Star Bridge Systems Online Help

TABLE OF CONTENTS

VIVA 2.2 AND CORELIB INSTALLATION PROCEDURES

Viva 2.2 Install

To install Viva 2.2, perform the following steps:

1. Remove prior versions of Viva.

   Note: Depending on which version of the Windows operating system running on your machine, these steps may slightly vary.

   - Click the Windows **Start** button, and from the Control Panel menu, choose **Add or Remove Programs**.
   - In the Add or Remove Programs window, scroll through the *Currently installed programs* list and select Viva.
   - Click **Remove**.

   Note: Removing prior versions of Viva does not remove saved projects and sheets stored in the Viva directory, or other directories.

2. Insert the Viva 2.2 CD to install the application.
3. Follow directions indicated in the InstallShield.
   - If CD install does not autoplay, browse the CD to locate Setup.exe in the root directory. Run Setup.exe, and follow InstallShield directions.

CoreLib_10x Install

To install Viva 2.2 CoreLib_10x, perform the following steps:

Note: It is not necessary to delete earlier editions of CoreLib. CoreLib installations do not overwrite earlier CoreLib editions. Please review the section on Library and System Management before installing a new version of CoreLib.

1. Insert the CoreLib_10x CD.
2. Follow directions indicated in the InstallShield.
   - If CD install does not autoplay, browse the CD to locate Setup.exe in the root directory. Run Setup.exe, and follow InstallShield directions.

KNOWN ISSUES

- Creating, closing or opening a project or opening a system while a program is compiling may cause Viva to crash.
  Workaround: Stop the compiler before opening or creating Project or System files.

- Com Object Tree (MainForm) Issues a GPF when user attempts to delete a top-level node. Com Class Tree crashes Viva if you drag a top-level node onto I2adlView.
  Workaround: Manipulation of top level nodes should be avoided.

- The Object Browser accepts ".oca" files from VisualBasic, but should only accept ".ocx" files. Selecting ".oca" files mistakenly included in the Browser results in no effect.
  Workaround: The user should be careful to select only the ".ocx" library reference, not the ".oca" reference files.

- Opening a project already in use crashes Viva.
  Workaround: Do not attempt to open a Viva file that is currently in use by Viva or other applications.

- Clicking on the Icon Editor button attempts to open MSPaint.exe, which is not at the hard-coded location in WinNT versions.
  Workaround: Copy MSPaint.exe to "C:\Program Files\Accessories\".

- When bit datasets are propagated to polymorphic objects having inputs directly feeding exposers, the primitive versions of those objects are not selected.
  Workaround: Select primitive objects on a top-level page when propagating bits to its inputs.

- Giving a control node name to more than one IO object on a sheet causes it to toss all but the first IO object when that sheet is converted to an object. This can manifest itself in a number of ways, including failure to route parts of an otherwise valid project.
  Workaround: Always ensure that control node names (Go, Done, Busy, Wait) are unique on all IO objects in a sheet.

- Changing your computer name causes Viva to fail to access the Viva.ini file.
  Workaround: If you change your computer name, create a directory with the same name under the "\documents and settings\" directory.

- When an object with no non-control nodes and no tree groups is saved to file, it is interpreted as a WIP sheet on a subsequent read-in.
  Workaround: Ensure that all objects have either a tree group or non-control nodes before saving.

- When the system name is changed using the "Save System As" option, the new system name is put into the system object tree but not into the system tree.
  Workaround: To reset the system object tree, click the I2ADL Editor Tab.

- The Synthesizer issues an error and crashes when it encounters an object assigned to a System that does not have any System Objects.
  Workaround: Never assign an object to a System that does not have System Objects, such as the base system.

- When reading a text file, Viva terminates at a comma when reading in an attribute value. This causes truncation of string constants containing commas.

Workaround: Use a mechanism other than Viva-rendered string constants for storage of persistent strings if they contain non-alpha-numeric characters.

- Compiling an input with a constant value whose DataSet is NULL, or which has NULL as a child, will cause Viva to crash.
  Workaround: Avoid using constants in conjunction with the NULL DataSet.

- Occasional file corruption occurs.
  Workaround: Save Viva files frequently. Use a versioning preference to minimize loss of work. Do not overwrite previously saved files when attempting to save a file after a GPF.

- "Ambiguous Object" warnings are occasionally issued incorrectly.
  Workaround: If project works as expected, ignore the warning.

- If Viva is closed before the changes to a new project have been saved, Viva asks if the changes should be saved. If you choose 'yes', and then cancel, Viva closes without saving your work.
  Workaround: Save Viva files before closing Viva and before switching to a different or new Viva project.

Performance Characteristics and Limitations

Most of the limitations are related to the PCI Controller core. The following is a list of some of the limitations of the hypercomputer system.

- Windows will only drive 32 bits wide on the bus. The current Controller only accepts 32 bits of data.
- No support for Initiator or Slave Burst modes.
- No communication protocol exists between the Controller and the other chips it communicates with.
- The current HC Implementor does not have a communication protocol in it: it merely polls the immediate state of the inter-chip pins connected to the Controller. The TunnelCTL object exists for a sample of a basic double-buffer communication protocol.
- The emulator is not a timing based process. It is a busy loop event dispatcher. The $Clock name holds its cycle, which may vary from several hertz to several kilohertz depending on the amount of objects being emulated.
- The PCI Controller chip will only run at 66 MHz because of the actual speed of the chip.
- A combination of the current CoreLib and the Resources.txt file will not currently automatically route things between FPGAs.

ERROR MESSAGES

The following is a list of error messages you could encounter while using Viva. Messages are contained in the ErrorMessage.txt file in the VivaSystem directory. The text of the messages could be localized as long as the file format does not change.

- "<Message>" is not a valid COM Programmatic ID. Valid Formats are:
  - [Class Name]::[MemberName]
  - [Object Name].[Member Name]
  - [Object Name]'s Dispatch
  - Example: RichEdit1.(Get)Text
- <Message> has timed out. Do you want it to continue?
- <Message> is not a valid executable VIVA file.
- <Message> is not a valid fixed point value.
- <Message> is not a valid floating point value. Format is #.#e+#
- <Message> is not a valid integer value.
- A connected node was found on an I2adl 'def'.
- A parent object was indicated, but none was found.
- Aborting compile because <Message> objects could not be routed. One possible cause of this is to be out of PORTBITs in the ATIO system.
- Ambiguous object "<Message>" loaded. Replace existing object with this one?
- Ambiguous resolution of object with inputs (<Message>) A project and/or library cleanup is recommended.
- An empty behavior sheet cannot be converted into an object.
- An invalid "Data Set" name was entered.
- At least one child Data Set must be entered. Data Set: <Message>
- Attempt to expand empty node list. Alert system programmer.
- Attempt to get invalid property "<Message>".
- Attempt to remove defunct node. Alert system programmer.
- Attempt to set invalid property "<Message>".
- Base FPGA system missing PPort/APort attribute. System: <Message>
- Because the requested system is not available, will try to use another system.
- Behavior Sheet has changed. Do you want to close and save changes? ('Cancel' will leave this sheet active.)
- Bit map file is missing start-of-data indicator. FileName: <Message>
- Bogus delete from node list.
- Borland UI forms are no longer supported.
- Call to <Message> failed.
- Cannot change system-generated Data Sets.
- Cannot delete system-defined primitive objects.
- Cannot delete Viva Subsystem tree nodes.
- Cannot find an exposer for Data Set: <Message>
- Cannot initialize executor because translator does not exist.
- Cannot instantiate Class "<Message>". Its COM Library is not loaded.

- Cannot load <Message>; a COM object with this name is already loaded.
- Cannot load <Message>; a form with this name is already loaded.
- Cannot load <Message>; this form is already loaded.
- Cannot load File "<Message>". It is not a valid Type Library.
- Cannot open Chip Data file. File Name: <Message>
- Cannot open Sheet file.
- Cannot resolve Variant Data Set collector.
- Cannot synthesize an empty sheet.
- Can't erase file. Might still be open or read-only.
- Can't rename file. Might still be open or read-only.
- ChangeDataSet called on a nonchangable node.
- Check for mismatched input nodes on Object.
- Childless Data Set encountered. Data Set: <Message>
- Class "<Message>" does not exist.
- Class <Message> does not exist.
- Class function "<Message>" called without a valid COM Interface pointer.
- Coding of this object is proprietary.
- Collector/$Cast/Exposer removal logic only works for Data Sets with two children. Data Sets: <Message>
- Collector/$Cast/Exposer removal logic requires non-Variant/Vector Data Sets. Data Sets: <Message>
- Component <Message>. Aborting translation.
- Components of matching Data Sets are incompatible. See system programmer.
- Composite objects have been changed but not updated. Continue compile?
- Cost subtraction resulted in negative value.
- Could not create EDIF Image.
- Could not create file for output: <Message> May be an invalid file name.
- Could not create project save file.
- Could not create Save file for EDIF image.
- Could not create Save file for the sheet.
- Could not create save file for the system.
- Could not finalize the new style FPGA board. Bit Map file: <Message>
- Could not initialize the new style FPGA board. Bit Map file: <Message>
- Could not open the device driver for the new style FPGA board.
- Could not place GND or VCC object in system: <Message> Recommend changing the system description to avoid routing through a communication subsystem. Opening the VivaSystem\AddGndVccObjects sheet will add these objects into the X86CPU system.
- Could not send the Bit Map file to the new style FPGA board. Bit Map file: <Message>
- Could not update all of the Data Sets attached to the input/output objects on the behavior page.
- Could not update all of the Data Sets attached to this object. Please resolve the inconsistencies.

- Create a new I2adl "definition" object from this I2adl "reference" object?
- Data Set "<Message>" already exists. Ok to update?
- Data Sets that have more than two child Data Sets do not work with the standard Variant exposers and collectors. A redesign is recommended. Update the Data Set anyway?
- Delete this group and all objects therein?
- Did not create a new unique Wip page name.
- Discarding obsolete attribute string. <Message> Please make a new Save file.
- Duplicate entry for attribute "<Message>". Discarding latter entry.
- Empty node lists should set to "NullNodeList" not "NULL".
- End Of Systems not found in Resource Editor.
- Exeception thrown in <Message>.
- Failed to create COM object with Progammatic ID of "<Message>".
- Failed to create COM object with Progammatic ID of "<Message>".
- Failed to spawn "<Message>".
- Failure to load DLL containing user interface form. Ask the system manager to try making the DLL static linked. In C++ Builder, click on Project, Options, Packages tab. Uncheck the "Build with runtime packages" box.
- Failure to load XILINX program file.
- Failure to resolve polymorphic variant Object. Check input nodes.
- File <Message> already exists. Ok to replace?
- File is read-only. Do you want to make it read-write?
- FPGA systems require the presence of a ground object. System <Message>
- Global symbol duplication:
- I2adl object had SourceSinkLink but not InSystem/SaveIOLinks. Save file is corrupt.
- Icon cannot be directed to itself. Icon name: <Message>
- Incompatibility between the object's nodes and its behavior's input/output objects.
- Input object's location doesn't point to this system.
- Insufficient system resources for routing fixed object <Message>. Try another system? [Y/N]
- Insufficient system resources for routing.
- Interface Form referenced by <Message> is not open.
- Internal Error Trap <Message>
- Internal topology error.
- Invalid "Shared Resource" index number on resource: <Message>.
- Invalid child Data Set name: <Message>.
- Invalid entry.
- Invalid ImageType.
- Invalid index number encountered in nTList.
- Invalid input node type.
- Invalid or missing executable behavior.
- Invalid output node type.
- Invalid Project file version number.
- Invalid Property Index for Event.

- Invalid PrototypeObject.
- Invalid resource quantity encountered. Object: <Message>; Quantity: <Message>
- Invalid Sheet file version number.
- Invalid system file version number.
- Invalid value. Must be <Message>.
- Invalid Wip index number. Save file is corrupt.
- May only delete "Composite Objects" tree groups.
- Method was called with a NULL object pointer.
- Mismatch of Data Sets may be causing truncation of data.
- Negative resource quantity encountered. Object: <Message>
- No default X86UI form found.
- No help is available for this library. If such help file exists, then you need to have it installed.
- No help is available on this topic.
- No more resources are available.
- No Viva executable file specified.
- Node attribute for non-existent node.
- NodeIndex exceeds Input/Output count.
- NodeIndexes cannot be negative.
- Non-primitive object sent to translator.
- Not able to create the FPGA "bit" file. File name: <Message>.
- Not able to open the file.
- Not able to open the 'Generate Constants' file.
- Not able to open the Project file.
- Object has missing behavior page.
- Object has no available system.
- Object has undefined variant input.
- Object input nodes are out of sync with the behavior description.
- Object is referenced in object <Message>. Delete anyway?
- Object is referenced in system <Message>. Delete anyway?
- Object name cannot be blank.
- Object Name is referenced in polymorphic expansion of Object <Message>. Delete anyway?
- Object name starts with "Sheet". Is this OK?
- Object output nodes are out of sync with the behavior description.
- Object resource is unknown.
- Object with defunct behavior page has matching can be patched from system/library Object. Perform repair?
- Object with missing behavior page can be repaired using a system/library Object. Perform repair?
- ObjectType was not 'I2adl'.
- Ok to close Wip sheet without saving changes?
- OK to delete form <Message>?
- OK to delete global COM object <Message>?
- OK to permanently delete this Object?
- Old file format detected. Convert to the new format by saving out the file. <Message>
- Only one object on the sheet.

- Only output nodes can have information rate values.
- 'OtherEnd' was called on a non-transport object.
- Please resolve the inconsistencies.
- Project has been changed since it was compiled. Save persistence information anyway?
- Project Object's behavior is not in Project Behavior list.
- Property <Message> is not valid.
- Read/Write string over 10240 characters.
- Redefinition of Data Set does not match original.
- Reference to nonexistent COM member <Message>.
- Reference to nonexistent COM object referenced by <Message>.
- Reference to undefined Data Set <Message>. Using Variant Data Set.
- Reference to undefined node.
- Requested file does not exist.
- Resource cost table contained a negative cost value.
- Resource declaration doesn't match resource definition or provides resource is used for IOOffset Already.
- Resource declaration doesn't match resource definition.
- Resource Prototype System not found in Resource Editor. Prototype Name: <Message>
- Save the changes made to the widget form?
- Search Engine tried to remove nodes from a Variant collector that was connected.
- See system programmer. A dynamic call to add nodes is required.
- Sequencing error in 1-pass synthesis algorithm. Alert system programmer. Remove?
- SourceSinkLink found on unknown object type. Save file is corrupt.
- SourceSinkLink index number is out of range. Save file is corrupt.
- SourceSinkLink was not loaded. Save file is corrupt.
- String not enclosed by matching quotation marks.
- Subsystem <Message> selected. Save just this subsystem?
- Syntax error.
- Synthesizer failure. Cannot resolve collector for Data Sets [<Message>].
- Synthesizer failure. Project contains polymorphic objects that cannot be resolved.
- System <Message> referred to by this object is not valid.
- System generated Data Sets cannot be deleted.
- System object does not have a valid resource/information rate. System: <Message>; Object: <Message>; Resource: <Message>
- The following ambiguous objects were encountered: <Message> A project and/or library cleanup is recommended.
- The license for this version of VIVA has expired. Please call Star Bridge Systems to renew.
- The license for this version of VIVA will expire in <Message> days. Please renew the license with Star Bridge Systems.
- The node type is unknown (not input or output).

- The path "<Message>" on the BuildDirectory system attribute is invalid. Can't invoke Xilinx build tools.
- The Router should not have found the root self connection node.
- The string table is empty (not preloaded with the NULL string).
- The user's application contains a low-level $Select object that cannot be resolved. The selector for the $Select object (Input S) must be a constant 0 or 1, resolvable at compilation time. Suggest using an ordinary Select object instead.
- This Data Set cannot be deleted because it is a child Data Set of <Message>.
- This data set cannot be deleted because it is referenced in object <Message>.
- This is not a Viva text base Save file.
- This object has no complex behavior. Did you intend to use this object as a prototype for a primitive object?
- This old FPGA project must be recompiled.
- This option is not supported at this time.
- This type of executor cannot have an active thread.
- This type of executor cannot have events.
- This type of executor cannot have I/O events.
- tkFloat requires 'Float' Data Set in Viva.
- To save a sheet, it must be active and non-empty.
- Total Quantity not positive on resource: <Message>.
- Tried to delete a sheet that was not in the WipTree.
- Tried to translate a system that does not have a translate method.
- Type Library "<Message>" does not exist. You need to have it registered on this machine.
- Unable to open this image. The bitmap is named "I2adlView.bmp", and is located in your Viva directory.
- Unable to save Bit Map file.
- Unable to save form "<Message>". It may be locked by another process, or you may not have permission to write to this directory.
- Undefined behavior page.
- Undefined global label. <Message> Input will be treated as zero.
- Undefined Input Data Set on top-level page.
- Unknown primitive object execute type.
- Unspecified exception thrown by <Message>.
- Update canceled! Create a new object?
- Using a VariantIn on a Data Set with more than two components will result in the loss of data. Data Set: <Message>
- Variant Data Sets cannot be constants.
- VIVA has detected the presence of Data Sets that are not referenced. Do you wish to remove them?
- VList does not allow the Count to be negative.
- VList garbage collection requires 4 byte integers and void pointers.
- VList garbage collection was active before it was initialized.
- VList garbage collection was never activated.
- VLists have a maximum capacity of a billion.

- Warning! Input/output nodes have been changed. Changing the order of the nodes, the number of nodes, or their Data Sets could invalidate any behavior page that already contains the Object. Do you want to proceed?
- Would you like to save the changes made to this project?
- Xilinx tools are not working. Make sure they are installed. Do not run inside the Borland C++ Builder debugger. Abort the compile?
- You cannot delete <Message>, because it is a dependency of: <Message>
- You cannot remove this libary reference, because it is a dependency of <Message>.
- You can't create an image for an object that does not yet exist.
- Your current system requires the DLPortIO device driver to program the FPGAs. Run Port95nt.exe to install this DLL.

STAR BRIDGE SYSTEMS HARDWARE

Pinout/PE/PE Whole Board

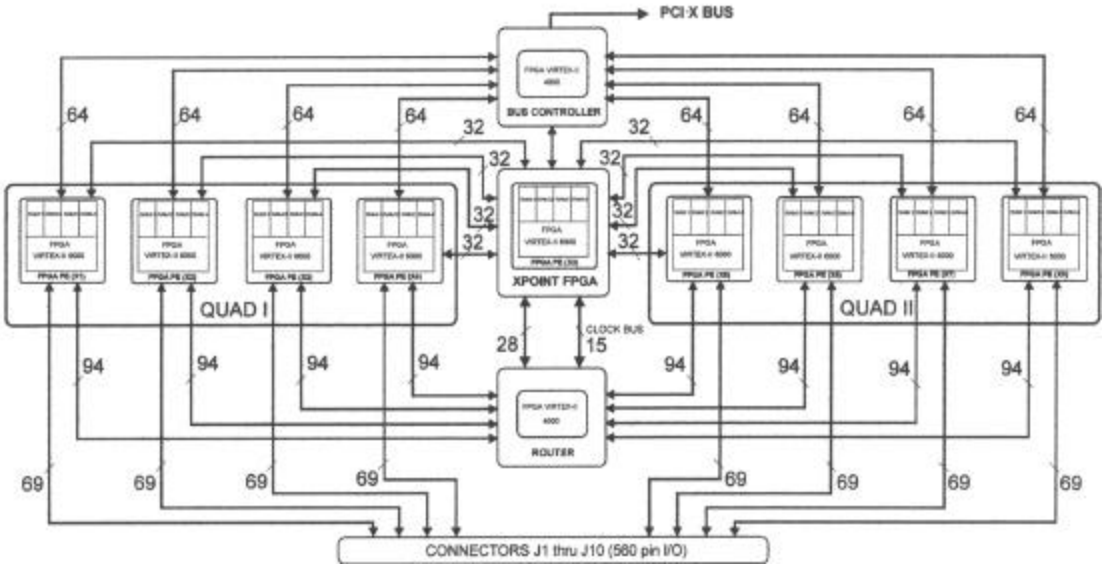
Useful information about hypercomputer structure that includes such things as schematics and pinouts is available in the Hardware Reference Guide. You will find board diagrams for Star Bridge Systems hypercomputers in the section titled *Hypercomputer Board Diagrams and Descriptions*. The Hardware reference guide is located at the following URL:

http://www.starbridgesystems.com/forum/viewtopic.php?t=108

Hypercomputer Board Diagrams and Descriptions

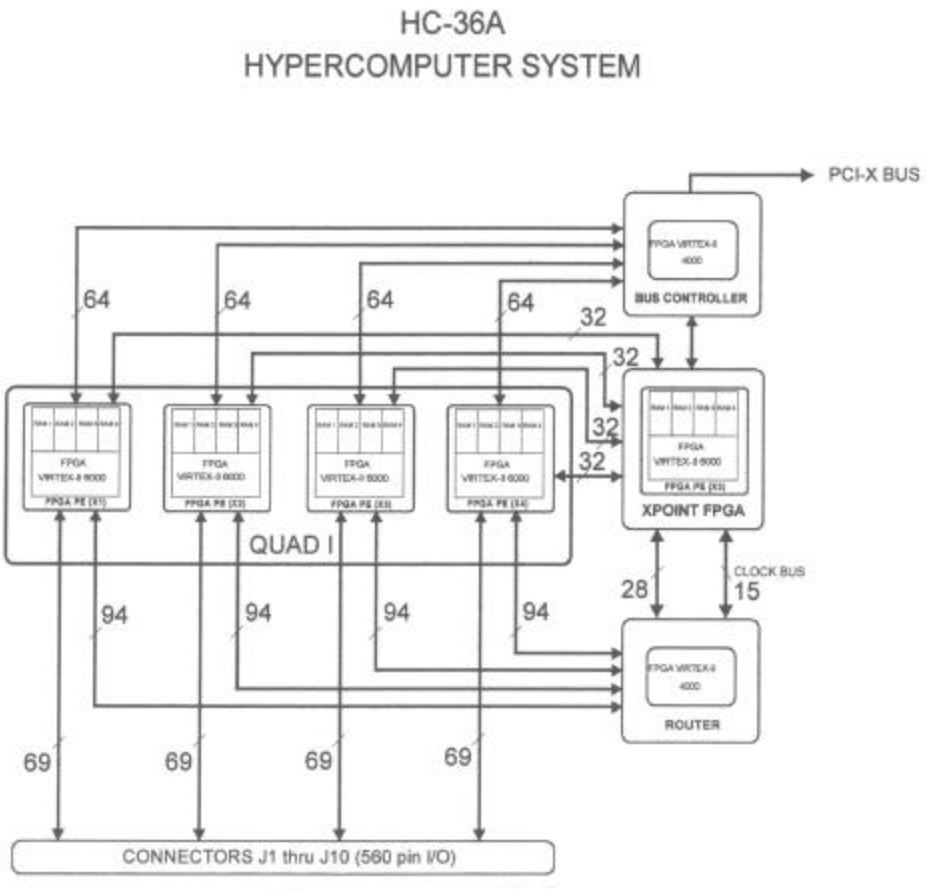
HC-62 Specifications and Board Diagram

- PCI/X Bus Interface 133 MHz, 64 bit
- 2 Virtex II 4000 chips
- 9 Virtex II 6000 chips
- 18 G Byte Ram with 36 64-bit parallel memory channels
- 450 G Bits/sec Inter chip communications
- 500 plus External I/O pins



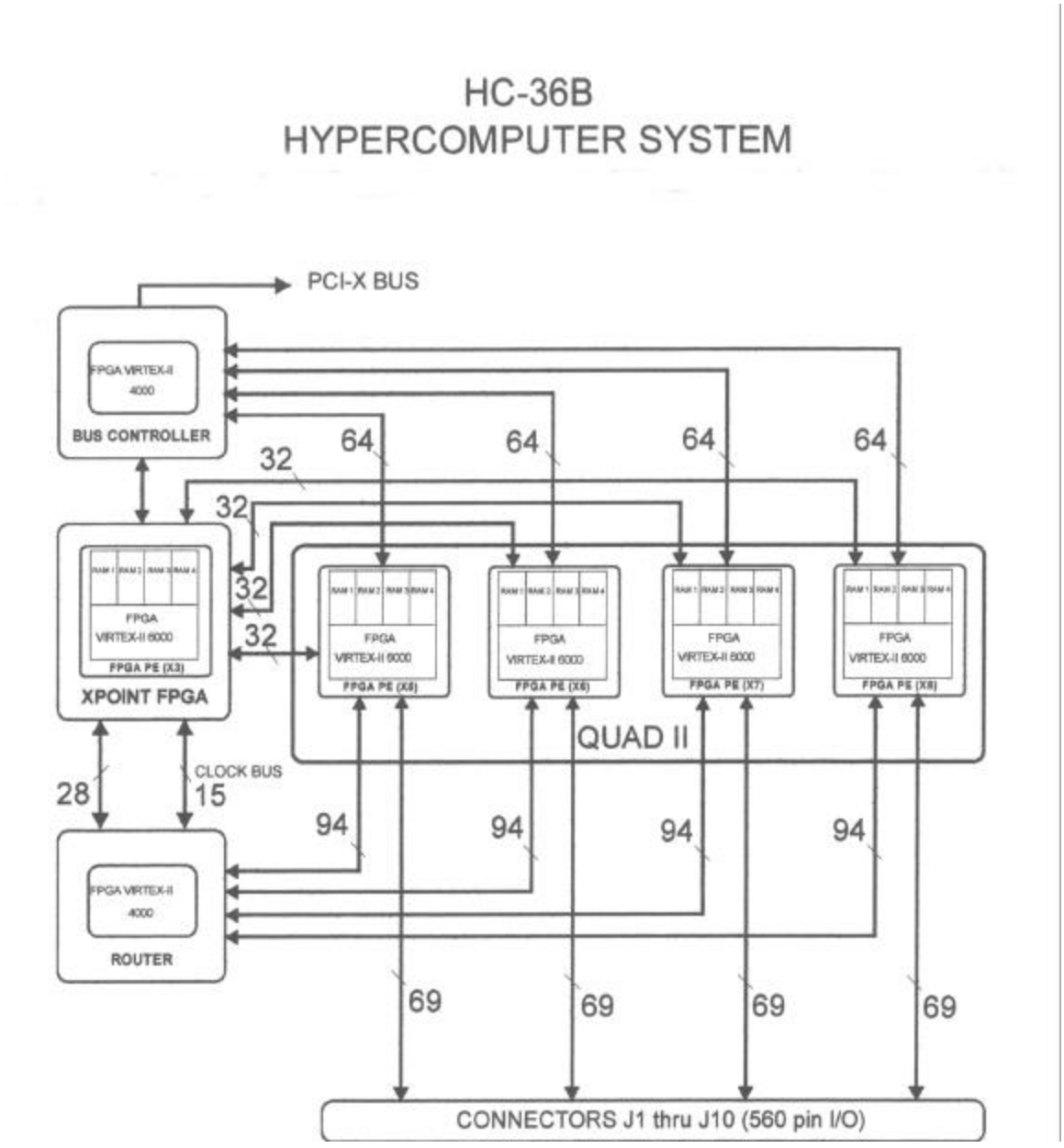
HC-36a Specifications and Board Diagram

- PCI/X Bus Interface 133 MHz, 64 bit
- 2 Virtex II 4000 chips
- 5 Virtex II 6000 chips
- 10 G Byte Ram with 20 64-bit parallel memory channels
- 225 G Bits/sec Inter chip communications
- 200 plus External I/O pins



HC-36b Specifications and Board Diagram

- PCI/X Bus Interface 133 MHz, 64 bit
- 2 Virtex II 4000 chips
- 5 Virtex II 6000 chips
- 10 G Byte Ram with 20 64-bit parallel memory channels
- 225 G Bits/sec Inter chip communications
- 200 plus External I/O pins



FPGA Architecture

For in-depth documentation on the Xilinx Virtex-II family of FPGAs, including architecture, point your browser to the following URL:

http://direct.xilinx.com/bvdocs/publications/ds031.pdf.

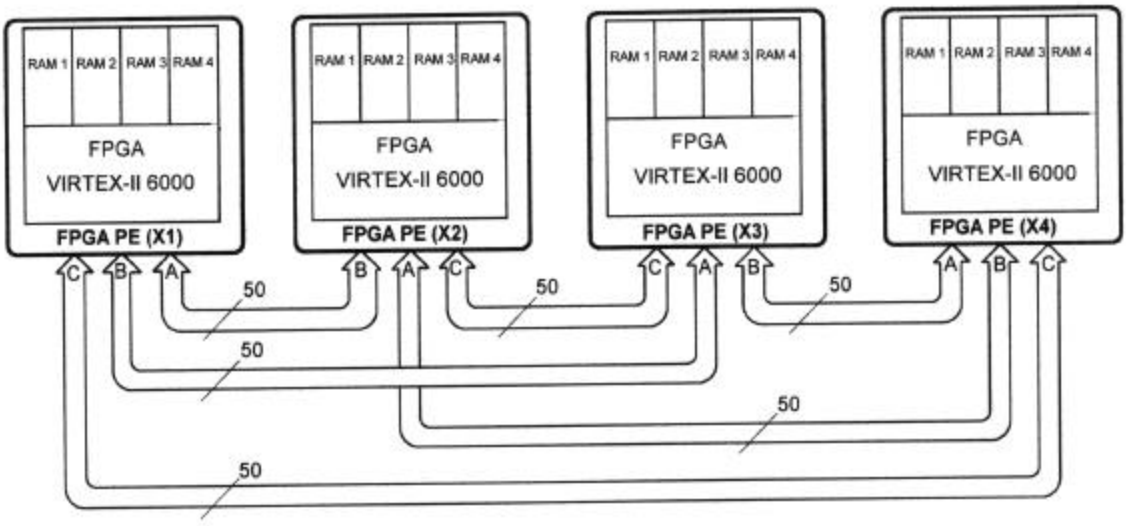
Data Rates

Data Rates Over the PCI-X Bus

The average speed of data transferred through the PCI-X bus is 3.2 bits per clock cycle. This average speed is not a limitation of the PCI-X bus. Rather, it is a result of the combination of the PCI Controller and the system description in use. Star Bridge Systems expects rates to improve.

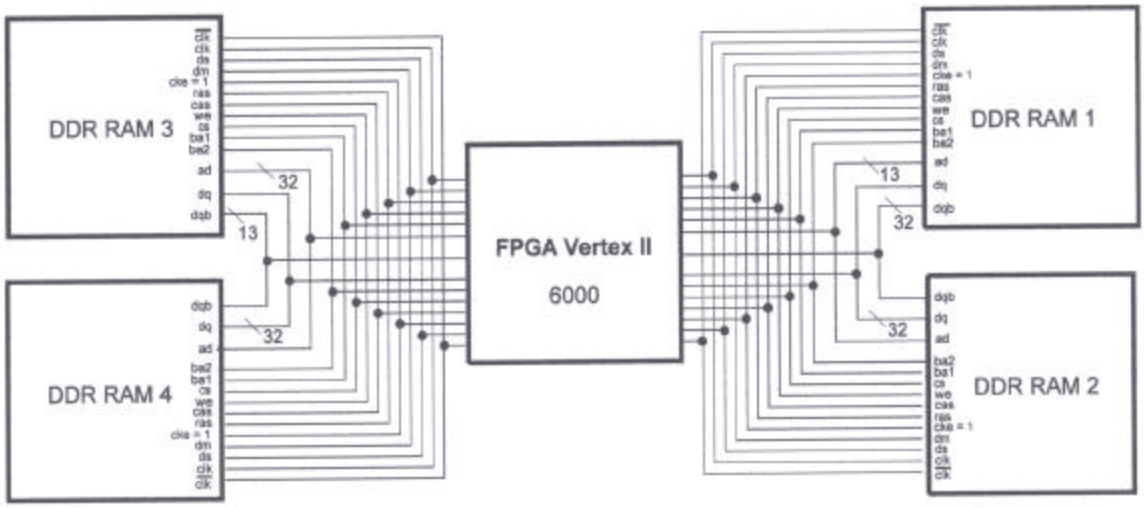
Data Rates from PE to PE

Rates from PE to PE are based on the chip speed multiplied by 50 (lines from PE to PE). The Virtex DCMs will go up to 400 MHz. It will be possible to double data rate (DDR) the pins.



Data Rates from PE to Memory on the Hardware

There are thirty-two lines from each memory module to the FPGA. These lines use DDR, though, so there are effectively sixty-four lines.



Programming FPGAs on the HC Board

The following ‘C’ code shows how to program FPGAs on the Star Bridge Systems Hypercomputer system. The code reads a bitstream file, which contains the program for the FPGA, and writes the program from that file to the Hypercomputer system to program the FPGA.

Sample C Code for Programming FPGAs

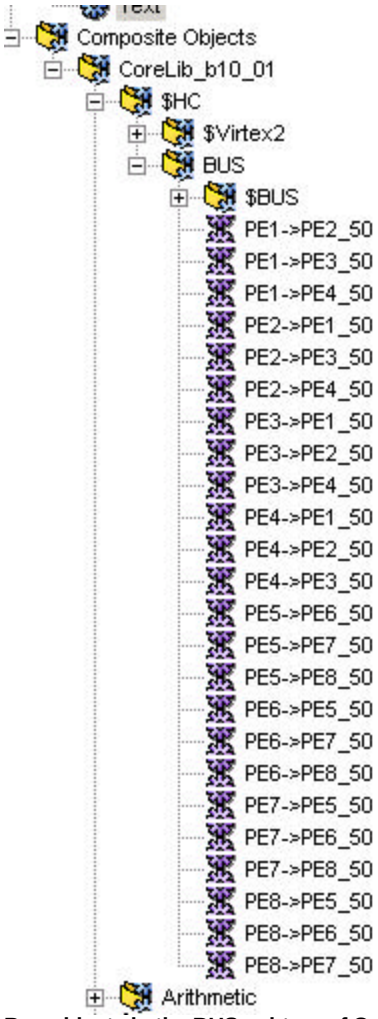
```
// MS Winsdk header to define IOCTL calls for communication with the driver
#include <winioctl.h>

// Viva header defining calls to the HC device driver
#include "Hcioctl.h"

// Program the new style board by sending the bit stream files through the device driver to
// the  intended chip.

int ProgramNewStyleBoard(VivaSystem *BoardSystem)
{
        // Open the device driver for the new style board.
        //     "1" means the first device driver (Board1).
        //  APP_SYMBOLIC_NAME”1” = device name
        HANDLE  DeviceHandle = CreateFile(APP_SYMBOLIC_NAME "1",
                GENERIC_READ | GENERIC_WRITE, 0,
                NULL, OPEN_EXISTING, FILE_FLAG_NO_BUFFERING, NULL);

        if (DeviceHandle == INVALID_HANDLE_VALUE)
        {
                // Could not open the device driver for the new style board.
                Error();
```

```
        return -1;
}

// opens the bit stream file for reading
int     FileHandle = open(BitStreamFileName, O_RDONLY | O_BINARY);

if (FileHandle == -1)
        return –1;

// Start out reading the bit stream file one character at a time.
// Read until character = -1 to get to start of data
while (true)
{
        HC_PROGRAM_STRUCT   BitStreamData;
        unsigned long       BytesWritten;
         int  CharsRead = read(FileHandle, BitStreamData.progData, 1);

        // if we got to the eof without getting past the header -- error
        if (CharsRead <= 0)
        {
                // Bit stream file is missing the start-of-data indicator.
                Error(BitStreamFileName);
                break;
        }

        // If not start-of-data, loop and read next byte
        if (BitStartData.progData[0] != 0xff)
                continue;

        // Select our chip by putting a "1" bit in the proper location.
        // The bmChipSelect field is a bitmap indicating to the driver which
        // chip(s) is to be programmed.  For example, if we were programming
        // chips 1 and 4, the binary value would contain 1s in the 1 and 4 positions:
        //  (00001001)
        BitStreamData.bmChipSelect = (BM_CHIP_SELECT) (1 <<
                        ChipNumber);

        // Initialize the FPGA chip.
        if (!DeviceIoControl(DeviceHandle, HCIOCTL_PROG_INIT,
                        &BitMapData.bmChipSelect,
                        sizeof(BM_CHIP_SELECT), NULL, 0, &BytesWritten,
                        NULL))
        {
                // Could not initialize the new style board.
                Error(BitStreamFileName);

                return -1;
        }

        // Write out the first byte of data and then the rest of the file 16K at a time.
        while (CharsRead > 0)
        {
                if (!DeviceIoControl(DeviceHandle, HCIOCTL_PROG_FPGA,
                        &BitMapData,
                        sizeof(BM_CHIP_SELECT) + CharsRead, NULL, 0, &BytesWritten,
                        NULL))
```

```
                        {
                                // Could not send the bit stream file to the new style board.
                                Error(BitStreamFileName);
                                return –1;
                        }

                        CharsRead = read(FileHandle, BitMapData.progData,
                                MAX_DATA_LENGTH);
                }

                // Finalize the programming of the FPGA chip.
                if (!DeviceIoControl(DeviceHandle, HCIOCTL_PROG_END,
                        &BitMapData.bmChipSelect,
                        sizeof(BM_CHIP_SELECT), NULL, 0, &BytesWritten, NULL))
                {
                        // Could not finalize the new style board.
                        Error(BitStreamFileName);
                        return -1
                }

                break;
        }

        // close handle to bitstream file
        close(FileHandle);

        // close handle to device
        CloseHandle(DeviceHandle);
}
```

Communication Among FPGAs

If you would like to create a bus between two objects, use the Bus objects. Bus objects are part of the CoreLib $HC subtree. The list of available Bus objects is illustrated in the following figure.



Bus objects in the BUS subtree of CoreLib

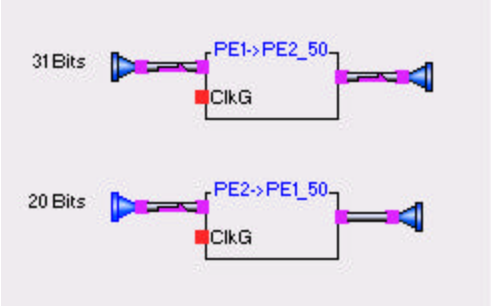
Use the bus object for whatever combination of chips you want to communicate between. You can determine which bus object to use by the way it is named. That is, components of the bus object name indicate the sender, receiver, and maximum bandwidth.

Sender->Receiver_Max Bandwidth
Sample file name: PE1->PE2_50

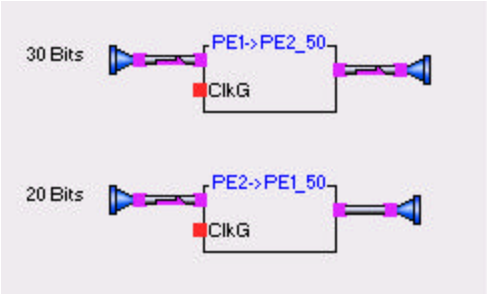
In the object named PE1->PE2_50, PE1 is the sender, PE2 is the receiver, and 50 lines is the maximum bandwidth.

All objects output the same Data Set that is put in the object. The only constraint is that you can't input more bits than the maximum bandwidth.

The following examples illustrate non-permissible and permissible chip-to-chip communication constructions.



Non-permissible construction—too many bits



Permissible construction

CLOCKS AND TIMING CONSTRAINTS

There are many ways to generate and use clocks in Viva. By default, registers are clocked from a global clock, ClkG. There are different means by which ClkG may be implemented or overloaded.

ClkG

ClkG is a globally available clock signal, and is the default clock source for all library objects. In order to reference ClkG, you must create a Bit input named ClkG, with the Resource, *Global. Any object with a global ClkG input will use the instance of ClkG that is gernerated in whatever system is currently being used. The ClkG input on an object does not need to be connected to anything since ClkG is a global signal. All synchronous library objects have ClkG inputs so that the clock may be overloaded. This is accomplished by connecting a transport that is carrying a clock signal other than ClkG to an object's ClkG input. We will see techniques for creating and using alternate clock signals later in this document.

ClkG Implementations

Every system has its own ClkG implementation. The ClkG signal must originate somewhere, and be made globally available. This is usually accomplished with one or more behavioral systems. A behavioral system is a system whose resources are generated in a Viva behavior. Behavioral systems are always compiled, and as such, their inputs and outputs will show up in your behavior's widget form.

x86 ClkG Implementation

In the x86 system, ClkG is generated in an Implementor object in the x86/x86ClkImport system. The behavior of this object takes the emulator-native clock signal, $Clock, and translates it into ClkG. The signal $Clock is generated in the emulator in a manner that is independant of propogation delay.

The implementor object contains a ClkG output, which also carries a *Global Resource. This output makes the signal globally available to all objects in the project. The ClkG implementor multiplexes the source of ClkG between $Clock and the Step input. By setting the Run/Step input to 1, you may provide manual clock oscillations by toggling the Step input.

FAI ClkG Implementation

Note: Some of the information contained under the heading *FAI ClkG Implementation* is only applicable to Hyper Algorithmic Logic (HAL). ClkG is a function of the hardware and thus has no functionality in the simulator.

In the FAI system, it takes a combination several behavioral systems to implement ClkG in a phase-synchronous manner. These include the ClockSystem, the MetaClock system, and a ClkImport system for each FPGA. The ClockSystem contains a Viva behavior that implements a ring oscillator, whose speed is programmable through the Speed and 2x inputs. The Speed input allows you to incrementally increase the speed of ClkG, while the 2x input doubles the current speed. Like the x86 implementation, we have the ability to step the clock through the use of the Run/Step and Step inputs. The current speed of ClkG is output in Khz via the Khz output. If the clock is operating properly, the Khz output oscillate between a small range of values. This does not mean that the clock is unstable. This changes in the Khz output occur becuase of the sampling of the signal through the ATIO system. If the Khz input is static (max value), there has been a problem programming the board, and you are not communicating with the hardware.

The MetaClock system is needed to bridge the gap between a physical clock signal, and the abstract global signal that your objects' behaviors reference. For instance, in each FPGA, the

clock signal comes from a global clock buffer (BUFG), but all of your objects reference an abstract global signal. The MetaClock system provides an abstract transfer layer between the generation of ClkG in the ClockSystem, and each FPGA's ClkImport system. In order to make ClkG phase synchronous in all FPGA's ClkG is brought in from a globally-shared bus line.

Note: Star Bridge Systems hypercomputers synchronize the local DCM to the input pin.

Each FPGA has a ClkImport System, which literally imports the clock signal into the FPGA. The Implementor for this system contains an IPAD and a BUFG. The IPAD has a PadLoc attribute, defining the location of the bus wire carrying the clock signal. The clock signal enters a BUFG so that it may be available to all registers through clock buffer routing resources. On an FAI board, all FPGAs get their clock signal from a single global bus wire. Although the ring oscillator that generates ClkG resides in XPoint, this chip is no exception. ClkG is routed out of XPoint on one bus line, and then bridged to the other, on which it is imported back into the chip.

Timing Constraints

It is very easy to provide timing constraints for an entire clock domain on an individual FPGA. This is accomplished by adding a TimeSpec object to a given FPGA's ClkImport Implementor. The TimeSpec object may be found in XPoint's object library. If you are providing constraints for an FPGA other than XPoint, you should first change the system attribute of the TimeSpec object to match the FPGA you are working with. Then you must enter a TIMESPEC attribute. Here you specify which clock domain you are dealing with, and the period of the clock in that domain. In this example, the TIMESPEC attribute's value is: 'TSCLKG=PERIOD:CLKG:20'. Note that our clock domain, ClkG, is spelled in all caps in the attribure value. Here we are specifying that ClkG will have a period of 20 nanoseconds. Assuming our design meets this constraint in the Xilinx tools, we would be able to run ClkG at 50 Mhz and be guaranteed timing closure. You may provide TimeSpec objects for all FPGAs and you may provide multiple TimeSpec objects for multiple clock domains on an individual FPGA.

Other Clock Implementations

It is possible for you to create other clock domains, or to alter ClkG's behavior. For example, you may instance additional ring oscillators to create other clocks. There are other clock sources that may be used as well. Some FAI boards are equipped with a 100Mhz oscillator. The 100Mhz clock system makes this signal globally available. However, each FPGA that utilizes this clock will still need a ClkImport system that describes the location of the bus line carrying this signal (D29 for XPoint, see below). In XPoint, the PCI and ISA clocks are also available. See the PCITunnel object for an example of PCI clock usage. If you've ever wondered how the speed of ClkG is calculated, it is measured against the 8Mhz ISA clock.

Using the Default System Clock

In the hardware, clocks are routed on their own special wires. Clocks go through buffers and operators. By default, Star Bridge Systems uses one DCM object synchronized with the PCI bus to run the clock. This DCM object is instanced in the clock import implementor for every chip. The global signal ClkG uses a communication system called MetaClock. MetaClock routes the ClkG between the objects and the clock for the system in which objects are placed.

Plugging in Your Own Clock

When you do not want to use the default clock system, you can plug in your own clock. If such is the case, you would only plug in a clock to two objects: a DCM and a clock pin. Star Bridge Systems recommends that you read the Xilinx documentation for the DCM and the BUFG objects before attempting to construct your own clock.

Note: An input horn called $Clock is the default X86 CPU clock and is equivalent to the emulator cycle time.

RESOURCE USAGE

To track resource usage, perform the following steps:

1. From the System menu, choose Enable Editor.
   - Enabling the Editor enables the Resource Editor and the System Editor. When Editors are enabled, they appear as tabs at the bottom of the workspace.
2. After synthesizing a program, select the Resource Editor tab.
3. Click Display Usage.
   - Resources tracked as used are displayed.

Note: For exact resources used, refer to the Xilinx log files . The default log file location is in the VivaSystem directory.

MULTIPLE USER SUPPORT

Viva.ini File

The Preferences option on the Tools drop-down menu allows users to select a number of settings to customize VIVA. These preferences are stored in the Viva.ini file. In VIVA 2.2, this file is being moved from the system directory (C:\Viva\VivaSystem) to the user's settings directory (C:\Documents and Settings\<username>). Thus, each person on a shared use machine should have a distinct user name to avoid overwriting someone else's preferences. When VIVA 2.2 is installed, the existing preferences can be preserved by copying the Viva.ini file from the system directory to the user's settings directory.

FPGA Chip Numbers

Viva only supports one project running on a single FPGA chip at a time in a multi-user environment. When a Viva project is loaded onto an FPGA chip, the previous program is erased. Therefore, care must be taken to assign different chip numbers to each simultaneous user so they will not overwrite each other's programs. The chip number— PE1, PE2, PE3, and so on—comes from the system file that is opened.

"BuildDir" System Attribute

When an FPGA project is compiled, the Xilinx tools create a number of disk files. The new system attribute "BuildDir=<directory>" allows the user to override C:\Viva\VivaSystem as the directory for these files. This attribute is entered on the base system in the system editor. It allows simultaneous Xilinx compiles by placing the files created by Xilinx in different directories.

The <directory> part of the attribute may be a fully qualified name (starts with a drive letter or backslash), or it may be relative to one of the following directories:

" "	C:\Viva\VivaSystem
"$V\"	C:\Viva
"$S\"	C:\Viva\VivaSystem
"$P\"	Current project's directory

SPAWN

Spawn Facilitates FPGA Reconfiguration

The Spawn object, located in the X86 CPU system, allows you to execute another Save File. The Save File can either be a Viva Executable file (.vex) or a Viva Project file (.idl) specified by its FileName attribute (set in the Attribute Editor).

Spawn calls VivaRun.exe if a .vex file is specified for its FileName attribute, or Viva.exe if an .idl file is specified. If the Viva Project file specified is not compiled, then Viva will compile it before attempting to execute it.

Spawn returns data from the called process once it has terminated, via its “Result” node.

An attribute of "Exclusive=true" on a Spawn object will cause the calling process to halt and wait for the termination of the child process.

All spawned processes self-terminate when their caller terminates.

Go

Go is a notification mechanism that a Viva execution process has begun. A behavior can have any number of these. The Send node issues whatever data was passed to this process on the command line, if applicable; 0 otherwise. The Done node sends out a 0 value, then a 1 value, when the process begins. Use it to trigger any initialization functionality that is warranted by any behavior.

Stop

The inverse of the Go object, the Stop object is used for three purposes: (1) to communicate to the behavior that it has received a termination request; (2) for the behavior to communicate to Viva that it is ready to be terminated, and (3) to allow the behavior to relay data to the calling process. Data on the Stop of the executed behavior appears on the Result of the Spawn.

The “Stopping” node sends out a 0 value, then a 1 value, when the process has received a termination request. The process will not actually terminate until *all* Stop objects have received a high state into their “Go” nodes. The data given the “Stop” node of the last Stop object to receive this signal will be the data passed back to the calling process.

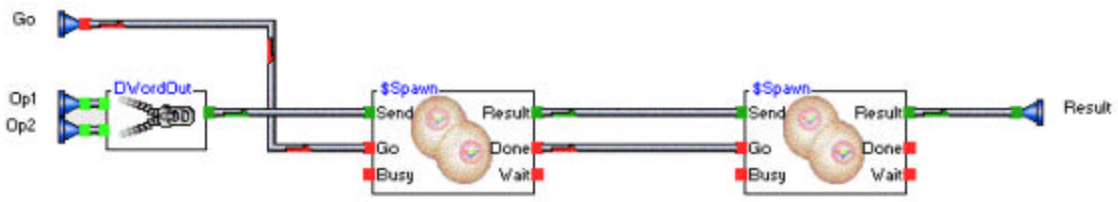
Implementing Spawn

In this example, the Spawn object calls two different Viva “executable” files (.vex) in sequence, which both target the same PE chip. Data is passed to the called process through the “Send” node, and data is received from the called process through the “Result” node.

In this case, the “Result” output should be five plus the product of “Op1” and “Op2”.

“config1.vex”, called by the first $Spawn, will program PE7 with a behavior having a multiply object, then split the parameter (given through the “Send” node) back into Words and multiply the two of them.

“config2.vex”, called by the second $Spawn, reconfigures PE7 with a behavior having an Add object, and then adds five to the number passed, which is the product of “Op1” and “Op2”.



In this process, we run two different FPGA behaviors at different times on the same chip, without having any FPGA-targeting objects in this behavior, and without having had to compile anything of substance.

For a simple test case, put a value of 3 on the Op1 input, and 5 on the Op2 input, and set the Go input high. After two processes return, you will have to press the “Bring Widget Form to Front” button. The Result output should have a value of 20.

SAVE FILE (TEXT BASE)

When Viva saves a file, it groups the following components sequentially: (1) header, (2) COM dependency references, (3) DataSet definitions, (4) object prototypes, (5) object definitions, and (6) system definitions. The components within each group are listed in the file in the order in which they were created. Therefore, objects added to a Viva project/sheet/system will always appear at the end of the save file.

Comparing or merging Viva save files is similar to comparing or merging text files for any other programming language since the listed components can be defined in any order, with the exception of the header.

Star Bridge Systems recommends observing the following precautions regarding save files:

- Be sure not to interject return characters where they do not belong. For example, do not include a closing parenthesis on a line separate from the last list item that it enclosed.
- Identifier names should not contain non-alphanumeric characters other than "_" and "$".
- Always make backups before editing save files.

Save File Components

Header

The first component of the file must be "// VIVA ". Other parts of the header identify the type of file and at what date/time it was created.

COM Dependency Reference

ComLibrary

ComLibrary "[file name]"

Example: ComLibrary "C:\VIVA\VIVASYSTEM\VIVAATOM.DLL"

[file name] The fully qualified path and file name of the COM Type Library.

ComForm

ComForm "[file name]"

Example: ComForm "C:\viva\Tests\Edit.cfm"

[file name] The fully qualified path and file name of the COM Form file (*.cfm).

ComObject

ComObject [Prog ID] [object name]

Example: ComObject VIVAATOMLib.x86Atoms x86Atoms1

[Prog ID] The Viva-defined (not Windows-defined) programmatic ID of the COM class to instantiate. Consists of the name of the containing library and the COM class name, separated by ".".

[object name] The alphanumeric tag by which this instance will be referenced within Viva.

DataSet Definition

DataSet [name] = ([child1], [child2]…); //_Attributes [context type], [color], [tree group], [COM variable type];

Example 1: DataSet Fix32 = (Word , Word); //_Attributes 4,12632256,System\Static

Example 2: DataSet "DWord*" = (Word , Word); //_Attributes 1,11141375,System\Static,16403

[name] The name of the DataSet. Can contain any printable character other than a tab or return; if it contains characters other than a through z, 0 through 9, or underscore (_), then it must be enclosed in quotes.

[child] The name of a DataSet that is a component of this DataSet. Can have any number of "children", or component DataSets, but do not define a DataSet that has less than 1 or more than 200 children.

[context type] Numeric value corresponding to one of the values listed in the Attribute box of the Dataset Editor. Equal to two to the power of the zero-based ordinal value of the entry in the Attribute box. This information is used only by widgets.

[color] The 24-bit numeric code for the display color for this DataSet on nodes.

[treegroup] Optional. The name of the tree in which this object will be displayed. Consists of the name of the parent group on each level separated by "\".

[COM variable type] Optional.

Object Definition and Object Prototype

Object ([outputs]) [name]([inputs]); //_Attributes [Attribute list]
{
[documentation]
//_ Object Prototypes
[object prototype list]
// Behavior Topology
[node connection listing]
}

The portion enclosed in "{" and "}" is the behavior definition, and does not apply to primitive objects or to object prototypes.

An object definition without inputs, outputs, or attributes is interpreted as a WIP sheet.

Example:

```
Object ( Bit O) Mux( Bit A, Bit B, Bit S)
//_Attributes System=X86CPU,Documentation=Bit
{
    // Mux - Variant Select Case
    //
    // Date Last Modified:
    // 09 Nov 2002
    //
    // Author:
    // Samuel Brown
    // SBS Inc.
     //_ Object Prototypes
     Object ( Bit A) Input;  //_GUI 9,12
     Object ( Bit B) Input:A;  //_GUI 9,21
     Object ( Bit S) Input:B;  //_GUI 9,24
     Object Output( Bit O) ;  //_GUI 78,14
     Object ( Bit Out1) INVERT( Bit In1) ;  //_GUI 22,15
     Object ( Bit Out) AND( Bit In1, Bit In2) ;  //_GUI 45,11
     Object ( Bit Out) AND:A( Bit In1, Bit In2) ;  //_GUI 46,20
     Object ( Bit Out1, Bit Out2, Bit Out3) Junction
        ( Bit In0) ;  //_GUI 19,24
     Object ( Bit Out) OR( Bit In1, Bit In2) ;  //_GUI 61,12

     // Behavior Topology
     Output = OR;
     INVERT = Junction.Out1;  //_GUI 20,18
     AND.In1 = Input;
     AND.In2 = INVERT;  //_GUI 40,16, 40,18
     AND:A.In1 = Input:A;
     AND:A.In2 = Junction.Out2;
     Junction = Input:B;
     OR.In1 = AND;
     OR.In2 = AND:A;  //_GUI 58,17
}
```

Object Definition

[outputs]	A comma-separated list of output parameters, each consisting of the dataset of the output, followed by its name.
[name]	The name of the object being prototyped. If it contains whitespace, then it must be enclosed in quotes.
[inputs]	A comma-separated list of input parameters, each consisting of the dataset of the input, followed by its name.
[Attribute list]	A comma-separated list of attributes for the object, in the standard format of [name]=[value], with no intervening whitespace.

Behavior definition

[documentation]	Used to populate the Documentation field of the object, as seen on the right-hand side of the Attribute Editor. Can span any number of lines; each signified by "//" prefix. Note that "//_" is used as a terminator for the documentation portion.
[object prototype list]	The newline-delimited list of object prototypes (as explained earlier) of objects used in the behavior of this object. In Viva, you reference and prototype an object using the same statement. Each object name in this listing must be unique for this behavior. For each object havi ng a name ambiguous with that of another object in the same behavior, Viva adds a suffix beginning with ":"; such suffix does not become part of the actual object name once the file is loaded.
[node connection listing]	The newline-delimited list of node connections, taking the form *[object name].[input node name] = [object name].[output node name]* [node name] can be omitted for a node if it is the only one in its list (inputs or outputs).

System definition

System X86
//_Attributes [attribute list]
{
[nested system definitions]
}

Example:

```
System X86
//_Attributes 1,Resource=Default,DefaultInputWidget=ScrollBar,DefaultOutputWidget=ScrollBar,
WidgetSystem=X86UI,WidgetResource=TIMESLICE,WidgetLocation=?,DefaultTargetSystem=X8
6CPU
{
 System X86CPU
 //_Attributes 2,Resource=X86Main
 {
 }
 System X86UI
 //_Attributes 3,Resource=X86UI
 {
 }
 System UIXCPU
 //_Attributes 7,Resource=UIBus
 {
 }
 System X86ClkImport
 //_Attributes 6,Resource=ClockImport
 {
 }
}
```


[attribute list]	List of attributes in the same format as in Object definitions. The first attribute listed must be the 1-based ordinal value of the system type as defined in the SystemType combo box in the System Editor. The second attribute must be “Resource=” + the name of the system’s resource prototype as listed in the Resource Prototype combo box in the System Editor

Object Attributes with Special Meaning

Some object attributes have special meaning at load time, rather than existing as standard object attributes. Their names, valid values, and meanings are listed below.

Name	Effect/meaning and usage
Primitive	Marks the applicable object as primitive.
CurrentSheet	This behavior will be the sheet that is displayed when the project is opened.
CompileSheet	This is the compiled behavior for this project.
Link	Used by Viva when writing compiled behaviors. Do not edit.
InfoRate	Specifies InfoRate value for object and/or nodes. Valid values are “>”, “=”, and “<”. Specify a value for a node using “:” + [node name] + “=” + [value] Example: InfoRate=">":Out2=">"
KeepObject	Sets the Keep Object field, seen in the Attribute Editor. Values are "Both sides", "One side", and "Always".
ProgID	Directive to set this object as a member (function, get/set property, or event dispatcher) for the COM class or static instance, instance generator, or dispatch identifier for a global instance, specified by the value of this attribute. This takes one of 4 forms:
Member of static Instance:	Name of instance (as defined in a ComObject) + “.” + name of member Example: ProgID=Chartfx1.Type
Member of class:	Type Library ID + “::” + Name of class + “::” + name of member. Example: ProgID=VIVASTANDARDOBJECTSLib::VivaString::LeftString
COM instance generator:	Type Library ID + “::” + Name of class + “::” + “[Create]” Example: ProgID=VIVASTANDARDOBJECTSLib::VivaString::[Create]
Dispatch identifier: (avoid editing; consider read-only)	Name of instance + “’s Dispatch” Example: ProgID=VivaString1's Dispatch

COM

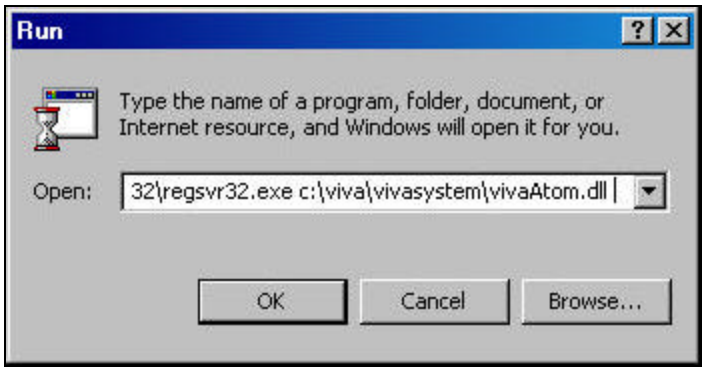
Component Object Model (COM) is an operating system-wide convention for code exposure. It is a sort of "universal interface." Code written in any language can be exposed using COM and used in any language that uses the COM interface. OLE, ActiveX, COM+, and DCOM are all derived from COM.

Popular programming languages have proprietary systems for using COM components via the COM interface and for creating COM components. While Viva does not presently have the ability to create COM components, it does enable you to use existing COM components, including OLE and ActiveX controls.

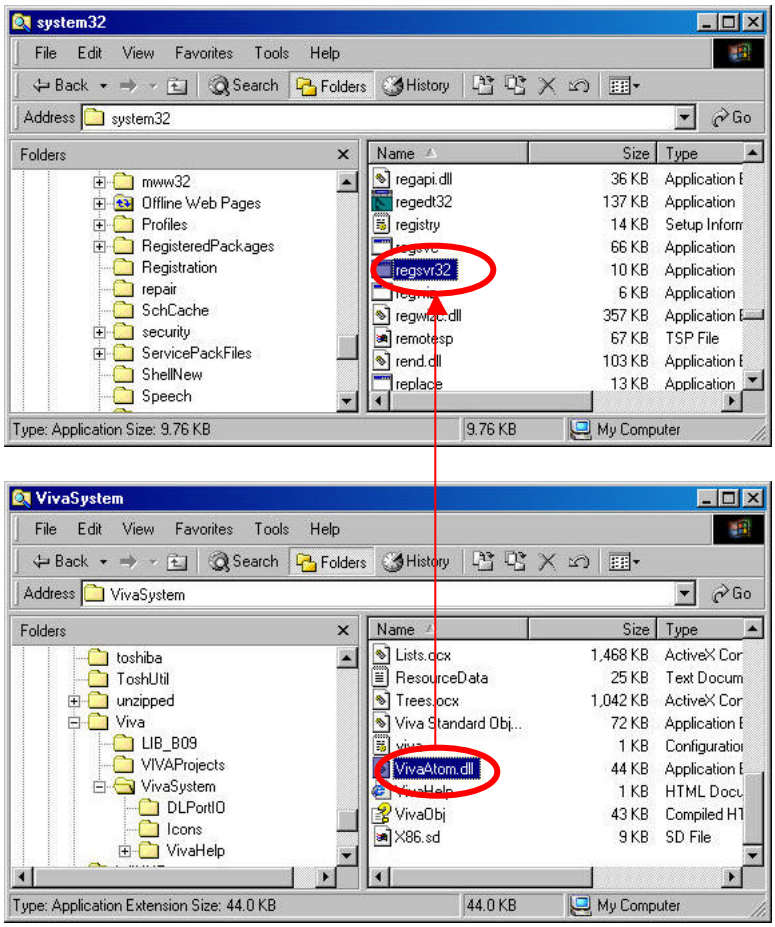
The functionality of COM components is implemented in Dynamic Link Libraries (DLL files) and ActiveX Control files (.OCX files). These files contain a “type library.” A type library contains one or more COM objects (perhaps better known as Classes), ActiveX controls, data type definitions, and/or enumerations. To use one of these files, it must be registered on your system, whether you are developing or executing them.

Registering DLL/OCX

To make a type library usable, you must register it on your system. To register a DLL or OCX file on your system, you must run the program "Regsvr32.exe" located in the Windows/System directory. The path and filename of the file that you want registered is passed a parameter on the command line. You can accomplish the same thing by dragging the icon of the DLL or OCX file over the Regsvr32 icon. See the following illustrations for examples of the two ways to register COM objects.



Command Line COM Registration

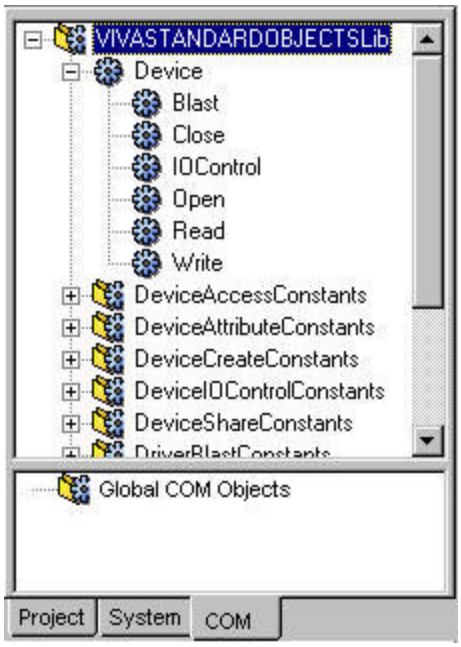


Drag & Drop COM Registration

Using COM in Viva

To view a list of existing type libraries on your system, and the components and definitions that they contain, activate the Object Browser. To activate the Object Browser, choose Object Browser from the Tools menu.

You can include any of these libraries for use in Viva through the Object Browser window. Once you have included the type libraries containing the components that you want to use, you can instantiate these components and access their member variables, properties, functions, and events. Use of the included COM libraries is done in the same manner as other objects: use the class and object trees found on the COM tab of the object tree pane.



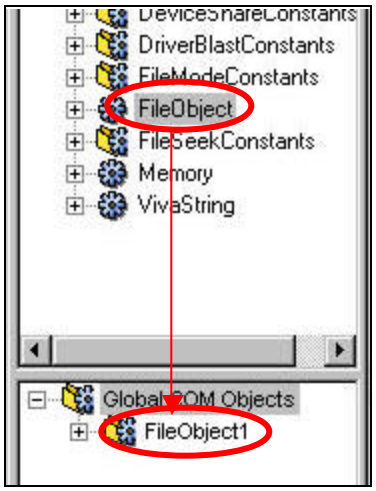
COM Tab/Object Tree

The COM class tree, the top part of the COM Tab/Object Tree figure above, contains the Viva standard object library. Each class and enumeration within that library is shown as a branch of the tree. For example, Device and DeviceAttributeConstants are both branches (classes) of the Viva standard object library, while Open and Read are both enumerations of the Device class.

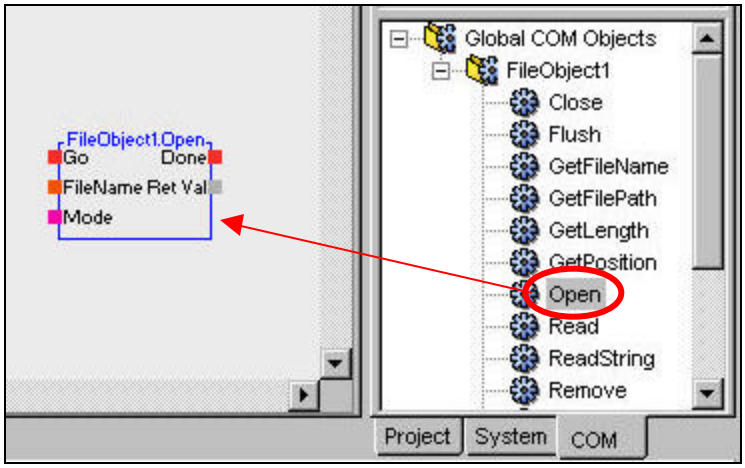
Each branch that represents an enumeration can be used by dragging and dropping it onto an application sheet. Placing an enumeration on an application sheet will instantiate an input having a name and constant value appropriate to the enumeration entry represented, as illustrated in the following example.

COM Object Enumeration

Dragging an object that represents a class onto the COM object tree (bottom part of the right-hand pane) will add an object to the COM object tree that represents a global instance of that class (see COM Class Global Instance figure). This new object will contain detailed object members for each of the class's properties, functions, and events. Dragging one of these object members onto the application workspace creates a reference to that member. This is illustrated in the COM Class Member Reference figure. The COM object tree also contains objects for all COM forms and their controls.



COM Class Global Instance



COM Class Member Reference

At runtime, except for events, these members can be accessed by sending a high signal to the 'Go' node of the member. The behavior of these members is listed in the following table.

Member	*Behavior*
Properties	The present value of the variable is sent out the output node named "Current". Any value sent to the node named "New" will then become the new value of the variable.
Functions	Each input node, other than Go, if any, serves as a parameter to the function. If the function has a return value, then it will be sent out the second output node before the Done node fires. If any node corresponding to a non-optional parameter is not hooked up, then an exception will be thrown when the function is invoked.
Events	Each output node, other than the Done node, is a parameter passed with the event. When a COM event fires in Viva, all events generated on attached transports are forced to process immediately. Awareness of this information will enable you to avoid infinite loops.

COM Class Member Behaviors

Once a member has completed its execution, the ‘Done’ node sends out a high state. Also, sending a low state to the ‘Go’ node will cause the member to send a low state from the ‘Done’ node.

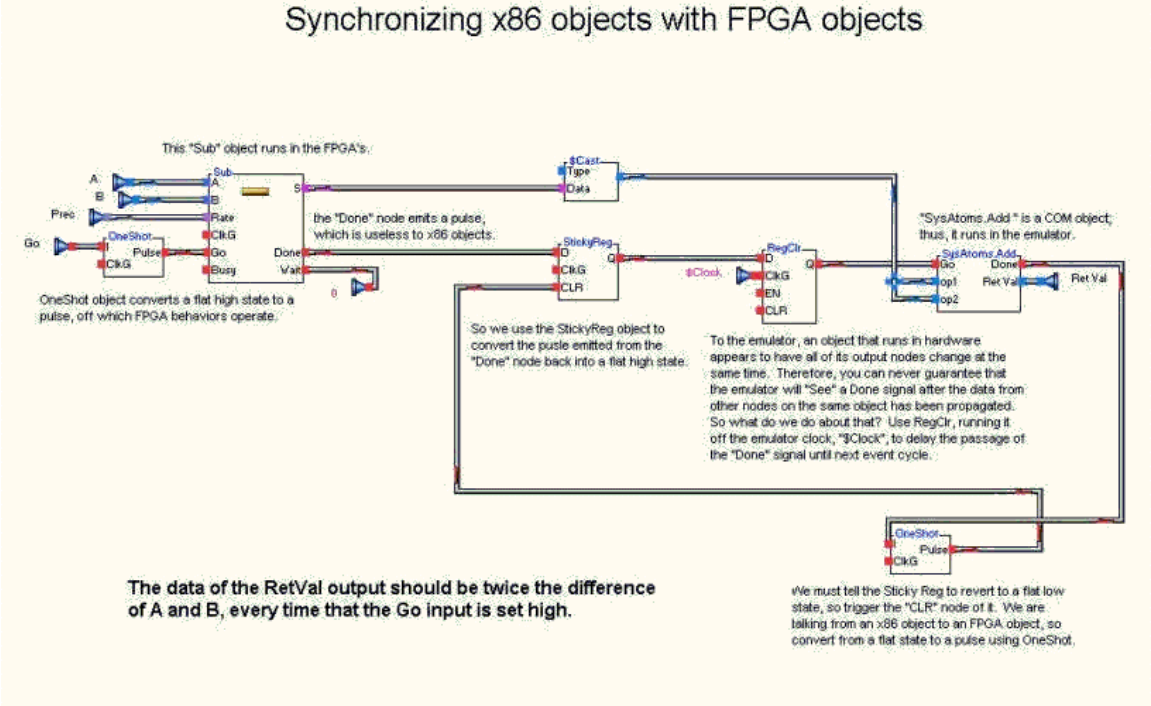
X86 Versus FPGA Applications

There is an important difference between the execution of application models that target the X86 processor and those that run on FPGAs. On FPGA-targeted models, the ‘Go’ node should be in a high state for a single machine clock cycle (referred to as a "pulse"); the ‘Done’ node also generates a pulse.

Objects in the X86 system work differently from those targeting an FPGA system. They require the ‘Go’ node to go high and stay high (referred to as a "flat" state), until the system processes it. Likewise, the ‘Done’ node does not merely generate a pulse, but is set and left in a high state, until the ‘Go’ node is set to a low "flat" state; the ‘Done’ node will then be set to a low "flat" state when its events are processed.

To bridge the gap between these two execution models, CoreLib provides two objects: OneShot and StickyReg. The OneShot object generates a pulse through its output node when it receives a high “flat” state in its input node. The StickyReg object emits a "flat" state from its ‘Done’ node when the D node receives a pulse.

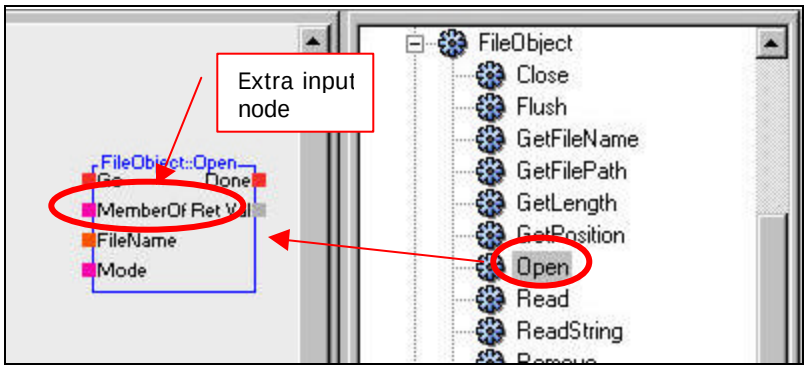
An example of the usage of OneShot and StickyReg is shown in the following figure.



x86/FPGA Application Example

Dynamic COM Objects

Dynamic COM members, which can be referenced by dragging an object representing a member from the COM class tree (upper part of the right-hand pane) rather than the COM object tree, function in the same manner as regular object members, except that they are not associated with a specific instance of a class. These objects will have an extra input node, 'Member Of', that is used to specify which object they apply to.

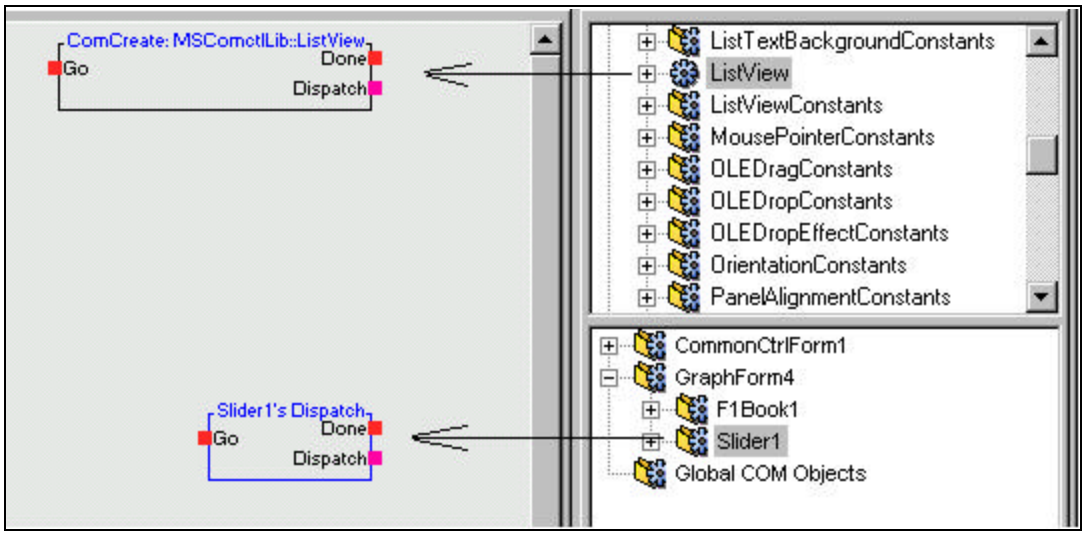


Dynamic COM Class Object

The input to this type of node is a pointer to a class specific 'Dispatch' appropriate to the class to which the object member belongs.

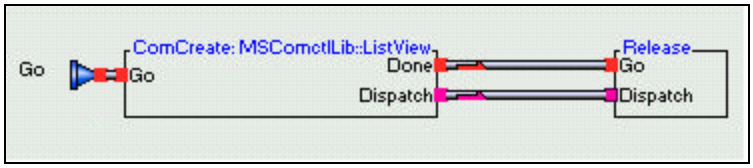
To acquire the dispatch pointer of an existing COM object, use a Dispatch object, which you can generate by dragging a node representing a COM object in the COM object tree onto the application editor. Once given a 'Go' signal, it will send the dispatch pointer out the node named 'Dispatch'.

You can also instantiate COM objects at runtime using a ComCreate object. Dragging a class object representing a COM class from the COM class tree onto the application editor can generate a ComCreate object. Note the name of the newly created object: "ComCreate: ", then the name Library ID for the library that the class resides in, then "::", then the class name. The following figure illustrates how the class object will appear in a save file.



COM Class Objects Example

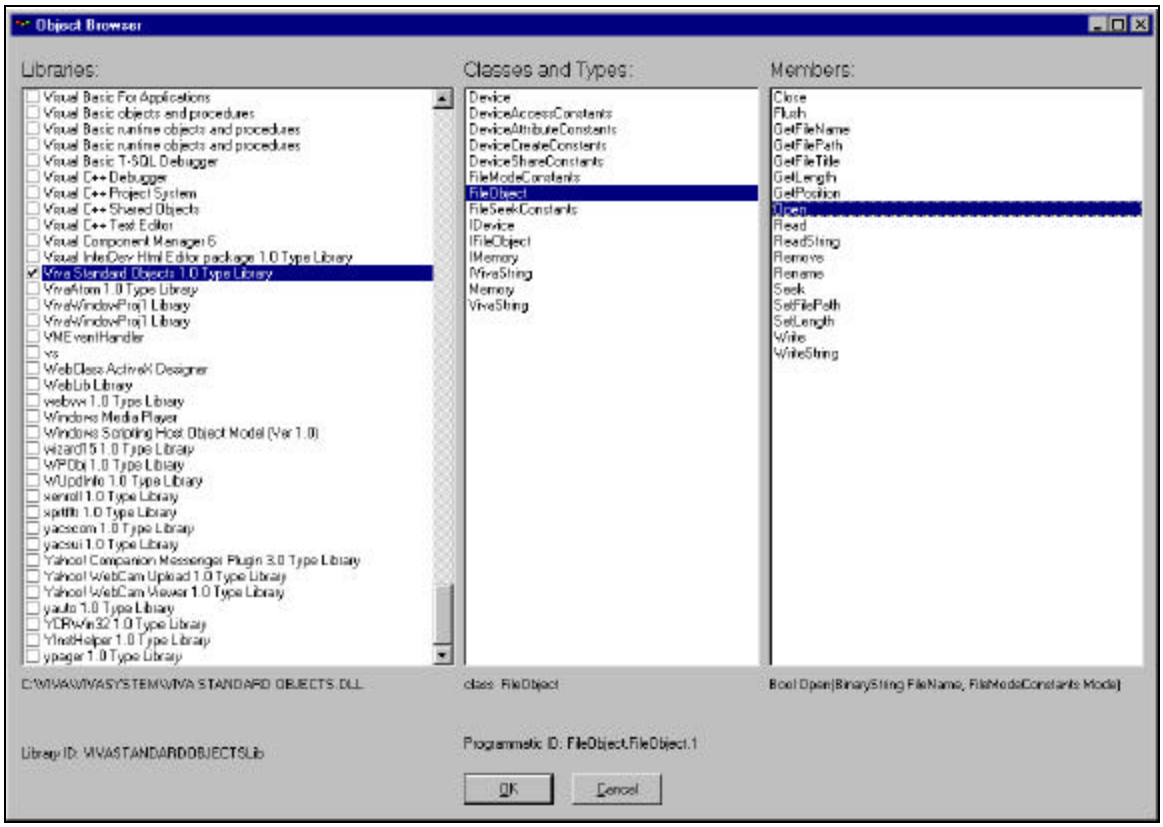
COM objects created at runtime are not automatically released. They can outlive their calling program, and even be used in other processes. You must use a Release object to destroy them. This object is located in the Primitive Objects branch of the Project objects tree. Simply pass in the dispatch pointer and send it a 'Go' signal.



COM Object Destruction

Object Browser

The Object Browser (Tools menu) contains a list of existing type libraries on your system. These libraries also contain lists of the components and definitions included in each library.



Object Browser

The ‘Libraries’ list box (left-hand pane) contains the list of type libraries registered on your system. If you select one of these items, the list of its components and definitions appears in the ‘Classes and Types’ list box (middle pane). If you select one of these, the list of its members appears in the ‘Members’ list box (right-hand pane).

You can include a Library for use in VIVA by "checking" its entry in the Libraries list box. Simply click on the check box next to the desired library to include it. The library can likewise be un-included by un-checking the entry.

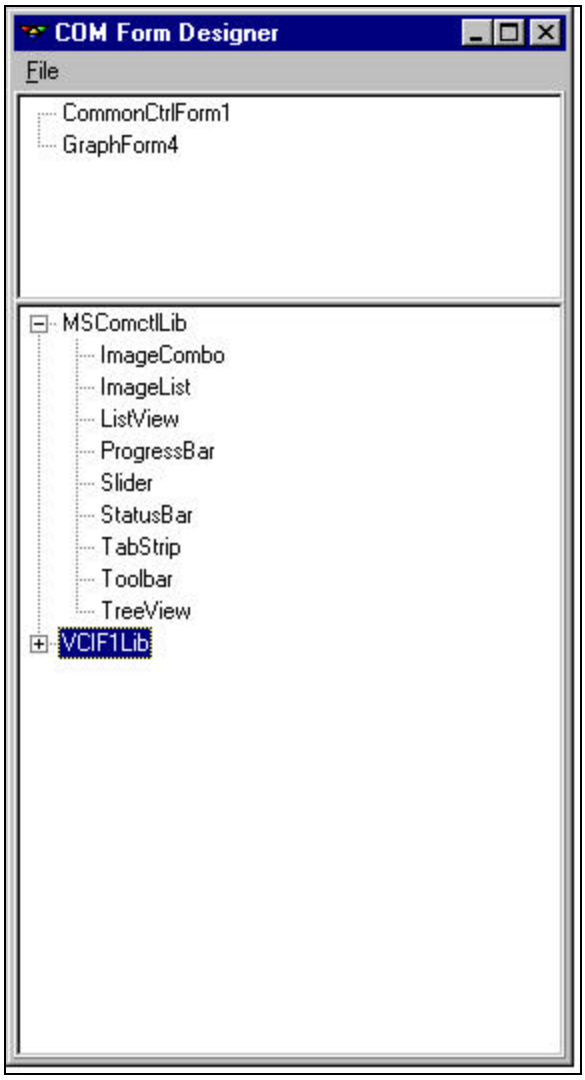
Note: You must click **OK** to apply any changes you make to the selection, or de-selection, of any type libraries.

The caption immediately below the Libraries list box is the path and filename of the actual file that contains the Library, and the caption below that is the Library ID, if it exists, for the library selected. It is used in Viva save files, prefixed to a class name, to indicate that it is located in this library.

The caption immediately below the Classes and Types list box is simply the type of the selected entry, followed by its name while the caption below that is the Windows Programmatic ID (not to be confused with Viva Programmatic ID, used in save files), if it exists, for any class selected. If you are not already familiar with programmatic IDs, know that any class that does not have a Windows Programmatic ID cannot be directly instantiated via the COM interface.

COM Form Designer

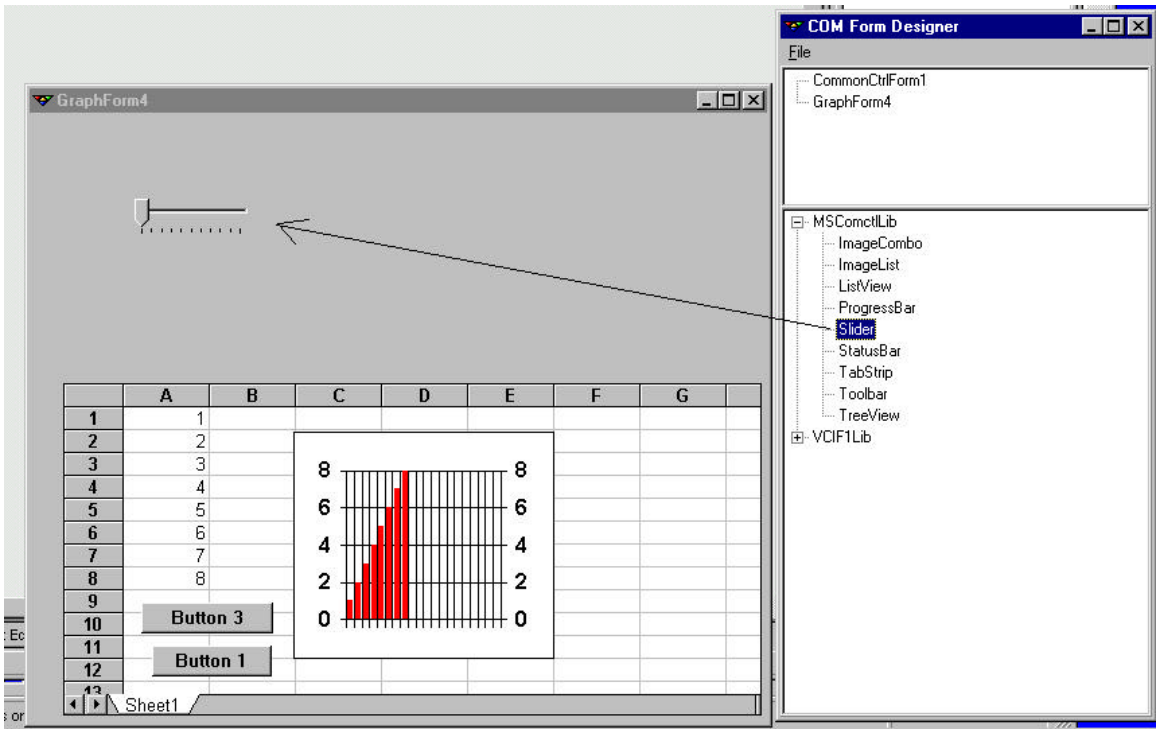
The COM Form Designer (Tools menu) is a utility that is used to create forms that host ActiveX Controls. The COM Form Designer File menu contains New, Open, Save, and Save As commands, along with a recently used file history.



COM Form Designer

The forms tree (top pane) contains a list of all forms in the current project. **By default, there are none**. Selecting the File/New menu command will create a new form. Clicking on a branch in this tree will display its corresponding form. Pressing the delete key will close the form corresponding to the selected branch.

The ActiveX controls tree (bottom pane) contains a list of available ActiveX controls for the libraries that have been selected for use in this project using the Object Browser. Dragging one of these branches onto an ActiveX form will instantiate that control at the location specified. The control can then be resized or deleted. To delete the control, simply select it by clicking it and then pressing the delete key.



Right-clicking a control brings up an available actions list for the selected control. This list may or may not contain entries.

When a Viva file is saved, any forms that are dependant on that file are also saved, even if they did not originate from the current project.

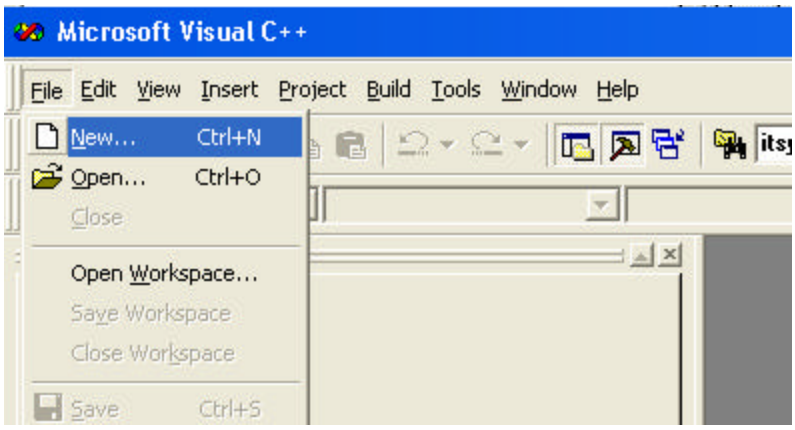
All forms and controls are listed in and can be used from the COM object tree.

Creating your own COM components in Microsoft Visual C++ 6

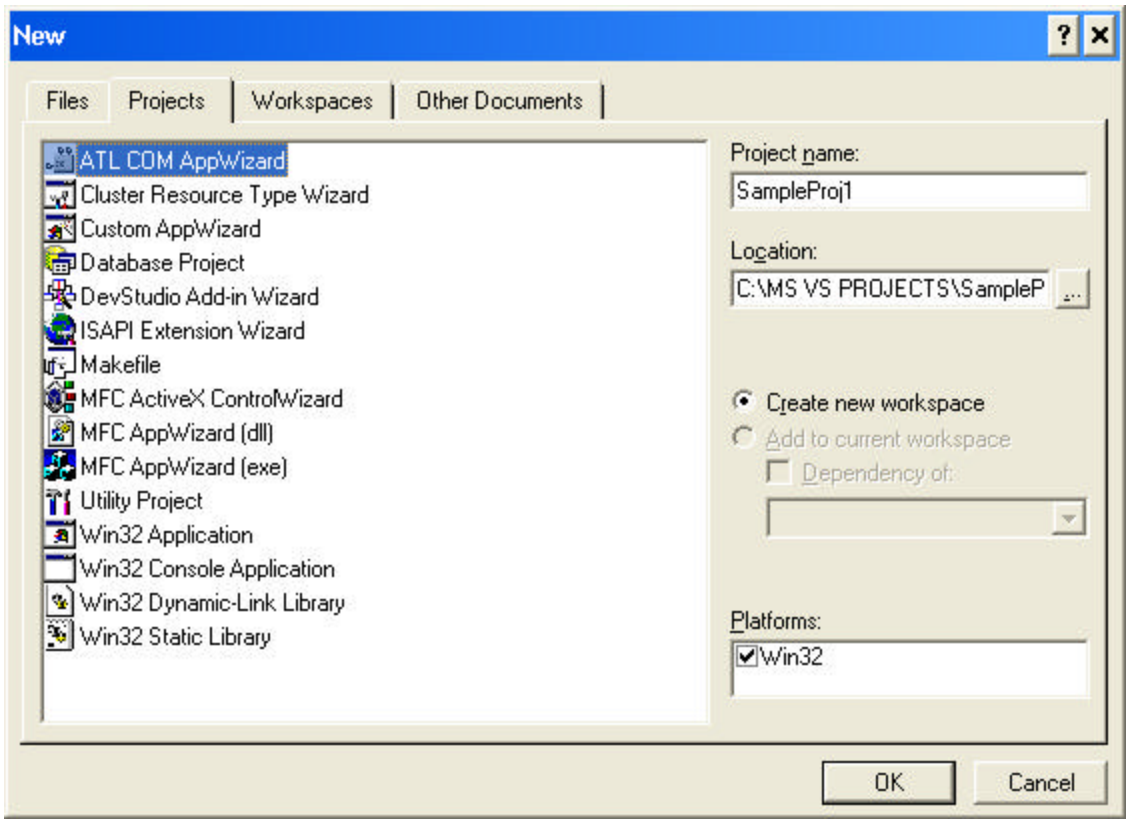
Creating Simple COM Components

This part will step you through creating a COM library (DLL), creating a simple COM component within that library, and adding functions and an event to the COM component.

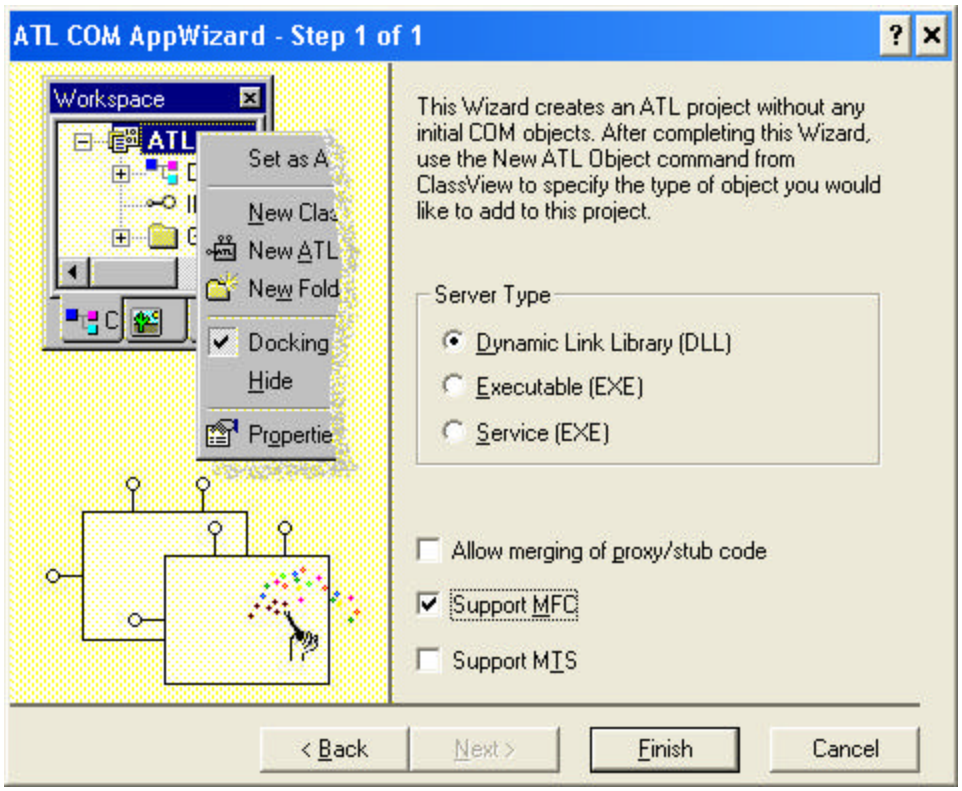
First, you will need a project for generating the DLL.



Create an “ATL COM AppWizard” application, highlighted in the graphic below:



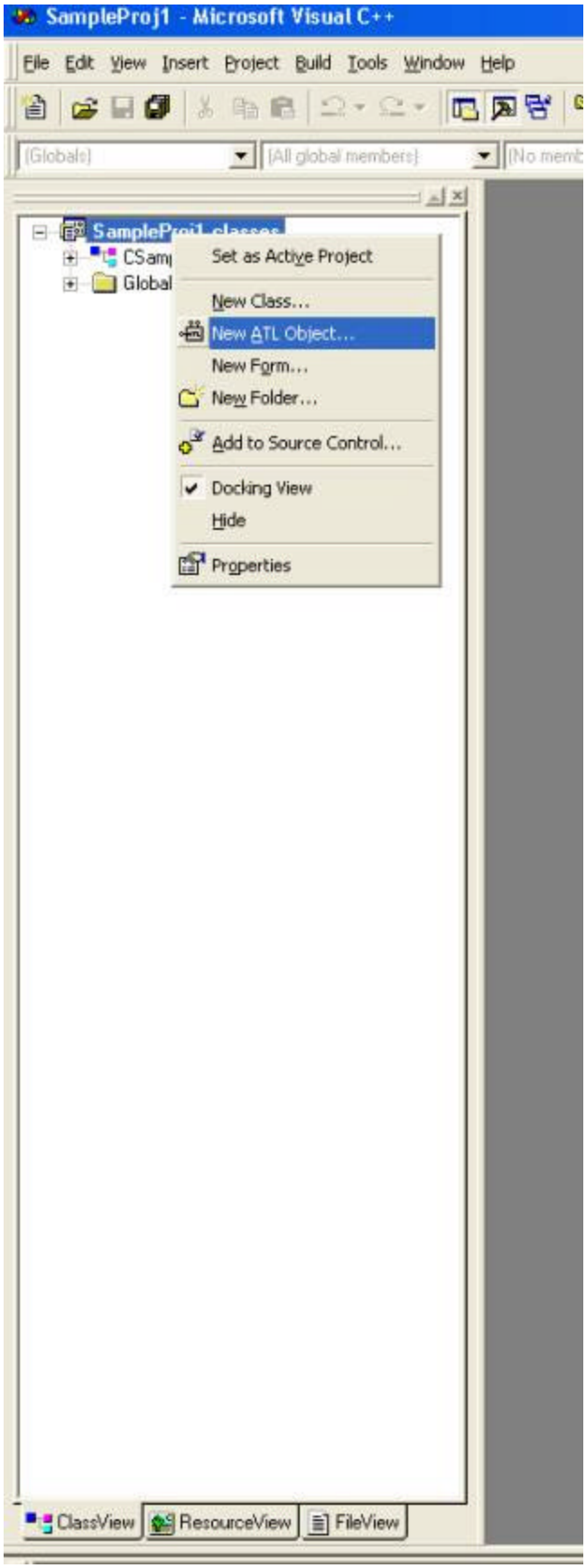
Once you name your project (SampleProj1 in this case), clicked OK to invoke the AppWizard for the project. From the interface form, select the “Support MFC” checkbox.



Click Finish. A dialog box appears that indicates what files it will create for the project. Click OK.

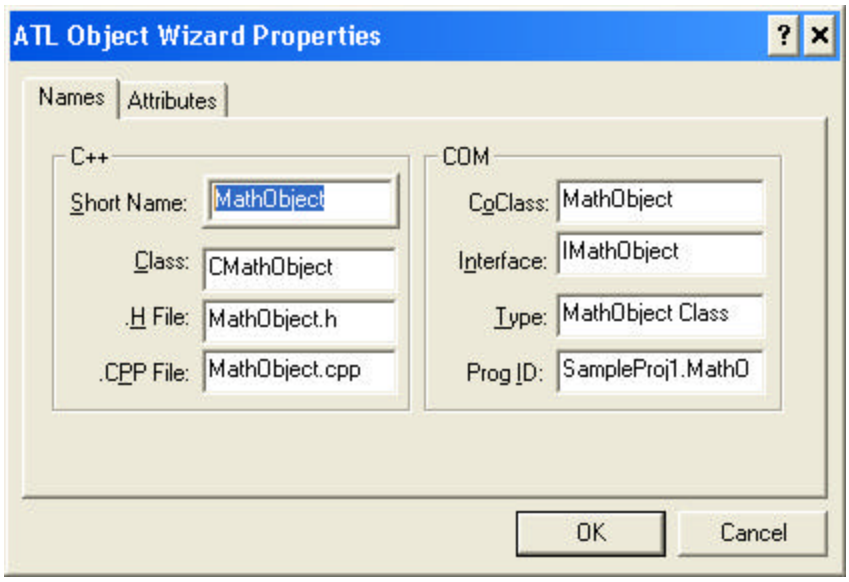
You have now created the framework for creating your COM Library and any components that you want to add to it.

To add a COM component, switch to the Class View tab in the workspace window. Right-click the topmost node and select the New ATL Object… option from the menu that appears.

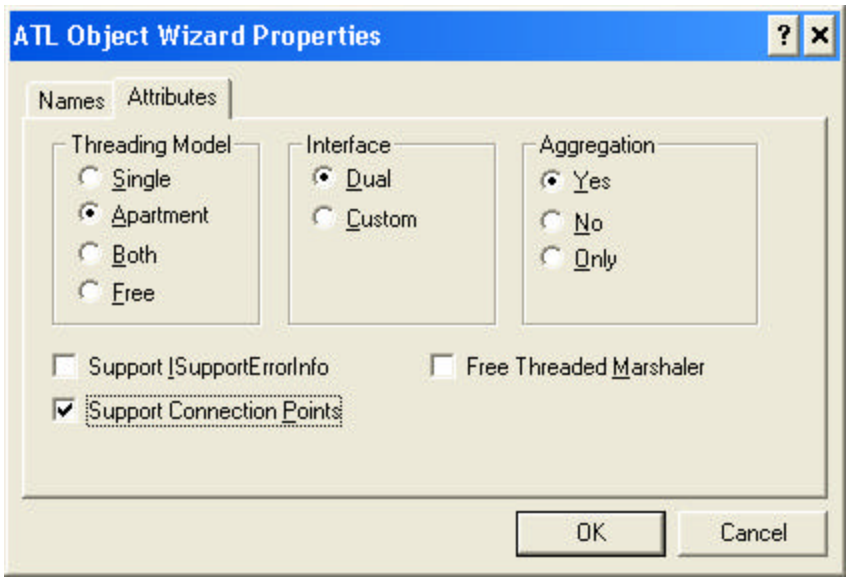

SampleProj1 - Microsoft Visual C++
File Edit View Insert Project Build Tools Window Help
(Globals)
(All global members)
Set as Active Project
New Class...
New ATL Object...
New Form...
New Folder...
Add to Source Control...
Docking View
Hide
Properties
ClassView
ResourceView
FileView

From the dialog that appears, select Simple Object in the right pane and click Next.

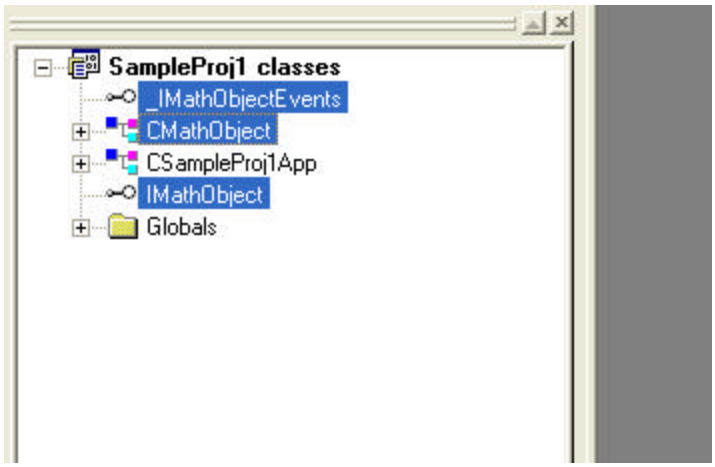
You must now name the component that you want to create. You only have to fill in the Short Name: box. Name it MathObject so that future references to it will fit in the context of your project.



Click the Attributes tab. Select the Support Connection Points option, which will create a Dispatch Interface that you will use to create an Event for this component. Click OK.



One new class and two new “interfaces” are created in your project to support the new component. The new class and interfaces are highlighted in the following graphic. An interface in this context is the specification and mechanism by which other applications can interact with the component, via COM. This interface can contain functions, properties, and events that you can use to interact with the component.



CMathObject is the class in which the implementation code for this component will be placed, a process discussed later.

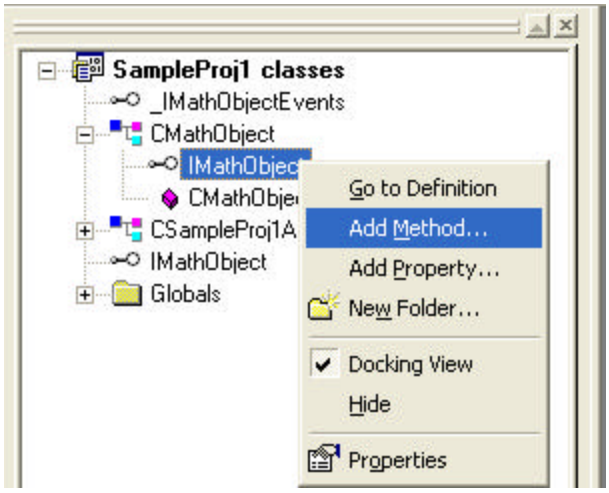
IMathObject is the interface used by other applications to invoke functionality of this component. This is known as an “Incoming” interface. Functions and properties can be placed in this interface.

_IMathObjectEvents is the interface that can trigger events, which are special functions that are used to talk to other applications. This is known as a “Outgoing” Interface. A function can be set up in another application that “listens” for an event from a component, and is called whenever the component triggers the event.

To add a function to the component, expand the CMathObject node. Notice that it has a node named “ImathObject,” which you may note, is the same name as the node for the IMathObject

interface explained earlier. The CmathObject node, subordinate to the CMathObject class node, is for the Interface implementation. The other IMathObject node is for the interface definition. The meaning and use of this is explained later.

Right-click the IMathObject node that is subordinate to the CMathObject class node, and select "Add Method…" from the menu that appears. There is an important difference between doing this and adding a method to the CMathObject class: only methods added to the interface implementation for this class will actually be exposed via COM.



A dialog appears asking you to enter a function name and a parameter list. Note the following about functions on interfaces:

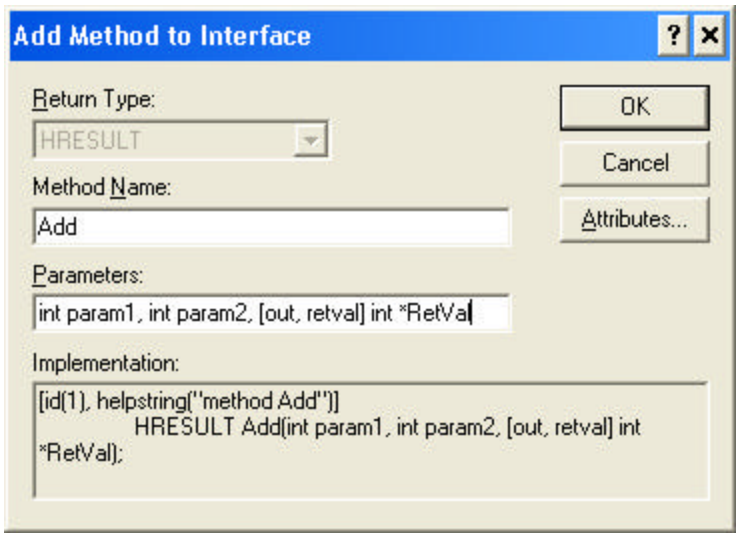
- Do not give the function a name that is ambiguous with any identifier in a C++ library that you have included.
- The valid parameter types for COM in Visual C++ are:

Data Type	Description	Default sign
VARIANT_BOOL	COM standard 32-bit boolean type. false = 0; true = -1	unsigned
byte	8 bits	(not applicable)
double	64-bit floating point number	(not applicable)
error_status_t	32-bit unsigned integer for returning status values for error handling.	unsigned
float	32-bit floating point number	(not applicable)
handle_t	primitive handle type for binding	(not applicable)
hyper	64-bit integer	signed
int	32-bit integer. On 16-bit platforms, cannot appear in remote functions without a size qualifier such as **short**, **small**, **long** or **hyper**.	signed
long	32-bit integer	signed
short	16-bt integer	signed

small	8-bit integer	signed
wchar_t	16-bit predefined type for wide characters.	unsigned
BSTR	COM standard string type. Pointer to type wchar_t.	(not applicable)
VARIANT	Container of any COM supported type. Wrapper class is _variant_t	(not applicable)

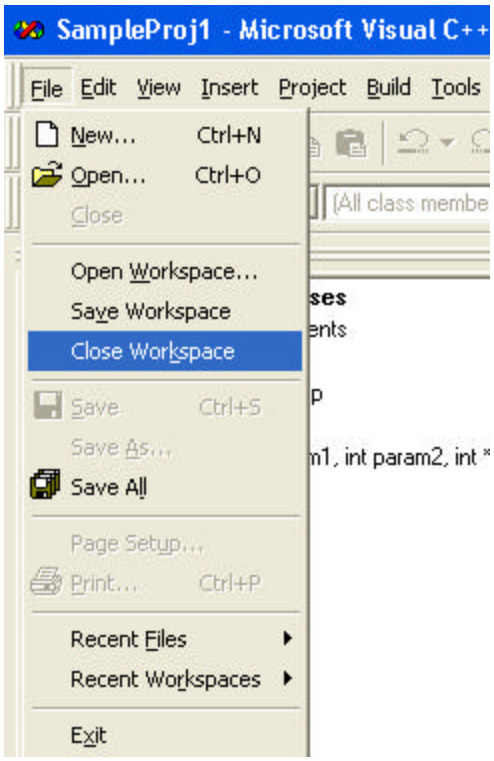
Parameter types can also be a pointer to any of the types listed in the previous table.

For this sample case, type the Method Name and Parameters as follows:

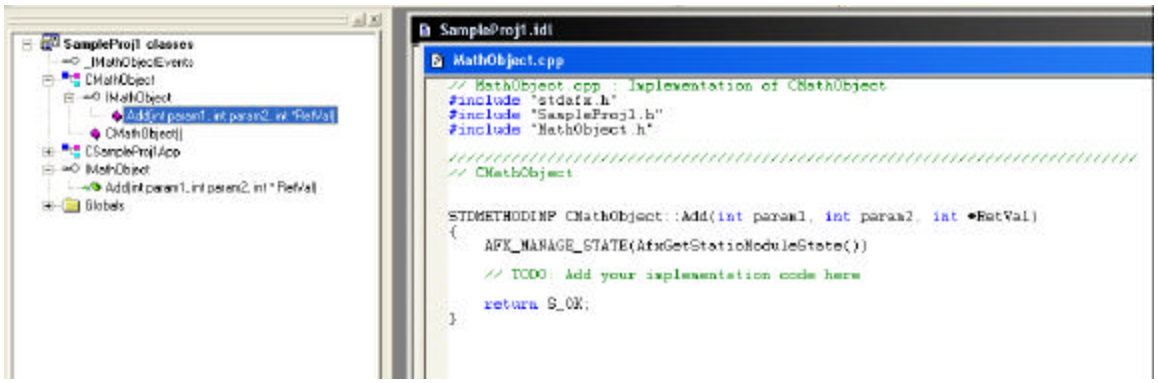


The [out, retval] prefix to the last parameter is the marker that it is the return value for the function. This can only be applied to the last parameter for a method, and must be a pointer to the type that you want to return from the method.

Click OK. To get the Visual C++ IDE to update the workspace window to reflect the changes, save all files, close your workspace, and re-open it.



After forcing an update, notice that new nodes have been added subordinate to both the interface implementation node and the interface definition node. Double-click the new node that is subordinate to the IMathObject interface implementation node (highlighted in the following figure) to view the code that implements the new function.



The function body has an AFX_MANAGE_STATE macro and a statement to return the status code S_OK.

Do not remove the AFX_MANAGE_STATE macro; it does something that needs to happen at the beginning of this function call. Also, it should always be the first line of code in this function body.

Do not remove the other line either; there is no need to return any code other than S_OK unless something wrong happened. There are many status codes that can be returned, but as a general rule, if you encounter an error condition, call the Error function, which is a member of CMathObject, which passes it a string that describes the problem and returns DISP_E_EXCEPTION. Most COM-utilizing programming environments throw an exception when this code is returned with the message passed in as a parameter to the Error function.

For this example, add the following line in place of the “TODO” comment:

*RetVal = param1 + param2;

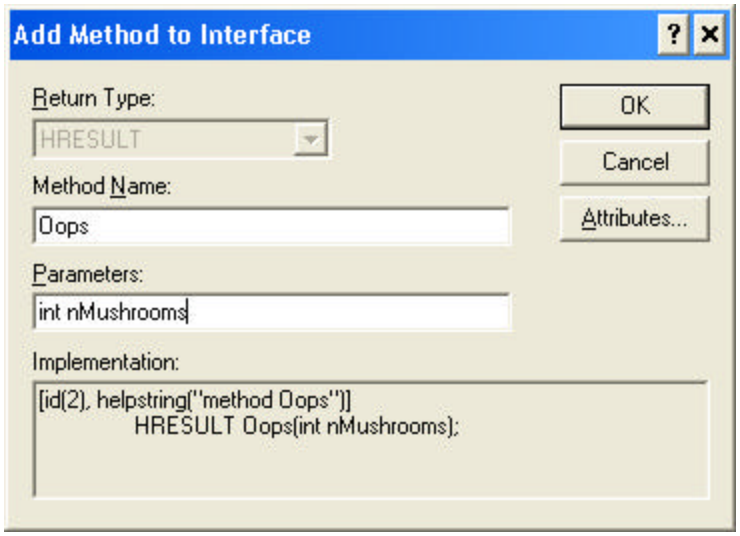
The RetVal parameter passed in by reference holds the value that will be returned from this function to program using this function. Do not confuse this with the status code returned. The S_OK is for the OLE automation system to notify you to proceed as intended. The program calling this function does not see this. The calling program would be notified, however, if you were to return some error code, such as DISP_E_EXCEPTION.

The function body should now read as follows:

```
STDMETHODIMP CMathObject::Add(int param1, int param2, int *RetVal)
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState())

    *RetVal = param1 + param2;

    return S_OK;
}
```

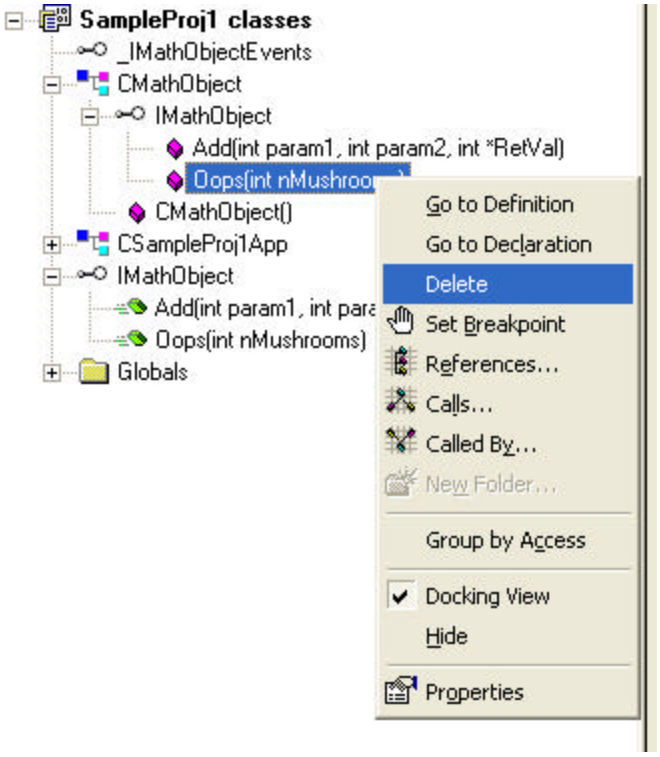
This function will, when called, simply add the numbers passed in by the first two parameters and set the value pointed to by RetVal to it, thus returning it to the calling process.

The following illustrates adding a new function to the IMathObject interface of this component.



Suppose you want to change the name of the function and end up botching the entire project beyond recognition of the Visual C++ AppWizard. The best way to deal with a bad function is to delete it and start over. Use the following procedure to remove a function from a component.

Right-click the node under the interface implementation (not definition), and select “Delete” from the menu that appears.

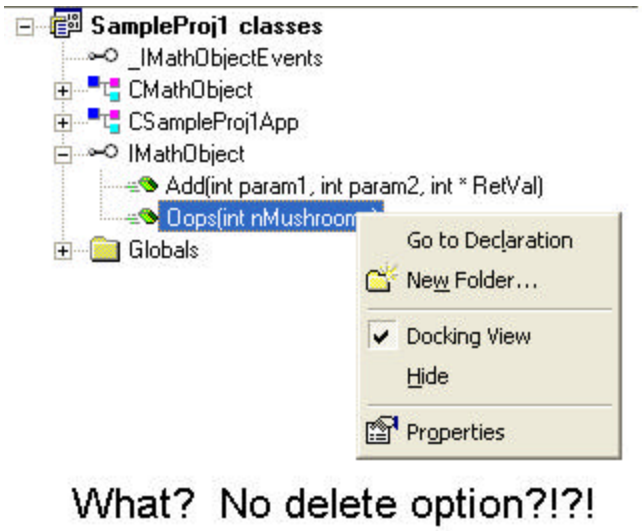


You will be told that the function body will be commented out, rather than deleted, and you will be asked if you really want to delete the function. Specify Yes. You will have to manually delete the function body to permanently remove it.

MathObject.cpp

```
// MathObject.cpp : Implementation of CMathObject
#include "stdafx.h"
#include "SampleProj1.h"
#include "MathObject.h"

/////////////////////////////////////////////////////////////////////////////
// CMathObject


STDMETHODIMP CMathObject::Add(int param1, int param2, int *RetVal)
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState())

    *RetVal = param1 + param2;

    return S_OK;
}

//DEL STDMETHODIMP CMathObject::Oops(int nMushrooms)
//DEL {
//DEL   AFX_MANAGE_STATE(AfxGetStaticModuleState())
//DEL
//DEL   // The purpose of life is to eat lots of candy
//DEL
//DEL   return S_OK;
//DEL }
```

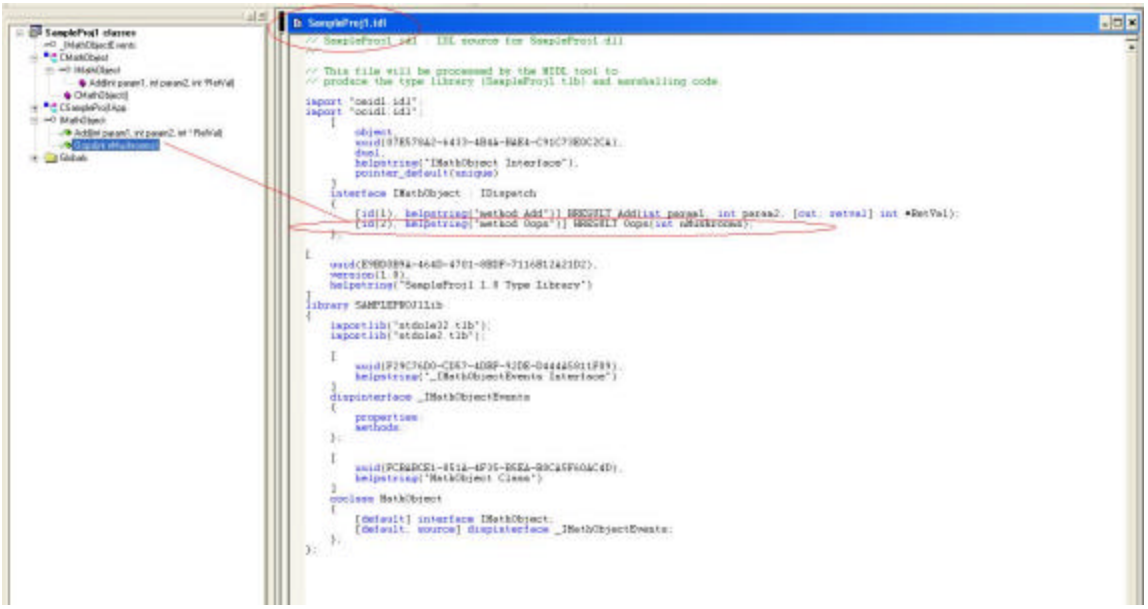
Note that the function still exists under the interface definition and there is no option to delete it.



What? No delete option?!?!

The interface definition links to the IDL file associated with the COM Type Library rather than the C++ code that implements its functionality. An IDL file describes the Type Library, including all of its dependencies, components, type definitions, and interfaces, including each of their members (functions, properties, and events). When you need to change an interface beyond what the AppWizard handles, you must edit the IDL file.

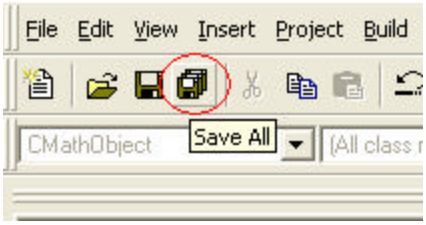
Double-click the “Oops” node to bring up the IDL file. This IDL file is also accessible via the “File View” tab of the Project window.

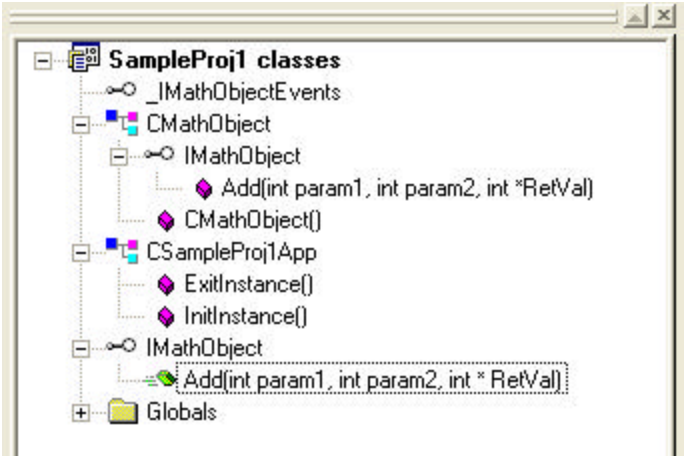


To remove the “Oops” function from the interface definition, simply locate and remove the line of code defining it (circled in red in the preceding figure). The IMathObject interface definition should now read as follows:

```
interface IMathObject : IDispatch
{
    [id(1), helpstring("method Add")] HRESULT Add(int param1, int param2, [out, retval] int *RetVal);
};
```

You will have to save all files before the tree in the Project window will be updated to reflect this change.





It's finally gone!

Now, select "Rebuild All" from the "Build" menu to compile the Type Library. This will both build and register the DLL.

You built a COM Type Library!

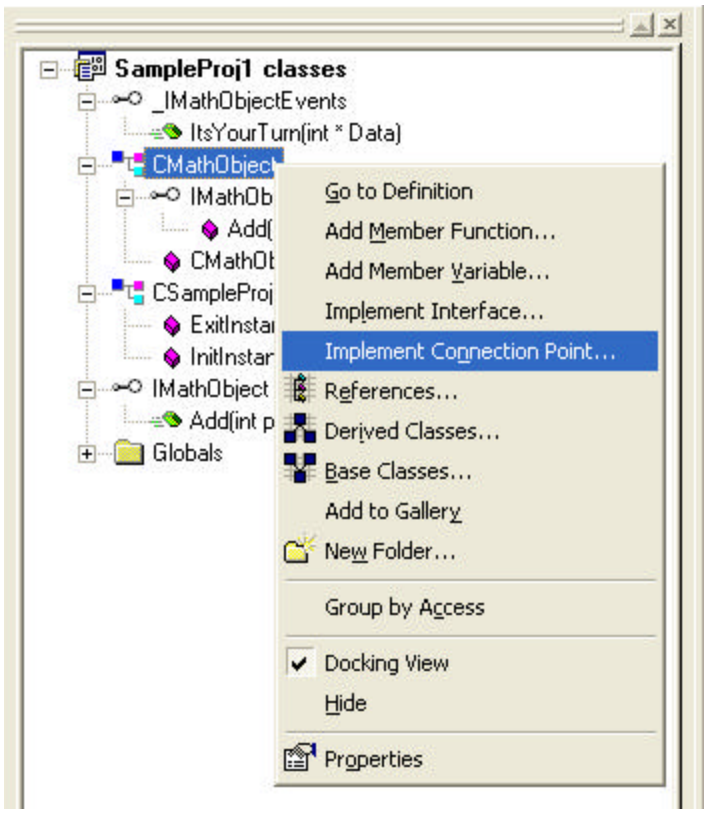
Events

Any function that is added to the _IMathObjectEvents interface can be called by this component, not by a program using this component. The program that this component resides in will experience and event when this component calls such a function.

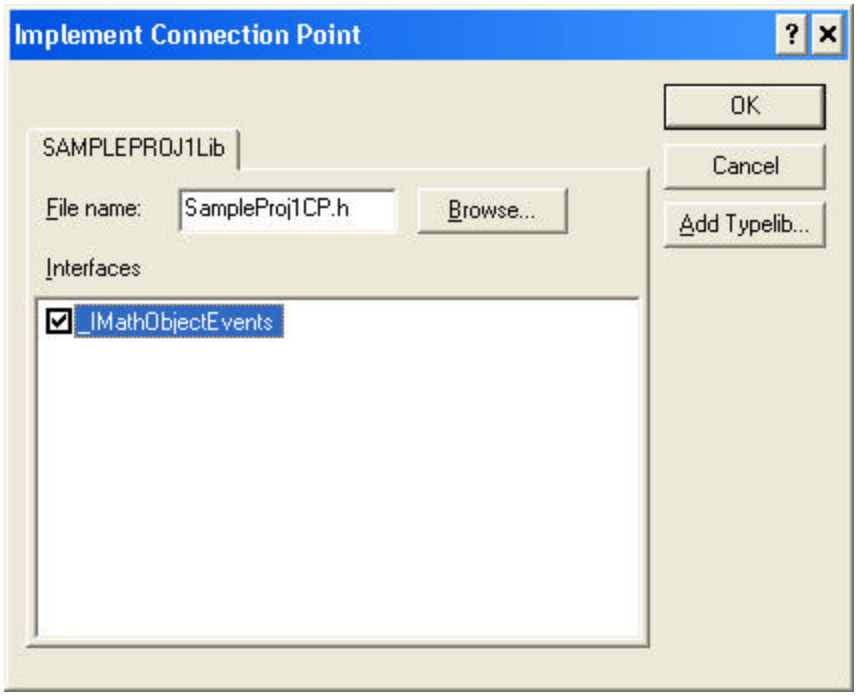
Events are normally used to notify the process hosting a component that a pre-determined condition has been met on behalf of the component. In this example, we will build an event that can call into another program that is set up to deal with it, and actually get results back from that program! This assumes that the program does its job.

IMathObject has both an interface definition and an interface implementation, whereas _IMathObjectEvents has only the former because we haven't told the CMathObject class to implement this interface yet. This process was done automatically for ImathObject but must be done manually for _IMathObjectEvents.

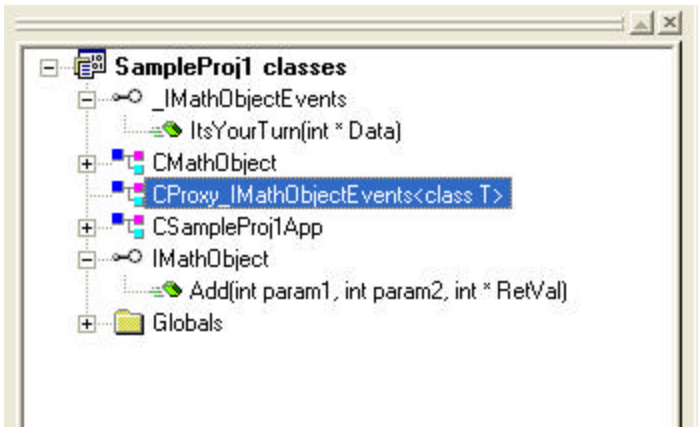
Create an implementation for this interface in CMathObject by right-clicking its node in the Project window, and selecting "Implement Connection Point" from the menu that appears.



Check the box that reads _ImathObjectEvents. Click OK.

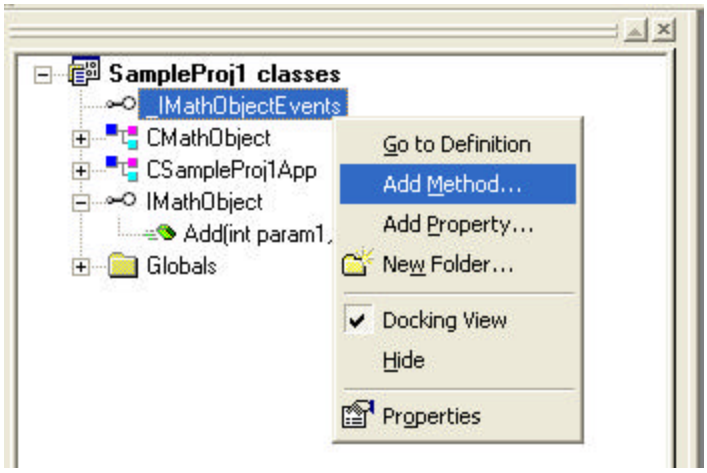


You now have a new class: CProxy_IMathObjectEvents. This class will contain the implementation code for the IMathObjectEvents interface, and CMathObject can directly use its members because CMathObject now inherits additionally from it.

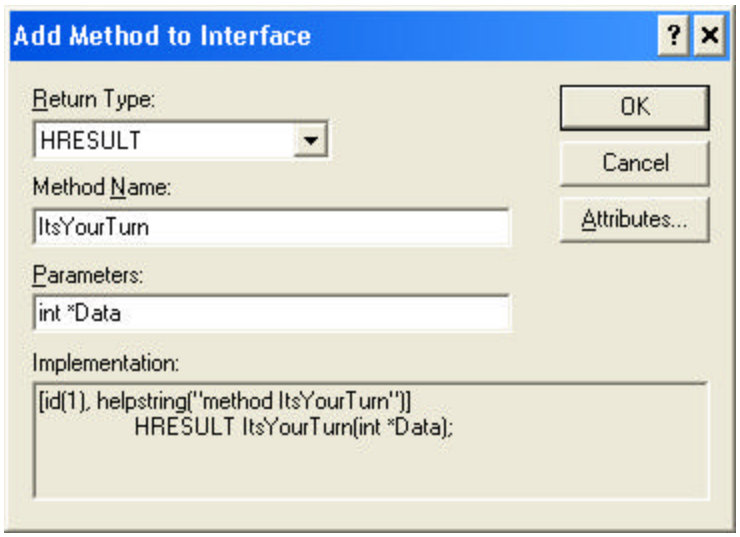


Incidentally, you have to repeat this process every time that you change the IMathObjectEvents interface to get the implementation to update.

Add a function to the _IMathObjectEvents interface in the same manner as you added the Add function to the IMathObject interface.

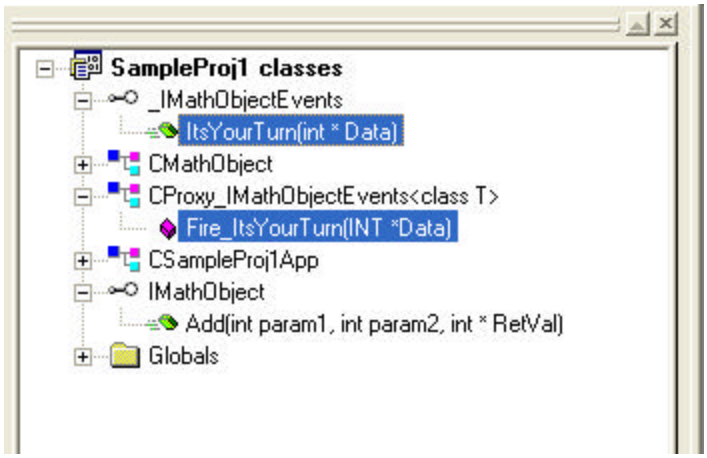


Give it the name and parameter list indicated below:



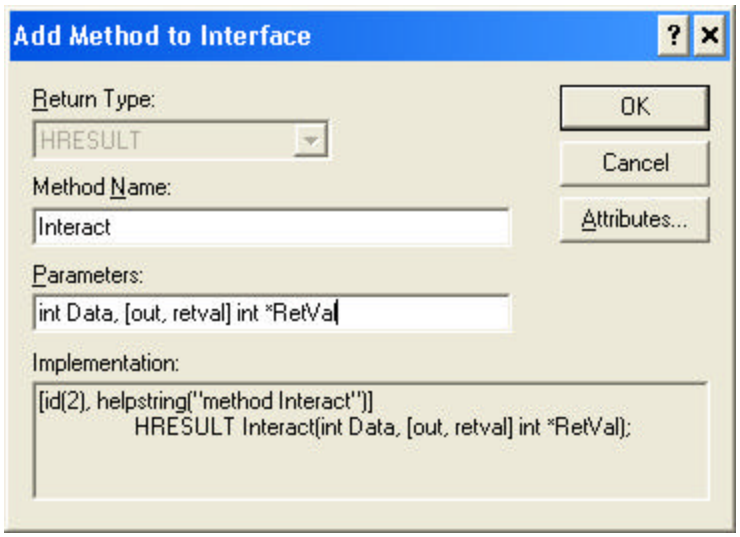
Note that the parameter is passed by reference. This is so that the process that receives this event can modify the argument passed, which allows you to call into the hosting application and receive data back from it. You will then have two-way communication with the host.

Now tell CMathObject to re-implement _IMathObjectEvents.

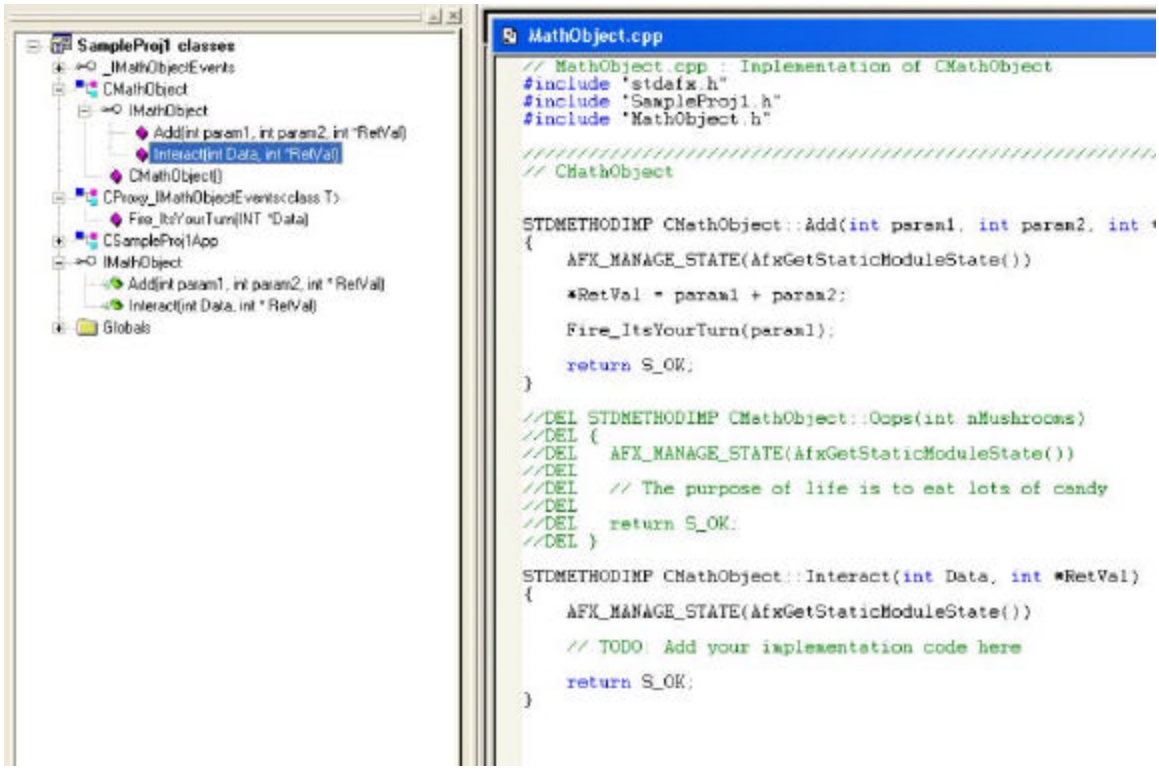


You now have a Fire_ItsYourTurn(INT *Data) node subordinate to the CProxy_IMathObjectEvents class node. It represents your implementation of the ItsYourTurn event.

Now, add another function to IMathObject to invoke the event. Define the function as follows:



Go to the function's C++ implementation code by, once again, double-clicking the node that represents it, subordinate to the IMathObject node.



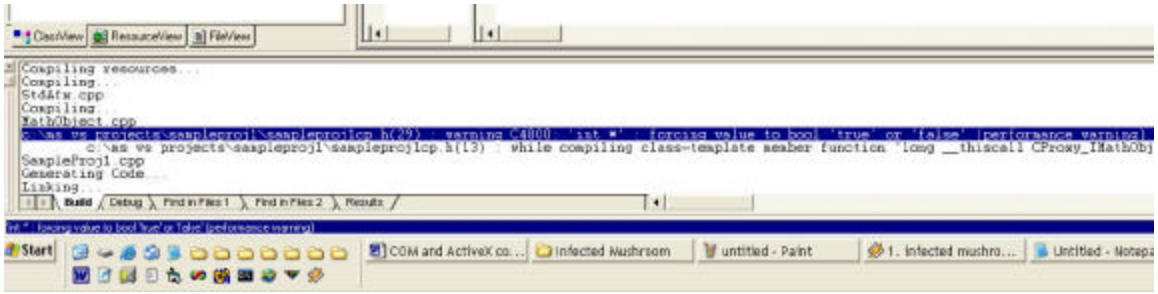
Now change the function body of Interact to look like this:

```
STDMETHODIMP CMathObject::Interact(int Data, int *RetVal)
{
        AFX_MANAGE_STATE(AfxGetStaticModuleState())

        // Invoke the event!
        Fire_ItsYourTurn(&Data);
```



```
        // And for kicks, lets display Data after our hosting application has had a chance
        //      to process it.  What did it do to Data?  Who cares.  It could do anything; the
        //      important thing is that it's two-way interaction, baby!  If you don't care where
        //      you end up, then it doesn't matter which way you go.
        char msg[20];
        itoa(Data, msg, 10);
        AfxMessageBox(msg);

        *RetVal = Data;

        return S_OK;
}
```

Now, rebuild and…

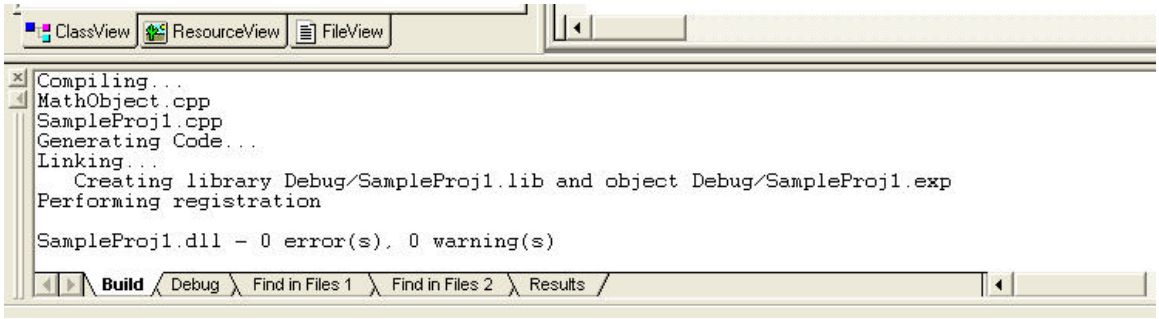


This warning indicates that the parameter you passed as a pointer is going to be converted to a [bool] so the address of Data will be either 0x00000000 or 0x00000001. Where the code goes to stuff Data into a variant, to conform to COM dispatch protocol, there is no [int*] override in the Variant class. Thus, the compiler do coerces your [int *] into a [bool].

Note the cast circled in red below:


```
template <class T>
class CProxy_IMathObjectEvents : public IConnectionPointImpl<T, &DIID__IMathObject
{
    //Warning this class may be recreated by the wizard.
public:
    HRESULT Fire_ItsYourTurn(INT * Data)
    {
        CComVariant varResult;
        T* pT = static_cast<T*>(this);
        int nConnectionIndex;
        CComVariant* pvars = new CComVariant[1];
        int nConnections = m_vec.GetSize();

        for (nConnectionIndex = 0; nConnectionIndex < nConnections; nConnectionInc
        {
            pT->Lock();
            CComPtr<IUnknown> sp = m_vec.GetAt(nConnectionIndex);
            pT->Unlock();
            IDispatch* pDispatch = reinterpret_cast<IDispatch*>(sp.p);
            if (pDispatch != NULL)
            {
                VariantClear(&varResult);
                pvars[0] = (int) Data;
                DISPPARAMS disp = { pvars, NULL, 1, 0 };
                pDispatch->Invoke(0x1, IID_NULL, LOCALE_USER_DEFAULT, DISPATCH_ME
            }
        }
        delete[] pvars;
        return varResult.scode;

    }
};
```

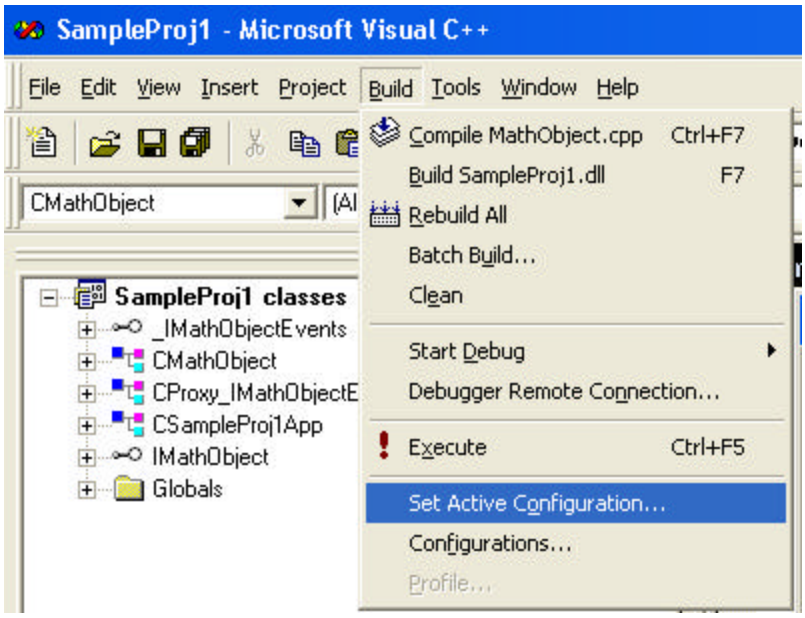
Once that is changed, rebuild again.

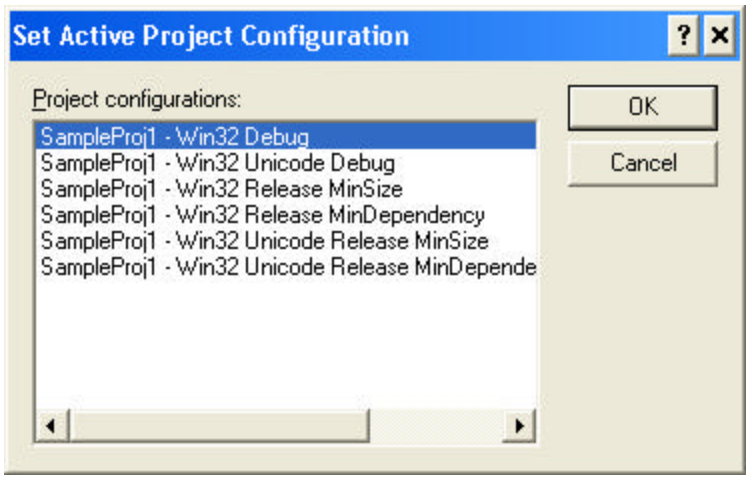


No warnings.

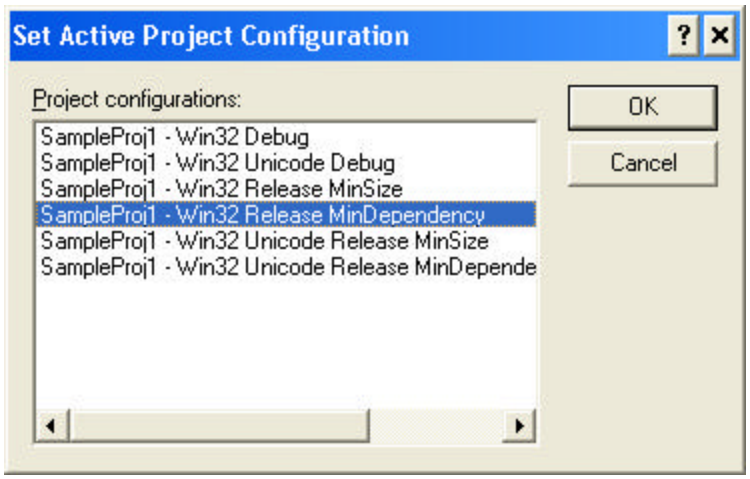
Don't mind that the type cast to is signed, whereas an address is unsigned. All the bits will still be there.

The DLL that you have created cannot be used on another machine that does not also have Microsoft Visual C++ 6 installed, as is, because the "active build configuration" is set to Win32 Debug, which is default. To amend this problem, select the "Set Active Configuration" option from the "Build" menu:





Select “SampleProj1 – Release MinDependency”, as shown below. Click OK.



Now when you rebuild, you will have a self-sufficient DLL that can run on any machine that has Windows 95 or later running on it.

For a listing of tutorials on creating COM/ActiveX components in Visual C++ and other languages, point your browser to the following URL:

http://www.guruischool.com/index.jsp

For help on creating ActiveX controls in Visual Basic, see:
http://msdn.microsoft.com/vbasic/techinfo/training/activex.asp

CURRENT FILE I/O IMPLEMENTATION

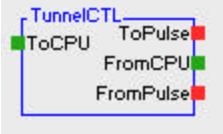
The File I/O object example resides in the CoreLib tree group. The driver for the HC machines supports sending data in 2KB chunks. To take advantage of this, the $File I/O object is overloaded in the system descriptions with an X86 version included by default. The parameters are described in the object documentation shown on the Object Attributes screen. If the example takes too long to compile, change the size on the Queues by changing the bitlength of the Size input.

File I/O

Viva File I/O is based on communication of the host machine with the FPGA hardware across the PCI bus. The File I/O in Viva consists of streaming data across the PCI bus. The bus communicates in 32-bit chunks using the Viva device driver. All Viva communication across the bus is handled by the TunnelCTL object. The following demonstrates how the TunnelCTL object is used to allow the host CPU to communicate with the FPGAs.

TunnelCTL

The TunnelCTL object directs data traffic across the PCI bus. The object consists of an input, ToCPU, and three outputs: (1) ToPulse, (2) FromCPU, and (3) FromPulse, as shown below. This is the defined interface for talking between the host and the FPGA hardware.



Input

ToCPU

The ToCPU parameter is used to write data from the FPGA hardware to the host. This is a Dword (32 bits) that receives data from the hardware for the host. The data should be written upon a request for data from the host. The host indicates a request for data with the ToPulse.

Outputs

ToPulse

The ToPulse is the signal generated when the host sends a read command to the Viva driver. The pulse is generated for every 32 bits of data requested.

FromCPU

FromCPU outputs data received from the host. FromCPU is also a Dword. The buffer of data sent from the host can be of any size, but it is transferred across the bus in 32-bit pieces.

FromPulse

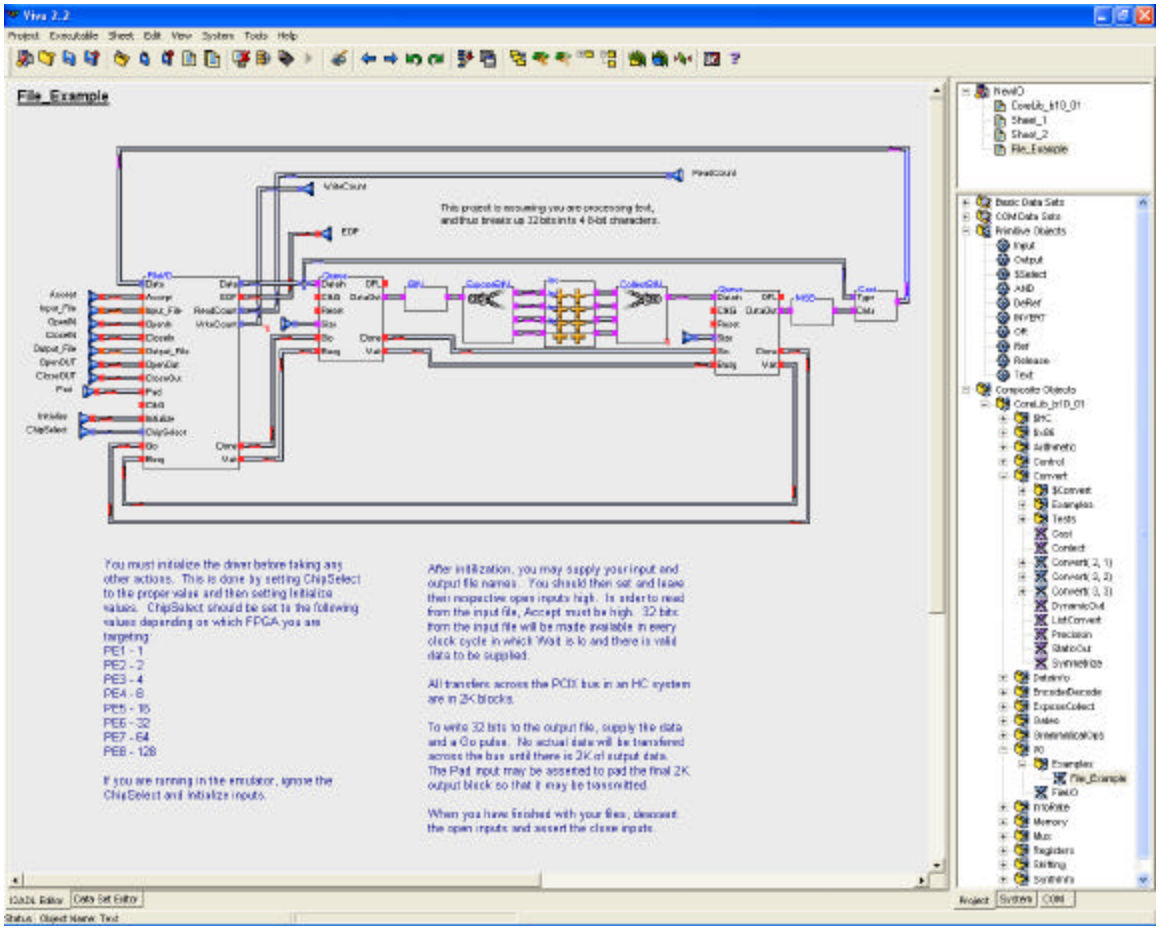
The FromPulse is generated for each 32 bits of data written from the host to the FPGA hardware. For example, if a command is sent to write 64 bytes of data to the hardware, there would be eight FromPulses generated, one for each 32 bits.

Initialization

To use the TunnelCTL, the device driver requires prior knowledge of which FPGA is to be targeted. The device driver uses a bitmap to *activate* the target FPGAs. To target one or more FPGAs, you send a WORD to the device driver with the associated bits set. For example, if you wanted to target PE5 and PE7, you would need to send a hex value of 0x0050 (binary 0000000001010000). A Viva COM object is provided to do this initialization: In CoreLib_b10_01\$HC\$Virtex2\$FileI/O\File there is an object called ChipSelect. To initialize FileIO, set the CS input to the desired target FPGA and assert SetCS.

A File I/O Application

Below is the sample application provided with Corelib. This application reads in a 2K block from the *Input_File* each time *Accept* is asserted.



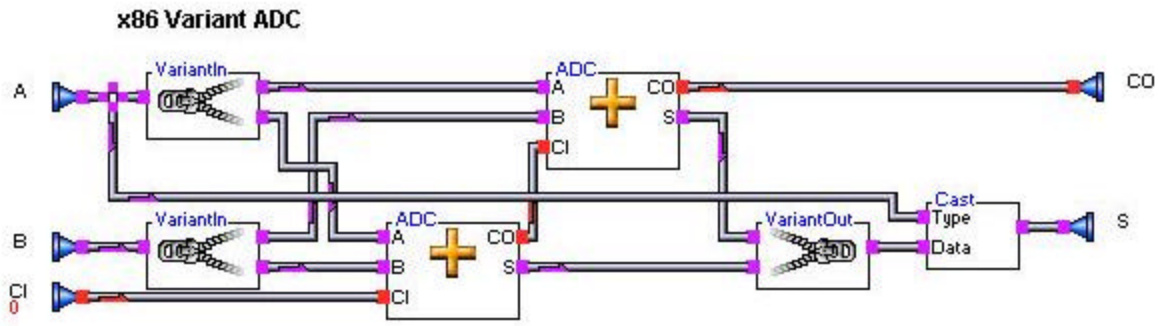
LIBRARY AND SYSTEM MANAGEMENT

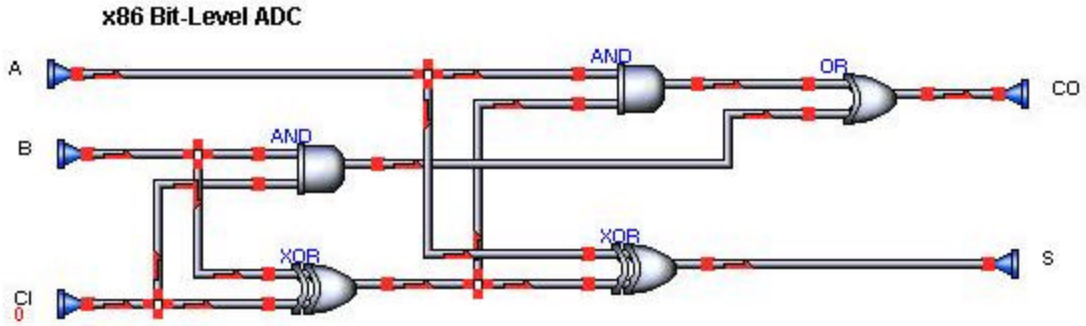
Managing library and system objects in your projects is a critical skill in Viva development because many common objects have different implementations in different systems. If you decide to change the system in your project, say from x86 to HAL, you must make sure that your project contains the correct objects.

Ambiguous Objects

We know that an object's name and number of inputs and outputs define the object's footprint. Most objects have several footprint variants, meaning versions of the object with the same number of inputs and outputs, but where at least one input or output carries a different Data Set. When two different footprint variants of an object carry the same Data Sets on all inputs and outputs, they are ambiguous. Viva does not allow ambiguous objects to exist in the same project. The reason for this is that the synthesizer would be forced to make a random decision when resolving objects through Variant Select and thus the object's behavior would never be guaranteed.

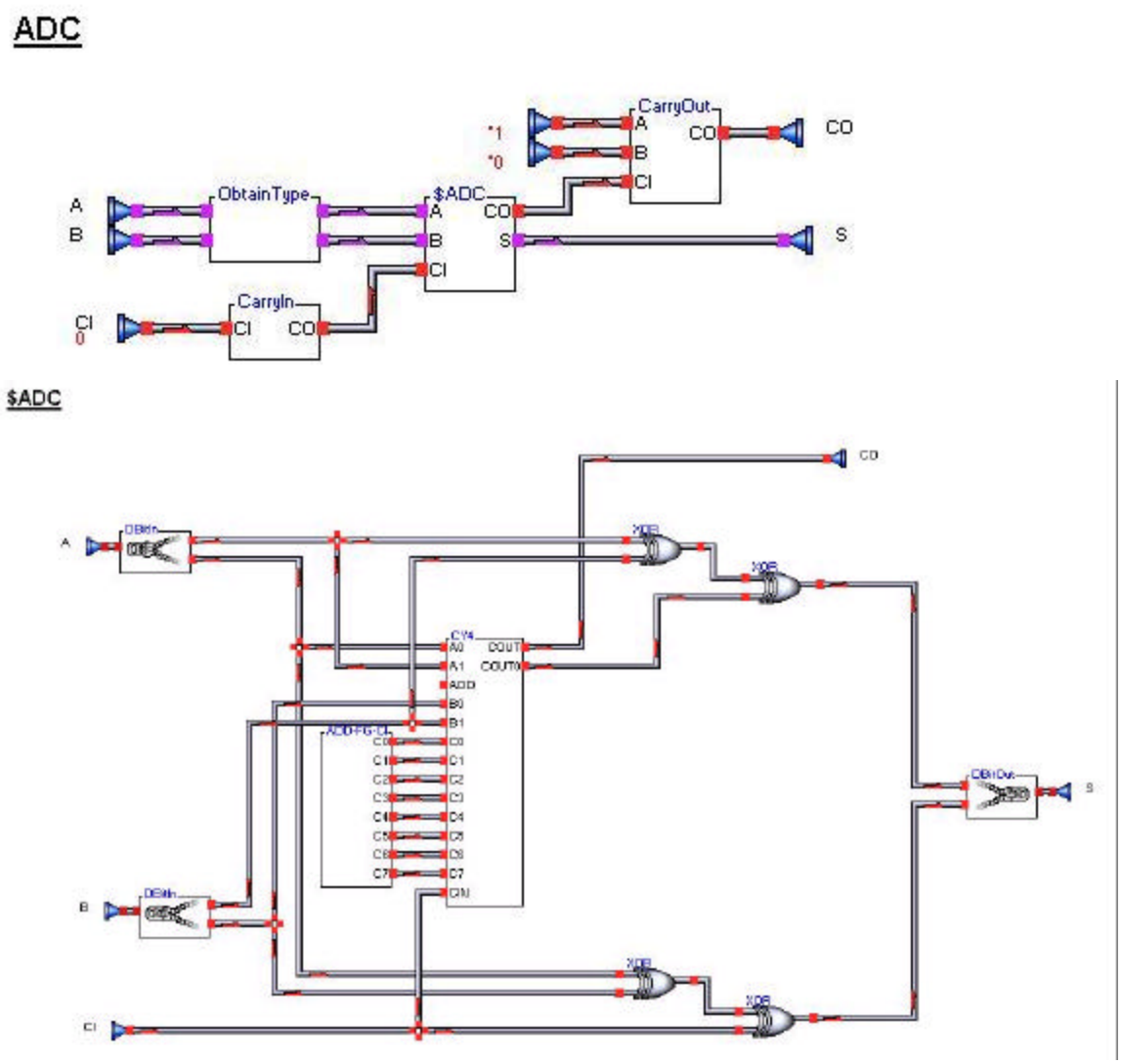
To demonstrate the necessity of ambiguous objects, let's examine the x86 and HAL implementations of Add with Carry, or full adder (ADC). Both of these objects are contained in the individual System Object Libraries. The Variant version of an x86 ADC demonstrates a classic recursive footprint. We only have one leaf node version of an x86 ADC, the Bit case. Note that it simply defines the behavior of a bit-level ADC.



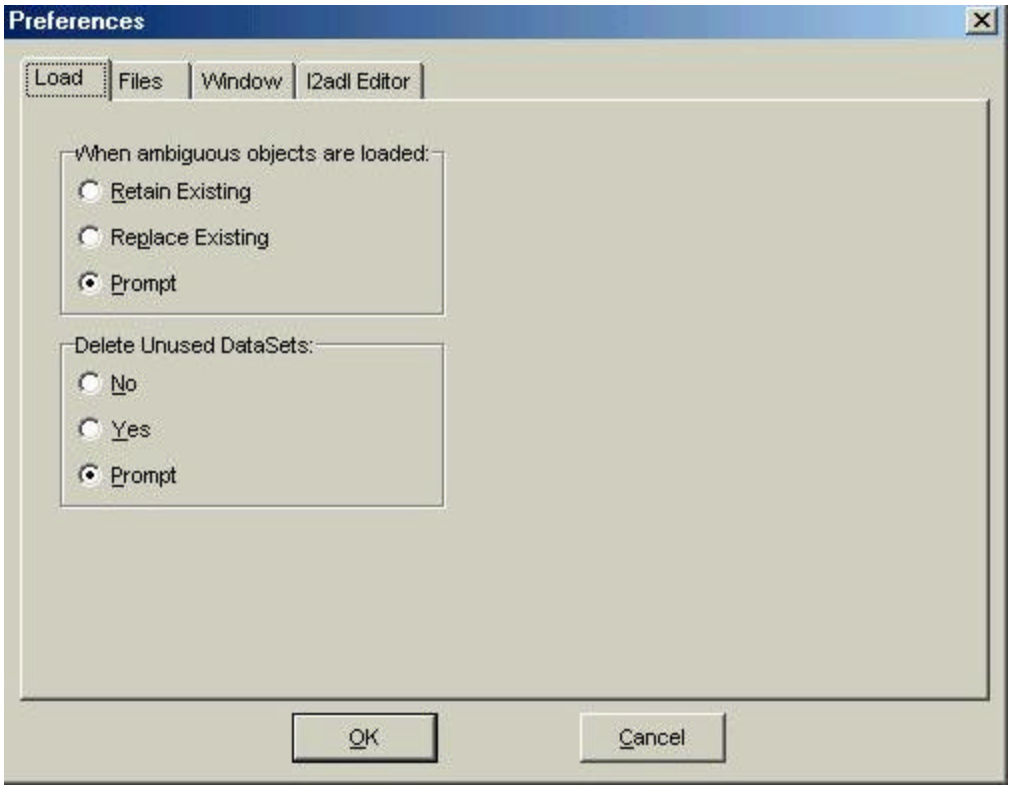


In the HAL implementation, ADC is not itself recursive, but uses a helper object (think of a recursive helper function in Scheme). The helper object, $ADC displays our classic recursive footprint. The ADC object itself contains two objects, CarryIn and CarryOut, which respectively start and end the fast-carry propagation chain. This is why the HAL-specific implementation of ADC is necessary. The CLBs in the FPGAs contain special fast-carry propagation routing resources that we must use if we expect our adders to perform. The x86 implementation will technically work, but the carry propagation will use normal routing resources, resulting in an

unacceptable number of layers of logic. The $ADC object has two leaf node versions, a DBit and a single Bit. Each of these leaf nodes represents exactly one CLB, since a CLB is capable of performing a one or two bit addition. It is necessary to have the single bit implementation to accommodate all possible Data Sets, but the DBit version represents a more optimal use of resources.



Fortunately ambiguous object management is very easy if you understand the basic concept. Since Viva does not allow ambiguous objects to reside in the same project, Viva performs an ambiguous object removal when a sheet or system description containing an object that is ambiguous with one that already exists in the project is opened. You simply need to know how to manage this object removal to effortlessly move between system implementations. Ambiguous object removal is controlled via a user preference located in the Load tab of the Preferences window.



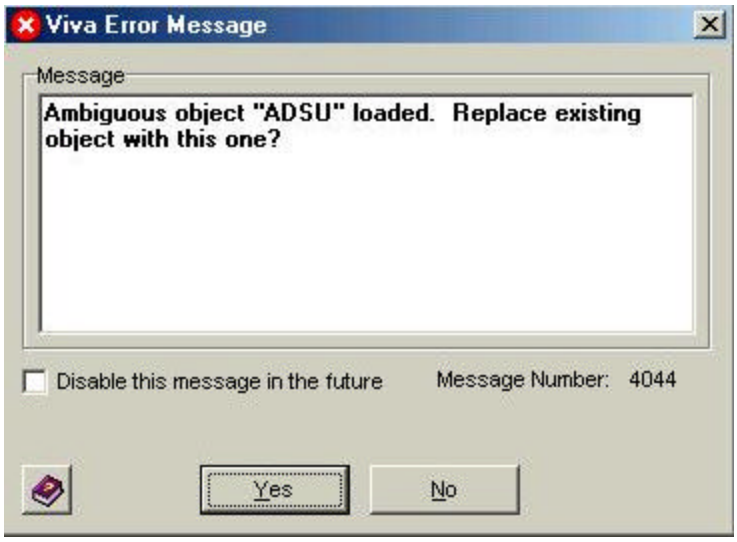
There are three choices, Retain Existing, Replace Existing and Prompt:

Retain Existing: This option will discard all ambiguous objects in the library or system file that is being opened.

Replace Existing: This option will discard all ambiguous objects already in the project in favor of those in the file that is being opened. Use this option for effortless system switching. This option should also be used when updating libraries, since new objects need to replace their ambiguous predecessors.

Prompt: This option allows you to choose whether to retain or replace each ambiguous object. This option is often used in library development.

Most users should generally use the Replace Existing option. When the Prompt preference is selected, Viva will present a dialog box with error# 4044 to allow you to choose between the objects. If you check the 'Disable this message in the future' box, Viva will apply your choice on this object to all subsequent objects. So if you choose to replace the existing object and disable the error message, Viva will behave as though you had selected the Replace Existing preference.



Overloading and Underloading

Overloading in Viva is similar in concept to overloading functions in C++. Functions of the same name in C++ are selected by a combination of their name and their parameter types. In Viva, objects with the same name and number of inputs and outputs (nodes) are overloaded by changing the input Data Sets. The object resolver will match the least variant object when deciding which object to build for any given instance.

Underloading in Viva refers to the replacement of objects when you merge two files. For example, when you load a system description, you will want to replace any existing objects of the same name and node count. This is done because many systems vary on their implementation of low-level objects such as registers and adders. Again, you will want to replace such objects when you load a system because the primitives that implement those objects in various hardware and software platforms are fundamentally different. Specifically, if you load CoreLib first, and then the system description, you likely wanted to replace all the CoreLib objects with the system equivalents. If you did not replace ADC, your adders will not work correctly in the hardware. All of the objects overloaded and underloaded by systems are typically instanced in the system descriptions in a sheet called Objects. The objective is to maintain top-level objects that all have the same interface and only replace the low-level system objects. Hence, the term underloading.

Updating Library Versions

Note: Star Bridge Systems strongly suggests that you consider the following information before updating CoreLib.

If your project is working the way you want it to, there is likely not a need to update the library. To ensure object integrity, you only want one CoreLib object tree. In the case that you do want to update, use these precautions:

- Do not store your own homemade objects in the CoreLib tree because you will need to delete the existing tree if you want to update.
- Avoid using objects that begin with $ in your project. There is no guarantee such objects will remain in the new library.
- Set your preferences to replace existing before importing the new library.
- Ensure that you replace all objects. That is, do not try to use the CoreLib9 Add with the CoreLib10 Add and so on.

- Remove the previous CoreLib tree group before or after you import. Do not mix objects from an old version with objects from a new version of CoreLib.

Should you want to update to a new version of CoreLib after reviewing the precautions, perform the following steps:

1. Delete the old CoreLib sheet.
2. Delete the old CoreLib tree group.
3. Open the new CoreLib sheet.

Sharing Designs and Libraries Among Multiple Users

Libraries, sheets, projects, and system descriptions cannot be written to simultaneously. To save a file, employ a mechanism to ensure non-simultaneous writing.

USING ON-CHIP MEMORY

On-chip Memory Objects

Objects you would use to implement on-chip memory are located in the Memory subtree of CoreLib and include the RAM, Queue, and ROM objects.

RAM

A RAM object implements the on-chip memory. In the x86 system, this object uses the Viva Standard COM objects to allocate system memory.

Inputs

A RAM object has the following inputs: Data, Address, Read, Write, Go, and Wait.

RAM Inputs	Description
Data and Address	The allocation size in bytes is the bytelength of the data times two to the bitlength of the address.
Read and Write	Read and Write are mutually exclusive flags that determine the operation you will perform. They both cannot be set; that is, they are not bi-directional.
Go	Go does not need to be a pulse. As long as a Go is held high, RAM is enabled for Reading or Writing.
Wait	If Wait is set, the signal passes through directly to Busy.

Outputs

A RAM object has the following outputs: Data Output (O) and Done.

RAM Outputs	Description
Data Output (O)	Data Output is the data stored at the address location when Read is asserted.
Done	Done is always kept high as long as Read and Go are asserted and Wait is not asserted. For example, if you assert Read and then assert Go, Done will be asserted one clock cycle later.

Queue

If the Queue contains data, it triggers the dequeue off a not Wait state. That is, for each clock cycle, a data element is pulled off the Queue in the order it was put in the Queue, as long as Wait is low. Then, a Done is fired for each valid data element pulled off the Queue.

Inputs

A Queue object has the following inputs: DataIn, Size, Reset, Go, and Wait.

Queue Inputs	Description
DataIn and Size	The byte allocation size of the Queue is determined by bit length of the DataIn times two to the bit length of the Size.
Reset	Reset clears the Queue and resets the Queue to an empty state.
Go	Data is put on the Queue on every clock cycle that Go is held high.
Wait	For each clock cycle, a data element is pulled off the Queue in the order it was put in the Queue, as long as Wait is low.

Outputs

A Queue object has the following outputs: DataOut and Done.

Queue Outputs	Description
DataOut	DataOut is the currently de-queued data element.
Done	Done is fired for each valid data element pulled off the Queue.

ROM

The ROM object initializes itself in the first sixteen clock cycles of run time.

Inputs

A ROM object has the following inputs: Data, Address, Go, Wait, and Busy.

ROM Inputs	Description
Data	Data input is a list that contains two to the N data elements, with a minimum of sixteen data elements.
Address	During initialization, the Address determines the number of data elements to store. For example, if you want to store 32 data elements, your Address would be MSB005. For look-up, the Address is the location of the stored, or initialized, data.
Go	Go is optional. If you do not overload Go, the ROM object initializes itself in the first sixteen clock cycles. Overload Go if you do not want to initialize ROM at startup, but rather at some other time, or if you want to reload the ROM.
Wait	If Wait is high, ROM will not initialize.

Outputs

A ROM object has the following outputs: Data Output (O), Done, and Busy.

RAM Outputs	Description
Data Output (O)	The output always reflects the data stored at the given Address location. The Go has no effect on the Data Output.
Done	Done is asserted as a pulse at the completion of initialization.
Busy	Busy is held high during initialization.

Examples of Using On-chip Memory Objects

There are documented examples of using on-chip memory objects. The examples are located in the Memory subtree of CoreLib and include the RAM, Queue, and ROM objects. To see the examples, open the Examples folder on the Memory subtree of CoreLib and drag an object example to the Viva workspace. Then, double-click the object to see the example.

IMPORTING EDIF INTO VIVA

The PHP script that follows facilitates importing EDIF into Viva. Most VHDL/Verilog tools will have an EDIF export option. This script provides a basic interface 3rd party code. The script will output several files including a wrapper cell, Viva sheet, and merged FPGAStrings file. All the regular expressions in the script are PCRE (Perl).

Suppose you have some VHDL code for a Montgomery multiplier. The following are the steps to interface that object into Viva using the included PHP script:

1. Install PHP (version 4.1 or later) and put the PHP directory in your path. (The CGI-only version is fine in this case.) This file and the parse.php script exist with the newer Viva installs at c:\viva\php\
2. Create a new project with the Xilinx Project Manager named mont.
3. Add the source files.
4. Synthesize the all of the objects.
5. Exit the Project Manager and locate the mont.xst file it made.
6. Open that file and change the output format to read -ofmt EDIF instead of -ofmt NGD.
7. Rerun that script from the command line with a command like the following to generate mont.edn:

   ```
   xst -ifn mont.xst -ofn log.txt
   ```

8. Assuming parse.php and FPGAStrings_V09.txt is in the same directory, execute the parser with a command like the following:

   ```
   php -q parse.php mont.edn FPGAStrings_V09.txt PE5
   ```

9. Launch Viva.
10. If you want, open CoreLib.
11. Open a system (in this case, PE5).
12. Enable the System Editor and select the System Editor tab.
13. In the system tree, drill down to the actual chip system (HC\HC_PE5\PE5).
14. Change the FPGAStrings attribute to be the generated strings file (c:\myloc\mixed_mont_fpgastrings.txt).
15. Select the I2ADL tab.
16. Open the generated sheet (wrappermont.ipg).
17. Change the clk inputs to be a global resource ClkG.

Note: There are currently some issues with handling dual drivers/IOB merging. Also, none of the SIMPRIMS, etc. are automatically inserted. You might consider compiling the libraries in the Xilinx\VHDL directory and adding those to a standard FPGA strings file. Synthesize with XST, NGDBuild, and then NGD2EDIF.

Note: If, after running the Montgomery multiplier, it fails the timing constraint in the Par tool, add `XilinxPar=-w -x -l 5` to the system description to ignore the timing constraint.

Regarding output on the Montgomery multiplier and other objects using multipliers, keep in mind that the native mult18x18 objects are signed. The script will build MSB (unsigned) input and output horns by default. You will want to convert and cast those MSB inputs and outputs to Dint or something similar to actually see the output in its proper form.

The script will remerge FPGA strings files. Keep in mind, though, that if you have two libraries of the same name, only one of them will end up in the final strings file. That should cause no problem as long as the EDIF is generated by the same tool every time.

Parse.php

```
<?php
	set_time_limit(0);
	function getmicrotime(){
		list($usec, $sec) = explode(" ",microtime());
		return ((float)$usec + (float)$sec);
	}
	function build_parse_tree(&$str){
		//just take out all the renames
		$str=preg_replace('/\(\s*rename\s+(\S+)\s+"[^"]+"\s*\)/','\\1',$str);

		//tokenize
		$str=str_replace('(','`!(`!',$str);
		$str=str_replace(')','`!)`!',$str);


		$arr=explode('`!',$str);//kaboom!
		$tree=array(array());
		foreach($arr as $val){
			$val=trim($val);
			if(!strlen($val)) continue;
			if($val=='(')
				array_unshift($tree,array());
			elseif($val==')'){
				$temp=array_shift($tree);
				array_push($tree[0],$temp);
			}
			else
				array_push($tree[0],$val);
		}
		return($tree[0][0]);//the first two arrays were work units only
	}
	function rebuildString(&$arr,$level=0){
		reset($arr);
		$output=str_repeat("\t",$level)."(\n";
		while(list($key,$val)=each($arr)){
			if(is_array($val)){
				$output.=rebuildString($val,$level+1);
			}
			else $output.=str_repeat("\t",$level).$val."\n";
		}
		$output.=str_repeat("\t",$level).")\n";
		return($output);
	}

	set_magic_quotes_runtime(0);
	if($_SERVER['argc']<4){
		echo "Usage: php -q parse.php <edif filename> <FPGA strings file> <system name>";
		exit;
	}
	$systemname=$_SERVER['argv'][3];
	$fpgastringsfile=$_SERVER['argv'][2];
	$starttime=getmicrotime();
	echo "Reading ".$_SERVER['argv'][1];
	$edf=strtolower(file_get_contents($_SERVER['argv'][1]));
```

```
	set_magic_quotes_runtime(get_magic_quotes_gpc());
	$sz=number_format(filesize($_SERVER['argv'][1])/1024,1);
	$time=number_format(getmicrotime()-$starttime,4);
	echo "\nCompleted file read of $sz KB in $time sec.\nNow parsing....\n";
	$starttime=getmicrotime();
	$pt=build_parse_tree($edf);
	reset($pt);
	ob_start();
	var_dump($pt);
	$ret_val = ob_get_contents();
	ob_end_clean();
	$handle=fopen("lastdump.txt",'wb');
	if(!$handle){
		echo "Unable to open lastdump.txt for writing! Exiting....\n";
		exit;
	}
	fwrite($handle,$ret_val);
	fclose($handle);

	//find the design entry:
	for($i=count($pt)-1;$i>=0;$i--){
		if(substr($pt[$i][0],0,6)=='design'){//we don't care about the design name
			$des=trim(substr($pt[$i][1][0],8));
			$lib=trim(substr($pt[$i][1][1][0],10));
			$time=number_format(getmicrotime()-$starttime,4);
			echo "Parsed and dumped to lastdump.txt in $time sec.\nThe top level cell
was '$des' in library '$lib'\nExtracting the interface for it....\n";
			break;
		}
	}
	$lasti=0;
	//if we were smart we wouldn't do a linear search....
	$i=0;
	while($i<count($pt)){
		//find the library
		if(preg_match("/library\s+$lib/",$pt[$i][0])){
			$midlib=$pt[$i];
			reset($midlib);
			//find the cell
			$lastkey=-1;
			next($midlib);//first one is library
			while(list($key,$val)=each($midlib)){
				if(substr($val[0],0,4)=='cell'){
					if(trim(substr($val[0],4))==$des){
						$lastkey=$key;
						break;
					}
				}
			}
			if($lastkey>=0){
				$lasti=$i;
				break;
			}
		}
		$i++;
	}
```

```
if($lastkey>=0){
	echo "Congrats! We found the top level cell at index $lasti,$lastkey\n";
}
else{
	echo "Aborting.... We couldn't find the top level cell.\n";
	exit;
}
echo "Outputing library file....\n";
$i=count($pt);
$handle=fopen("library_$des.edf",'wb');
if(!$handle){
	echo "Unable to open library_$des.edf for writing! Exiting....\n";
	exit;
}
while($i>0){
	$i--;
	//find the library
	if(preg_match('/(library|external)\s+/',$pt[$i][0]))
		fwrite($handle,rebuildString($pt[$i]));
}
fclose($handle);
echo "Finished writing 'library_$des.edf'\n\nNow parsing the top level object interface....\n";

$topcell=$pt[$lasti][$lastkey];
reset($topcell);
while(list($key,$val)=each($topcell)){
	if(substr($val[0],0,4)=='view'){
		$viewkey=$key;
		$view=trim(substr($val[0],5));
		break;
	}
}
$topview=$topcell[$key];
reset($topview);
while(list($key,$val)=each($topview)){
	if(substr($val[0],0,9)=='interface'){
		$interfacekey=$key;
		break;
	}
}
$topinterface=$topview[$interfacekey];
$interface=array();
$bitsofoutput=0;
$bitsofinput=0;
$maxbitsofinput=0;
$maxbitsofoutput=0;
reset($topinterface);
while(list($key,$val)=each($topinterface)){
	if(is_array($val)){
		$index='';
		$size='';
		if(substr($val[0],0,4)=='port'){
			if(is_array($val[1])){
				//possibly an array of bits:
				if(substr($val[1][0],0,5)=='array'){
```

```
		$index=trim(preg_replace('/array\s+(\S+)\s+\S+/','\\1',$val[1][0]));

		$size=trim(preg_replace('/array\s+\S+\s+(\S+)/','\\1',$val[1][0]));
                                        }
                                        else{
                                                $index=trim(substr($val[0],5));
                                                $size=1;
                                        }
                                }
                                else{//standard single-bit port
                                        $index=trim(substr($val[0],5));
                                        $size=1;
                                }
                        }
                        else continue;
                        $interface[$index]['size']=$size;
                        $interface[$index]['output']=strpos($val[count($val)-
1][0],'input',min(9,strlen($val[count($val)-1][0])))===false;
                        if($interface[$index]['output']){
                                $bitsofoutput+=$size;
                                $maxbitsofoutput=max($maxbitsofoutput,$size);
                        }
                        else{
                                $bitsofinput+=$size;
                                $maxbitsofinput=max($maxbitsofinput,$size);
                        }

                }
        }

        echo "Finished parsing interface with ".count($interface)." port entries.\nWriting the wrapper
file....\n";
        //print_r($interface);
        $handle=fopen("wrapper$des.edf",'wb');
        if(!$handle){
                echo "Unable to open wrapper$des.edf for writing! Exiting....\n";
                exit;
        }
        $cellstr="(cell wrapper$des (cellType GENERIC)\n\t(view net (viewType
NETLIST)\n\t\t(interface\n";
        //output all the wrapper ins and outs
        reset($interface);
        while(list($key,$val)=each($interface)){
                if($val['output']) $dir='output';
                else $dir='input';
                for($i=0;$i<$val['size'];$i++){
                        $cellstr.="\t\t\t(port $key$i (direction $dir))\n";
                }
        }
        //output the contents with instances of CellRef, LibRef
        $cellstr.="\t\t)\n\t\t(contents (instance inst$des (viewRef $view (cellRef $des (libraryRef
$lib))))";

        //output the joins that tie everyting together
        reset($interface);
        while(list($key,$val)=each($interface)){
```

```
		if($val['size']==1){
			$i=0;
			$cellstr.="\t\t\t(net net$key$i (joined (portRef $key$i) (portRef $key
(instanceRef inst$des))))\n";
		}
		else{
			for($i=0;$i<$val['size'];$i++){
				$cellstr.="\t\t\t(net net$key$i (joined (portRef $key$i) (portRef
(member $key $i) (instanceRef inst$des))))\n";
			}
		}
	}

	$cellstr.="\t\t)\n\t)\n)";
	fwrite($handle,$cellstr);
	fclose($handle);
	echo "Finished writing wrapper$des.edf.\n\nWriting Viva importable sheet....\n";
	$handle=fopen("wrapper$des.ipg",'wb');
	if(!$handle){
		echo "Unable to open wrapper$des.ipg for writing! Exiting....\n";
		exit;
	}

	//some initial layout locations:
	//remember units are 5px
	$left=15;
	$realleft=10;
	$top=10;//under the title?
	$varwidth=15;
	$objectwidth=strlen("wrapper$des")<<1;//10px per char? What about long node names?
	echo "Object has $bitsofinput inputs (max chunk is $maxbitsofinput) and $bitsofoutput outputs
(max chunk is $maxbitsofoutput)\n";
	$right=$objectwidth+($varwidth*($maxbitsofinput+$maxbitsofoutput))+$left;
	$middle=$varwidth*$maxbitsofinput+$left;
	$putstr='';
	$midstr='';
	$varstr='';
	$plugstr='';
	$tempalpha='';
	$once=false;
	$alpha='A';
	$subalpha='A';
	$zero='0';
	$one='1';
	$incer=$top;
	reset($interface);
	while(list($key,$val)=each($interface)){//outputs first
		if(!$val['output']) continue;
		$thisalpha=':'.$alpha;
		if(!$once){
			$comma='';
			$once=true;
			$thisalpha='';
		}
		else{
			$comma=',';
```

```
                        $alpha++;
                }
                if($val['size']==1){
                        $keyword='Bit';
                        $midstr.="$comma Bit $key$zero";
                        $plugstr.="Output$thisalpha=wrapper$des.$key$zero;\n";
                }
                else{
                        $keyword='MSB'.sprintf("%03.0f",$val['size']);
                        for($i=0;$i<$val['size'];$i++){
                                $oldalpha=$tempalpha;
                                $offsety=$incer+($i*3);
                                $offsetx=($i*$varwidth)+$objectwidth+$middle;
                                if($i==0){
                                        if($tempalpha)$tempalpha=':'.$subalpha;

        $plugstr.="MSB002Out$tempalpha.In2=wrapper$des.$key$one;\n";
                                }
                                elseif($i==1){
                                        if($tempalpha)$tempalpha=':'.$subalpha++;

        $plugstr.="MSB002Out$tempalpha.In1=wrapper$des.$key$zero;\n";
                                        $varstr.="\tObject ( MSB002 Out) MSB002Out$tempalpha(
Bit In1, Bit In2) ; //_GUI $offsetx,$offsety\n";
                                }
                                else{
                                        $tempalpha=':'.$subalpha++;

        $plugstr.="VariantOut$tempalpha.In2=wrapper$des.$key$i;\n";

        if($i==2)$plugstr.="VariantOut$tempalpha.In1=MSB002Out$oldalpha;\n";
                                        else
$plugstr.="VariantOut$tempalpha.In1=VariantOut$oldalpha;\n";
                                        $varstr.="\tObject ( Variant Out) VariantOut$tempalpha(
Variant In1, Variant In2) ; //_GUI $offsetx,$offsety\n";
                                }

                                $midstr.="$comma Bit $key$i";
                                $comma=',';
                        }
                        $plugstr.="Output$thisalpha=VariantOut$tempalpha;\n";
                }
                $incer+=$val['size']*3-3;
                $putstr.="Object Output$thisalpha ($keyword $key); //_GUI $right,$incer\n";
        }

        $midstr.=") wrapper$des (";
        $once=false;
        $tempalpha='';
        $alpha='A';
        $subalpha='A';
        $incer=$top;
        reset($interface);
        while(list($key,$val)=each($interface)){
                if($val['output']) continue;
                $thisalpha=':'.$alpha;
```

```
                if(!$once){
                        $comma='';
                        $once=true;
                        $thisalpha='';
                }
                else{
                        $comma=',';
                        $alpha++;
                }
                if($val['size']==1){
                        $keyword='Bit';
                        $midstr.="$comma Bit $key$zero";
                        $plugstr.="wrapper$des.$key$zero=Input$thisalpha;\n";
                }
                else{
                        $keyword='MSB'.sprintf("%03.0f",$val['size']);
                        for($i=0;$i<$val['size'];$i++){
                                $oldalpha=$tempalpha;
                                $offsety=$incer+($i*3);
                                $offsetx=$middle-($i*$varwidth);
                                if($i==0){
                                        if($tempalpha)$tempalpha=':'.$subalpha;

        $plugstr.="wrapper$des.$key$one=VariantIn$tempalpha.Out2;\n";
                                }
                                elseif($i==1){
                                        if($tempalpha)$tempalpha=':'.$subalpha++;

        $plugstr.="wrapper$des.$key$zero=VariantIn$tempalpha.Out1;\n";
                                        $varstr.="\tObject ( Variant Out1, Variant Out2)
VariantIn$tempalpha( Variant In) ; //_GUI $offsetx,$offsety\n";
                                }
                                else{
                                        $tempalpha=':'.$subalpha++;

        $plugstr.="wrapper$des.$key$i=VariantIn$tempalpha.Out2;\n";
                                        $plugstr.="VariantIn$oldalpha=VariantIn$tempalpha.Out1;\n";
                                        $varstr.="\tObject ( Variant Out1, Variant Out2)
VariantIn$tempalpha( Variant In) ; //_GUI $offsetx,$offsety\n";
                                }

                                $midstr.="$comma Bit $key$i";
                                $comma=',';
                        }
                        $plugstr.="VariantIn$tempalpha=Input$thisalpha;\n";
                }
                $incer+=$val['size']*3-3;
                $putstr.="Object ($keyword $key) Input$thisalpha ; //_GUI $left,$incer\n";
        }
        fwrite($handle,"// VIVA  Implementation Independent Recursive Description Language\n");
        fwrite($handle,"// Sheet   ".date("m/d/Y h:i:s A T"));
//      fwrite($handle,"\nDataSet List = ( Variant , Variant ); //_Attributes 1,16711808\n");
        fwrite($handle,"\nObject $des\n{\n\tObject (");
        fwrite($handle,"$midstr); //_GUI $middle,$top\n");
        fwrite($handle,"\t//_Attributes
Primitive,System=$systemname,Resource=CLB,LibRef=wrapper$des\n$putstr\n$varstr\n$plugstr}\n");
```

```
		fwrite($handle,"Object ($midstr);\n");
		fwrite($handle,"//_Attributes
Primitive,System=$systemname,Resource=CLB,LibRef=wrapper$des\n");
		fclose($handle);
		echo "Finished writing wrapper$des.ipg.\n\nNow scanning the system description file
$fpgastringsfile....\n";
		$fpgastrings=strtolower(file_get_contents($fpgastringsfile));
		preg_match_all('/\(\s*(library|external)\s+(\S+)/',$fpgastrings,$lls1);
		preg_match_all('/\(\s*(library|external)\s+\(\s*rename\s+(\S+)/',$fpgastrings,$lls2);
		$lls=array_merge($lls1[2],$lls2[2]);
		$llstop=array_merge($lls1[0],$lls2[0]);
		$handle=fopen("mixed_{$des}_fpgastrings.txt",'wb');
		if(!$handle){
			echo "Unable to open mixed_{$des}_fpgastrings.txt for writing! Exiting....\n";
			exit;
		}
		$mixerlib='';
		$i=0;
		while($i<count($pt)){
			//find the libraries that aren't already in there
			$arrs=array();
			if(preg_match('/(library|external)\s+(\S+)/',$pt[$i][0],$arrs)){
				if(!array_search($arrs[2],$lls))
					$mixerlib.=rebuildString($pt[$i]);
			}
			$i++;
		}
		$fpgastrings=preg_replace('/\(\s*cell\s+\[fpgafilename\]/',$cellstr."\n\n\n(cell
[FPGAFileName]",$fpgastrings);
		$fpgastrings=str_replace('[fpgafilename]','[FPGAFileName]',$fpgastrings,1);
		fwrite($handle,str_replace($llstop[0],$mixerlib.$llstop[0],$fpgastrings));
		fclose($handle);
		echo "Finished writing 'mixed_{$des}_fpgastrings.txt'\nScript complete.\n";



?>
```


VIVA SYSTEM DESCRIPTIONS

A Viva System specifies the runtime environment for Viva. A Viva System consists of one or more of the following five parts: (1) Executor, (2) Behavior, (3) Resource, (4) Translator, and (5) Initializer.

- **Executor: Physical or logical engine that runs the behavior of the system.**

  For example:

  X86 = Event thread
  FPGA = FPGA
  PIN Communication System = Actual wires/pins
  Viva defined executor (Implementor [sic]) = A Viva behavior that becomes part of the execution environment

  The following Executors are defined by Viva under 'System Types' in the Viva System Editor:
  - Default: Used to define the top level system which, by itself, does not execute anything.
  - X86Main: Backend emulator engine. It includes emulated registers and gates (i.e. X86CPU).
  - X86UI: Converts IO horns to widgets.
  - FPGA: Causes the behavior to run on the FPGA hardware (i.e. PE1, PE2, . . . PEn).
  - Pin Communication (obsolete): Actual wires or pins
  - I2adl Implementation: Causes the behavior to always get built (same as naming the object Implementor (i.e. Implementor -)
  - Fixed Bus: Communication between two systems (i.e. UIXCPU – found in x86.sd)

- **Behavior: List of operations that a system is capable of performing. Consumes resources.**

  Example behaviors include the following:

  AND
  OR
  Transport
  Mult

- **Resource: Defines a consumable element of a system.**

Note: Refer to the section titled Resource Usage for information on how to track used resources.

  Elements of a resource are:

  - Name: Unique identifier. Some common names are: Gate, Clb, Pins, Pads and Timeslice.
  - Cost: Calculated from remaining quantity and intrinsic cost. The cost increases exponentially as the remaining quantity decreases.
  - Intrinsic cost: The cost of using one resource.
  - Information rate: The throughput of the resource.

- Quantity: The number of resources available to the system
- Type: One of the following:
- POOL: Parallel resource. Sample POOL resources are: CLB, IOBUF, TBUF, DPADS.
- PINS: Resource with a specific location. Sample PINS resources are: WIRENODE, APINS, IOPINS.
- TDM: Time division multiplex resource – sequential. A TDM resource is a TIMESLICE.
- PADS: Resource with specific locations (obsolete)

- **Translator: Converts from native atomic i2adl to system code. The Translator creates the run image. The default Translator generates an EDIF netlist that can be overridden with a system attribute "Translator" naming an external program that will perform the translation.**

  Note: See step 5 (of **Steps to building a system**) for values that must be set.

- **Initializer: Programs the chips. The default Initializer can be overridden with a system attribute "Initializer".**

  The following values must be set to "true". They are placeholders for future Viva features:

  "TranslatorExclusive"
  "InitializerExclusive"
  "PostInitializerExclusive"

Steps to Building a System

To build a system, perform the following steps:

1. From Viva, select the "System" drop-down menu and select "Enable Editor."

2. Select the "System Editor" tab at the bottom of the window.

3. From the System Editor, define a new system:
   a. Click "Create New System" (found at the top left of the system editor window).
   b. Select type: If using a predefined type, existing system elements like translators and initializers can be reused. This specifies the executor.

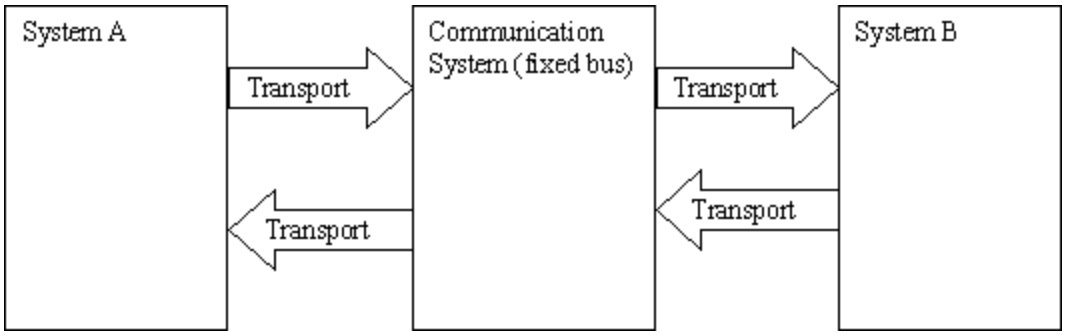
4. Define resources
   a. Enter a system name in the first column of the Resource Editor.
   b. For each resource, provide the resource name, type, quantity, intrinsic cost, information rate and pins.

5. Build behavior atoms by doing the following:
   a. Define an interface by making an object with inputs and/or outputs defined but no internal behavior, or by copying the footprint of a single existing object. Copying is done by wrapping up a sheet that contains a single object.
   b. Associate resources with behaviors by adding the Libref, Resource, System, and, optionally EdifFile and ResourceQty. If no EdifFile is specified, Viva will assume the edif cell already exists in the FPGAStrings file. ResourceQty defaults to 1.
   c. Wrapping the sheet into an object. This should give a warning message that there is only a single object or no defined behavior. Select 'Yes' to proceed.

6. Associate Behavior that is always built with a system. An example of this would be the clock. Most systems will not work without a clock, so most systems have an implementor with a clock output. There are two ways to define an implementor:
    a. Have a subsystem with system type of I2adlImplementation.
    b. Have an object named Implementor.

Building Communication Systems

When synthesizing, Viva partitions everything into systems. Communication systems are fixed bus systems that go between two other systems.



All communication systems contain objects named 'Transport' that define the system connectivity. The following information refers to the Transport object and not the object interconnections.

The communication system requires an Implementor with an input and output horn with the 'ProvidesResource' attribute. This specifies the number of Transports to be instantiated.

The actual pins consumed are defined by the resource definition. Transports consume resources to perform transport behavior, so the number of resources in the resource definition defines the maximum number of transports.

System Attribute Strings

Base System Attributes

- "DefaultInputWidget" overrides ScrollBar as the default widget for Input objects that do not have a "Widget" attribute.
- "DefaultOutputWidget" overrides ScrollBar as the default widget for Output objects that do not have a "Widget" attribute.
- "WidgetResource" tells Viva which resource should be used on Input/Outputs that don't have their own "Resource" attribute. This attribute is required on the base system.
- "WidgetSystem" tells Viva which system should be assigned to Inputs/Outputs that don't have their own "System" attribute. This attribute is required on the base system.
- "DefaultTargetSystem" determines the default system to be used on objects not explicitly assigned a system. It is only a default system because objects that are not implemented by that system or will not fit in that system are assigned to another system.
- "EmulatorSpeed" allows the user to control the speed of the emulator at the expense/benefit of the Windows response rate. The default value is 50 emulator iterations per Windows refresh.
- "ResourceData" overrides the default ResourceData.txt file in the Viva\VivaSystem directory. The Resource Editor uses this file name when reading/saving files. The file

name may be fully qualified (starts with a drive letter or "\") or relative to one of the following directories:

" "	C:\Viva\VivaSystem
"$V\"	C:\Viva
"$S\"	C:\Viva\VivaSystem
"$P\"	Current project's directory

- "WidgetLocation" is not used and will be removed.

FPGA Subsystem Attributes

- "FPGAStrings" allows each FPGA subsystem to have its own netlist header file. This provides individual descriptions for the different FPGAs used in HAL and HC-xx systems.
- "PartType" is used to inform the Xilinx tools the type of chip being programmed. This attribute is required on and is only valid on FPGA chip subsystems.
- "APort" indicates the port number used to program up projects on the array of chips (PE1 – PE9) on the old style FPGA board. This attribute is only valid on root FPGA board subsystems (normally FAI or HC) that contain the ProgramInfo attribute. The absence of this attribute indicates the new style FPGA board.
- "PPort" indicates the port number used to program up projects on the program chip (XPoint) on the old style FPGA board. This attribute is only valid on root FPGA board subsystems (normally FAI or HC) that contain the ProgramInfo attribute. The absence of this attribute indicates the new style FPGA board.
- "ProgramInfo" is used to identify root FPGA board systems (normally FAI or HC). The value must be "Board1".
- "ChipIndex" is used to cross reference the chip numbers on the FPGA hardware board to the FPGA subsystem names. On the old style FPGA boards, chip number 0 is the program chip (XPoint) and chip numbers 1 – 9 are the array of chips (PE1 – PE9). On the new style FPGA boards, chip numbers 0 – 8 are the array of chips (PE1 – PE9).
- "MinNumberOfObjects" reduces compile time by allowing the system designer to only run the Xilinx tools on the FPGA chips that are actually being used.

Fixed Bus Subsystem Attributes

- "IOAddress" indicates the port number used to read and write data on the program chip (XPoint) on the old style FPGA board. This attribute is required on and only valid on the Fixed Bus subsystem (ATIO).

Base & FPGA Subsystem Attributes

- "FPGAStrings" allows each FPGA subsystem to have its own net list header file. The file name may be fully qualified (starts with a drive letter or "\") or relative to one of the following directories:

" "	C:\Viva\VivaSystem
"$V\"	C:\Viva
"$S\"	C:\Viva\VivaSystem
"$P\"	Current project's directory

- "Xilinx<program>" is used to override the default command line parameters on the Xilinx batch file programs. The default values are:

  XilinxNgdBuild = -u

XilinxMap = -pr b
XilinxPar = -w
XilinxBitGen = -w

Implementation System Attributes

- "Executor" names the external program that will perform the project execution for the subsystem.
- "Initializer" names the external program that will perform the initialization for the subsystem.
- "PostInitializer" names the external program that will perform the post initialization functions for the subsystem.
- "Translator" names the external program that will perform the translation for the subsystem.
- The following attributes must be set to "true". They are placeholders for future Viva features:

  "TranslatorExclusive"
  "InitializerExclusive"
  "PostInitializerExclusive"

Identifying a System Description's Communication System

To identify an existing system description's communication system, perform the following steps:

1. Launch Viva.
2. Load the system description.
3. From the System menu, choose Enable Editor.
4. Select the System Editor tab located at the bottom of the Viva workspace,
5. In the pane on the right, expand the system hierarchy tree.
6. Select each item in the tree and in the System Type box, note which items are labeled Fixed Bus.

   Note: Each subsystem that is labeled Fixed Bus is a communication system that plays a role in the overall system description's behavioral communication.

7. Select the I2ADL Editor tab.
8. Select the System tab located at the base of the pane on the right.
9. Expand each system identified as a Fixed Bus system type and note that each system contains at least two Transport objects, and that some systems contain Implementor objects.
10. Open a Transport object.

- Any Transport object you open has an input and an output horn. One horn belongs to the system to which the Transport object is subordinate, and the other horn belongs to an external system.

11. Right-click a node on the Transport object to open the Edit Attributes dialog box.

- Use the Edit Attributes dialog box to determine which resource both nodes on the Transport object are using. Also, identify what the external system is.

- The resource used by nodes on the Transport object comes from one of two places:

    1. A resource name that belongs to the resource prototype assigned to that system.

    or

    2. An object (in the Implementor object) with the ProvidesResource attribute.

In the case that the resource comes from **(1)** *a resource name that belongs to the resource prototype assigned to the particular system*, perform the following:

1. Select the Resource Editor tab that is located at the bottom of the workspace.
2. Locate the resource in the Resource Editor.

- If the resource is pins, then that communication system is wiring.
- If the resource is Time Division Multiplexing (TDM), then the communication system is turn-based.

In the case that the resource comes from (**2**) *an object (in the Implementor object) with the ProvidesResource attribute*, perform the following:

1. Drag the Implementor object specific to the system onto the Viva workspace.
2. Right-click each horn on the Implementor object to locate the horn with the ProvidesResource attribute.

- In the HC system, the PCI out object holds special handling for the chip system and the X86 system between which you are communicating. Driver calls occur on the X86 side.

Sample System Description

Viva contains an X86 system that is always loaded by default. In this section, we will discuss this system as a sample system description.

The X86 system consists of the following four subsystems:

X86CPU: This system executes on the host (System Type = X86Main).
X86UI: This system is the Widget form providing the user interface.
UIXCPU: This is a communication system for communicating between the above two systems.
X86ClkImport: The clock.

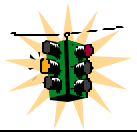
Viva User Guide

PURPOSE

This document is an introduction to the world of hyper computing technology through the use of VIVA, a complete software development package for field programmable gate array (FPGA) hardware. Star Bridge Systems, Inc. (SBS) has developed VIVA to enable you with the ability of implementing your solution in a fraction of the time required by any other method! By using VIVA, multiple FPGAs can be configured and reconfigured to provide a wide variety of functions in a parallel computing environment. VIVA is written to take advantage of only the portion of an FPGA necessary to accomplish a given task.

If you are new to VIVA, there are a few terms used that may not be familiar to you. Every attempt has been made to include a definition for each term throughout the document. In addition, a *Glossary of Terms* is provided as an appendix to this document.

Throughout this document you will see the following icons associated with specific content as described below.

Icon	*Description*
	This icon identifies the definition of a term used in this document. A summary of defined terms is found in the Appendix
	Note. This icon identifies informational messages that could be helpful in understanding VIVA
	A cautionary message indicating potential issues for the VIVA user

INTRODUCTION TO VIVA

VIVA has been developed as a windows-based development environment that enables you to harness the power of hyper computing. Within the VIVA environment there are a number of features that you will need to become familiar with in order to be proficient at using this powerful tool. Some of the features and functionality will be very familiar to you as you start using it.

We introduce you to a few of the VIVA techniques in this section. However, most of the features and techniques are discussed in detail in the remaining sections of this document.

Dragging & Dropping Objects

To drag and drop an object into the application workspace you simply point to the object, left-click and hold on the object in the Object Tree, then drag the object over the workspace and release the mouse button.

An interesting feature of the VIVA application programming environment is that if an object is dropped onto the sheet in such a way that one or more of its input or output nodes fall within node snap tolerance range of another node, VIVA will connect the nodes automatically if Data Set

types are compatible and if the node functions are appropriate (Input-to-Output or Output-to-Input).

If nodes are not connected automatically, then you will be responsible for connecting them through the use of transports. Transports are links between input and output nodes.

Drawing Transports
Begin a transport by left-clicking on an input or output node. The transport attaches from the node to the mouse pointer. Moving the mouse around the node will cycle the transport through 90-degree turns. Left-click in the VIVA workspace to designate an anchor point for the transport and continue dragging and extending the transport. Left-clicking on an output or input node will connect the free end of the transport to the node. If connecting nodes have compatible data types, a transport can also be terminated at the last fixed location by right-clicking in the application workspace.

Transport Junctions
Double-clicking a transport will create a junction at the clicked location of the transport. A junction exposes four nodes. One node is an input node and the remaining three are output nodes.

Editing Transports
Left-click an existing transport to make it the active object, it will turn blue when activated. To delete the transport, press the delete key on your keyboard or select Delete from the Edit menu. To Cut the transport, press ctrl-X on your keyboard, or select Cut from the Edit menu. To Copy the transport, press ctrl-C on your keyboard, or select Copy from the Edit Menu. To paste a copied transport onto the worksheet, press ctrl-V on your keyboard or select Paste from the Edit menu.

To undo cut, paste and delete operations, select Undo from the Edit menu, click the Undo button on the toolbar, or press ctrl-Z on your keyboard.

Transport Rules
- An output node can only be connected to an input node.
- An input node can only be connected to an output node.
- Only like data types may be used.

The remainder of this document will provide you with more in depth discussion of the VIVA environment including concepts and techniques.

ADDITIONS TO VIVA 2.2

EDIF File Support

- The new object attribute "EdifFile" supports EDIF file imports into EDIF NetFiles built by VIVA.
- Xilinx cell properties can be included in an EDIF netlist by using the "FPGA" object attributes.

CHANGES AND ADDITIONS IN CORELIB_10

- All library objects are now based on the VIVA grammatical data sets Variant, List, Num and Contextual, as described on the opening sheet of CoreLib_10. The Contextual data sets are user-extensible. Examples of these contextual data sets include: Signed, Fixed, Floating, Complex and Pipelined.
- In X86 Emulation Mode, memory objects (RAM, ROM, Queue, FIFO, Stacks) can now be used.
- File I/O has been added, allowing VIVA to access and manipulate files from hardware or from the emulator.
- Arithmetic operators now support Complex numbers.
- Bit Length Arithmetic operators have been added to the Grammatical Operators.
- Trigonometric Functions have been added.
- Operators automatically perform implicit conversions to make them symmetrical, which means that if one variable input is not equal in length to the other, the shorter one is automatically expanded to match the larger, avoiding resolution problems.
- Encode and Decode objects can now be Variant.
- Full Set of Virtex Block RAM overloads automatically allocates the optimal amount of FPGA block RAM for the application, eliminating the need to specify individual block RAM allocation.
- New Time Division Multiplexing (TDM) features allow a programmer to specifically tune an algorithm for speed of execution or conservation of resources. Time-division-multiplexing allows you to share an operator between multiple data flow paths, using less space, but taking more time.
- New Parallel→Serial and Serial→Parallel objects have been created which convert parallel data sets into serial, or vice versa. These objects are essentially shift registers with a parametizable output interface and a GoDoneBusyWait control interface. A Parallel → Serial object makes a large data set pass through a smaller interface in time. Serial → Parallel does the opposite.
- TDM (time-division-multiplexed) bus communication passes large data sets over a small number of bus lines in time by using Parallel→Serial on the sending side and Serial→Parallel on the receiving side.
- New List Operators have been added that allow manipulation of lists: ListBroadcast, N_Bit_LSB_List, N_Bit_MSB_List, ReverseList
- VIVA builds the fastest carry-propagation circuitry possible in a Virtex chip, increasing the performance time of Adders, which are notoriously slow hardware objects due to the carry signal which must propagate through many layers of logic.

BUGS AND KNOWN ISSUES

- Creating, closing or opening a project or opening a system while a program is compiling may cause VIVA to crash.
  Workaround - Stop the compiler before opening or creating Project or System files.

- Com Object Tree (MainForm) Issues a GPF when user attempts to delete a top-level node. Com Class Tree crashes VIVA if you drag a top-level node onto I2adlView.
  Workaround - Manipulation of top level nodes should be avoided.

- The Object Browser accepts ".oca" files from VisualBasic, but should only accept ".ocx" files. Selecting ".oca" files mistakenly included in the Browser results in no effect.
  Workaround - The user should be careful to select only the ".ocx" library reference, not the ".oca" reference files.

- Opening a project already in use crashes VIVA.
  Workaround - Do not attempt to open a VIVA file that is currently in use by VIVA or other applications.

- Clicking on the Icon Editor button attempts to open MSPaint.exe, which is not at the hard-coded location in WinNT versions.
  Workaround - Copy MSPaint.exe to "C:\Program Files\Accessories\".

- When bit datasets are propagated to polymorphic objects having inputs directly feeding exposers, the primitive versions of those objects are not selected.
  Workaround - Select primitive objects on a top-level page when propagating bits to its inputs.

- Giving a control node name to more than one IO object on a sheet causes it to toss all but the first IO object when that sheet is converted to an object. This can manifest itself in a number of ways, including failure to route parts of an otherwise valid project.
  Workaround - Always ensure that control node names (Go, Done, Busy, Wait) are unique on all IO objects in a sheet.

- Changing your computer name causes VIVA to fail to access the Viva.ini file.
  Workaround - If you change your computer name, create a directory with the same name under the "\documents and settings\" directory.

- When an object with no non-control nodes and no tree groups is saved to file, it is interpreted as a WIP sheet on a subsequent read-in.
  Workaround - Ensure that all objects have either a tree group or non-control nodes before saving.

- When the system name is changed using the "Save System As" option, the new system name is put into the system object tree but not into the system tree.
  Workaround: To reset the system object tree, click the I2ADL Editor Tab.

- The Synthesizer issues an error and crashes when it encounters an object assigned to a System that does not have any System Objects.
  Workaround: Never assign an object to a System that does not have System Objects, such as the base system.

- When reading a text file, VIVA terminates at a comma when reading in an attribute value. This causes truncation of string constants containing commas.
  Workaround: Use a mechanism other than VIVA-rendered string constants for storage of persistent strings if they contain non-alpha-numeric characters.

- Compiling an input with a constant value whose DataSet is NULL, or which has NULL as a child, will cause VIVA to crash.
  Workaround: Avoid using constants in conjunction with the NULL DataSet.

- Occasional file corruption occurs.
  Workaround: Save VIVA files frequently. Use a versioning preference to minimize loss of work. Do not overwrite previously saved files when attempting to save a file after a GPF.

- "Ambiguous Object" warnings are occasionally issued incorrectly.
  Workaround: If project works as expected, ignore the warning.

- If VIVA is closed before the changes to a new project have been saved, VIVA asks if the changes should be saved. If you choose 'yes', and then cancel, VIVA closes without saving your work.
  Workaround: Save VIVA files before closing VIVA and before switching to a different or new VIVA project.

SECTION 1 – USER INTERFACE

This section will provide you with a description of the VIVA user interface (UI) as well as the basic development concepts used as a VIVA programmer.

VIVA provides an intuitive and robust Graphical User Interface (GUI). The components of the VIVA UI are the main menu, toolbar, programming workspace, tree views, and status bar. Each of these components will be discussed in detail in the following sections. If you are already familiar with other windows-based applications, becoming familiar with the VIVA UI will be quite easy. If you are not familiar with windows-based applications, don't worry; learning your way around VIVA will not be a difficult task. The following figure illustrates the VIVA UI as it appears when you start the VIVA application.

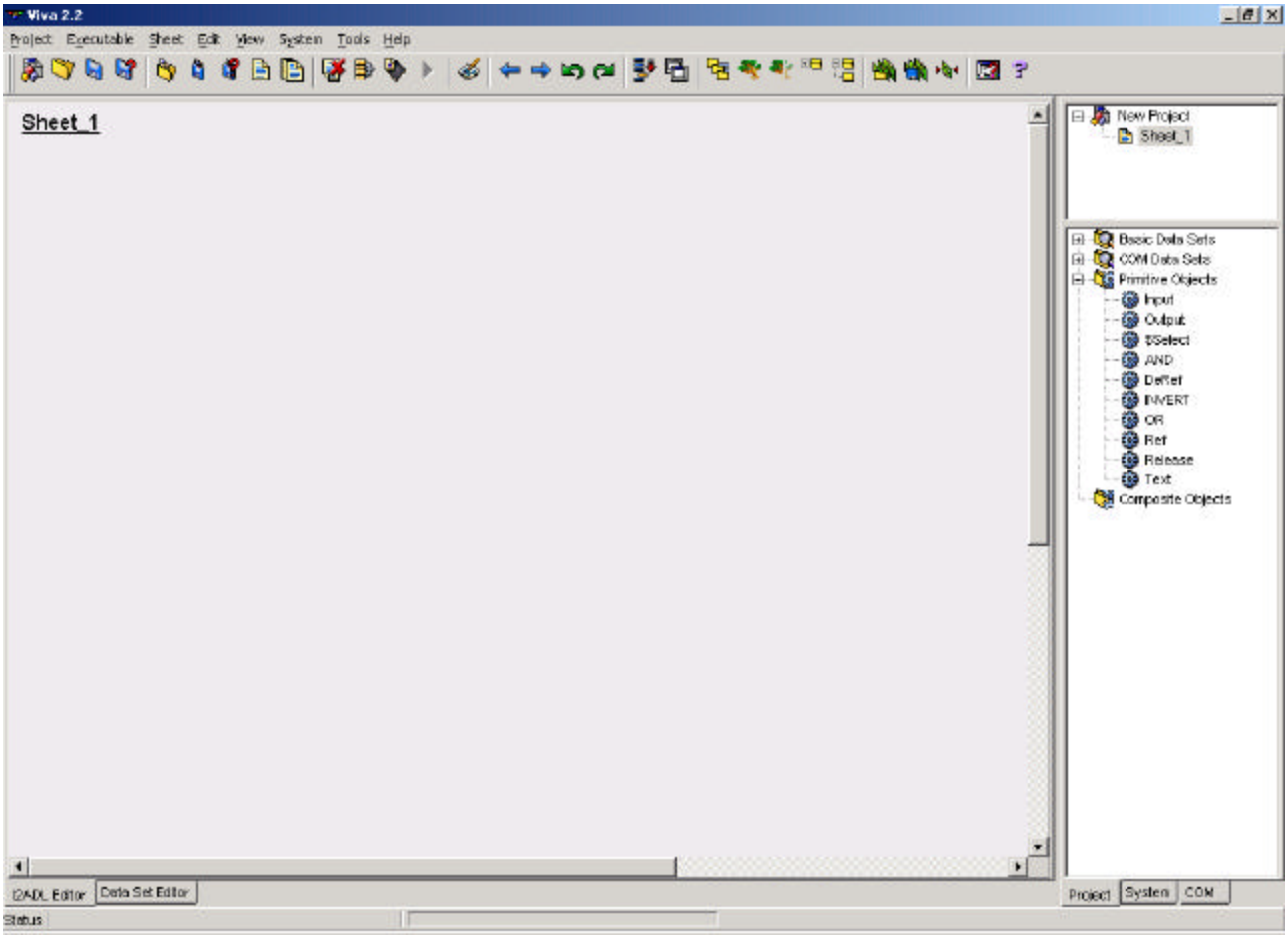


Figure 1 - The VIVA User Interface

1.1 The Main Menu

You will be able to access most VIVA commands from the main menu. There are also keyboard shortcut keys associated with some of the main menu commands. As you become more familiar with VIVA's features, you will be able to more quickly access VIVA commands through the shortcut keys.

The main menu, which looks like the figure below, is located just under the application title bar.

Figure 2 - VIVA Main Menu

1.1.1 Project Menu

Project - file containing a collection of application worksheets, data sets, various objects, and a system description.

System Description – Specific characteristics and definitions of resource management for the system you are targeting. For example, the default target system is the X86.

When you first start the VIVA application, a new project is automatically loaded for you. If you need to save a project and/or retrieve a previously created project, you will need to use the project menu to do so. The project menu is shown in figure 3 below.

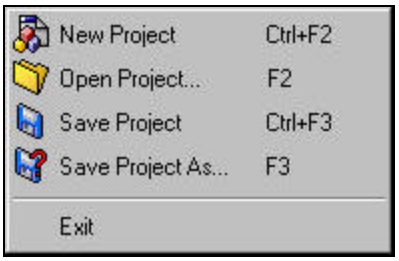


Figure 3 - Project Menu

New Project

Selecting the 'New Project' menu option will create a new VIVA project with a single, default behavior sheet named 'Sheet_1'. and the X86 system description. If changes have been made to the current project when this operation is invoked, the following message is displayed:

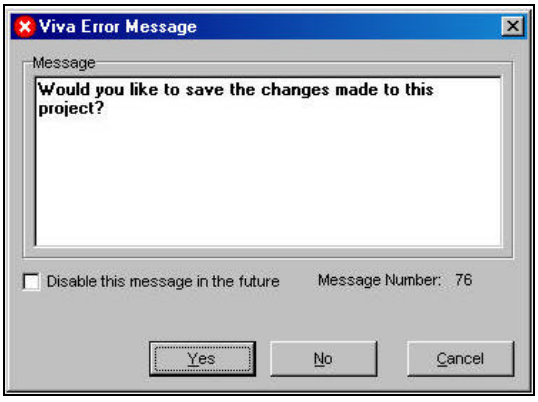


Figure 4 - Save Changes Message

On new projects, The Confirm Close UnNamed Projects preference enables you to turn off the message, “Would you like to save the changes made to this project?”.

The intent of this message is to prompt you to save any changes to the currently loaded file so you will not lose your work. Selecting the ‘Yes’ option will allow you to save the changes while selecting ‘No’ will discard any work you have done since the previous file save operation. ‘Cancel’ just allows you to continue working on the currently loaded file and discontinue trying to load a new one.

Open Project

The ‘Open Project’ menu option gives you the ability to open a file that has been previously saved. After selecting the ‘Open Project’ option, a standard file browsing dialog is displayed. After you have selected a project file using the file browse function, that project’s application worksheets, programming objects, and system description, are loaded for your use.

Save Project

After editing a project file, there are two save options available on the project menu; they are ‘Save Project’ and ‘Save Project As’. The ‘Save Project’ option will save all of the changes you have made to the currently open project file without prompting you for any further information…unless you have not previously named the file. In that case, you will be prompted for the name of the project file. All changes to project application worksheets, objects and system descriptions will be saved in the name given.

Save Project As

The ‘Save Project As’ menu option allows you change the name of the project file you are attempting to save. If you try and use the same file name and location as another project file, a message is displayed asking you to verify that overwriting the existing file is OK; see Figure 5 below.

Exit

Selecting the ‘Exit’ menu option will terminate you VIVA session. Any changes made to the currently open file will be lost unless you save them before exiting! You will be prompted by the message illustrated in figure 4 if you have made any changes that have not been saved.

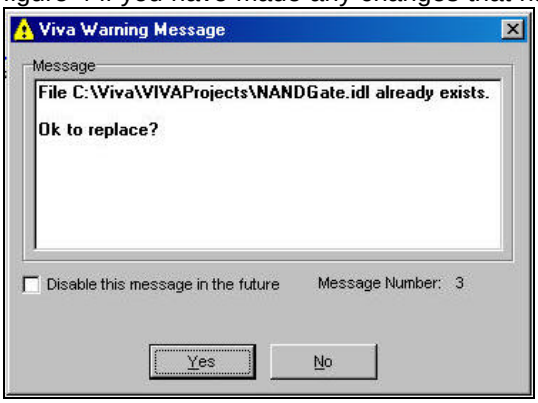


Figure 5 - Replacement Warning Message

VIVA project files are saved with the file extension of .idl

Clicking ‘Yes’ will overwrite the existing project file with the currently opened one. Clicking ‘No’ discontinues the save command.

The project menu will also display a list of the most recently opened and saved project files. This is referred to as the project history. Clicking on a project file in the list opens that file, as if you had opened the project using the 'Open Project' menu option. You can control the number (0 – 15) of recently opened projects displayed using the Preferences option found in the Tools menu.

1.1.2 Executable Menu

EXECUTABLE - A COMPILED APPLICATION THAT CAN BE RUN INDEPENDENTLY FROM THE VIVA EDITOR CANNOT BE OPENED OR EDITED IN VIVA.

In order to create an executable, you must first use the application editor to create an application, successfully compile it, and then save it as an executable file. The executable menu is shown below.

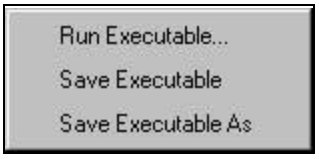


Figure 6 - Executable Menu

Run Executable
To use the 'Run Executable' menu option, you must have already created and saved a VIVA executable file. Selecting the 'Run Executable' menu option will open a standard folder browser that can be used to locate and load an executable file.

Compiled VIVA executables have a file extension of .vex

Save Executable
Once you have created an application that you would like to save as an executable file, simply select the 'Save Executable' menu option. You will then be presented with a standard folder dialog that will require a file name and location. Once you have provided a name and location, clicking 'OK' in the dialog box will complete the save command. You will then be able to select and run the executable file using the 'Run Executable' menu option. If you have previously saved the executable, VIVA will not present the folder browser dialog and will automatically save the current executable.

Save Executable As
If you have previously saved an executable, selecting the 'Save Executable' menu option will automatically re-save the executable file using the current name and location. You can use the 'Save Executable As' menu option to save an executable with a different name or location.

Because a saved executable must run independently from the VIVA editor, all you need to do to execute it is double-click the file name from within Windows Explorer. The executable will load and execute in the same manner as when it was compiled within the VIVA editor!

1.1.3 Sheet Menu

Sheet – application editor workspace to display the application "blueprint".

The application worksheet, or "sheet" for short, provides you with the workspace required to create and display your application's details. The sheet is analogous to a whiteboard where you can create and edit your application. Figure 7, on the following page, provides an illustration of the sheet menu.

Application – a collection of objects that perform a desired function, or functions.

Open Sheet

The 'Open Sheet' menu option will open a standard folder browser dialog box that will allow you to locate a previously saved application worksheet file. After you select a file, all objects and data sets associated with that sheet will be loaded into the current VIVA project. The newly loaded sheet will be added to the sheet list and will become the active sheet in the application editor workspace.

Save Sheet

If the current sheet has not previously been named, you will be presented with a standard folder browser dialog that will allow you to create a sheet file for the current sheet. Otherwise, VIVA will save all changes to the current sheet and does not require user interaction. This is similar to the previously discussed "save" commands.

VIVA application worksheets are saved with a file extension of .ipg

Save Sheet As

If you have previously named a sheet, selecting the 'Save Sheet' menu option will automatically save the sheet using the current name. You can use the 'Save Sheet As' menu option to save a sheet with a different name or in a different location. If you attempt to save the current sheet with the same name and in the same location using the 'save as' command, you will receive the message illustrated in figure 5.

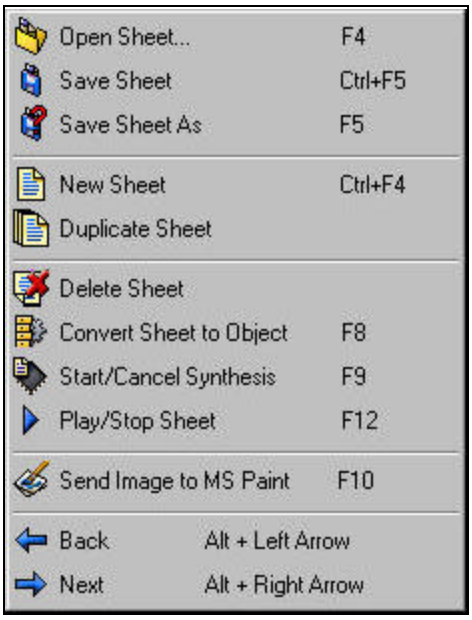


Figure 7 - Sheet Menu

New Sheet

The 'New Sheet' command simply opens a new sheet within your current Viva project. The new sheet is blank and is included in the project sheet list. The sheet list is a tree view of the project located in the upper right-hand pane of the VIVA UI.

Duplicate Sheet

Selecting the 'Duplicate Sheet' command creates an exact copy of the currently opened sheet and includes it in the project sheet list.

Delete Sheet

This command is self-explanatory; it will delete the currently opened application worksheet. You will be prompted for confirmation of the deletion. You can turn off the confirmation associated with deleting a sheet with the Confirm Delete WIPSheet preference.

Convert Sheet To Object

Once you have created an application on a sheet, you can then convert the sheet into a VIVA object. Inputs and outputs of the created object will be of the same order and type as the defined application. For example, if you have 2 inputs, A and B, and a single output, Q, all of which are word types, the resulting VIVA object has the same inputs and outputs as word types. Before saving the object, you are given a final chance to enter the information about the object; information such as the object name, input and output node names, and documenting the function of the object and its intended use. Object information is collected through the 'Convert

Sheet to Object' dialog box; see figure 8. To maintain consistency, the 'Convert Sheet to Object' dialog box is essentially the same as the 'Edit Attributes' dialog box (which will be explained is a subsequent section).

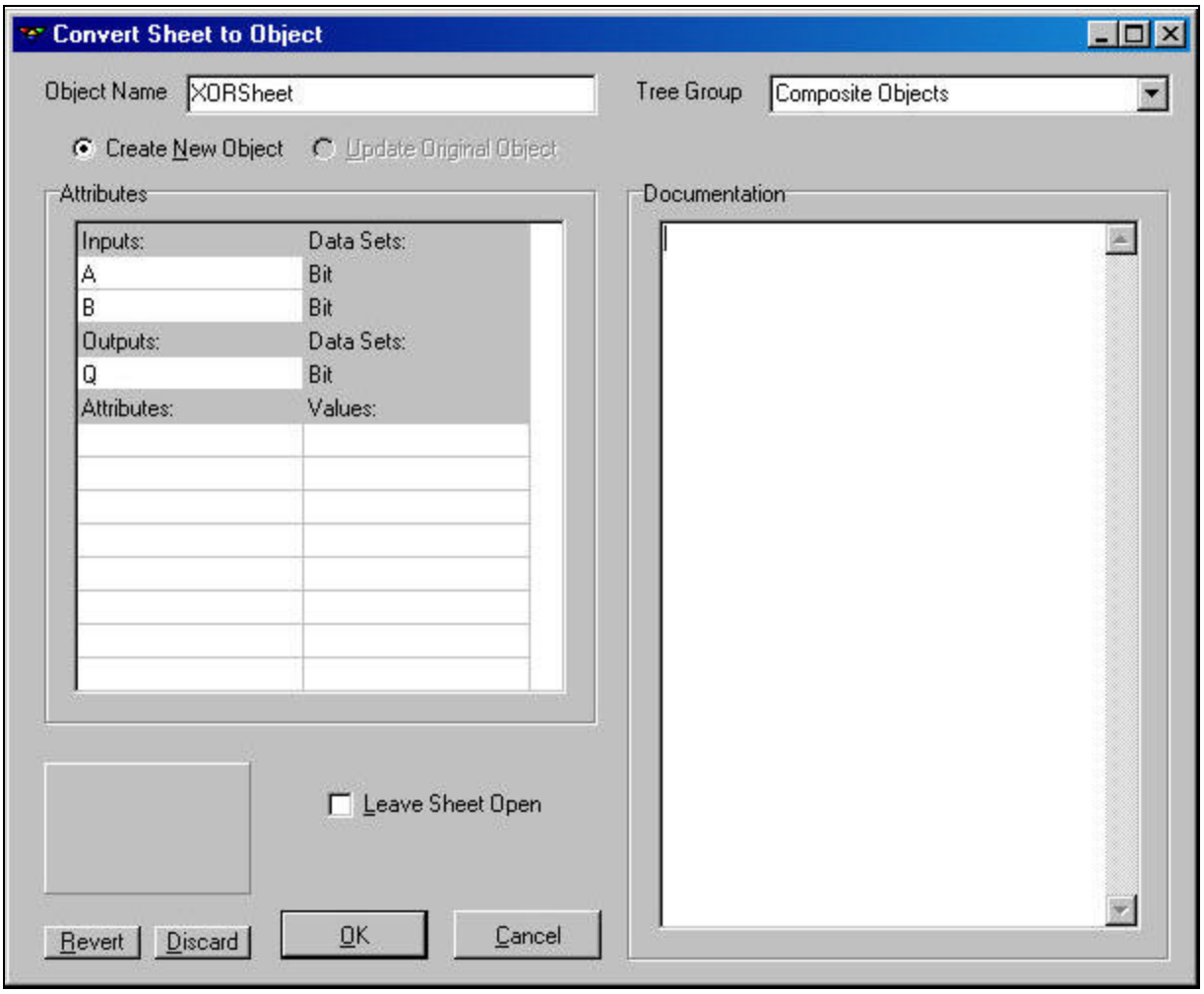


Figure 8 - Convert Sheet to Object Dialog Box

Each of the fields and sections of the 'Convert Sheet to Object' dialog box are explained in the following table.

Dialog Box Field/Section	*Field/Section Description*
Object Name	The name of the object. Name is case sensitive.
Tree Group	The intended object group for this object after conversion. You can select one of the existing groups from the drop-down list.
Create New Object	Select this radio button if you intend to create a new object.
Update Original Object	Select this radio button if you intend to update the original object. Auto-selected if name is changed on existing object.
Attributes	This section provides a description of all of your sheet's inputs, outputs, specific attributes, and data types.

Documentation	Use this section to describe the object's function and how it should be used. This is a free text field. However, some consistency in description format is beneficial.
Leave Sheet Open	By default, VIVA closes a sheet after it is converted into an object. Checking this box will keep the sheet open after conversion.
Revert	If your object has the same name as one of the .BMP files in the VivaSystem\Icons folder, it will display that icon by default. 'Discard' will rename the .BMP file, causing the icon to be removed from the object. The Revert button will re-instate the .BMP file name.
Discard	See 'Revert' above.
OK/Cancel	Standard buttons to complete or cancel the intended action. In this case it is the conversion of a sheet into a VIVA object. Cancel will not discard icon changes because changes take effect immediately.

Table 1 - Convert Sheet to Object Dialog Box Description

Start/Cancel Synthesis
Using this menu command will compile and execute the current application worksheet. Selecting this menu command during the compilation of an application worksheet will stop the compilation process and return you to the worksheet editor.

Synthesis – to compile an application worksheet.

Play/Stop Sheet
After you have compiled an application worksheet, you can choose to execute it at any time using this command. If you have yet to compile a worksheet, this command is unavailable. Changes made since the last compile will not be reflected until the sheet is recompiled.

Send Image to MS Paint
This menu command allows you to take a snap shot of your application worksheet for use as a Microsoft Paint graphic. Only the visible part of the sheet is captured.

The most common uses for this command are for documentation and reviewing the worksheet contents by others not using VIVA.

Back and Next
These two commands are used to navigate forward and backwards through the project worksheets that have recently been viewed.

The sheet menu will also display a list of the most recently opened/saved sheet files. This is referred to as the sheet history. Clicking on a sheet file in the list opens that file, as if you had opened the sheet using the 'Open Sheet' menu option. You can control the number (0 – 15) of recently opened projects displayed using the Preferences option found in the Tools menu.

1.1.4 Edit Menu

Commands on the VIVA Edit menu are similar to those found in other windows-based programs.

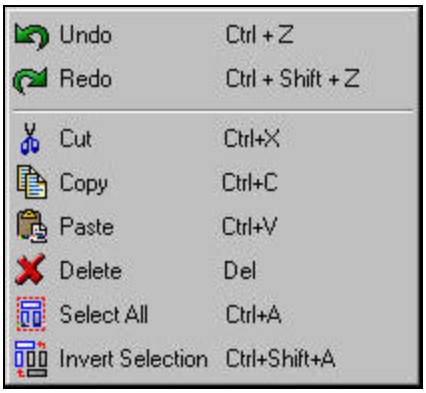


Figure 9 - Edit Menu

Undo/Redo
Your edits are maintained by VIVA so that if you decide to 'undo' a particular action, you can do so using the 'Undo' command. 'Redo' is the negation of an undo.

Cut
After selecting a specific piece of your application worksheet, this command is used to remove it from its current location on the sheet.

Copy
After selecting a specific piece of your application worksheet, this command is used to save the selection to the Viva clipboard, allowing you to duplicate the selection elsewhere in the project.

Paste
Once you have either “cut” or “copied” a particular selection of your application, using the ‘Paste’ command will place the cut or copied selection onto the selected application worksheet.

Delete
This one should be a bit obvious. You can delete a selection with this command.

Select All
Using this command will select all of the contents of the current application worksheet, as opposed to particular portions of the worksheet.

Invert Selection
This command allows you to invert a previous selection. In other words, select the worksheet contents not previously selected and unselect the content that was selected.

1.1.5 View Menu

The purpose of the view menu commands is to allow you to do some customization of the VIVA UI to better suit your particular likes and dislikes.

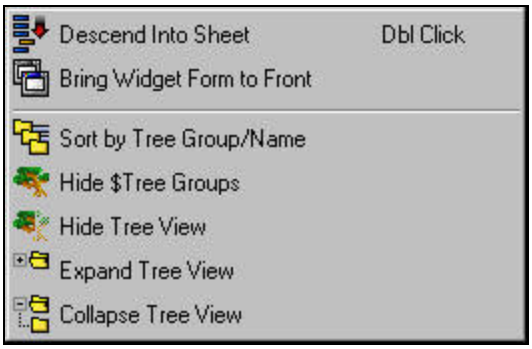


Figure 10 - View Menu

Descend Into Sheet
This command provides you with the capability to drill down into the details of a VIVA composite object. By placing an object onto the current application worksheet, you can then use this command to view the particular details of that object. For example, if you have created a composite object, and used that object in your current programming efforts, using the ‘Descend Into Sheet’ command will show you the detail behind your composite object.

This command can also be executed by double-clicking on the desired object. As the detailed view of the object is presented in a new worksheet, the worksheet is added to the current project sheet list with the name of the object as the name of the sheet. This command has no effect on primitive objects because such objects have no detail to display. The command is also ineffective if more than one composite object is selected.

Bring Widget Form to Front

If you are executing an application, and have lost its “widget interface window”, the ‘Bring Widget Form to Front’ button enables you to bring that window to the front of your desktop, making it visible again.

Sort by Tree Group/Name
This command will sort the object tree in alphabetical order. The object tree is found in the lower right-hand pane of the VIVA UI.

Hide $Tree Groups
If you would like to hide all of the objects in tree groups that begin with the ‘$’ character, use this command.

Hide/Show Tree View
Using this command, you can expand/reduce the amount of application workspace by hiding/displaying the object tree views found in the right-hand panes of the VIVA UI.

Expand Tree View
Opens tree groups (or branches) of the object tree so that all objects are visible.

Collapse Tree View
This will display only the root tree groups in the object tree. You can accomplish the same thing by clicking on the minus sign associated with each object separately. Expanding the object data sets is simply a matter of clicking on the plus signs for the desired object data set.

1.1.6 System Menu

System – the description of a target environment’s behavior.

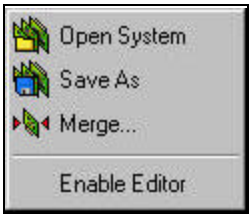


Figure 11 - System Menu

Open System
The ‘Open System’ command loads a system description into the current VIVA session.

Save As/Merge
These commands are reserved for advanced use in developing new system interfaces.

Enable Editor
Activate the Resource Editor and the System Editor. The VIVA UI indicated the activation of these editors by displaying two appropriately labeled tabs at the bottom of the application workspace. The editor tabs look like this:

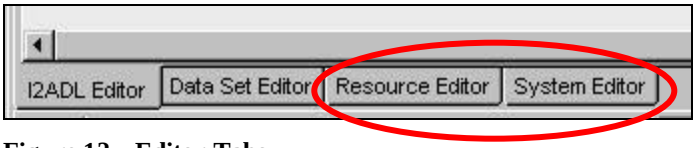


Figure 12 - Editor Tabs

The system menu will also display a list of the most recently opened system files. This is referred to as the system history. Clicking on a system file in the list opens that file, as if you had opened the system using the 'Open System' menu option. You can control the number (0 – 15) of recently opened systems displayed using the preferences option found in the tools menu.

1.1.7 Tools Menu

The tools menu allows you access to the available user-controllable settings within VIVA as well as various predefined objects.

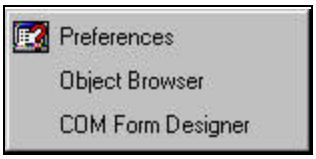


Figure 13 - Tools Menu

Preferences
There are four separate tabs associated with the VIVA preferences; they are Load, Files, Window, and I2ADL Editor. These tabs represent the user-controllable settings available in VIVA for customizing how you would like VIVA to look and function. Each tab will be discussed in detail.

Load
The purpose of this tab is to provide you with an opportunity to determine the behavior of VIVA as it loads a project. There are only two basic load functions available for modification.

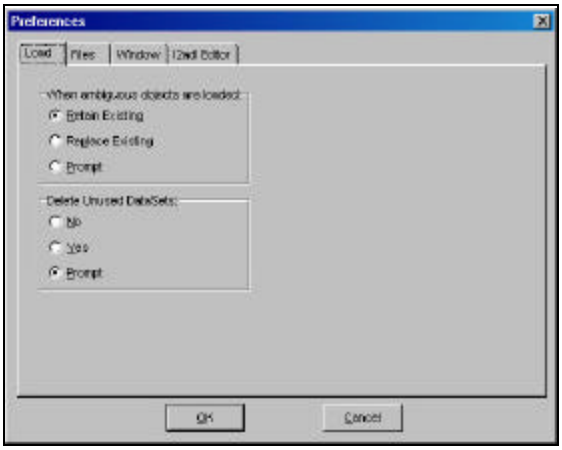


Figure 14 - Tools/Preferences Load Tab

Selecting either the 'Retain Existing' or the 'Replace Existing' radio button will perform the specified action automatically without user intervention. However, if you select the 'Prompt' radio button, VIVA will prompt you to select the retain or replace option. **The default selection is 'Retain Existing'**.

Star Bridge Systems recommends that you generally specify 'Replace Existing.'

The second function available on the 'Load' tab deals with unused data sets. An unused data set is one that has no objects in use by the project on any of the application worksheets. VIVA will automatically check for unused data sets. If VIVA finds unused data sets, it performs the action you specify on the 'Load' tab for unused data sets.

Selecting either the 'No' or 'Yes' radio button on the 'Load' tab will perform the selected action without user intervention. If you select the 'Prompt' radio button, VIVA will allow you to specify the action to be performed. This is done through the message illustrated in figure 15 below. You can control the action that VIVA will take by clicking on the 'Yes' or 'No' button when the message is displayed. While the message is displayed, you have the option of resetting your unused data set preference by checking the 'Disable this message in the future' check box. Once you select the 'Yes' or 'No' with the box checked, your preference will be adjusted accordingly. **The default selection is 'Prompt'**.

Figure 15 - Unused Data Sets Message

Remember, these two functions only take place when you load a project using the Project/Open Project menu command.

Files

The purpose of this tab is to provide you with the opportunity to change the number of recently used files presented in the different histories and specify the VIVA behavior associated with any save command.

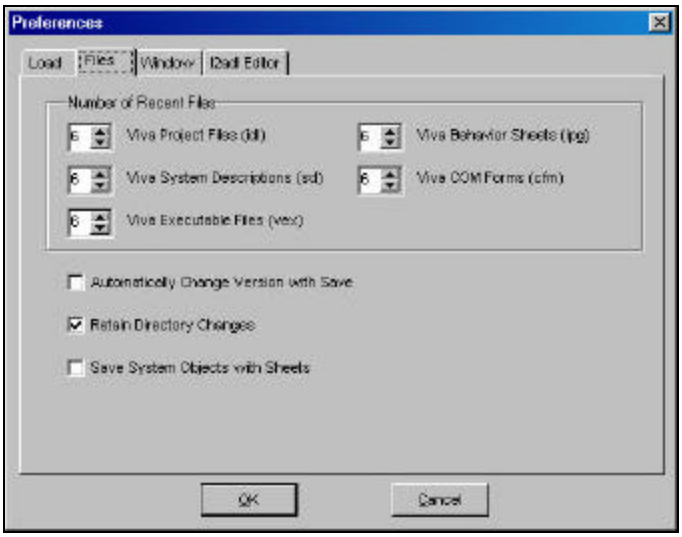


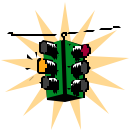
Figure 16 - Tools/Preferences Files Tab

The file histories available are project, sheets, systems, COM forms, and executables. The number of recently used files displayed is changed by either entering a number directly or selecting a number using the spin boxes. **The allowable range is 0 to 15 with the default value being 6 for each file type**.

Each specific file type is discussed in their respective sections of this document.

The **'Automatically Change Version with Save'** option allows you to specify an automatic file naming convention when saving files of the same name. This will help you avoid an inadvertent loss of data. For example, if you are working on a project named 'MyProject', and it has already been saved to disk, having this option specified will change the file name to 'MyProject0'. The rules for file automatic versioning are:

- If the name of the file being saved does not contain both alpha and numeric characters, then the new file name will be created by appending a "0" (zero) to the original file name; like the example above.
- If the last character of the file name being saved is numeric, then the new file name increments the numeric suffix. For example, 'MyProject0' becomes 'MyProject1', 'MyProject10' would become 'MyProject11', and so on.
- If the last character of the file name being saved is alphabetic, then the new file name increments the alphabetic suffix. For example, 'MyProject2003a' becomes 'MyProject2003b', 'MyProject2003q' would be come 'MyProject2003r', and so on.



Application worksheets that contain polymorphic behaviors will not have their file names changed!

Enabling the **'Retain Directory Changes'** option tells VIVA to remember the most recently browsed-to locations when displaying file open and file save dialog boxes. This can save time when selecting a location for saving file to or retrieving files from specific locations.

Selecting the 'Save System Object with Sheets' option allows you to embed the currently used system objects in the application worksheets you are saving. This will provide the capability for you to save custom objects with individual sheets.

Window

The window tab contains only two simple settings. The first, titled 'Object/System Tree', allows you to choose whether or not the tree views in the right-hand pane automatically adjust their positioning to fit the contents of the pane. As the object/system tree panes expand and contract, the application workspace is adjusted as well.

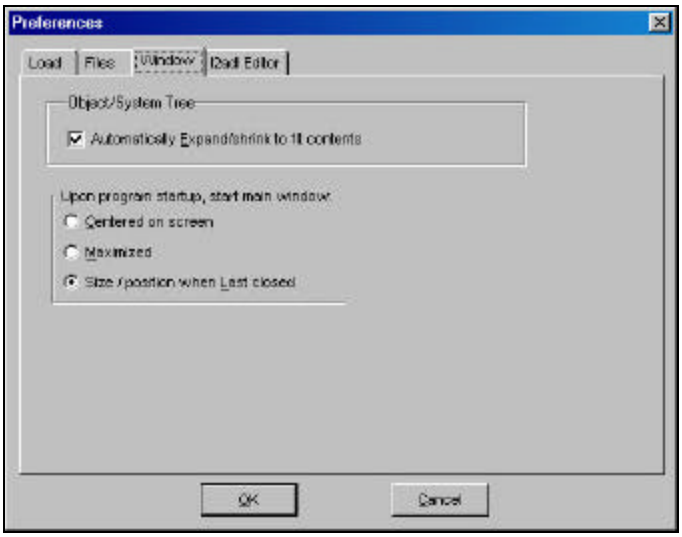


Figure 17 - Tools/Preference Window Tab

The only other setting on the windows tab allows you to specify where the VIVA application window is displayed, in relationship to your desktop, at the time the application is started. The options for this function are centered, maximized, and the size and position of the window when it was closed last.

I2ADL Editor
You may set the background color of the application workspace by clicking on the 'Change' button and selecting a color from the displayed color palette. **The default value is white**.

The 'Node Snap Tolerance' setting, in pixels, is used to determine how closely an object node needs to come to another object node in order to become automatically connected. **The default value is 11 pixels**.

The 'Confirm' section of this preferences tab allows you to specify when a confirmation dialog is provided by VIVA. By checking the boxes provided, a confirmation message will be displayed when the associated action takes place. **The default setting is for all boxes to be checked**. For example, with the 'Close Unnamed Projects' box checked, you will be prompted to confirm a save of the open project you are trying to close if you have not already saved it with a name other than the system generated one.

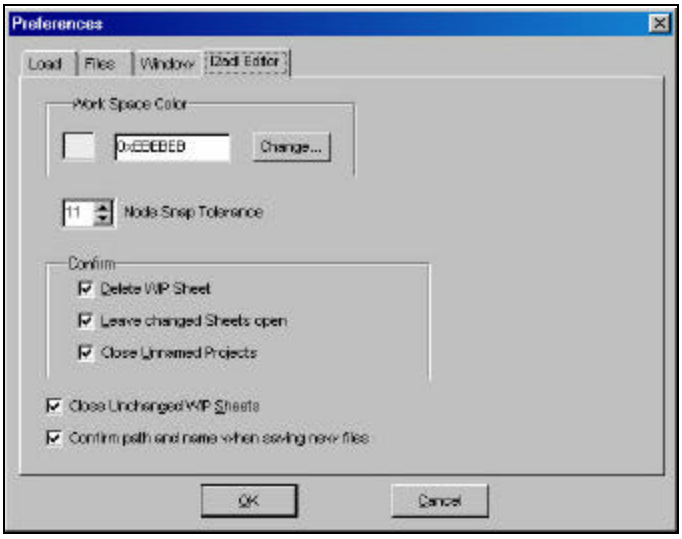


Figure 18 - Tools/Preferences I2ADL Editor Tab

The 'Leave changed sheets open' is used for when you want VIVA to keep a sheet open, even though you have made changes to it, when you switch from one sheet to another.

The 'Confirm path and name when saving new files' option indicates to VIVA that whenever you save a new file without first providing a name or a location, you will receive a prompt to specify them. Otherwise, VIVA will save the new file in the default VIVA directory using 'New <File>' as the default name; where <File> is the type of file being saved. For example, a project will be saved as 'new project' and a new sheet will be saved as 'new sheet'. **The default is to have this option checked and force a name and location when you save a new file**.

Object Browser/COM Form Designer
Both of these menu commands deal with Component Object Model (COM) features. They allow you to utilize predefined COM objects as well as create forms that host ActiveX controls.

These features are discussed in detail in sections 3.5 and 3.6 respectively.

1.1.8 Help Menu

The VIVA help menu provides features similar to other applications help menus. This is where you can find the VIVA online reference, your installation's version number, and copyright information.

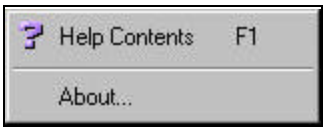


Figure 19 - Help Menu

Help Contents
It is the intent of this menu command to provide you with this User Guide file in an HTML format. By using HTML, the file becomes a convenient to navigate through self-help system. As you learn VIVA, this file will help answer your questions. As you become more proficient with VIVA, your reliance on this file will become less.

About
Anytime you need to reference the version and copyright information for your VIVA installation, you use this menu option. The following figure illustrates the 'About' window.

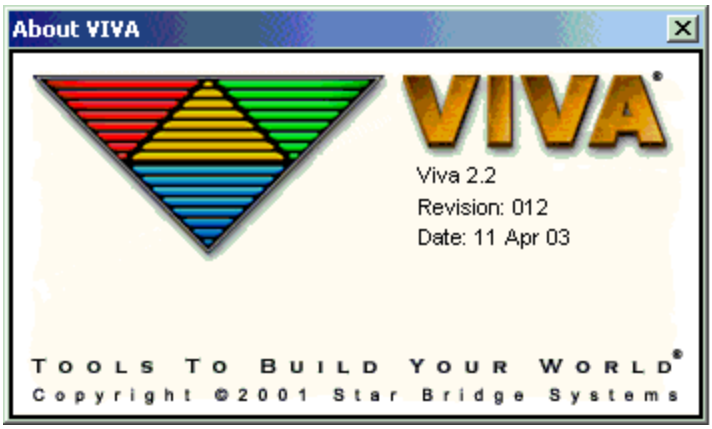


Figure 20 - Help About Window

1.2 The Tool Bar

Most of the major functions available from the menus can be executed more efficiently by using the tool bar. The tool bar is located directly under the menu. Of course, it does require some familiarization with the tool bar icons. However, it will not take long!

Figure 21 - The VIVA Tool Bar

Each icon, its description, and the menu command equivalent is provided in the table below. Floating your cursor over each of the tool bar icons will automatically display the associated description.

Icon	*Description*	*Menu Equivalent*	*Keybaord*

	New Project	Project/New Project	Ctrl+F2
	Open an Existing Project	Project/Open Project...	F2
	Save Project	Project/Save Project	Ctrl+F3
	Save Project Under a New Name	Project/Save Project As...	F3
	Open Sheet	Sheet/Open Sheet...	F4
	Save Sheet	Sheet/Save Sheet	Ctrl+F5
	Save Sheet Under a New Name	Sheet/Save As...	F5
	Create a New Sheet	Sheet/New Sheet	Ctrl+F4
	Duplicate Sheet	Sheet/ Duplicate Sheet	
	Delete Sheet	Sheet/Delete Sheet	
	Convert Sheet to Object	Sheet/Convert Sheet to Object	F8
	Start/Cancel Synthesis	Sheet/Start/Cancel Synthesis	F9
	Play/Stop Sheet (Play)	Sheet/Play/Stop Sheet	F12
	Play/Stop Sheet (Stop)	Sheet/Play/Stop Sheet	F12
	Send Image to MS Paint	Sheet/Send Image to MS Paint	F10
	Back	Sheet/Back	Alt+Left Arrow
	Next	Sheet/Next	Alt+Right Arrow
	Undo	Edit/Undo	Ctrl+Z
	Redo	Edit/Redo	Ctrl+Shift+Z
	Descend Into Sheet	View/Descend Into Sheet	Double-Click
	Bring Widget Form to Front	View/Bring Widget Form to Front	
	Sort By Tree/Group Name	View/Sort By Tree/Group Name	

	Hide $Tree Groups	View/Hide $Tree Groups	
	Hide Tree View	View/Hide Tree View	
	-	View/Expand Tree View	
	-	View/Collapse Tree View	
	Open System	System/Open System	
	Save System	System/Save As	
	Merge System	System/Merge...	
	VIVA Preferences	Tools/Preferences	
	Help Contents	Help/Help Contents	F1

SECTION 2 - VIVA OBJECTS

A VIVA object is similar to an object in C++, Java and other programming languages. The primary difference between a VIVA object and an OOP-language object is that VIVA objects are defined more in terms of decision paths, data flow and parallelism. Basically, an object is responsible for performing some sort of data manipulation, decision making, or to facilitate efficient data flow. VIVA object attributes include input parameters and output values. These parameters and values are represented as "nodes".

Node – an attachment point to a VIVA object. Nodes are either inputs to or outputs from an object. They are depicted by a colored square.

Nodes are represented in the VIVA UI as small colored squares attached to the objects. Input parameters are displayed as nodes on the left side of the object while output values show up as nodes on the right side. The example below shows the VIVA 'AND' object. The input nodes are the red squares on the left and the output node is the one on the right.

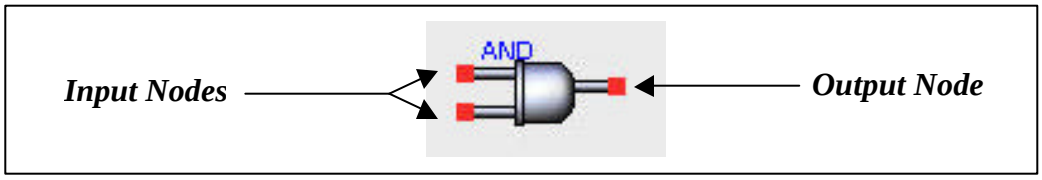


Figure 22 - VIVA Nodes Example

An application developed using VIVA is hierarchical in nature. That is, complex objects are built using more simple objects. There are several primitive objects as part of the standard VIVA object library. Objects can also found in application worksheets and system descriptions. This section will discuss objects and their properties.

As you design your applications for implementation using VIVA, it is important to remember that your core algorithms should be designed as early as possible. Having adequate algorithm designs is crucial to successfully constructing your solution using VIVA's efficient data flow and parallel capabilities.

Several Primitive Objects are resident within VIVA itself. Objects are also found in behavior sheets and system descriptions. This section discusses Primitive Objects, Object Attributes, Object Footprints and Object Polymorphism.

2.1 Object Attributes

Each VIVA object possesses a variable number of attributes. Right-clicking an object accesses the attributes dialog for that object. Different objects have different allowable attributes. For example, an input object would have the attributes illustrated in figure 23 below.

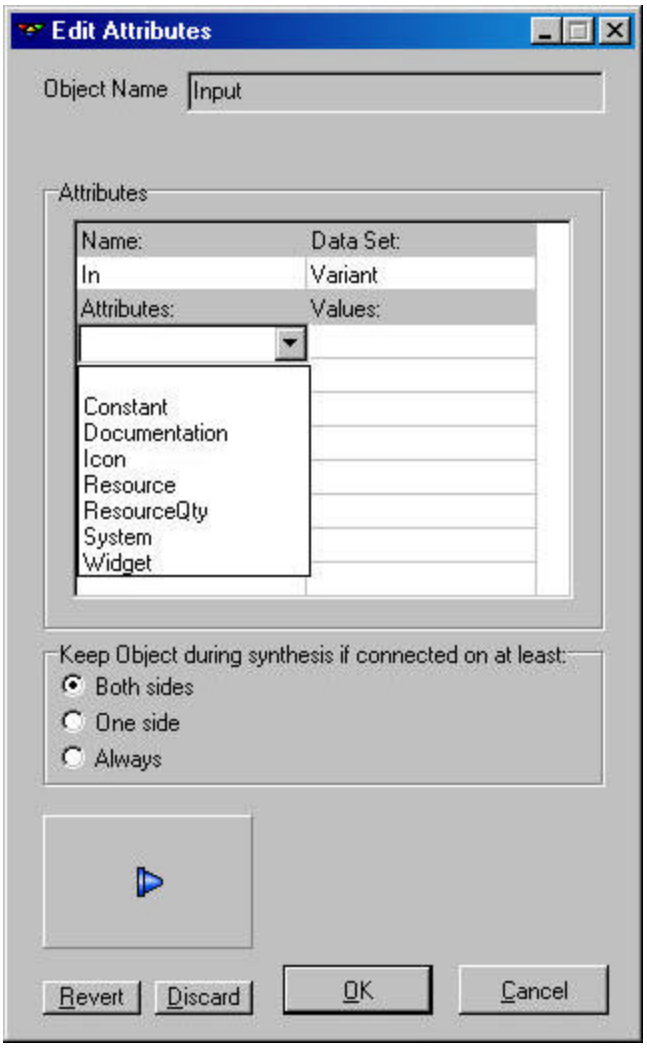


Figure 23 - Object Attributes

2.1.1 Object Attribute Definitions

This section provides the definition for each of the attributes found in the drop down list found on the 'Edit Attributes' dialog.

Attribute	Description
Constant	Used to constrain the value of an object to a fixed value that is typed in the 'Values' column. This is typically used with input objects. By default, all constant input can be overloaded in a sheet containing the object. Connecting a transport to the input node does this. A constant input that is not overloaded will expose its predefined value. Placing an asterisk before the value will "hide" the input, meaning it will not show up in the object footprint. It will therefore not be overloadable.

Documentation	Used to add object documentation that is visible next to the object's name in the Object Tree.
Icon	Name of a bitmap file residing in the Viva\VivaSystem\Icons directory that will be used as the object icon.
Resource	When Resource is selected from the Name dropdown list, the Values column becomes a dropdown list of Resources defined for the currently loaded system. Select the Resource you wish to tie the object to.
ResourceQty	In some cases, multiple instances of a Resource can be assigned as an object attribute. Specify the desired number of resources to assign by typing a number in the Values column.
System	Allows for the assignment of the object to a specific System. A list of Systems is populated in the Values column. This Attribute can be used to partition various objects into the Hypercomputers resources. Processing Elements defined within the currently loaded System Description. This Attribute can also be used to tie VIVA objects to I/O buses and System clocks.
Widget	Use this attribute to override the default widget-type for a data set on a particular input or output.

2.2 Object Footprints

The "footprint" of an object is made up of the name and the name and number of inputs and outputs the object contains. As an example, the following diagram illustrates a VIVA application that contains an object with 3 data inputs, labeled 'A', 'B', and 'C', and a single output value labeled 'O'. There are also nodes labeled 'Go', 'Busy', 'Done, and 'Wait'.

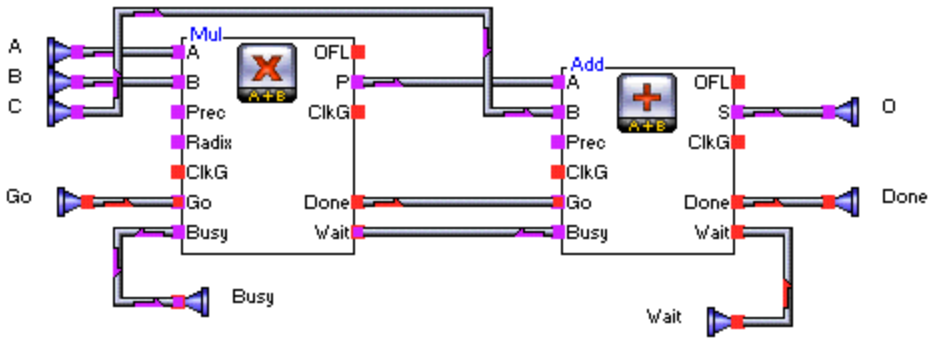
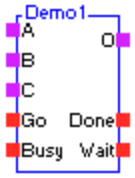


Figure 24 - Object Footprint Example One

The object in figure 23 multiplies the input values of A and B then adding the product to the input

value of C. When this sheet is converted to an object, using the name Demo1, the objects footprint becomes:



Each data input for the object in figure 23 became an input node on the left side of the Demo1 object. Each data output on the sheet became an output node on the right side of the Demo1 object. Node data types are maintained from the application sheet to the object footprint upon conversion. If we change the application in figure 23 to specifically work with the data type of byte, the object would look similar to the one in figure 23 but the nodes would indicate a different type by displaying a different color. This is shown in figure 24 below.

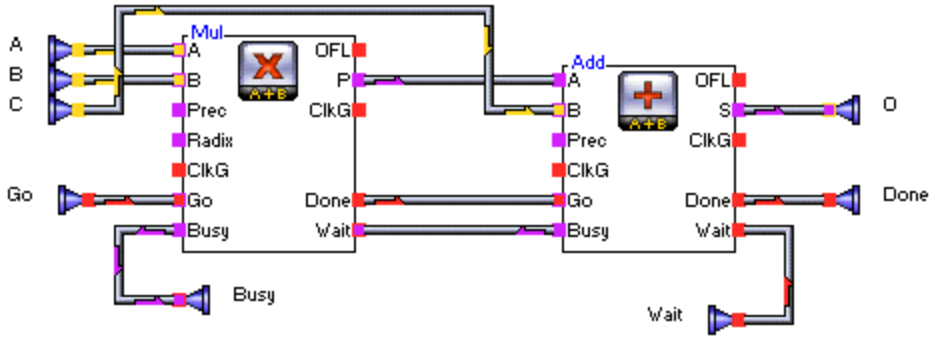
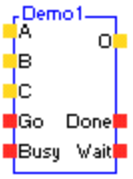


Figure 25 - Object Footprint Example Two

The object performs that same function as before and the footprint remains the same. The only difference is the data type being used for the inputs and outputs.



More information on object footprints is provided in the section titled 'Variant Select'

2.3 Primitive Objects

The following named objects are part of what is referred to as VIVA's primitive objects. These objects are found in the project tree group called 'Primitive Objects'.

2.3.1 Input

The input object is used to designate input parameters to an object. As a general rule, each input object on an application worksheet becomes an input node when the sheet is converted to an object itself.

2.3.2 Output

The output object is used to designate output values from an object. As a general rule, each output object on a behavior sheet becomes an output node when the sheet is converted to an object itself.

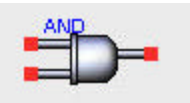
2.3.3 $Select

$Select is a special multiplexor that can provide a selectively compiled portion of an application. Upon compilation, the $Select object is removed from the application and the applicable compiled value is put in its place. The only applicable values for the $Select object's 'S' input node must be either a 1 or a 0 at compile time. The other inputs for the $Select object may be of any data set. If input 'S' resolves to 0 at compile time, the #0 input is compiled into the application. If input 'S' resolves to 1, the #1 input is included in the compilation.

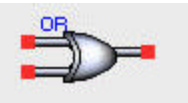
For more information on the $Select object, see the section on selective synthesis.

2.3.4 AND, OR, INVERT

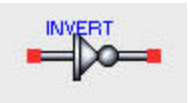
There are three bit-wise objects that make up VIVA's primary logical operators. They are the AND, OR, and INVERT objects. The representation of and truth tables for each of these logical operators are shown below.



A	B	Output
0	0	0
0	1	0
1	0	0
1	1	1



A	B	Output
0	0	0
0	1	1
1	0	1
1	1	1



A	Output
0	1
1	0

2.3.5 Text

The text object is the primary means of documenting VIVA applications. To add a text object to an application sheet, simply drag and drop the object on the desired sheet. Once the object has been placed on a sheet, you can enter the appropriate documentation.

There is no limit to the number of text objects you can place on a sheet. You can format the text attributes by right-clicking the text box and setting the desired attributes; for example, text font.

2.4 Object Polymorphism

Polymorphism – the ability for an object to retain a predefined behavioral design yet retain the ability to handle various data set types.

Polymorphism is achieved using a combination of overloading, recursion, and compile-time resolution.

2.4.1 Overloading

Overloading – having different objects of the same name with different behavioral characteristics or operate on different data set types.

An example of overloading would be a 'Convert' object that has the ability to convert an unsigned integer to a signed integer. Then, using a similarly named object, 'Convert', convert a fixed point number to a signed integer.

When you create overloaded objects, they are grouped together into the Composite Objects group in the object tree. VIVA resolves an overloaded object appropriate to the application at compile time. Using this technique, you can create applications that will work in various situations without having to modify it!

2.4.2 Recursion

You may ask, "How can a single predefined behavioral design define how to perform addition for all unsigned integers, or all signed integers, etc?" The answer is in the use of recursion.

An object that is depicted with input and output nodes is associated with a series of overloaded functions. By definition, these overloaded functions have the same name yet handle the different implementations of that particular function for different input data types. Composite objects with variant input types can be self-referential (recursive) as long as the referenced objects operate on simpler data types than the original data type. This rule allows for the creation of recursive

decompositions until the object is broken down into an operation on a more fundamental data type. The following table illustrates the hierarchical order of currently defined VIVA data types.

Order	*VIVA Data Type*	*# of Bits*	*Data Type Description*
Low	Bit	1	The lowest order and most basic type.
	Dbit	2	2-bit cardinal data type
	Nibble	4	4-bit cardinal data type
	Byte	8	8-bit cardinal data type
	Word	16	16-bit cardinal data type
	Int	16	16-bit signed integer
	Fix16	16	16-bit fixed point
	Dword	32	32-bit cardinal data type
	Dint	32	32-bit signed integer
	Fix32	32	32-bit fixed point
	Float	32	32-bit IEEE floating point
	Qword	64	64-bit cardinal data type
High	Double	64	64-bit IEEE floating point

Table 2 - VIVA Data Types

2.4.3 Data Set Polymorphism

Data set polymorphism refers the ability for VIVA to create an appropriate implementation at compile time depending on the defined data type of a variant object's inputs at that time. That is, VIVA can wait until compile time to determine the precise implementation of your application based on a conditionally defined input for given objects.

2.4.4 Data Rate Polymorphism

Data rate polymorphism is similar to data set polymorphism in that VIVA can select an appropriate implementation based on compile time needs. However, unlike data set polymorphism, VIVA will select the precise implementation of your application based on the need for speed (data rate).

Further details on the concept of object polymorphism can be found in the following sections.

Section 4.4 – Vectors
Section 4.6 – Data Set Recursion
Section 6 – Variant Select, Recursive Footprints, Data Set Polymorphism

SECTION 3 – COM OBJECTS

3.1 What is COM?

Basically, Component Object Model (COM) is an operating system wide convention for code exposure. It is a sort of "universal interface"; code written in any language can be exposed using COM and be used in any language that utilizes the COM interface. OLE, ActiveX, COM+, and DCOM are all derived from COM.

Most popular programming languages have their own system for both using COM components via the COM interface, and for creating COM components. While VIVA does not presently have the ability to create COM components, it does enable the user to utilize existing COM components, including OLE and ActiveX controls.

The functionality of COM components is implemented in Dynamic Link Libraries (DLL files) and ActiveX Control files (.OCX files). These files contain what is known as a “type library”. A type library contains one or more COM Objects (perhaps better known as "Classes"), ActiveX controls, data type definitions, and/or enumerations. In order to utilize one of these files, it must be registered on your system; whether you are developing or executing them. Registration of COM object files is discussed in the following section.

3.2 Registering DLL/OCX

In order to make a type library available for use, you must first register it on your system. This section provides instruction on how to go about the registration process.

To register a DLL or OCX file on your system, you must run the program "Regsvr32.exe" located in the Windows/System directory. The path and filename of the file that you want registered is passed a parameter on the command line. You can accomplish the same thing by dragging the icon of the DLL or OCX file over the Regsvr32 icon. See the illustrations below for an example of what the registration process looks like.

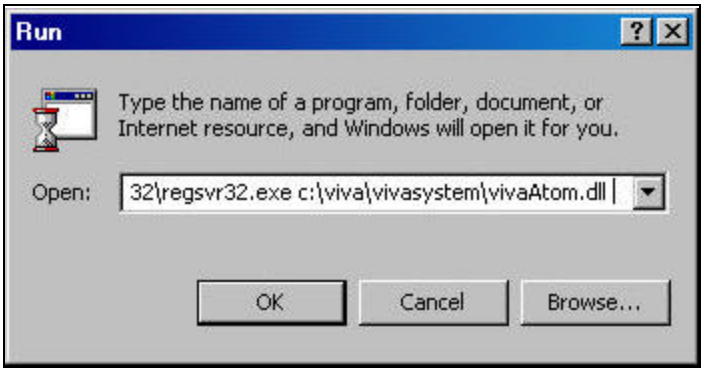


Figure 26 - Command Line COM Registration

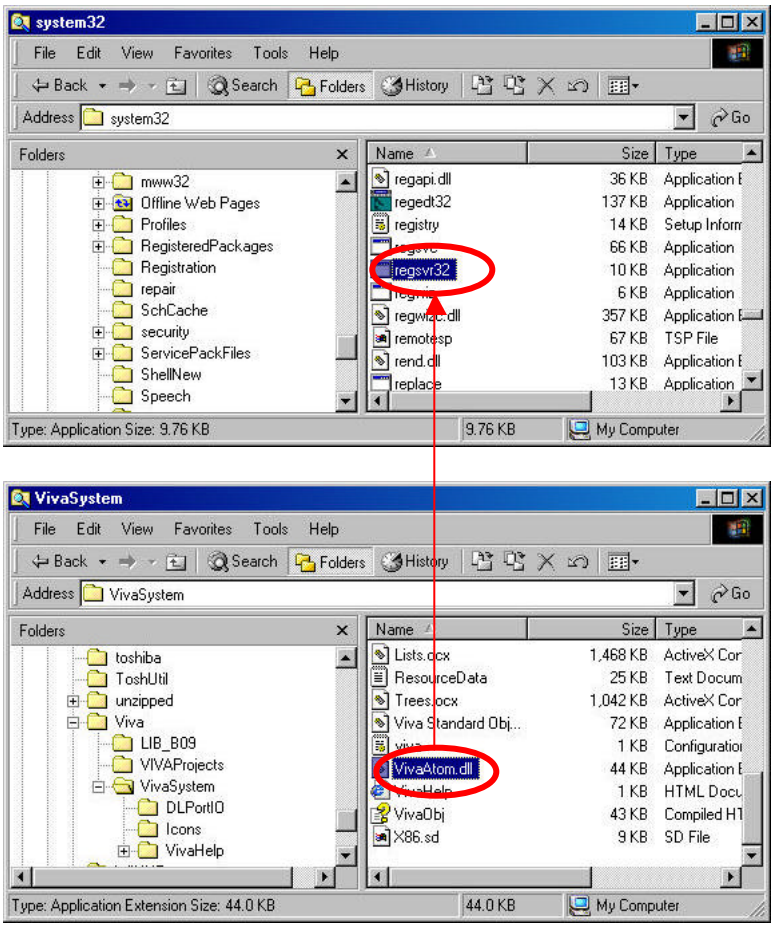


Figure 27 - Drag & Drop COM Registration

3.3 Using COM in VIVA

To view a list of existing type libraries on your system, and the components and definitions that they contain, activate the 'Object Browser' using the appropriate menu option from the 'Tools' menu.

You can also include any of these libraries for use in VIVA using the object browser menu option as well. Once you have included the type libraries containing the components that you want to use, you can instantiate these components and access their member variables, properties, functions, and events. Use of the included COM libraries is done in the same manner as other objects; by using the class and object trees found on the COM tab of the object tree pane.

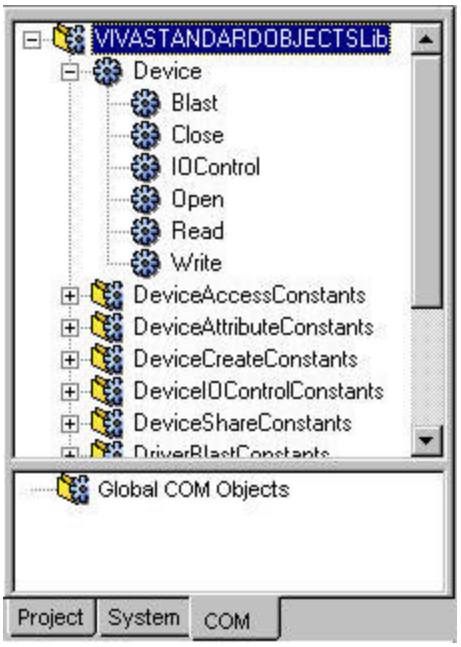


Figure 28 - COM Tab/Object Tree

The COM class tree, the top part of figure 28 above, contains the VIVA standard object library. Each class and enumeration within that library is shown as a branch of the tree. For example, 'Device' and 'DeviceAttributeConstants' are both branches (classes) of the VIVA standard object library while 'Open' and 'Read' are both enumerations of the 'Device' class.

Each branch that represents an enumeration can be used by dragging and dropping it onto an application sheet. Placing an enumeration on an application sheet will instantiate an input having a name and constant value appropriate to the enumeration entry represented. An example is provided below.

Figure 29 - COM Object Enumeration

Dragging an object that represents a class onto the COM object tree (bottom part of the right-hand pane) will add an object to the COM object tree that represents a global instance of that class (see figure 30). This new object will contain detailed object members for each of the class's properties, functions, and events. Dragging one of these object members onto the application workspace will create reference to that member. This is illustrated in figure 31 below. The COM object tree also contains objects for all COM forms and their controls.

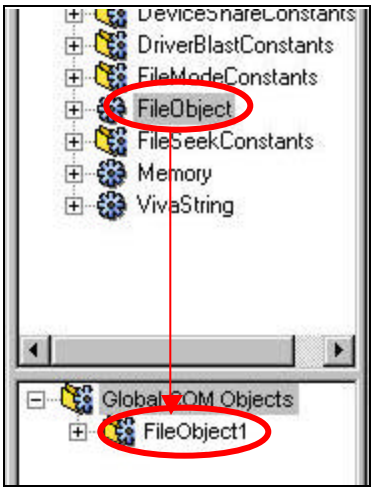


Figure 30 - COM Class Global Instance

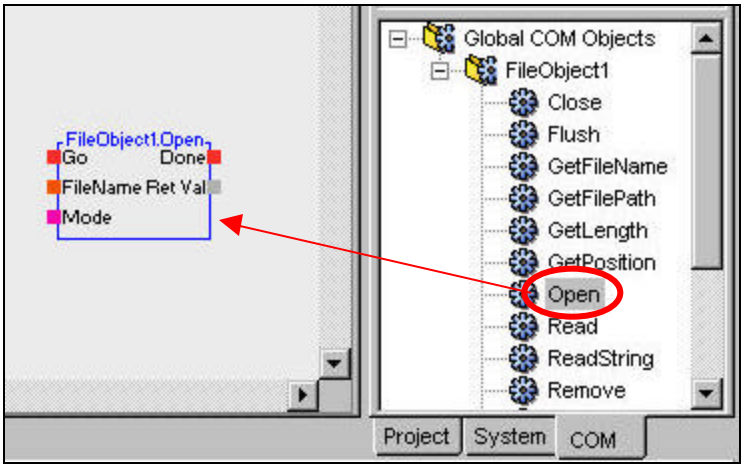


Figure 31 - COM Class Member Refernce

At runtime, except for events, these members can be accessed by sending a high signal to the 'Go' node of the member. The behavior of these members is listed in the following table.

Member	*Behavior*
Properties	The present value of the variable is sent out the output node named "Current". Any value sent to the node named "New" will then become the new value of the variable.
Functions	Each input node, other than Go, if any, serves as a parameter to the function. If the function has a return value, then it will be sent out the second output node before the Done node fires. If any node corresponding to a non-optional parameter is not hooked up, then an exception will be thrown when the function is invoked.
Events	Each output node other than the Done node is a parameter passed with the event. When a Com event fires in Viva, all events generated on attached transports are forced to be processed immediately. Know this to avoid infinite loops.

Table 3 - COM Class Member Behaviors

Once a member has completed its execution, the 'Done' node sends out a high state. Also, sending a low state to the 'Go' node will cause the member to send a low state from the 'Done' node.

3.3.1 X86 Versus FPGA Applications

There is an important difference between the execution of application models that target the x86 processor and those that run on FPGA's. On FPGA-targeted models, the 'Go' node should be in a high state for a single machine clock cycle (referred to as a "pulse"); the 'Done' node also generating a pulse.

Objects in the x86 system work very differently from those targeting an FPGA system. They require the 'Go' node to go high and stay high (referred to as a "sticky" state), until the system gets around to processing it. Likewise, the 'Done' node does not merely generate a pulse, but is set and left in a high state, until the 'Go' node is set to a low "sticky" state; the 'Done' node will then be set to a low "sticky" state when its events have been processed.

To bridge the gap between these two execution models, MetaLib provides two objects: OneShot and FAI2x86_GDBW. The OneShot object generates a pulse through its output node when it receives a high sticky state in its input node. The FAI2x86_GDBW object emits a "sticky" state from its 'Done' node when the 'Go' node receives a pulse.

An example of the usage of OneShot and FAI2x86_GDBW is shown in figure 32.

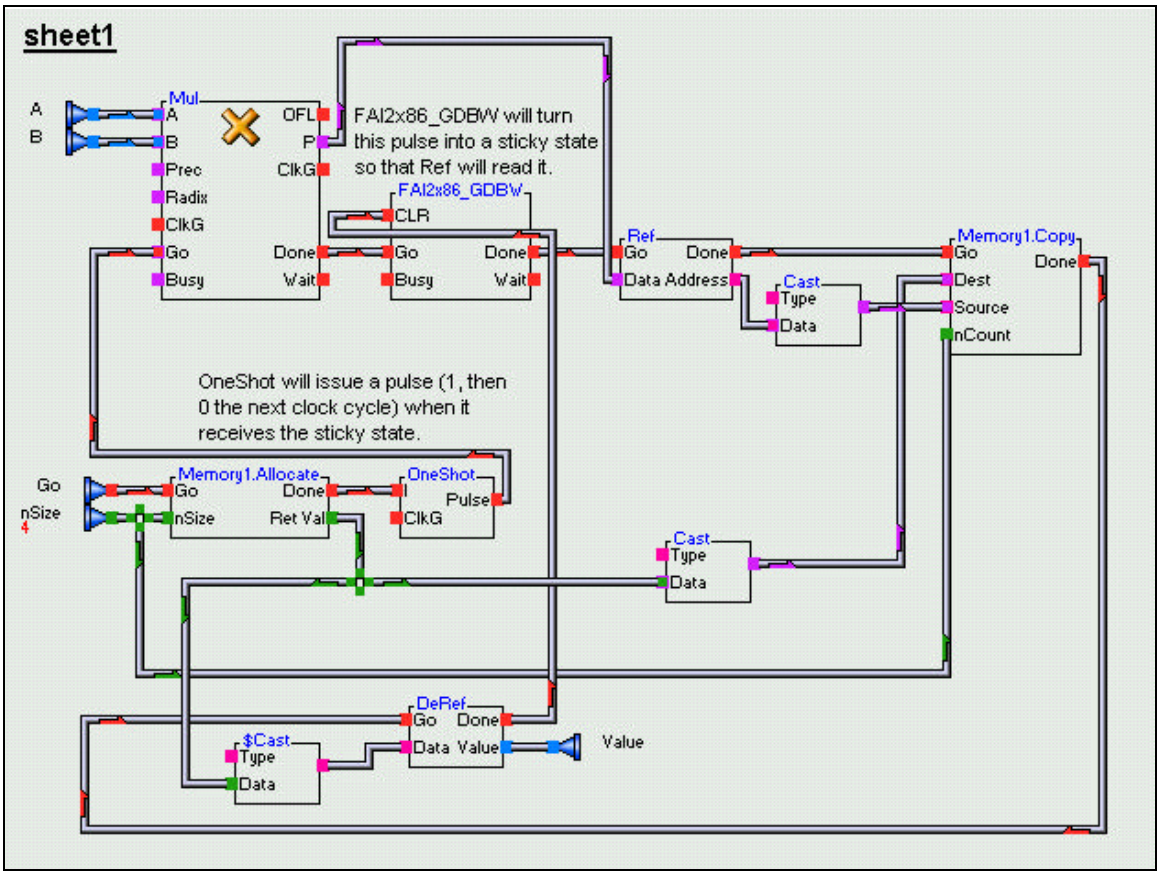


Figure 32 - x86/FPGA Application Example

3.4 Dynamic Com Objects

Dynamic COM members, which can be referenced by dragging an object representing a member from the COM class tree (upper part of the right-hand pane) rather than the COM object tree, function in the same manner as regular object members, except that they are not associated with a specific instance of a class. These objects will have an extra input node, 'Member Of', that is used to specify which object they apply to.

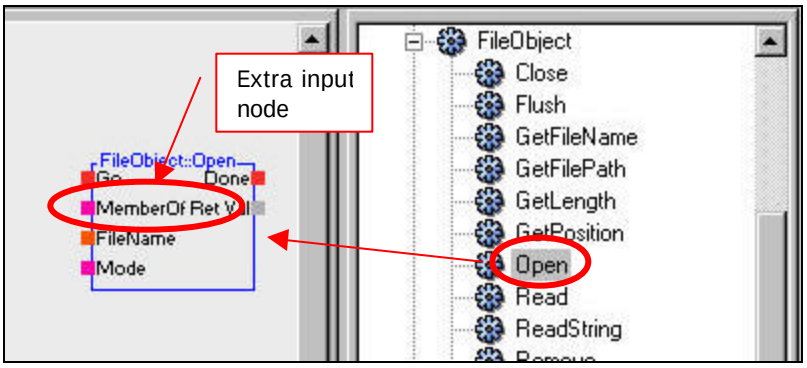


Figure 33 – Dynamic COM Class Object

The input to this type of node is a pointer to a class specific 'Dispatch' appropriate to the class that the object member belongs.

Dispatch – the interface by which the functionality of a COM object is invoked. In VIVA, it is used as an identifier for a COM class instance.

To acquire the dispatch pointer of an existing COM object, use a Dispatch object, which you can generate by dragging a node representing a COM object in the COM object tree onto the application editor. Once given a 'Go' signal, it will send the dispatch pointer out the node named 'Dispatch'.

You can also instantiate COM objects at runtime using a ComCreate object. Dragging a class object representing a COM class from the COM class tree onto the application editor can generate a ComCreate object. Note the name of the newly created object: "ComCreate: ", then the name Library ID for the library that the class resides in, then "::", then the class name. The following figure illustrates how the class object will appear in a save files.

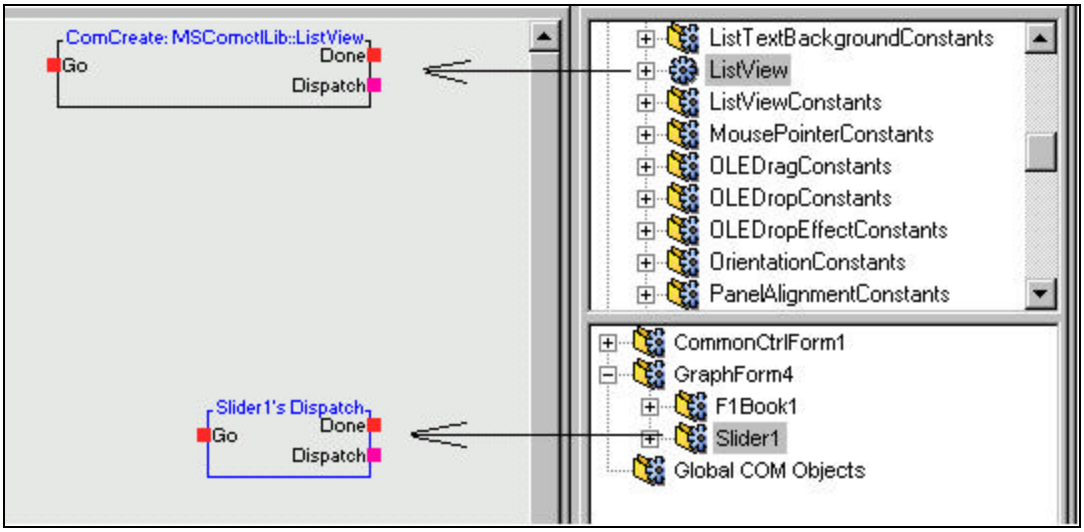


Figure 34 - COM Class Objects Example

COM objects created at runtime are not automatically released. They can outlive their calling program, and even be used in other processes. You must use a Release object to destroy them. This object is located in the Primitive Objects branch of the Project objects tree. Simply pass in the dispatch pointer and send it a 'Go' signal.

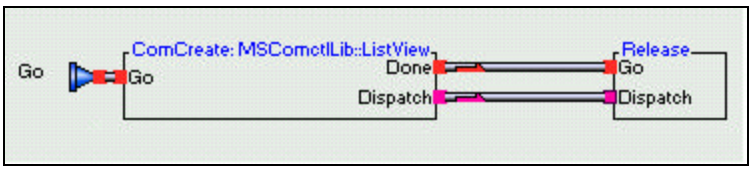


Figure 35 - COM Object Destruction

3.5 Object Browser

The Object Browser contains a list of existing type libraries on your system. These libraries also contain lists of the components and definitions included in each library.

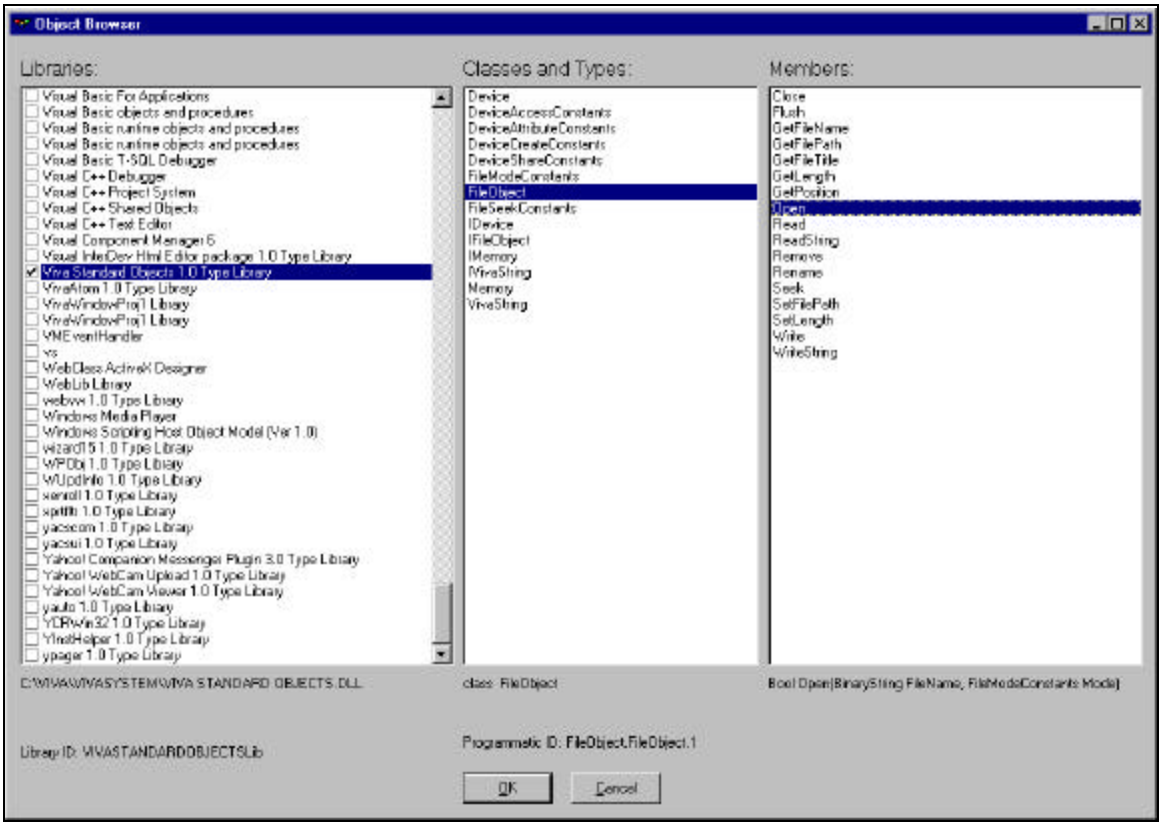


Figure 36 - Object Browser

The 'Libraries' list box (left-hand pane) contains a list of type libraries registered on your system. If you select one of these items, the list of its components and definitions appears in the 'Classes and Types' list box (middle pane). If you select one of these, the list of its members appears in the 'Members' list box (right-hand pane).

You can include a Library for use in VIVA by "checking" its entry in the libraries list box. Simply click on the check box next to the desired library in order to include it. The library can likewise be un-included by un-checking its entry.

You must click on the 'OK' button in order to apply any changes you make to the selection, or de-selection, of any type libraries.

The caption immediately below the libraries list box is the path and filename of the actual file that contains the Library while the below that is the Library ID, if it exists, for the library selected. It is used in VIVA save files, prefixed to a class name, to indicate that it is located in this library.

The caption immediately below the Classes and Types list box is simply the type of the selected entry, followed by its name while the caption below that is the Windows Programmatic ID (not to

be confused with VIVA Programmatic ID, used in save files), if it exists, for any class selected. If you are not already familiar with programmatic IDs, know that any class that does not have a Windows Programmatic ID cannot be directly instantiated via the COM interface.

3.6 COM Form Designer

The COM form designer is a utility that is used to create forms which host ActiveX Controls. This utility is available from the 'Tools' menu. The COM form designer 'File' menu contains Save, Save As, and Load commands; along with a recently used file history.

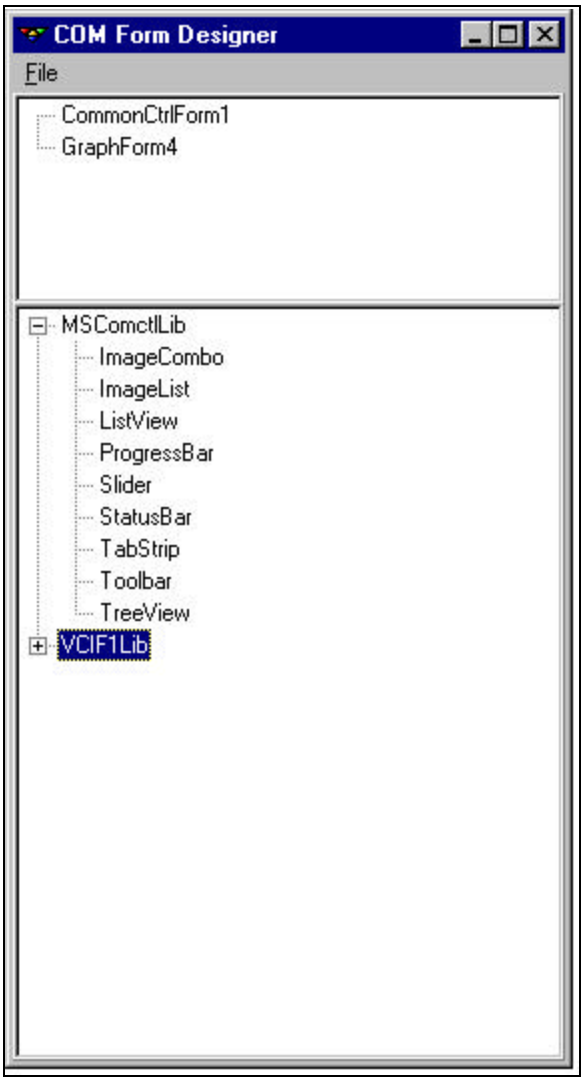
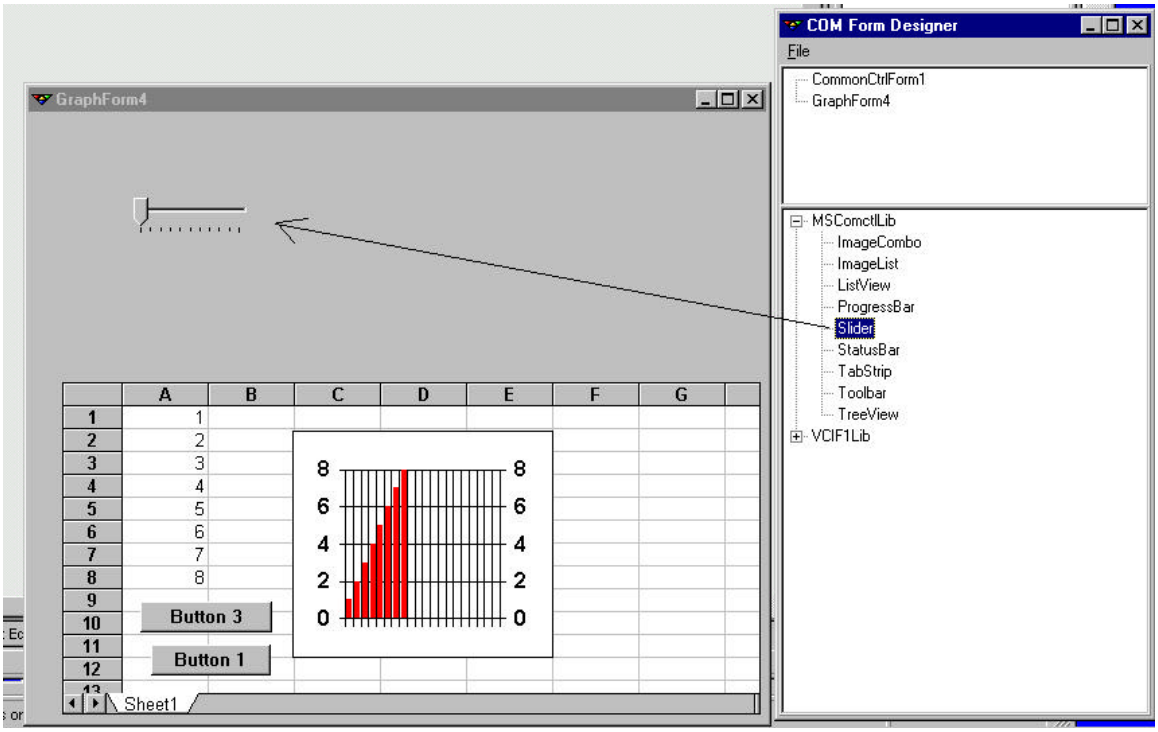


Figure 37 - COM Form Designer

The forms tree (top pane) contains a list of all forms in the current project. **By default, there are none**. Selecting the File/New menu command will create a new form. Clicking on a branch in this tree will display its corresponding form. Pressing the delete key will close the form corresponding to the selected branch.

The ActiveX controls tree (bottom pane) contains a list of available ActiveX controls for the libraries that have been selected for use in this project using the object browser. Dragging one of these branches onto an ActiveX form will instantiate that control at the location specified. The control can then be resized or deleted. To delete the control, simply select it by clicking on it the pressing the delete key.



Right-clicking on a control will bring up an available actions list for the selected control. This list may or may not contain entries.

When a VIVA file is saved, any forms that are dependent on that file are also saved, even if they did not originate from the current project.

All forms and controls are listed in and can be utilized using the COM object tree.

SECTION 4 - DATA SETS

The purpose of every VIVA application is to manipulate data. It may operate on dynamic input or static information defined within the application itself, or both. In all cases, an application generates output data. Nearly every VIVA object has data input and output nodes. The ability to create and utilize highly structured data types in a consistent, yet flexible fashion is central to VIVA.

4.1 Consistent Structure and Design

Data Sets are used to define the data types of input and output nodes on an object. As a general rule, input and outputs connected together must use identical Data Sets. Fundamental Data Sets are the building blocks of all other Data Sets. These Data Sets are always included in a VIVA 2.0 project. VIVA provides the following Fundamental Data Sets.

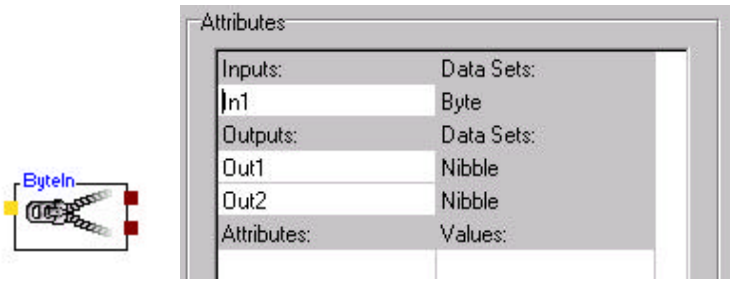
Data Set	*Definition*
Bit	The atomic data set and represents an 'On'/'Off', 0/1, or True/False
Variant	The Variant Data Set is used to represent data when the actual Data Set is unknown at the time of design & development. This core feature enables VIVA to formulate generalized polymorphic objects that work independently of the input data types. The VariantOut will aggregate two arbitrary Data Sets into a larger predefined Data Set. The resolution of a Variant Data Set exposer into the appropriate Data Set exposer will be performed when the input Data Sets become known.
List	A special data set consisting of two Variants. The data type does not get elaborated until compile time, when it is resolved into known Data Sets. More information on List Data Sets is given later in this section.

4.2 Exposers and Collectors

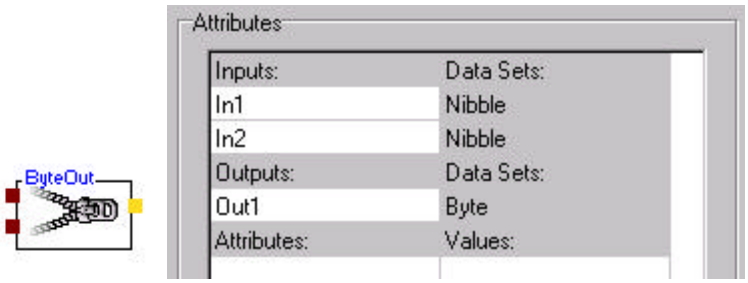
Data Set Collectors and Exposers are used to compose and decompose Data Sets. Exposers take a higher order Data Set as input to expose its lower order constituent Data Sets as output. Conversely, Collectors get multiple Data Sets as input and compose a higher-level Data Set that get exposed as output. For example, in VIVA 2.0, a byte is defined as 2 nibbles. Therefore, a byte exposer (shown below) accepts a byte as input and exposes two nibbles whereas a byte collector accepts 2 nibbles as input and composes a byte output.

A byte exposer and collector are illustrated below.

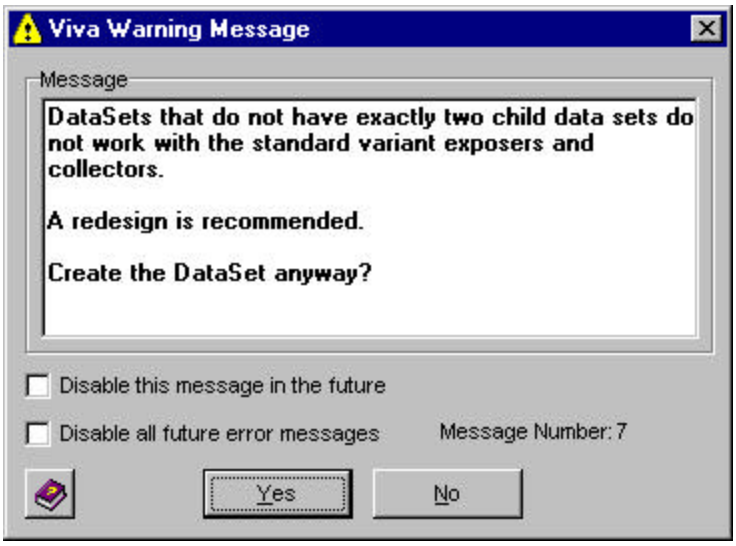
Byte Exposer:



Byte Collector:



A built-in feature of VIVA is the automatic creation of data set exposers and collectors. When working in the Data Set Editor, if you save a data set without exactly two components (or child data sets), the following warning displays.



If your intent was to create a Data Set with # children !=2, click 'Yes', otherwise click 'No' to return to the Data Set Editor and alter your Data Set design.

4.3 Bit Patterns (MSB, LSB, and BIN Data Sets)

The VariantIn Data Set, a primitive object, functions as an exposer. VariantIn allows you to input one data set; VariantIn then splits the data set into two or more Data Sets that contain a bitlength sum equal to that of the input Data Set.

VariantIn, used in conjunction with the MSB (most significant bit), LSB (least significant bit), or BIN (binary tree) objects, determines the type of exposed Data Set. That is, if you run a Data Set through the MSB object and then VariantIn, you will get the top bit out the top of the splitter, and the rest of the bits out the bottom port, which is what the packaged ExposeMSB(2,1) object does. LSB supplies the opposite. BIN supplies an even number out both the top and the bottom, or, in the case of an odd bitlength, one more out the bottom.

MSB, LSB, and BIN objects are located in the GrammaticalOps tree group.

Patterns are also applicable to lists. In the ExposeCollect tree group, you will find pack/unpack objects that do functions to expose and collect items in lists. For example, if you run a list through UnpackBIN you will get two lists out, each containing half the items.

4.4 Custom Data Sets and The Data Set Editor

If data sets are required that do not exist in an available library, additional data sets can be defined using the Data Set Editor.

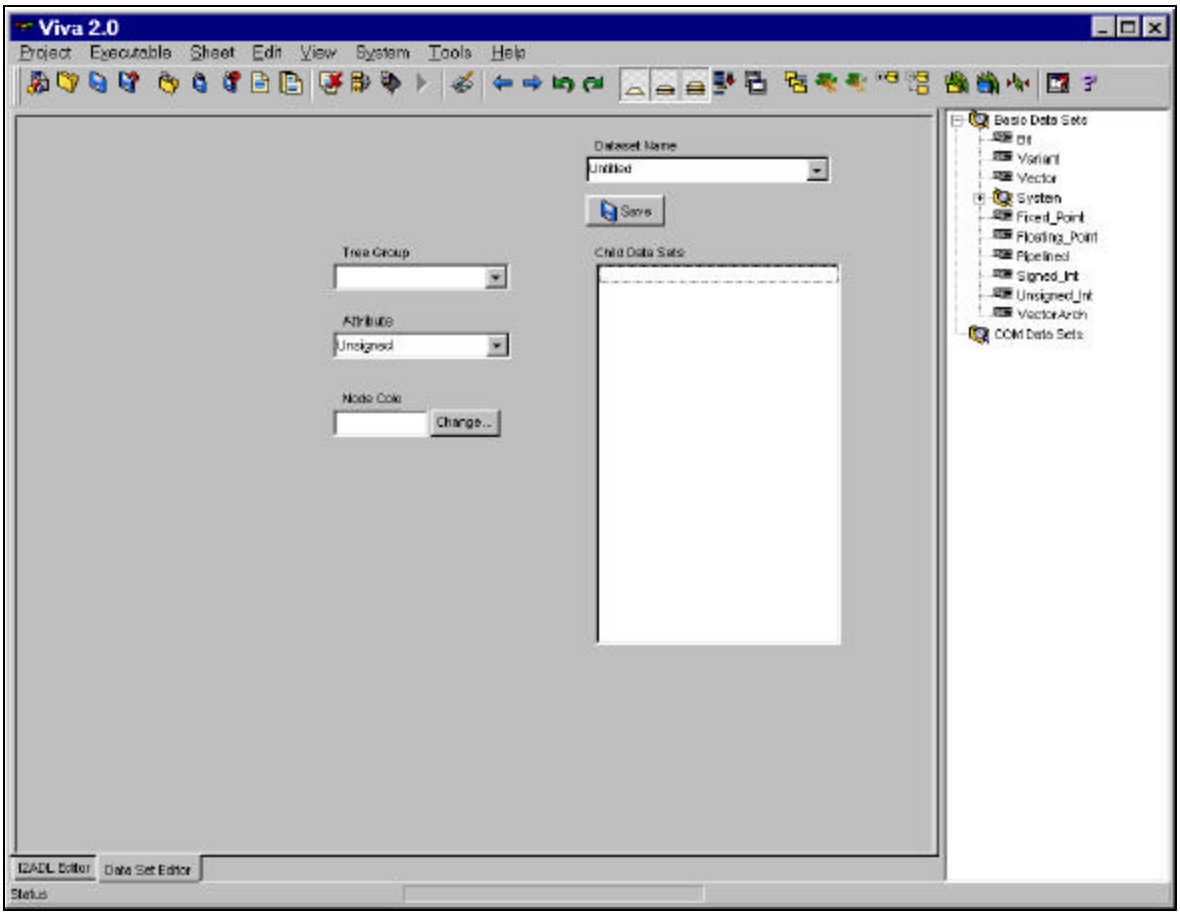


Figure 38 - Data Set Editor

'Pressing the Data Set Editor' tab at the bottom of the main VIVA window activates the Data Set Editor.

Data Set Tree

The Data Set Tree displays a hierarchical view of the data sets defined within the current project. It is located on the right side of the Data Set Editor. Some of the operations that are available for use with the data set tree are:

- Double-clicking a Data Set in the tree loads it into the Editor.
- To delete a data set, simply highlight it within the data set tree (by pointing to it and left clicking) and press the delete key on your keyboard.
- Right-clicking a data set in the tree and selecting the 'Add to Child Data Set List' option, adds it to the element list of the currently loaded data set.
- You may also drag and drop a Data Set from the Data Set tree into the element list of the current Data Set.

Dataset Name

Displays the name of the data set currently loaded into the editor. This text box is also used to name a new or rename an existing data set. The exposer and collector created for a data set will

share the data set's name, suffixed by the words "In" and "Out". For example, a data set called 'MyData', VIVA will create the exposer 'MyDataIn' and the collector 'MyDataOut'.

Elements
The Elements list displays all of the components of the current data set. Elements may be added to the list by dragging them from the Data Set Tree and dropping them in the Elements list. Data sets may also be added by double-clicking them in the Data Set Tree. Elements in the list may be removed by left-clicking them in the Elements list and pressing the delete key on your keyboard.

Attributes
Attributes are used to specify how numeric data sets will be structured. When creating custom data sets, you may choose from the following attributes: Unsigned, Signed, Fixed Point, Floating Point, or Complex.

Assigning an attribute to your data set is optional.

Node Colors
Use this control to select the node colors applied to the objects in your data set. To change the color of a data set, click the 'Change' button and select or define a new color with the standard Color dialog.

Assigning a color to your data set is optional.

IMPORTANT: After completing the definition for a new data set, press the 'Save' button!

4.5 Further Discussion on List

Dynamic data sets are employed when the user wishes to treat an aggregate of data as a single data set. For example, a screen coordinate may consist of a pair of integers and, while VIVA would allow the user to define a special data set named COORDINATE consisting of a pair of integers, it can be cumbersome to have to define all sorts of special data sets. In the more general case, you may wish to treat a vector of real numbers, and may not know the exact length.

VIVA circumvents this difficulty through the use of List. A list consists of the congregation of any two data sets. Under the Element list for the List Data Set, you will find that a list consists of two variant types. Because the input types are Variant, a list can consist of the congregation of two Lists.

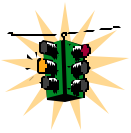
To perform a polymorphic interpretation of a Vector, VIVA provides the data set exposer and collector for the List Data Set. They can be found in the data set exposer sub-tree; their names are 'ListIn' and 'ListOut'.

4.6 Casting Data

In some cases, data set collectors may not be unique. For example, a 16-bit signed integer and a 16-bit cardinal (unsigned integer) are both composed from a pair of Bytes. The collector is ambiguous. In this case, VIVA will select the more fundamental exposer. To compose two Bytes into an Int type requires that either IntOut be explicitly specified, or that the output of the Variant Collector be cast as an integer using the generic system cast operator.

The system cast operator, Cast, is a special operator designed to convert between various data set types. In all cases, the data, beginning at the lowest bit of the source, is transferred to the destination with excess bits being thrown away. If there are insufficient bits to fill the destination, the upper bits in the destination are filled with zero.

One standard and necessary use for the system Cast operator is to force the data set at the output of objects. In many instances, a VariantOut composes the final result. VIVA attempts to resolve the output using the available Data Set Collectors. In some instances, the Data Set Collector may not be unique. As we saw above, a 16-bit signed integer and a 16-bit Cardinal (unsigned integer) are both composed from a pair of Bytes and their respective collectors are structurally ambiguous. To compose two Bytes into an Int requires that either the output be cast into the correct data set through the Cast operator.



For cardinal numbers, Cast provides a correct conversion. Use of Cast with integers, fixed point, or more complex types can result in unintended behavior. Cast is intended to be a bit transfer operator, not a conversion operator**. For numeric conversions, the user should use a convert operator**.

4.7 Data Set Recursion

Earlier, it was stated that most data sets were composed of two matching data sets.

Here is an example without Data Set recursion:

Assume some object, X, that operates on a single Byte input and produces a single, Byte output. We design, develop and test our operator with Byte input and output. If support for Word data types became a requirement of the operator, we would have to create a new object, probably using the X as a template, which supports Word data types. Alternatively, we could modify X to support both Byte and Word data types. Either way, design gets revisited, code gets edited and a new QA cycle is probably in order.

Here is the same scenario using symmetrical data sets:

Once the Byte operator is created, we create a Variant form of the operator. The footprint of the Variant operator is the same as the byte operator, however it's input and output nodes are Variant. This Variant object is defined as two of itself with the input being divided between the two. When VIVA compiles your application, it asks what Data Sets its top-level inputs and outputs should be. Suppose you chose Byte. VIVA would synthesize only the Byte operator. If you choose Word, VIVA will enter a Recursive loop in which it applies the provided Data Set

(Word) to the Variant form of the operator. Since there is no form of X that explicitly supports Word Data Sets, VIVA chooses to recurse on the Variant form of X with a Word Data Set as input. The Variant X operator splits the data in half, one Byte each and presents these inputs to two instances of itself. VIVA now searches for a form of X that operates on Byte Data Sets. This form of X is defined and VIVA synthesizes two instances of the Byte operator. The outputs of the two Byte operators are aggregated by Variant collectors and exposed as the originally defined Data Set of Word.

Further discussion on this powerful feature can be found in the sections titled 'Variant Select and Data Set Polymorphism' and 'Selective Synthesis' and 'Data Set Polymorphism'.

Advanced use of Data Set Polymorphism can be found in the sections on 'Multi-Dimensional Recursion and Behavioral Techniques'.

SECTION 5 - GO DONE BUSY WAIT

Go Done Busy Wait (GDBW) is the primary state control abstraction in Viva. GDBW allows you to create a distributed one-hot state machine throughout any design. GDBW also permits complex synchronizing between objects with the ability to “stall the pipe” at any point without losing data or commands. More sophisticated techniques allow for synthesizing recursive state machines with very simple parameterization.

GDBW Basics

It is important to note the node reversal inherent to GDBW. Wait is an input, and Busy is an ouput; node placement might imply the opposite. Go->Done and Wait<-Busy defines a complete control handshake between two objects. This handshake defines a protocol both objects must follow.

Go – Input

Go tells an object that data presented to it is valid and the object should begin processing data. Generally, Go must be a single clock cycle pulse because most objects cannot accept data every clock cycle, and their Go instantiates an initialization state. Fully pipelined objects are an exception to this rule. Go is usually obtained from an upstream object's Done. If you provide a Go signal to an object from the widget interface, you will generally want to filter that signal through a OneShot object to ensure the object receives a single pulse.

Done – Output

The Done signal indicates that data an object is presenting is valid. Done is generated as a single clock cycle pulse to conform with the expectations of a downstream object's Go.

Busy – Output

Busy is an object's means of communicating that it is not currently able to accept new data and a Go. Busy will propagate from a downstream object to an upstream object's Wait. Busy tells an upstream object that it is responsible for maintaining the data that it is currently providing. Busy will go high on the same clock cycle that a Go is received and should remain high through the clock cycle that the object in question produces a Done.

Wait – Input

Wait is generally received from a downstream object's Busy output. When an object's Wait input is high, that object is responsible for maintaining the state of all data that it is providing. It is also responsible for not generating another Done signal, as that signal will most likely be ignored by the downstream object.

Pipeline Object

The Pipeline object is generally placed between two synchronous objects to buffer data and to arbitrate GDBW. Since the Pipeline buffers data, it allows two sequential synchronous objects to iterate on two different sets of data concurrently, while still maintaining their GDBW contract. Some objects have Pipelines embedded in their front ends to automatically provide this service. The top-level arithmetic operators are examples of objects that have embedded Pipelines.

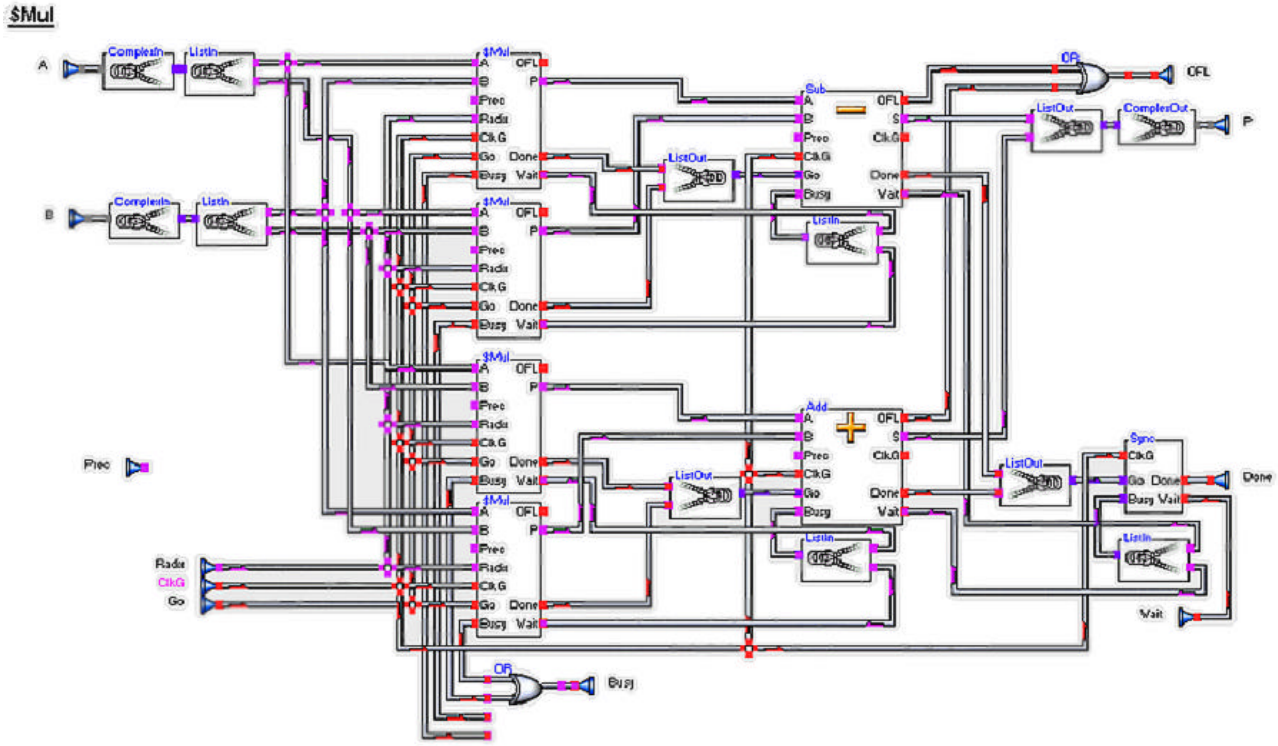
Syncing

As is common in hardware systems, most submodules do not complete on the same clock cycle. Therefore, if two modules run parallel (A and B) so that both output to a common third module (C), you must ensure A and B complete before attempting C. Do not AND the Done signals from A and B since they have little chance of happening on the same cycle; in that situation, the AND will block all Done signals. Instead, use the Sync object.

The Sync object will take as an input a list of Done signals and produce from those a single Done pulse when all the Done signals included in the list have fired. It returns a list of Busy signals formatted to the same list structure of Done signals coming in. You can use the ListOut objects to combine your signals for the Done and the ListIn objects to split the Busy signals.

Syncs are built into what is called a SyncRegister, and SyncRegisters are built into most top-level operators such as Add and Mul. Hence the variant Done and Busy nodes on most top-level objects.

The $Mul(Complex/Var) provides a good example of Synchronizing objects.



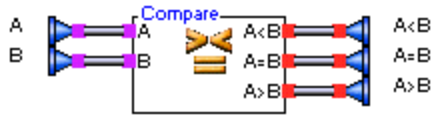
Note the following:

- The Busy signals coming off the first group of parallel objects are OR'd together and before output. That lets objects upstream know that at least one $mul object is busy; therefore, we don't want their data at the moment. It's tempting to pack those into a list but we've only got one Go signal; there is no value in individual Busy signals here.
- Remember the Add and Sub have Sync objects built into them, which is why we can plug lists into the Go signals on them.

- Notice the Busy signals are split the same way the Done signals are combined.
- We do see the Sync object at the end. This forces both the Add and Sub to have completed before the data at P is ready. The incoming wait is plugged in there as well, forcing the Add and Sub to output the same data when high and forcing the whole object to propagate a stall.

SECTION 6 - SELECTIVE SYNTHESIS

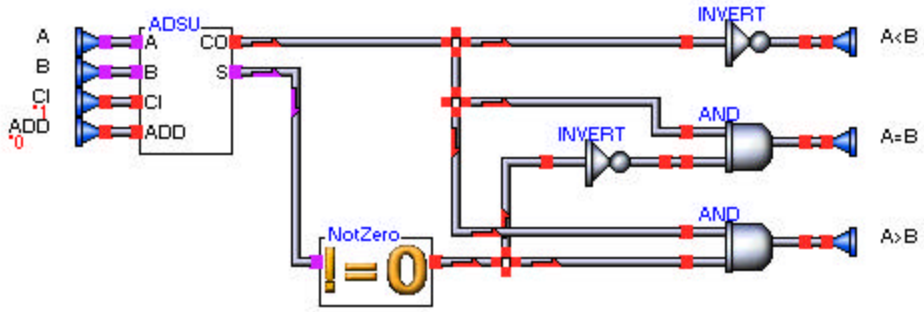
Selective Synthesis is a tool that allows you to steer the VIVA synthesizer to expand certain objects and remove others. Essentially, it is a synthesize-time case statement that allows you to choose between different algorithm topologies. As an example let's consider the Compare object from CoreLib's Data Info tree group.



The inputs are Variant, and this object is capable of performing a compare between unsigned integers, signed integers, fixed point and floating point numbers. The only constraint is that A and B carry the same data set. The algorithm for comparing numbers is not universal across data sets. Let's take a quick look at the algorithms.

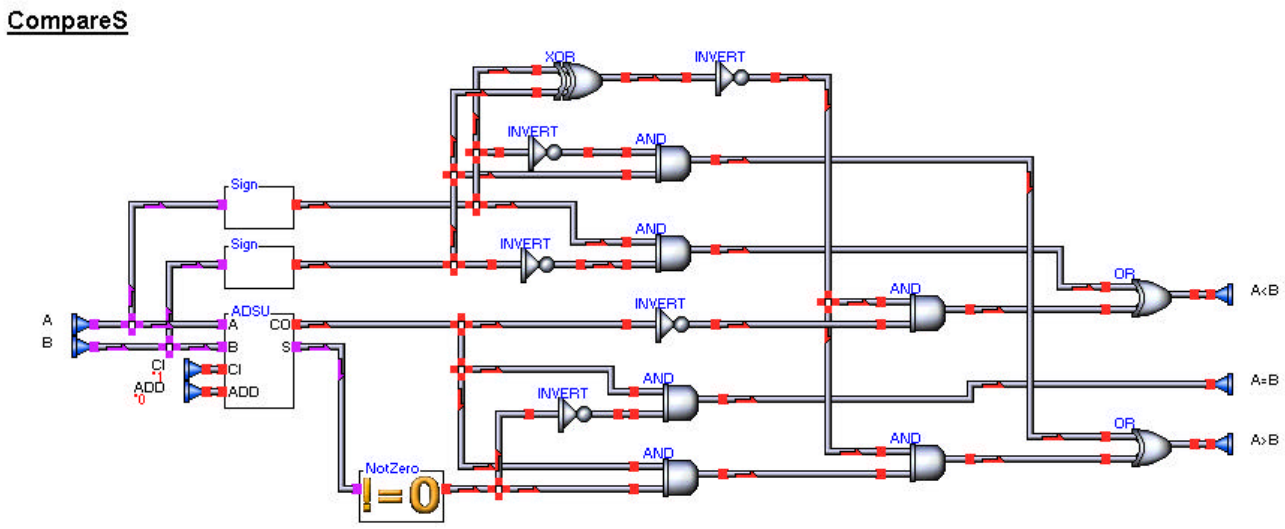
Unsigned Integer Compare:

CompareU



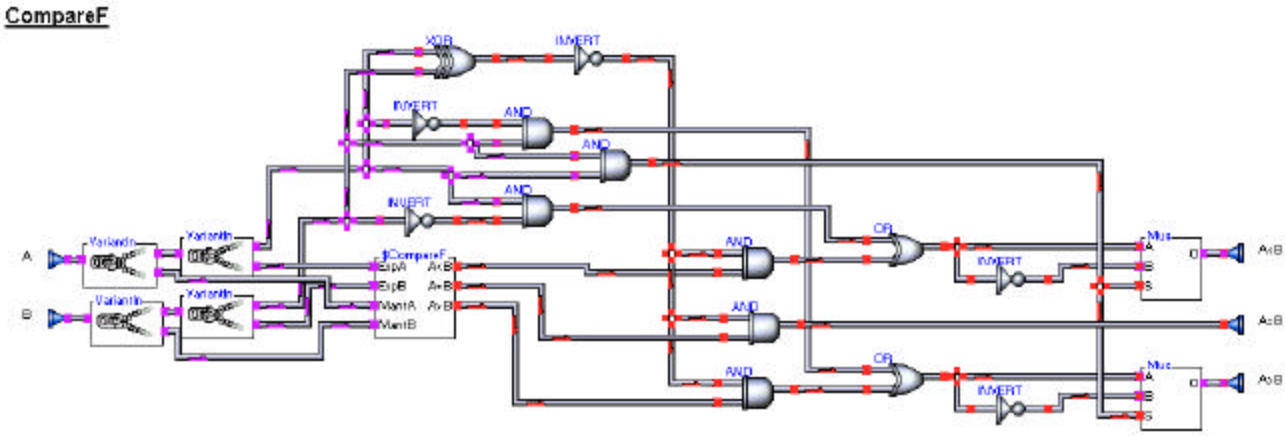
The unsigned integer compare simply performs a subtraction of B from A. On the ADSU object, the ADD input chooses between addition and subtraction. In this case (ADD = 0) we are subtracting. When subtracting, our carry input and output must be considered. CI = 1 signifies that we do not have a borrow. Consequently, if CO = 0, B is less than A, because we would be propagating a borrow.

Signed and Fixed Point Compare:



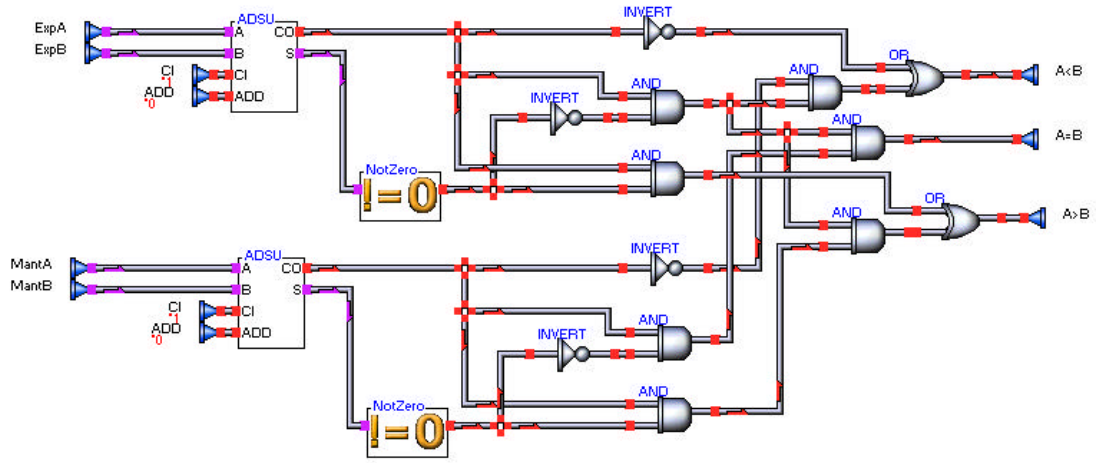
Here again we perform a subtraction of B from A. However, we have additional logic that handles the signs of the number so that a negative number is less than a positive, regardless of magnitude.

Floating Point Compare:



This object contains logic to handle the sign bits in much the same way as the preceding object. The comparison of the exponents and mantissas are handled in a subordinate object, $CompareF. In this object, we see our basic unsigned compare used for the exponents and mantissas.

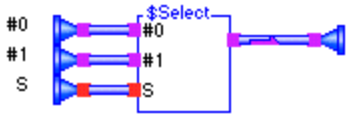
$CompareF



We have now seen three distinct algorithms for comparing numbers. Their behaviors have been wrapped up into three objects, CompareU, CompareS and CompareF. What we want to do now is combine them all into a single Compare object. We will use selective synthesis to do this.

Compare

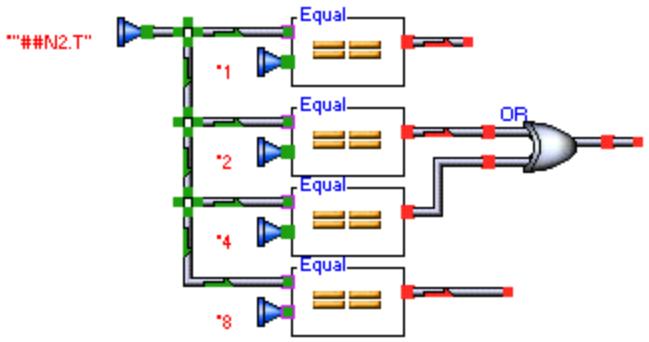
The Compare object contains all three of the data type-dependant compare objects. Selective Synthesis is instantiated to choose between them based on the data type coming in. To accomplish this we sandwich our objects in between $Select objects (Primitive Objects Tree Group) and drive the select signal off of a data type parameter. Lexically, $Select is a special multiplexor whose S input must be a constant 0 or 1 at compile time.



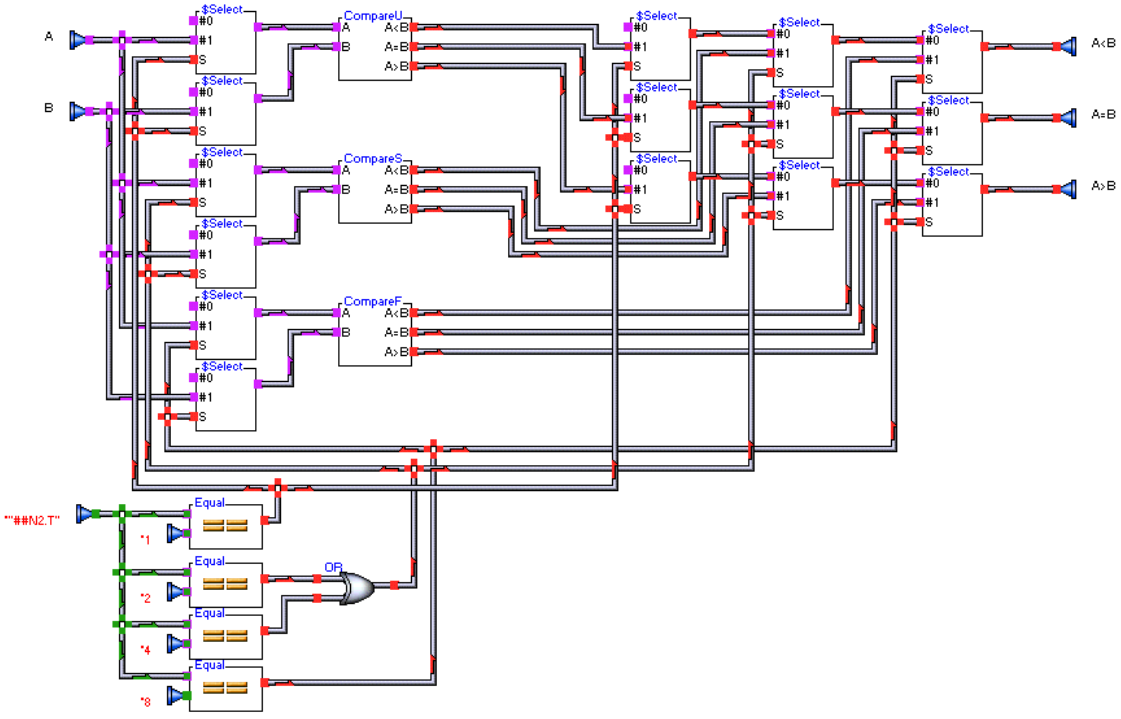
The inputs #0 and #1 will many times have different data sets driving them. The $Select object allows you to propogate one and only one of the inputs. In the compare object we are driving the S inputs of the $Select objects off a special constant, "##N2.T." This is a macro that returns a DWORD describing the data type of the second input node of the compare object. Go and Wait are counted as nodes 0 and 1 respectively. The other nodes are then sequentially numbered from the top down. In our example N2 is the A input. The "T" in the Viva Macro refers to the type attribute of the data set. We have in effect the following psuedo-code case statement:

```
Switch( ##N2.T ) {
        Case "1" : Unsigned Int
        Case "2" : Signed Int
        Case "4" : Fixed Point
        Case "8" : Floating Point
}
```


In the compare object we will synthesize the same algorithm for signed int and fixed point, so cases 2 and 4 are OR'ed together.



Compare



In order to synthesize one of the internal Compare objects and not the others, we must sandwich each data type-dependant Compare object in between $Select objects. On the left-hand side we have three pairs of $Select objects. Each pair has a unique S, and only one of the three S signals can be 1. If we were synthesizing an unsigned int compare, A and B would pass through the top

two $Selects into the CompareU object. The other two sets of $Selects will have S = 0, and will therefore propagate no data to their respective objects.

To complete the selective synthesis, the outputs of each object must also pass through $Selects. On the right hand side we have three sets of three $Selects. The first (from left to right) will pass the unsigned int outputs through if ##N2.T = 1. It will not pass anything through otherwise. For the unsigned int case, the last two columns of $Selects will simply pass the unsigned int outputs through. The second column passes through signed int and fixed point, and the last column passes through floating point.

We now have an object that will automatically choose between three unique algorithms based upon the type of data it receives.

SECTION 7 - VARIANT SELECT, RECURSIVE FOOTPRINTS, AND DATA SET POLYMORPHISM

Variant Select is the backbone of VIVA's recursive synthesis. It also forms the basis for data set polymorphism and the resolution of high-level objects into primitive objects. Unlike many other graphical programming interfaces, where objects represent cores that are linked together by the tool, every object in Viva is formally and recursively synthesized from input data sets down to primitive objects. These primitive objects could be from the Primitive Objects Tree (AND, OR, INVERT, etc.) or they could be primitive objects found in a system library.

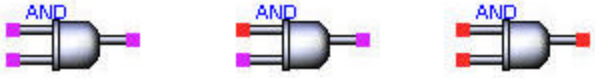
To understand the process represented by Variant Select, we must first understand object footprints. An object's footprint is a combination of the name of the object and the number of inputs and outputs it contains. In the following diagram, the two AND gates have the same footprint, even though the object on the left has two Variant inputs and one Variant output, and the other has all Bit inputs and ouputs.

AND1.jpg

The following two AND gates do not have the same footprint, even though they both have Variant inputs and outputs in common. The object on the left has two inputs and the object on the right has three inputs.

On the next set of two objects, we have differing data sets and input/output node names. These objects still have the same footprint however. What is important is only the name of the object (case-sensitive), and the number of inputs and outputs.

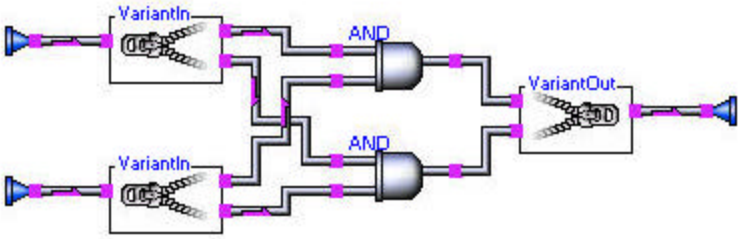
When writing a Viva program, one will usually use the most variant version of an object. Of the three AND gates that follow (with identical footprints), the one on the left is the most variant.



The recursive synthesis process takes this most variant object and recurses around the data sets that are input to the object until all that is left is a set of primitive objects. This is possible because at each step of the recursive process Viva chooses the least-variant version of an object.

7.1 Recursive Footprints

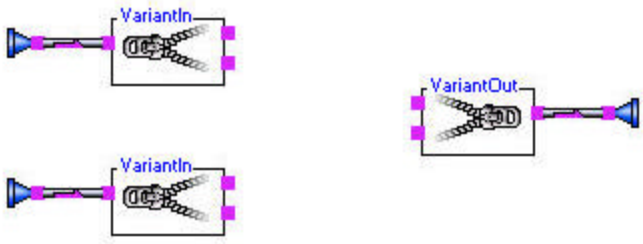
Let's first look at the recursive footprint contained in our most variant AND.



We can see that the inputs are both piped into Variant Exposers that will split the incoming data sets into their two children data sets. We also see two Variant AND objects that take the high-order and low-order children data sets respectively. These AND objects are self-references. If we drill into them, we will see the same object. Coming out of these self-referential AND objects, we collect the high-order and low-order data sets back into the original data set that was input to the object.

Let's take a moment to see how we build these self-referential recursive footprints.

1. Open a fresh copy of Viva and do not load any systems or library files. All we have at this point are the objects in the Primitive objects group.
2. Drag two inputs and one output onto the I2ADL sheet. Attach a VariantIN to each of the inputs and a VariantOut to the ouptut. Your sheet should now look like the following diagram:



i.

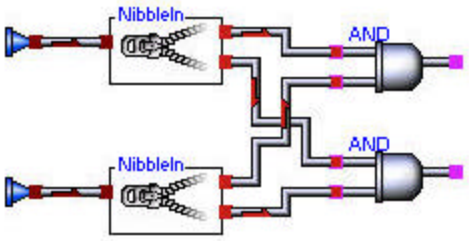
3. We now want to wrap this behavior sheet up as an object. You can do this by selecting "Convert Sheet to Object" from the sheet menu, or pressing the 'Convert Sheet to Object Button, or by pressing F8.
4. We need to give our object a name. In the Object Name text box enter AND. Click OK or press enter. You should see your AND object in the Composite Objects Tree Group.
5. Double-click on the object.
6. Now drag two copies of the same object onto the sheet (this is the self-referencing part).
7. Route the high-order output nodes of the VariantIns to the top AND and the low-order to the bottom AND. Also pipe the outputs of the ANDs to the VariantOut. Your behavior sheet should now look like Fprint1.jpg:
8. Repeat step #3 to update the object. Make sure the Update Original Object radio button is selected. This is the default.

We have now created a Variant AND object, that when coupled with the bit-level AND from Primitive Objects will handle any input data sets, as long as the two inputs carry the same data set.

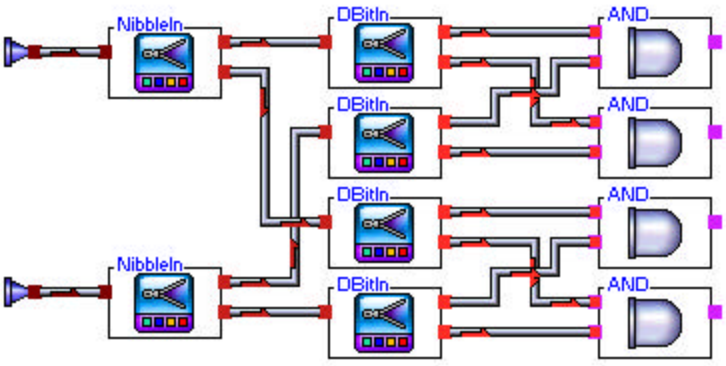
Let's take a look at how this is done. Keep in mind that we currently have two AND objects with the same footprint.

1. Starting with a new behavior sheet let's drag our Variant AND object out and attach inputs and outputs to it.
2. Now let's assign the data set of both inputs to be nibbles.
3. When we hit play, our behavior will be recursively expanded until all four bits that make up either of the nibbles go to four separate Primitive ANDs. Let's look at each step of the recursion.

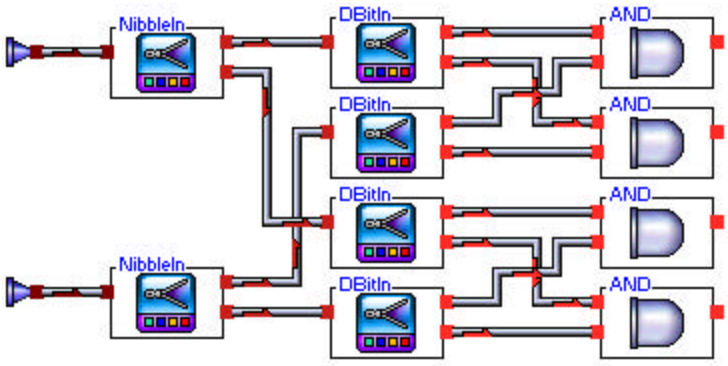
In the first step, our AND gate receives two nibbles which go directly to the VariantIns. These VariantIns will be replaced by NibbleIns by Variant Select. So far our recursion looks like this:



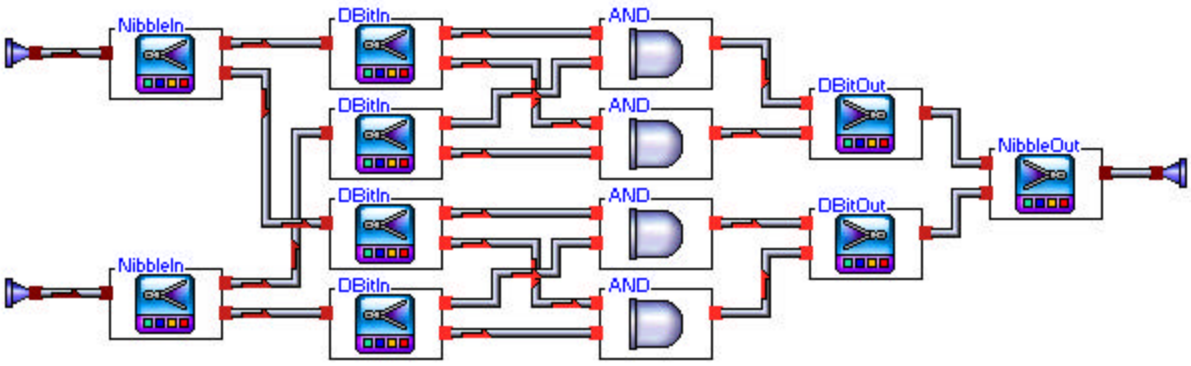
Our nibbles have been broken into DBits, and now we have two ANDs with DBit inputs. Our Variant AND is still the least variant AND. In this step we replace the VariantIns with DBitIns.



Now we have four AND objects with matched Bit inputs. The variant AND is no longer our least Variant, and these four ANDs will be immediately replaced with bit-wise ANDs from Primitive Objects. The Primitive ANDs act as our *Base Case* and terminate the recursion.



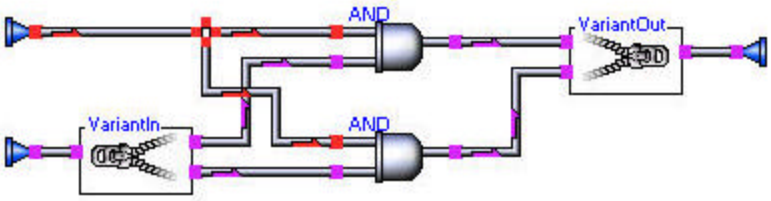
All that is left to do now is 'return' from our recursive calls and rebuild our incoming Data Set. The final expansion of the objects looks like this:



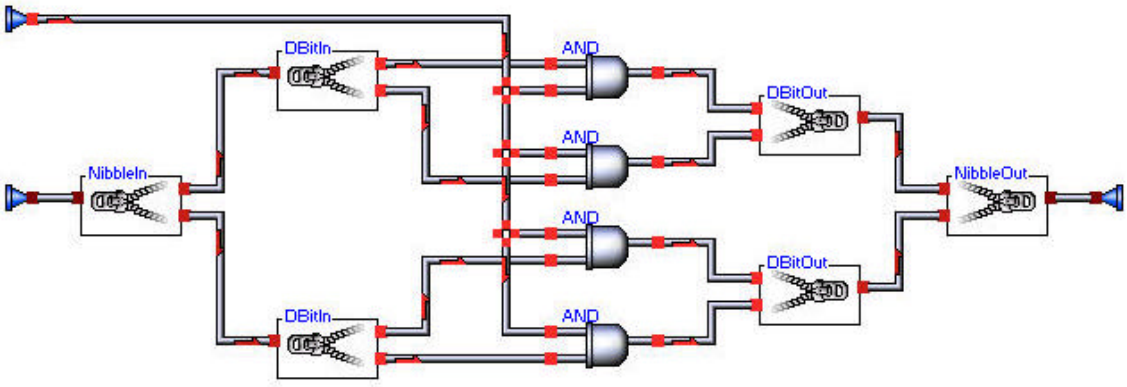
These two objects are all that is needed to cover AND of any two matching data sets. This process will work identically for any data set as long as the two incoming data sets match.

Now let's consider the case where our incoming data sets do not match. Let's say we want to have one input to our AND be a Bit, and the other be an arbitrary Data Set (Variant). This will give us the ability to gate the data on the Variant input. We will need to build one more version of our AND gate to accomplish this. Let's say that the top input will be the bit and the bottom will be the Variant.

Let's drill into our Variant AND. Delete the top Variant input and the corresponding VariantIn. Replace with a Bit input as follows:

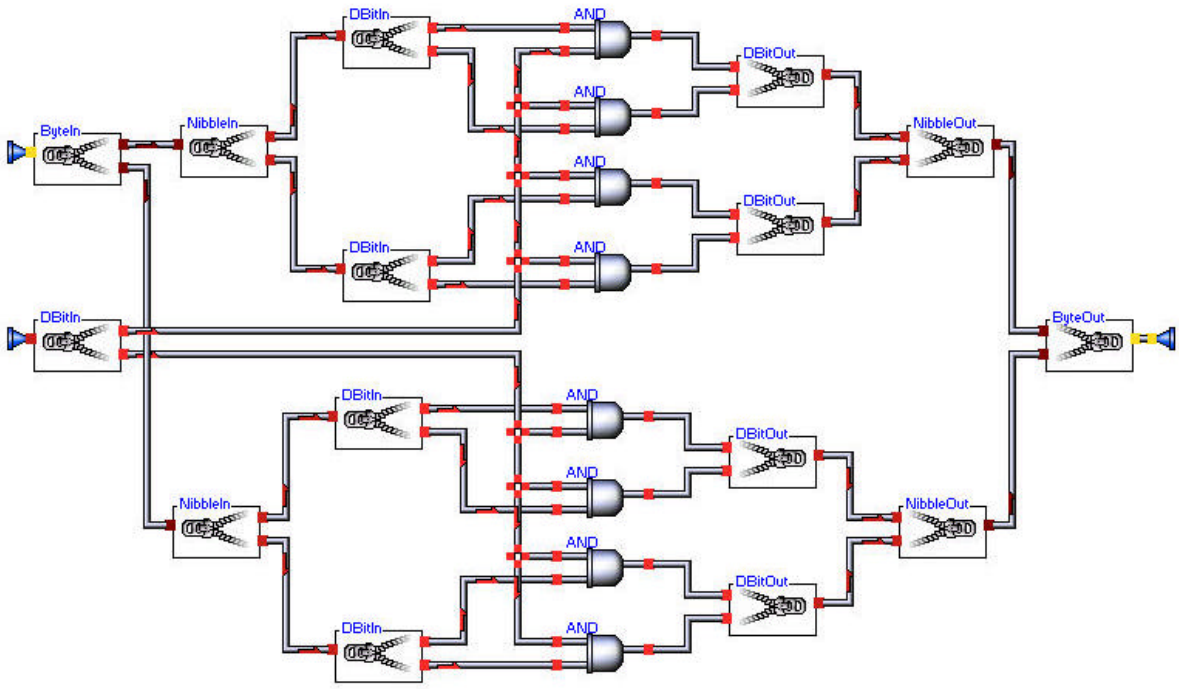


Now when we wrap this behavior up as an object we want to select the *Create New Object* radio button so that the behavior of our original object is not overwritten. This will create a third AND object with an interchangeable footprint. In this object we only recurse on the bottom input, and simply propagate the top input through the recursion. If we were to place a Bit input and a Nibble input on our new AND, the following structure would be synthesized:

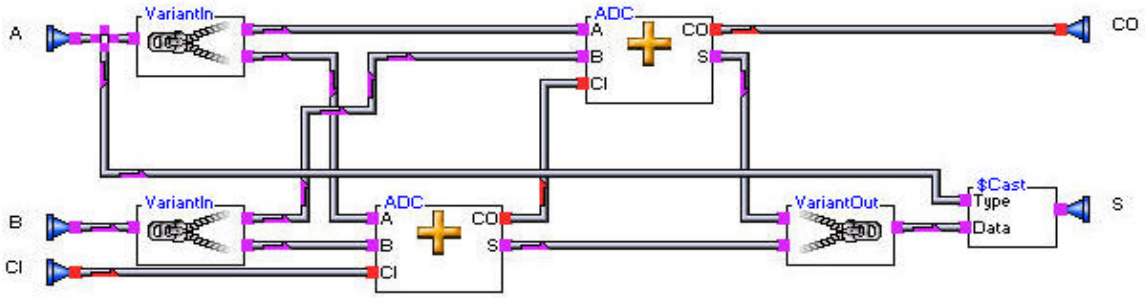


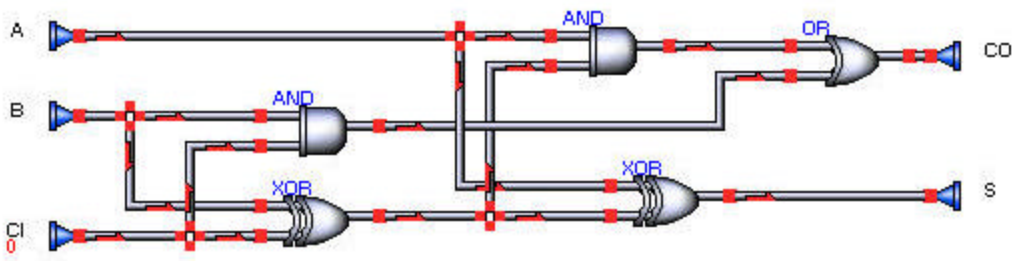
Of course, when we are programming it would always be preferable to use the most variant (two variant inputs) version of our AND. If used in the preceding case, the most variant AND would immediately be replaced in the first stage of recursion. If we make another AND object with the Bit and Variant inputs reversed, we may use the most variant AND object and now we do not have to remember whether the Bit goes on the top or the bottom. Let's assume we've switched the inputs and created a fourth AND object, so now we have the following ANDs that share the same footprint:

If we use the most variant AND, we no longer have to worry about the inputs carrying the same data set. Let's suppose we placed a Byte and a DBit on the inputs of our most variant AND. We would generate the following structure:



Usually when we are creating data set polymorphic objects we program a variant version and a bit-level resolution version. In our example, the recursion terminating leaf-node is a Primitive Object and did not need to be explicitly programmed. By programming a variant object and a bit-level object we may generally cover all the cases we need. Below are the two objects necessary to synthesize a data set polymorphic ADC (add with carry):





SECTION 8 - VIVA SYSTEMS

VIVA can be used as the development environment for any hardware platform that can be defined in a *system description*. A system description defines the physical attributes and resources of a computing platform. In discussing a Viva System, we are typically referring to a high-level system description stored in a system description file with an .sd file extension. FAI.sd is one such system description. There are, however, many types of Viva Systems and FAI.sd itself describes numerous systems. The most common distinction between various systems is the classification of physical or behavioral. A physical system is a physical object, such as an FPGA, a shared bus on the FAI board or a sequential microprocessor on a Hypercomputers PC card. A behavioral system contains a Viva-defined behavior, which is ultimately implemented in a physical system. Simply put, a behavioral system contains VIVA code, whereas a physical system is a direct hardware reference.

8.1 FPGA Systems

VIVA is primarily targeted at Reconfigurable Computing (RC) hardware platforms. RC platforms are typically built upon Field Programmable Gate Arrays, or FPGAs. Hypercomputers are the most comprehensive RC platform available on the market today. A Hypercomputer contains one or more FAI boards. An FAI board is a set of two or more FPGAs with associated communications buses, memory, I/O capabilities and various clocks. The system description for an FAI board is contained in FAI.sd. On a HAL-15 Hypercomputer, FAI.sd contains 10 actual FPGA subsystems. They are Xpoint and PE1 through PE9. These systems represent the resources and structure that the FPGAs on a 10-chip FAI board contain. Ultimately, all Viva Behavior, with the exception of inter-chip transports and transports to the Widget Interface, is mapped into the configurable logic of these FPGAs.

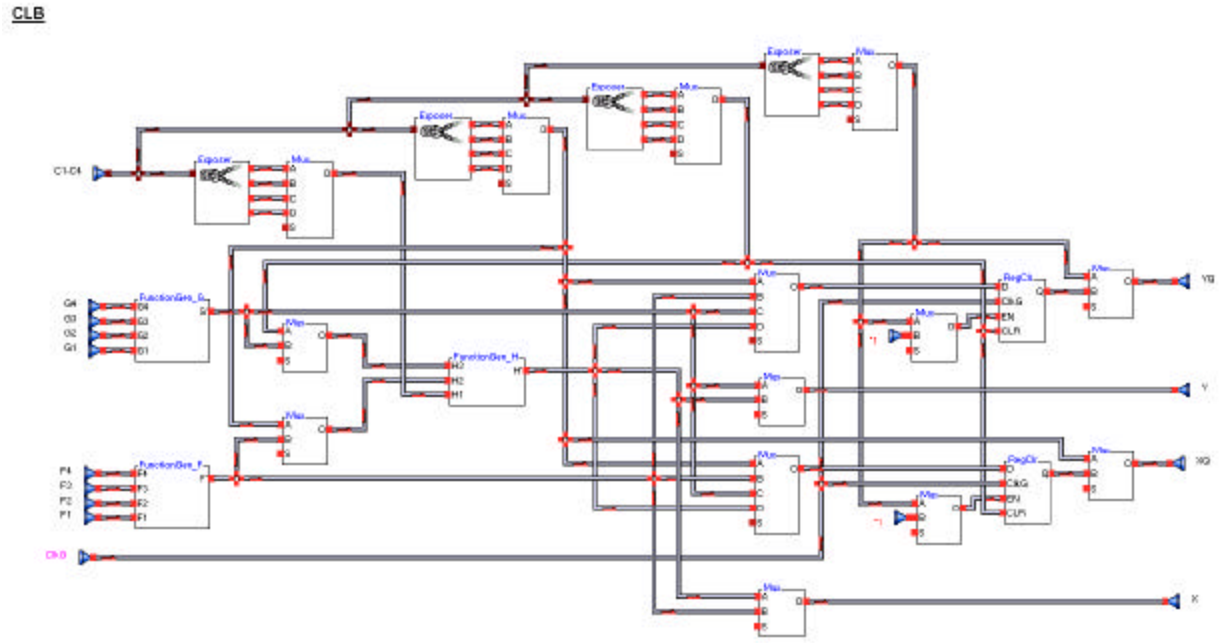
8.2 Configurable Logic Blocks

During the build process, Viva performs a formal recursive synthesis on all objects in an application, during which it expands complex variant objects into gate-level logic. All internal logic, whether it be simple gates, adders, multipliers or objects of higher complexity, is mapped into Configurable Logic Blocks (CLBs). CLBs exist en masse on an FPGA and contain inputs, outputs, function generators, multiplexors and Flip Flops. Of primary interest to us are the function generators and Flip Flops. Let's consider the slightly simplified diagram (fig. 1) of a CLB from a Xilinx 4062 FPGA. Note: this representation does not include the fast-carry logic and RAM elements associated with the F and G function generators. These will be covered separately.

A 4062 CLB contains 3 Function Generators, F, G and H. The function generators basically contain programmable LUTs (look-up tables). When the chip is programmed, the LUTs are coded so that their outputs reflect the output of the combinational logic function of the function generators. F and G are 4-input LUTs, and H is a 3-input LUT. The combinations of LUTs in a single CLB are capable of implementing any independent functions of up to 5 inputs, and some functions of between 6 and 9 inputs. All of the logic that your Viva program represents will be coded into these LUTs when the FAI board is programmed.

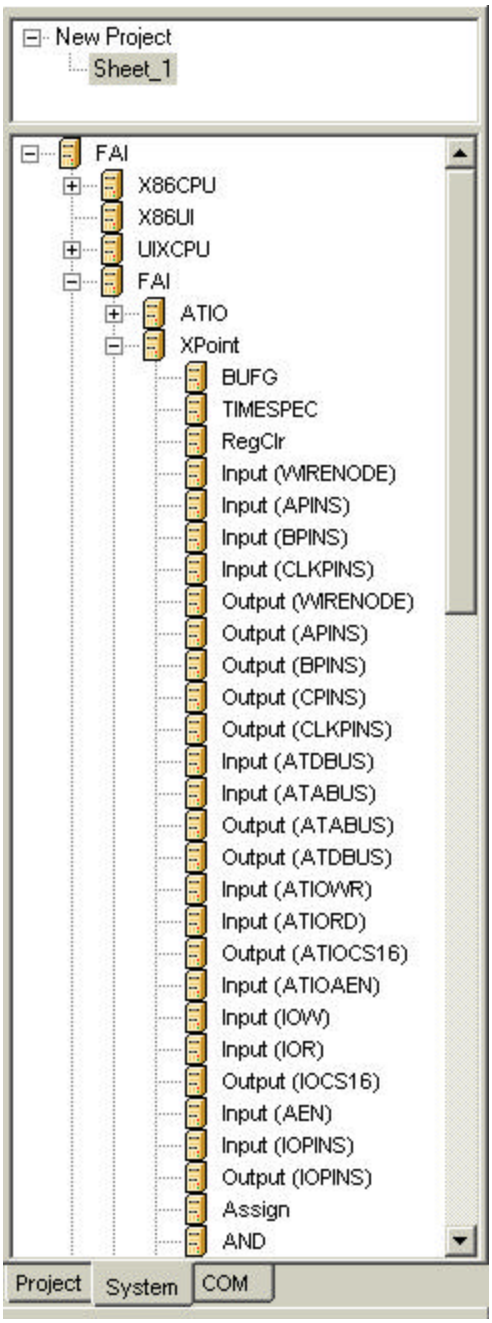
Each CLB also contains two edge-triggered D-type Flip Flops. These Flip Flops are the hardware implementation of Bit-level RegCLR objects. (And all other Registers as well.) If you have a 16-

Bit RegCLR, it will utilize the Flip-Flops of eight CLBs. Each CLB is capable of implementing up to 2-Bits of a register. Optimal speeds are obtainable in FPGAs when application design has as few layers of logic as possible in between Flip Flops. Most Viva library objects are designed with single layers of logic where the function generators feed the registers in their same CLB.

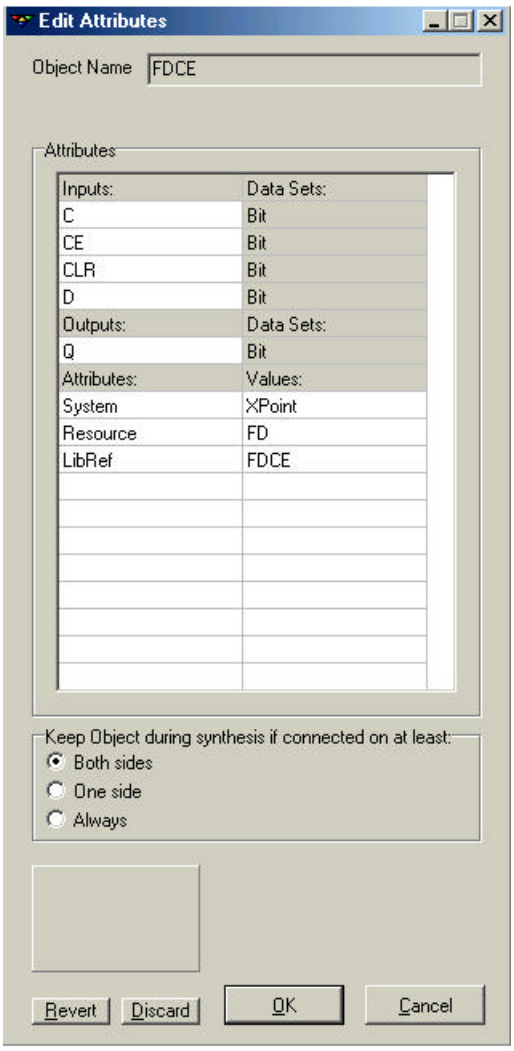


We can see that a 4062 FPGA contains 8 data inputs and 4 data outputs. Multiplexors are used to determine internal routing of the CLB. In other words, the multiplexors determine whether the Flip-Flops are fed by the function generators or by an external source via data on of the 'C' inputs. Multiplexors also arbitrate the data outputs sources.

Each FPGA system has its own object library associated with it. These are a set of primitive objects that can be implemented in the FPGAs directly. When application behavior is targeted to an FPGA system, the synthesizer recursively expands application behavior and resolves all recursion leaf-nodes in the system library. Primitive system objects have no behavior, so you cannot drill into them. To view an FPGA library, open FAI.sd, select Enable Editor under the System menu, click on the System Objects tab at the bottom of the VIVA window and expand the following system: FAI\FAI\XPoint.



If you view the Attributes editor for any Primitive system object, let's say FDCE from the FAI\FAI\Xpoint System, you should see three attributes. These are System, Resource, and LibRef.



Let's examine the meaning of each of the attributes of the FDCE object.

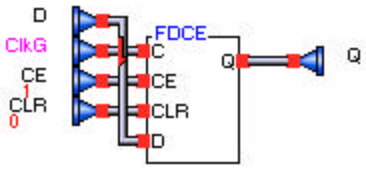
System -> Xpoint
This means that this object belongs to the Xpoint System. You can create system objects by setting the System attribute when you wrap up a behavior into an object. The behavior of a Primitive object is just the inputs and outputs that compose its interface.

Resource -> FD
Every primitive object contains a Resource attribute. This attribute contains an entry into the resource prototype shown in the Resource Editor. Viva calculates the number of resources it consumes based on this attribute. Each resource also has a cost associated with it that may affect the system into which it is partitioned. This particular resource, FD, refers to one of the Flip Flops, as this is what FDCE implements.

LibRef -> FDCE
This attribute points to the real behavior of this object. The 'FDCE' string refers to the name of a netlist contained in Viva/VivaSystem/FPGAStrings.txt. All of these netlists are primitive Xilinx objects that map directly into FPGAs.

The FDCE object is ultimately the bit-level representation of all of our registers. The FDCE netlist contains the information necessary to configure one Flip Flop as a RegCLR. If we look at any of the Reg objects inside the Xpoint System library, we will see that the behavior of each of these references the FDCE object.

RegClr



Included in the system object library is everything Viva needs to map your behavior into the FPGAs. The objects may as simple as inputs and outputs to communication systems or AND, OR and INVERT. You have objects that implement tri-state buffers (IOTBuf), Logic Zero (GND), Logic One (VCC), and I/O Pads (OPAD, DPAD). You also have objects that implement LUT-based RAM and fast-carry propagation chains.

SECTION 9 - SYNTHESIZER GRAPHICS DISPLAY

The Synthesizer Graphics Display is presented during the compilation phase of a VIVA program.

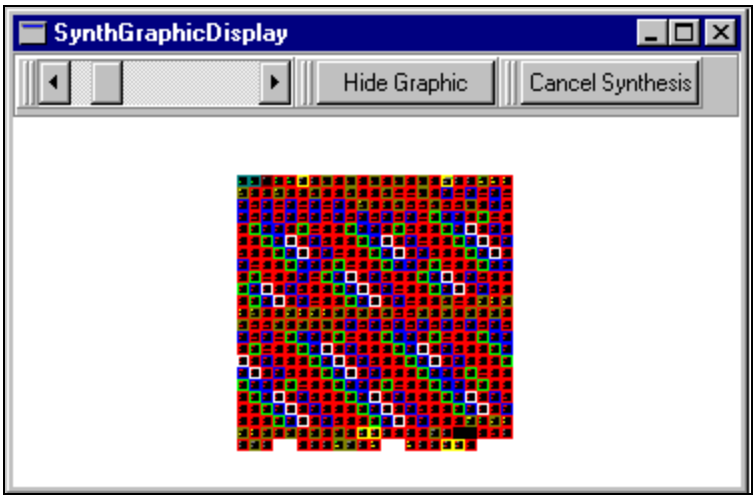


Figure 39 - Compile Time Graphics Display

The items depicted in the display consist of the individual components of the portion of the program being compiled. Pressing the arrows in the control bar with the mouse can alter the size of the objects.

By hiding the synthesizer graphics display, the user can decrease the compilation time.

The colors of these Objects are as follows:

Object	*Color*
High Level Objects	Yellow
Primitive Objects:	
Transport	Red
AND	Blue
OR	Lime
INVERT	White
ASSIGN	Aqua
INPUT	Teal
OUTPUT	Maroon
Other	Olive

SECTION 10 – VIVA PROGRAMMING EXAMPLE

A Simple "Hello World" Style Viva Application

This is a simple tutorial designed to teach a beginning user the basics of VIVA and its graphical user interface system. It will go over VIVA's basic environment, such as how to create objects, open and save projects and sheets, and the menu selections. It will then go over a step–by–step tutorial that involves using a few simple objects in order to familiarize the user with the VIVA system.

You must have VIVA running! Start the VIVA application by executing Viva.exe in the base VIVA directory either directly or through the use of a shortcut. Then, before we begin programming in VIVA, we need to load a system and some libraries.

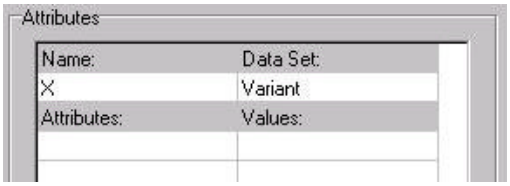
- First, select the System/Open System tool command or click on the 'open system' tool bar icon. This will open a file menu. Double click on the FAI.sd file.
- Now we will load a couple of library files we will need. These libraries are in the form of "sheets" that contain all of the objects that were in that sheet when it was saved. Use the Sheet/Open Sheet menu option or click on the 'open sheet' tool bar icon. Double-click on the FAI_Lib file (you will likely get a message saying VIVA has detected datasets that are not referenced. Just click on the 'No' option).
- Now open another sheet, this time CoreLib. You will get a message saying that some ambiguous objects were not loaded. The meaning of this will be explained in another tutorial. For now, just click the 'Ok' button on the message box.

Note how several new objects appeared on the right side of the screen after loading each of the libraries. These are all previously saved "programs" that can be freely used in new projects. The illustrations on the next page show what you might see before and after loading the system and libraries. Now we are ready to begin creating a simple program! For our program, we will create an application that squares a variable-input number.

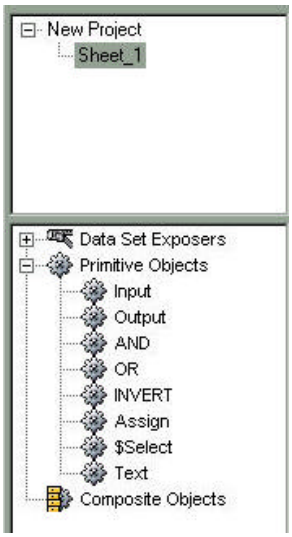
First, click the "new sheet" button, explained above. You should be looking at a blank sheet, with "Sheet_1" on the upper-left hand corner of it.

First, click and drag an the 'Input' object to the application editor side of the screen. As you drag the cursor over the sheet, the object turns into a symbol representing that object. In this case, the input object will look like this.

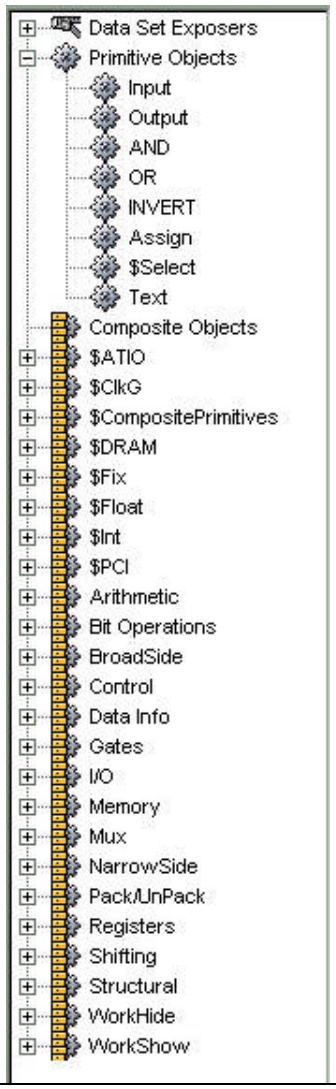
Now right click on the input symbol. This opens up a window that describes this symbol.



Before:

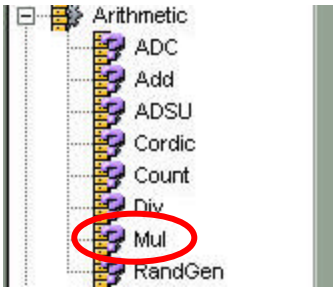


After:

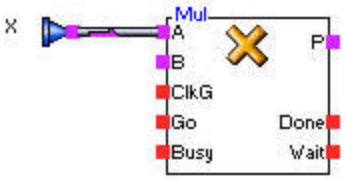


Go ahead and change the name to “X”, as above. Also, click on the “Variant” box under the “Data Set:” This opens up a list of possible data sets that this input can use. After looking through the list a little, go back up to the top and select “Variant”. Note that this was the default Data Set. Click ‘Ok’ to continue.

Now we need a multiplier. Go back to the objects side of the screen, and expand the “Arithmetic” option. Drag the ‘Mul’ object over onto the sheet.

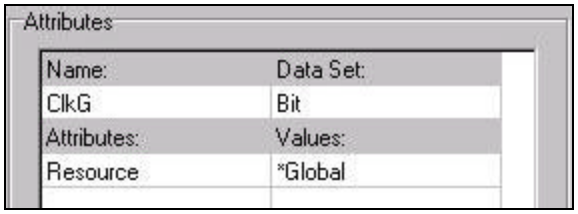


Notice the red and purple squares on the sides of the multiplier object you just pulled out. These are the inputs and outputs that this object has. First, lets connect our input to the inputs of the multiplier. Click on the input's purple box and move the cursor over to the "A" box on the multiplier. A "pipe" will appear representing the connection. Click on the "A" box to complete the connection. You can delete the pipe by clicking on it to select it and pressing the delete key.



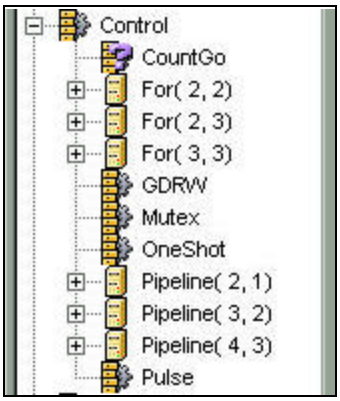
Since we are just squaring the number, we need to also connect the "X" to the "B" input of the multiplier. Do this by double clicking on the pipe. This creates a "split" in the pipe where you can connect multiple outputs from it. Connect one of these extra outputs to the "B" on the multiplier. Alternatively, you can get a pipe from the "B" on the multiplier and double clicking it on the pipe.

Now, lets connect the ClkG input on the multiplier. This represents how often to tick the clock that drives the circuit. Drag another input to the sheet, this time dropping it right on the ClkG on the multiplier. Notice how the name of the input symbol automatically changes to "ClkG", and it is also already connected to the ClkG on the multiplier by a pipe. The easiest way to set the input of the clock is to make it a global resource. Right click on your new ClkG input symbol. Click on the box below attributes and change it to "resource". Click on the box below Values and change it to "*global"

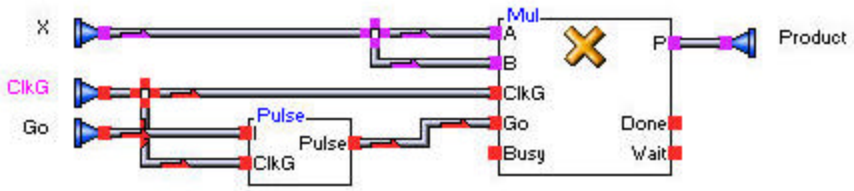


In order to finish this program, we'll need one more object and a bunch of input and output symbols to connect the multiplier. The last object we need is a "Pulse" under the control treegroup.

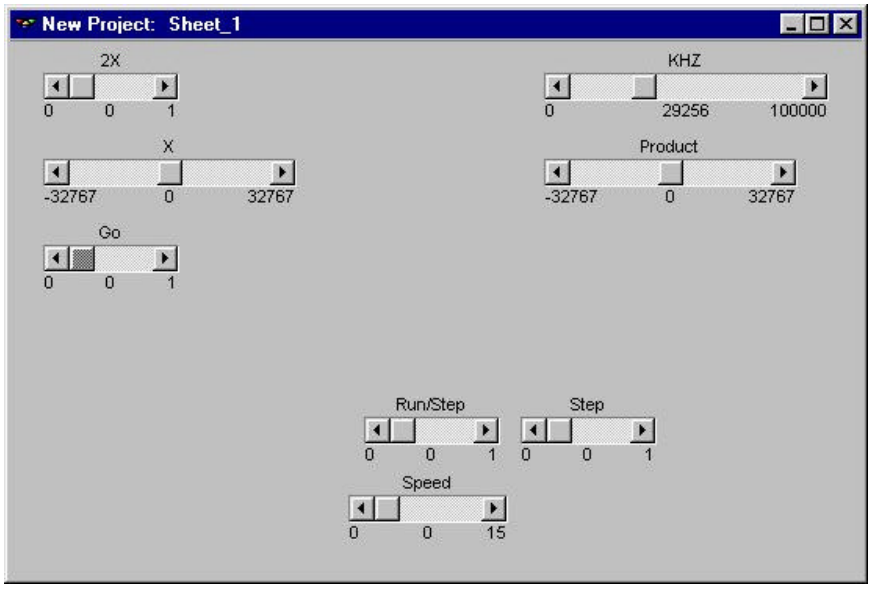
Pulse takes a go input and sends a single "go" signal to the multiplier, which will cause it to make its computation and output an answer.



Now connect up a bunch of inputs and an output, as in the picture below:



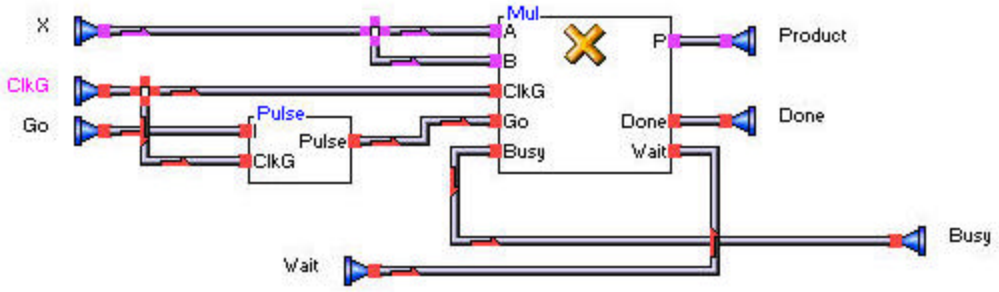
This is all you need to create a functional square root program. Now press the play/pause button on the toolbar. A window will pop up asking what data set you want to use for the undefined dataset. Go ahead and select int. Alternatively, you could have selected the desired data set when you changed the name of the “X” input node. Viva will now compile your program and then pop up a run window.



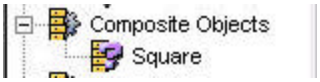
Note how there are scrollbars labeled for each of your inputs (except ClkG, which was converted to a global resource) and output. Go ahead and change the value of the X scrollbar, and toggle to go scrollbar. An answer will appear on the Product output, which should be the correct answer

unless you input a value that is too large for the Int dataset. After you are satisfied that the program works, close the run window.

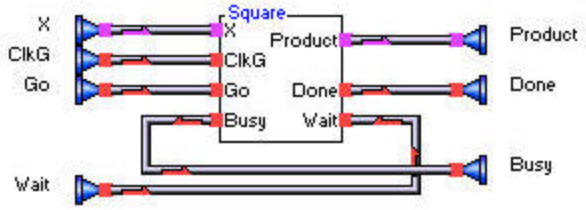
Now, lets convert your square root machine into an object. First, though, lets hook up the rest of the inputs and outputs. Although they are not necessary for this tutorial, it is an important habit to get into, as these will come in very useful for later on.



Press the convert sheet to object button on the toolbar. A window will pop up with a summery of the object and asking for a name. Name it “Square” and press ok. The sheet you had open closes and a new object becomes available.



Now click the new sheet button and drag this new object onto the screen. It has all the inputs and outputs your object had. Go ahead and connect them with inputs and outputs:



Go ahead and run it to verify it works exactly as the object you just made did!

This concludes the sample application tutorial.

