

Table of Contents

Virtio Innovator - Design Creation Tutorial.....	1
CONFIDENTIALITY NOTICE	2
RESTRICTED RIGHTS LEGEND.....	2
Trademarks	2
Documentation Conventions	2
History	3
Step-by-Step Example	3
Step-by-Step	3
Example.....	3
Introduction.....	4
Starting a New Design Project	4
Adding Processes to the Design	5
Describing The Tester Process	7
Describing the DUT Process.....	14
Browsing Design Information	16
Setting the Environment for Compilation and Linking	19
Setting the Project Specific Environment.....	21
C++ Code Generation and Compilation.....	22
Debugging the Design.....	23
Tracing Signal Values	27
Using the Waveform Viewer During a Prototyping Session	28
Using the Waveform Viewer for Post-processing Analysis.....	31
The Test Bench Concept	32
Modify the Prototype for Use with a Test Bench.....	33
Creating a Test Bench	36
Running a Test Bench	39
Symbols.....	41
Creating Magic-C Symbols	42
Modifying a Symbol or Its Implementation After Creation	45
Using Magic-C Symbols	46
Opening a Symbol or Its Implementation.....	48
Code Examples	51
MAGIC-C Code Examples	51
Introduction.....	52
Creating Process Concurrency	52
Reset: Bringing an FSM to an Initial State	53
Starting and Stopping a Process	54
Timers	54

Generating Clock Signals Using the Clock Construct.....	55
Generating a Clock Signal with a Duty Cycle of 30%.....	56
Getting the Current Prototyping Time	57
Synchronizing Data Transmission Using a Clock	58
Synchronizing Data Transmission Between FSMs.....	59
Clocked Magic-C Loops	60
Do-While / Repeat-Until Clocked Loop	60
Clocked WHILE Loop	61
C-Style Clocked FOR Loop.....	62
Interrupt	64
Matched Filter Design	64
Overview.....	64
Introduction.....	65
Matched Filter Specification.....	65
System Partitioning	66
Specification of Matched Filter (Initial Version).....	68
Data Generator Specification.....	68
Filter Specification	70
Adding Delay and Time to the Model	72
Using Reset in a MAGIC-C Model.....	74
Handling Protocol Refinement in MAGIC-C.....	76
Summary	80
UAR Design	80
UAR Design	81
Using MAGIC-C	81
Introduction.....	81
Universal Asynchronous Receiver Specification.....	81
Change in Specification: Noise Resistant Behavior	83
Basic Test Bench Framework.....	84
Communication Protocol	85
User-Defined Data Type Across Model Interfaces.....	85
Refinement Steps for the UAR Framework.....	86
Initial version	87
Version 2	92
Version 3	95
Version 4	99
Final version	101
Summary	107
Glossary	109
Index	111

Virtio Innovator - Design Creation Tutorial



Virtio Innovator

Design Creation Tutorial



Copyright© 1999-2001 Virtio Corp.

CONFIDENTIALITY NOTICE

No part of this publication may be reproduced in whole or in part by any means (including photocopying or storage in an information storage/retrieval system) or transmitted in any form or by any means without prior written permission from Virtio Corporation.

Information in this document is subject to change without notice and does not present a commitment on the part of Virtio. The information contained herein is the propriety and confidential information of Virtio or its licensors, and is supplied subject to, and may be used only by Virtio's customer in accordance with, a written agreement between Virtio and its customer. Except as may be explicitly set forth in such agreement, Virtio does not make, and expressly disclaims, any representation or warranties of its completeness, accuracy or usefulness of the information contained in this document. Virtio does not warrant that use of such information will not infringe any third party rights, nor does Virtio assume any liability for damages or costs of any kind that may result from use of such information.

This document contains unpublished confidential information and is not to be disclosed or used except as authorized by written contract with Virtio. In the event of publication, the following notice is applicable:

RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the Government is subject to restrictions set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013.

© 1999-2001 Virtio Corporation. All rights reserved.

The entire notice above must be reproduced on all authorized copies.

Trademarks

Virtio Innovator™, Integrator™, and Explorer™ are trademarks or registered trademarks of Virtio Corporation. Windows® 95, 98, and NT 4.0 are registered trademarks of Microsoft. Microsoft Visual C++ is a product name of Microsoft. Adobe Acrobat® is a registered trademark of Adobe.

Documentation Conventions

Each of the documentation conventions used in this *Design Tutorial* is explained below and an example of provided.

- Bold, italicized text in navy blue specifies a name or label on an Innovator window dialog box that cannot be altered by the user. Examples include window names, toolbar names, toolbar button names, labels on dialog boxes (check boxes, text-entry sections, etc.), pull-down menu entries, keyboard key names, etc. This text appears as follows:

Navigate Toolbar

- The `courier` font in navy blue indicates text that the user has or can enter. It would also include the names for a Process, Block, Symbol, etc., since this name was entered by a user at one time. It looks like:

```
-c -nologo -I . -I "$(CPP_DIR)" -I "$(ENGINE)" -MTd -zi
```

- Underlined text in blue indicates a hyperlink. Aligning the cursor over this hyperlink and single-clicking sends one to another page or web site. It's appearance is as follows:

[Design Browser](#)

History

Version 2.01	Virtual Silicon 2.0 (Beta) documentation	11/12/1999
Version 2.02	Virtual Silicon 2.0 (Product) documentation	02/10/2000
Version 2.03	Virtio Integrator (Product) documentation	01/08/2001
Version 2.04	Virtio Integrator (Product) documentation	01/15/2001
Version 2.05	Virtio Integrator (Product) documentation	01/19/2001
Version 2.06	Virtio Integrator (Product) documentation	02/04/2001
Version 2.07	Virtio Innovator (Product) documentation	04/30/2001

Step-by-Step Example

Step-by-Step Example

This chapter illustrates the many features of the Virtio Integrated Development Environments through an ongoing example design. By the end of this chapter, the user should be able to create, compile, and debug a Virtio prototype.

Note: The MAGIC-C code for all the examples in this tutorial is installed at the same time as the Virtio Innovator. The examples are available under `<virtio_innovator_install_path>\IDE\examples`, where `<virtio_innovator_install_path>` is the absolute path chosen during Virtio Innovator installation.

Introduction

This chapter takes the reader through the creation of a running design example called `tutorial`. Along the way numerous features of the Virtio Innovator are illustrated. Since this is a process with many steps, it has been broken into individual pages as follows:

1. [Starting a New Design Project](#)
2. [Adding Processes to the Design](#)
3. [Describing the Tester Process](#)
4. [Describing the DUT Process](#)
5. [Browsing Design Information](#)
6. [Setting the Environment for Compilation and Linking](#)
7. [Setting the Project Specific Environment](#)
8. [C++ Code Generation and Compilation](#)
9. [Debugging the Design](#)
10. [Tracing Signal Values](#)
11. [The Test Bench Concept](#)
12. [Creating a Test Bench](#)
13. [Running a Test Bench](#)
14. [Creating Symbols](#)

A running design example called `tutorial` in this and the next chapter is continually modified to illustrate usage of different Virtio Innovator features.

The design `tutorial` is extremely simple since it has only the following two devices:

1. A `Design Under Test (DUT)`
2. Another device called `Tester`.

The devices perform a simple handshaking protocol five times. The `Tester` device initiates communication by sending a `go` message. The `DUT` device responds by sending a `go_more` message. `Tester` halts the handshaking after the `go` message has been sent five times, having used an internal clock to count the number of times handshaking has occurred.

Starting a New Design Project

To begin creating a new design:

1. From the **File** menu, select **New Project**.
2. Provide a design name (for example, `tutorial`) and a directory for the design (for example, `c:\vs_tutorial`). If the directory does not exist, the editor creates the new directory. Use a valid C identifier name for the design.

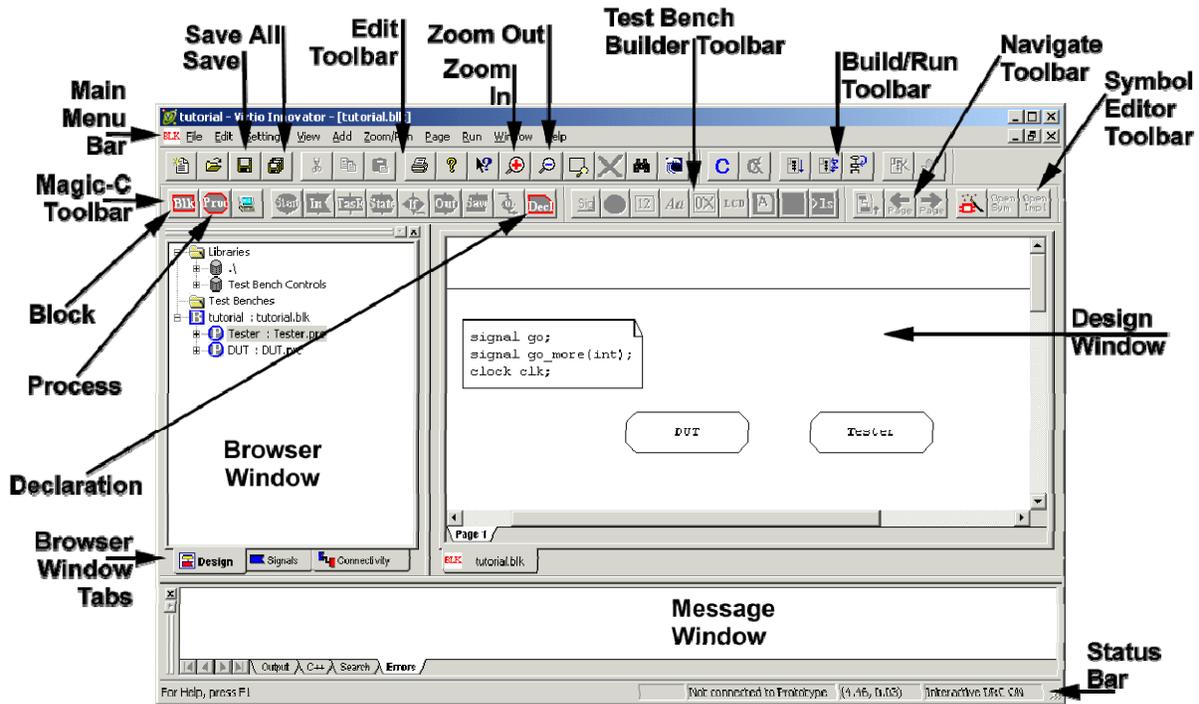


Figure 1: Layout of a Prototype in the Virtio Innovator

The design is created as a **Block**. This Block is added to the Design Window as the top-level Block of the design. The first page of the `tutorial.blk` block opens in the Design Window.

Adding Processes to the Design

Now we will add two processes, the first named `DUT` and the second named `Tester`, to the top-level Block as shown in Figure 1. To do so, follow the steps on this (to add the `DUT` Process) and the next (to add the `Tester` Process) pages:

1. Left-click once on the Process icon  on the Magic-C Toolbar or click **Add** on the Main Menu Bar and select **Process**. The Magic-C Toolbar

 is located at the top of

the Virtio Innovator screen. [If the Magic-C Toolbar is not visible, first click **View** from the Main Menu Bar then select **Magic-C Toolbar**.]

Note: While the top level Block is displayed in the Design Window, only the Block () , Process () and Add CPU () icons are available on the MAGIC-C Toolbar since these reflect the only permitted actions. The rest of the icons on this toolbar are grayed-out. As we will soon see, these other icons become available when editing an implementation of a Process.

2. Select the desired location of the first Process and left-click to place it. An error message (**Reference has no implementation.**) will appear in the Message Window at the bottom of the Virtio Innovator screen (see Figure 1 on the previous page for Message Window location). Don't be concerned about this error message as it will disappear as soon as we define the contents of this new Process.
3. While the newly created process is still selected, type the name of the process inside the Process construct text window. Be sure to use a legal C variable name. For example, in the case of the **DUT** process, type in "**DUT**".

Note: To resize the Process construct drag any of the highlighted corners while the construct is selected.

4. To change the properties of text, select the text and right-click to display a drop-down menu.
5. Select **Properties** to display the Properties dialog box shown in Figure 2a. If necessary, change the text font using the **Font** button and/or the background color using the **Background Color** button.

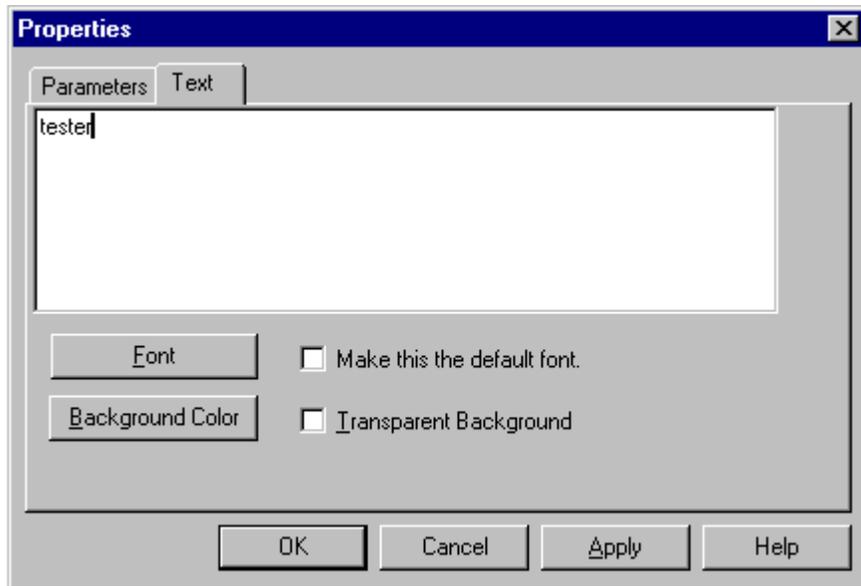


Figure 2a: Changing Text Properties

6. Repeat steps 1 through 3 above to create the other Process named `Tester`.
7. Define two **MAGIC-C** signals called `go` and `go_more` by placing the two C-language statements listed below in the Declaration construct (see Figure 2b) of the `tutorial` Block. Note that the `go_more` signal carries an integer as a payload. [For additional details, see the *Magic-C Language Reference Manual*, Process and Declaration.]


```
signal go;
signal go_more(int);
```
8. Define a clock named `clk` by placing the statement `clock clk;` in the Declaration construct. Now the Declaration construct in the `tutorial` Block should look like the one depicted in Figure 2b below.

```
signal go;
signal go_more(int);
clock clk;
```

Figure 2b: Contents of the Declaration Box In The Top Level Block Of The `tutorial` Design

Note: In every level of the hierarchy, the common MAGIC-C Variables and Signals for that scope are defined in a MAGIC-C Declaration construct. All MAGIC-C Signals employ broadcast semantics. In other words, every Process in the `tutorial` Block that has a Signal-In MAGIC-C construct for a given Signal automatically receives that Signal. The user does not need to send the signal explicitly to each Process.

9. Now that the processes are positioned in the `tutorial` Block, click the **Save File**



icon on the Edit Toolbar to save the design. The Edit Toolbar



is located

at the top of the Virtio Innovator screen. [If the Edit Toolbar is not visible, first click **View** from the Main Menu Bar then select **Edit Toolbar**.]

Describing The Tester Process

Describing each Process, an example of which is shown in Figure 3a, is the most labor intensive part of this design example. The Process is that level in the hierarchy where actual work is being done by the prototype when it runs. Each process contains a Finite State Machine (FSM) or some portion thereof.

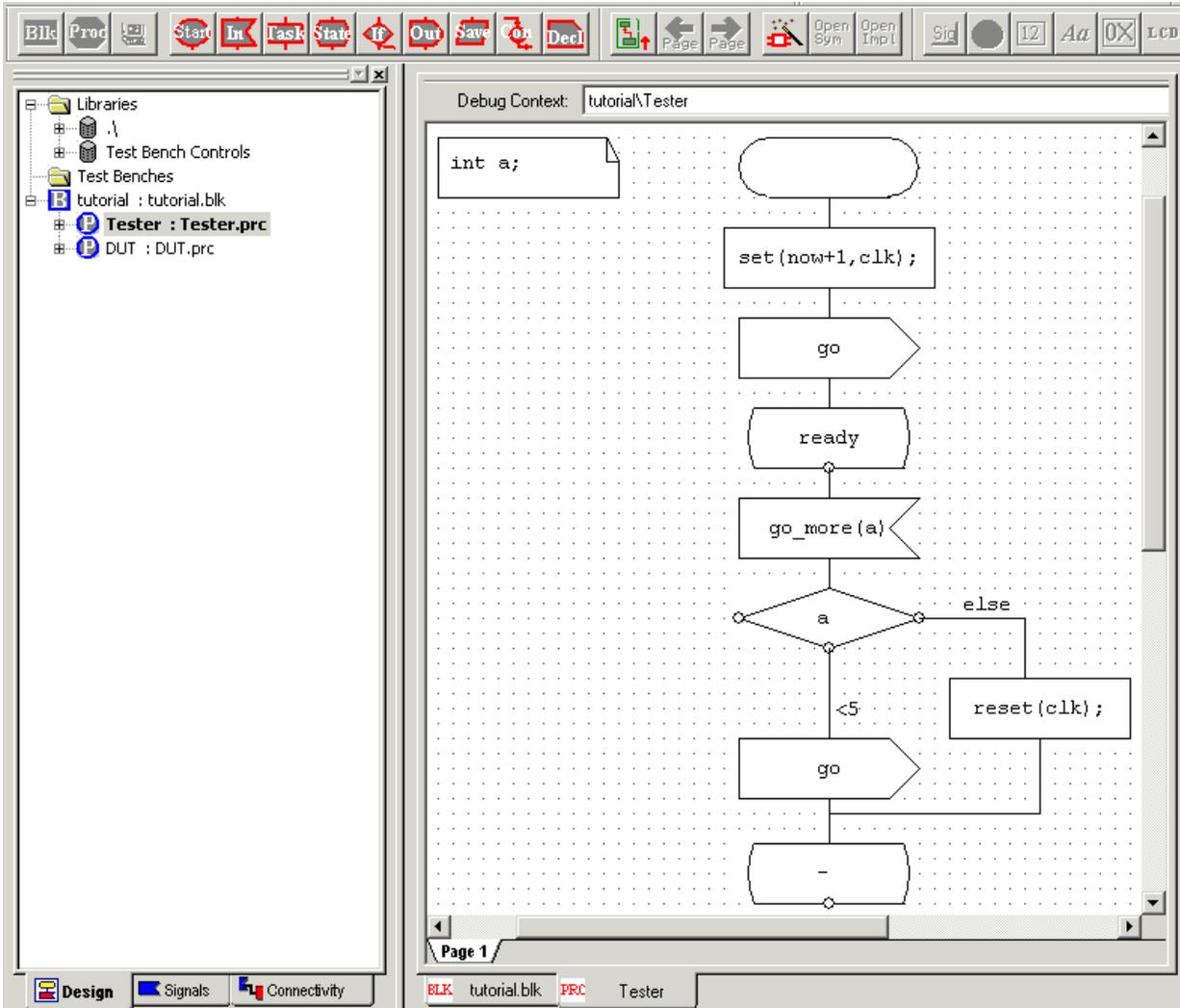


Figure 3a: Describing Process Tester

To describe the `Tester` Process, proceed as follows:

1. Double-click on the `Tester` Process to open the implementation of that Process and move one level down in the design hierarchy. A dialog box will appear asking if you wish to create a new file.
2. Click on the **YES** button or press the keyboard **Enter** button to create the new Process. The `Tester` Process implementation opens in the Design Window and is ready for editing.
3. The **MAGIC-C** Start construct is displayed and should be selected. The Start construct designates the first state of the Finite State Machine. If the **Start** MAGIC-C construct is not selected, select it by locating the cursor over it (note that the cursor becomes a crosshair) and left-clicking. [Be careful to select the Start construct (its outline will turn red) and not the text box within it (the text box turns yellow when selected and a text cursor is placed inside it).] In Figure 3b only the text box in a Task construct is selected while in Figure 3c the entire State construct named `ready` is selected.

4. Left-click on the **Add Task**  icon on the Magic-C Toolbar located near the top of the screen. The Task construct is automatically connected to the tail of the Start construct. Although you should not do so now, in general the steps for adding a new MAGIC-C construct are:
- Deselect any selected MAGIC-C constructs.
 - Left-click the icon of the desired construct, move the cursor over the Design Window and left-click again to place the construct at a convenient location.
 - To make a connection between two constructs, select the first construct, and then left-click on the connection-point at the center bottom of that same construct.
 - Move the cursor anywhere within the second construct and left-click.

Note: There are clearly defined rules regarding which Magic-C constructs can connect to which other constructs. As a result, some direct construct-to-construct connections are not permitted and cannot be made by the user. This topic is beyond the scope of this tutorial, however for specifics of these rules see the *Magic-C Language Reference Manual*.

5. Note that the inner box of the Task construct is yellow. The yellow color denotes a selected text box. Any legal C code can be placed here, including any MAGIC-C keywords, data types, or pre-defined functions. For the example depicted in Figure 3a, the clock signal (`clk`) will now be set to have a period of 1 prototyping unit. To do so enter the following text in this Task construct:

```
set(now + 1, clk);
```

Figure 3b depicts the Task construct with a highlighted text box.

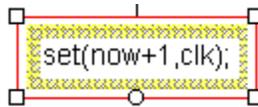


Figure 3b: Selected Text Box in Task Construct of Project Tester

Note: As you proceed through the instructions on this page, error messages will appear in the Message Window at the bottom of the Virtio Innovator screen. In most cases the offending text or construct can be highlighted by double-clicking on the error message in the Message Window. However for now do not be concerned about error messages. Once the tutorial design is completed by the end of the next page all of these messages will be gone because all error conditions will have been removed.

6. Select the Task box by locating the cursor over it (the cursor becomes a crosshair) and left-clicking.



7. Left-Click on the **Signal-Out**  icon on the Magic-C Toolbar. A Signal-Out construct will automatically be placed below and connected to the Task construct. Note that the text box inside the Signal-Out construct is selected (its border is yellow).

- Type `go` in the text box of the Signal-Out construct.

Note: Typing `go` in the Signal-Out construct broadcasts this signal to all Processes. In order for other Processes to receive broadcast signals the following two additional conditions must be met: 1) The same signal must be declared at a higher level of the design hierarchy (thereby providing the signal to Processes lower in the hierarchy; this step was done previously when `go` was defined in the tutorial Block); 2) The same signal must be received with a Signal-In construct. This rule always applies when using Magic-C syntax, which employs broadcast semantics for signals. See the [MAGIC-C Language Reference Manual](#) for further details.

- While the MAGIC-C Signal-Out box is selected, left-click on the **Add State**  icon to add a State construct that is connected to the Signal-Out construct. Name this state `ready` and select the `ready` State construct. The content of the Design Window of your Tester process should match that shown in Figure 3c.

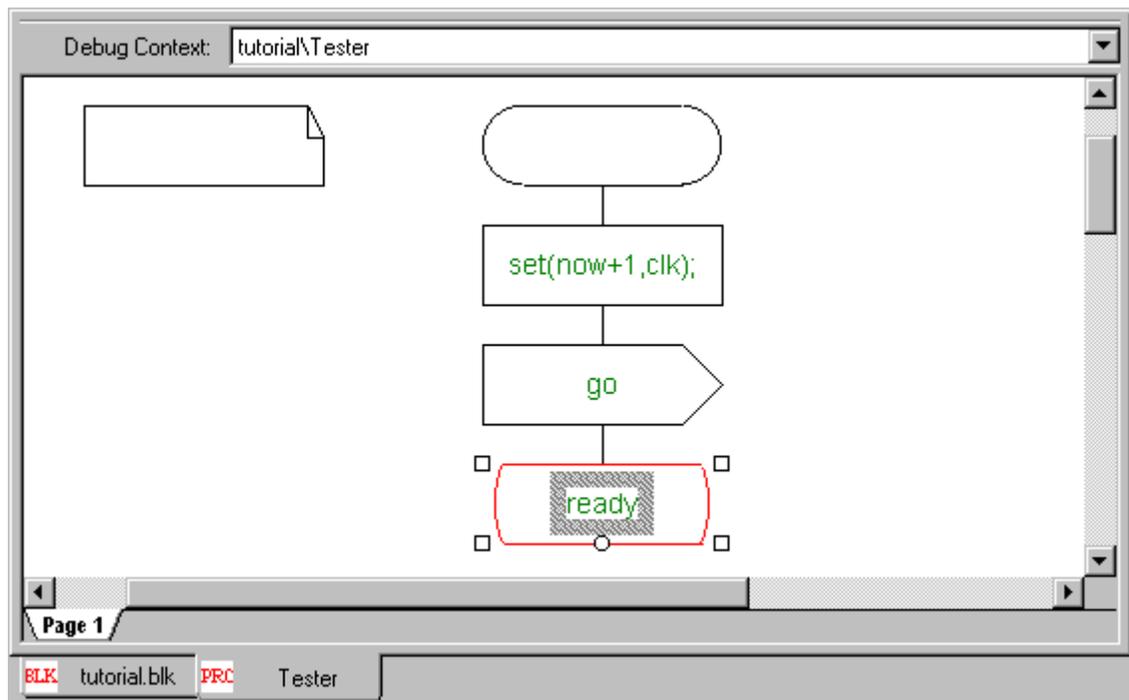


Figure 3c: Partially Completed Process Tester

- Click the **Signal-In**  icon on the Magic-C Toolbar to add the signal input called `go_more(a)`. By doing this, we allow the `Tester` Process, when it is in the `ready` state, to receive the `go_more` signal and depart the `ready` state. `Tester` then executes the code located after the Signal-In construct.

Note: The `go_more` signal has an argument called `a`. The variable `a` takes the value of the payload that comes with the signal `go_more`. The variable `a` is local to the `Tester` Process.

Note: The construct connected to the bottom of a State construct MUST always be a Signal-In construct according to the semantics of MAGIC-C. As such, state transitions in MAGIC-C are always guarded by the receipt of a signal. To state the same concept differently, MAGIC-C Finite State Machines advance upon receipt of signal(s).

11. Declare variable `a` in the Declaration construct by typing `int a;` as shown in Figure 3d.

12. Select the Signal-In construct and left-click the **Decision**  icon on the MAGIC-C Toolbar. A Decision construct will automatically appear, be connected to the Signal-In construct and have a text cursor placed inside it. [Note that the text box is highlighted in yellow.]

13. Type `a` on the keyboard to place this character inside the Decision construct and thereby test the value of payload `a` received with signal `go_more`.

14. Click the Signal-Out  icon on the Magic-C Toolbar. A Signal-Out construct will automatically appear, be connected to the Decision construct and have a text cursor placed inside it. [Note that the text box is highlighted in yellow.]

15. Type `go` on the keyboard to place these characters inside the Signal-Out construct and thereby broadcast signal `go`.

16. Left click on the line connecting the Decision construct to the Signal-Out construct. A text box will be highlighted in gray next to the connecting line (which is sometimes called an 'arc'). Place the cursor over the gray text box until it turns to a pencil and left-click (to highlight the box in yellow and place a text cursor). Then type the characters `<5`, which has the effect of sending signal `go` only if `a < 5`. Figure 3d depicts how the Process `Tester` should now appear.

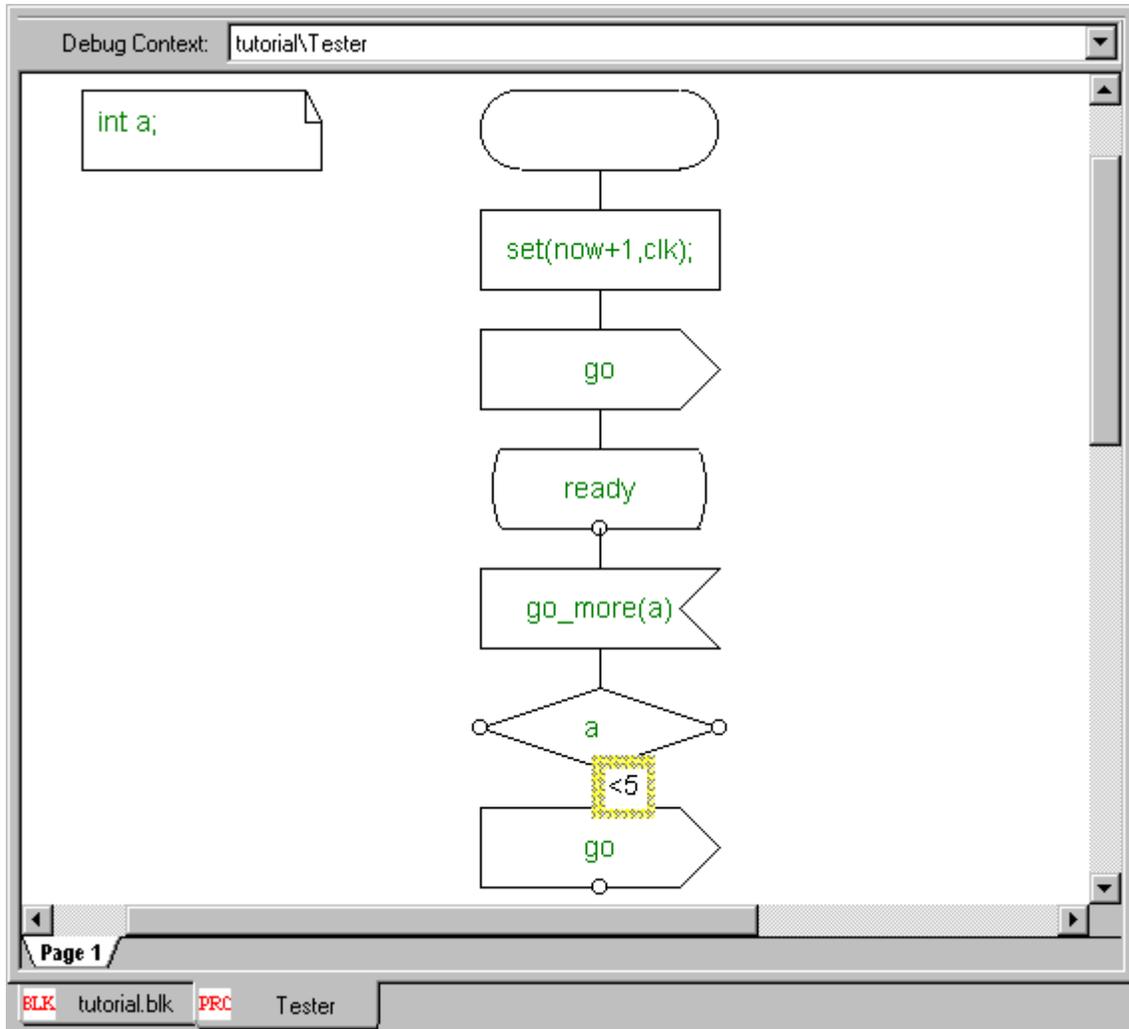


Figure 3d: Partially Completed Process Tester

Note: Steps 13 through 16 above introduced a key concept that we will briefly explain. The Decision construct added in step 13 acts like a C-language Case statement. The condition under test is the caption inside the Decision construct (which is the value of signal payload `a` in this case; conditions must always be a legal C expression). The possible values for a Decision construct are (legal C expressions) in text boxes associated with each of the lines connecting the left, right and bottom corners of the construct to the rest of the design. In step 18 we displayed one of these connecting line text boxes and added a value to it.

17. Now connect a State construct to the Signal-Out construct by selecting the Signal-out construct (its outline will be highlighted in red as shown in Figure 3c) and clicking the



State icon. Type a dash ("-") inside the new State construct. The dash means "return to the previous state", which in this case means the `ready` state. An alternative to using the dash is to call the newly created state "`ready`", thereby automatically referring to the previously defined `ready` state of this FSM.

Note: This rule holds throughout all MAGIC-C specifications and its use in place of loops to return to a previously defined state removes visual clutter from the prototype. To refer to a previously defined state, simply reuse the same name inside a subsequent state.

18. Now we will connect another path from the Decision construct for conditions other than $a < 5$. To do so, deselect any constructs that are selected (click outside all of them).

Then Add a Task construct by left-clicking on the  icon on the Magic-C Toolbar. Place it to the right of the `go` Signal-Out construct. Type `reset(clk);` inside the new Task construct. Whenever this block is executed, it will reset the clock named `clk`, causing it to stop running.

19. Now connect the Decision construct with the Task construct containing the `reset(clk);` text. Do this by selecting the Decision construct, left-clicking on the right corner of the Decision construct (the cursor changes to a pencil when placed over the corner) and then left-clicking inside the Task construct. A line or 'arc' now connects these two constructs.

20. Type `else` in the text box corresponding to the new arc to handle all cases not explicitly covered by the Decision construct. The lower part of Process `Tester` should now look like Figure 3e.

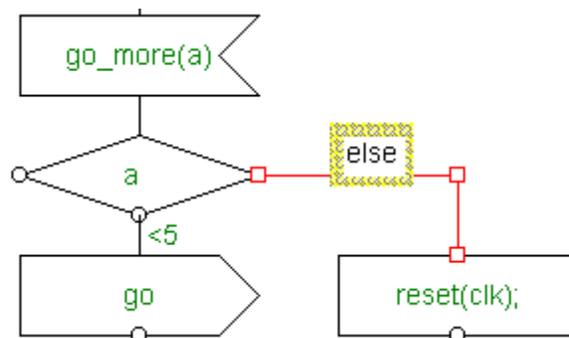


Figure 3e: Lower Portion of Partially Completed Process `Tester`

21. Now connect the Task construct which resets the clock with the State construct containing a dash. To do so move the cursor over the bottom of the Task construct until the cursor changes to a pencil, left-click, and move the cursor over the dash State construct. Left-click inside the dash State to complete the connection. [It may be necessary to move the connecting line to make it look like Figure 3a. To do so, left-click to select the line (it turns red), move the cursor over the line (the cursor changes to a short line with two arrowheads) then click and hold the left mouse button while dragging to move the line. To move an individual vertex, align the cursor over it (the cursor changes to a crosshair icon with the text `Point` next to it), click and hold the left mouse button then drag.]

Note: Connections are always made between MAGIC-C constructs, even if some connections appear to connect to an existing link. The connection between the Task construct and the dash State construct is an example of this.

22. Now check that your design agrees with that shown in Figure 3a. In addition, there should be no error messages in the Message Window. If there are error messages, review the steps above to make sure you have made all connections properly, been precise about using the specified characters or text strings inside constructs and have not added any extra constructs.

The **Tester** Process is now complete. Save it by pressing the **Save-file** icon  on the Edit Toolbar.

Note: The semantics of MAGIC-C dictate the types of construct to which each Magic-C polygon (construct) can directly connect. For example a Task construct can connect directly to a Signal-Out polygon placed immediately below the Task construct. However a Task construct cannot connect directly to a Signal-In polygon in a Magic-C Finite State Machine. Enforcement of these semantic rules by the Virtio Innovator can be seen when one selects an already placed Magic-C construct and then clicks an icon on the Magic-C toolbar to place another construct. In such a case, the second construct will automatically be connected to the first if only if such a connection is permitted by the semantics of Magic-C.

Now go on to the next page to create the **DUT** Process.

Describing the DUT Process

As mentioned previously, the **DUT** Process depicted in Figure 4 synchronizes its output (the `go_more` signal) with the incoming clock represented by the `go` signal. Now let's go ahead and create it as follows:

1. Assuming that the **Tester** Process is still shown in the Design Window, return to the parent (the top level Block in this case) by left-clicking the **Open Parent**



icon on the Navigate Toolbar. The Navigate Toolbar  is located at the top of the Virtio Innovator screen. [If the Navigate Toolbar is not visible, first click **View** from the Main Menu Bar then select **Navigate Toolbar**.]

2. Double-click on the **DUT** Process construct to open it as a Process view (one level down in the hierarchy) in the Design Window.
3. Select **YES** when asked to create a new Process. The **DUT** Process page opens in the Design Window and is ready for editing.
4. The MAGIC-C Start construct is displayed and should already be selected. If it is not selected, select it by clicking on it.

5. Left-click on the **Add Task**  icon on the Magic-C Toolbar. The Task construct is automatically connected to the tail of the Start construct.

6. Place the following text in the text box of the Task construct:

```
x=0;
```

7. Left-click the **Add State**  icon on the Magic-C Toolbar to add a State construct and automatically connect it to the Task construct. Name this State construct `idle`.
8. Left-Click on the **Signal-In**  icon on the Magic-C Toolbar.
9. Type `go` in the text box. This will receive the `go` signal sent by the `Tester` Process.
10. While the MAGIC-C Signal-In box is selected, left-click on the **Add State**  icon on the Magic-C Toolbar to add a State construct.
11. Name this state `wait_for_clk` then select the entire State construct by left-clicking on it.
12. Click on the **Signal-In**  icon on the Magic-C Toolbar to add a Signal-In construct. Then enter `clk` in the text box of this new construct. By doing so you are allowing the `DUT` Process, while in the `wait_for_clk` state, to receive the `clk` signal from the `Tester` Process.
13. Left-click the **Signal-Out**  icon on the MAGIC-C Toolbar to add a new Signal-Out construct.
14. Now we will generate the signal called `go_more(x)` which will be received by the `Tester` Process. To do so select the text box inside the Signal-Out construct and enter the text `go_more(x)`.

Note: As explained above, the `go_more` signal has an argument called `x`. The variable `x` is the payload accompanies the signal `go_more`. The variable `x` is local to the `DUT` Process. However when this signal is received in the `Tester` Process the payload will be assigned to variable `x` that is local to `Tester`.

15. Declare `x` in the Declaration construct of the `DUT` Process by typing `int x;` in the text box of this construct.
16. Select the Signal-Out construct and left-click the **Task**  icon on the MAGIC-C Toolbar.
17. Type `x+=1` inside the Task construct to increment variable `x`.
18. Now connect a State construct to the Task construct by selecting the Task construct and clicking the State icon  on the Magic-C Toolbar. Then type `idle` as the name inside the newly created State construct. This name, since it is identical to the name of another, previously defined construct, refers to that previously defined construct. The result is that the FSM enters the previously defined state (which in Figure 4 is immediately above the `go` Signal-In construct). This approach can be used in place of loops to return to a previously defined state, thereby removing visual clutter from the prototype.

19. To save all the open documents click **File** on the Main Menu Bar and select **Save All** or simply left-click the **Save-all** icon  on the Edit Toolbar. Once again no errors should be listed in the Message Window if the Process was created correctly.

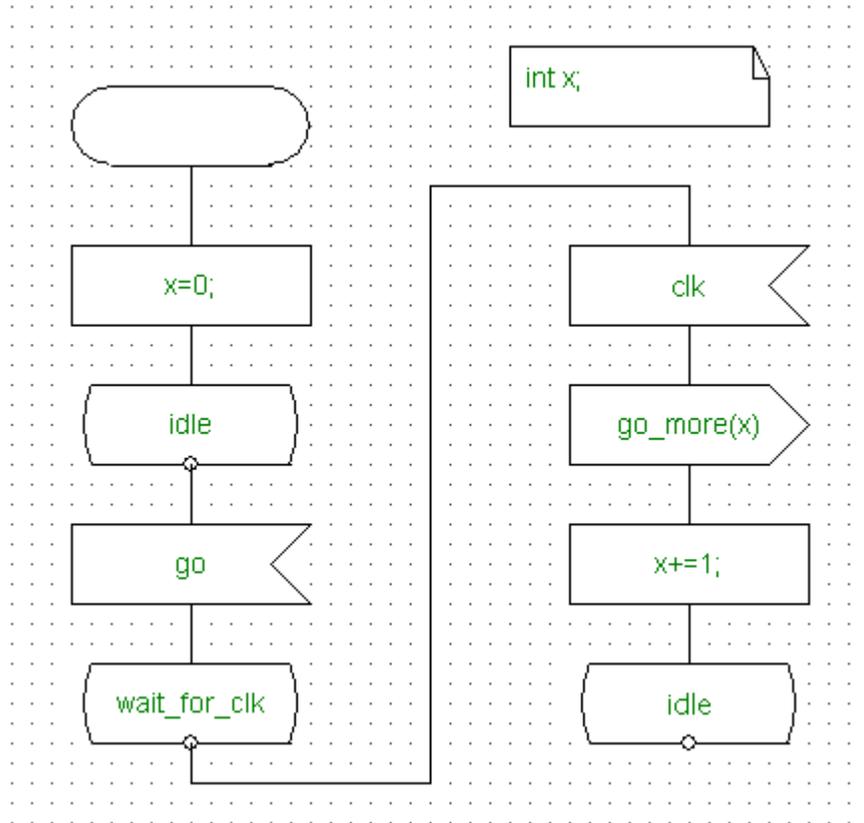


Figure 4: The FSM behavior of the DUT Process

Browsing Design Information

The Virtio Innovator provides three browsers that are used to gather information about a prototype. One of these at a time can be displayed in the Browser Window. If this Window is not visible, click **View** on the Main Menu Bar and select **Browser Window**. These are:

- [Design Browser](#)
- [Signal Browser](#)
- [Connectivity Browser](#)

Each of these browsers is explained below.

Design Browser

To gather information about the design Blocks, Processes and States, use the Design Browser Window. To display it, click on the  **Design** tab beneath the Browser Window to the left of the Design Window.

The Design Browser shows the hierarchy of a prototype and the states within every Process. As shown in Figure 5a, a Block (for example `tutorial`) is depicted using a Block  icon. A Process (for example `DUT`) is shown with a Process  icon. Each state of a Process is listed with next to a State  icon. To display the implementation of any item listed in the Design Browser, double-click on any icon and the resulting object will be highlighted in the Design Window. To expand and collapse the prototype hierarchy depicted in the Design Browser, use + and – icons respectively.

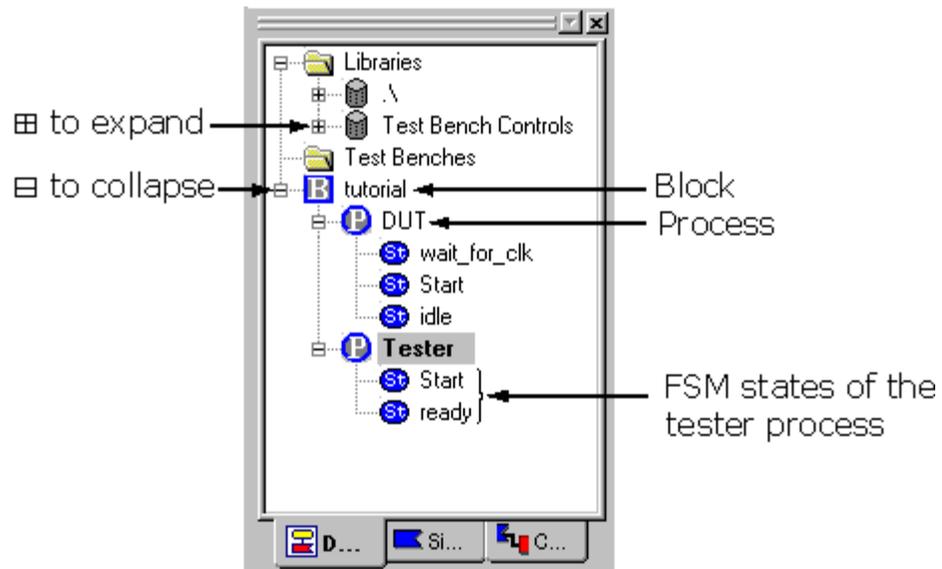


Figure 5a: Using The Design Browser

Signal Browser

To view **MAGIC-C** signals in the design, click the  **Signals** tab at the base of the Browser Window. The Signal Browser window opens, as depicted in Figure 5b.

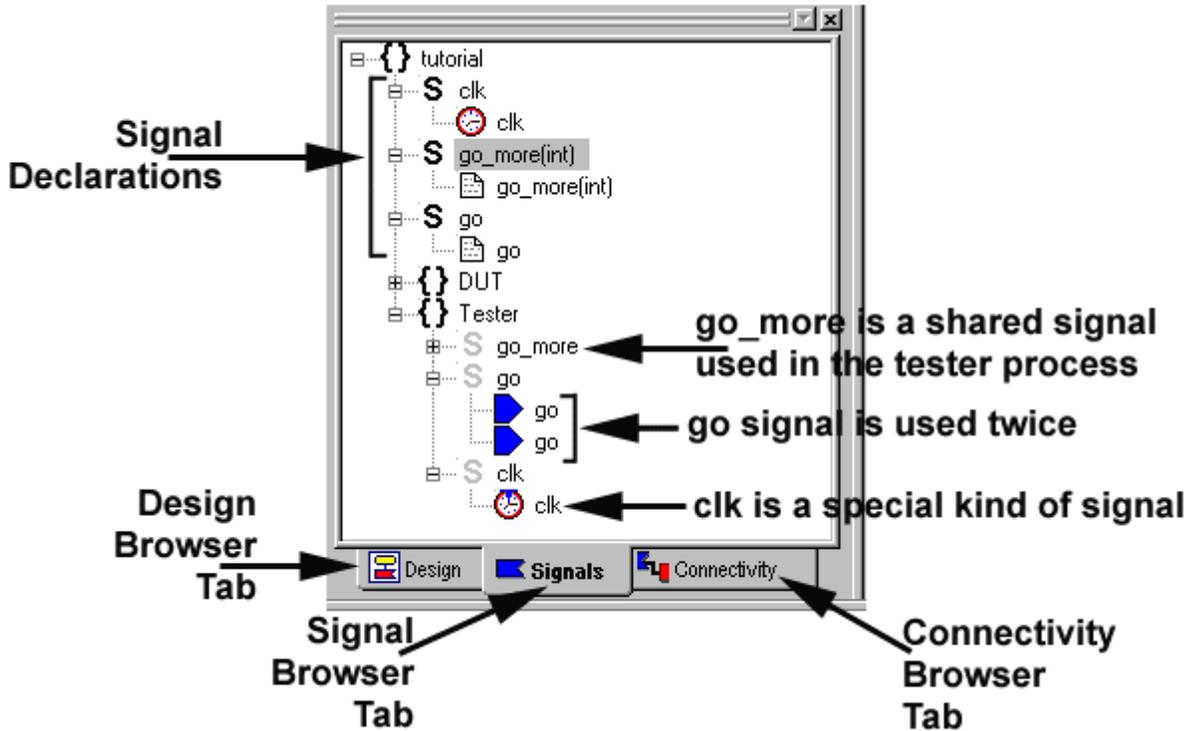


Figure 5b: The Signal Browser

The Signal browser shows where each signal has been declared in a prototype, thereby indicating the *scope* of a signal in the design hierarchy. It also shows where each signal has been used or consumed in a design. For example, in the `tutorial` design, `clk` is a special signal, marked by the MAGIC-C Clock icon that has been used in the `Tester` process. Signal `go` has been broadcast twice via the MAGIC-C Signal-Out construct, as indicated by the icon. To highlight in the Design Window the use of any signal, double-click on an icon in the Signal Browser.

Connectivity Browser

Detailed connectivity information about a signal can be requested in the [Signal Browser](#). The results of such a request are provided in the [Connectivity Browser](#).

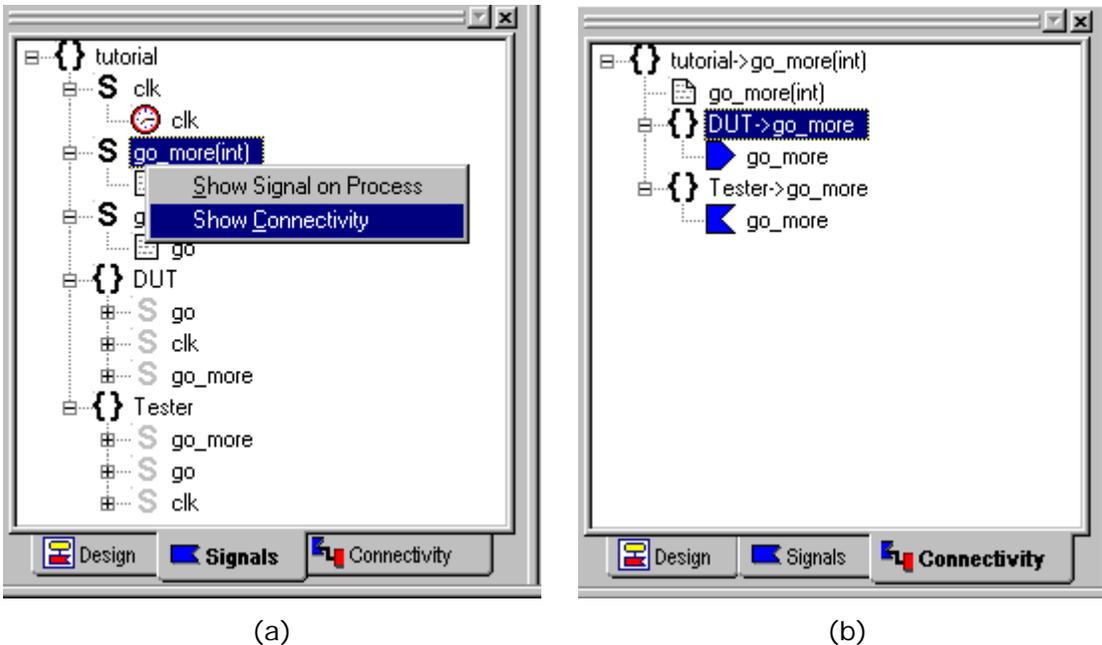


Figure 5: (c) Request Information About the `go_more` Signal; (d) Connectivity Browser Provides Query Results

For example, to find connectivity information for the `go_more` signal:

1. Right-click on the `go_more` declaration in the [Signal Browser](#) to display a pull-down menu.
2. Select **Show Connectivity** as shown in Figure 5(c).
3. The [Connectivity Browser](#) is displayed as depicted in Figure 5(d) and shows the following:
 - A. **Signal scope:** The scope of the `go_more` signal. It is visible in the `tutorial`, `DUT` and `Tester` modules.
 - B. **Signal naming:** The local name of the signal in a scope (`-><local_name>`). In this example, the signal is known as `go_more` in all scopes.
 - C. **Signal direction:** The direction of a signal. Observe that `go_more` is broadcast from the `DUT` process (as indicated by the  icon) and consumed in the `Tester` process (note the  icon).

Note: When using Symbols a signal might be known by different names inside a Symbol. See the [Virtio Innovator User's Manual](#) for further details.

Setting the Environment for Compilation and Linking

To instruct the Virtio Innovator to use the [Microsoft Visual C++ Compiler](#):

1. If Visual C++ is already installed, the Innovator reads its location and that of associated directories directly from the Windows® registry when the Innovator is first installed. If Visual C++ is not installed, install it now.
2. Verify that you have Service Pack 3 or later for Visual C++. To do so it may be necessary to check the version number of individual files as explained on the System Requirements page (in the Overview & Installation chapter) of the *Virtio Innovator User's Manual*.
3. In the Virtio Innovator, click **Settings** on the Main Menu Bar and choose **Directories**. The dialog box depicted in Figure 6 will appear.

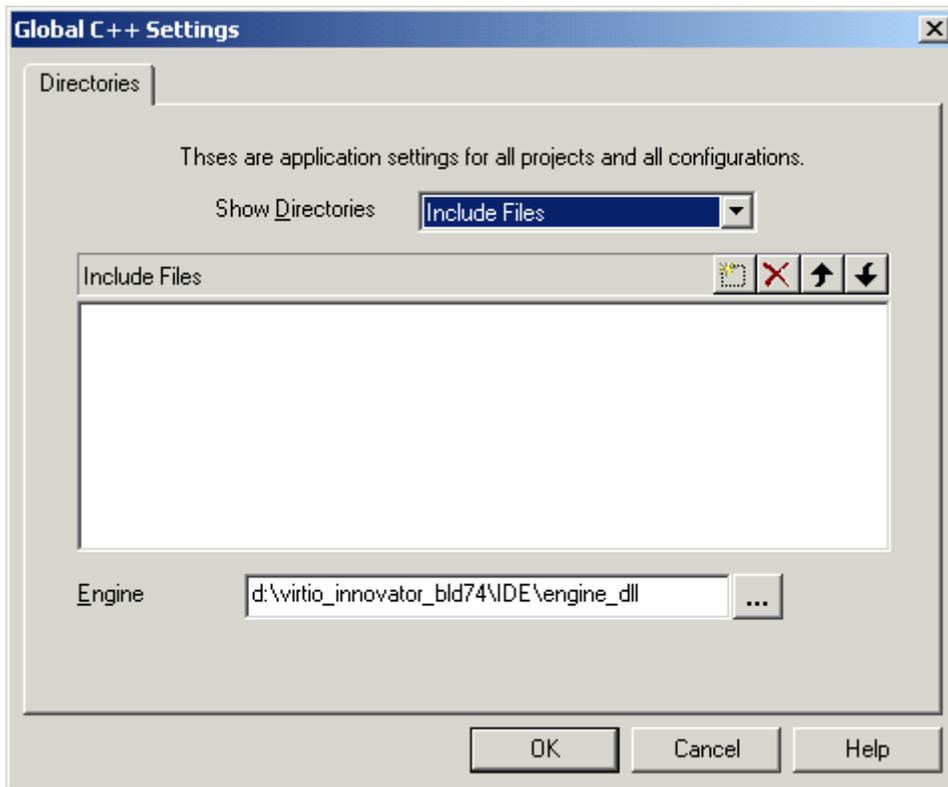


Figure 6: The Global C++ Settings Dialog Box

4. Verify that correct directories for Visual C++ Include Files, Library Files and Executable Files have been specified. To do so:
 - a. Click the  icon to the right of the **Show Directories:** text to display a pull-down menu of directories. Select the Include Files directory.
 - b. Check the path(s) specified in the **Include Files** box. To edit the path double-click on it and manually enter a new path or click the browse button  to the far right of the selection rectangle. One or more new paths can also be entered by clicking the New (Insert) icon .
 - c. Repeat steps 3a and 3b above for each of the other two directories (Library Files and Executable Files).

Note: The paths to the Include Files, Library Files and Executable Files directories are taken from the Windows registry when the Virtio Innovator is first installed. This assumes that Visual C++ was loaded onto your machine prior to installing the Virtio Innovator.

5. Verify the path to the Virtio Innovator by checking the text to the right of the **Engine** text. This should point to the `<virtio_innovator_install_path>\IDE\engine_dll` where `<virtio_innovator_install_path>` is the absolute path chosen during Virtio Innovator installation.

Note: Do not change the engine directory unless the location of the prototyping engine is changed after installation of the Virtio Innovator. Upon installation of the Virtio Innovator, the location of the engine files is saved in the registry. When the Virtio Innovator is invoked for the first time, it reads the location of the engine from the registry. The next time the Innovator is opened it “remembers” the last location of the engine files.

Setting the Project Specific Environment

To set the environment for a specific design:

1. Click **Settings** on the Main Menu Bar and choose **Project Settings**. The dialog box depicted in Figure 7 will appear.

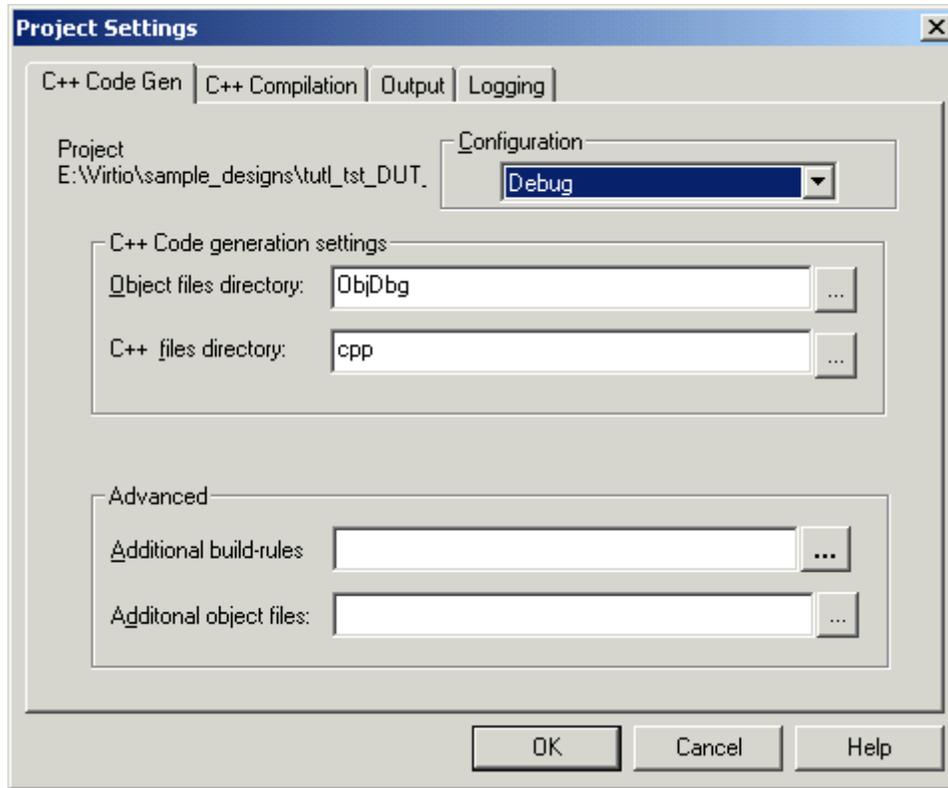


Figure 7: The Project Settings C++ Code Generation Dialog Box

2. Click on the **C++ Code Gen** tab.
3. Verify that the **Configuration** is set to **Debug** and that **C++ Code Generation Settings** has **Object files** set to **ObjDbg** and **C++ files** set to **cpp**.
4. Click on the **C++ Compilation** tab and, when prompted about creation of **ObjDbg** and **cpp** directories, click on **Yes**.
5. Verify that the **Configuration** is set to **Debug**, and check that the Compiler arguments on the **C++ Compilation** page are set to:


```
-c -nologo -I . -I "$(CPP_DIR)" -I "$(ENGINE)" -MTd -zi
```
6. Click on the **Output** tab and verify that the **Command Interpreter**, **Trace State Changes** and **Use Smart Engine** options are all checked. Also make sure that the **Configuration** is set to **Debug**.
6. Click on the **Output** tab and verify that the **Configuration** is set to **Debug**.
7. Click **OK**.

C++ Code Generation and Compilation

Now that the prototype is complete, it will need to be compiled and linked in order to generate an executable file. To generate such a file, click the  icon on the Build/Run



Toolbar. The Build/Run Toolbar is located at the top of the Virtio Innovator screen. [If the Build/Run Toolbar is not visible, first click **View** from the Main Menu Bar then select **Build Toolbar**.]

While the prototype is being compiled, the Message Window below the Design Window displays the progress of compilation and linking. If errors occur during code generation and compilation they are sent to this Window. In most cases, double-clicking on a message in the Message Window will display the offending statement of the prototype in the Design Window.

Note: If the Message Window is not visible below the Design Window, click **View** on the Main Menu Bar and select **Message Window** if it does not have a check mark next to it.

Debugging the Design

After the design has been successfully compiled it is debugged. When doing so one can:

- Single Step a prototyping session.
- Set Breakpoints to debug a prototype.

Each of these topics is covered below.

Single Stepping During a Prototyping Session

The simplest mode of debugging is single-stepping. Let's first consider how to single-step a prototyping session and how to stop one. Then we'll go on to a live example using the design created earlier.

After compilation one can single-step a prototyping session by:

- Clicking the **Single-Step**  icon on the Build/Run Toolbar.
- or*
- Pressing **F10** on the keyboard.
- or*
- Clicking **Run** on the Main Menu Bar and selecting **Debug Step** from the pull-down menu that appears.

After the first single-step operation, a Prototype Cockpit window will open. The Prototype Cockpit window will contain any Test Bench(es) that were defined as well as a message pane. Also, when the prototyping session is single-stepped in any one of these ways, the session will advance a single Magic-C construct. Continuing to click the **Single-Step** icon (or press **F10**) advances the prototyping session.

To terminate a running prototyping session:

- Click the **Abort Prototyping Session**  icon on the Build/Run Toolbar.
or
- Pressing **Shift-F5** on the keyboard.
or
- Clicking **Run** on the Main Menu Bar and selecting **Debug Stop** from the pull-down menu that appears.

Now let's walk through single-stepping the prototype created on earlier pages of this tutorial. To do so:

- A. Display the main Block by clicking on the tab at the bottom of the Design Window labeled `tutorial.blk`. The `Tester` and `DUT` Processes will appear in the Design Window along with a single Declaration construct.
- B. Click the **Single-Step**  icon on the Build/Run Toolbar once. The `DUT` Process implementation will appear in the Design Window. The Prototype Cockpit window opens but is immediately minimized. The Waveform Viewer window is not displayed by default. The settings for these latter two windows are not a problem since we will not need them now.
- C. Click the **Single-Step** icon on the Build/Run Toolbar two more times for a total of three times. Note that the `idle` state of the `DUT` Process has been reached. This is the case because the `DUT` Process was added to the `tutorial` Block first.

Note: The next MAGIC-C construct to be scheduled for execution is shown by the  icon on the design. The current state of a Finite State Machine is pointed to by the  icon.

- D. Click the **Single-Step** icon on the Build/Run Toolbar one more (the fourth) time. The prototype returns to the Start state of the `Tester` Process because the `DUT` Process is waiting for the `go` signal which must be generated by `Tester`.
- E. Continue clicking the **Single-Step** icon on the Build/Run Toolbar. On the fifth single-step the period of the clock, `clk`, is set to 1 time unit. On the sixth single-step the `go` signal is broadcast for the first time. And on the seventh single-step the `ready` state of the `DUT` Process is reached.
- F. Click the **Abort Prototyping Session**  icon on the Build/Run Toolbar to terminate the session.

Setting Breakpoints in a Prototype

To investigate specific portions of a design it is useful to set breakpoints. In the current design, for example, the two Processes `Tester` and `DUT` perform two-way handshaking.

As mentioned earlier, Process `Tester` initiates communication by sending the signal `go` and Process `DUT` responds by sending the signal `go_more`. To see this functionality:

1. Increase the viewing area of the `Design Window` by closing the `Browser Window` as follows: Click **View** on the Main Menu Bar and select **Browser Window**. To increase the viewing area further, close the `Message Window` by clicking **View** on the Main Menu Bar and selecting **Message Window**.
2. Display the two Processes `Tester` and `DUT` adjacent to one another in the `Design Window` by clicking **Window** on the Main Menu Bar and selecting **Tile Vertically**. The `Tester` Process, `DUT` Process and `tutorial` Block should all appear in the `Design Window`.
3. Since it is not currently needed, close the `tutorial` Block by clicking the lower of the two close window icons  in the upper right corner of the Virtio Innovator screen. The upper of these two icons closes the Virtio Innovator application.
4. Rearrange the two remaining panes in the `Design Window` by resizing them so that together they occupy the full screen.
5. Because the `DUT` Process awaits the `go` signal before communicating, put a breakpoint in the Signal-In construct containing `go`. To do so make the `DUT` Process the in-focus window, right-click on the Signal-In construct containing `go` and select **Toggle Breakpoint on this context** from the pull-down menu that appears. This is depicted in Figure 8.

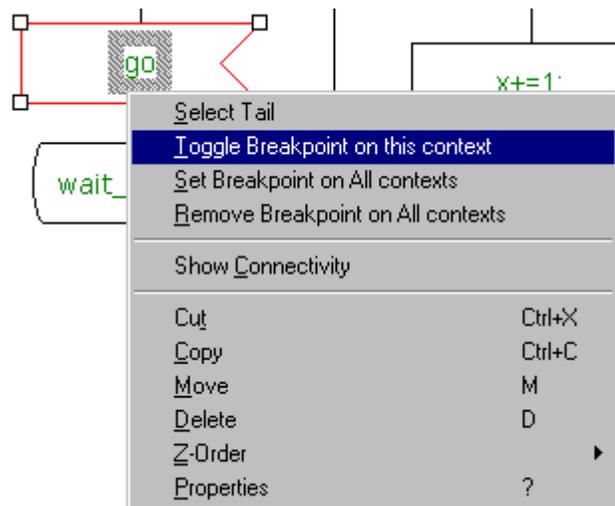


Figure 8: Setting A Breakpoint

6. Similarly, set a breakpoint on the Signal-In construct containing `go_more(a)` in the `Tester` Process. The `Design Window` containing both Processes now looks like the one in Figure 9.

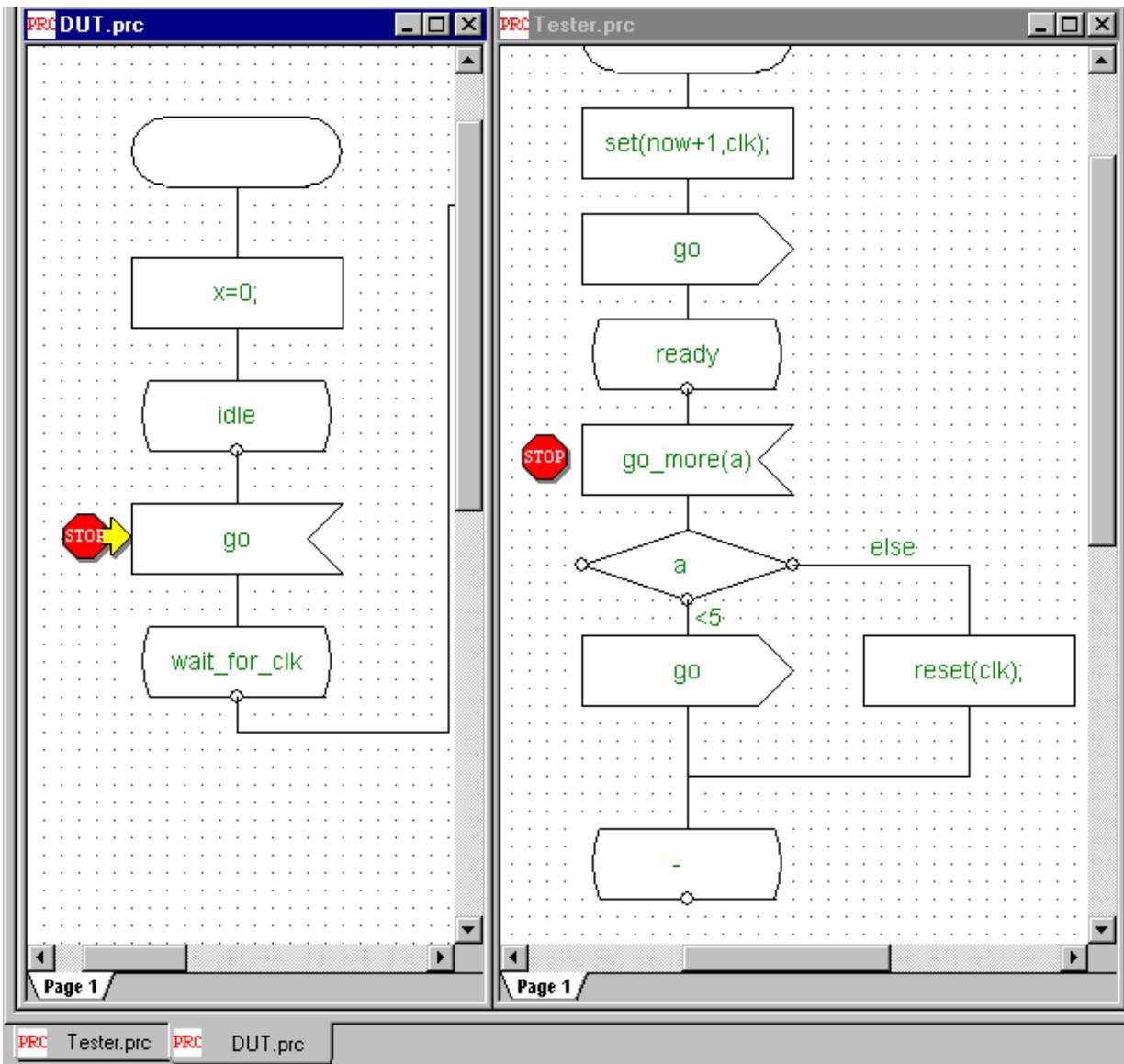


Figure 9: The Location of Breakpoints

7. Start execution by clicking the **Go**  icon on the Build/Run Toolbar or by pressing **F5** on the keyboard.
8. Minimize the Waveform Viewer and Prototype Cockpit windows. We will not need them.
9. Note that execution has stopped in the **DUT** Process on receipt of signal **go** in the Signal-In construct. This point is marked by the **Stop**  icon . This **go** signal was transmitted at initialization by the **Tester** Process just after the Start state.

10. Click the **Go**  icon again to continue execution. Note that execution stops in the Tester Process on receipt of signal `go_more`. The DUT Process transmitted this signal just after it had received a `go` signal.

11. Click the **Go**  icon repeatedly to transfer control of the prototyping session to each Process in turn, thereby obeying the handshake mechanism. Note that this signal exchange continues for six repetitions. After that point Tester stops sending the `go` signal, thus terminating two-way communication.

12. To terminate the prototyping session, click the **Abort Prototyping Session**



icon on the Build/Run Toolbar.

13. If the Waveform Viewer window is open or minimized, close it by displaying it then clicking the  icon in the upper right corner.

14. Remove any breakpoints by repeating step 5 above for each of the two breakpoints.

Note: Breakpoints can be set both **before** running and while running a prototyping session. For the latter case a breakpoint can be toggled whenever the session is not running. Also, single-stepping operations can be mixed with breakpoints for better debugging granularity. The combination of these tools ensures easy and dynamically-variable investigation of any prototype section.

Tracing Signal Values

There are several ways to inspect the values of **MAGIC-C** signals, variables and state transitions. These techniques include the following:

- Using MAGIC-C functions like `vs_printf` to trace design activity.
- Tracing **FSM** state changes in the Message Window.
- Using advanced Test Bench controls (this will be covered on the "Creating a Test Bench" and "Running a Test Bench" pages) to control and observe a prototyping session.
- Tracing of signal and variable value changes interactively while a session is running.
- Post-processing of signal and variable values from a prototyping session run.

To illustrate how to monitor signal values, the `tutorial` design is reused, as shown in Figure 9 below. In this prototype there are three signals: `go`, `go_more`, and a special clock called `clk`. On the next two pages of this tutorial the Waveform viewer will be used during a prototyping session (on the "Using the Waveform Viewer During a Prototyping Session" page) and after the session has terminated (on the "Using the Waveform Viewer for Post-Processing Analysis" page).

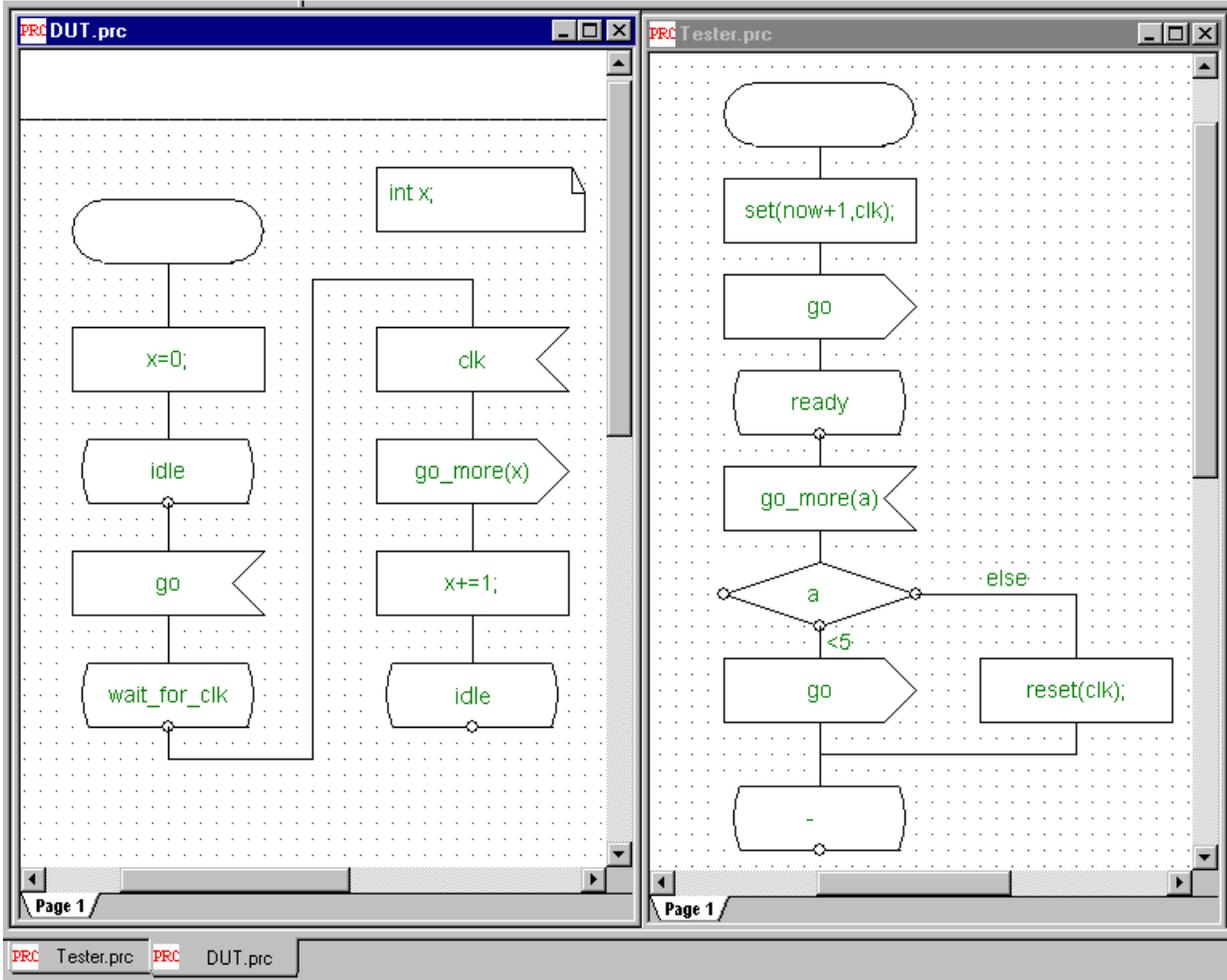


Figure 9: Behavior of the Tutorial Example

Using the Waveform Viewer During a Prototyping Session

To view signal value changes in the *Waveform Viewer*:

1. Click **Settings** on the Main Menu Bar and select **Project Settings**.
2. Click the **Logging** tab. The dialog box depicted in Figure 10 will appear.

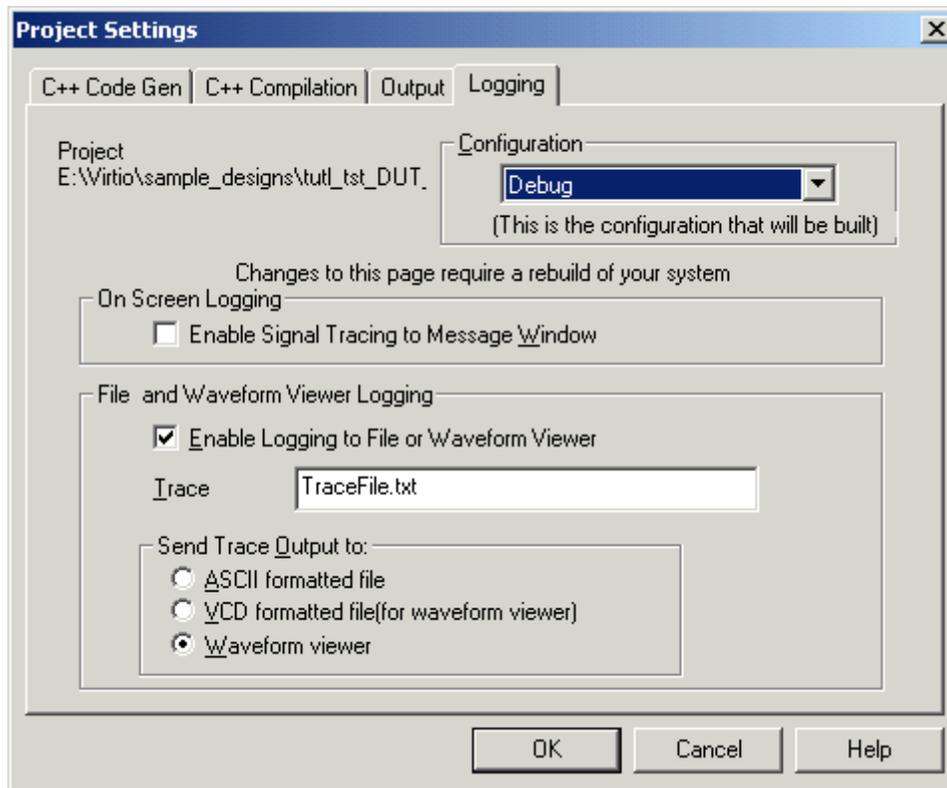


Figure 10: The Project Settings Logging Tab Dialog Box

3. Verify that the *Configuration* is set to *Debug*.
4. Activate the *Enable Logging to File or Waveform Viewer* option by adding a checkmark in the box.
5. For the *Send Trace Output to* option, select the *Waveform Viewer* radio button.
6. Click **OK**.

7. Recompile the design by clicking the *Compile*  icon on the Build/Run Toolbar.
8. Replace the two breakpoints that were set then removed earlier. See [Setting Breakpoints](#) to do so.

9. To start tracing signals, launch the prototyping session by:

Clicking the *Go Until*  icon on the Build/Run Toolbar.

or

Clicking Run on the Main Menu Bar and selecting *Debug Go Until*.

8. When a pop-up window opens, type **1** in the text box to indicate that the session should run for 1 time unit. The Waveform Viewer will open and its content will be as depicted in Figure 11a.

Note: If the Prototype Cockpit window (with a Messages sub-window inside it) opens in front of the Waveform Viewer, simply minimize the Prototype Cockpit window.

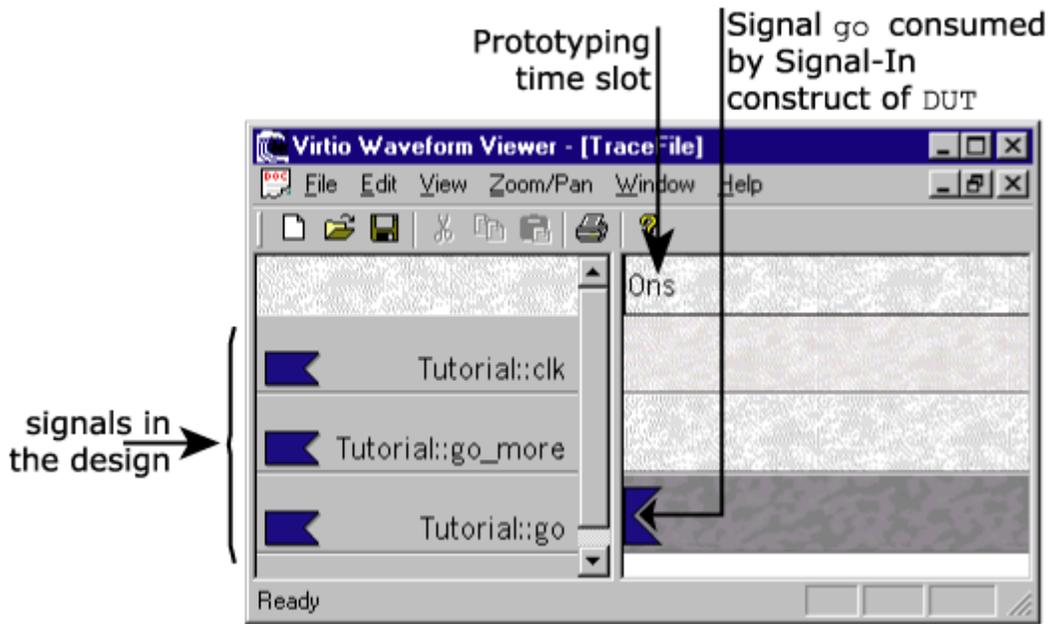


Figure 11a: The Virtio Waveform Viewer

In Figure 11a, the left pane of the Waveform Viewer window lists signals that are currently being traced. On the top row of the right pane, the current prototyping session time is displayed. For example, in Figure 11a the session has stopped at zero time units. All signal activity is recorded in the right pane.

Note: By default, all signals are traced. Signals that needn't be traced can be specified by clicking **Settings** on the Main Menu Bar and choosing **Signal Logging**. This is described in the *Virtio Innovator User's Manual*.

The **Signal-In** icon in the right pane of the Waveform Viewer represents a signal which has been received by a Process. For example Figure 11a shows signal `go` was received by the DUT Signal-In construct during the zero time units interval. It can be inferred from a knowledge of the design that signal `go` was transmitted at initialization by the `Tester` Process just after its `Start` state. The DUT process received it for the first time just after its `idle` state. To check this assertion, use debugging techniques like single-stepping. (see [Debugging the Design](#)).

8. Continue the prototyping session by clicking the **Go Until** icon  and entering 1 in the dialog box to tell the session to advance 1 time unit. Then make the Waveform Viewer the in-focus window. The Waveform Viewer content will be as depicted in Figure 11b, in which clock signal `clk` is triggered once indicating that another time slot has started (clock signals are special because they mark prototyping session time and advance a Finite State Machine during a prototyping session). Also visible in Figure 11b is the DUT process having transmitted the signal `go_more` and this signal having been received by the Signal-In construct in Process `Tester`. As noted above, the signal activity for `go_more` is displayed in the Waveform Viewer by a Signal-In icon corresponding to the `Tutorial::go_more` signal.

Note: Signals are always shown in the Waveform Viewer when they are consumed, not when they are generated.

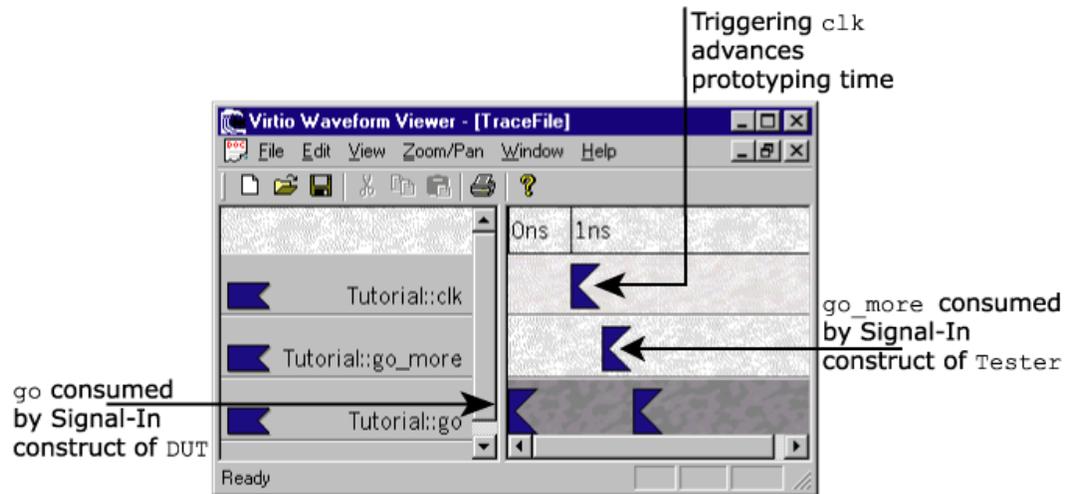


Figure 11b: Signal Values After Time Zero

10. Stop the prototyping session by clicking the **Abort Prototyping Session**



icon on the Build/Run Toolbar. The Prototype Cockpit window will close automatically.

11. Stop the Waveform Viewer by:

Clicking **File** on the **Waveform Viewer** Main Menu Bar and selecting **Exit**.

or

Clicking the **Close**  icon on the Waveform Viewer window.

Using the Waveform Viewer for Post-processing Analysis

The Waveform Viewer can also be used for post-processing analysis after the prototyping session has terminated. This feature might be useful, for example, to store results of a particularly long run and analyze them later with the Waveform Viewer. Set up this tracing mode as follows:

- A. Click **Settings** on the Main Menu Bar and select **Project Settings**.
- B. Click on the **Logging** tab and check activate **Enable Logging to File or Waveform Viewer** by adding a checkmark in the box.
- C. Set **Trace** in the **File and Waveform Viewer Logging** section to `tutorial.vcd`.
- D. For the **Send Trace Output to:** option, click the **VCD formatted file (for waveform viewer)** radio button (VCD = Value Change Dump) .
- E. Click on OK.

- F. Recompile the prototype by clicking on the **Compile**  icon on the Build/Run Toolbar.

The setting changes made above tell the Virtio Innovator to output prototyping session results in a **VCD** text file called `tutorial.vcd`. After the prototyping session is over the Waveform Viewer can be used to analyze the content of the VCD file. The procedure below is an example of how this is done.

1. Verify that both breakpoints have been removed by checking both the **Tester** Process and the **DUT** Process implementations. If not, remove them.
2. Start the prototyping session by clicking on the **Go Until**  icon or by clicking **Run** on the Main Menu Bar and selecting **Debug Go Until**. When a pop-up window opens, type `10` in the text box to indicate that the session should be run for 10 time units (there is no activity after 6 time units.)
3. Abort the prototyping session, then invoke the Waveform Viewer by clicking the **Waveform Viewer**  icon on the Edit Toolbar.
4. Load the VCD file `tutorial.vcd` by clicking **File** on the Main Menu Bar of the Waveform Viewer and selecting **Open**. The Waveform Viewer displays the prototyping session results as shown in Figure 12.
5. Save and close the prototype.

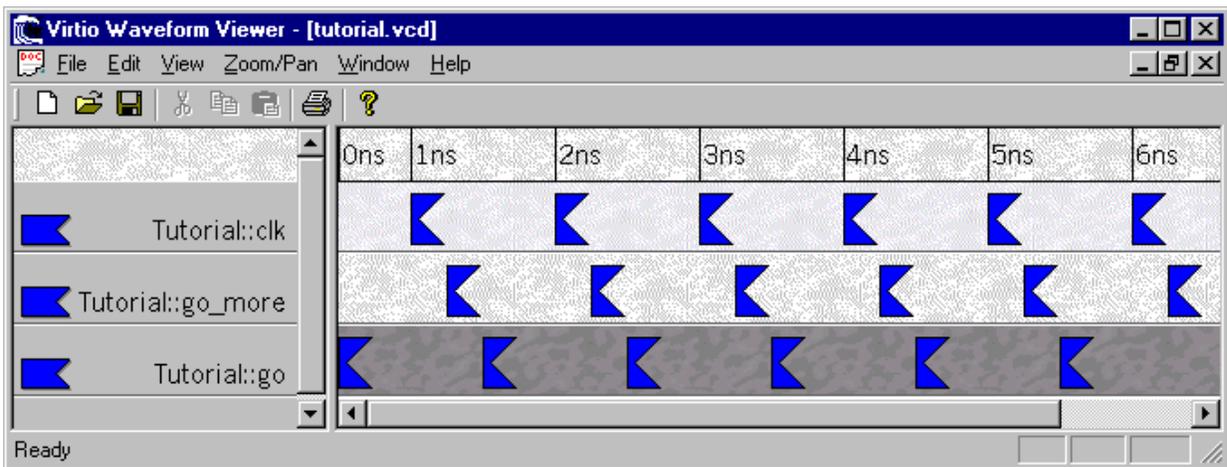


Figure 12: Viewing a VCD File

The Test Bench Concept

As discussed in the [Tracing Signal Values](#) section, the Process `Tester` injects `go` and `clock` signals into the `DUT` process. While this approach may be sufficient for simple designs, a more comprehensive manner of prototyping and exercising a device interface is needed. For example the following features are highly desirable:

- **Interactive Signal Injection and Debugging:** Especially in the early stages of a design, it is useful to inject signals interactively and determine their

effects by monitoring portions of the design. The interface providing this capability should also permit monitoring and modification of prototype variables.

- **Modeling a System User Interface:** Often completed designs support a user interface for human interaction. One example is a wireless phone with buttons for dialing, an LCD panel, and an LED array. A graphical interface permitting addition and control of input/output devices (such as an LCD and buttons) would be ideal for providing this capability. This "Test Bench" could then be used to manipulate the virtual prototype in the same way that the target system will eventually be manipulated by a user.

Both of these features are provided by a Virtio Innovator Test Bench. A Test Bench acts as a graphical user interface while a prototyping session is running. It is created by adding Test Bench "controls" such as Signal Buttons, LEDs, LCDs, a keyboard, etc. in the Test Bench Builder provided as part of the Virtio Innovator. As an example of its use, the [tutorial](#) design described early will be used to illustrate creation of a Test Bench.

Note: For more information about Test Bench controls refer to the *Virtio Innovator User's Manual*.

Recall that in the design, Process `Tester` (see [Figure 3a](#)) broadcasts a `go` signal to initiate and continue communication with Process `DUT` (See [Figure 4](#)). As a first step, transmission of the `go` signal in the `Tester` process is replaced with a test bench control that injects this signal into the prototype.

The value of local integer variable `a` in the `Tester` Process will also be monitored using the same Test Bench. Note that the payload of the `go_more` signal updates this variable. The value of `a` indicates the total number of handshakes that have occurred between the two Processes.

Modify the Prototype for Use with a Test Bench

To remove broadcasting of the `go` signal by Process `Tester`:

1. Verify that the prototype has been saved and closed before starting this procedure.
2. Using Windows Explorer, create a new directory, for example, `c:\tutorial_tb`, outside the directory containing the `tutorial` design. This new directory will be used to create a version of the `tutorial` prototype with a Test Bench.
3. Copy the following files from the `tutorial` design directory into the new directory (for example, `c:\tutorial_tb`) created in the previous step: `DUT.prc`, `Tester.prc`, `tutorial.blk` and `tutorial.vsp`. Note that it is not necessary to copy any sub-directories.
4. Using Windows Explorer, locate the new directory (for example, `c:\tutorial_tb`) and double-click on `tutorial.vsp`, the tutorial project file. The Virtio Innovator opens with the new copy of the `tutorial` project loaded.

5. Use the Signal Browser, as shown in Figure 13, to identify where in the `Tester` process signal `go` is broadcast. (For more information on the Signal Browser, review the [Signal Browser](#) section or refer to the *Virtio Innovator User's Manual*.) The intent is to remove the broadcasting of signal `go` from the `Tester` process.
6. To remove every broadcast of signal `go` from the `Tester` process, double-click on the icon marked with **(1)** in Figure 13 to highlight the Signal-Out construct. It appears after the `Start` state in the Design Window.

7. Click the **Delete**  icon on the Edit Toolbar to delete the Signal-Out construct.

Note: There are four ways to remove and discard a MAGIC-C construct once it has been selected: 1) Press the **Delete**  icon on the Edit Toolbar; 2) Press the **Delete** key on the keyboard; 3) Use the keyboard shortcut **Ctrl-D**; 4) Right click on the selected construct to display a pull-down menu and select **Delete**. There are three ways to cut a construct and place it on the Windows clipboard after selecting it: 1) Right-click and choose **Cut** from the pull-down menu; 2) Use the keyboard shortcut **Ctrl-X**; 3) Click **Edit** on the Main Menu Bar and select **Cut**. For more information, refer to the *Virtio Innovator User's Manual*.

8. Connect the Task construct (containing the statement `set(now+1, clk);`) and the `ready` State construct by first left clicking on the bubble at the bottom of the Task construct and then left-clicking at the top of the `ready` State construct.

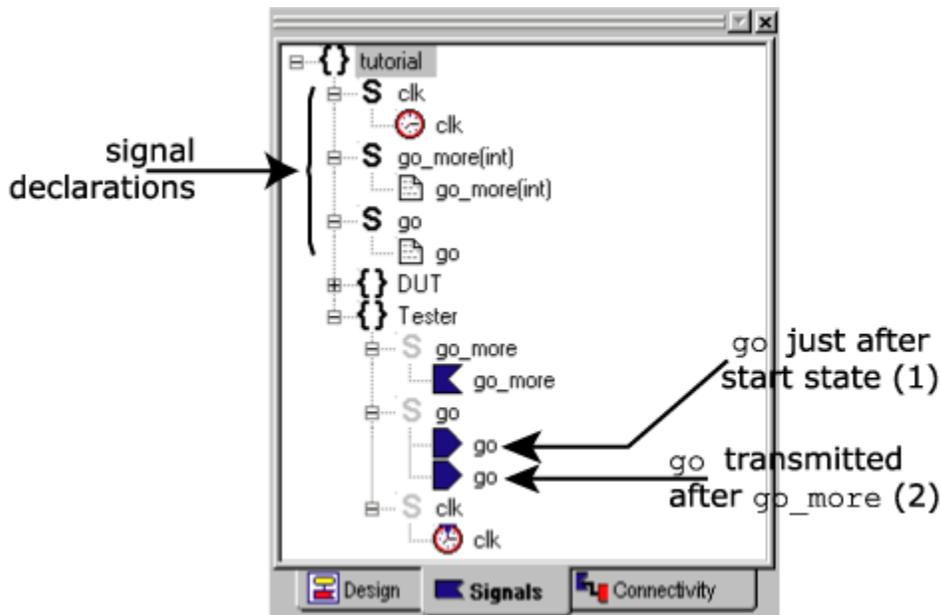


Figure 13: Remove The "go" Signal From The Tester Process

9. Remove the second broadcast of signal `go` (the icon marked by **(2)** in Figure 13) in the same manner as described above.

10. Connect the MAGIC-C Decision construct and the (-) State construct by double-clicking as explained above. Then type the condition `< 5` in the textbox corresponding to the connection.

11. In the Declaration box of the `Tester` process, replace the C keyword `int` with `VS_int`. By declaring variable `a` as type `VS_int` as shown in Figure 14, it is marked as an integer variable that can be monitored by a Test Bench. For more details on Virtio Innovator data types (such as `VS_float` and `VS_bool`), refer to the *Virtio Innovator User's Manual*.

12. Click on any position outside the Declaration box to deselect it. As a result of these changes, the implementation of Process `Tester` has now been modified to that depicted in Figure 14. It will no longer send the `go` signal and integer variable `a` can now be monitored from a Test Bench. Note that the `DUT` process remains unchanged.

13. Save the design.

14. Compile the modified prototype by clicking the  icon on the Build/Run Toolbar.

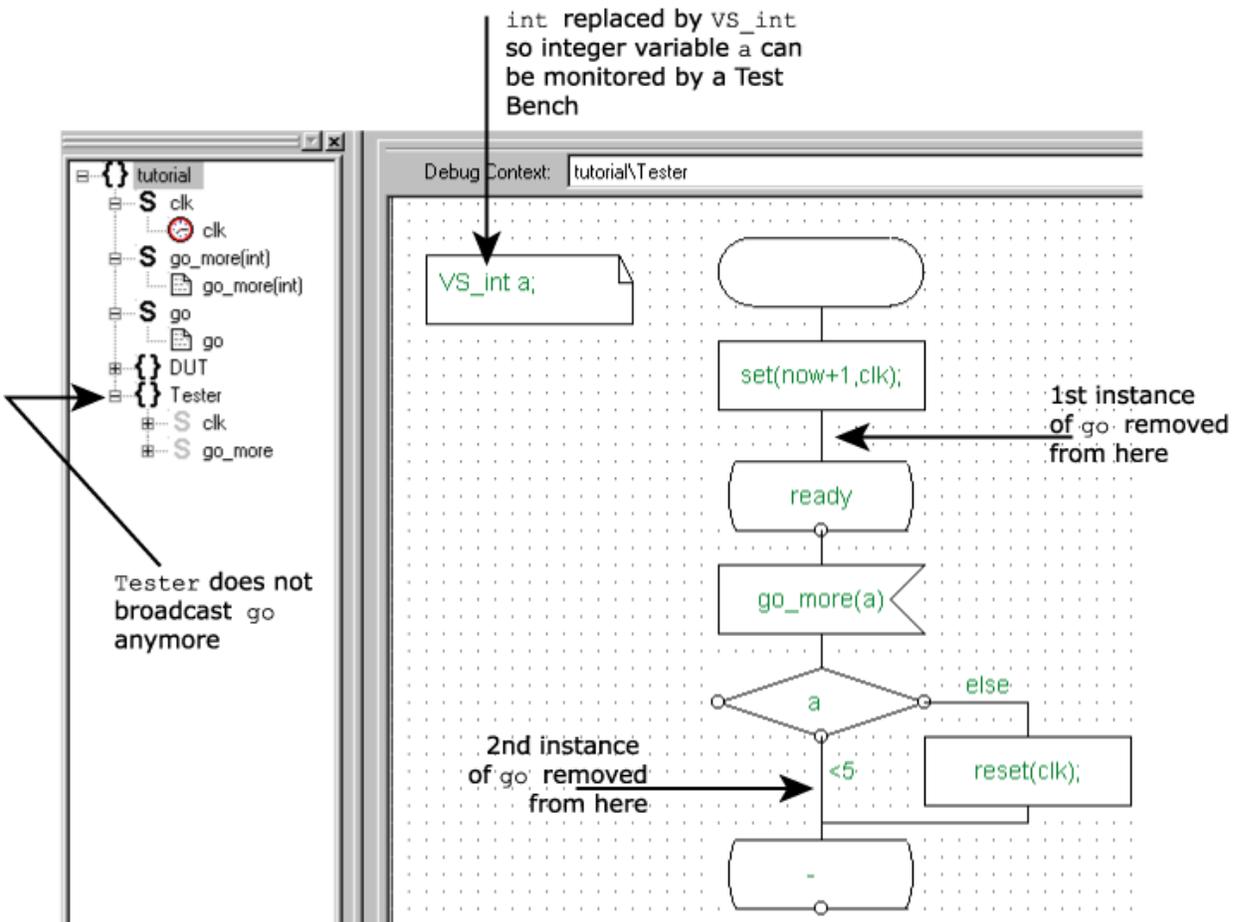


Figure 14: Implementation Of Process Tester For Use With A Test Bench

Creating a Test Bench

In this section, a Test Bench is created to replace the functionality of the Process `Tester`. The Test Bench will contain a Signal Button that sends signal `go` to the `DUT` process and a Register to view the current value of variable `a`. To create the Test Bench:

1. Open the Design Browser by clicking on the  tab at the base of the Browser Window. [If the Browser Window is not visible, first click **View** from the Main Menu Bar then select **Browser Window**.]
2. Right-click on the **Test Benches** caption in the Design Browser and select **Add Project Test Bench** from the pull-down menu as shown on the left half of Figure 15.
3. The dialog box shown in the right half of Figure 15 will appear. Name the Test Bench by typing the text `mytb` in the form as shown. Leave the **HTML Test bench** check box blank.

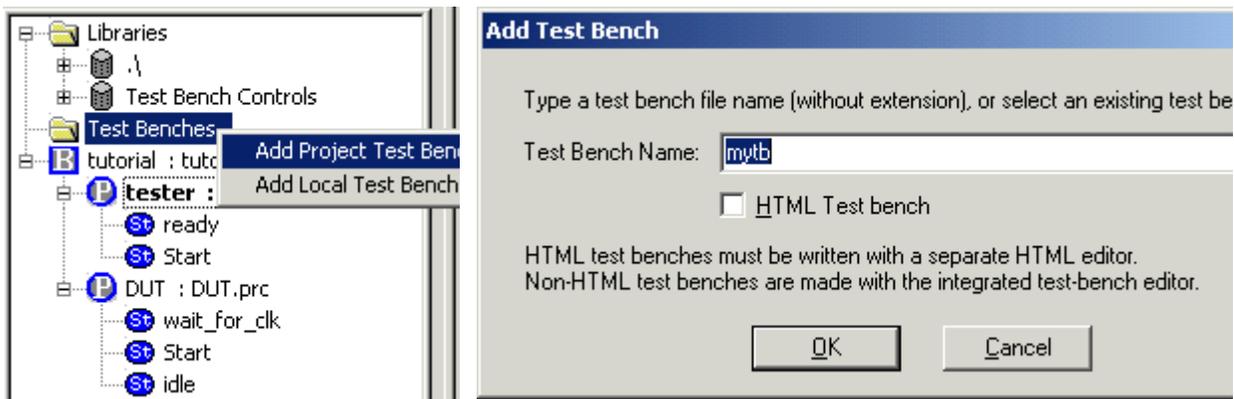


Figure 15: Create A New Test Bench

4. Click **OK**. A blank Design Window (the right pane) is displayed. It is here that the controls forming the Test Bench will be added. The steps below will lead you through adding a Signal Button Test Bench control labeled `send_go` that sends the `go` signal.



5. Locate the Test Bench Builder Toolbar at the top of the Virtio Innovator screen. If it is not visible, first click **View** from the Main Menu Bar then select **Test Bench Builder Toolbar**.
6. Add a Signal Button to the Test Bench in one of the following ways:

Click the Signal Button  icon on the Test Bench Builder Toolbar.

or

Click **Add** on the Main Menu Bar and select **Signal Button**.

or

Expand the Test Bench Controls library block in the Design Browser and double-click on the Signal Button entry.

Note that the mouse cursor converts to an **Add** icon.

7. Click anywhere in the Design Window to place the button. This opens the **Button Property Page** depicted in Figure 16a.

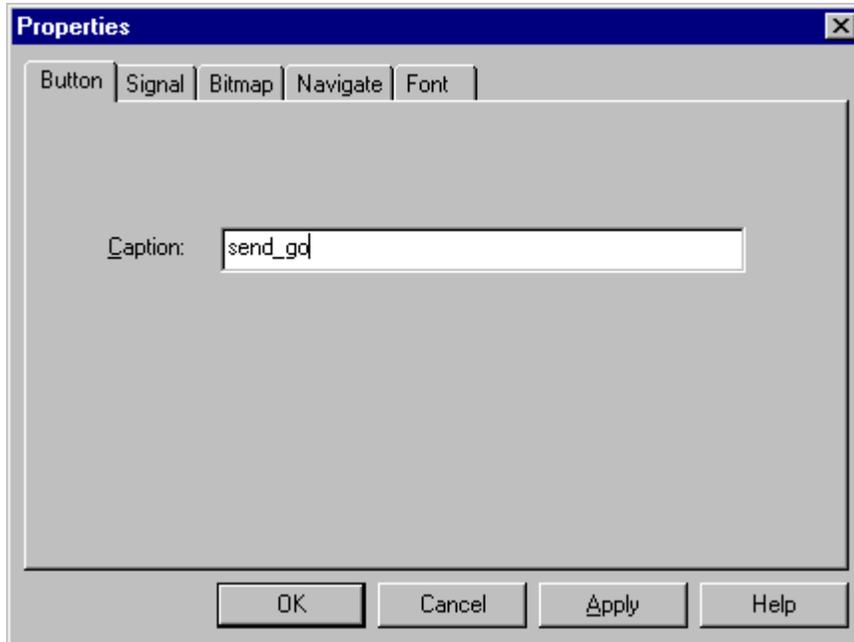


Figure 16a: Adding A Signal Button Test Bench Control

8. Type `send_go` in the **Caption:** box as shown in Figure 16a.
9. Click the **Signal** tab of the Button Property Page. The Signal Property Page depicted in Figure 16b will appear (without the drop down menu in the center containing three lines with the text `Tutorial::go`, `Tutorial::go_more` and `Tutorial::clk`).

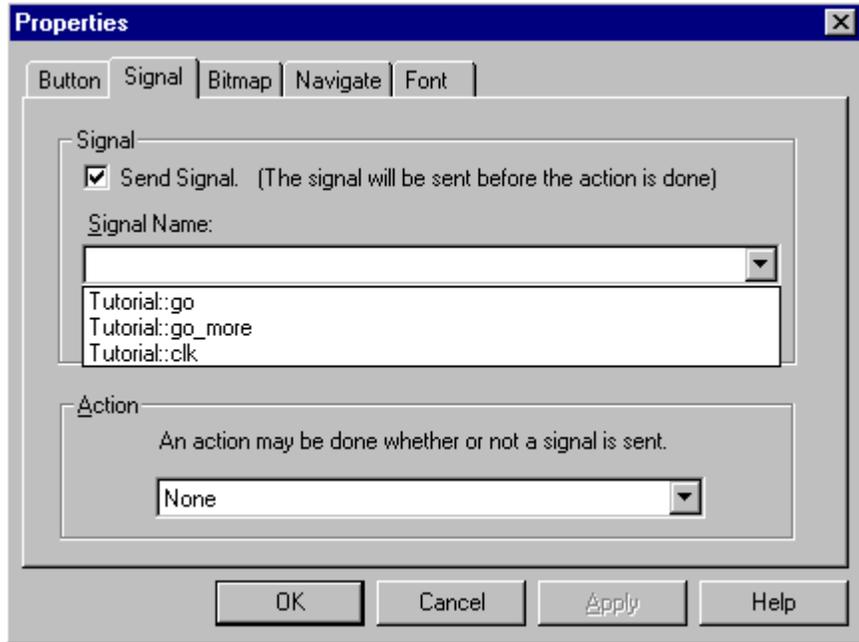


Figure 16b: Selecting A Signal To Inject Into The Prototype

10. Check the **Send Signal** check box to indicate that a signal is to be injected into the prototyping engine.
11. To select a specific signal to be injected, open the drop-down menu to the right of the **Signal Name:** box by clicking on the  icon at the right edge of this box. This menu, which is shown in the center of Figure 16b, lists all signals in the prototype that can be injected during a prototyping session. This menu should list `Tutorial::go`, `Tutorial::go_more` and `Tutorial::clk`.
12. Select the `Tutorial::go` signal from the menu.
13. Verify that **Payload Size** is set to **None** since signal `go` does not have an associated payload. [This option is covered by the pull-down menu in Figure 16b.]
14. An action, such as Run, Single-Step, Pause, etc. can be associated with clicking of the `send_go` button. In this case set the **Action** to **Run**. This will cause the prototyping engine to commence or resume execution when the `send_go` button is pressed while a prototyping session is running.
15. A bitmap can be superimposed on the button using the **Bitmap** tab, another Test Bench can be run using the **Navigate** tab and a custom font can be selected using the **Font** tab. For now, however, it is not necessary to use these other features so click **OK**. A button with the caption `send_go` appears in the Design Window. The location of this button can be changed by dragging it to the desired position.

Now we will add a Register Test Bench control so the value of integer variable `a` in the `Tester` process can be viewed as it changes. Recall that variable `a` is a count of how many times the `go_more` signal has been broadcast from the `DUT` process, and it can assume the values 0, 1, 2, etc. It represents how many times handshaking has occurred.

16. Click on the Register Control  icon on the Test Bench Builder Toolbar shown in Figure 17. The mouse cursor converts to an **Add** icon.
17. Click anywhere on the Design Window to place the Register. This opens the Register Property Page. The Test Bench `mytb` now looks as shown in Figure 17.
18. Click the **Register** tab on the Register Property Page and click the **Signed** box in the **Display as:** section.
19. From the drop-down menu next to the **Send signal and/or do action** caption (click  to see this drop-down menu), select **Never** to reflect that this Register will not be used for signal injection.
20. Click on the **Variable** tab (these tabs are located near the top of the Property Page) to display the Variable Property Page.
21. Check the **Connected to variable named:** box to indicate that this register monitors the value of a prototype variable.
22. Select `TutorialTester::a` as the variable to be monitored from the drop-down menu in the **Variables** section.
23. Click **OK**. Note that a Register with an initial value of 0 appears in the Design Window. Its location can be changed by dragging it to the desired position.

Save the Test Bench file `mytb.tb` by clicking on the **Save**  icon.

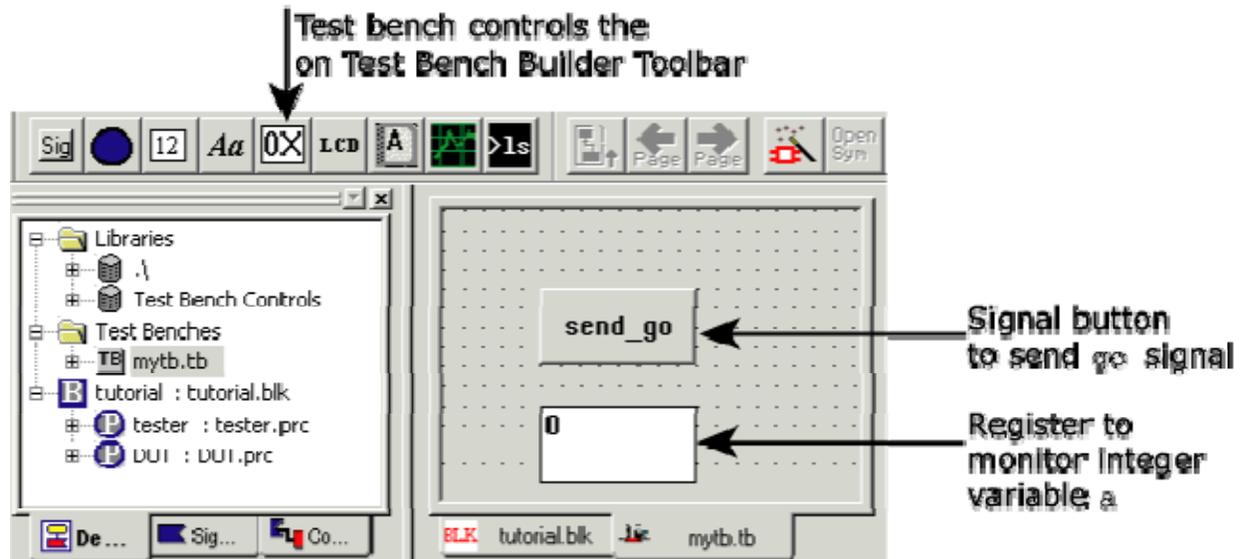


Figure 17: Test Bench Builder Toolbar (Top Left) And The Controls in Test Bench `mytb`

Running a Test Bench

No special actions are required to run a Test Bench. Whenever a prototype containing Test Benches is run, a window appears containing the Test Bench controls for that prototype.

In this section, the Test Bench `mytb` (created in the Section [Creating a Test Bench](#)) is used to control prototyping of the `DUT` process by injecting signals. The effect of the signal injection is also monitored by watching the value of variable `Tutorial::a`.

To launch Test Bench `mytb`:

1. Insert a breakpoint in the `ready` state of the `Tester` process as described in [Setting Breakpoints](#).
2. Start the prototyping session by pressing **F5** on the keyboard or by clicking the **Go**  icon. The Prototype Cockpit will be displayed when the prototyping session first launches, although the session is suspended when it reaches the `ready` state of the `Tester` process due to the inserted breakpoint. The Test Bench `mytb` depicted in Figure 18 will open inside the Prototype Cockpit window.

Note: The Test Bench depicted in Figure 18 may be covered by a Messages pane (which is not the same as the Message Window at the base of the Virtio Innovator screen) when the prototyping session begins. If so, move the Messages pane (the Prototyping Cockpit window may need to be expanded first) and the Test Bench will be behind it in its own pane. Then resize the Test Bench pane to look like that shown in Figure 18. Be sure to keep the Messages pane visible as Test Bench errors will appear here.

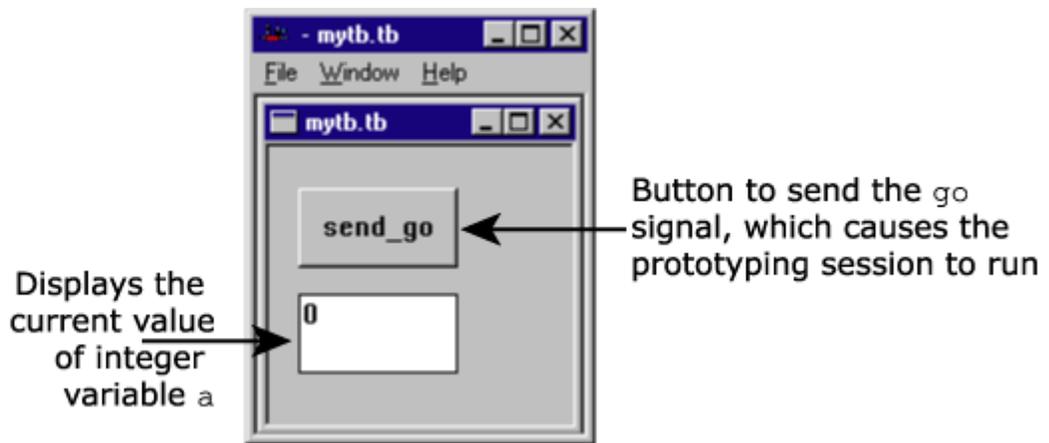


Figure 18: Test Bench `mytb`

3. Click on the **send_go** button in Test Bench `mytb`.
4. Click on the **Single-step**  icon. Since the prototyping session has stopped in the `Tester` process at the breakpoint (in the `ready` State construct immediately before the `go_more` Signal-In construct), one can infer that the `go_more` signal is about to be consumed. This implies that a `go` signal was indeed sent when the **send_go** button was clicked since the `DUT` process broadcasts `go_more` only after it receives the `go` signal.

5. Click on the **send_go** button again and then single-step to the Signal-In construct in the **Tester** process. Since the prototyping session reaches this construct, one can conclude that the **go_more** signal was received again. Note that now the Register control in the Test Bench changes from 0 to 1. Recall that integer variable **a** in the **Tester** process is updated by the payload of the **go_more** signal, so having its value change from 0 to 1 indicates that handshaking occurred for the second time.
6. Repeat step 5 again to verify that the prototype works as intended.
7. Terminate the prototyping session by clicking the **Abort Prototyping Session**



icon on the Build/Run Toolbar.

Symbols

In the tutorial prototype created above note that the **DUT** process relies on shared global signals (**go**, **go_more**, **clk**) to operate correctly. This design approach suffers from the following disadvantages:

- **Incorrect Operation Due to Conflicting Global Signal Names:** If the **DUT** Process described above is used as a component in another prototype, any signals in Processes other than **DUT** must have unique names. Otherwise all signals sharing the same name will be activated simultaneously. This can result in incorrect operation in cases where, for example, a common signal name such as **clk** is reused in an unrelated block of logic. In addition, the Virtio Innovator might not be able to detect such name conflicts.
- **Difficulty Reusing Logic Blocks:** After creating a generic prototype of a device or subassembly, it is often helpful to make further use of a design or some portion thereof. However conflicting signal names can severely limit a designer's ability to do so easily.
- **Inability to Readily Customize Reused Logic Blocks:** It is frequently desirable to customize at runtime a generic logic block based on the value of a specific parameter. One obvious example might be the matching of device clock frequency to a particular manufacturer's specification. Shared signal and variable names mean that the implementation of a prototype must be changed with each reuse in order implement design-specific customization.

To avoid such problems Virtio has defined in **MAGIC-C** a graphical construct, the **Symbol**, that encapsulates design information and permits easy customization thereof. Communication between **MAGIC-C** Symbols occurs through their *pins* only. As a result conflicts between global and/or shared signals or variables are no longer an issue.

By using **MAGIC-C** Symbols and by defining parameters for them, self-contained, reusable and customizable components can easily be created. Such components can then be stored in a **MAGIC-C Symbol Library** for later use. Refer to the *Virtio Innovator User's Manual* for more information on Symbols, parameters and Symbol libraries.

The next two pages, named "Creating Magic-C Symbols" and "Using Magic-C Symbols" respectively, discuss converting existing Process implementations (for **Tester** and **DUT** in

our case) to Symbols then adding and connecting those Symbols to the top level Block (`tutorial.blk` for this tutorial).

Creating Magic-C Symbols

On this page the `tutorial` design is modified to use **MAGIC-C** Symbols. Note that there are three shared signals: `go`, `go_more` and `clk` in the Declaration box of the `tutorial` implementation. Therefore, three Symbol pins are required on both the `Tester` and the `DUT` Processes to continue using the same handshaking approach. Start by creating a Symbol for the `Tester` Process as follows:

1. Disable the interactive Design Rule Checker (DRC) by clicking **Settings** on the Main Menu Bar and selecting **Error Checking Off**. This is done because the interactive DRC generates errors during design entry and is not relevant during this example.
2. Edit the Declaration box of the `tutorial` Block by deleting declarations for signals `go` and `go_more`, and for clock `clk`. Shared signals are not used with Symbols.
3. Edit the `Tester` Process by adding the following to its Declaration box:


```
clock clk;
```
4. This effectively makes the `clk` a local signal inside the `Tester` Process.
5. To transmit the `clk` outside the Symbol boundary, a Symbol pin is required. We will call this pin `eclock` and will transmit it every time that `clk` is triggered. To make `eclock` a Symbol pin, first add the following declaration to the `Tester` Process:


```
extern_signal eclock;
```
6. Next, add the constructs shown in Figure 19 to the `Tester` Process. Note that these do not connect to any of the other constructs created previously. The effect of adding these constructs is to activate Symbol pin `eclock` and drive it across the Symbol boundary whenever local clock signal `clk` is triggered. This signal is then consumed by any other Processes that take `eclock` as an input.

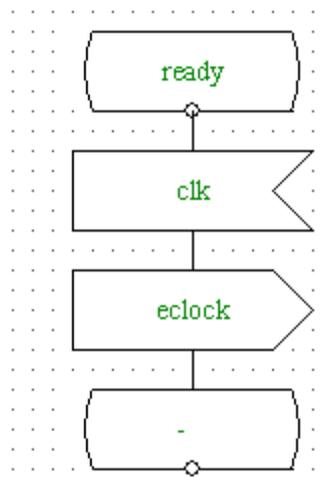


Figure 19: eclock Symbol Pin Driven by Local Clock clk

7. Now add the following two declarations to the `Tester` Process to declare that `go` and `go_more` are two Symbol pins of the `Tester` Process:

```
extern_signal go;

extern_signal go_more(int);
```

8. Create a Symbol for the `Tester` Process, and verify editing of that Process as follows:

a. Launch the New Symbol Wizard by clicking **Edit** on the Main Menu Bar and choosing **New symbol Wizard** or by simply clicking on the  icon on the Symbol Editor

Toolbar. The Symbol Editor Toolbar  is located at the top of the Virtio Innovator screen. [If the Symbol Editor Toolbar is not visible, first click **View** from the Main Menu Bar then select **Navigate Toolbar**.]

b. Check the **Make a New Symbol for the Current Document** box to make the new Symbol from the current Process, then select **Next**.

c. Verify that the **Symbol Name** field reads `Tester`, indicating that a new Symbol is being created for the `Tester` Process (`.prc`). The Symbol will be stored as a file named `Tester.sym` in the current directory.

d. Verify that the **Symbol Location (Library Director)** shows `.\` as the current directory for creating the new Symbol for `Tester`.

e. Click on the **Next** button to open the Symbol Pins dialog box depicted in Figure 20 (the pull-down dialog box will not be shown when the dialog box first appears; it must be opened).

f. In Symbol Pins dialog box, type a **Pin Name** value of `go` and use the drop-down menu for Location to select a value of `Right`. [Display the pull-down menu using the  icon.] Then click on the **Add** button to create the new pin.

g. Similarly, add `go_more` and `eclock` as two other Symbol pins, in that order. Set the value of the **Location** field as `Right` for both the pins. Figure 20 depicts the Symbol Pins dialog box as the third of the three pins is being added.

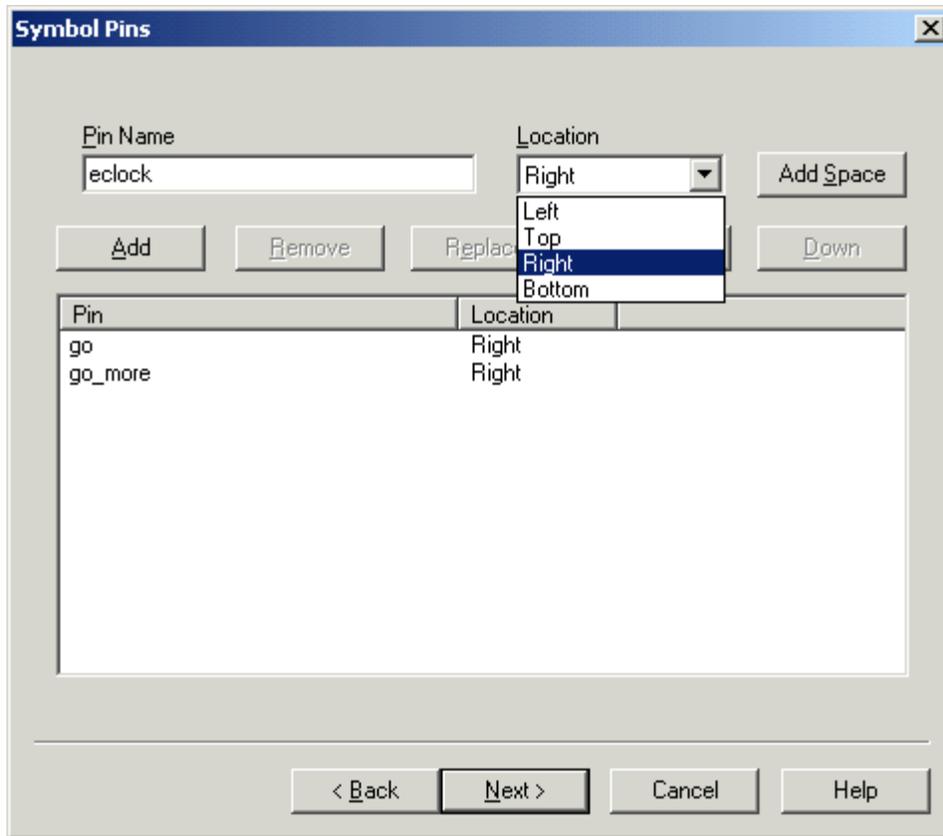


Figure 20: Symbol Pins Dialog Box For The Tester Process

- h. Click on the **Next** button to open the **Finish Symbol** dialog box. Note that the **Back** button can be used to modify any information previously entered for the Symbol.
- i. Click on **Finish** to create the Symbol. The Design Window now shows the Symbol schematic or outline as depicted in Figure 21. Expanding the **Libraries** icon in the Design Browser (beneath the  icon) shows that the new Symbol has been added to the current directory.

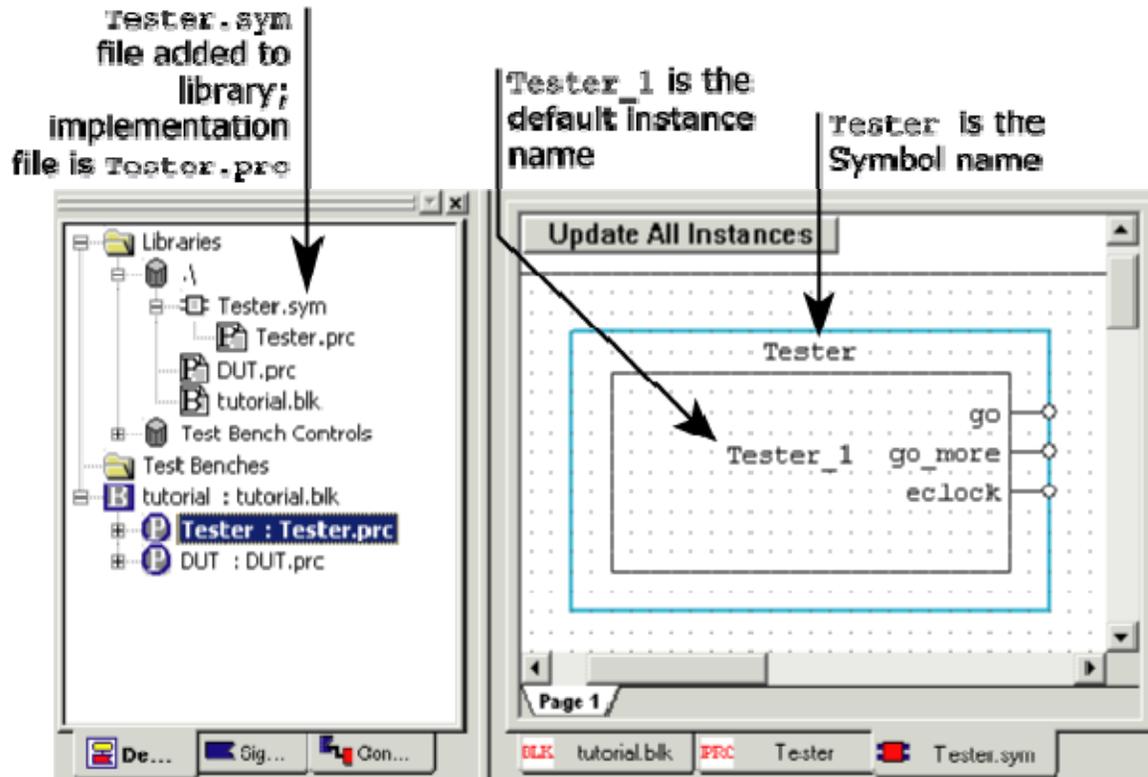


Figure 21: New Symbol For The Tester Process

9. Now edit and create a Symbol for the `DUT` Process as follows:
 - a. Load the `DUT` Process in the Design Window by double-clicking on the `DUT` Process  icon in the Design Browser.
 - b. Add the following to the `DUT` Process Declaration box:


```
extern_signal go;
extern_signal go_more(int);
extern_signal clk;
```
 - c. Create a Symbol for the `DUT` Process using the New Symbol Wizard as described in steps 8a through 8i above. Note that the Symbol pins should be created in the following order: `go`, `go_more`, `clk`. They should appear on the left side of the Symbol (to do this, set the **Location** property value to `Left` on the Symbol Pins dialog box). In the Design Browser, `DUT.sym` appears as a component in the current directory.

Modifying a Symbol or Its Implementation After Creation

Once a Symbol has been created, either its implementation (the `Tester` or `DUT` Process in the example immediately above) or its graphical representation (depicted in the right pane of Figure 21) can be edited. It is also easy to move between editing the implementation (the contents of a Process or Block that appears in the Design Window) and the graphical representation (shown in the Design Window while in Symbol Edit Mode) or vice versa.

To open for viewing or editing a Symbol implementation (Process or Block) while editing a Symbol (.sym), right-click on the Design Window to display a pull-down menu and choose

Open Implementation. Alternatively, click the **Open Implementation**  button on the Symbol Editor Toolbar.

To open a Symbol's graphical representation (for the purpose of adding or editing pins, changing names, adding a parameter, etc.) in Symbol Edit Mode while displaying its implementation (Process or Block), right-click on the Design Window to display a pull-down menu and choose **Open Symbol**. One can also simply press the **Open Symbol**

 button on the Symbol Editor Toolbar.

Using Magic-C Symbols

Now we will use the Symbols for **Tester** and **DUT** created earlier to implement our **tutorial** design. To do so we must instantiate the two newly created Symbols in the prototype. To do so:

1. Load the **tutorial** Block in the Design Window.
2. Select the **Tester** Process in the **tutorial** design by clicking once on the associated Process construct in the Design Window. Now delete it in any **one** of the following three ways:
 - a. By pressing the **Delete** key on the keyboard.
 - b. By clicking the  icon on the Edit Toolbar.
 - c. By right-clicking on the Process to display a pull-down menu and selecting **Delete**.
3. Repeat step 2 above for the **DUT** Process in the **tutorial** design.
4. If it is not already visible, open the Design Browser (click **View** on the Main Menu Bar and select **Browser Window** then click the  icon at the bottom of the Browser Window). Then expand the **Libraries** section at the top of the Design Browser by clicking on any expand icons (#) beneath the  Libraries and  icons. The Library section of the Design Browser will now look like Figure 22a.

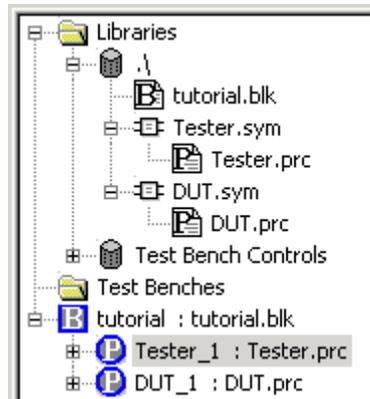


Figure 22a: Design Browser With Library Section Fully Expanded

5. Right-click on `Tester.sym` and select **Add Instance** from the pull-down menu that appears.
6. Move the cursor to the location in the Design Window for placing an instance of the `Tester` Symbol and left-click. Note that the instance name is modified when it is placed (for example, `Tester_1`). This name can later be changed by right-clicking on the instance name (having a **P** icon next to it) below the `tutorial` top level Block in the Design Browser and selecting **Change Instance Name** from the pull-down menu that appears.
7. Repeat steps 5 and 6 to place an instance of `DUT.sym`. Note that an instance of the `DUT` Process appears next to the **P** icon in the Design Browser.
8. Connect the pins of each Symbol as follows:
 - a. To connect the `go` Symbol pins for both Processes, left-click on the `go` Symbol pin of the `Tester_1` instance (note that the cursor has changed to a pencil).
 - b. Move the cursor over the `go` Symbol pin of the `DUT_1` instance and left-click to connect.
 - c. Label the wire connecting both pins by typing a name (for example, `g`) in text the box corresponding to the wire. Declare the wire name in a Declaration construct in the `tutorial` Block as:

```
signal g;
```

- e. Similarly, connect the `go_more` pins of the two Symbols by a wire named `gm`.
- f. Now connect the `eclock` pin of `Tester_1` and the `clk` pin of `DUT_1` with a wire named `c`. The following additional declarations are required in the Declaration construct of the `tutorial` Block:

```
signal gm(int);
```

```
signal c;
```

Note: It is optional to name the wires connecting Symbol pins. The Virtio Innovator uses a temporary wire name for unnamed wires. However, to trace signal values using the Waveform Viewer, it is recommended that wires be named for easier tracing and debugging.

The Symbol version of the `tutorial` design appears in Figure 22b. The design can be compiled and the signal values traced as described on earlier pages of this tutorial to verify that the design behaves in a fashion identical to the non-Symbol version.

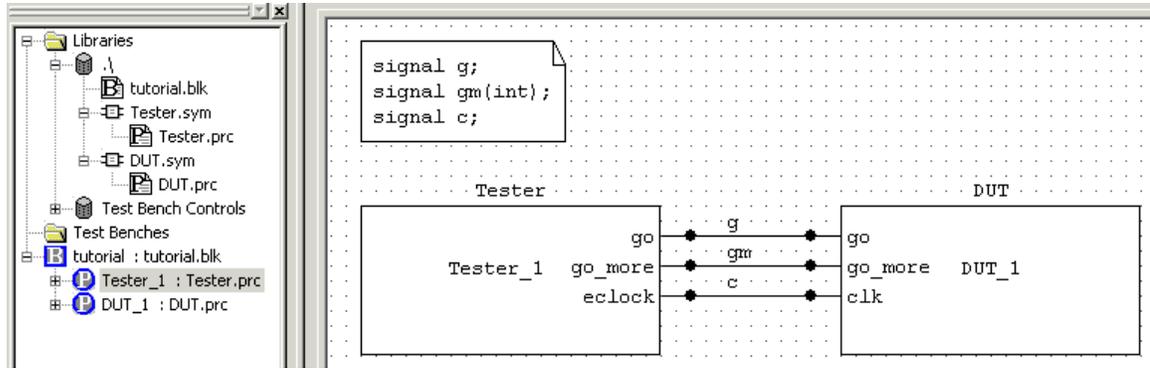


Figure 22b: Tutorial Design Using Symbols

- Now that the Symbol version of `tutorial` is completed, recompile the prototype and verify that there are no errors. Then run it to confirm operation, placing breakpoints and/or modifying elements such as the Test Bench if necessary.

Opening a Symbol or Its Implementation

Once a Symbol has been created and instantiated, either its implementation (the `Tester` or `DUT` Process for example) or its outline (its graphical representation) can be edited. The methods for accessing each of these are explained below. However let's first define the three key terms used to refer to Symbol elements in a prototype.

Symbol Implementation

A Symbol implementation is the underlying Process or Block description that describes the behavior of a Symbol. Figure 23 depicts the Symbol implementation for Symbol instance `Tester_1`.

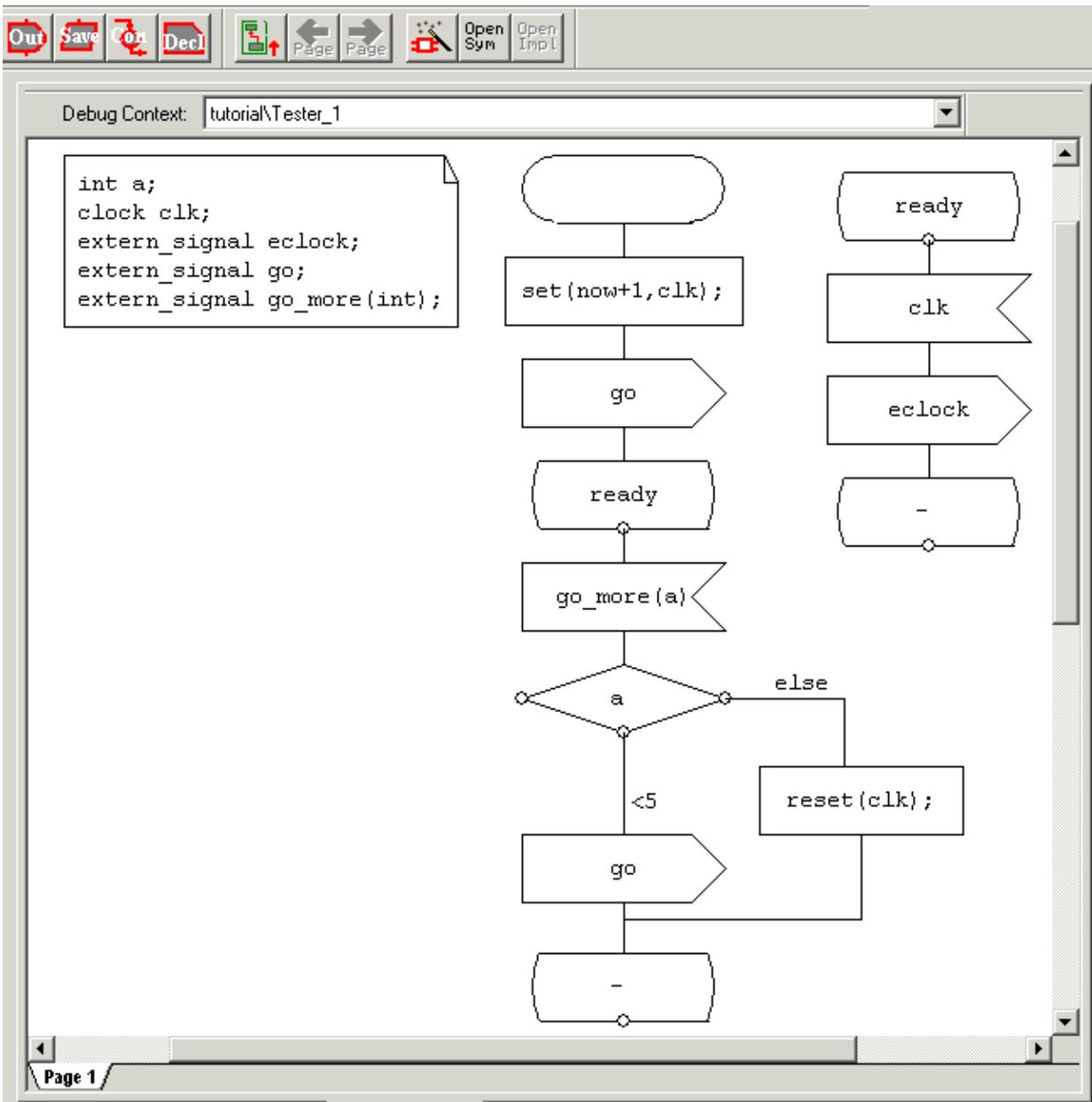


Figure 23: Symbol Implementation for Process Tester

Symbol Instance

A Symbol instance represents one placement of a Symbol implementation. It contains the pins and their names. Figure 24 depicts the Symbol instances for [Tester_1](#) and [DUT_1](#).

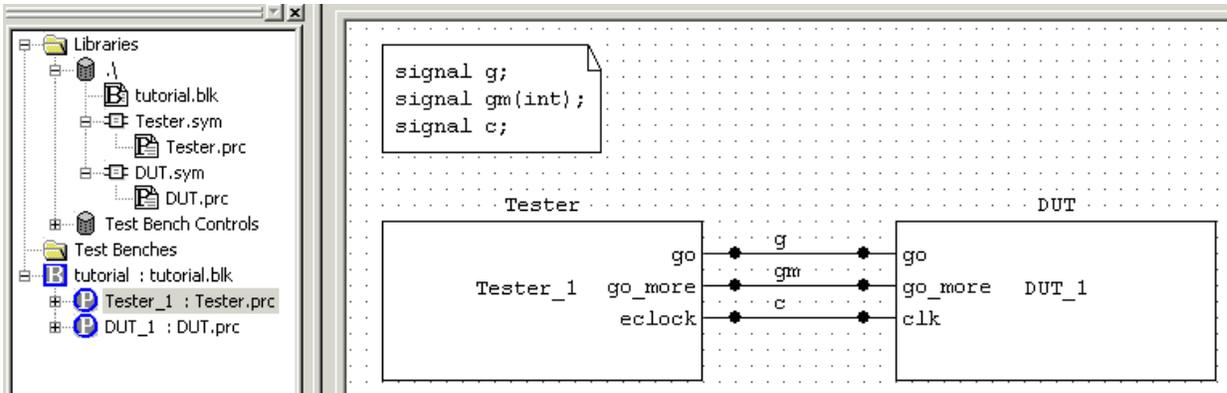


Figure 24: Symbol Instances Tester_1 and DUT_1

Symbol Outline

A Symbol outline is the rectangular graphical representation that specifies the layout for a Symbol instance. It contains the pins and their names, as well as the name of the underlying Block or Process. Figure 25 depicts the Symbol outline for `Tester_1` in Symbol Edit Mode. In this mode, it is possible to edit pin names, add new pins, etc.

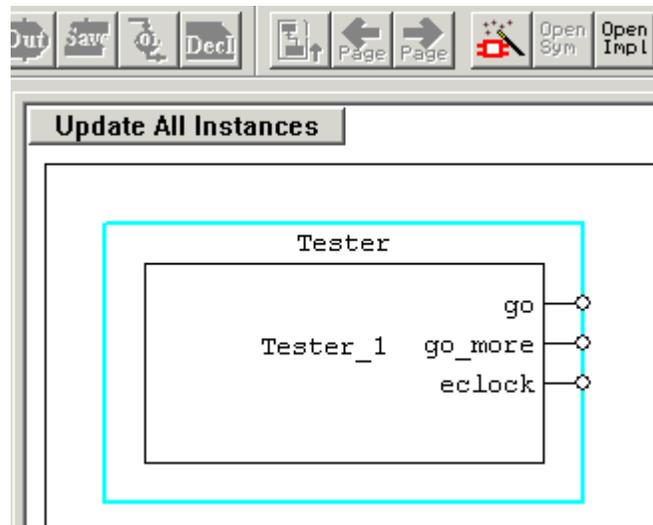


Figure 25: Symbol Implementation for Process Tester

Opening a Symbol Implementation While Viewing a Symbol Instance

To display in the Design Window a Symbol implementation (Process or Block) while a Symbol instance is visible:

- Right-click on the Symbol instance in the Design Window to display a pull-down menu and choose **Open Implementation**.

OR

- Double-click on the Symbol instance.

Opening a Symbol Implementation While Viewing a Symbol Outline

To access a Symbol implementation (Process or Block) from a Symbol outline:

- Right-click in the Design Window to display a pull-down menu and choose **Open Implementation**.

OR

- Click on the  icon on the Symbol Editor Toolbar.

Opening a Symbol Outline While Viewing a Symbol Implementation

To access a Symbol outline while a Symbol implementation is visible in the Design Window:

- Right-click in the Design Window to display a pull-down menu and choose **Open Symbol**.

OR

- Click on the  icon on the Symbol Editor Toolbar.

Opening a Symbol Outline While Viewing a Symbol Instance

To open for viewing or editing a Symbol (in Symbol Edit Mode) while viewing an instance of the same Symbol, right-click on the Symbol instance in the Design Window to display a pull-down menu and choose **Open Symbol**.

Code Examples

MAGIC-C Code Examples

This section contains some **MAGIC-C** code examples that describe approaches to common tasks. These examples are expected to help users master the idioms of MAGIC-C.

Introduction

The examples of code provided in this chapter illustrate the following:

- [Creating process concurrency](#)
- [Reset: Bringing an FSM to an initial state](#)
- [Starting and stopping a process](#)
- [Timers](#)
- [Generating clock signals using the clock construct](#)
- [Generating a clock signal with a duty cycle of 30%](#)
- [Getting the current prototyping time](#)
- [Synchronizing data transmission using a clock](#)
- [Synchronizing data transmission between FSMs](#)
- [Clocked MAGIC-C Loops](#)
- [Interrupts](#)

Creating Process Concurrency

In **MAGIC-C**, concurrency is achieved by separating a design into several processes (FSMs). Figure 1 shows an example with four concurrent processes.

- Do not abuse the addition of Processes as this reduces the execution speed of a prototyping session.
- Use multiple Processes only where there are operations that are performed in parallel in the real system.
- Use C function calls inside of Task blocks for operations that are considered atomic within the context of the design.

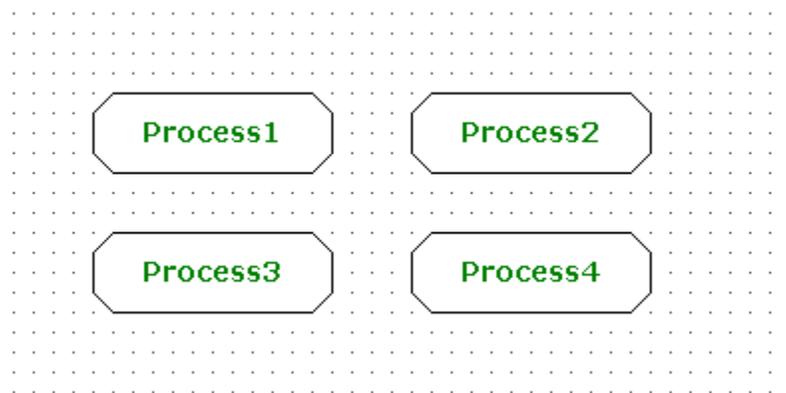


Figure 1: Creating Process Concurrency

Reset: Bringing an FSM to an Initial State

This is a very useful way to restart all the processes (FSMs) in the system. **MAGIC-C** uses broadcast of signals to listeners that are within the scope of the sent signal. Therefore, a single reset signal defined at the top level (system level) can be used to bring all the processes in a design to their initial state.

The "*" in a State construct  means "every state in this process." In the example shown in Figure 2, the current state of the process does not matter. All processes return to their initial states upon receiving the reset signal.

Every process requires a Start construct  to point to its first state. All the initialization code of a process should be put in a Task construct  between the Start and the first State construct. The initialization code is executed when the prototyping session starts (at prototyping time zero) and after every reset signal. As a result of this initialization, all the processes reach a known state state.

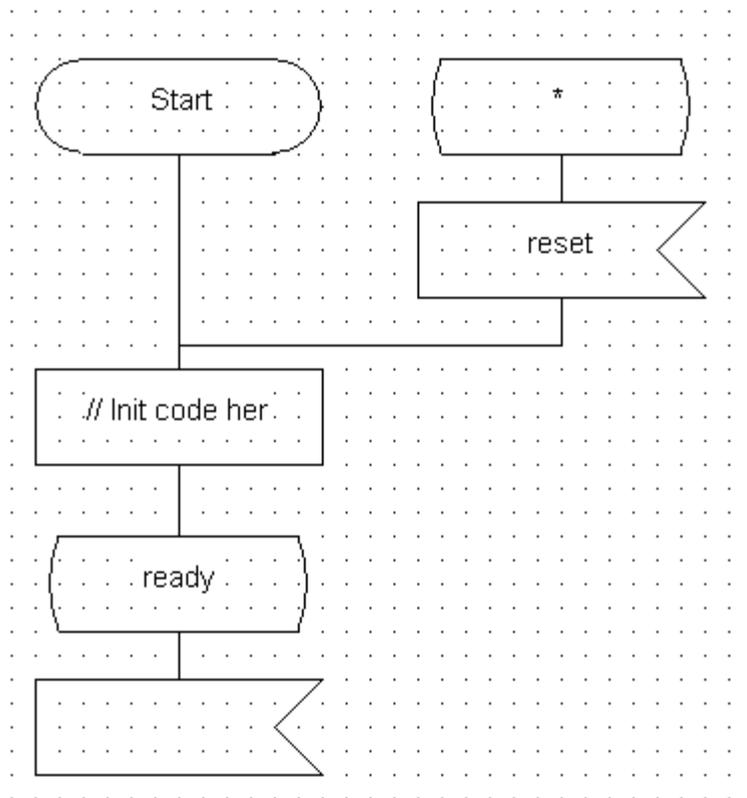


Figure 2: Bringing An FSM To A Known Initial State With A Reset Signal

Starting and Stopping a Process

MAGIC-C does not support run-time creation and destruction of processes (FSMs). To achieve similar results, processes can be started and stopped by using explicit signals.

In the example shown in Figure 3, the signal `start_process` is used to activate the process, and the `stop_process` signal is used to halt the process. Whenever the process receives a `stop_process` signal, regardless of the state it is in, it changes to the `process_halted` state. To resume operation, the process must receive a `start_process` signal.

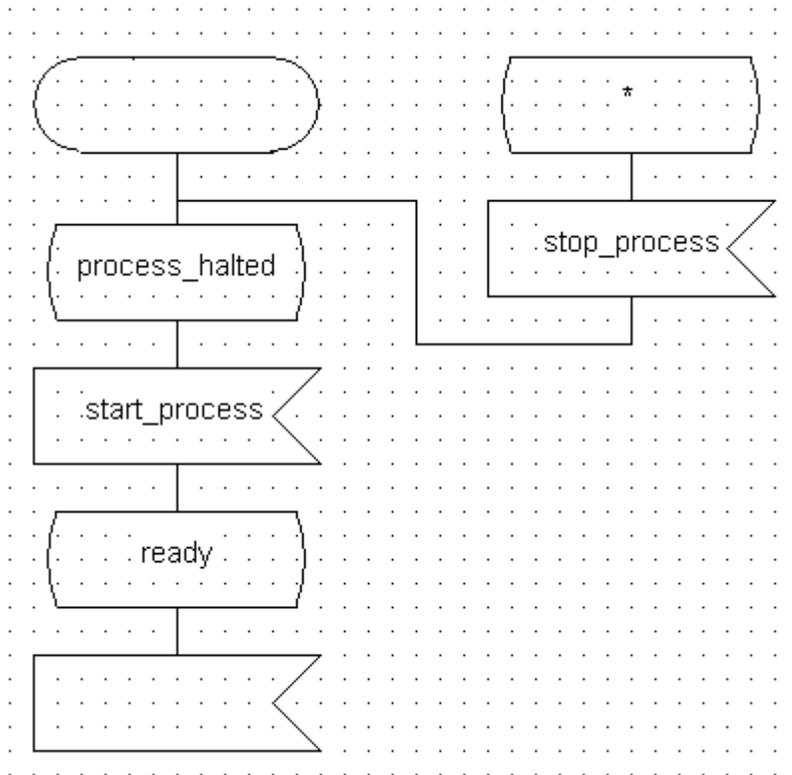


Figure 3: Starting And Stopping A Process

Timers

Timers are signals that are scheduled for broadcast at some future time. Before a timer

can be used, it must be declared in the Declaration box  using the `timer` keyword. After a timer is declared, it can be scheduled for broadcast at a given prototyping time by calling the `set()` function. In the example shown in Figure 4, the timer `t` is set to expire

in two prototyping time units. To receive the timer message, a Signal-In construct  is used with the name of the timer. Timers can also be stopped before they expire by using the `reset()` function.

A timer example generating a periodic signal `clk` is given in Figure 4.

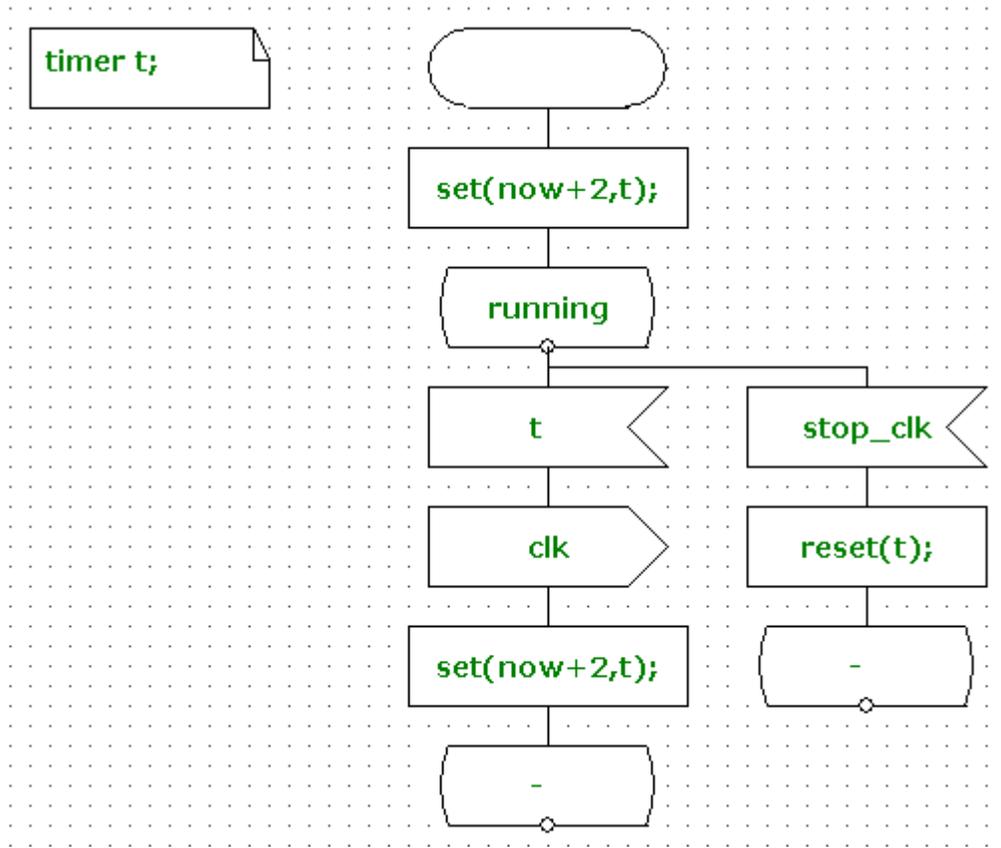


Figure 4 MAGIC-C Timers

Generating Clock Signals Using the Clock Construct

MAGIC-C supports generation of clock signals. A clock signal is a timer signal that is set *automatically* after being expired. On the left in Figure 5 is the declaration of the clock at the system level. The right side of the figure shows the setting and stopping of clock signals.

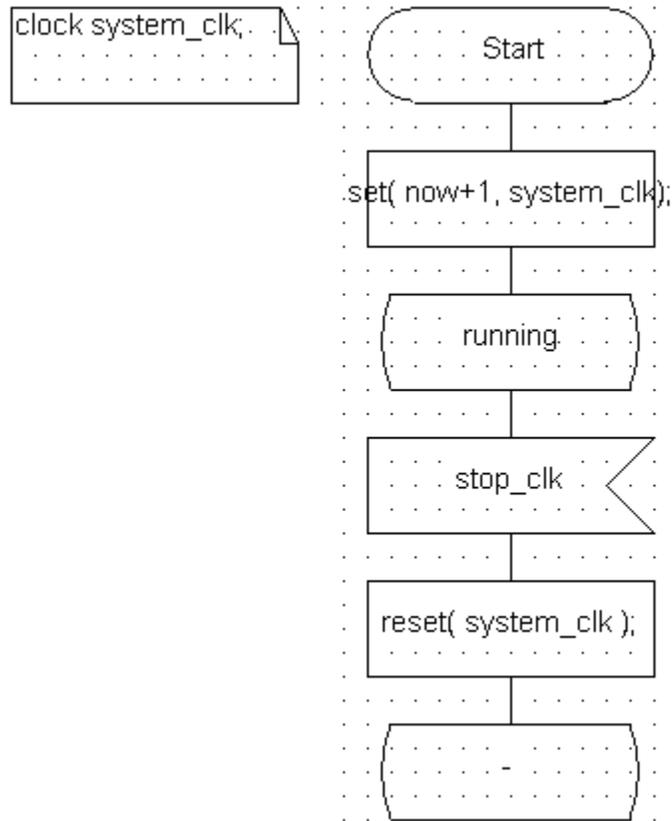


Figure 5: Generating Clock Signals Using Clock

Generating a Clock Signal with a Duty Cycle of 30%

When a clock with a duty cycle other than 50% is needed, it is possible to use timers set to a different expiration time. In the previous example (see [Figure 5](#)), stopping the free running clock required the use of a `stop_clk` signal to disable `system_clk`. The `stop_clk` signal is input from the `clk_high_state` to always stop the clock at the high state. However, the FSMs have no control on when the `stop_clk` arrives. To make sure that the `stop_clk` signal is not lost, this example uses the save construct after the `clk_low_state` (Figure 6). If `stop_clk` arrives when the process is in the `clk_low_state`, it is held until the next state to be processed.

A **MAGIC-C** state requires at least one transition to be a valid state. This example creates an artificial transition from state `idle` to state `idle` upon receipt of `stop_clk`.

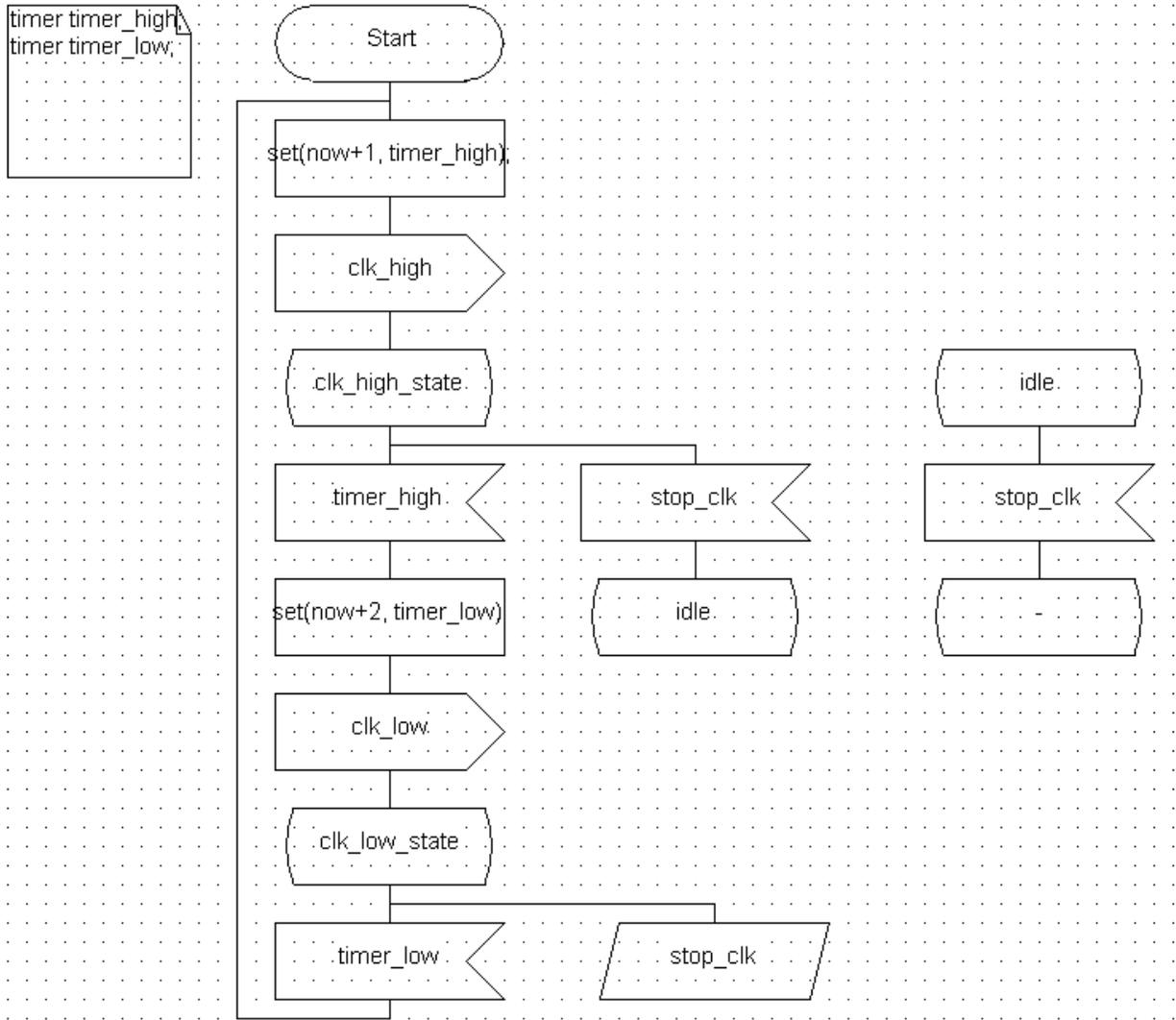


Figure 6: Generating a Clock Signal With A Duty Cycle Of 30%

Getting the Current Prototyping Time

The earlier examples involved timers and clocks, both of which cause the prototyping time to advance. In the Virtio Innovator, the current prototyping time can be accessed via the *now* keyword. The keyword *now* is represented as a 64-bit integer. Therefore, when accessed in the `printf` function, `I64d` is used as the type modifier for the `__int64` variable.

In the example shown in Figure 7, the arrival time of each of the clock signals is printed. When the prototyping time reaches 20, a `stop_clk` signal is sent to the clock generators.

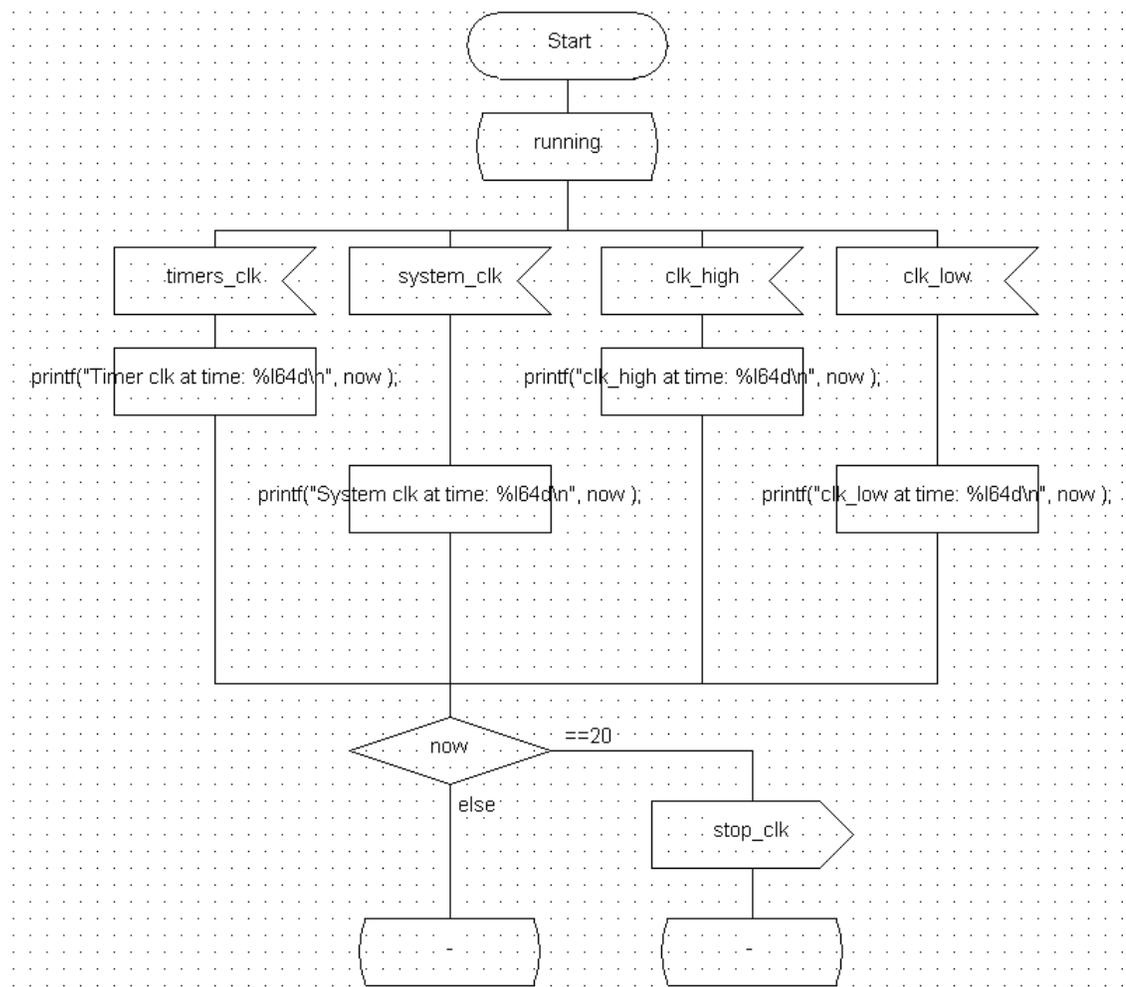


Figure 7: Getting The Current Prototyping Time

Synchronizing Data Transmission Using a Clock

It is often necessary to synchronize the sending of signals with a system clock. The example in Figure 8 synchronizes sending of the `output_results` signal with the `clk`.

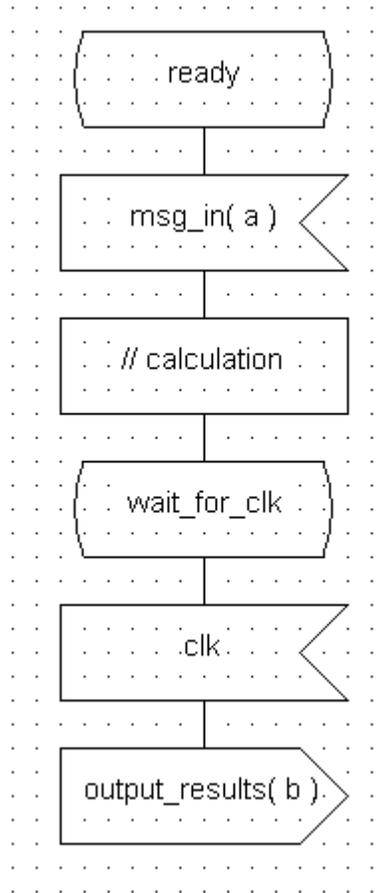


Figure 8: Synchronizing Transmission Of Data

Synchronizing Data Transmission Between FSMs

When exchanging data between two devices, some form of handshaking protocol is usually required. Figure 9 depicts logic that requests a piece of data from some other model in the prototype. Here, the **FSM** sends a signal carrying the address of the data. The device then enters a `wait_for_data` state to wait for the transmission of this data.

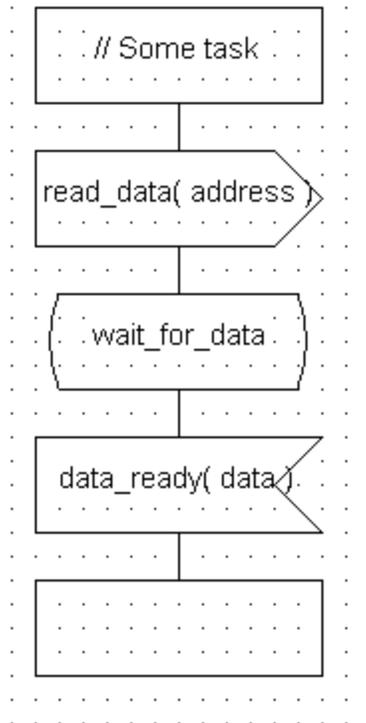


Figure 9: Synchronizing FSMs

Clocked Magic-C Loops

A basic loop structure where the enclosed statements, if any, are executed at each clock-cycle, is referred to as a `clocked loop`.

This section describes how to create standard clocked loops in `MAGIC-C` of the following kinds:

1. [Do-While / Repeat-Until Clocked Loop](#)
2. [Clocked WHILE Loop](#)
3. [C-Style Clocked FOR Loop](#)

Do-While / Repeat-Until Clocked Loop

A clocked **do-while** loop (or **repeat-until** loop in some languages) is shown below:

```

    <statement1>; // not part of do-while loop

    do {

        <statement2>;
    } while <cond>;
  
```

where `<statement2>` is assumed to be executed at each user-defined clock, say `clk`. The corresponding **MAGIC-C** code fragment to describe a clocked do-while loop is shown in Figure 10.

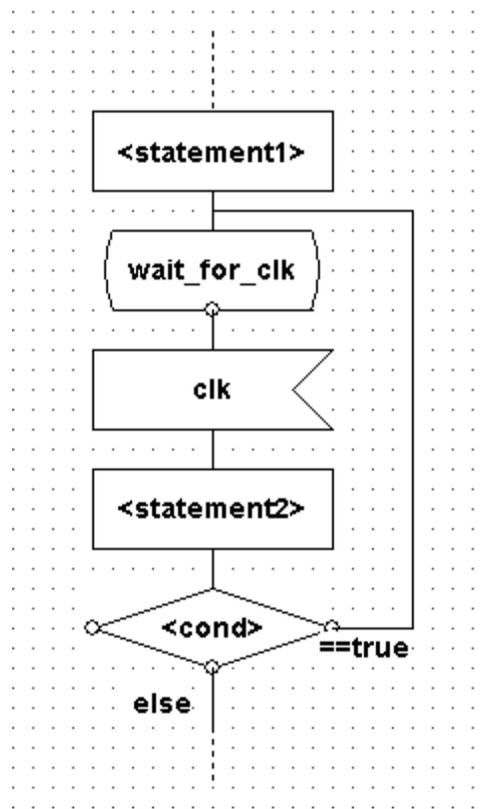


Figure 10 - A clocked DO-WHILE Loop.

Clocked WHILE Loop

Consider the following clocked **While** loop:

```

<statement1>; // not part of while loop

while ( <cond> ) {

  <statement2>;
};
  
```

where `<statement2>` is to be executed at each user-defined clock, say `clk`. It can be described in **MAGIC-C** as shown in Figure 11.

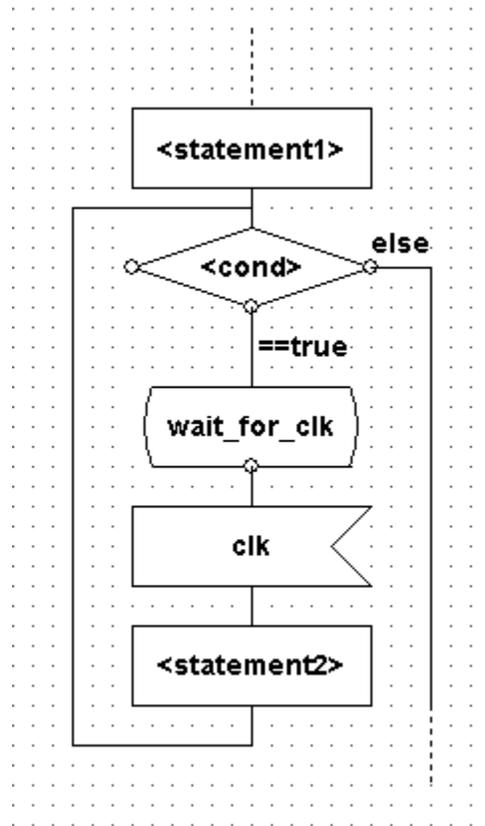


Figure 11 - A clocked WHILE loop.

C-Style Clocked FOR Loop

Consider the following C-style **For** loop:

```

<statement1>; // not part of for loop

for ( <init_statement>; <cond>; <end_statement> )
{
  <statement2>;
};

```

Using C semantics, we can convert this to an equivalent **While** loop:

```

<statement1>; // not part of for loop

<init_statement>;

while ( <cond> )

```

```
{
<statement2>;
<end_statement>;
};
```

Now, using the clocked **While** loop representation introduced in the previous section, an equivalent MAGIC-C code fragment can be written as shown in Figure 12.

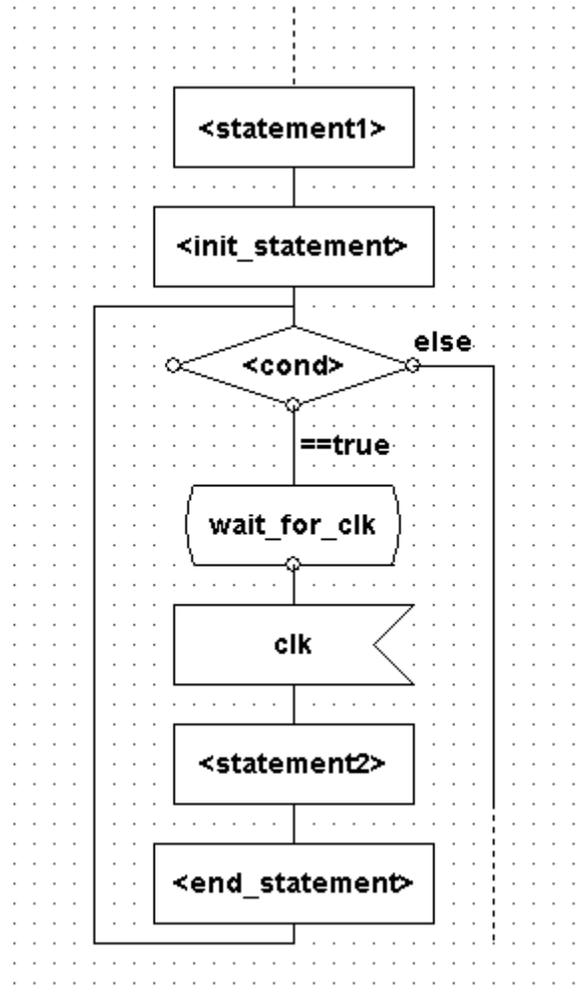


Figure 12 - A converted clocked FOR loop.

Note: It is trivial to write un-clocked DO-WHILE / REPEAT-UNTIL, WHILE and C-style FOR loops in MAGIC-C. In corresponding MAGIC-C code fragments shown above, remove all MAGIC-C State and Signal-In constructs. The resulting code executes the basic loop structure without any associated (clock) delay. An unclocked loop can be captured by: (1) a Task construct and nesting this in a MAGIC-C Decision loop, or (2) by a (ANSI)-C loop inside a Task construct. Both are

semantically equivalent, and have the same prototyping session performance. However, the former case allows the Virtio debugger to single step through every loop instance. In the latter case, the Task construct is executed in a single step, and no breakpoint can be set on a loop instance.

Interrupt

The behavior of an interrupt can be modeled in **MAGIC-C** by making use of two State constructs, one containing a star (i.e. '*') and one containing a dash (i.e. '-'), as depicted in Figure 13. The star State assures that the interrupt is sensitive in all States of the left hand MAGIC-C graph (the right hand graph represent the interrupt servicing). The interrupt servicing behavior is captured in the Task construct in between the two States. The dash State causes the execution to be resumed in the State in which the Process was at the moment the interrupt Signal occurred.

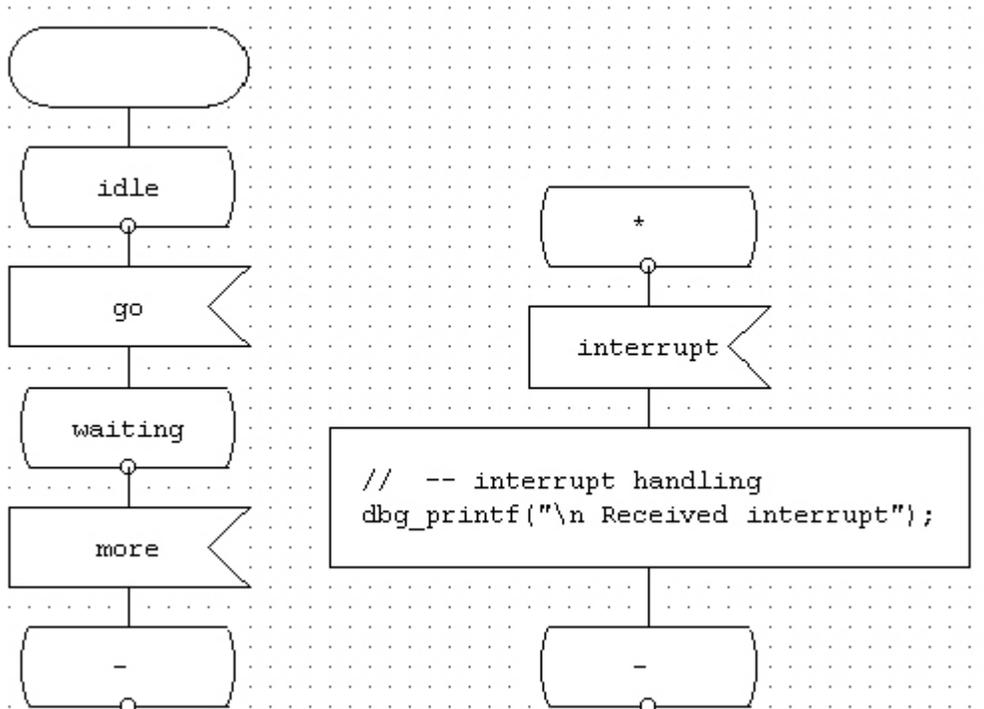


Figure 13: Modeling An Interrupt In MAGIC-C

Matched Filter Design

Overview

In this chapter, we start with the specification of a simple matched filter. Then, using the Virtio Innovator, a MAGIC-C description of the matched filter will be constructed. The design will then be refined to some degree. An in-depth understanding of the process involved in using MAGIC-C in a functional design will be reached by the end of this chapter.

Introduction

The following topics are covered in this chapter:

- [Matched filter specification](#)
- [System partition](#)
- [Specification of matched filter \(initial version\)](#)
- [Adding delay and time in the model](#)
- [Using reset in a MAGIC-C model](#)
- [Handling protocol refinement in MAGIC-C](#)
- [Summary](#)

Matched Filter Specification

Matched filters are commonly found in signal processing applications. For example, in simple radar, a matched filter can be used to filter out some of the noise in the returned signal. It can also pick out the peaks from the reflected signal. Implementation of a matched filter is application specific. The C code that represents a specific implementation of a matched filter is as follows:

```
void matched_filter (int din_prev, int din_now, int *acc_img, int
*acc_real)
{
// Depending on the phase the filter is in, we need
// to change the polarity of some input data.

static int phase=0;
int data_prev, data_now;
static int tap[31]; // taps of the filter

// coefficients of the filter
int coef[16] = {-11, -94, -130, -75, 57, 189, 219, 89,
-161, -389, -409, -98, 528, 1288, 1909, 2149};
int i;
```

Printed Documentation

```
// initialized all taps to have zero value
for(i=0; i<31; i++)
tap[i] = 0;
if (phase==0) {
data_prev = -din_prev;
data_now = din_now;
phase = 1;
}
else {
data_prev = din_prev;
data_now = -din_now;
phase = 0;
}
// shift the taps
for(i = 30; i >= 2; i--)
tap[i] = tap[i-2];

tap[0] = data_now;
tap[1] = data_prev;
*acc_real = tap[15]*coef[15];
*acc_img = (tap[14]+tap[16])*coef[14];

for(i=0;i<=13;i+=2) {
*acc_img += coef[i]*(tap[i]+tap[30-i]);
*acc_real += coef[i+1]*(tap[i+1]+tap[29-i]);
}
} /* end of matched_filter */
```

The matched filter takes in two input data and generates results in the form of a complex number. This type of system specification is commonly used in modern digital design. It captures the essential operations performed by the system using concise "C" programming constructs. In fact, many new IEEE standards use this form of representation to supplement the verbose description.

Note that even though the C description of the matched filter contains the essential operations of the filter, it requires additional analysis to test, and subsequently implement, the system in hardware.

In the following sections, the main steps of a top-down design in **MAGIC-C** are described.

System Partitioning

System partitioning is the process of defining the internal structure of the design. It involves understanding all operations performed by the design and dividing the design into smaller functional blocks. This division of a large program into smaller modules (or objects) is analogous to designing software. Similar to using interfaces for modules interaction in software, functional blocks in hardware communicate using signals passing. In general, a well-partitioned system fulfills the following requirements:

- Strong cohesion within a module: This means that all the functions performed by a module are closely related.
- Loose coupling between modules: This means that the dependencies between different modules are minimized.

While it is difficult to concretely measure these metrics, it is important that architects strive for these goals. In the previous, since an understanding of the matched filter is limited, it is sufficient to model it as one big **MAGIC-C** block, called `Filter`. This block is refined as the implementation proceeds.

Another aspect of system partition is the interface with respect to other systems. In the previous example, the interface is fairly straightforward. The matched filter receives two integers from the outside and returns the processed information in the form of a complex number. Based on this observation, it is obvious that another system needs to be modeled that will generate data to the matched filter if the intent is to test the final model. This additional system is called `Data_generator`.

With this information, the first cut system description of the matched filter can be entered in MAGIC-C. Figure 1 shows the first version of the matched filter system.

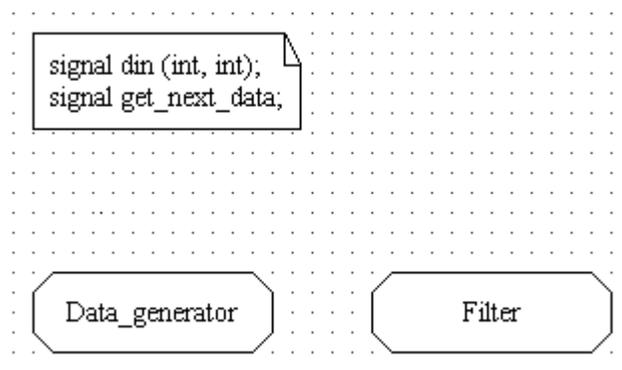


Figure 1 Match Filter System

In addition to declaring two communicating processes, `Data_generator` and `Filter`, we also declare some communicating signals between the two blocks are also declared. Note that signals in MAGIC-C are really messages sent from one finite state machine (FSM) to other FSMs. A MAGIC-C signal by itself does not contain any information other than an event occurrence. Additional information can be associated with a signal as shown in the previous example. Specifically, the signal `din` is declared to carry a payload of two

integers. Making an analogy to datapath/control design, signals are like the control part of the design, while the payloads are like part of the datapath.

In the C specification of the filter, there is an implicit synchronization between `Data_generator` (that exercises `matched_filter`) and `Filter` (by the nature of sequential processing in C). Such synchronization has to be explicitly instrumented in the MAGIC-C framework since all FSMs in the design run in parallel. In this example, a simple protocol is followed: When `Data_generator` has data to be sent to the filter, it sends the signal `din`, together with its payloads to notify the filter. The matched filter then proceeds to process the data. At the same time, `Data_generator` waits for the operations to complete. When the matched filter is finished with its job, it sends a `get_next_data` signal to notify `Data_generator` that it is ready to accept new data.

Specification of Matched Filter (Initial Version)

Even though an understanding of the matched filter is still limited at this point, with the "C" code specification, the system partition and the communication protocol in the previous section are enough information to complete the first draft of the design.

The system which will be designed here will consist of the following two parts:

1. [Data_generator Specification](#)
2. [Filter Specification](#)

Both are treated below in detail.

Data Generator Specification

To proceed, assume the following test scheme is used:

- `Data_generator` iterates 5 times. Each time, it sends two input data to the matched filter. The input data following the sequence of 0, 1, 2 ... 9. After the iterations, both `Data_generator` and `matched filter` are idle.

The process level description of the `Data_generator` is shown in Figure 2. For semantics of MAGIC-C constructs, refer to the *MAGIC-C Language Reference Manual*.

The process begins with the `Start` state. Naming of the `Start` state is optional since it only indicates where to begin when the prototyping session starts. Following the `Start` state, the user can perform any initialization work as desired. In this case, we initialize the variable `i` to 0.

The process then enters into a state called `idle`. The State and Signal-In MAGIC-C constructs appear together throughout most of the design. To transition into a state, the FSM (process) has to wait for a signal specified by the user. When state and Signal-In are used together, it can be interpreted as followed:

1. The next state of the precedes is the state specified.

2. The process waits for the signal specified.
3. When the signal arrives, the current state of the process is assigned to the value of the next state determined in #1 above.

In the example, the `Data_generator` process enters the state `idle` upon receiving the signal `get_next_data`. Note that the state transition does not happen until the `get_next_data` signal arrives.

The first thing after entering the `idle` state the process first checks to examine the value of `i` using the Decision MAGIC-C construct. If it is less than 10, it sends out the signal `din` (with its payload of `i` and `i+1`). Note that `true` is a keyword in MAGIC-C. If `i` is greater than or equal to 10, the process simply reverse to the original state. This means that after it sends out `din` five times, it ignores any invitation for sending additional data.

The Task box after the Decision construct can contain any legal C constructs. In this case, we increment `i` by 2.

When the `Data_generator` is finished with the iteration, it reverts to the original state and waits for the next `get_next_data` signal. This is how to accomplish the synchronization described previously. The "-" label in the state is an abbreviation for the current state.

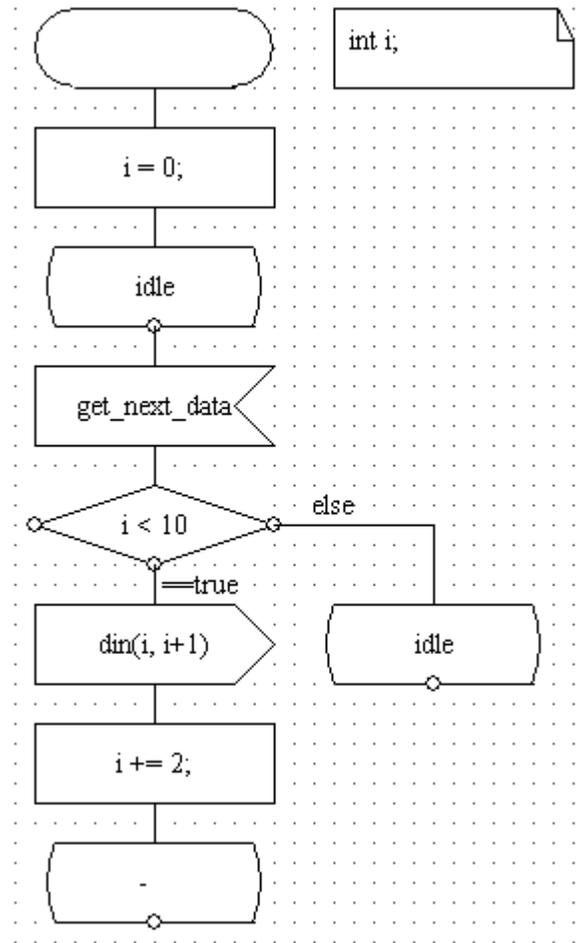


Figure 2 Data_generator Process Specification

Filter Specification

The specification of the `Filter` process is shown in Figure 3. As seen from the diagram. Most of the C code in the original specification is used directly in the design. This illustrates the attractiveness of the system. It provides an easy way to map a high-level concept into a framework in which fast prototyping and gradual refinement can occur. As the design complexity increases, it is important to obtain relatively accurate and quick results at the initial design phase. By providing a simple path that bridges the gap between high-level algorithmic concepts and a reasonable prototyping model, the opportunity for performing architectural exploration is greatly enhanced.

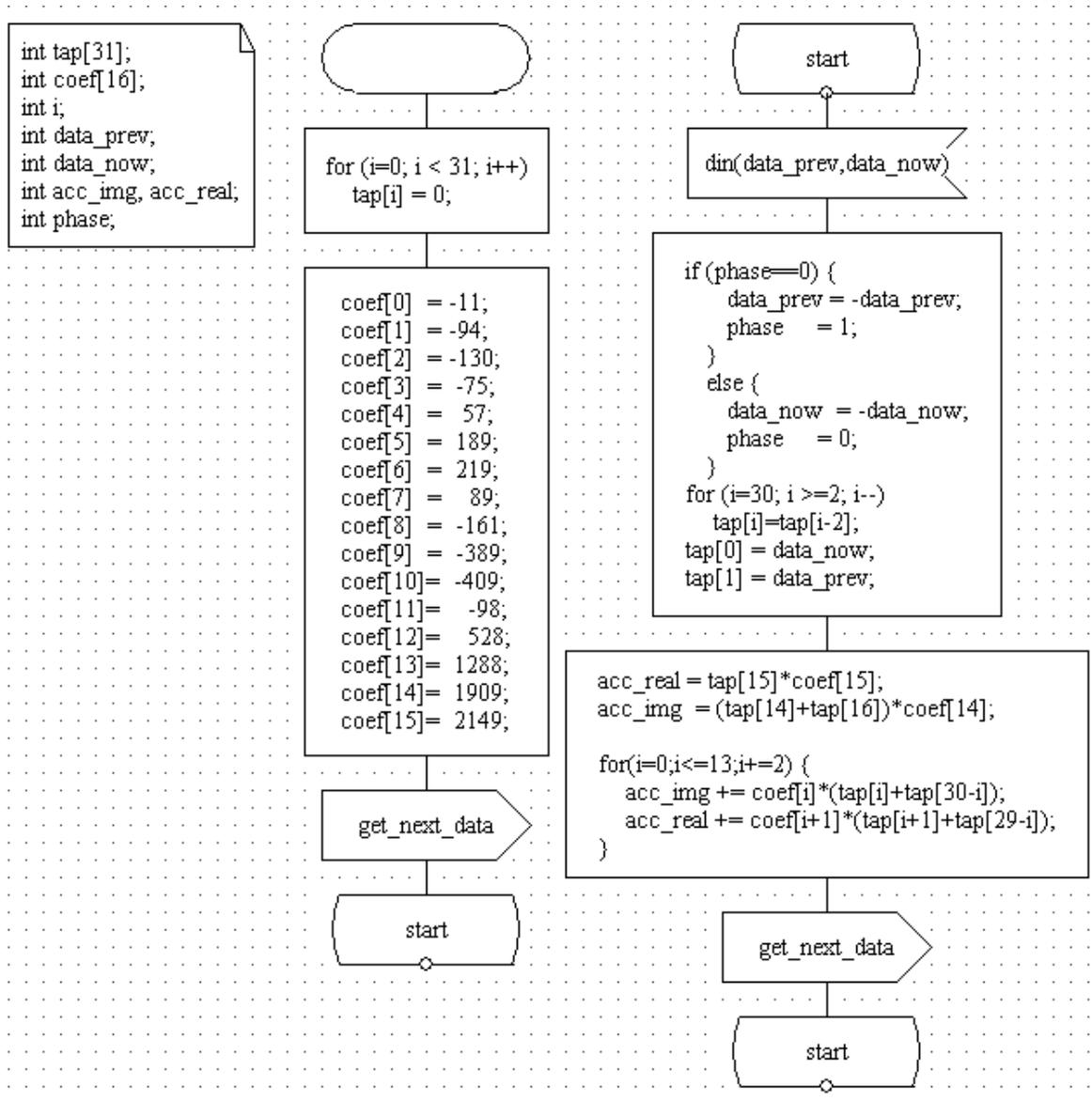


Figure 3: Matched Filter Specification

In general, architectural exploration intends to answer the following questions:

1. Does the proposed algorithm work?
2. If the algorithm is correct, how well does it work in terms of resource usage and the operating speed?

MAGIC-C answer the first question by providing a fast prototyping environment. Once the model is constructed, the user can verify the model by generating an executable file of the prototype that can run within the system.

To answer the second question, the user performs manual analysis on the model to determine the amount of resources it consumes and the latency of the system. Many

classic architectural-level *synthesis and optimization techniques* are applicable. See [Synthesis and Optimization of Digital Circuits](#), by Giovanni De Micheli.

It is important to understand the transparency of signals in MAGIC-C in order to construct a correct model. Essentially, any signal declared in a block is visible in all blocks and processes under it, unless the signal has been encapsulated inside a symbol.

In the example, due to its simplicity, global signals are used to illustrate communication between the processes. For larger designs, signals should be encapsulated inside symbols.

The model constructed so far is an *untimed* model. Like most tools for prototyping logic, combinational operation assumes zero delay for completion. Hence, from the system prototyping point of view, all five iterations of testing happen at the same time. While this model is useful for logic verification, the design needs to be refined further if it is to be implemented in hardware.

Adding Delay and Time to the Model

The timing details are one of the most important attributes of a system under design.

In this section, modeling some of the timing requirements refines the design. To do this, the following micro-architecture specifications are added to the design:

1. The data sent from `Data_generator` to `matched_filter` is synchronized with respect to the clock edge
2. The latency of the `matched_filter` cannot exceed 16 clock cycles

To support the clocking used in the system, we will use the clock data type in **MAGIC-C**. A clock signal can be scheduled to trigger periodically. It is achieved by using the `set()` built-in function in MAGIC-C. The `Data_generator` triggers the clock.

Since `Data_generator` now performs two distinct functions: sending data to `matched_filter` and generating a clock signal, it is desirable to partition it into smaller blocks.

The final structure of `Data_generator` is shown in Figure 4. It is made to be a block (instead of a process) which contains two parallel processes: `clock_gen` and `send_data`. The `clock_gen` process generates the clock signal, `clk_high`. The `send_data` is responsible for sending the data to the `matched_filter` with proper attention to the clock. Their structures are shown in Figure 5.

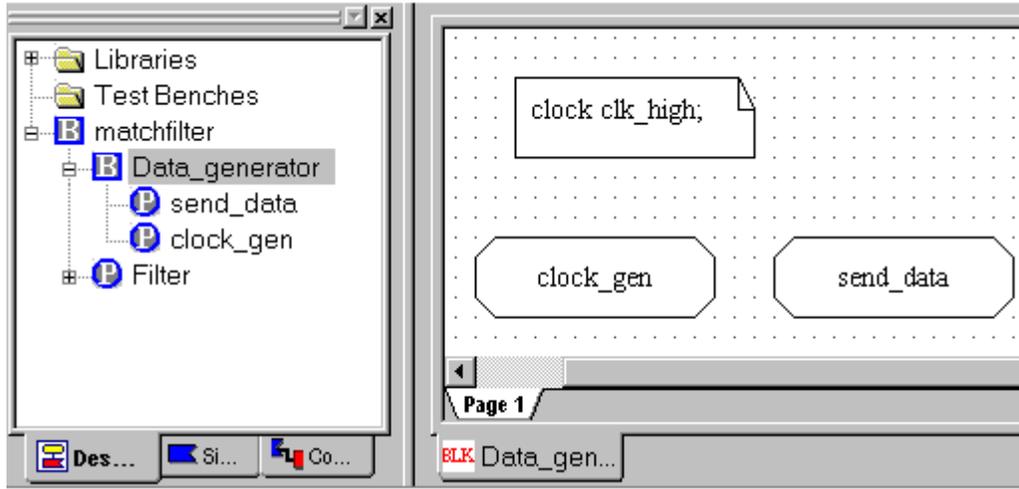


Figure 4: New Data_generator Block

In the `clk_gen` process shown in Figure 5, a timer `t` is used to allow a transition into a real state. The `set` command in the Task block indicates that `t` is triggered 1 prototyping time unit after current time (`now`). When the `clk_high` is set, it is triggered every 2 time units.

Note: The difference between a timer and a clock is that timer is only triggered once. However clock is triggered continuously once it is set (unless it is being reset). See the *Virtio Innovator User's Manual* for more information.

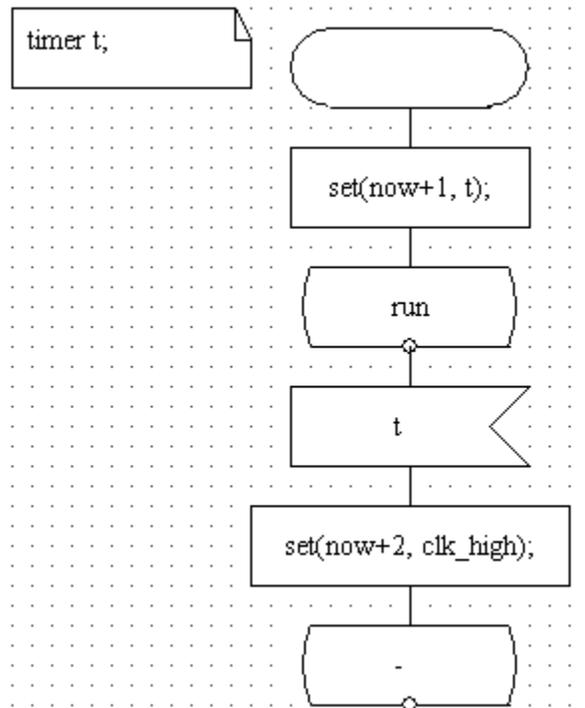


Figure 5 Description of clk_gen Process

The `send_data` process shown in (Figure 6) is very similar to the original `Data_generator` shown in Figure 2 except that it waits for the `clk_high` signal before sending the data out (`din`). By doing so, data is only sent to `matched_filter` at the clock edge.

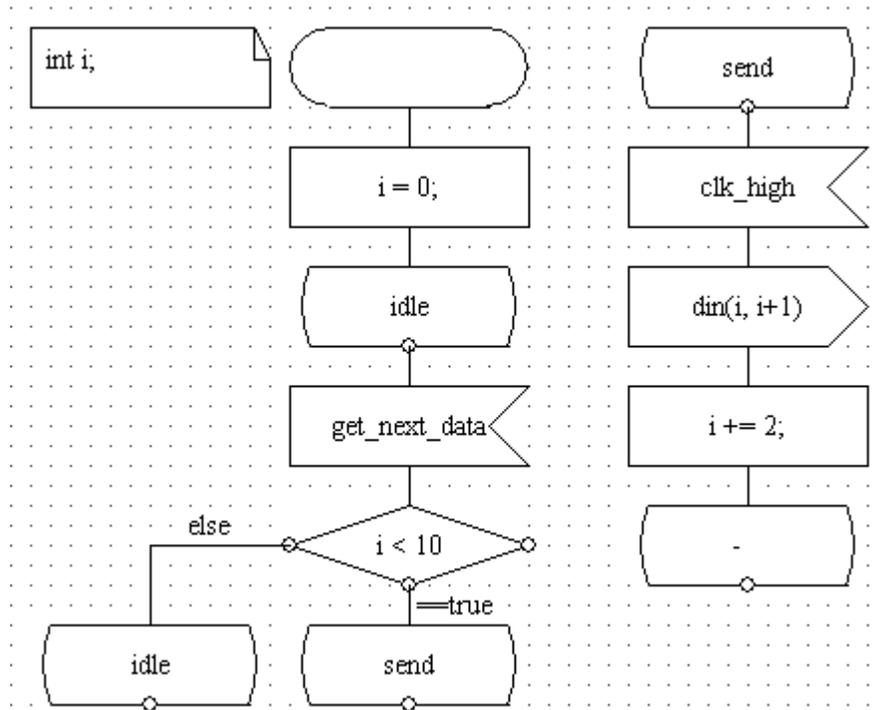


Figure 6 Structure of send_data Process

Using Reset in a MAGIC-C Model

This and the following sections illustrate how to refine `matched_filter` and react to changes in design specification.

Initialization in **MAGIC-C** is done in the Start state. While this is an intuitive approach, it is difficult to implement in hardware. Normally, initialization in actual hardware is done using reset signals. It is easy to *reset mechanism* in **MAGIC-C**, as illustrated in the revised design of the filter as shown in Figure 8.

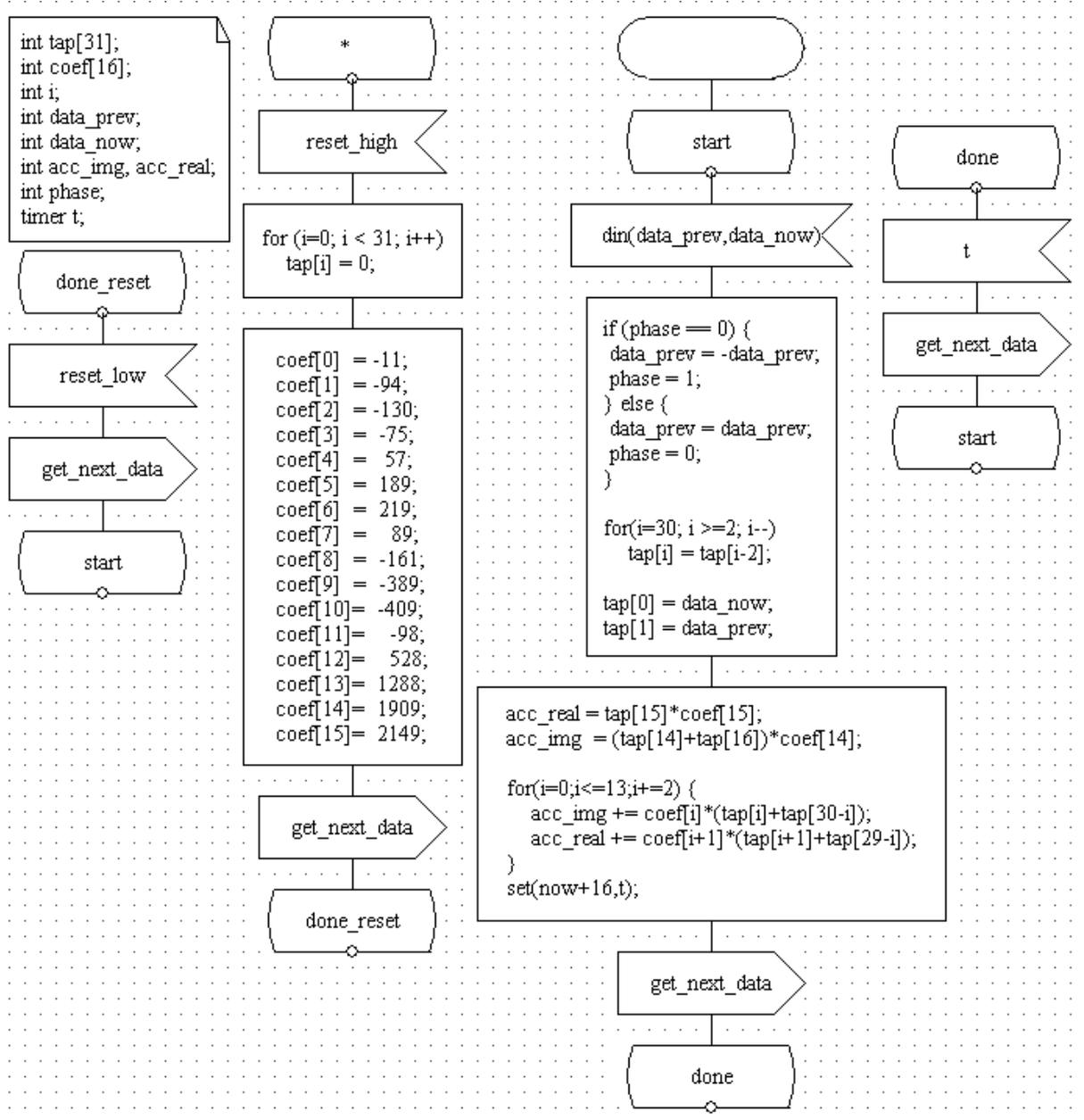


Figure 8 Revised Filter Design Using Reset

Note: The use of "*" state. It actually represents all the states in the FSM. Thus, in the example, whenever the `reset_high` signal is received, all the taps is set to zero, and the FSM transitions to the `done_reset` state upon receiving the `reset_low` signal. No additional initialization is needed now assuming that `reset_high` and `reset_low` are appropriately triggered when the system simulation starts.

Also note the use of explicit delay. After calculating the results, the filter does not send out the `get_next_data` signal immediately. Instead, it first waits for 16 cycles. In essence, this emulates the delay expected for the whole operation. This could be a useful modeling technique if the development of other parts of the system is well ahead of the current component. Integration and meaningful system testing can be done with good accuracy.

The setting of `reset_high` and `reset_low` are done in `clk_gen` process (for simplicity), and the modified design is shown in Figure 9.

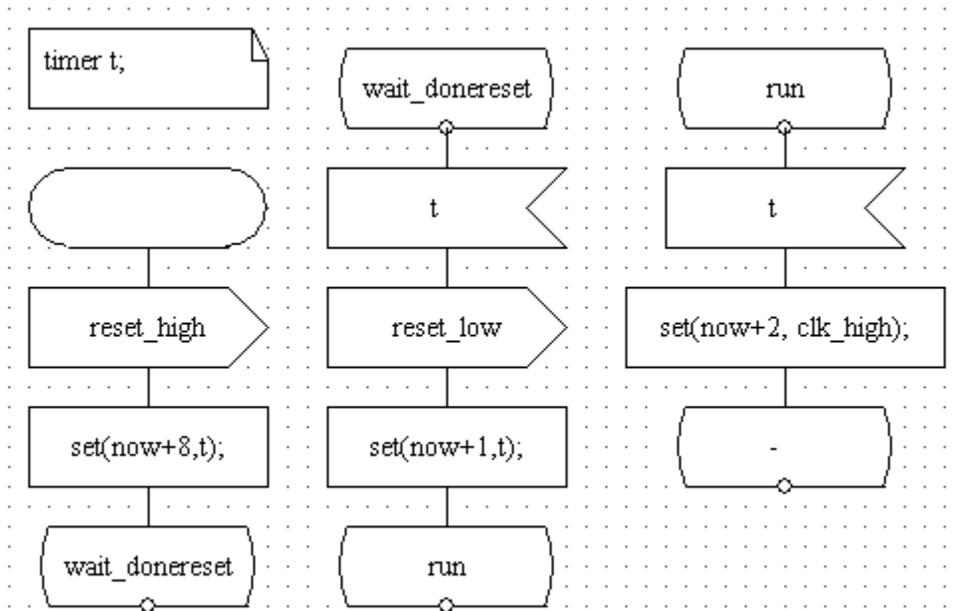


Figure 9 Revised Design of `clk_gen`

In the design, `reset` is asserted (by means of sending `reset_high`) for 8 clock cycles. After that `reset` is de-asserted (by means of sending `reset_low`) followed by asserting the clock.

Note: By using two signals, we can always model a level-sensitive signal like `reset`.

Handling Protocol Refinement in MAGIC-C

The final illustration demonstrates how to accommodate the change in communication mechanism used in the system. Assumes that the results of a micro-architecture study indicate that the neighborhood of `matched_filter` is very crowded, and the routing capacity around the area is very limited. Thus, is desirable to reduce the two dedicated buses (represented by two payloads of signal `din`) going into `matched_filter` to just one. Input data is given one at a time in different time phases.

Eliminating one of input buses has the following impact on `matched_filter`:

- Extra storage is required to store the first input data (and maybe for the increased number of states).
- Latency of the filtering operation is increased.

A designer would probably study these impacts carefully before committing to the change. Assuming the trade-off is desirable, the system is modified (as described below) to accommodate the micro-architectural change.

First, `din` is changed to carry one payload of integer instead of two. Second, the `send_data` process is changed, as shown in Figure 10, to model the two phases of data input.

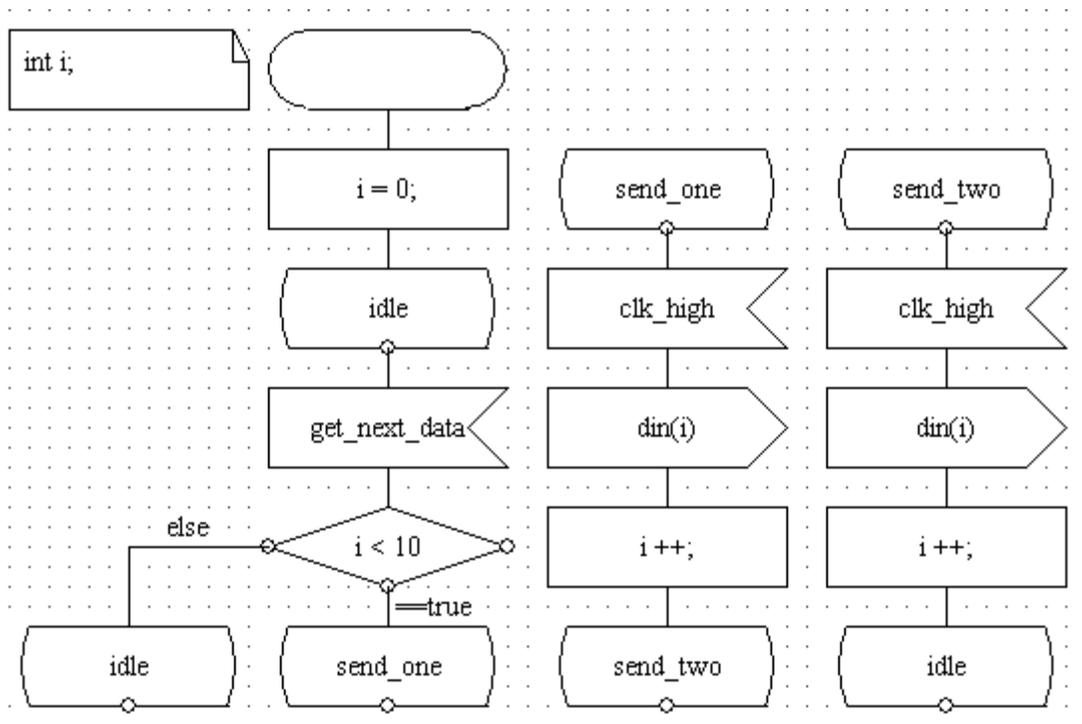


Figure 10 `send_data` Process Modified for Sending `din (int)`

An extra state, `send_two`, is introduced in the `send_data` process, and the original `send` state is renamed to `send_one`.

The change in the filter block is more extensive. Instead of changing it directly, it may be desirable to rethink the partition of the block. Currently, it is only represented using one **FSM**. Experiences in large design projects show that changes that occur at the later stage of the design happen mainly in the control part and interfaces between blocks. These changes are most likely to be caused by timing change, area concern and routing capability. While it is difficult to predict exactly the kind of changes that will take place, it is possible to design the system to reduce the impact caused by changes to system requirements. For example, `matched_filter` can be viewed as two separate parts:

1. The part that accepts input data from outside.
2. The part that processes the input data.

This may seem like a trivial view of the system, but implementation using this approach can be tremendously beneficial in certain cases. In our example, this approach would result in changes in the protocol only affecting the first part of the system.

Based on the this observation, the filter process is modified as shown in Figure 11.

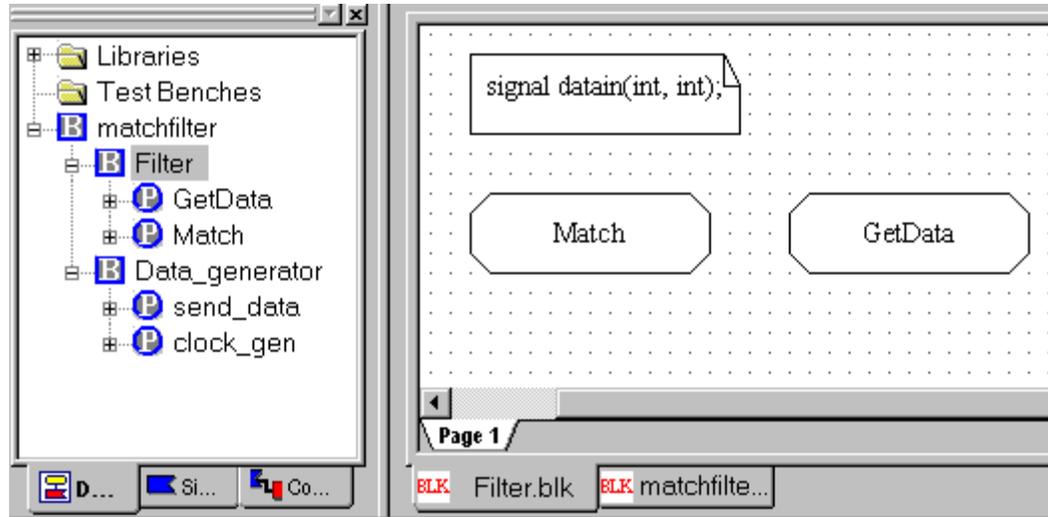


Figure 11 - Final Structure of the Filter Block

The original filter process is made into a block with two separate processes, *Match* and *GetData*. They communicate with the *datain* signal, which assumes the previous form of *din*, which carries two payloads.

The *GetData* process shown in Figure 12 is responsible for receiving input data from *Data_generator*. It assigns the input data to the payloads of *datain* during the different phases of the operation.

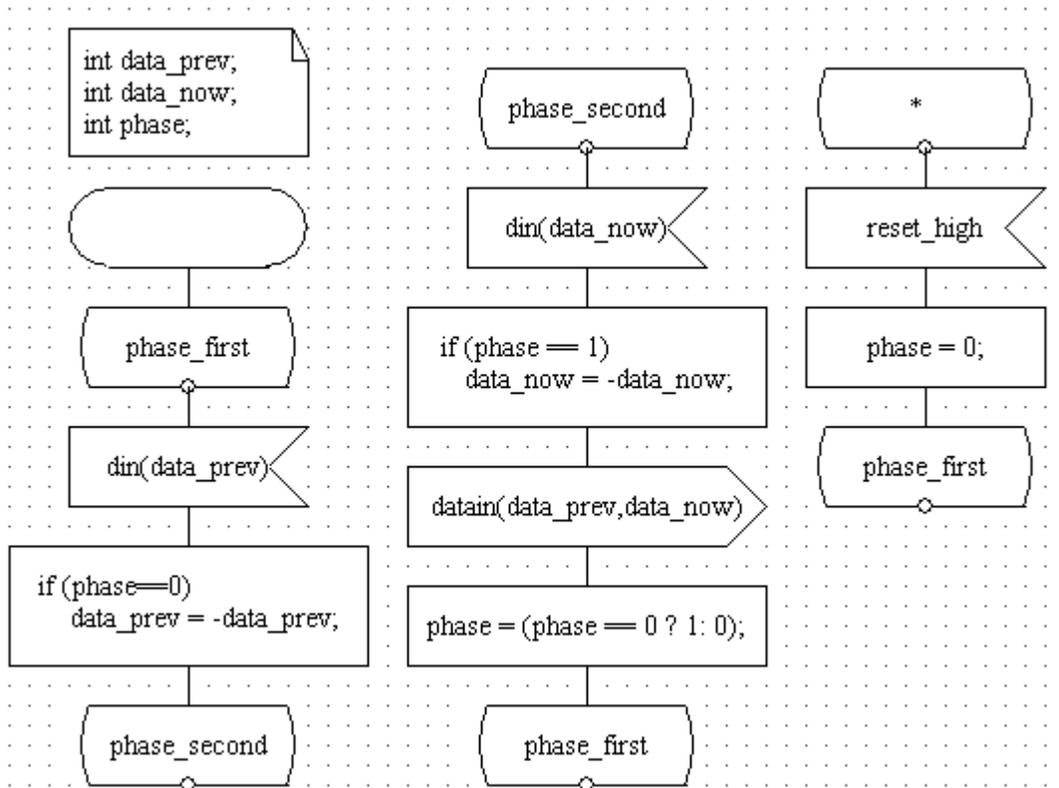


Figure 12 Phased Operation of GetData Process

The [Match](#) process is very similar to the original filter process. The original input signal `din` is replaced by `datain`. In the original filter process, the polarity of certain input data is changed according to the phase. In the new [Match](#) process, this function is moved to [GetData](#).

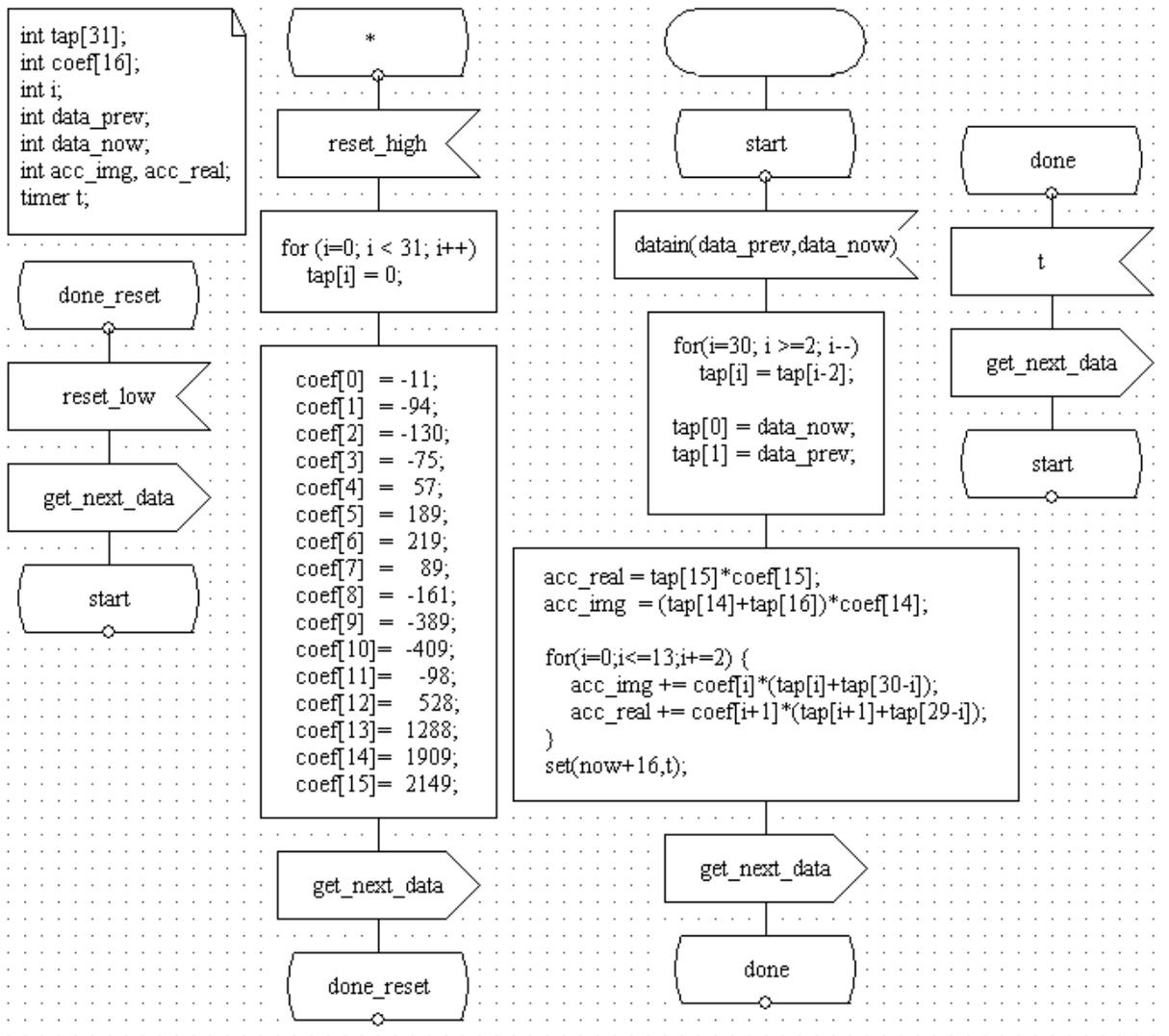


Figure 13 Match Process Using datain Signal from GetData

Summary

As illustrated in the example using `matched_filter`, **MAGIC-C** is a good language for high-level system design. The various constructs in **MAGIC-C** enable designers to model complex operations with relative ease. The communication mechanism used in the language is also quite simple to understand.

The Virtio Innovator is an ideal platform for doing **MAGIC-C** design. By integrating design entry, advanced debugging capabilities, very fast prototyping and code generation capabilities in a single environment, a designer's productivity is greatly enhanced.

UAR Design

UAR Design

Using MAGIC-C

In this chapter, we will design and test a simple Universal Asynchronous Receiver (UAR) using the Virtio Innovator. First, using the Virtio Innovator and the Test Bench builder, we will create a framework of GUI and non-GUI test benches. When the UAR is finally written in MAGIC-C, it should work seamlessly with the test benches. We will concentrate primarily on refining our design framework so that when the UAR is plugged into the design, we can easily determine if the UAR is functioning correctly. The increasing levels of refinements to our design and test framework constitute the bulk of this chapter. A good knowledge of Virtio Innovator Test Bench controls is assumed. As a final step, we will describe the UAR at Register Transfer Level (RTL). Note that for simplicity, we will just sketch the basic modules of the design; you can refer to the UAR Design Example directory for complete details about the design.

Introduction

The following topics are covered in this chapter:

1. [Universal Asynchronous Receiver Specification](#)
2. [Basic Test Bench Framework](#)
3. [Refinement Steps for the UAR Framework](#)
4. [Specification of UAR Framework \(initial version\)](#)
5. [Adding Delay to Image_gen model \(version 2\)](#)
6. [Data Transfer in BYTES \(version 3\)](#)
7. [Data Transfer in BITS \(version 4\)](#)
8. [Using UAR for Serial Data Reception \(Final Version\)](#)
9. [Summary](#)

Universal Asynchronous Receiver Specification

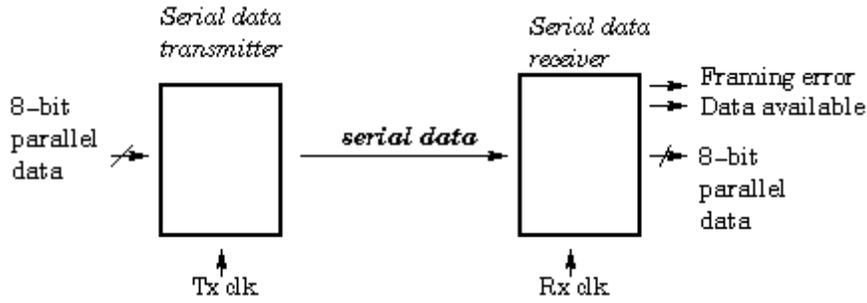


Figure 1: Universal Asynchronous Receiver Model

Figure 1 shows a simplified picture of an asynchronous serial interface of the type commonly used to transfer data in computer and communications systems. The mode of data transfer between the transmitter and the receiver is referred to *asynchronous* because the transfer over a serial transmission link is not controlled by or locked to a common clock. The clock controlling the transmission rate (*Tx clock*) and the clock to synchronize the receiver (*Rx clock*) are nominally of the same frequency, but are generated locally at each end of the transmission link and therefore cannot be assumed to be *locked*.

The spacing between the transmitted characters (represented by 8-bit data) may be of any length. In contrast, the timing of the bits within the character is well defined (and is related to the bit/ baud rate of the interface). The receiver must be able to detect the start of an incoming character and then store the value of each data bit, despite the fact that the relative frequency and phase of the *Tx* and *Rx* clocks may vary. In this example, we will concentrate on developing a RTL level UAR; the transmitter will be an abstract representation for simplicity of modeling.

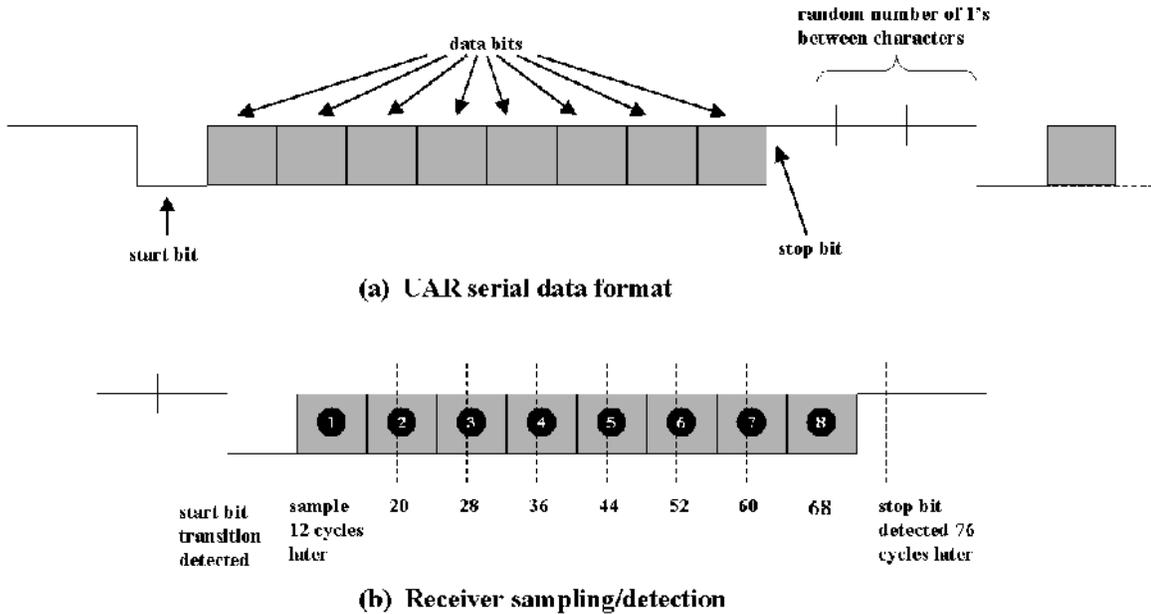


Figure 2: Data Format And Sampling Detection For The UAR

As shown in the data diagram in Figure 2 (a), the beginning and end of each character is delimited by a start bit whose value is always 0, and a stop bit whose value is always 1. In between characters, the transmitter outputs a constant value of 1. In operation, the receiver continually samples the input data.

Following a 1→0 data input transition, the eight data bits must be stored. Storing the data bits reliably is a potential problem, since for maximum reliability we wish to sample the data bits in the center of their bit times and not close to either edge, so that small differences between the *Tx* and *Rx* clocks can be accommodated. This may be accomplished by using an *Rx* clock frequency that is a multiple of the data bit rate. In our UAR design, we shall assume that the *Rx* clock signal is **eight times** the bit rate.

Following the detection of a start bit, the first data bit is input at the center of the bit time 12 clock cycles later as shown in Figure 2 (b). The stop bit should be detected 76 clock cycles later as shown in the figure. If so, the **Data Available** output is set high; if not, the **Framing Error** output is set. Both status outputs are reset low by the detection of the next start bit.

Change in Specification: Noise Resistant Behavior

Danger of spikes is suspected on the communication channel falsely starting the receiver. This means that a momentary LOW on the input to the receiver would be seen as a one-to-zero transition whereas it is really just noise.

To counter this, the specification is changed as follows:

The start bit is a one-to zero transition where the input signal is still zero four (or three or five) samples later.

Thus, the design should handle the above bit pattern to detect a valid start bit. The flexibility of 3-4-5 clock periods in the specification allows the simplest implementation. For this design, we use 5 clock periods after a high on the input.

In the following sections, the main steps in doing a top-down design in MAGIC-C will be given.

This UAR based design will probably of medium complexity because we need to model both a transmitter and receiver. So, we choose to use MAGIC-C symbols for representing every module in the design. Specifically, no shared signals will be used for communication protocol modeling.

Basic Test Bench Framework

Since the UAR is a basic component for a serial data receiver, we first plan to model such a receiver at an abstract level. Let's call this process `Receiver` with the UAR as a sub-module for data reception and conversion.

Now, the `Receiver` might potentially be any data-processing device. We arbitrarily choose to model a device that accepts frames of *video data* and displays it on a `LCD` screen. For simplicity, we will assume that the video data is raw, uncompressed data.

For our purposes, a video data frame will be represented by a **bitmap** with following information: width of the bitmap, its height, number of bytes needed to represent each pixel, size of the frame data (in bytes), and finally, a pointer to the frame data.

Also, we will, as a first approximation, assume a one-way handshaking mechanism for the `Receiver`. That is, after receiving each frame of data, it will send out a `DataReceived` signal as an acknowledgement to the (Universal Asynchronous) Transmitter that sent the data.

So, the `Receiver` should have one pin for an input signal, say

```
RecvData(int/*FrameWidth*/,
         int/*FrameHeight*/,
         int/*FrameNPixels*/,
         int/*FrameSize*/,
         BYTE* /*DataBuffer*/)
```

and an output signal `DataReceived` to indicate receipt of a frame of data.

Based on this observation, it is obvious that we need to model another system that will generate some data to the `Receiver` if we intend to test the final model. Let's called this system `Image_gen` that generates some video image frames to be used for testing the `Receiver`.

With the above information, we can describe the first cut system description of the UAR framework in MAGIC-C as shown in Figure 3.

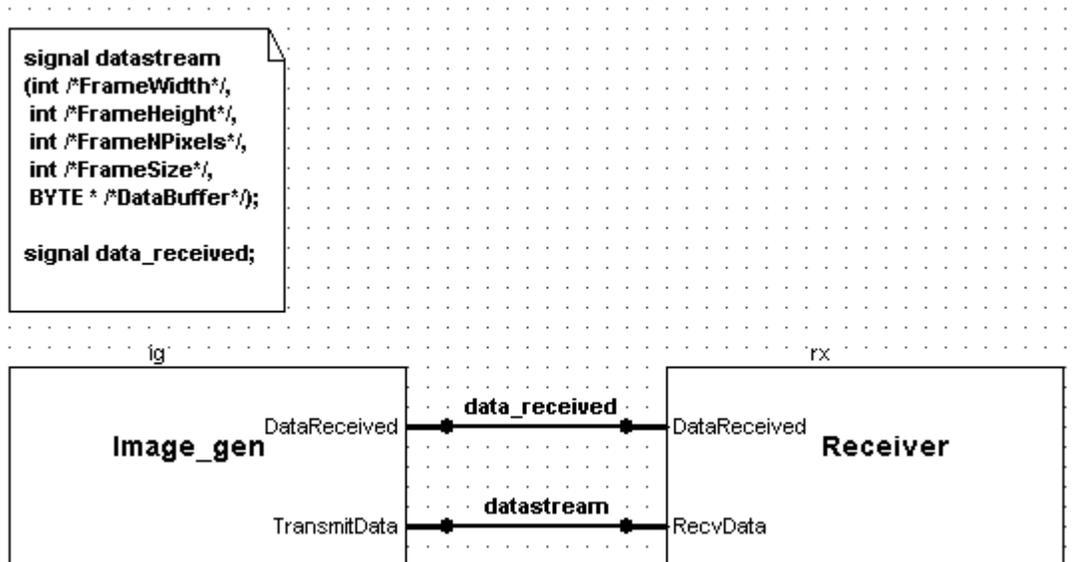


Figure 3 – UAR Framework.

Communication Protocol

Note the simple protocol for data exchange between both the modules: when `Image_gen` has video frame data to be sent to the `Receiver`, it sends the signal `datastream` (with the data as its payload) through its pin `TransmitData` to notify the `Receiver` on its `RecvData` pin.

The `Receiver` then proceeds to process the data. At the same time, `Image_gen` will wait for the operations to complete. When the `Receiver` has finished processing the current data frame, it sends a `DataReceived` signal to notify `Image_gen` that it is ready to accept new data.

User-Defined Data Type Across Model Interfaces

In the payload of the `RecvData`, the frame data is represented by a pointer to a `BYTE` array. Now, since `BYTE` is not a built-in data type, you can define a new data type by editing the `user.h` and adding

```
typedef unsigned char BYTE;
```

In general, you can define any new data type required to model your design. We will see later how we can define additional data types and C++ templates to model designs of increasing complexity.

Refinement Steps for the UAR Framework

We intend to refine the UAR Framework in stages using various constructs of **MAGIC-C**, with each stage showing refinements in one or more of the following aspects:

- **Data types** used in the design
- **Timing** constraints of various components
- **Communication protocol** among components
- **Modeling abstraction** of components (whether behavioral, algorithmic, RTL etc.)

Table 1 indicates the refinements performed in five stages. The details and rationale for the refinements will be discussed in the following sections.

STAGES	REFERENCES	REFINEMENTS
Initial Version	Section Specification of UAR Framework (initial version)	<ul style="list-style-type: none"> • Untimed model of the UA Transmitter (Image gen) and Receiver (Receiver module). • Data transmitted consists of a pointer to video frame data. • Communication between modules using Handshaking Protocol.
Version 2	Section Adding Delay to Image_gen model (version 2)	<ul style="list-style-type: none"> • Timing added to UA Transmitter (Image gen). A rough clock period is used for the transmitter. Receiver is still untimed. • Handshaking no longer used for communication. Transmitter broadcasts data independently of Receiver.

Version 3	Section Data Transfer in BYTEs (version 3)	<ul style="list-style-type: none"> • Data transmitted in the serial link is in units of a user-define data type BYTE. • Communication protocol enhanced; Transmitter indicates beginning of a video frame by sending BeginFrame signal. • Clock period of Transmitter adjusted for handling detection of beginning of a video frame by the Receiver.
Version 4	Section Data Transfer in BITs (version 4)	<ul style="list-style-type: none"> • Data transmitted in the serial link is in units of another user-define data type BIT. • Clock period of Transmitter adjusted further for handling detection of beginning of a video frame by the Receiver. • A high-level model of UAR added to the design.
Final Version	Section Using UAR for Serial Data Reception (Final Version)	<ul style="list-style-type: none"> • Transmitter made compliant to a UAR specification. • Timed version of Receiver modeled. • A timed, RTL model of UAR replaces the high-level model.

Table 1: Refinements For The UAR Design

Initial version

Specification of UAR Framework (initial version)

With the basic system partitioned and the handshaking communication protocol defined, we have enough information to complete the first draft of the design.

This version relies on the [Image_gen](#) module to send a pointer to the bitmap data that is processed by the [Receiver](#) module. So, there is additional code to allocate memory for

the bitmap data in both the modules. Also, as a space optimization, if the previous frame has the same dimensions as the current frame, we reuse memory allocated for the previous frame to store the new frame information. These details are not central to the design and will not be discussed in the following sections.

Image Generator Specification

We model the `Image_gen` module such that we can control operations of this module via an associated Virtio Innovator Test Bench, say `Send_data.tb`. Note that by using the Test Bench, we can also test in a prototype the *asynchronous* behavior of a data generator by starting and stopping it at any point in time.

For our purposes, we will send six LCD frames from `Image_gen` one at a time. To store the frames, declare the **VS_LCD** variables `frame1`, `frame2`, ..., `frame6`. We then connect these variables with six corresponding LCD/Bitmap Test Bench controls in the Virtio Innovator Test Bench `Send_data.tb`.

Also, to indicate to the user the current LCD frame being processed, declare the **VS_int** variables `led1`, `led2`, ..., `led6` to represent LEDs, one for each frame. Then, these **VS_** variables are connected to Test Bench.

The Figure 4 shows a portion of the Test Bench where the **VS_LCD** and **VS_int** variables have been connected to various Test Bench controls.

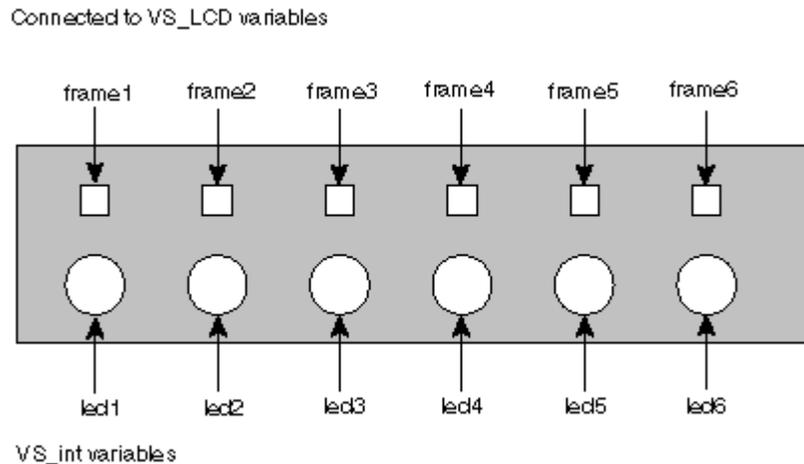


Figure 4: Connecting Variables To Test Bench Controls (send_data.tb)

To control the operations of the `Image_gen` from the Test Bench, define two internal MAGIC-C signals: `CopyImage` to start sending of each frame and `SenderReset` to stop any ongoing activities. Resetting will also send out an "empty" frame with frame data containing only 0s.

To send out the above signals `CopyImage` and `SenderReset` from the Test Bench `Send_data.tb`, create two signal buttons named **transmit** and **Reset** and connect these

buttons to the two signals respectively, so that the Test Bench appears as shown in Figure 5.

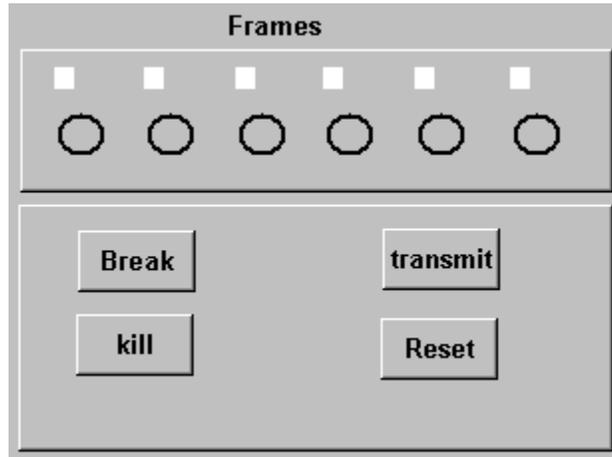


Figure 5 - Send_data.tb Test Bench.

The process begins with the `start` state. After performing some initializations like settings all LED's to 0, `Image_gen` will enter the `wait_for_transtb_sig` state.

When the user clicks on the **transmit** button in the Test Bench, signal `CopyImage` is sent to `Image_gen`. On receipt of this signal, it starts transmission of frames. After transmission of each frame, it goes to the `wait_recv_ack` state to wait for the acknowledgement from the `Receiver` as shown in Figure 6.

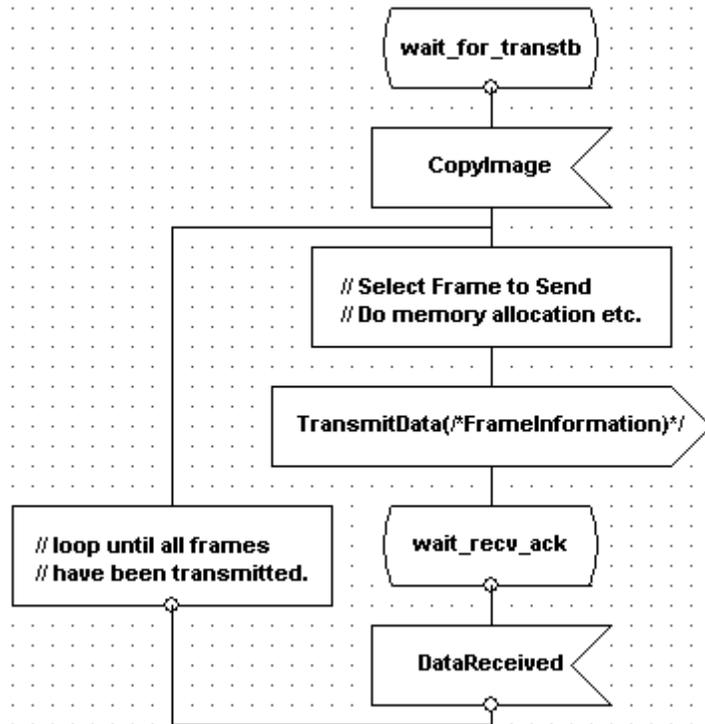


Figure 6: Simplified Description Of Image_gen (Version 1)

After receipt of the `DataReceived` signal from the `Receiver`, `Image_gen` continues with transmission of the next frame. Refer to `Uar_Design_Example\uar_step1, Image_gen.prc` (pages 2 and 3) for more details.

At any point while running a prototyping session, if the user clicks on the **Reset** button, `SenderReset` signal is sent. `Image_gen` will send an “empty” frame containing all 0’s and enter the `Wait_for_transtb_sig` State again (`Uar_Design_Example\uar_step1, Image_gen.prc`, page 1).

Receiver Specification

Like the `Image_gen` module, the `Receiver` module is to be modeled such that we can control operations of this module via another associated Virtio Innovator Test Bench, say `Receiver.tb`. Using this Test Bench, the `Receiver` can be operated *asynchronously* at any point in time.

The `Receiver` receives each frame of video data and stores it in **VS_LCD** variable `recv_frame`. This variable is connected with a LCD/Bitmap Test Bench control in the Virtio Innovator Test Bench `Receiver.tb`.

The `Receiver` begins with a `Start` state. After performing some initializations, `Receiver` enters the `wait_for_transmitter` state.

On arrival of `RecvData` signal, it processes the bitmap data. After the processing is over, this module sends back the `DataReceived` signal to acknowledge receipt of a data frame. A simplified version is shown in Figure 7. Refer to `Uar_Design_Example\uar_step1, Receiver.prc` for implementation details.

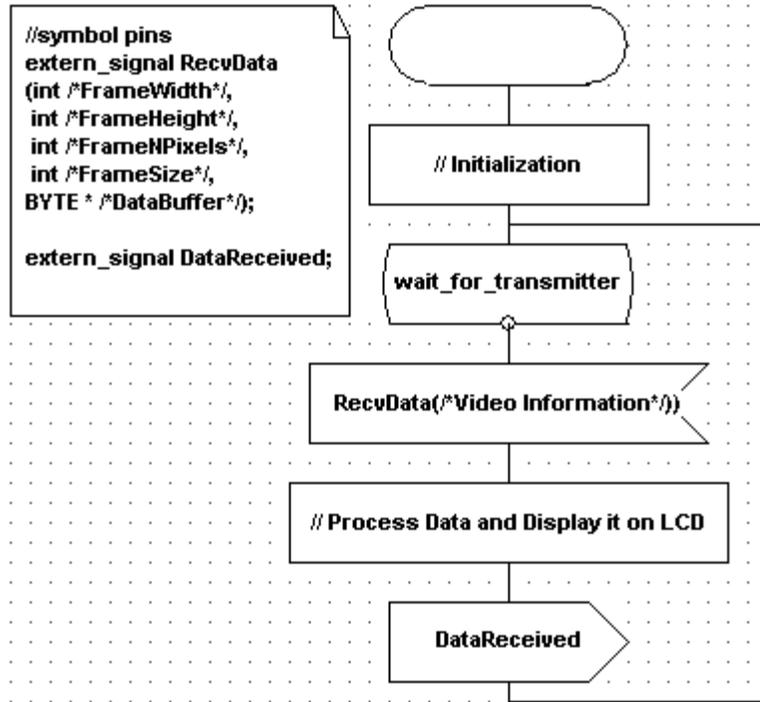


Figure 7: Simplified Receiver Model

Simulating the design shows that the [Image_gen](#) and [Receiver](#) behave as expected when frame data transmission is done. The Figure 8 shows the receiver displaying the second video frame. Note that the system prototype can be run at a very high speed because it has been modeled at a high level of abstraction.



Figure 8: Second Video Frame Being Displayed

The model we constructed so far is an *untimed* model where all frames are sent and received at the same time step. Furthermore, we rely on an *artificial* handshaking protocol for reliable transfer of data; no handshaking should be required for a serial data transfer.

Version 2

Adding Delay to Image_gen model (version 2)

As mentioned in the previous section, `Image_gen` should not wait for an acknowledgement from the `Receiver` before sending the next frame of data. The `Receiver` might in fact be non-operational when the data is being sent. We will perform *communication protocol refinement* between the two modules by removing the handshaking protocol needed in the previous version of the design. We will also create a timed model of `Image_gen` by adding delay to the model.

First, we create a new module called `clkgen` as shown in Figure 9 which generates a periodic clock of a given period (default = 80 time units). The operation of this module can be controlled by `reset` and `enable` pins. The clock period is a **VS_PARAM** and can be modified on a per-instance basis at compile-time (**VS_PARAMS** are run-time constants).

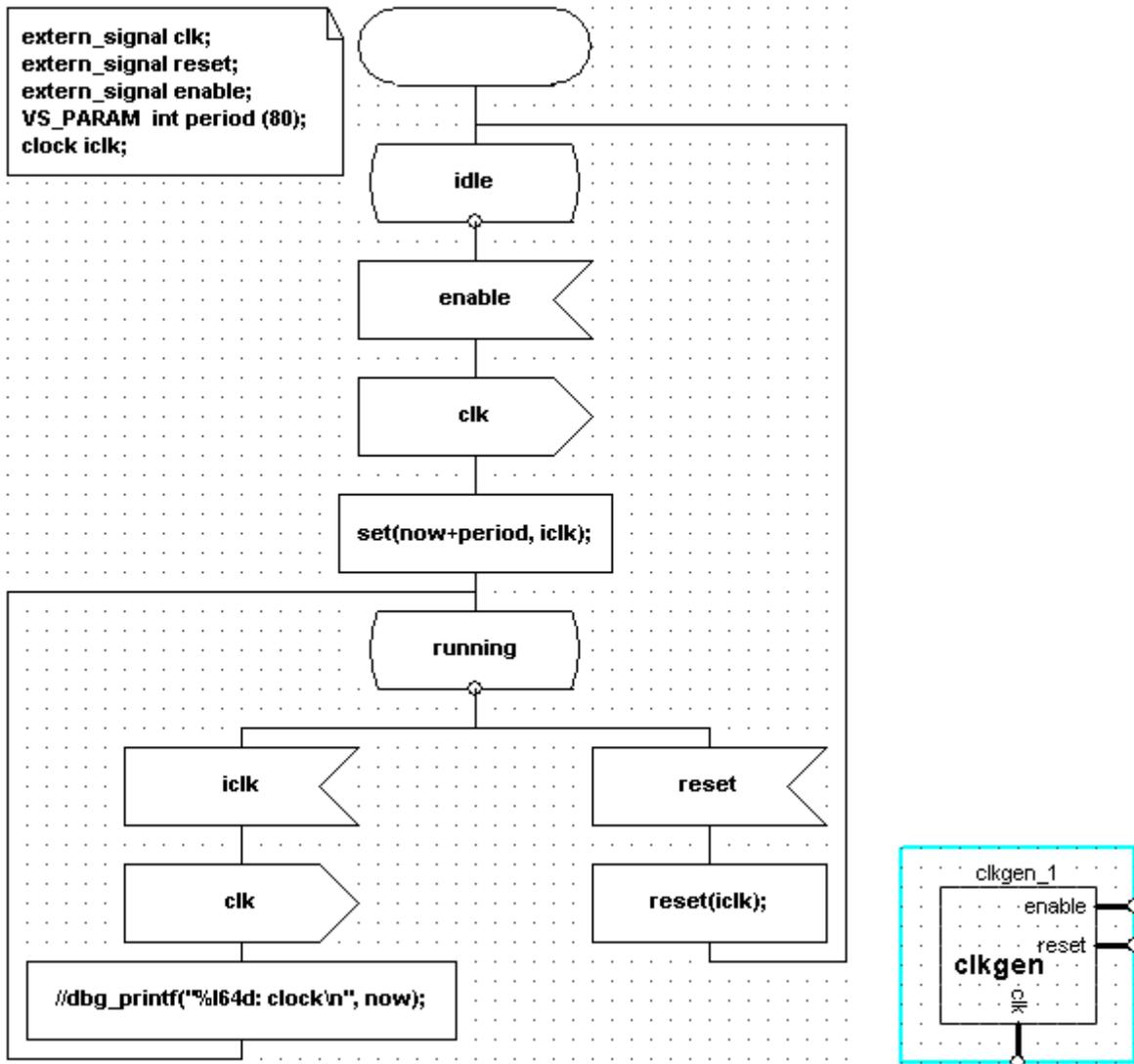


Figure 9 - `clkgen` process (a), and `clkgen` symbol (b)

First, we create another module called a `Data_generator` that contains a modified `Image_gen` connected to a `clkgen` module as shown in Figure 10.

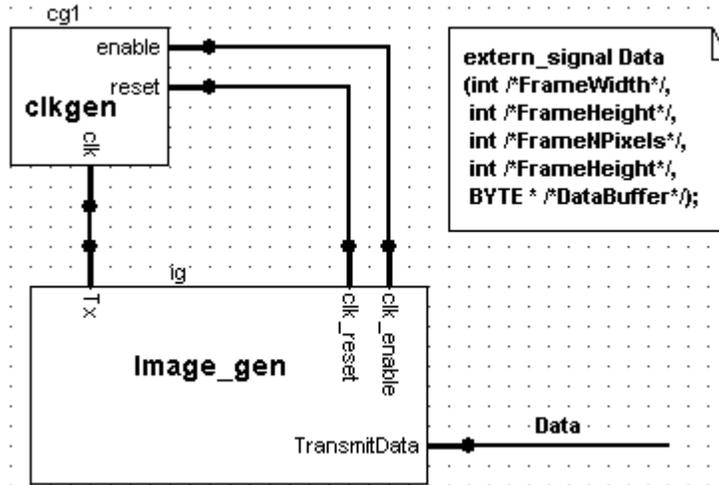


Figure 10 - Image_gen connected to a clock.

The `Image_gen` module now no longer waits for an acknowledgement from the `Receiver`. Instead, `Image_gen` transmits a frame every time its `Tx` port is triggered by the `clkgen` instance `cg1` as shown in the above figure. Accordingly, the modified description of `Image_gen` is as shown Figure 11. Contrast this description with that of version 1 (Figure 6). The clock period requirements of instance `cg1` is calculated as follows:

INFORMATION	SIZE (IN BYTES)
Frame Width	4
Frame Height	4
No of bits per Pixel	4
Frame Size	4
Frame Information	236472
Total Number of bytes per frame	236488

Table 2 - Video Frame Information.

Assuming a bit-rate of 1 bit / 80 clock cycles, transferring 1 byte needs 640 clock cycles; so a frame can be completely sent in $236488 * 640 = 151.352.320$ clock cycles.

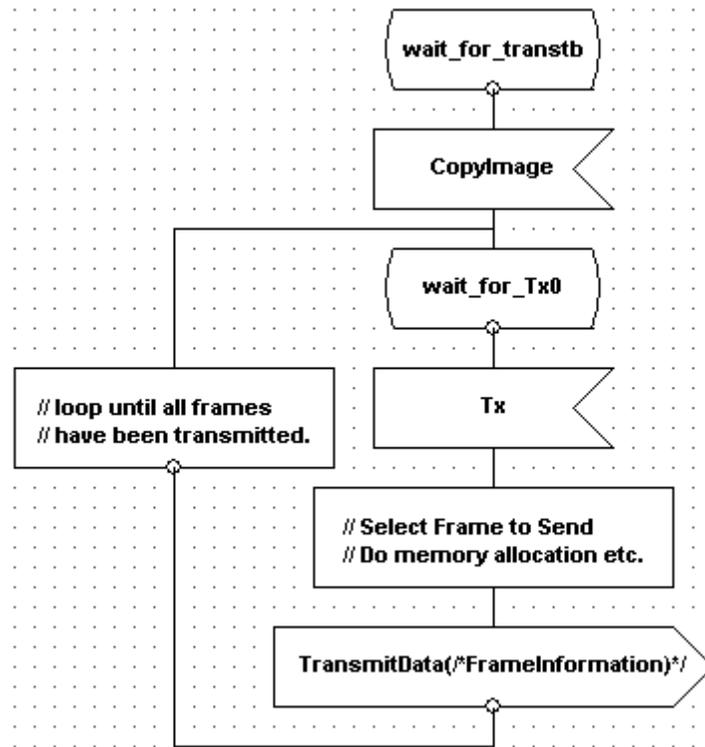


Figure 11 - Simplified description of Image_gen (version 2).

Correspondingly, *Receiver* does not send an acknowledgement (*DataReceived* signal) anymore on receipt of a data frame. Also, because we know that the actual *Receiver* will contain a UAR, we put the *Receiver* module inside another module called *uar_receiver*. For details, see *Uar_Design_Example\uar_step2*.

Version 3

Data Transfer in BYTEs (Version 3)

In both versions 1 and 2 of the design, the data types used for transfer was highly abstracted; we sent a list of information as shown in Table 2 as payloads to *TransmitData* signal. In particular, the data frame information was sent as a pointer to a BYTE array. For sake of discussion, let's refer to such signals (with video data frame information) as *parallel data*.

In actual devices, we send binary data over an established serial link. So, we will perform *data type refinement* by sending and receiving frame data in units of a BYTE rather than *parallel data* as in version 2. Note that we have defined BYTE as a user-defined data type in *user.h*, as described in section [User-Defined Data Type Across Model Interfaces](#).

Now, the version 2 of the design used only *parallel data*/signals. To reuse existing modules from this version of the design, we can plug in a *parallel data-to-BYTE* converter as a back-end in the existing *Data_generator*. Essentially, the new *Data_generator* transmits in terms of BYTES.

Similarly, we can plug in a reverse *BYTE-to-parallel data* as a front-end in the `uar_receiver`. This makes `uar_receiver` capable of receiving BYTE data. These converter modules should then be placed as shown in Figure 12. The converters have been shown by grayed rectangles.

Also, note that since the data transfer is in BYTE, the receiver needs to identify the beginning of a video frame. The `Data_generator` sending a `BeginFrame` signal accomplishes this; the receipt of this signal to `uar_receiver` indicates beginning of transmission of a new frame data. So, both the converters have an extra pin to send/receive the `BeginFrame` signal.

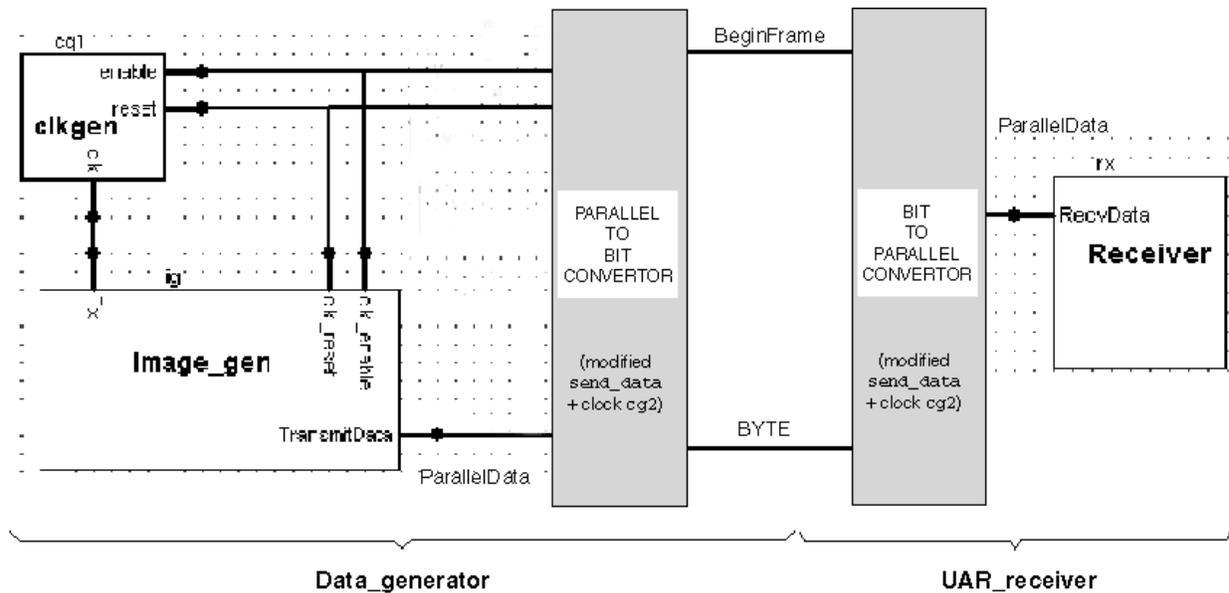


Figure 12: Parallel ↔ BYTE Converters In Version 3 Of The UAR Design

Now, before designing the converters, we need to write some utility functions to split an unsigned integer (4 bytes) into 4 1-byte blocks and combine 4 1-byte blocks to a single 4-byte chunk (integer). These functions can then be used to split integer payloads like Frame Width, Frame Height etc. in the `Data_generator` for transmission over the data link. On receipt of 4 consecutive 1-byte blocks, we can combine them to obtain the specific data again in the `uar_receiver`. For sake of performance and re-usability, we define two *C++ template functions* in `user.h`.

```
template <class F, class T> int unpack(F val, unsigned int
val_size, T arr[], unsigned int ele_size);

template <class F, class T> int pack(F const arr[], unsigned int
ele_size, T &val, unsigned int val_size);
```

The `unpack` function does the splitting into blocks of specified size. The `pack` function combines multiple blocks to create a bigger chunk.

By using this functions, the design of the converters become simple.

Parallel Data-to-BYTE Converter

The design of Parallel Data-to-byte converter is simple. We first create a module, say `Send_data` that splits the parallel data received from `Image_gen` into BYTES and then transmits it. A clock, say `cg2` connected to it, should drive this module. Then, every time the clock `cg2` connected to `send_data` is triggered, we send one BYTE of information. Such a converter is shown in Figure 13 (a).

The Figure 13(b) shows how `send_data` packs and sends frame width information. Here, we assume that the parallel data has been already received and variable `FrameWidth` has been initialized to the correct value. Then, using the `unpack` template function described above, we unpack the value of `FrameWidth` (32 bits) into 4 8-bit chunks which are stored in the `bytebuf` array. Similar processing is done for all other payloads of the parallel data such as frame height, frame size etc.

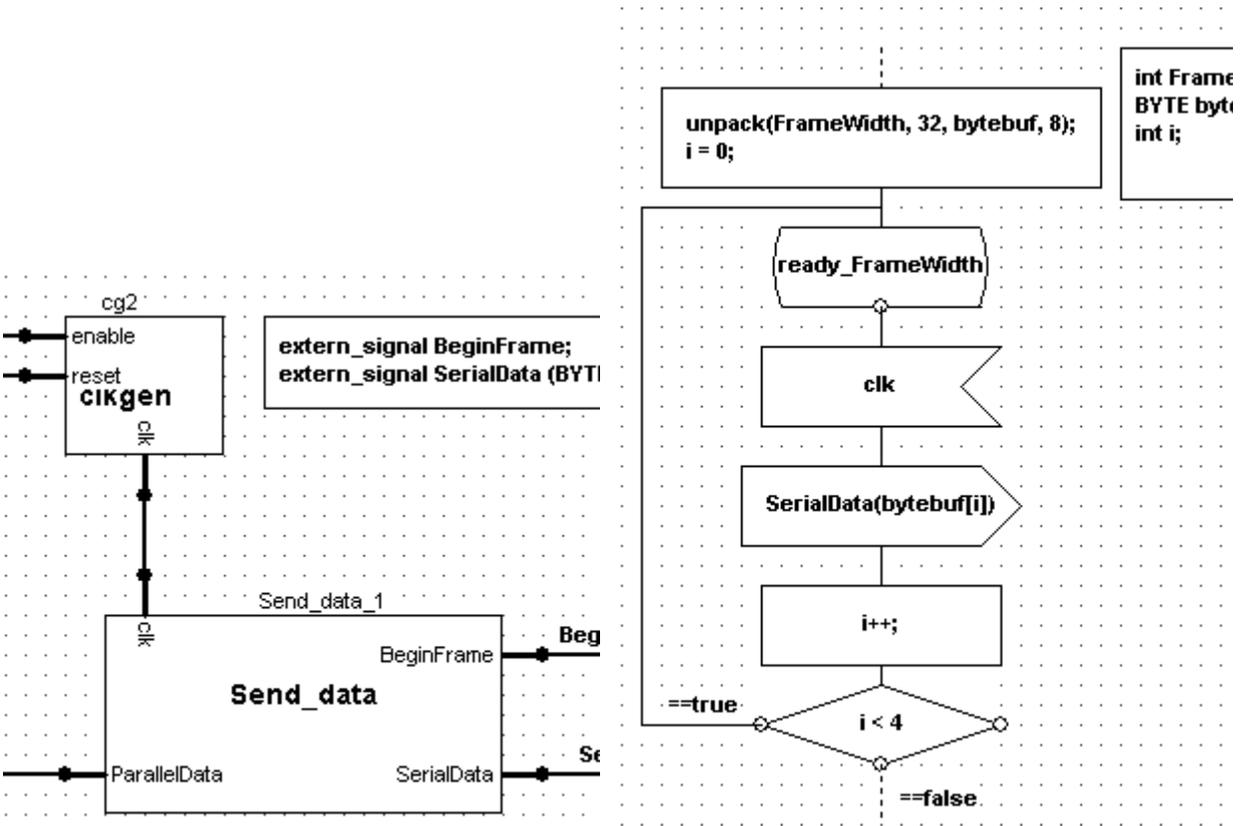
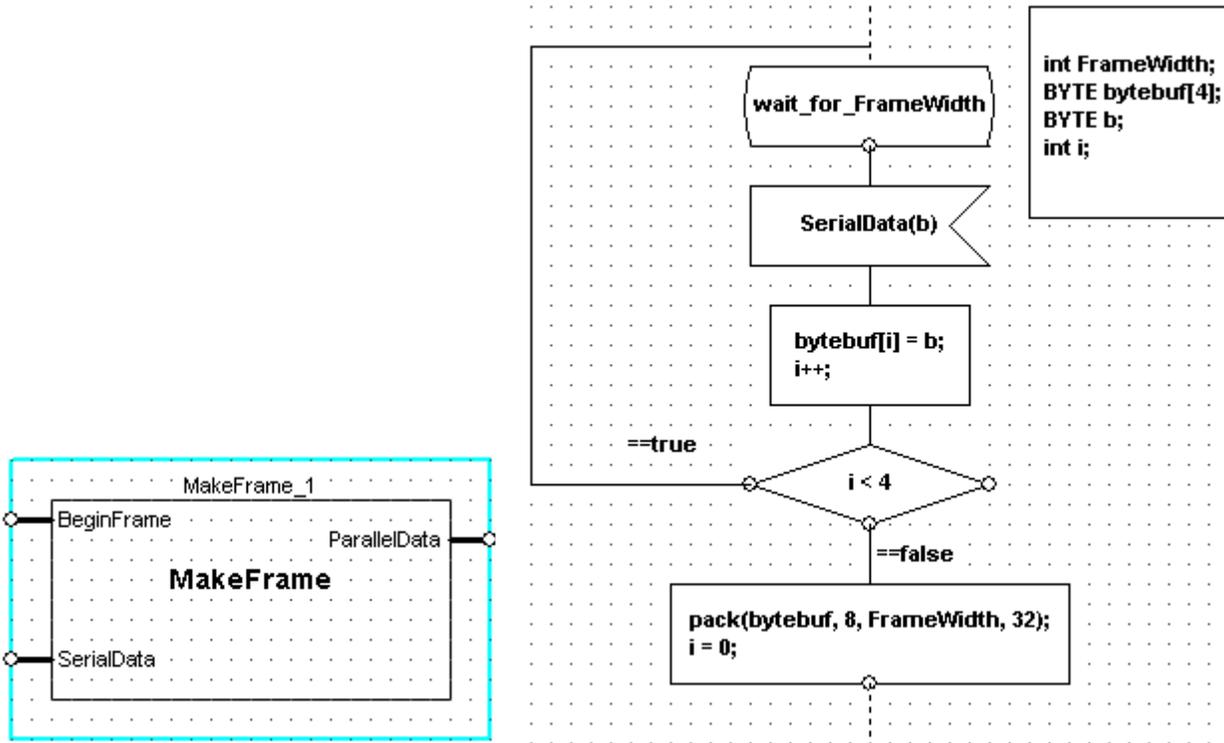


Figure 13 Parallel data -to-BYTE converter (a), and `Send_data` process (b).

BYTE-to-Parallel Data Converter

The BYTE-to-Parallel Data Converter module, say `MakeFrame` is also very simple. Each BYTE constituting a payload of the parallel data, say `FrameWidth` is temporarily stored in `bytebuf` array as they arrive from the `Data_generator`. When all the four bytes are

received, the array is packed and the value is assigned to `FrameWidth`. Similar processing is done for all other payloads. This `MakeFrame` converter is shown in Figure 14. Finally, the parallel data so retrieved from BYTES is sent to `Receiver`.



**Figure 14 - BYTE-to-Parallel Data symbol (a) and how it packs `FrameWidth` from four bytes (b).
Modified Clock Period Requirements**

There are now two instances of `clkgen` connected in `Data_generator` module; clock `cg1` is connected to `Image_gen` module (Figure 12) as in version 2 and clock `cg2` has now been connected to the `send_data` module (Figure 13(a)). We identify/refine the clock period of both the clocks here.

Note that we are sending another signal `BeginFrame` to the receiver so that it can identify beginning of a video frame. For simplicity, we assume that this transmission takes the same time as a BYTE of data. Now, since transmitting 1 byte needs 640 clock cycles, we add this to the total clock cycles required for transmitting one video frame information ($151,352,320 + 640 = 151,352,960$ clock cycles) to obtain the clock period for `clkgen` instance `cg1`.

As for clock `cg2`, `send_data` sends out one byte at a time, making the required clock period to be 640 clock cycles.

Note that `MakeFrame` does not have any timing restrictions yet because we have, for simplicity, modeled it to be synchronous with the arrival of the `SerialData` signal from

`Data_generator` (Figure 14). For details, see `Uar_Design_Example\uar_step3` in the tutorial directory.

Version 4

Data Transfer in BITS (version 4)

The version 3 used a data-type abstraction of transmitting data in units of a BYTE. For real-life devices, BITS of data is transmitted over an established serial link. We will perform another *data type refinement* by sending and receiving frame data in units of a BIT. We have defined BIT as a user-defined data type in `user.h`.

We choose to use a scheme very similar to Figure 12 to model transmission of BITS over the serial link. The scheme is shown in Figure 15.

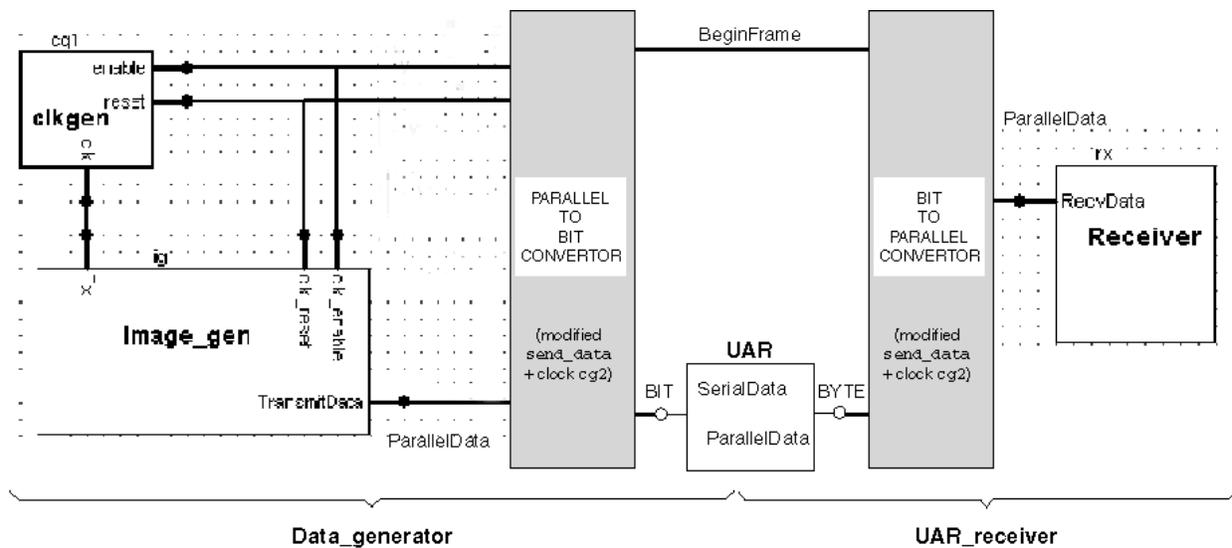


Figure 15: Parallel ↔ BIT Converters In Version 4 Of The UAR Design

In the `Data_generator` shown above, we modify the `Send_data` module connected to clock `cg2` to perform a parallel data-to-BIT conversion.

Note: In version 3 of the design, `Send_data` module with clock `cg2` did parallel data-to-BYTE conversion (Figure 13(a)).

In the `Uar_receiver`, we add a UAR that is essentially a BIT-to-BYTE converter as a front-end to it. The BYTE data output by UAR is then sent to the previously designed `MakeFrame` (BYTE-to-Parallel data converter) for further consumption.

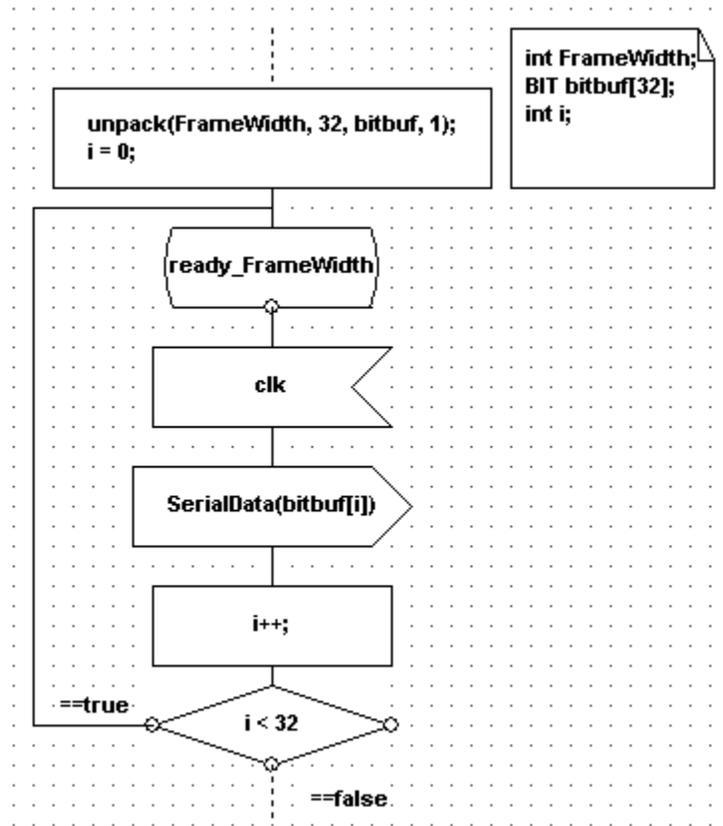


Figure 16: Send_data Unpacks FrameWidth And Transmits BITS

Parallel Data-to-BIT converter

We start by modifying the `send_data` module (Figure 13) to convert parallel data into BITS and transmit it. Here, we can use the same `pack` template function to do the conversion as shown Figure 16. Note that the only change from version 3 is that `FrameWidth` is now unpacked as 32 separate BITS and stored in the `bitbuf` array.

The clock period of `cg2` connected to `send_data` should be 80 clock cycles because each bit is assumed to take 80 clock cycles for transmission. Also, the clock period of `cg1` can now be reduced to $(151,352,320 + 80 = 151,352,400$ clock cycles) because sending `BeginFrame` signal takes the same time as transmission of 1 bit of data (80 clock cycles).

BIT-to-BYTE Converter (UAR)

There is one major change in the `uar_receiver`. We added a high-level model of a UAR as front-end to the `uar_receiver` as shown in Figure 15. The UAR model converts 8-bit information into a BYTE for consumption by the `MakeFrame` module. The rest of the design remains unchanged.

The used model for the UAR is very simple as depicted in Figure 17. It simply waits for a bit (as a payload of `SerialData`) to arrive. When the eighth bit arrives, it is packed into a BYTE using the same `pack` function and sent to `MakeFrame` (as a payload of `ParallelData` signal) for further processing as shown in Figure 17. Note that this model is *untimed*; it

has functional timing. For details about this version, see `Uar_Design_Example\uar_step4` in the tutorial design directory.

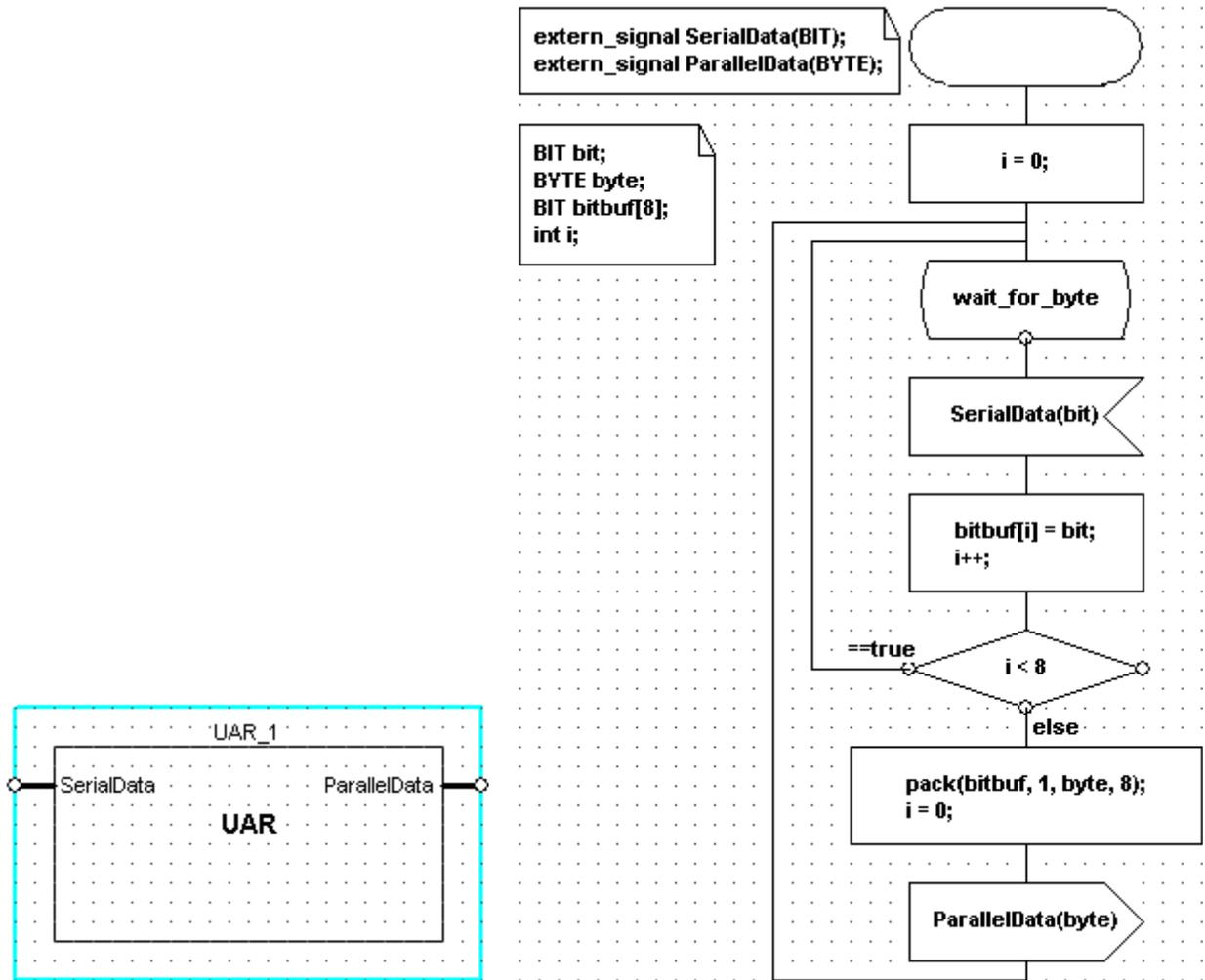


Figure 17 UAR symbol (a) and Simplified model of a UAR (b)

Final version

Using UAR for Serial Data Reception (Final Version)

In this section, we will discuss the final version of the UAR design. First, the `Data_generator` will be modified to become a high-level Universal Asynchronous transmitter. It will send start-bits and stop-bits between characters (8-bit data) as required by the UAR specification.

A circuit of components described at the register-transfer level (RTL) will now replace the simplified UAR in the `uar_receiver` ([version 4](#)). A separate clock say `RxClock` will have to drive the UAR because clocks are generated locally at each end of the transmission link and therefore cannot be assumed to be *locked*. We will show that the receiver works

correctly only when its frequency/clock period nominally matches that of the transmitter clock frequency/period.

Also, the `MakeFrame` module will be modified to receive information from the UAR about whether a transmission error has occurred. If error occurs, it aborts processing of the current video frame and waits for the next frame.

Data Generator as a Universal Asynchronous Transmitter

As required by the UAR specification, `Data_generator` will need to transmit possibly an arbitrary number of 1s (1 is stop-bit) between transmission of any two characters. Also, before starting transmission of a character, it will transmit the start bit 0. The UAR specification is described in section [Universal Asynchronous Receiver Specification](#).

For simplicity, we transmit a fixed sequence of 110 between transmission of two characters although it could have been any random sequence like 10, 111110, etc. Also, upon transmission of the last character/BYTE of a video frame information, we transmit another fixed sequence of 111.

Such simplifications are acceptable because our intent is to test the final RTL level UAR using the `Data_generator`. We do not intend to model any specific Universal Asynchronous transmitter in this design.

RTL description of UAR

We start by modifying the symbol for UAR from version 4 (Figure 17(a)) into the one as shown in Figure 18.

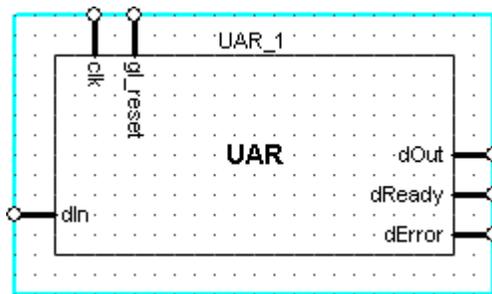


Figure 18: Symbol Of The RTL-Level UAR

The `dIn` pin is used to input the serial BIT data sent from a UART transmitter such as our `Data_generator`. The receiver clock, say `RxClock`, connected to the `clk` pin of the UAR synchronizes its operations. We will show while running a prototyping session later that this receiver clock frequency has to be nominally the same as the transmitter clock frequency for the UAR (receiver) to operate correctly.

To reset the UAR asynchronously, the `gl_reset` pin should be triggered. When a valid byte/character is available in the 8-bit bus `dOut`, `dReady` pin is triggered with a BIT payload set to 1. On the other hand, any reception error will be flagged by triggering of the `dError` pin.

Based on the desired behavior of the UAR described above, we come up with the following circuit shown in Figure 19.

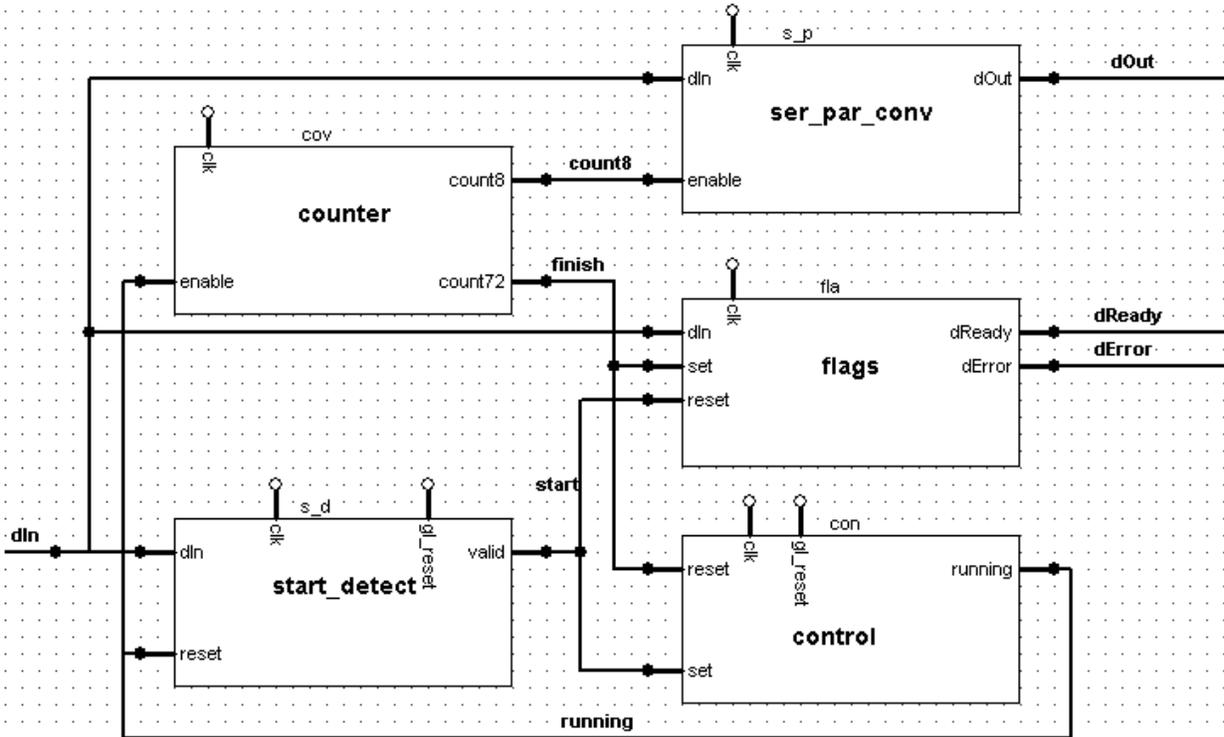


Figure 19 - RTL description of UAR.

The details of all the components of this circuit will not be described here. A brief outline of each sub-module of the UAR is as follows:

1. **start_detect**: This module deals with recognizing the beginning of a new data frame. It basically identifies arrival of a stop-bit (0). When beginning of a frame is identified, it sends a signal `valid` to **control** module; it also resets the **flags** module so that all UAR flags are reset.
2. **control**: On receipt of `valid` signal from the `start_detect`, **control** module switches to run mode. This module decides whether the UAR is idle or busy processing.
3. **counter**: This counter generates a pulse every 8th clock cycle to input the data (pin `count8` is triggered). Thus, the data is guaranteed to be sampled in the middle of the valid period as desired in the specification. Each bit of sampled data is stored in serial-parallel shift register. On the last bit of the data frame (indicated by triggering in the `count72` pin), the shift register is not updated, i.e. `count8` is not triggered. Instead `count72` triggering is used to determine if valid data is available in the shift register.
4. **ser_par_conv**: This module is essentially a serial-parallel shift register. With each triggering of `count8` signal by the counter indicating that data can be sampled at the middle of a valid period, the bit is shifted into the register. After 8 such shifts,

we have a byte of data. Whether this data is valid or not is decided by the **flags** module below.

5. **flags**: When the last bit of the data frame is received, `count72` port is triggered indicate this. Then this module attempts to identify if a valid stop-bit (1) has been detected. If it does not detect a valid stop-bit, it flags an error by sending the `dError` signal with a BIT payload to 1. Otherwise, `dReady` is triggered.

A `gl_reset` signal is included so that the components can be brought into a known state at the start of operations.

Clocking requirements of UAR

As decided earlier at the UAR specification, we will need to make the `Receiver` clock connected to UAR to be eight times the bit rate. Since the bit rate is 1 bit/ 80 clock cycles, so the `Receiver` clock period needs to be set to 10 clock cycles.

We could re-use an instance of the `clkgen` module (Figure 9) to drive the UAR. But since we want the user of the UAR design to control the `Receiver` clock frequency/period, we create another module called `varclkgen`. It is very similar to the `clkgen` module except that its clock `period` is defined as a **VS_INT** and can be controlled from the test bench. The default value of the clock period is set to 10 clock cycles.

MakeFrame to Handle Transmission Error

We modify `MakeFrame` module to handle any frame error indicated by the UAR. It is also nice to be able to reset the module.

For this reason, three new pins are added: `Enable`, `GetData`, and `ErrorData`. When `ErrorData` is triggered by the UAR, `MakeFrame` aborts processing of the current video frame and waits until the beginning of the next frame. `GetData` indicates that a valid byte is available in `SerialData` bus.

Controlling the Receiver

To enable the user to control the operations of the `Receiver`, we modify the `Receiver.tb` Test Bench to add the following Test Bench controls:

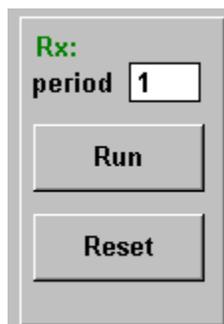


Figure 20: New Test Bench Controls For Receiver

Clicking on the **Run** button shown above will start the UAR Receiver. Clicking **Reset** button will cause the Receiver to reset asynchronously. Also, the value of the *register Test Bench control* is the period of the Rx clock; this period can be increased or decreased by using the arrow keys during a prototyping session.

To enable such control over the *Receiver*, we also create another control module called *recv_ctrl* that is connected to an input signal called *RecvData*. Clicking on the buttons above changes the value of *RecvData* signal. Correspondingly, the *recv_ctrl* module generates correct signal values to do the desired operations on all the components.

Based on the description on above sub-sections, our final *uar_receiver* appears as shown below in Figure 21.

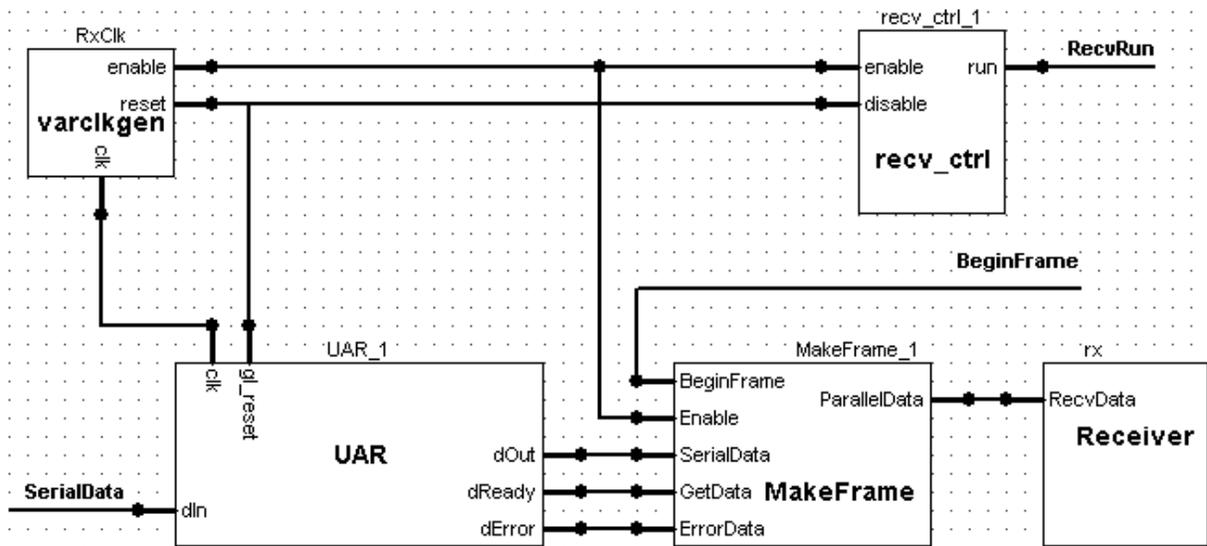


Figure 21: Final Description Of The *uar_receiver*



Figure 22: uar_receiver Received The Second Video Frame

Now, when the prototyping session starts, the new Test Bench controls come up for the *Receiver*. The user can independently control either the *Data_generator* or the *uar_receiver* using the appropriate Test Bench controls.

Also, we can vary the clock period/frequency of the *Receiver* clock *RxClock* using the arrow keys on the register Test Bench control of *Receiver.tb* Test Bench. It can be seen that if the *RxClock* period/frequency is varied such that it does not match the transmission rate, reception error will be issued by the UAR.

The Figure 22 shows both the `Data_generator` and the `uar_receiver` running; the second frame has been received successfully and the transmission of the third frame is in progress.

Summary

As illustrated in the `UAR Design`, `MAGIC-C` was used to specify a high-level system design. The design was then refined in stages using various constructs of `MAGIC-C`, with each stage showing a refinement in (a) data types (b) timing (c) communication protocol and/or (d) modeling abstraction. `MAGIC-C` was employed to describe behavioral level abstraction of components as well as doing RTL design.

The `MAGIC-C Symbol` was used extensively to show that it enables creation of self-contained, re-usable components. Also, the Virtio Innovator's Test Bench Builder tool was used to create `GUI`-based custom control of the design. In particular, error conditions that are not found easily by testing could be readily located in a prototype by dynamically changing the receiver clock frequency by using Test Bench controls during a running prototyping session.

Sample user-defined data types and functions were defined to demonstrate creation of infrastructure for a simpler and more concise design.

Glossary

D

DRC: Design Rule Checker

DUT: Design Under Test

F

FSM: Finite State Machine

G

GUI: Graphical User Interface

I

IDE: Integrated Design Environment

L

LCD: Liquid Crystal Display

LED: Light Emitting Diode

M

MAGIC-C: (ANSI) C-based specification language, enriched with graphical extensions, used as input in the Virtual Silicon integrated development environment.

R

RTL: Register Transfer Level

U

UAR: Universal Asynchronous Receiver

UI: User Interface

V

VCD: Value Change Dump

VS: Virtual Silicon

Index

-	
__int64	59
A	
Adding delay and time in the model.....	74
H	
Handling protocol refinement in MAGIC-C78	
History	3
M	
Matched Filter Design	67
Adding delay and time in the model ...	74
Data_generator Specification	70
Filter Specification	72
Handling protocol refinement in MAGIC-C	78
Introduction	67
Matched Filter Specification	67
Specification of Matched Filter (initial version)	70
Summary	82
System Partition	68
Using Reset in a MAGIC-C model	76
U	
UAR Design	83
Adding Delay to Image_gen model (version 2)	94
Basic Test Bench Framework.....	86
BIT-to-BYTE Converter (UAR)	102
BYTE-to-Parallel Data Converter	100
Change in Specification Noise Resistant Behavior	86
Communication Protocol.....	87
Controlling the Receiver.	106
Data Transfer in BITs (version 4)	101
Data Transfer in BYTES (version 3)....	97
Data_generator as a Universal Asynchronous Transmitter	104
Image Generator Specification	90
Introduction	83
MakeFrame to Handle Transmission Error.....	106
Modified Clock Period Requirements .	100
Parallel Data -to-BYTE Converter	99
Parallel Data-to-BIT converter	102
Receiver Specification	92
Refinement Steps for the UAR Framework	88
RTL description of UAR.....	104
Specification of UAR Framework (initial version)	90
Summary	109
Universal Asynchronous Receiver Specification	83
User-Defined Data Type Across Model Interfaces.....	88
Using UAR for Serial Data Reception (Final Version)	103
Using Reset in a MAGIC-C model	76