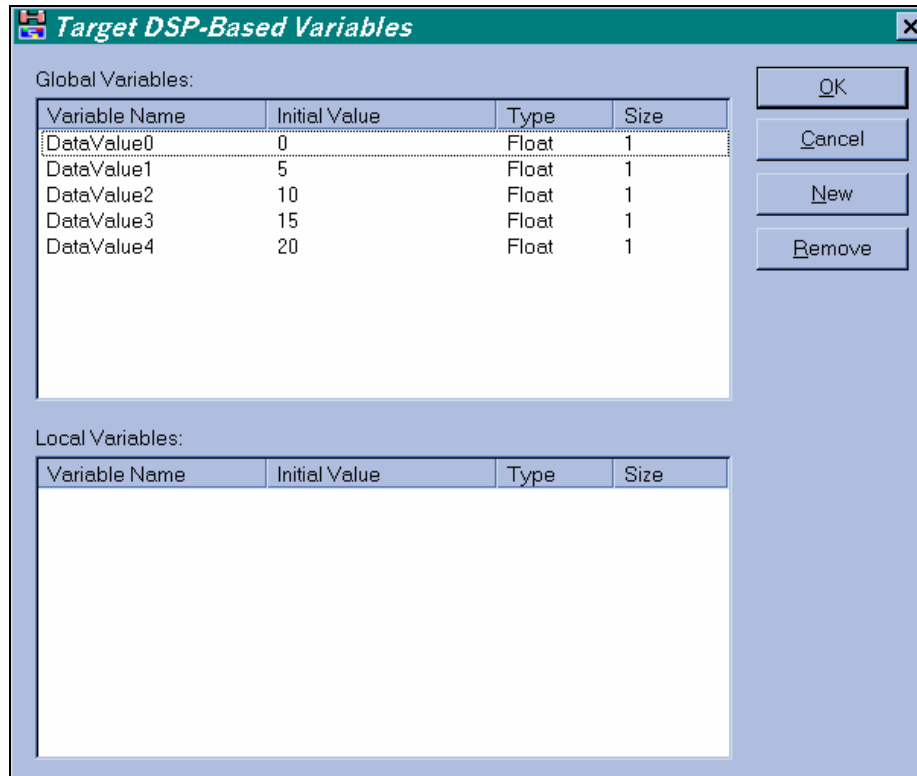
This same array initialization would be performed as follows using OORVL:



Variable Array Declaration and Attributes

Union of Variables

In some cases placing variables physically next to each other in memory to allow for an indexed method of using the variables, related to a base variable is important. OORVL currently supports this in the following manner; if Target Variables are created and named such that they are alphabetically ordered, their physical memory allocation will also be ordered. For example, in the following figure, the variables are all alphabetically in order, and the resulting physical memory layout will be guaranteed to be sequential. Once the variables are sequential in physical memory, the variable read functions, **variable read with offset** and **read element at**, will allow access to all variables using the base variable name, 'DataValue0'.



Union of Variables, alphabetical means sequential in memory

Functions

Functions are typically implemented as either Vector Components, or as Hierarchical Components (which are made up of Native Components, and/or Vector Components, and/or Hierarchical Components). A function may then be thought of as simply another component. As such, it inherently has pointers to its input data (represented as graphical line connections on the left side of the component), pointers to its output data (represented as graphical line connections on the right side of the component), and an associated (instantiated, also) set of parameters for the function, which are accessible by the user via a right click of the mouse on the component (or double-clicking on the component).

Types, Operators, and Expressions

Arithmetic Operators

There are five standard arithmetic operators: **add**, **subtract**, **multiply**, **divide**, and the **modulus** operator. Additionally, for fixed point operations, a fixed-point multiply is also available (the general add operator can handle fixed point math inherently).

Relational and Logical Operators

There are four different relational operators: **less than**, **greater than**, **less than or equal to**, and **greater than or equal to**. There are also two equality operators: **equal**, and **not equal**. All of these operators are found in the Relational Operators group in the Native Library.

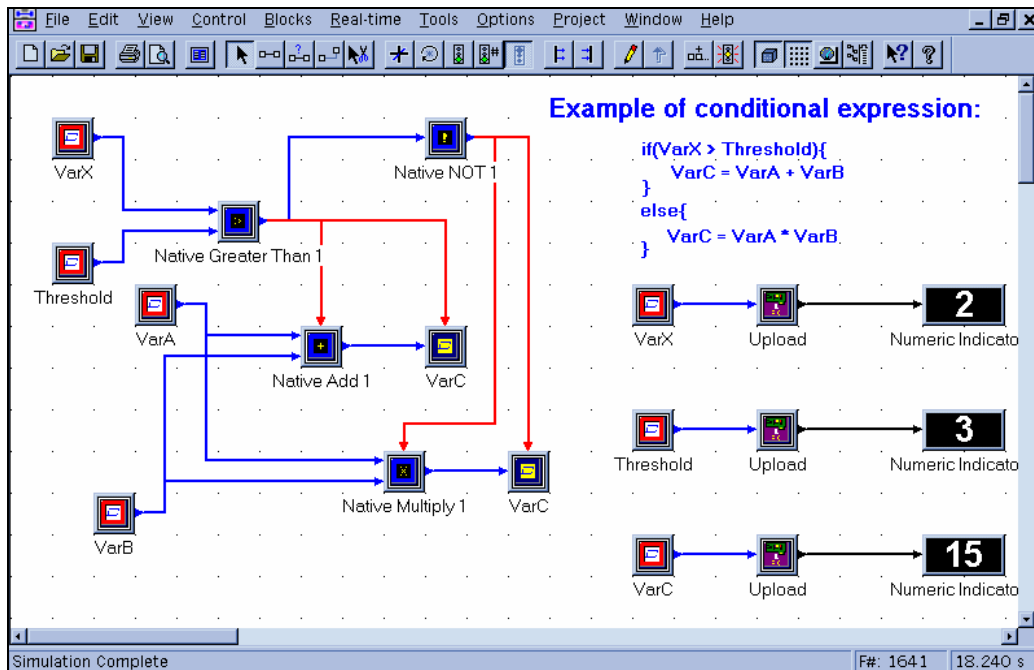
For all of these operators, there are two inputs and one output. The operators' input channel 0 is always compared to input channel 1. The result (a TRUE or FALSE value, 1 or 0), is produced at the output channel of the function. In C, the statement:

$$x < y$$

is represented in the LESS THAN function as

$$\text{input channel 0} < \text{input channel 1}$$

The resulting output (on channel 0) of the LESS THAN function is then 1 or 0, based on if the condition is met.



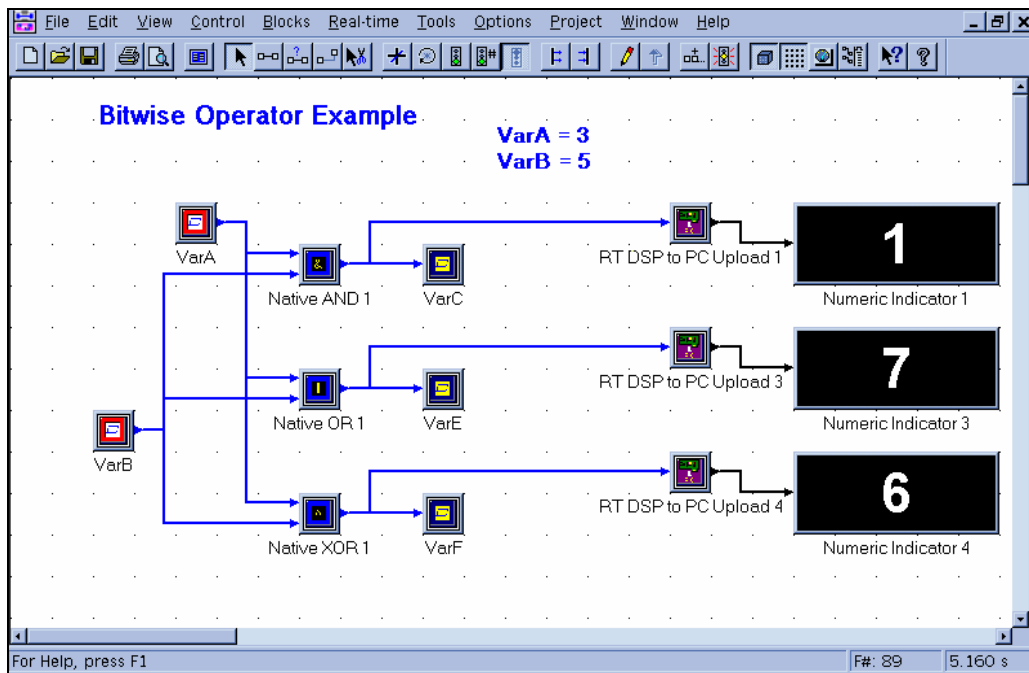
Conditional Example

In addition, there are three logical operators: **and**, **or**, and **not**, which are found in the Logical Functions group of components in the Native Library. The outputs of these functions are always either TRUE (1), or FALSE (0), based upon the logical operation on the input data.

Note: These should not be confused with the Bitwise operators, which operate on a bitwise-basis over the entire expression. Thinking about standard C and its operators (& vs &&, | vs ||, ! vs !!) may help in remembering this important distinction. Remember the difference between LOGICAL and BITWISE.

Bitwise Operators

There are six standard bitwise operators: **and**, **not**, **or**, **shift left**, **shift right**, and the **xor** operator. These operators operate on integer type data only (not floating point data), and are often useful along with conditional and/or relational language elements. Obviously, additional bitwise operators such as NAND and NOR may be made by simply combining some of the standard bitwise operators (AND and NOT for a NAND, OR and NOT for a NOR).



Example of Bitwise Operators

In the above example, VarA = 3 and VarB = 5. The results of the AND, OR, and XOR functions are correctly shown in the displays to the right as 1, 7, and 6, respectively.

Type Conversions

Data typing is supported by OORVL, but may be limited depending upon the target processor. Fixed point processors normally allow either integer (including fixed-point variable types for certain processors) or floating point operations, while floating point processors (those capable of hardware floating point operations) are generally limited to

integer and floating point data types. There exist Native Blocks for typing data either from INT to FLOAT, or from FLOAT to INT. It is important when mixing floating point and fixed (integer) arithmetic to ensure that the data is in the proper format for the function which operates on it, otherwise, unexpected results may occur. For some fixed-point processors, there exist Q-point conversion functions to allow variables to be converted from a particular Q-point to another Q-point.

Assignment Operators and Expressions

With a graphical, data-flow language such as OORVL, variable assignment and expressions are quite different from textual language representations. The assignment operator in C,

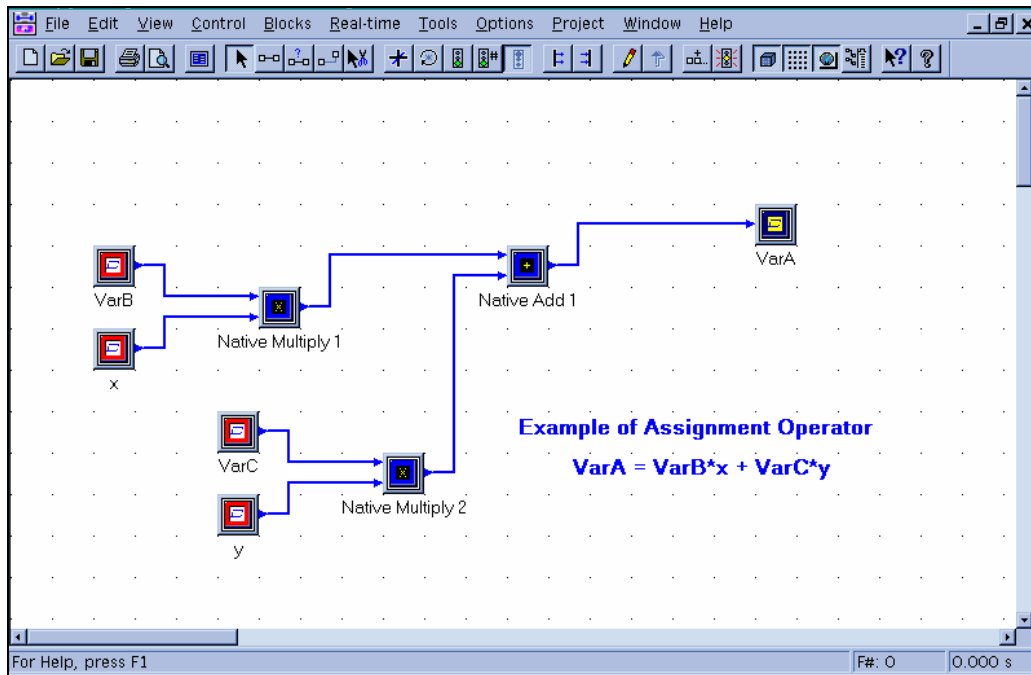
$$x = y$$

is actually described graphically as the following:

To form an expression, consisting of operators and variables such as is done in C,

$$\text{VarA} = \text{VarB} * x + \text{VarC} * y$$

would be described graphically as:



Example of Assignment Operator

Order of Operations

Unlike traditional textual based programs, the order of operations in an OORVL program is not based on operator precedence or parenthesis, but instead its based on the data flow of

the diagram. There are no nested sets of parenthesis to confuse the programmer. The order the functions are called in depends on how the diagram is laid out. The use of parenthesis is therefore handled inherently by the programmer as the diagram is drawn. For example, if the output of a multiply operation acts as input to an addition operation (or vice versa), then the multiplication operation must occur before the addition operation is allowed to be performed. This behavior is built into the compile sequence that is performed when the diagram is compiled. The user can optionally override the default compile order by manually setting the order in which the components are run.

Control Flow

Control flow is allowed within OORVL using conditional connections which allow for IF, WHILE, and FOR constructs to be created. For simple 'IF' type of connections, simply connect the output of a component, using a conditional connection mode, to another component (or group of components) which is desired to be controlled. In order to additionally be able to implement 'WHILE' and/or 'FOR' type of connections along with the 'IF' type of connection, the component (or components) desired to be controlled must reside within a Hierarchical Component. Using the combination of 'IF', 'WHILE', and 'FOR' conditional connections, a variety of control flow cases may be programmed graphically.

Functions and Program Structure

OORVL allows large tasks to be broken into a set of smaller related tasks using functions. Conversely, this allows a complex design to be developed using a set of smaller functions, each of which may in turn be composed of yet smaller functions. This ability to create functions is an important design feature of OORVL, and there are two main methods of creating functions – first, graphically using block components along with the concept of hierarchy (**Hierarchical Components**), and second, through use of an external textual language (C cross compiler or assembler), thus allowing functions (**Native or Vector Components**) to be creating using C or assembly as the base language.

<p>NOTE: Keep in mind, that to the user, the components all look and act the same – it is not important to understand what type of component is being used; it will look and perform the same as any other component! This makes it very easy for people of many different skill levels and expertise to effectively make use of OORVL.</p>
--

OORVL Hierarchical Components

Functions (or components) created graphically from other functions (or components) are called Hierarchical Components. Normally, these should make use of the most efficient set of Components to accomplish the function's specific task. Native Components, and Hierarchical Components made from Native Components, are generally used to create new Hierarchical Components to improve the efficiency of the design, both in memory space and

execution speed. This is not a limitation, however; all block components may be used to create a Hierarchical Component, including Vector Components and other Hierarchical Components (or even a combination of all of these component types).

Native and Vector Components

Functions may be created externally using C or Assembly and are referred to as Native or Vector Components depending on whether they are designed to handle single point or vector based data. The included Block Wizard assists in creating both types of components. It produces two sets of files, the PC-based set of files to create the graphical icon associated with the user interface of OORVL, and the DSP-based set of files used to create the actual object code for the specific target DSP chip. See the Block Wizard reference and associated examples for more detailed information.

Pointers and Arrays

Each graphical data flow lines represent basically a pointer to a 'chunk' of memory. This 'chunk' may be a single valued variable, or may be a sized array of some set length. In order to work with variable arrays and pointers, the components in the 'Variable Operators' group are used. The following Native Components are included in this group, and are described below:

Read Element At

This function reads the element at the specified address and offset of a variable array. This block outputs a single value located at the address, specified in the first input, plus the offset, specified in the second input. Most commonly, this block is used to access an element within an array of data where the array address is connected to the first input and the array index is connected to the second element. Note - the inputs and outputs of this block are treated as scalar values and not as vectors.

Variable Read

This function provides the value of a particular real-time global variable. The current value of the variable is available from this block, and when the block is running, changing this value causes an instantaneous one-time download of the new value. The initial value of the variable is also accessible from this block. Note - the outputs of this block are treated as scalar values and not as vectors.

Variable Read with Offset

This function provides the value of a real-time global variable using an offset from the base of the variable, or variable array. This block outputs the element at the offset, specified on the input channel, from the initial starting address of the variable array. Note - the inputs and outputs of this block are treated as scalar values and not as vectors.

Variable Write

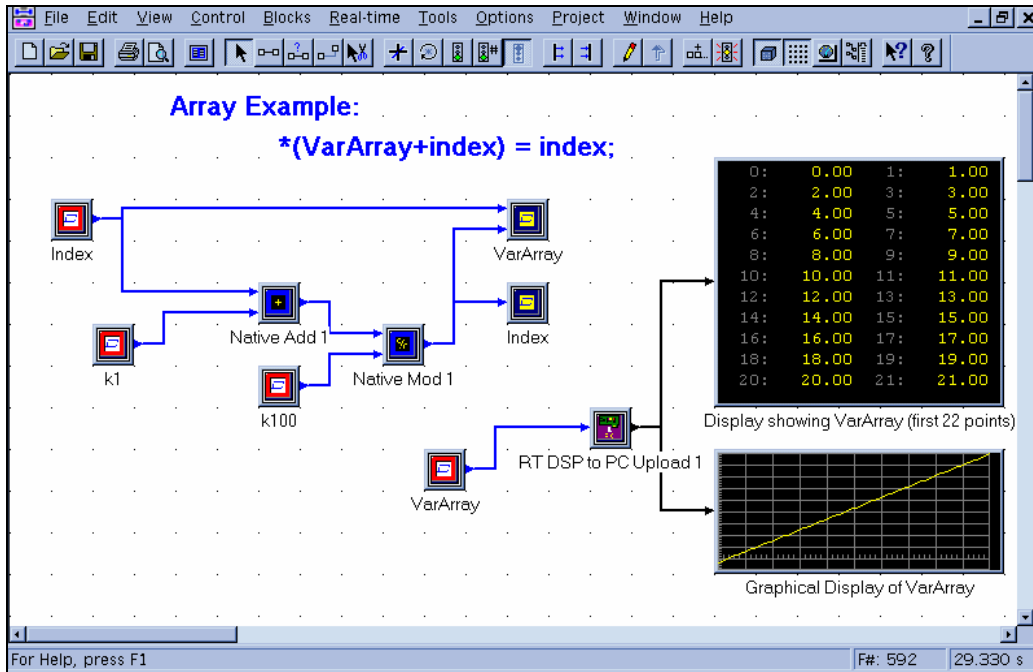
This function stores the input value in the variable specified. This block can be thought of as an assignment operator, as it replaces the current value of the variable with whatever is connected to the input channel. Note - the inputs of this block are treated as scalar values and not as vectors.

Variable Write with Offset

This function stores the value, (contained on the first input channel), at the offset, (specified by the second input channel), relative to the starting address of the variable array specified. Most often, this block is used to write values to different locations within an array of values. Note - the inputs of this block are treated as scalar values and not as vectors.

Write Element At

This function stores the value, at the first input, at the address created from the second and third input channels. The second input is a base address and the third input is an offset from the base address. Note - the inputs of this block are treated as scalar values and not as vectors.



Example of using arrays

Summary

Component-based design will play an important part of the future for DSP and related engineering areas, as well as software architecture design in general. OORVL represents a modern approach to minimizing the learning curve, improving the maintainability, and to solving some of the problems presented by classical textual-based language approaches. We hope you have success with this powerful new vehicle for next-generation algorithm design, and also hope you have fun in the process!

Technical Support and Professional Services

Visit the following sections of the National Instruments Web site at ni.com for technical support and professional services:

- **Support**—Online technical support resources at ni.com/support include the following:
 - **Self-Help Resources**—For answers and solutions, visit the award-winning National Instruments Web site for software drivers and updates, a searchable KnowledgeBase, product manuals, step-by-step troubleshooting wizards, thousands of example programs, tutorials, application notes, instrument drivers, and so on.
 - **Free Technical Support**—All registered users receive free Basic Service, which includes access to hundreds of Application Engineers worldwide in the NI Developer Exchange at ni.com/exchange. National Instruments Application Engineers make sure every question receives an answer.

For information about other technical support options in your area, visit ni.com/services or contact your local office at ni.com/contact.

- **Training and Certification**—Visit ni.com/training for self-paced training, eLearning virtual classrooms, interactive CDs, and Certification program information. You also can register for instructor-led, hands-on courses at locations around the world.
- **System Integration**—If you have time constraints, limited in-house technical resources, or other project challenges, National Instruments Alliance Partner members can help. To learn more, call your local NI office or visit ni.com/alliance.

If you searched ni.com and could not find the answers you need, contact your local office or NI corporate headquarters. Phone numbers for our worldwide offices are listed at the front of this manual. You also can visit the Worldwide Offices section of ni.com/niglobal to access the branch office Web sites, which provide up-to-date contact information, support phone numbers, email addresses, and current events.