

Catapult[®] C Synthesis SystemC User's Guide

Calypto Design Systems, Inc
1731 Technology Drive, Suite 340
San Jose, CA 95110
Tel: +1-408-850-2300
Fax: +1-408-850-2301

The software described herein is copyright ©2005-2013 Calypto Design Systems, Inc. All rights reserved. The software described herein, which contains confidential information and trade secrets, is property of Calypto Design Systems, Inc.

This manual is copyright ©2005-2013 Calypto Design Systems, Inc. Printed in U.S.A. All rights reserved. This document may not, in whole or in part, be copied, photocopied, reproduced, translated, transmitted, or reduced to any electronic medium or machine-readable form without prior written consent from Calypto Design Systems, Inc.

This manual and its contents are confidential information of Calypto Design Systems, Inc., and should be treated as confidential information by the user under the terms of the nondisclosure agreement and software license agreement, as applicable, between Calypto Design Systems, Inc. and user.

Last Updated: April 2013

Product Version: Release 7.0

Table of Contents

Chapter 1	
Introduction.....	5
Getting Started.....	5
Start with Templates.....	5
Compile/Simulate Design.....	5
Chapter 1	
Supported SystemC	1
Structural Elements	1
SC_MODULE.....	1
SC_METHOD.....	2
SC_CTHREAD.....	2
SC_THREAD.....	3
Wait Statements	3
SC_SIGNAL and SC_FIFO	4
Data Types.....	4
General Clock/Reset Rules for Synthesis	5
Rules for SC_METHOD.....	6
Rules for SC_CTHREAD.....	6
Rules for SC_THREAD	7
Chapter 2	
Modular I/O.....	9
The Modular I/O Class.....	10
Guidelines For Writing Modular I/O	10
Channel Class	11
Modular I/O Class.....	11
Derived Modular I/O Class.....	11
Port Definition	11
Clock and Reset Ports.....	12
Signal Definition.....	12
Construction and Object Naming	13
Reset	13
Class Methods.....	13
Inlined Functions	13
Design Functions	14
Component Functions.....	15
Binding Functions.....	16
Modular I/O Definitions	16
Blocking Read.....	16
Non-Blocking I/O.....	17
Blocking Write	18

Inferring Modular I/Os	18
Catapult Interconnect Libraries for SystemC	18
Transaction and Synthesis Views	19
Ports and Channels	20
How to use Complex Types	21
Point-to-point Example Designs	22
Point-to-Point I/O Library Reference	30
Upgrading from modular_io.h to ccs_p2p.h	36
Sample Memory Reference	37
Catapult Modular I/O Library	42
en_in	44
en_out	45
w_in	46
w_out	47
wait_event_in	48
wait_event_out	49
wait_in	50
wait_out	51
RAM and Modular I/O	52
Splitting API Calls and Pipelined RAMs/Busses	52
AHB Bus Master Example	53
Chapter 3	
Basic Design Examples	55
Combine the Supported Styles in Designs	55
Example 1: Reference C Design	56
Example 2: Using ac::wait Timing Constraint	56
Example 3: Using SC_THREAD/SC_FIFO	57
Example 4: Using SC_THREAD with Wait	58
Example 5: Using SC_CTHREAD and Modular I/O	59
Example 6: Mixed Design	60
Chapter 4	
RTL Examples	61
Example 7: Combinational Logic	61
Example 8: Rules for SC_MODULE at the RTL level	62
Appendix A	
Setting Up SystemC Project	
in MSVC++ Express 2008	63

Chapter 1

Introduction

This manual provides guidance on how to write SystemC code that can be synthesized and optimized by Catapult C. This document assumes a you have a thorough understanding of the syntax and semantics of SystemC.

Getting Started

SystemC is a flexible language that allows many ways to abstract a design. However, for best results, you should use the guidelines described in this section when creating SystemC for use with Catapult.

Start with Templates

You should use the templates provided in toolkits as a guide when creating SystemC designs and test benches for use with Catapult C. Templates can be found in the SystemC toolkits available in Catapult:

1. Select **Help > Toolkits...** to display the Toolkits page.
2. Select **What's New > New Features in 2010a > SystemC**.
3. Expand the folders and browse the available toolkits.
4. Select a toolkit that models your design and either launch the project or export the files to your workstation.

Compile/Simulate Design

Always make sure your design compiles and simulates successfully before setting it up for synthesis in Catapult. Starting with a design that successfully simulates eliminates any problems caused by SystemC source code issues during the design synthesis process.

You can use the templates provided in the SystemC toolkits to create a test bench for your design. SystemC test benches must provide a source and sink block for the design and be self-checking.

Chapter 1

Supported SystemC

SystemC supports many styles including abstract styles that are not suitable for use with RTL and lower level styles that are easily linked to RTL but not very abstract. Consequently, the SystemC supported by Catapult is a compromise that allows three levels of abstraction.

The SystemC supported by Catapult is based on the OSCI SystemC v1.3 synthesis subset. However, not all of the elements are supported and Catapult supports several SystemC constructs not listed in the synthesis subset. [Table 1-1](#) describes the SystemC constructs that are synthesizable by Catapult.

Table 1-1. Supported SystemC Constructs

Construct	Element
Hierarchy	SC_MODULE
Processes	SC_METHOD, SC_CTHREAD, SC_THREAD
Timing	wait()
Datatypes	sc_int, sc_uint, sc_fixed, sc_ufixed, sc_bit, sc_bv, sc_bigint, sc_biguint
Communication	SC_SIGNAL, SC_FIFO, clock, reset, Modular I/O

See the following topics for more information on the constructs supported by Catapult:

- [“Structural Elements”](#) on page 1
- [“Data Types”](#) on page 4
- [“General Clock/Reset Rules for Synthesis”](#) on page 5

Structural Elements

Structural elements are used to model hierarchy, processes, timing, and communications. The following topics describe the structural elements supported by Catapult:

SC_MODULE

The SC_MODULE is analogous to a Verilog module or a VHDL entity/architecture pair. It is very useful for decomposing designs in sub-units in order to manage complexity.

SC_METHOD

The SC_METHOD is a process that supports the use of sensitivity lists. The sensitivity list can include clock/reset or general signals but cannot mix a clock with other signals. Signal interfaces are also supported. SC_METHODs allow modeling at the Register Transfer Level (RTL) abstraction. Combinational and sequential processes are possible.

Example:

```
SC_METHOD(my_method);
sensitive << clock.pos();

#pragma design
void my_method () {
    if ( A_rdy.read() )
        dout->write(intermediate_A.read());
    else if ( B_rdy.read() )
        dout->write(intermediate_B.read());
}
```

SC_CTHREAD

The SC_CTHREAD is a process sensitive to the clock and reset signals and supports communication using sc_signal, sc_fifo, and modular I/O. It always schedules in superstate (Catapult may add time). Although the SC_CTHREAD is officially deprecated in SystemC IEEE1666, Catapult fully supports it for backward compatibility of legacy designs.

Example:

```
SC_CTHREAD(arbiter, clk.pos());
reset_signal_is(reset,true);

#pragma design
void arbiter () {
    while(1) {
        wait();
        bool A_rdy = A.num_available() > 0;
        bool B_rdy = B.num_available() > 0;
        if ( A_rdy )
            dout->write(A.read());
        else if ( B_rdy )
            dout->write(B.read());
    }
}
```


SC_THREAD

The SC_THREAD supports the use of sensitivity lists and adds a reset. SC_THREAD supports blocking and non-blocking FIFO/signal communication and Modular IO (clock required). It always schedules in superstate (Catapult may add time). The SC_THREAD process is the preferred construct, especially in the TLM context.

Example:

```
SC_THREAD(arbiter);
sensitive << trigger.pos(); //not require to be a clock

#pragma design
void arbiter () {
    while(1) {
        wait();
        bool A_rdy = A.num_available() > 0;
        bool B_rdy = B.num_available() > 0;
        if ( A_rdy )
            dout->write(A.read());
        else if ( B_rdy )
            dout->write(B.read());
    }
}
```

Note



Process control extensions (not yet adopted by OSCI) required for general reset support.

Wait Statements

Wait statements define the minimum latency and default throughput. Constraints, such as pipelining, are allowed to change throughput. Latency may be increased in order to resolve I/O and RAM conflicts and to save area.

Example:

```
void fir() {
    for (int i=0;i<N;i++)
        asr[i] = 0;
    out.write(0);
    wait();
    while(1) {
        SHIFT_LOOP:for (int i = N-1; i>0; i--)
            asr[i] = asr[i-1];
        asr[0] = in.read();
        wait(4);
        acc = 0;
        MAC_LOOP:for (int i=0;i < N;i++)
            acc += asr[i] * coeffs_[i];
        out.write(acc);
    }
}
```

```
}
```

SC_SIGNAL and SC_FIFO

For communications constructs, Catapult supports both `sc_signal` and `sc_fifo` primitive channels. While `sc_signals` provide a familiar coding style to RTL designers, `sc_fifos` allow very simple and concise modeling at the functional level. Using `SC_FIFOs`, one can efficiently model deterministic networks of communicating processes, such as Kahn Process Network (KPN).

Support for `SC_SIGNAL` and `SC_FIFO`:

- Supported `SC_SIGNAL`, `SC_IN` and `SC_OUT` methods include `read()`, `write()` and assignment. Explicit read/write methods (`x=in.read()` and `in.write(x)`) are recommended over simple assignment (`in=x` and `x=in`) because the simple form is not guaranteed to work correctly with all compilers.
- Supported `SC_FIFO*` methods include non-blocking read methods, blocking read/write methods and the `num_available()` method. Non-blocking write methods are not supported.
- All built-in data types are supported.
- The `ac_datatypes` and `sc_datatypes` are supported.
- The `ac_channel` datatypes are not supported between threads.
- Classes and structs are supported. When using `SC_FIFOs` that contain structs, set the directive `PRESERVE_STRUCTS` to true. If `PRESERVE_STRUCTS` is false, the synthesized `SC_FIFOs` will not be supported in the generated SystemC wrappers.
- Arrays of all `SC_INs`, `SC_OUTs`, `SC_SIGNALs` and `SC_FIFOs` are supported.
- `SC_SIGNAL` can be used to connect modules.
- `SC_FIFO` is not supported by Catapult to connect two modules. This limitation will be removed in a future release.
- `SC_SIGNAL` supports multiple readers and multiple bindings, but not multiple writers nor multiple writer bindings.
- `SC_FIFO` does not supported multiple readers/writers or multiple bindings.

Data Types

SystemC datatypes support is limited as described in the “[SystemC Data Types](#)” section of the *Catapult C Synthesis C++ to Hardware Concepts* manual. Because of flaws in these SystemC datatypes, you should use the algorithmic C datatypes described in the [AlgorithmicC Datatypes](#) manual.

General Clock/Reset Rules for Synthesis

Every design must have a directive that defines all of the clocks. This directive defines many properties of both the clock and reset. In SystemC designs, the directive settings are default values that can be overridden. For example, the `CLOCKS` directive defines the active edge, but that can be overridden by settings on each process or register to allow designs where registers are sensitive to different clock edges.

The following rules apply to clocks and resets:

- All scheduled blocks must have one clock and one or more resets.
- You must include every clock and reset in your design in the `CLOCKS` directive before issuing the "go compile" command.
- The `CLOCKS` directive is organized into groups including one clock, one asynchronous reset and one synchronous reset.
- The directive is organized by clock, but multiple clocks can be associated with the same reset.
- Derived clocks and resets are currently not supported. You must drive all clocks and resets from top-level primary inputs.
- All primary inputs (input ports in the top-level design) that are read by a process are checked to see if they match a port defined in the `CLOCKS` directive. These ports must be used as clocks and resets inside of the process.
- A process may only use signals from one group within the clock.
- Unbounded loops must contain at least one explicit `wait()` in each control path.

Any behavior encountered prior to the first wait statement is considered reset behavior. Any behavior after the first wait is considered operational behavior. Since the behavior inside the unbounded loop is considered operational behavior, the first wait that separates reset behavior from operational behavior must be located before the unbounded loop or as the first statement of the unbounded loop as shown.

The following example shows how the reset in an unbounded loop should be coded:

```
void mth( )
{
  // reset
  some_global_thing = 0 ;
  out = 0 ;
  wait();

  do
  {
    //reset + behavioural
    out = 1 ;
    wait(); // INVALID ñ Illegal according to spec
  }while( 1 ) ;
}
```

- Reset signals must be used consistently between two blocks. The two blocks must use the same reset phase and frequency (synchronous or asynchronous).
- Different blocks may be sensitive to different clock edges.

Rules for SC_METHOD

- All SC_METHODS are considered to be RTL, so they must be scheduled with fixed timing. Catapult does not run its scheduler to change the timing in the design.
- An SC_METHOD that is sensitive to non clock/reset signals is synthesized into a combinational block.
- An SC_METHOD that is sensitive to a clock is synthesized into a synchronous block. This block can also be sensitive to the asynchronous reset associated with the clock in the CLOCKS directive but not to any other signal.
- The SC_METHOD must define the polarity for both the clock and reset. This means that only the signal names are used from the Catapult CLOCKS directive.
- A reset is not required for an SC_METHOD.
- No logic can be on clock/reset lines (must only be used as clock or reset).
- No Clock Domain Crossing checking is performed.

Rules for SC_CTHREAD

- All synchronous SC_CTHREAD blocks must have at least one clock and one reset.
- An SC_CTHREAD must define the polarity for both clock and reset in the source.
- If the reset is defined in the source, then the active phase is extracted from the source.
- An SC_CTHREAD must define the clock polarity with a "reset_signal_is" associated with it.
- Catapult adds a reset to the process based on the CLOCKS directive if none is specified.

- SystemC only models synchronous resets, but Catapult allows you to define both synchronous and asynchronous clocks with the CLOCKS directive.
- If modular I/O or `sc_signal` in `SC_CTHREAD`
 - Modular I/O derives clock/reset from calling thread.
 - No Clock Domain Crossing checking.
- If `SC_FIFO` in `SC_CTHREAD`
 - Any scheduled process may have one clock and one or more resets in source code.
 - FIFO derives clock/reset from calling threads.
 - CDC conditions detected and handled automatically.

Rules for `SC_THREAD`

- Synchronous `SC_THREAD` blocks must have at least one clock and one reset defined when using modular I/O. If `SC_FIFO` is used, no clock or reset is required in the source code.
- `SC_THREAD` blocks that are not sensitive to clocks, but are sensitive to all their inputs and do not have blocking I/O are considered to be combinational.
- If the clock is defined in the source, then the active edge of the clock is extracted from the source, otherwise it is inherited from the CLOCKS directive.
 - If an `SC_THREAD` is sensitive to a clock, then the active edge is defined by the source code.
 - If an `SC_THREAD` is not sensitive to a clock, the default clock is connected, which can be changed with architectural constraints.
- Explicit resets on an `SC_THREAD` are not supported, Catapult adds a reset based on the CLOCKS directive.
 - Catapult adds a reset to the process based on the CLOCKS directive if none is specified.
- If modular I/O or `sc_signal` in `SC_THREAD`
 - Any scheduled process must have exactly one clock and at least one reset in source code.
 - Modular I/O derives clock/reset from calling thread.
 - No Clock Domain Crossing checking.
- If `SC_FIFO` in `SC_THREAD`
 - Hardware synthesized from an `SC_FIFO` derives a clock/reset from calling threads.

- CDC conditions detected and handled automatically.

Chapter 2

Modular I/O

Modular I/O is a powerful coding style that provides a clean separation between functionality and communication. It is the preferred method for describing explicit communication in SystemC source. Modular I/O provides a way for modules to communicate using transaction level interface functions. The interface function itself can be implemented either in a cycle-accurate manner or in a purely TLM style, using the TLM2.0 API. Catapult C supports both blocking and non-blocking transactions.

You can write custom modular I/Os or use the modular I/O header files provided in the `$MGC_HOME/shared/include` directory described in [Table 2-1](#).

Note



The modular I/O in the *modular_io.h* are templated to use any data type and should work for most designs. Writing custom modular I/O should only be done if you have an interface that does not match any of those provided in *modular_io.h*.

Table 2-1. Modular I/O Header Files

Table 2-2.

File	Description
<code>modular_io.h</code>	Basic point-to-point library that provides basic I/Os needed by most designs.
<code>ram_*.h</code>	RAM templates for singleport, dualport, and separate RW.

If modular I/Os are written similar to those provided in *modular_io.h*, they are automatically recognized and optimized by Catapult resulting in fewer registers and less latency. For more information, see [“Catapult Modular I/O Library”](#) on page 42.

See the following topics for more information on modular I/Os and how to use them:

- [“The Modular I/O Class”](#) on page 10
- [“Modular I/O Definitions”](#) on page 16
- [“Inferring Modular I/Os”](#) on page 18
- [“Catapult Modular I/O Library”](#) on page 42
- [“RAM and Modular I/O”](#) on page 52

The Modular I/O Class

The modular I/O implementation is based on a class containing port definitions and function calls to describe *transactions* related to those ports. The following topics describe guidelines and the elements of the modular I/O class contained in the *modular_io.h* header file.

Guidelines For Writing Modular I/O	10
Modular I/O Class.....	11
Derived Modular I/O Class.....	11
Port Definition	11
Clock and Reset Ports.....	12
Signal Definition.....	12
Construction and Object Naming	13
Reset	13
Class Methods.....	13
Inlined Functions	13
Design Functions	14
Component Functions.....	15
Binding Functions.....	16

Guidelines For Writing Modular I/O

You should use the following guidelines when writing modular I/Os for use with Catapult:

- The *modulario* pragma can be used with any function.
- Recommended flow is to make the function a class member.
- Code must be cycle accurate and have at least one wait for synchronous I/O.
- Multiple data inputs or outputs should be combined into a single interface variable using a struct.

Example

```
void reset() { request.write(false); }
#pragma design modulario
T get() {
    while (true) {
        request.write(true);
        wait();
        if ( valid.read() ) break;
    }
    request.write(false);
    return data.read();
}
```


Channel Class

Every modular I/O should have a channel class that allows it to be connected without connecting the individual signals. This simplifies the user code and simplifies switching between different modular I/Os. Note that this is templated to allow any datatype for the "data" signal.

```
// A "channel" class which encapsulates the two wire handshake signal.
template <class T>
class wait_hs
{
public:
    sc_signal <bool> valid;
    sc_signal <T> data;
    sc_signal <bool> request;
    ...
};
```

Modular I/O Class

The *class T* template argument allows this class to be adapted to any datatype as shown in the following example:

```
//! \class wait_in \\brief A modularI/O component providing two signal
handshake.
/*! \class wait_in \detailed
 * The wait_in module is an example of a modular_io component.
 */
template <class T>
class wait_in {
```

Derived Modular I/O Class

The modular I/O class may optionally be derived from another class. If the modular I/O class is derived from an SC_MODULE, then it may contain threads and methods. Classes that are not derived from SC_MODULE will not have the methods available to build the SystemC structure for threads and methods.

The following example shows a modular I/O class derived from an SC_MODULE:

```
template <class T>
class wait_in : public sc_module {
    ...
    SC_HAS_PROCESS(wait_in);
```

Port Definition

The class includes a list of ports that are declared public in the class. They are declared public so synthesis flows can access these ports to help with debugging and wrapper generation.

Current supported types are "sc_in" and "sc_out". The sc_fifo type is supported for SystemC between threads, but it is not supported for modular I/O.

Ports are not required for modular I/O that connects between two threads in the same level of hierarchy.

```
public:
    sc_in<bool> valid;
    sc_in<T> data;
    sc_out<bool> request;
```

Clock and Reset Ports

If the modular I/O is derived from an SC_MODULE, you will need to define clock and reset ports. These ports are normal sc_in ports on the SC_MODULE. They will then be connected normally to internal clocked methods and threads.

```
sc_in<bool> clock;
sc_out<bool> reset;
```

Note



The module that declares the modular I/O needs to connect the clock and reset ports in the constructor.

The following example shows a construct for a module that includes a modular I/O with clock and reset:

```
SC_CTOR(my_module) :
    clk("clk") ,
    rst ("rst"),
    input1("input1"),
    output("output")
{
    input1.clock(clk);
    input1.rst(rst);

    SC_CTHREAD(arbiter, clk.pos());
    reset_signal_is(rst, 1);
}
```

Signal Definition

The modular I/O may also contain signal declarations. These signals can be used to communicate between processes in the Modular I/O and may also be connected to objects outside of the modular I/O. The recommended approach for signals is to make them private data members and to use accessor functions if they need to be read and written outside of the modular I/O. Accessor functions can be general functions and do not need to be mapped to modular I/O methods.

Construction and Object Naming

The only constructor allowed for the modular I/O class must take a “const sc_module_name” as an argument. This name must then be concatenated with a string for each of the ports in the modular I/O. If you don't provide a unique name, then Catapult will resolve the name and it may be difficult to predict the final port name.

The constructor is only allowed to construct "structural" elements of the design, like modules, methods and threads. A separate function is provided for reset.

```
wait_in (const sc_module_name &name) :  
    valid(concat(name, "valid")),  
    data(concat(name, "data")),  
    request(concat(name, "request")) {}
```

If the module is derived from an SC_MODULE, be sure to construct the SC_MODULE class with the name argument.

```
wait_in (const sc_module_name &name) : sc_module(name),  
    valid(concat(name, "valid")),  
    data(concat(name, "data")),  
    request(concat(name, "request")) {}
```

Reset

A modular I/O must define its reset behavior, both for the port signals and for any static data. The reset function must be called in the reset area of the design that uses this I/O.

```
void reset() { request.write(false); }
```

If different modular I/O methods are called by different threads, then multiple reset functions will be required (one for each thread).

Class Methods

When an instance of a modular I/O is declared, the transaction functions define how to use the I/O. You can access ports directly, but this is not recommended.

Inlined Functions


Functions declared without pragmas are *inlined* into the caller. This means that these functions run directly in the core process in the generated RTL.

No special rules are applied to the timing of these functions, they are scheduled as part of the calling function. The most common application is configuration register or status data.

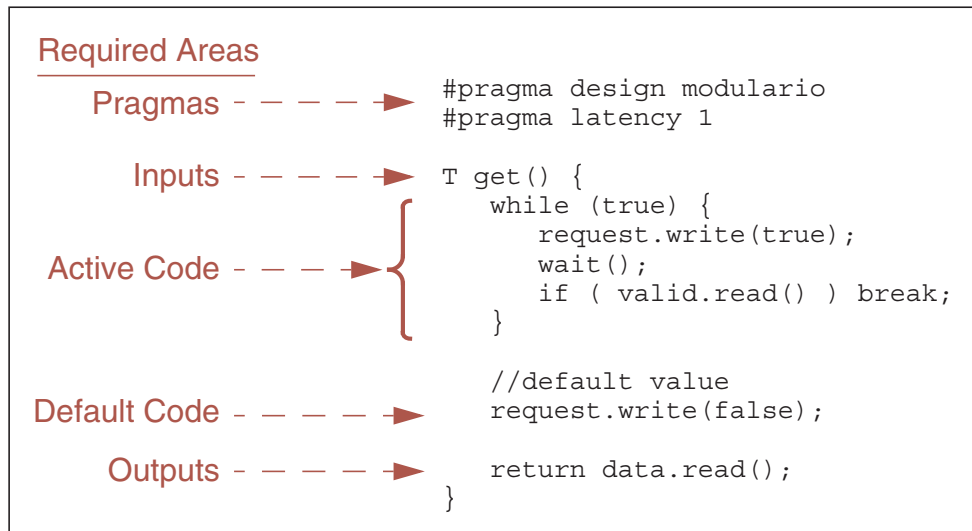
```
bool available() const { return valid.read(); }
```

Design Functions

If the function describes specific I/O timing and it has a variable latency, then it should be mapped to a "design modulario." This will cause Catapult to restrict the timing on the I/O. Catapult will also move this code into a separate process that is tightly controlled by the core design using a "start/done" handshake.

Note  The tool currently allows pipeline constraints to be set on modular I/Os, but you should not set pipelining constraints. The throughput of the modular I/O method must be defined by the number of wait statements in the design.

Below is the example of the "get" function, followed by a detailed description of the required areas in this code:



Pragmas:

```
#pragma design modulario  
#pragma latency 1
```

The design function requires two pragmas. The *modulario* pragma identifies the code as a modular I/O. This pragma causes the modular I/O to be synthesized as a concurrent process.

Catapult is allowed to add latency to modular I/O unless the fixed I/O mode is used. The latency pragma is used to define the synthesized latency of the I/O. If the latency pragma does not match the true minimum latency of the design, then you may have a deadlock in RTL simulation. The latency of the design is defined by the wait statements in the active code.

Inputs:

```
T get() {
```

If any arguments passed to the functions are inputs from the core, they are synchronized with the start signal that is added to the RTL. The get function doesn't have any arguments, so there are no inputs from the core. The put function in the wait_out modular I/O in `$MGC_HOME/shared/include/modular_io.h` has an example of this.

Active Code:

```
while (true) {  
    request.write(true);  
    wait();  
    if ( valid.read() ) break;  
}
```

This region defines what the modular I/O does when it's active. The throughput of the I/O is defined by the waits, but Catapult is allowed to increase the latency (as long as the latency pragma is consistent).

If an output has a "default value" (as defined in the next section), then Catapult will drive that port to the default value on any edge that it is not driven in the source. For example:

```
request.write(true);  
wait();  
// If request has default value of false, it will be driven false here  
wait();  
request.write(false);
```

Default Code:

```
//default value  
request.write(false);
```

The generated RTL will include code to stop the modular I/O process when it's not active. The code after the last wait is used to define the value for any control signal when that happens. If an output has no default value, then the RTL will continue to drive the last value.

Outputs:

```
    return data.read();  
}
```

Finally, the output of the modular I/O are returned to the core. This output becomes a signal synchronized with the "done" signal from the modular I/O. The return can be in the middle of the active code, but it is important to be sure the default code is called before you return. Otherwise, their control signals will be held stable and will not return to their default value.

Component Functions

If the function has a fixed latency, then it can be mapped into the "CCORE" flow in Catapult. This will build a component that is driven by the core and required to respond to the core within a fixed amount of time.

The most common application for this flow is RAM interfaces.

 **Note** This flow is still under development. Design functions can be used instead, but they will have a slightly larger area because of the handshake logic.

Binding Functions

Every modular I/O is required to provide the following binding functions to automate connecting the modular I/O using a channel.

```
// some utility binding functions:
template <class C>
void bind (C& c) {
    valid(c.valid);
    data(c.data);
    request(c.request);
}

template <class C>
void operator() (C& c) {
    bind(c);
}
};
```

Modular I/O Definitions

The following topics describe the different types of modular I/Os:

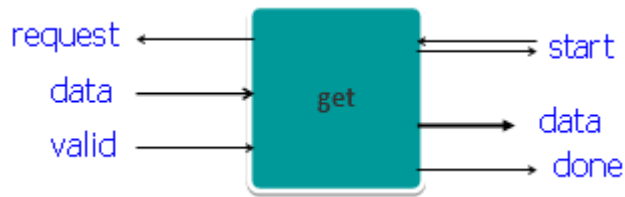
[“Blocking Read”](#) on page 16

[“Non-Blocking I/O”](#) on page 17

[“Blocking Write”](#) on page 18

Blocking Read

A blocking read modular I/O stalls the design until it returns control to the calling function. The following example shows a blocking read where the while loop runs until the valid signal is true.

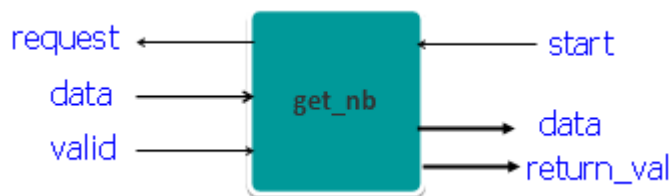
Example Blocking Read:

```
void reset() { request.write(false); }  
#pragma design modulario  
T get() {  
    while (true) {  
        request.write(true);  
        wait();  
        if ( valid.read() ) break;  
    }  
    request.write(false);  
    return data.read();  
}
```

Non-Blocking I/O

The code for modular I/O functions that do not need a full handshake must have exactly one wait. The resulting RTL I/O does not stall the core:

- Start signal from the core starts the protocol.
- Return value is read exactly one cycle after the process starts.

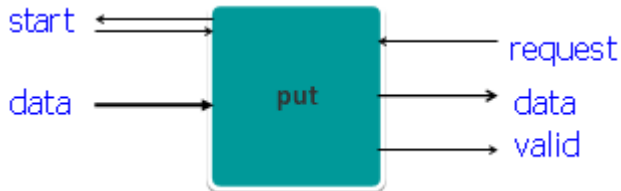
Example Non-Blocking I/O:

```
void reset() { request.write(false); }  
#pragma design modulario  
bool get_nb(T &result) {  
    request.write(true);  
    wait();  
    request.write(false);  
    result = data.read();  
    return valid.read();  
}
```

Blocking Write

Blocking writes are modular I/O functions that do not return a value. The following example function is blocking because of the while loop. The core stalls if it attempts to write, and the target can not accept data (back pressure).

Example Blocking Write:



```
void reset() { valid.write(false); }
#pragma design modulario
void put(T d) {
    while (true) {
        o_dat.write(d);
        o_vld.write(true);
        wait();
        if (i_rdy.read() ) break;
    }
    o_vld.write(false);
}
```

Inferring Modular I/Os

Modular I/Os written similar to those provided in the *\$MGC_HOME/shared/include/modular_io.h* are automatically recognized and optimized by Catapult, resulting in fewer registers and less latency.

You should use the modular I/Os from the *modular_io.h* as a template and modify the port and variable names. However the number and direction of ports and their usage must exactly match what is in *modular_io.h* to get the optimized modular I/O. If the coding style does not match exactly, Catapult synthesizes the modular I/O as a separate process with increased registers and latency.

Catapult Interconnect Libraries for SystemC

Catapult includes two libraries of interfaces to simplify transferring data between threads in SystemC. One is a point-to-point library and the other provides sample memories. These libraries are written in SystemC and can be extended or replaced with custom libraries. However, most designs should be built using the standard libraries.

The “p2p” library includes point-to-point connections that transmit data with synchronization or allow two blocks to be synchronized. The p2p interfaces are found in


```
<catapult_install_dir>Mgc_home/shared/include/ccs_p2p.h.
```

The “sample memory” library provides three example shared memories. The sample memories are placeholders to use when a generated RAM is not available. Once a RAM generator has been run to build a specific cut of a RAM, the sample memory will be replaced. The shared memory interfaces are found in

```
<catapult_install_dir>Mgc_home/shared/include/ccs_sample_mem.h.
```

Transaction and Synthesis Views

All interface libraries provide both transaction and pin-level synthesis views. The transaction view only defines the transmission of data and does not include signal-level detail. A design with only transaction level interfaces will simulate on the order of about 10-100x faster than a design with all synthesis level interfaces. Interconnects should be simulated using the transaction view whenever possible.

The synthesis view defines the exact pin-accurate protocol used to transmit one value across the interface. This detail is required for synthesis, so Catapult will always use the synthesis view. The synthesis view is also used when connecting to pin-accurate RTL. For example, the SCVerify flow will use the synthesis view to connect the original design to the generated VHDL or Verilog.

The interface libraries use nested classes to manage switching between TLM and SYN views. All library elements have the structure:

```
base_class<abstraction_t>::class_member
```

where “abstraction_t” can have the following enumerated types types:

- AUTO (default) – This setting gives the following behavior:
 - TLM view is used for source simulation
 - SYN view is used for synthesis
 - SYN view is used during SCVerify
- TLM – This setting gives the following behavior:
 - TLM view is used for source simulation
 - SYN view is used for synthesis
 - TLM view is used during SCVerify
- SYN – This setting gives the following behavior:
 - SYN view is used for source simulation
 - SYN view is used for synthesis

- SYN view is used during SCVerify

Most channels should have the “AUTO” setting (if you’re not sure, just leave this blank). If the pin-accurate view is needed during source simulation, change this to the “SYN” setting. If a channel is only used in the testbench, then it may have the “TLM” setting to help simulation speed.

Ports and Channels

SystemC separates objects into “channel” and “port” types, with restrictions on which port can be connected to a channel. Each channel may be mapped to one output and one input port.

All interconnect libraries are also structured into ports and channels and have this structure:

`base_class<>::chan` – This is the “channel” type, which is used to connect two threads or modules together. The channel may contain RTL code. For example, the `p2p_fifo` implements the fifo in the channel. For memories, the channel is referred to as “mem”

`base_class<>::in`, `base_class<>::out` – These are the in and out port types. The ports include the protocol and interface port definitions. They also include binding code, to automate binding the port to the channel or to other ports. For memories, these are referred to as “rd_port”, “wr_port” or “rw_port”

Ports and channels can be bound together without referring to the underlying signals to bind. When declaring an instance that has a simple p2p, for example, use this code in the constructor:

```
#pragma hls_design top
SC_MODULE (Simple_connections) {

    // Interfaces
    sc_in <bool> clk;
    sc_in <bool> rst;
    p2p<>::in<ac_int<10> > input;

    SC_CTOR(Simple_connections)
    : clk("clk")
    , rst("rst")
    , in("in")
    , out("out")
    , sub_inst("sub_inst")
    {
        sub_inst.clk(clk);
        sub_inst.rst(rst);
        sub_inst.in(input);
        sub_inst.out(channel);
    }

    private:
    Simple_connections_sub sub_inst;

    p2p<>::chan<ac_int<10> > channel;
};
```

In this example, the input port “input” is bound to the port “in” on “sub_inst”. Note that the binding does not need to refer to any of the contents of the p2p. This example also shows how to bind a channel to the “out” port on “sub_inst”.

How to use Complex Types

The underlying datatypes for both the p2p and memory interconnects are based on `sc_lv`. Using a vector as the base type provides better compatibility with simulators/debuggers and simplifies mixed-language design.

To use a complex type, write custom versions of the functions listed below and include them with the type. The interconnect library assumes these functions are defined, so the compiler will generate an error if these are missing or incorrect. The packing of the complex type is defined by these functions.

Another approach is to declare all interconnects with standard types, like “`ac_int<size> xyz`”. Then convert the complex type to the `ac_int` before calling the read or write methods.

This example uses an RGB class with three 8-bit values:

```
struct RGB_type {
    char r,g,b;
}
```

First, a type is added to define the total number of bits and if the type is “signed”. These structs are defined for the standard types in `<catapult_install_dir>/Mgc_home/shared/include/ccs_types.h`. Complex types should not be defined as signed:

```
// Specialized template for RGB_type
template <> // Template arguments may be added as needed
struct mc_typedef_T_traits< RGB_type > {
    enum { bitwidth = 24, // Total bitwidth is 8*3 = 24 bits
          issigned = 0
    };
};
```

Next, two functions are defined to convert the `RGB_type` to and from a flat vector. The type conversion functions to and from `sc_lv` are provided in `<catapult_install_dir>/Mgc_home/shared/include/mc_typeconv.h`. Note that the packing is red in the MSB position, then blue and green and the packing must match exactly between these two functions.

```
template<>
void type_to_vector(const RGB_type &in, int length, sc_lv<24>& rvec) {
    rvec.range(23,14) = in.r;
    rvec.range(13,8) = in.g;
    rvec.range(7,0) = in.b;
}

template<>
void vector_to_type(const sc_lv<24> &in, bool issigned, RGB *result) {
```

```
result->r = in.range(23,14);  
result->g = in.range(13,8);  
result->b = in.range(7,0);  
}
```

Point-to-point Example Designs

This section describes the most common coding styles used to describe flow control. Each design simply reads the inputs and writes the result. Each design has two threads to show how to use ports and channels.

Fully Blocking IO – No flushing

This coding style is the simplest to write, but doesn't allow the design to flush. Instead the design will hard stall on any input or output, which can leave data “stuck” in the pipeline when the input stalls. This example uses the simple p2p, but this coding style also works with p2p_fifo or, if there is no back pressure, p2p_valid.

Note that a “reset_read” and “reset_write” function is called for every port and a “reset_read” and “reset_write” is called for every channel. While the reset functions will not affect the TLM simulation, they are required for the synthesis run to define the reset value of the control signals and for any pin-accurate simulation.

Header File:

```
#include <ccs_p2p.h>  
typedef design_T ac_int<10>;  
  
#pragma hls_design  
SC_MODULE (Blocking_IO_Example) {  
  
    // Interfaces  
    sc_in <bool> clk;  
    sc_in <bool> rst;  
    // Declare in and out, abstraction defaults to TLM  
    p2p<>::in<design_T> in;  
    p2p<>::out<design_T> out;  
  
    SC_CTOR(Blocking_IO_Example)  
    : clk("clk")  
    , rst("rst")  
    , in("in")  
    , out("out")  
    , chan("chan")  
    {  
        // Design has two threads "block1" and "block2"  
        SC_CTHREAD(block1,clk.pos());  
        reset_signal_is(rst,true);  
  
        SC_CTHREAD(block2,clk.pos());  
        reset_signal_is(rst,true);  
    }  
}
```

```
    }

private:
    // Threads
    void block1();
    void block2();

    // Local Connection
    p2p<>::chan<design_T> chan;
};
```

The C file defines the two threads. The read and write functions are called with no additional logic needed. In the case that there are multiple inputs or outputs, simply call the read or write when you need to access the IO.

```
#include "Blocking_IO_example.h"

void Blocking_IO_Example::block1() {
    in.reset_read();
    chan.reset_write();
    wait(1);
    while (1) {
        chan.write(in.read());
    }
}

void Blocking_IO_Example::block2() {
    out.reset_write();
    chan.reset_read();
    wait();
    while(1) {
        out.write(chan.read());
    }
}
```

Fully Blocking IO - With flushing

A different coding style is currently required if you want to have a design that will flush if there is no valid input data. The header file is the same for this case as for the "Fully Blocking IO - No flushing" case, so only the cpp file is shown.

```
#include "Simple_example.h"

void Blocking_IO_Example::block1() {
    in.reset_read();
    chan.reset_write();
    while (1) {
        wait();
        design_T temp;
        if ( in.nb_read(temp) ) { // NOTE: Uses nb_read
            chan.write(temp);
        }
    }
}
```

```
void Blocking_IO_Example::block2() {
    out.reset_write();
    chan.reset_read();
    while(1) {
        wait();
        design_T temp;
        if ( chan.nb_read(temp) ) {
            out.write(temp);
        }
    }
}
```

To allow this design to flush, the “read” function is replaced with the “nb_read” function for this input. This function takes a reference as its argument and returns true if the data is valid. To make sure that all outputs are valid, the output is guarded by the “if”. In this coding style all the code in the design should be inside of the “if”.

The main disadvantage of this coding style is that only one input is allowed. An input concentrator (essentially “and” gates on the control logic in the channel) can be used to merge multiple inputs into one to allow this coding style to be used if there are multiple inputs. The number of outputs is not restricted.

No Back Pressure

The “rdy” signal is not needed if there is no back pressure, so the simple p2p is replaced with p2p_valid. The updated header is shown below. Note that the synthesized result will have two fewer ports because there are no ready flags.

```
#include <ccs_p2p.h>
typedef design_T ac_int<10>;

#pragma hls_design
SC_MODULE (Valid_IO_Example) {

    // Interfaces
    sc_in <bool> clk;
    sc_in <bool> rst;
    // Declare in and out, abstraction defaults to TLM
    p2p_valid<>::in<design_T> in;
    p2p_valid<>::out<design_T> out;

    SC_CTOR(Valid_IO_Example)
    : clk("clk")
    , rst("rst")
    , in("in")
    , out("out")
    , chan("chan")
    {
        // Design has two threads "block1" and "block2"
        SC_CTHREAD(block1,clk.pos());
        reset_signal_is(rst,true);

        SC_CTHREAD(block2,clk.pos());
        reset_signal_is(rst,true);
    }
}
```

```

    }

private:
    // Threads
    void block1();
    void block2();

    // Local Connection
    p2p_valid<>::chan<design_T> chan;
};

```

The source code now uses non-blocking writes. For this design, the output is assumed to be taken, so some care is needed when testing this design, especially in source simulation. Either a blocking read or the “if” statement with the non-blocking read will work for p2p_valid, but the “if” style with the nb_read is shown.

```

void Valid_IO_Example::block1() {
    in.reset_read();
    chan.reset_write();
    while (1) {
        wait();
        design_T temp;
        if ( in.nb_read(temp) ) { // NOTE: Optional nb_read to allow
flushing
            chan.nb_write(temp); // NOTE: Uses nb_write
        }
    }
}

void Valid_IO_Example::block2() {
    out.reset_write();
    chan.reset_read();
    while(1) {
        wait();
        design_T temp;
        if ( chan.nb_read(temp) ) {
            out.nb_write(temp);
        }
    }
}

```

Connecting to RTL

If the source code needs to be connected to existing RTL or another pin-level interface, the source code can be declared with the SYN option. The channel and all ports connected to that channel must also be declared SYN. However, different channels on the same thread may be declared with different levels of abstraction. The code in the thread should not need to change. The signals inside the SYN port/channel should only be accessed directly for fixed timing threads, for example, an FSM implemented in an SC_METHOD.

```

#include <ccs_p2p.h>
typedef design_T ac_int<10>;

```

```
#pragma hls_design
SC_MODULE (Blocking_IO_Example) {

    // Interfaces
    sc_in <bool> clk;
    sc_in <bool> rst;
    // Declare in and out
    // Abstraction forced to SYN to allow access to signal elements.
    p2p<SYN>::in<design_T> in;
    p2p<SYN>::out<design_T> out;

    SC_CTOR(Blocking_IO_Example)
    : clk("clk")
      , rst("rst")
      , in("in")
      , out("out")
      , chan("chan")
    {
        // Design has two threads "block1" and "block2"
        SC_CTHREAD(block1,clk.pos());
        reset_signal_is(rst,true);

        SC_CTHREAD(block2,clk.pos());
        reset_signal_is(rst,true);
    }

private:
    // Threads
    void block1();
    void block2();

    // Local Connection
    p2p<SYN>::chan<design_T> chan;
};
```

Now that the channels are declared SYN, you can access the elements of the channel directly. For example:

```
chan.vld.read()
```

Shared Memory Example Designs

This section describes the most common coding styles used to describe share memories. The [] operator is used except for the bit-mask example. In all cases, a blocking synchronization (p2p_sync) is used to synchronize blocks to the memories can be shared between the internal threads. The designs also have memory ports to show how these are used.

Separate Read/Write

This example copies data, reversing the order at each step. A sync channel is used to synchronize at the end of each frame, which is why the synchronization is outside of the “for” loop. Synchronization is also allowed after each write to the RAM and Catapult will maintain the relative order of RAM and synchronization IO.


```
#include <ccs_p2p.h>
#include <ccs_sample_mem.h>
typedef design_T ac_int<10>;

#pragma hls_design
SC_MODULE (Separate_RW_RAM_Example) {

    // Interfaces
    sc_in <bool> clk;
    sc_in <bool> rst;
    p2p_sync<>::in<> start;
    p2p_sync<>::out<> done;
    mem_lrlw<>::rd_port<design_T,64> ram_in_port;
    mem_lrlw<>::wr_port<design_T,64> ram_out_port;

    SC_CTOR(Separate_RW_RAM_Example)
    : clk("clk")
    , rst("rst")
    , ram_in_port("ram_in_port")
    , ram_out_port("ram_out_port")
    , start("start")
    , done("done")
    , valid("valid")
    , local_RAM("local_RAM")
    {
        SC_CTHREAD(block1,clk.pos());
        reset_signal_is(rst,true);

        SC_CTHREAD(block2,clk.pos());
        reset_signal_is(rst,true);

        local_RAM.rclk(clk); // Explicitly bind RAM clock ports
        local_RAM.wclk(clk);
    }
private:
    // RAM Instances
    mem_lrlw<>::mem<design_T,64> local_RAM;

    // Threads
    void block1();
    void block2();

    // Other connections
    p2p_sync<>::chan<> valid;
};
```

For this design, the “if-valid” style is used with synchronization channels. This style requires that the design is pipelined or that the valid synchronization input is not asserted very often. Also, the core loop requires at least one “wait” statement or the TLM simulation will hang (the wait is not needed for fully blocking designs):

```
void Separate_RW_RAM_Example::block1() {
    ram_in_port.reset();
    local_RAM.reset_write();
    valid.reset_sync_out();
    start.reset_sync_in();
```

```

while (1) {
    wait();
    if ( start.nb_sync_in() ) { // if-valid style allows flushing
        for ( int i = 0; i < 64; i++ ) {
            // Bracket operator access for shared RAM and port
            local_RAM[i] = ram_in_port[63-i];
        }
        valid.sync_out();
    }
}

void Separate_RW_RAM_Example::block2() {
    local_RAM.reset_read();
    valid.reset_sync_in();
    done.reset_sync_out();
    ram_out_port.reset();
    while(1) {
        wait();
        if ( valid.nb_sync_in() ) { // if-valid style allows flushing
            for ( int i = 0; i < 64; i++ ) {
                // Bracket operator access for shared RAM and port
                ram_out_port[i] = local_RAM[63-i];
            }
            done.sync_out();
        }
    }
}

```

Dual-port

When using the dual-port type, the “mem” object has ports “A” and “B”, but the rw_ports are just like the single-port interface.

Like the separate read-write example, this example copies data, reversing the order at each step. A sync channel is used to synchronize at the end of each frame, which is why the synchronization is outside of the “for” loop.

```

#include <ccs_p2p.h>
#include <ccs_sample_mem.h>
typedef design_T ac_int<10>;

#pragma hls_design
SC_MODULE (Dual_Port_Example) {

    // Interfaces
    sc_in <bool> clk;
    sc_in <bool> rst;
    p2p_sync<>::in<> start;
    p2p_sync<>::out<> done;
    mem_2p<>::rw_port<design_T,64> ram_in_port;
    mem_2p<>::rw_port<design_T,64> ram_out_port;

    SC_CTOR(Dual_Port_Example)
        : clk("clk")

```

```

    , rst("rst")
    , ram_in_port("ram_in_port")
    , ram_out_port("ram_out_port")
    , start("start")
    , done("done")
    , valid("valid")
    , local_RAM("local_RAM")
{
    SC_CTHREAD(block1,clk.pos());
    reset_signal_is(rst,true);

    SC_CTHREAD(block2,clk.pos());
    reset_signal_is(rst,true);

    local_RAM.clkA(clk); // NOTE: Bind clock for each port here
    local_RAM.clkB(clk);
}
private:
    // RAM Instances
    mem_2p<>::mem<design_T,64> local_RAM;

    // Threads
    void block1();
    void block2();

    // Other connections
    p2p_sync<>::chan<> valid;
};

```

When accessing the local RAM, the port (A or B) must be specified. Only one thread may access the port.

```

void Dual_Port_Example::block1() {
    ram_in_port.reset();
    local_RAM.A.reset();
    valid.reset_sync_out();
    start.reset_sync_in();
    while (1) {
        wait();
        if (start.nb_sync_in()) {
            for ( int i = 0; i < 64; i++ ) {
                local_RAM.A[i] = ram_in_port[63-i];
            }
            valid.sync_out();
        }
    }
}

void Dual_Port_Example::block2() {
    local_RAM.B.reset();
    valid.reset_sync_in();
    done.reset_sync_out();
    ram_out_port.reset();
    while (1) {
        wait();
        if (valid.nb_sync_in() ) {
            for ( int i = 0; i < 64; i++ ) {

```

```
        ram_out_port[i] = local_RAM.B[63-i];
    }
    done.sync_out();
}
}
```

Writing with a bit-mask

This example is based on the dual-port example. The only difference is that block1 has been modified to only write the 8 lsb bits to the 10-bit RAM. Only the modified “block1” thread is shown. The [] operator cannot be used because two arguments must be passed to the write operation.

This code will leave the two msb bits in the RAM undefined, so additional code may be needed to initialize the RAM a known value before using the mask writes.

```
void Dual_Port_Example::block1() {
    ram_in_port.reset();
    local_RAM.A.reset();
    valid.reset_sync_out();
    start.reset_sync_in();
    while (1) {
        wait();
        if (start.nb_sync_in()) {
            design_T mask = 0; // Write all bits
            // Mask off bits 8 and 9
            mask[9] = 1;
            mask[8] = 1;
            for (int i = 0; i < 64; i++) {
                local_RAM.A.write(i, ram_in_port[63-i], mask);
            }
            valid.sync_out();
        }
    }
}
```

Point-to-Point I/O Library Reference

This section is a reference for the ccs_p2p library. The library can be found at:

<catapult_install_dir>/Mgc_home/shared/include/ccs_p2p.h - Include this file

<catapult_install_dir>/Mgc_home/shared/include/ccs_p2p – Contains p2p implementation

Naming Conventions

Non-blocking IO is prefixed with “nb_”. Non-blocking IO require special verification considerations because the values generated by the RTL may be different than the source code. Please refer to the definitions below for a full API.

The access functions are generally named “read” and “write” for data transfers, except for the synchronizer and event channels, which do not transfer data.

Each p2p will contain one or more of the following signals in the SYN view:

- vld – This is a feed-forward signal that indicates when data is valid.
- dat – This is a feed-forward signal that contains the data being transmitted.
- rdy – This is a feed-back signal that indicates the downstream block is ready to consume data.

When any of these signals are passed through an interface an “i_” is prepended to the name for inputs and a “o_” is prepended to the name for outputs. During SystemC simulation and in the generated RTL, the name of the p2p object is also pre-pended to the signal name. For example, a port declared:

```
p2p<>::in<ac_int<10> > input;
```

And constructed like:

```
input ("input")
```

will contain the following signal names:

```
input_i_vld, input_i_dat, input_o_rdy
```

Changing the string used during construction only affects the SystemC naming. Catapult will always name the signals based on the variable name in SystemC.

Signal

- Channel definition: `sc_signal<class T>`
- Read port definition: `sc_in<class T>`
 - IO type: Input
 - Methods: `T read()` – Returns current value of signal. Consumes no time.
- Write port definition: `sc_out<class T>`
 - IO Type: Output
 - Methods: `write(T data)` – Assigns data to signal and triggers an update event if signal changes. Consumes no time.

The signal is the only interconnect object allowed for synchronous dataflow modeling. The signal is also allowed between TTL blocks, but only if it is synchronized by other connections.

Blocking FIFO

Three signals used to directly connect two blocks.

<name>_vld: Data is valid, active high, driven active by the source when data is valid.

<name>_dat: Data to be transmitted across interconnect,

<name>_rdy: Target block is ready to take the data; this signal is high in any cycle

Data is transmitted in any cycle where "vld" and "rdy" are active.

The interconnect has a latency of one and supports a throughput of one.

- Channel definition: `p2p_fifo<>::chan<class T,unsigned int DEPTH>` - Constructor requires a string to name the object.
 - Reset Methods
 - `reset_read()` – Must be called by reader before first wait.
 - `reset_write()` – Must be called by writer before first wait.
- Write port definition: `p2p_fifo<>::in<T>`
 - IO Type: Input
 - Methods:
 - `reset_read()` – Must be called before first wait.
 - `T read()` – Attempts to read from FIFO and stalls process until read can complete if FIFO is empty. Consumes one or more cycles.
 - `bool nb_read(T &data)` – Attempts to read from FIFO and set data to read value. Returns true if data is read and false if data is not read. Data is undefined if value is not read. Consumes exactly one cycle.
- Read port definition: `p2p_fifo::out<T>`
 - IO Type: Output
 - Methods:
 - `reset_write()` – Must be called before first wait.
 - `write(T data)` – Attempts to write “data” into fifo and stalls process until write can complete if FIFO is full. Consumes one or more cycles.

- `bool nb_write(T data)` – Attempts to write “data” into FIFO and returns true if write is successful and false if it is not. Does not block process or guarantee write completes. Consumes exactly one cycle.

Blocking Channel

Three signals used to directly connect two blocks using the same protocol as a FIFO.

`<name>_vld`: Data is valid, active high, driven active by the source when data is valid.

`<name>_dat`: Data to be transmitted across interconnect,

`<name>_rdy`: Target block is ready to take the data; this signal is high in any cycle

Data is transmitted in any cycle where "vld" and "rdy" are active.

Underlying datatypes are `sc_lv` to allow clean mixed-language simulation.

The interconnect has a latency of one and supports a throughput of one.

- Channel definition: `p2p<>::chan<class T >` - Constructor requires a string to name the object.
 - Reset Methods
 - `reset_read()` – Must be called by reader before first wait.
 - `reset_write()` – Must be called by writer before first wait.
- Write port definition: `p2p<>::in<T>`
 - IO Type: Input
 - Methods:
 - `reset_read()` – Must be called before first wait.
 - `T read()` – Attempts to read from FIFO and stalls process until read can complete if FIFO is empty. Consumes one or more cycles.
 - `bool nb_read(T &data)` – Attempts to read from FIFO and set data to read value. Returns true if data is read and false if data is not read. Data is undefined if value is not read. Consumes no time.
- Read port definition: `p2p::out<T>`
 - IO Type: Output
 - Methods:
 - `reset_write()` – Must be called before first wait.

- `write(T data)` – Attempts to write “data” into fifo and stalls process until write can complete if FIFO is full. Consumes one or more cycles.
- `bool nb_write(T data)` – Attempts to write “data” into FIFO and returns true if write is successful and false if it is not. Does not block process or guarantee write completes. Consumes exactly one cycle.

Valid Channel

- Channel definition: `p2p_valid<>::chan<class T >` - Constructor requires a string to name the object.
 - Reset Methods
 - `reset_read()` – Must be called by reader before first wait.
 - `reset_write()` – Must be called by writer before first wait.
- Write port definition: `p2p_valid<>::in<T>`
 - IO Type: Input
 - Methods:
 - `reset_read()` - Must be called before first wait.
 - `T read()` – Attempts to read from FIFO and stalls process until read can complete if FIFO is empty. Consumes one or more cycles.
 - `bool nb_read(T &data)` – Attempts to read from FIFO and set data to read value. Returns true if data is read and false if data is not read. Data is undefined if value is not read. Consumes no time.
- Read port definition: `p2p_valid <>::out<T>`
 - IO Type: Output
 - Methods:
 - `reset_write()` – Must be called before first wait.
 - `bool nb_write(T data)` – Attempts to write “data” into FIFO and returns true if write is successful and false if it is not. Does not block process or guarantee write completes. Consumes exactly one cycle.

Synchronizer

Two signals used to directly connect two blocks for a peer-to-peer handshake

`<name>_vld`: Active high: Driven active by the source when it is ready

`<name>_rdy`: Active high: Driven active by the target when it is ready

Causes source or target to stall until both blocks are ready.

Underlying datatypes are `sc_lv` to allow clean mixed-language simulation.

The interconnect has a latency of one and supports a throughput of one.

- Channel definition: `p2p_sync<>::chan<>` - Constructor requires a string to name the object.
 - Reset Methods
 - `reset_sync_in()` – Must be called by reader before first wait.
 - `reset_sync_out()` – Must be called by writer before first wait.
- Write port definition: `p2p_sync<>::in<>`
 - IO Type: Input Synchronizer
 - Methods:
 - `reset_sync_in()` – Must be called before first wait.
 - `sync_in()` – Stalls design until done is called by block at other end of sync channel. Consumes one or more cycles.
 - `bool nb_sync_in()` – Asserts ready for one cycle and returns true if valid is set. Expected to be used only with the “`if(in.nb_sync_in())`” coding style.
- Read port definition: `p2p_sync<>::out<>`
 - IO Type: Output Synchronizer
 - Methods:
 - `reset_sync_out()` – Must be called before first wait.
 - `sync_out()` – Stalls design until start is called by block at other end of sync channel. Consumes one or more cycles.

Event

One signal used to trigger a block from a controller.

`<name>_vld`: Active high: Active for one cycle when controller triggers target.

Underlying datatypes are `sc_lv` to allow clean mixed-language simulation.

Back pressure is not supported by this channel, so event is "missed" if target is not waiting.

The interconnect has a latency of one and supports a throughput of one.

- Channel definition: `p2p_event<>::chan<>` - Constructor requires a string to name the object.
 - Reset Methods
 - `reset_wait_for()` - Must be called by reader before first wait
 - `reset_notify()` - Must be called by writer before first wait.
- Write port definition: `p2p_event<>::in<>`
 - IO Type: Input Synchronizer
 - Methods:
 - `reset_wait_for()` - Stalls design until notify is called by block at other end of event channel. Consumes one or more cycles.
 - `Bool nb_valid()` - Checks if valid is set and returns true if it is. Expected to be used only with the “`if(in.nb_valid())`” coding style.
- Read port definition: `p2p_event<>::out<>`
 - IO Type: Output Synchronizer
 - Methods:
 - `reset_notify()` - Must be called before first wait.
 - `nb_notify()` - Sends an event. Does not block or report if event is not received. Consumes exactly one cycle.

Upgrading from `modular_io.h` to `ccs_p2p.h`

All designs using `modular_io.h` should be ported to the `ccs_p2p.h` library. The table below shows the new P2P model for each of the old `modular_io.h` models:

Table 2-3. Mapping Channels to P2P Library

modular_io.h	ccs_p2p.h
<code>wait_hs<T></code>	<code>p2p<>::chan<T></code>
<code>en_hs<T></code>	<code>p2p_valid<>::chan<T></code>
<code>w_hs<T></code>	<code>signal<T></code> (a special wire type is not needed)
<code>sync_chan</code>	<code>p2p_sync<>::chan<></code>
<code>event_chan</code>	<code>p2p_event<>::chan<></code>

Table 2-4. Mapping Ports to P2P Library

modular_io.h	ccs_p2p.h
wait_in<T>	p2p<>::in<T>, p2p_fifo<>::in<T>
wait_out<T>	p2p<>::out<T>, p2p_fifo<>::out<T>
en_in<T>	p2p_valid<>::in<T>
en_out<T>	p2p_valid<>::out<T>
w_in<T>	sc_in<T> (a special wire type is not needed)
w_out<T>	sc_out<T> (a special wire type is not needed)
sync_in	p2p_sync<>::in<>
sync_out	p2p_sync<>::out<>
wait_event_in, event_in	p2p_event<>::in<>
wait_event_out, event_out	p2p_event<>::out<>

API

The put(T dat) and T get() functions in modular_io.h have been replaced with write(T dat) and T read() to be consistent with the TLM 2.0 standard API.

Sample Memory Reference

This section is a reference for the ccs_sample_mem library. The library can be found at:

<catapult_install_dir>/Mgc_home/shared/include/ccs_sample_mem.h - Include this file

<catapult_install_dir>/Mgc_home/shared/include/ccs_sample_mem – Contains sample memory implementation

Sample memories are provided as a starting point when no generated RAM is yet available for a design. RAMs are implemented with non-blocking interfaces and assume a coding style where a separate channel (usually a p2p_sync) is used to synchronize accesses to the shared memory between two threads.

Memories that are local to a process should generally be declared as local arrays and mapped using the standard interface synthesis flow. Sample memories also include Catapult libraries to allow this mapping.

To enable using a specific type of sample memory, it must be added to the list of selected libraries in the libraries tab. The solution library add command can also be called to add this library for any technology. The following commands add all the sample libraries for the Sample 65nm library:

```
solution library add calypto_mem_1rlw -rtlsyntool DesignCompiler -vendor  
Sample -technology 065nm  
solution library add calypto_mem_1p -rtlsyntool DesignCompiler -vendor  
Sample -technology 065nm  
solution library add calypto_mem_2p -rtlsyntool DesignCompiler -vendor  
Sample -technology 065nm
```

Naming Conventions and Implementation Rules

All RAM interfaces are non-blocking, so there is no special notation for blocking or non-blocking. This means that a separate channel, usually a “sync” or “event” channel should be used with the RAM.

All RAMs provide both [] operators and read or write access methods. The [] operators only allow simple reads and writes, so the read and write methods must be used for masked writes or other complex memory accesses.

For the dual-port RAM, where there are two ports with the same functionality. The ports are named “A” and “B” and any access to the RAM must define which ports is being used.

All Sample RAMs have one or more clock pins. These pins must be bound explicitly for internal RAMs. Synthesis assumes that the clock used for a RAM port is the same clock used by the attached thread. All accesses to Sample RAMs (both reads and writes) are sequential, one-cycle accesses.

Sample RAMs are configured to be read-first.

All Sample RAMs include light sleep, deep sleep and shut down pins for memory management. However, these pins are only driven to their active state and are not accessible from class method calls in the sample memories.

Separate Read/Write RAM

NOTE: When using this RAM, the memory index analysis on the addresses is not analyzed. If a thread reads and writes to the same RAM, then cycle constraints and/or wait statements must be used to constrain the relative timing of the reads and writes.

The read/write RAM has one port for writing values and one port for reading values. The ports do not share any pins, so there is no special designation for which port is being used. The signals for writing to the RAM are listed below.

- Clock ports:

- wclk – Clock for the write port.
- rclk – Clock for the read port.
- Write ports:
 - wme – Write enable. Memory will be written when this signal is high.
 - wadr – Write address
 - d – Write data
 - wm – Write mask. This signal is active low, so setting `wm = 0` will write all input bits.
- Read ports:
 - rme – Read enable. Memory output `q` is defined if this signal is high.
 - radr – Read address
 - q – Read data
- Channel definition: `mem_1r1w<>::chan<class T, unsigned int size >` - Constructor requires a string to name the object.
 - Reset Methods
 - `reset_read()` – Must be called by reader before first wait.
 - `reset_write()` – Must be called by writer before first wait.
- Write port definition: `mem_1r1w<>::wr_port<T,size>`
 - IO Type: Memory Input
 - Methods:
 - `T read(unsigned int address)` – Reads from the memory at location “address”.
 - `T operator [unsigned int address]` – Reads from the memory at location “address”.
- Read port definition: `mem_1r1w::rd_port<T,size>`
 - IO Type: Memory Output
 - Methods:
 - `write(T data, ac_int<bits(T)> mask=0)` – Writes to the memory at location “address”. The optional mask can be used to only partially write to the memory.
 - `T &operator[unsigned int address]` - Writes to the memory at location “address”. No bit-mask may be specified (the entire word is written).

Single-port RAM

NOTE: Only one port is available for this memory, so additional hardware must be written to manage sharing it between multiple threads.

The single-port RAM has one port for writing or reading values.

- Clock port:
 - clk – Clock for the read/write port.
- Read/write port:
 - me – Must be high to do a read or write. Memory is always read if me is high.
 - rw – Memory is written when the value is 1.
 - wadr – Write address
 - radr – Read address
 - d – Write data
 - wm – Write mask. This signal is active low, so setting `wm = 0` will write all input bits.
 - q – Read data
- Channel definition: `mem_1p<>::chan<class T, unsigned int size >` - Constructor requires a string to name the object.
 - Reset Methods
 - `reset ()` – Must be called by reader or writer before first access.
- Write port definition: `mem_1p<>::rw_port<T,size>`
 - IO Type: Memory Input
 - Methods:
 - `reset ()` – Must be called by reader or writer before first access.
 - `T readwrite(bool en_write, unsigned int address, T data = 0, ac_int<bits(T)> mask = 0)` – Reads and optionally writes to the memory. Allows a read and write operation in one cycle. Sample RAM is configured to be read first.
 - `T read(unsigned int address)` – Reads from the memory at location “address”.
 - `T operator [unsigned int address]` – Reads from the memory at location “address”.
 - `write(T data, ac_int<bits(T)> mask=0)` – Writes to the memory at location “address”. The optional mask can be used to only partially write to the memory.

- T &operator[unsigned int address] - Writes to the memory at location “address”. No bit-mask may be specified (the entire word is written).

Dual-port RAM

NOTE: When using this RAM, the memory index analysis on the addresses is not analyzed. If a thread reads and writes to the same RAM, then cycle constraints and/or wait statements must be used to constrain the relative timing of the reads and writes.

The dual-port RAM has two ports, A and B. To access the RAM or bind ports, you must refer to the A or B port:

```
mem_2p<>::mem<T,size> my_RAM;
...
x = my_RAM.A[address];
...
my_RAM.B[address] = x;
```

- Clock ports:
 - clka – Clock for the read/write port A.
 - clkb – Clock for the read/write port B.
- Read/write port A:
 - mea – Must be high to do a read or write. Memory is always read if me is high.
 - rwa –Memory is written when the value is 1.
 - wadra – Write address
 - radra – Read address
 - da – Write data
 - wma – Write mask. This signal is active low, so setting wm = 0 will write all input bits.
 - qa – Read data
- Read/write port B:
 - meb – Must be high to do a read or write. Memory is always read if me is high.
 - rwb –Memory is written when the value is 1.
 - wadrb – Write address
 - radrb – Read address
 - db – Write data

- wmb – Write mask. This signal is active low, so setting `wm = 0` will write all input bits.
- qb – Read data
- Channel definition: `mem_2p<>::chan<class T, unsigned int size >` - Constructor requires a string to name the object.
 - Reset Methods
 - `reset ()` – Must be called by reader or writer before first access.
 - All methods must be called on port A or B. For example, `my_mem.A.reset();`
- Write port definition: `mem_2p<>::rw_port<T,size>`
 - IO Type: Memory Input
 - Methods:
 - `reset ()` – Must be called by reader or writer before first access.
 - `T readwrite(bool en_write, unsigned int address, T data = 0, ac_int<bits(T)> mask = 0)` – Reads and optionally writes to the memory. Allows a read and write operation in one cycle. Sample RAM is configured to be read first.
 - `T read(unsigned int address)` – Reads from the memory at location “address”.
 - `T operator [unsigned int address]` – Reads from the memory at location “address”.
 - `write(T data,ac_int<bits(T)> mask=0)` – Writes to the memory at location “address”. The optional mask can be used to only partially write to the memory.
 - `T &operator[unsigned int address]` - Writes to the memory at location “address”. No bit-mask may be specified (the entire word is written).

Catapult Modular I/O Library

The modular I/O in the *modular_io.h* are templated to use any data type and should work for most designs. Writing custom modular I/O should only be done if you have an interface that does not match any of those provided in *modular_io.h*

The modular I/Os provided in *modular_io.h* are summarized in [Table 2-1](#) and described in the following sections.

Table 2-5. Optimized Modular I/O Summary

Table 2-6.

File	Description
en_in	Input with partial handshake (blocking read).
en_out	Output with partial handshake (non-blocking write).
w_in	Wire input (non-blocking read).
w_out	Wire output (non-blocking write).
wait_event_in	Start signal (blocking read).
wait_event_out	Done pulse (non-blocking write).
wait_in	Input with full handshake (blocking read).
wait_out	Output with full handshake (blocking write).

en_in

Scope: SC_MODULE interface

Input with partial handshake (blocking read).

Usage

```
SC_MODULE(test) {  
    ...  
    en_in<int> din;  
    ...  
  
    void foo() {  
        ...  
        int tmp = din.get();  
        ...  
    }  
}
```

Description

The en_in modular I/O implements a partial handshake that models ready-to-receive behavior for design inputs, the modular I/O only waits for valid data but does not request data. The design is stalled when the modular I/O function is called and data is not available.

Example

```
template <class T>  
class en_in : public get_if<T>  
{  
public:  
    sc_in<T> i_dat; //Input data  
    sc_in<bool> i_vld; //Input valid driven high when data is ready for  
    reading  
    ...  
    void reset() { }; //No reset behavior needed  
  
#pragma design modulario  
    T get() { //Input read method  
        while (true) {  
            if ( i_vld.read() ) break; //Wait until valid driven high, then return  
            data  
            wait();  
        }  
        return i_dat.read();  
    }  
}
```

Related Topics

[wait_event_in](#)

[wait_in](#)

en_out

Scope: SC_MODULE interface

Output with partial handshake (non-blocking write).

Usage

```
SC_MODULE(test) {  
    ...  
    en_out<int > dout;  
    ...  
  
    void foo() {  
        ...  
        int tmp = data;  
        dout.write(tmp);  
        ...  
    }  
}
```

Description

The en_out modular I/O implements a partial handshake that models ready-to-send behavior for design outputs. The write is non-blocking so it completes without any stalling each time the modular I/O function is called.

Example

```
template <class T>  
class en_out : public put_if<T>  
{  
public:  
    sc_out<bool> o_vld; //Output valid signal drive high with write data  
    sc_out<T>    o_dat; //Output data  
  
    ...  
    void reset() { o_vld.write(false); o_dat.write(0); } //Reset should set  
    output valid to false  
  
    #pragma design modulario  
    void put(T d) { //Modular I/O write method  
        o_dat.write(d); //Write output data  
        o_vld.write(true); //Drive valid high when writing  
        wait();  
        o_vld.write(false); //Drive valid low when finished writing  
    }  
}
```

Related Topics

[wait_event_out](#)
[w_in](#)

[w_out](#)

w_in

Scope: SC_MODULE interface

Wire input (non-blocking read).

Usage

```
SC_MODULE(test) {  
    ...  
    w_in<int> data_in;  
    ...  
    void foo() {  
        ...  
        int tmp = data_in.get();  
        ...  
    }  
}
```

Description

The w_in modular I/O implements a input interface without any handshake, meaning that the reading of data is unsynchronized external to the module. However this I/O is still scheduled, so it is registered based on how it feeds through the design pipeline. If registers are not desired, an sc_in or sc_signal using the [hls_direct_input](#) pragma can be used.

Example

```
template <class T>  
class w_in : public get_if<T>  
{  
public:  
    sc_in<T> i_dat; //Input data  
  
    ...  
  
    #pragma design modulario  
    T get() {  
        return i_dat.read(); //Unsynchronized Combinational read  
    }  
}
```

Related Topics

[en_out](#)
[wait_event_out](#)

[w_out](#)

w_out

Scope: SC_MODULE interface

Wire output (non-blocking write).

Usage

```
SC_MODULE(test) {  
    ...  
    w_out<int> data_out;  
    ...  
  
    void foo() {  
        ...  
        int tmp = 1021;  
        data_out.put(tmp);  
        ...  
    }  
}
```

Description

The w_out modular I/O implements an output interface without any handshake, meaning that the writing of data is unsynchronized external to the module.

Example

```
template <class T>  
class w_out : public put_if<T>  
{  
public:  
    sc_out<T>    o_dat; //Unsynchronized output data  
    ..  
  
    #pragma design modulario  
    void put(T d) {  
        o_dat.write(d); //Registered write of data  
        wait();  
    }  
}
```

Related Topics

[en_in](#)
[wait_event_out](#)

[w_in](#)

wait_event_in

Scope: SC_MODULE interface

Start signal (blocking read).

Usage

```
SC_MODULE(test) {  
    ...  
    wait_event_in start;  
    ...  
  
    void foo() {  
        ...  
        start.wait_for(); //Wait for start  
        for(int i=0; i<720; i++) {  
            ...  
            ...  
        }  
    }  
}
```

Limitations

- This modular I/O is synthesized as a separate process.
- The wait-for-start signal is always scheduled before the first data read. To align the start with the input data, you need to build start into the modular I/O where data is read.

Description

The wait_event_in modular I/O input implements a start signal that stalls the design until the input valid is sampled true. There is no data transferred with this modular I/O, and it is intended for synchronization only.

Example

```
class wait_event_in  
{  
public:  
    sc_in<bool> i_val; //Start signal drive high to indicate start  
    ...  
#pragma design modulario  
    void wait_for() //Modular I/O start function  
        while (true) {  
            wait();  
            if (i_val.read()) break; //Stall design until start signal is  
true  
        }  
}
```

Related Topics

[en_in](#)

[wait_in](#)

wait_event_out

Scope: SC_MODULE interface

Done pulse (non-blocking write).

Usage

```
SC_MODULE(test) {  
    ...  
    wait_event_out done;  
    ...  
  
    void foo() {  
        ...  
        for(int i=0;i<720;i++) {  
            data.write(stuff);  
        }  
        done.pulse();  
    }  
    ...  
}
```

Limitations

- This modular I/O is synthesized as a separate process.
- The done *pulse* signal is always scheduled in a separate clock cycle after other modular writes. To align the done signal with the output data, you need to build done into the modular I/O where data is written.

Description

The wait_event_out modular I/O output implements a done *pulse* that is driven high for one clock cycle when the modular I/O function is called. There is no data transferred using this modular I/O and it is used for synchronization only.

Example

```
class wait_event_out  
{  
public:  
    sc_out<bool> o_val; //signal set true when done  
  
    ...  
    void reset() {o_val = false;} //done should be reset to false  
    #pragma design modulario  
    void pulse() {  
        o_val = true; //drive done pulse high for one clock cycle  
        wait();  
        o_val = false; //set done false  
    }  
}
```

Related Topics

[en_out](#)
[w_in](#)

[w_out](#)

wait_in

Scope: SC_MODULE interface

Input with full handshake (Blocking Read)

Usage

```
SC_MODULE(test) {  
    ...  
    wait_in<int > din;  
    ...  
  
    void foo() {  
        ...  
        int tmp = din.get();  
        ...  
    }  
}
```

Description

The wait_in modular I/O implements a full handshake that models read-to-send/ready-to-receive behavior for design inputs. The design is stalled when the modular I/O function is called and data is not available.

Example

```
template <class T> //templatized for data type  
class wait_in  
{  
public:  
    sc_in<bool> i_vld; //Input indicates valid data ready for reading, will  
    stall hardware when driven low  
    sc_in<T> i_dat; // Input data  
    sc_out<bool> o_rdy; //Output indicates ready to read data when driven  
    high  
    void reset() { o_rdy.write(false); } //Reset should drive ready low  
    bool available() const { return i_vld.read(); }  
  
#pragma design modulario  
    T get() { //Method for reading input data  
        while (true) {  
            o_rdy.write(true); //drive ready for data signal high  
            wait();  
            if ( i_vld.read() ) break; //Exit and return data when input available  
        }  
        o_rdy.write(false); //Disable ready for data when finished reading  
        return i_dat.read();  
    }  
}
```

Related Topics

[en_in](#)

[wait_in](#)

wait_out

Scope: SC_MODULE interface

Output with full handshake (Blocking Write)

Usage

```
SC_MODULE(test){  
...  
    wait_out<int > dout;  
...  
  
void foo(){  
...  
    int tmp = data;  
    dout.write(tmp);  
...  
}
```

Description

The wait_out modular I/O implements a full handshake that models read-to-send/ready-to-receive behavior for design outputs. The design is stalled when the modular I/O function is called and data cannot be written because the downstream design/module is not ready to receive data.

Example

```
template <class T>  
class wait_out :  
{  
public:  
    sc_out<bool> o_vld; //Output indicates ready to write  
    sc_out<T> o_dat; //Output data  
    sc_in<bool> i_rdy; //Input driven high indicates write has completed,  
    //driven low will stall the write  
...  
    void reset() { o_vld.write(false); o_dat.write(0); } //Reset should drive  
    //valid signal low. Data reset is optional.  
  
#pragma design modulario  
    void put(T d) { //Modular I/O write  
        while (true) {  
            o_dat.write(d); //Write data  
            o_vld.write(true); //Drive valid high, indicates ready to send  
            wait();  
            if (i_rdy.read() ) break; //Keep driving write data and valid till  
            //write is received.  
        }  
        o_vld.write(false); //Disable valid signal after write completes.  
    }  
}
```

RAM and Modular I/O

- RAMs inside of a thread should be declared as an array.
- Shared RAMs require Modular I/O style connections.
 - All outputs from a SC_CTHREAD are registered
 - Source will have a combinational RAM read simulation model because true combinational RAM reads are not allowed.
- Source defines ports, but not technology details.
 - Target will instantiate technology mapped RAM model.
- C++ tricks are used to allow [] operators for read/write.

Example RAM

```
ram<1, 64, addr_t, data_t> mem1;

#pragma design
#pragma pipeline_init_interval 1
void ethread () {
    input1.reset();
    input2.reset();
    output.reset();

    while(true) {
        wait();
        for ( int i = 0; i < 64; i++ )
            mem1.mem_write(i,i);
        addr_t adr = input1.get() + 1;
        data_t d = input2.get() + 1;
        data_t t = d + mem1[adr];
        mem1[adr] = t;
        output.put(t);
    }
}
```

Splitting API Calls and Pipelined RAMs/Busses

- Some I/Os require pipelined access.
 - This is not allowed by Catapult in an individual modular I/O function.
 - The solution is to split the modular I/O function into multiple parts.
- The current flow requires user constraints between modular I/O calls.
 - Use labels to give names to operations so that a general script can be run.
 - This allows investigation of constraints directly in the source.

AHB Bus Master Example

```
void burst_read (addr_t base_addr, data_t rdata[], int len){
    bool grant;
    tot_t vec;
    bool rdy;

    REQ: request(true); // Modular I/O Call
    WAIT_GRANT: grant = wait_for_grant(); // Modular I/O Call

    for (int i=0;i<len;i++){
        vec[0] = !grant;
        vec.set_slc(1,data_t(0));
        vec.set_slc(33,base_addr+i);
        if (i==0)
            vec.set_slc(65,BUS_TRANS_NSEQ);
        else
            vec.set_slc(65,BUS_TRANS_SEQ);

        vec.set_slc(67,BURST_TYPE_INCR);

        PUT_ADDR: put(vec); // Modular I/O Call
        GET_DATA: rdata[i]=get(); // Modular I/O Call
        CHK_READY:rdy = get_rdy(); // Modular I/O Call
    }
    RELEASE: request(!rdy); // Modular I/O Call
}
```


Chapter 3

Basic Design Examples

Catapult provides support for three levels of abstraction in the SystemC language:

- **Untimed:** This input style allows SystemC to be used without changing the use model of Catapult. The input in SystemC is explicitly parallel, but the flow generally has all of the same benefits, and drawbacks, as the untimed C++ flow.
- **RTL:** While we expect most RTL designs will still be done in Verilog or VHDL, Catapult provides the ability to write most RTL constructs in SystemC, so you don't have to switch languages.
- **CATB (Cycle Accurate at Transaction Boundary):** This style includes timing, primarily for the description of interfaces. Transactions are reorganized based on constraints, but each transaction has cycle accurate timing defined in the source that stays in tact during synthesis.

Combine the Supported Styles in Designs

The styles based on SC_THREAD (Examples 3 and 4) are useful learning tools and can be used in systems that do not require an explicit reset. However, their use in real designs is limited unless the Open SystemC Initiative (OSCI) updates the language to allow explicit reset.

The recommended approach is to use sequential C++ (Examples 1 and 2) to describe your algorithm and datapath. The SC_THREAD style with Modular I/O (Example 5) should then be used to describe your complex, pin-level protocol and interface. Generally, you will be able to wrap the algorithmic C++ code with a SC_THREAD to build and integrate your design.

For more complex designs with multiple algorithmic processes, the recommended flow is to use `sc_fifo` to connect these processes. This is the purpose of example 6 and you will find several more complex examples of this type in other folders. For the 2010a release, you will not be able to add C++ hierarchy inside of SystemC processes, so some minor re-write will be required to build a multi-process design. Also for the 2010a release, `sc_fifo` objects may not cross hierarchy boundaries, so you will need to use modular I/O to connect between different modules.

All of these examples are available in the Catapult Toolkits directory. To access the toolkits from the Catapult session window, choose the **Help > Toolkits...** menu item to open the Toolkits window. Then select “**What’s New > New Features in 2010a > SystemC > Basic.**” Each example is in its own toolkit.

In order to compare the different levels of abstraction and coding styles, two chained FIR filters are used. This is a simple design that is easy to describe at all levels. The generated RTL from all of these designs has the same handshake protocol for "din" and "dout".

Example 1: Reference C Design

The first design is the reference design in C++. It can be found in the “Reference C Design for SystemC Examples” toolkit.

Table 3-1.

Design Files for Example 1

C_fir_cascade.h	SystemC design
directives.tcl	Catapult script
TB_C_fir_cascade.cxx	Testbench

Running Catapult with the "directives.tcl" file will setup the design with the following constraints:

- Throughput of 1.
- Full handshake on the primary I/O (done to more easily see the how this compares to the CATB style).
- The two scopes in the design are mapped to hierarchy using pragmas.
- The verification flow is run by Catapult on the source and also on the generated RTL.

This file is for reference. You'll find a comparison to this style in the other examples.

Example 2: Using ac::wait Timing Constraint

The second example adds new timing constraints to the source using "ac::wait()". This construct has no affect on simulation, but can be used as a synthesis constraint. During synthesis, wait statements can be used to constraint I/O and loops, they do not affect general operations. Designs with wait statements can also be pipelined, so you can think of them more as latency constraints instead of throughput constraints.

This example design can be found in the toolkit “How to use ac::wait() in C code.”

Table 3-2.

Design Files for Example 2

C_fir_cascade.h	SystemC design with ac::wait added
directives.tcl	Catapult Script
TB_C_fir_cascade.cxx	Testbench

How this design is different than the reference design in example 1:

- The "ac_wait.h" file is included and several wait statements are called in the "process1" design.
- The wait statements increase the design latency, but the design can still be pipelined with $II=1$. Wait statements only give minimum time constraints, so the scheduler has the option to increase the design latency beyond what is in the source constraint.
- If you look in the Gantt chart, you will see that the input data in "process1" is read in the first cycle and the coefficients are read in the third cycle. This is because of the wait statements.

Known Issue: If you add an "ac::wait()" at the start of a loop in C++, then you will see two waits at the start of that loop after synthesis. For pure C++ designs, Catapult will always add a wait at the start of every loop as part of the scheduling process.

Example 3: Using SC_THREAD/SC_FIFO

This is the first SystemC example. It is the closest to the C++ behavior and is still truly an "untimed" design. This style is only suitable for pure dataflow designs with blocking communication. The design has been re-structured to have the two processes run in an SC_MODULE. The testbench has also been put into an SC_MODULE that instantiates the main design.

This example design can be found in the toolkit "How to use SC_THREAD and SC_FIFO."

Table 3-3.

Design Files for Example 3

C_fir_cascade.h	SystemC design for synthesis
directives.tcl	Catapult script
TB_C_fir_cascade.cxx	SystemC testbench module
fir_filter.h	SystemC main design

A description of all the SystemC is beyond the scope of this document, but here are a few important points:

- General
 - The testbench uses the built-in error handler in SystemC. This is recommended because it will give additional information (time, current block) when there is an error. This can be seen partly in the main.cpp and partly in the "sink" function in the testbench.h file.
 - Arrays of ports cannot easily be named, so the default names in SystemC will be used for the coeffs ports.

- The simulation is truly untimed, so only "sc_fifo" types can be used. Attempting to use signals will give strange problems due to unpredictable delta cycle differences between blocks.
- Only the blocking methods on the sc_fifo can be used in an untimed design. Using any non-blocking method will cause the SystemC simulation to hang or cause problems similar to using signals.
- The sc_fifo inputs must be read before any non-blocking signal reads are done. If you don't read the blocking inputs first, you may not be able to synchronize the signal reads into the right delta cycles. Any non-blocking read before any blocking read can lead to strange simulation behavior.
- Setup for verification in Catapult
 - The testbench must include the file "mc_scverify.h" and use the CCS_DESIGN macro to designate the design. This provides the hooks for SCVerify to work correctly.
 - The sc_fifo ports must be bound to ports with handshake in the RTL for verification to function correctly. This means that they cannot be wire inputs in this design.
 - The testbench has been re-structured into a "source" and "sink" to better handle the SCVerify flow for SystemC. Note: In the SystemC flow, the generated RTL is connected directly to the original SystemC testbench. So, this testbench must be able to handle both the timing of the source and the timing of the RTL.

How this design is different than the reference design in example 1:

This design is written in SystemC, so the two processes run in parallel.

Example 4: Using SC_THREAD with Wait

This is the first example with wait statements. The example still uses SC_THREAD and SC_FIFO, but now each process is clocked so sc_signal can be used for the coefficients. However, reset is not well supported in SystemC for SC_THREADS, so there is no explicit reset in this design.

This example design can be found in the toolkit “How to use wait in an SC_THREAD.” The timing in this design has been modified to match the timing results from Example 2.

Table 3-4.**Design Files for Example 4**

sc_fir_cascade.h	SystemC design for synthesis
directives.tcl	Catapult script
testbench.h	SystemC testbench module
fir_filter.h	SystemC main design

How this is different than Example 3:

- The design now has a clock, which will add additional time between the reads and writes for the channels. However, the FIFO reads and writes are still blocking, so the processes will stop and wait if they cannot read or write data.
- Additional wait statements have been added to match the timing of example 2.
- An important difference between this design and Example 2 is that the source simulation has timing in it because the wait statements affect the source simulation.

Known Issue: The `CLOCKS` directive in Catapult must exactly match the clock name in the source. Catapult is not able to infer clocks without information from the directive.

Example 5: Using SC_CTHREAD and Modular I/O

The main problem with the four examples above is that they cannot be directly connected to RTL. This example switches to a style with explicit signals on the interface that will allow direct pin mapping to any RTL design. This style also supports explicit reset, so it is the preferred style for describing complex interfaces in SystemC.

In order to maintain an abstract core code, an interface class is used to describe "transactions" on the interface. The class generated by using this approach is called a Modular I/O. The functions that implement the transactions are called the Modular I/O API.

A detailed description of how Modular I/O works is beyond the scope of this discussion. Some important points are:

- A Modular I/O is user definable. You can define your own protocols.
- A Modular I/O API function will synthesize into a process in your RTL that is tightly synchronized with the core.
- The simulation behavior of the SystemC source only maintains the order of access for an individual API function call. This means that scheduling and pipelining optimizations can be used on these designs, but that you need to be aware that the relative timing between modular I/O can change.

This example design can be found in the toolkit "How to mix SC_FIFO with Modular I/O."

Table 3-5.

Design Files for Example 5

<code>MIO_fir_cascade.h</code>	SystemC design for synthesis
<code>directives.tcl</code>	Catapult script
<code>testbench.h</code>	SystemC testbench module
<code>main.cxx</code>	SystemC main design
<code>local_connect.h</code>	Extra modular I/O class used to connect two processes

How this design is different than Example 4:

- All of the "sc_fifo" interfaces are replaced with "modular I/O". There are wait_in and wait_out interfaces from the library that ships with Catapult and a "local_connect.h" file that describes the local connection for the "intermediate" variable.
- The "wait_in" and "wait_out" Modular I/O are defined as ports on a design, so the testbench has been re-factored to have separate modules for "source" and "sink".
- All communication in this design is done with signals. This means it could be directly connected to RTL code and, in most cases, the SystemC source could be simulated with the RTL.

Known Issue: Using any Modular I/O output will always add at least one additional cycle of latency to your design. The output Modular I/O is effectively a one-stage FIFO, so the output can be stored by the Modular I/O while the core design runs.

Example 6: Mixed Design

For most algorithms mapped to hardware, most of the communication can be accurately modeled using untimed connections. The recommended approach for these designs is to implement the internal communication using SC_FIFO and to implement the interfaces using modular I/O. This allows accurate protocols on the interface, but still allows for interface synthesis and I/O optimization between the algorithmic processes.

This example design can be found in the toolkit “How to describe pin-level interfaces with Modular I/O.”

Table 3-6.

Design Files for Example 6

MIO_fir_cascade.h	SystemC design for synthesis
directives.tcl	Catapult script
testbench.h	SystemC testbench module
main.cxx	SystemC main design

How this design is different than Example 4:

The connection "interconnect" is modeled using an sc_fifo instead of using a custom Modular I/O.

Chapter 4

RTL Examples

This chapter discusses how to write RTL in SystemC for Catapult. These RTL blocks are intended to be combined with abstract code using [Modular I/O](#). This chapter assumes you are familiar with RTL and SystemC, so it focuses mostly on how Catapult manages clocks and resets.

You will also notice that all of the examples are based on SC_METHOD. Catapult expects any design described using an SC_METHOD to be RTL.

Near the end of this chapter, you will find the AHB master interfaces that are used for the bus based Modular I/O examples in Chapter 4.

All of these examples are available in the Catapult Toolkits directory. To access the toolkits from the Catapult session window, choose the **Help > Toolkits...** menu item to open the Toolkits window. Then select “**What’s New > New Features in 2010a > SystemC > Writing RTL using SystemC.**” Each example is in its own toolkit.

Table 4-1.

Common Files

inc/fir_filter.h	Common algorithm file, modified to be more RTL compatible.
inc/types.h	Common datatypes file.
inc/local_modular_io.h	Special file, only needed for the AHB interface.

Example 7: Combinational Logic

The example toolkit, “Writing Combinational Logic with SystemC,” shows how to describe combinational logic using the multiply, accumulate hardware from the FIR filter.

Table 4-2.

Design Files for Example 7

comb_fir_filter.h	Describes hardware design
main.cpp	Defines sc_main
testbench.h	Defines a synchronous testbench

When describing combinational logic, make sure you make the SC_METHOD sensitive to all the inputs. Catapult will not allow a combinational design unless it is sensitive to all its inputs.

You will also notice that the GUI shows combinational designs differently:

- The architectural constraints window will not allow you to change constraints on these blocks. They are also shown in a faded color because you cannot set constraints.
- The loops in the design will be unrolled automatically and also will be shown as unrolled in architectural constraints.
- No cycle netlist or Gantt chart is generated for combinational designs.

However, the entire tool flow, including synthesis constraints and SCVerify will run on this design.

Example 8: Rules for SC_MODULE at the RTL level

At the RTL level, the rules are similar to the rules in RTL level VHDL and Verilog. Each register must be associated with a clock, but there is restriction in Catapult on the number of clocks and resets.

The example toolkit, “Writing RTL with no reset in SystemC,” shows how to describe and verify logic without a reset.

Table 4-3.

Design Files for Example 8

<code>RTL_fir_cascade.h</code>	Describes hardware design
<code>main.cpp</code>	Defines <code>sc_main</code>
<code>testbench.h</code>	Defines a synchronous testbench

In this case, the testbench will wait for a set amount of time for the don't care logic to "flush" from the shift register for the FIR filter. Then it will start testing starting with the first non-zero value.

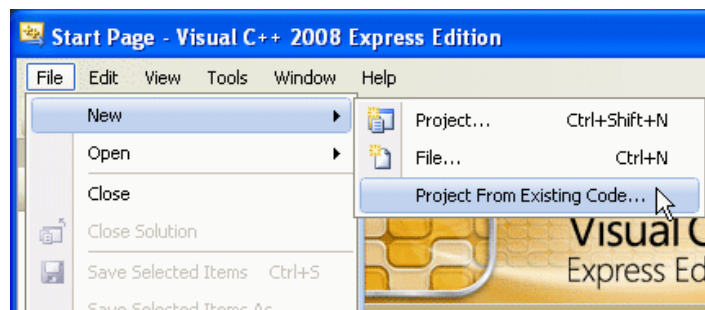
The directives for Catapult still require that you define both a clock and a reset, but the reset will not be used by this block.

Appendix A

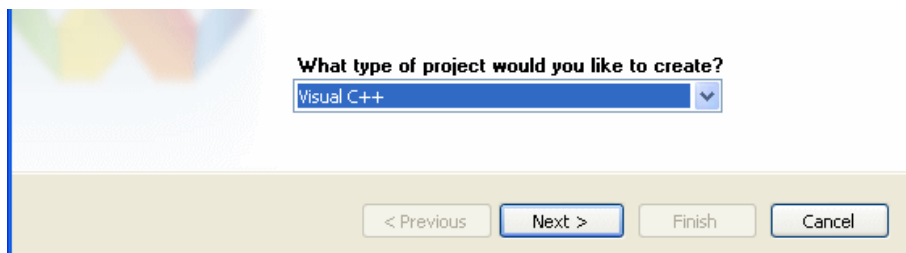
Setting Up SystemC Project in MSVC++ Express 2008

Use the following steps to set up SystemC projects in MSVC++ Express 2008 using a Catapult C precompiled library.

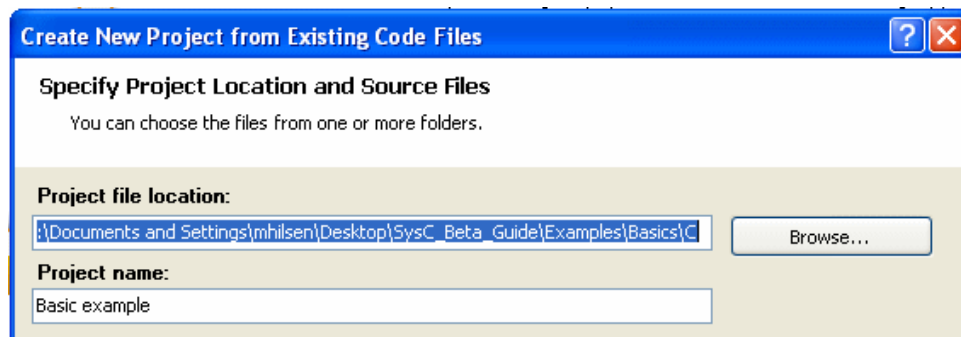
1. Start a new project by choosing the **File -> New -> Project From Existing Code...** menu item. That will open a setup wizard window.



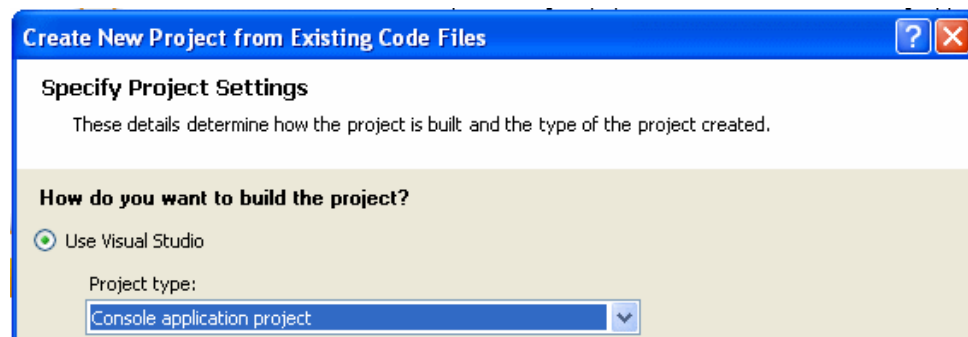
2. In the setup wizard window, set the project type to "Visual C++" and click **Next**.



3. Add the locations of projects files and project name, then click **Next**.

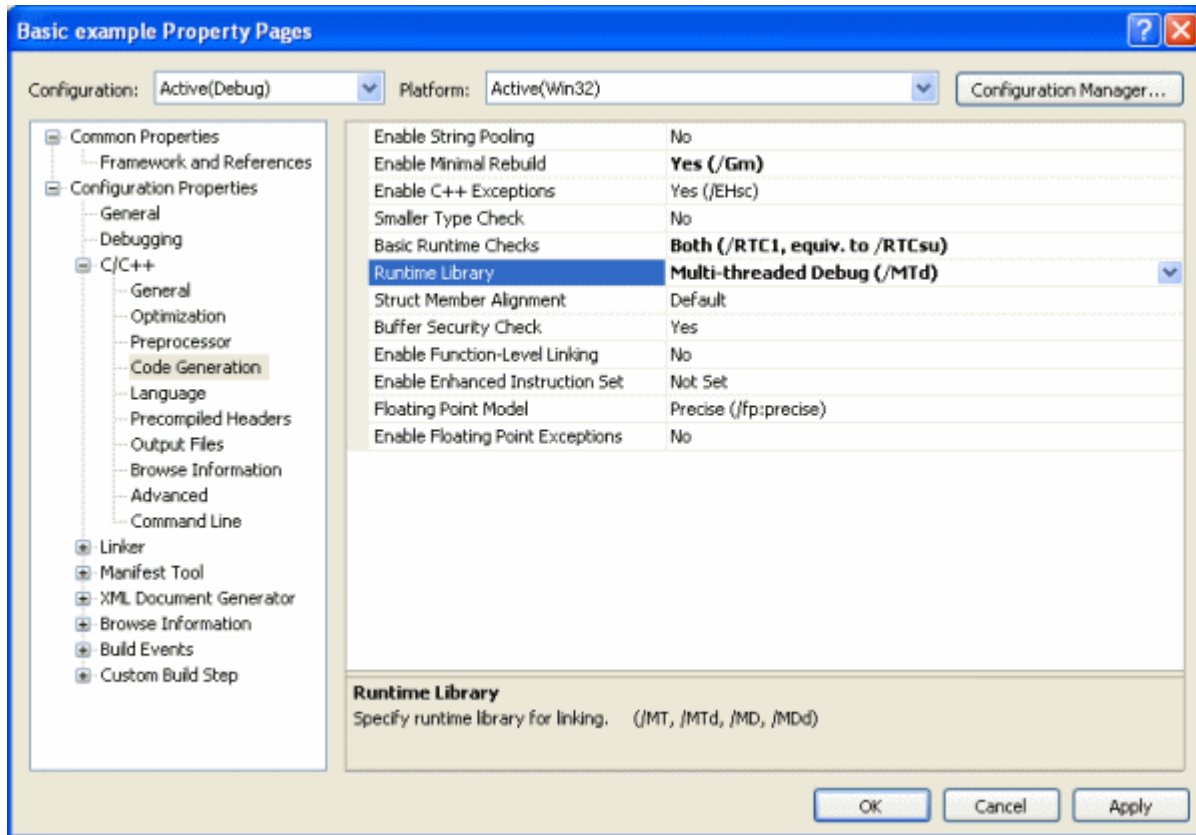


4. Choose "Console application project" and click **Finish**.

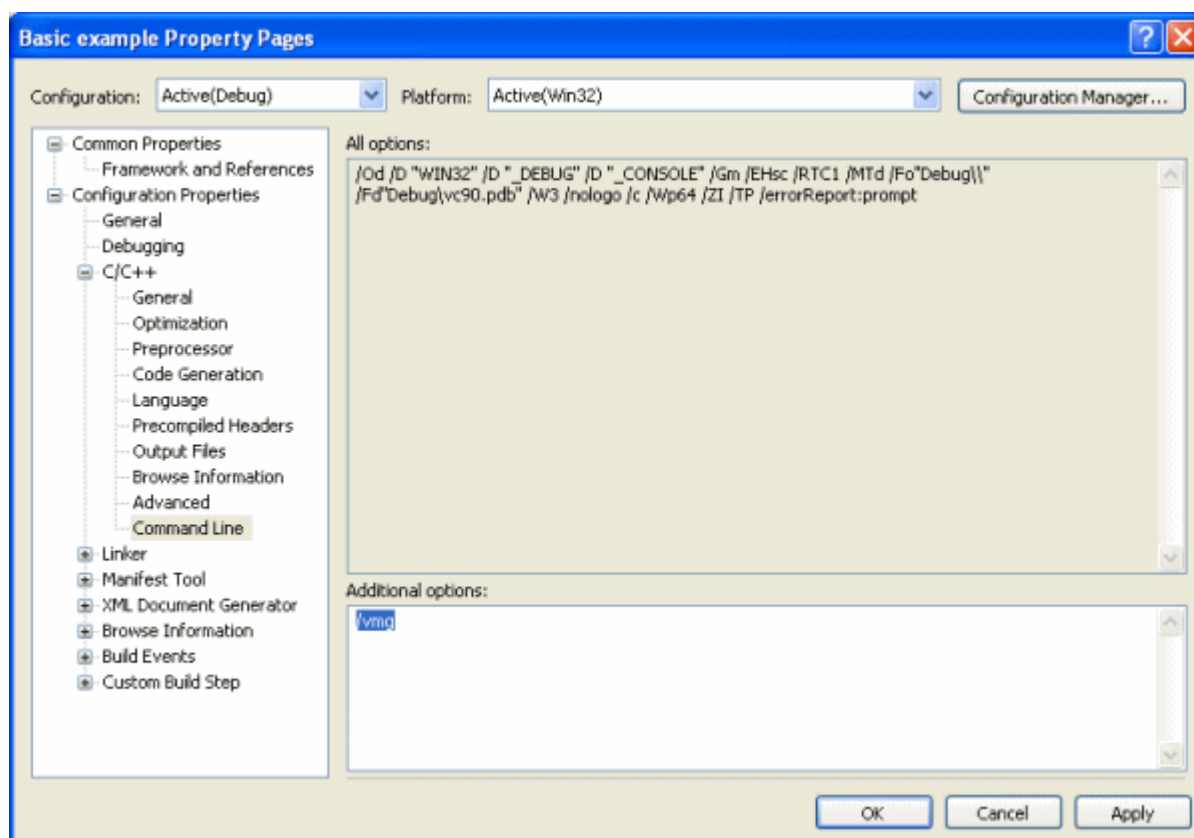


5. Change the project properties by choosing the **Project -> Properties...** menu item.

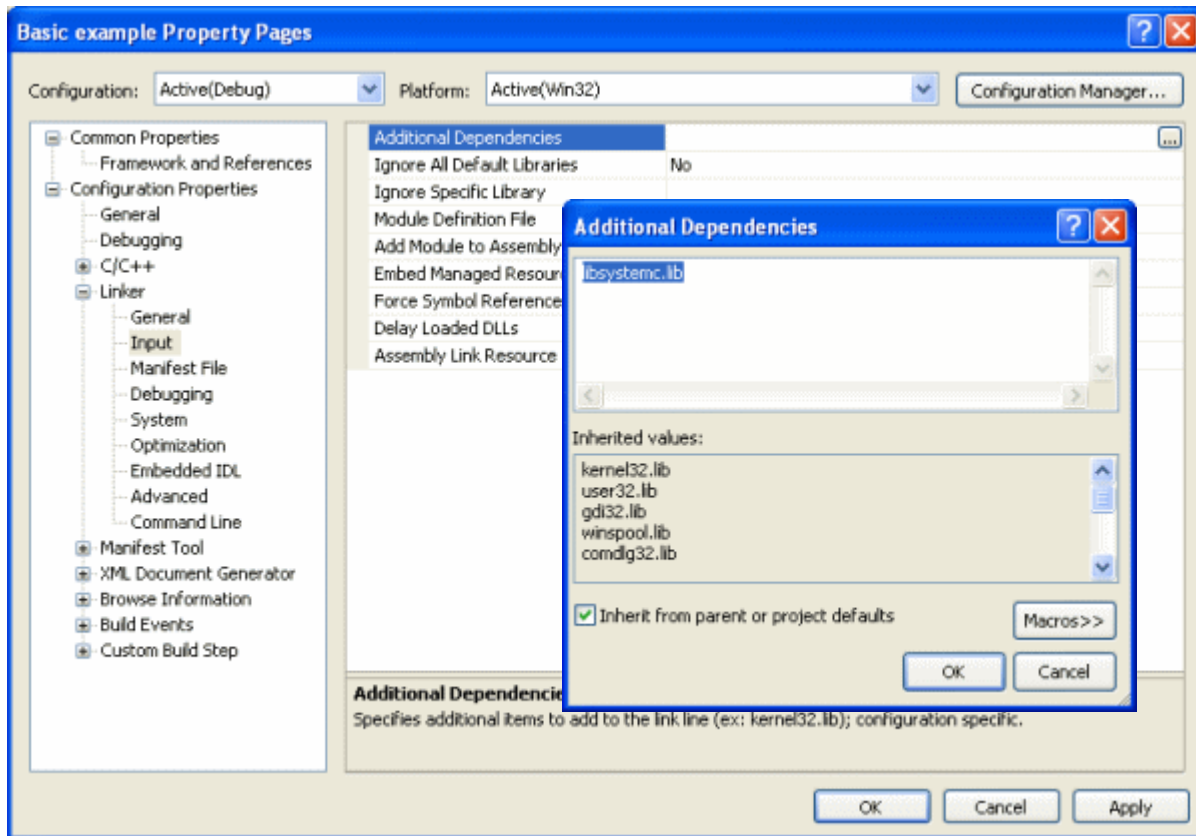
6. Set C/C++ runtime library to "Multi-threaded Debug (/MTd)" under **C/C++ -> Code Generation -> Runtime Library**.



7. Add "/vmg" under C/C++ -> Command Line -> Additional options:



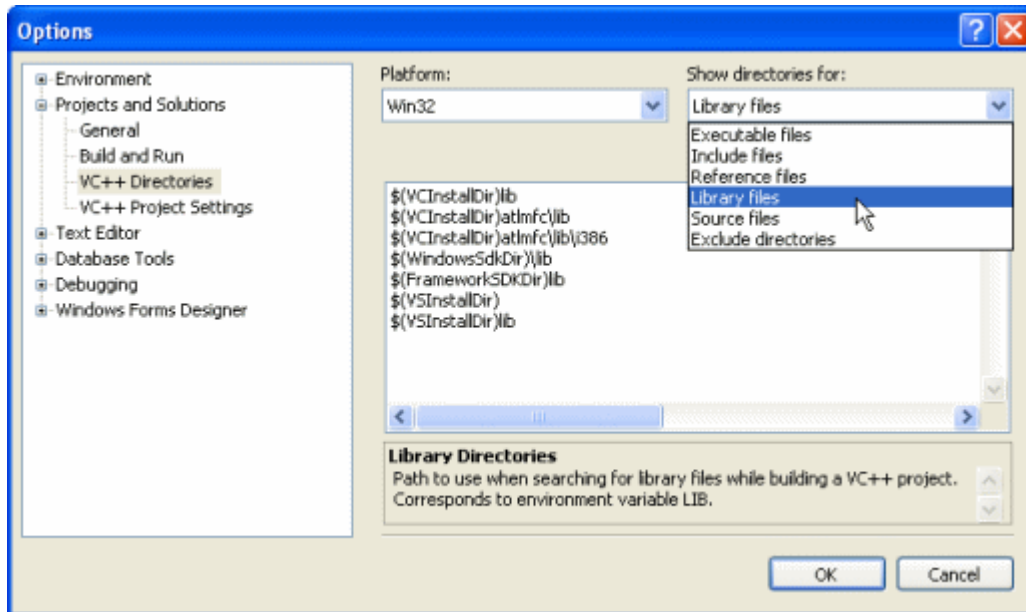
8. Type "libsystemc.lib" for the Catapult precompiled SystemC library under **Linker -> Input -> Additional Dependencies**:



To update the include file and library directory search paths for all projects, do the following.

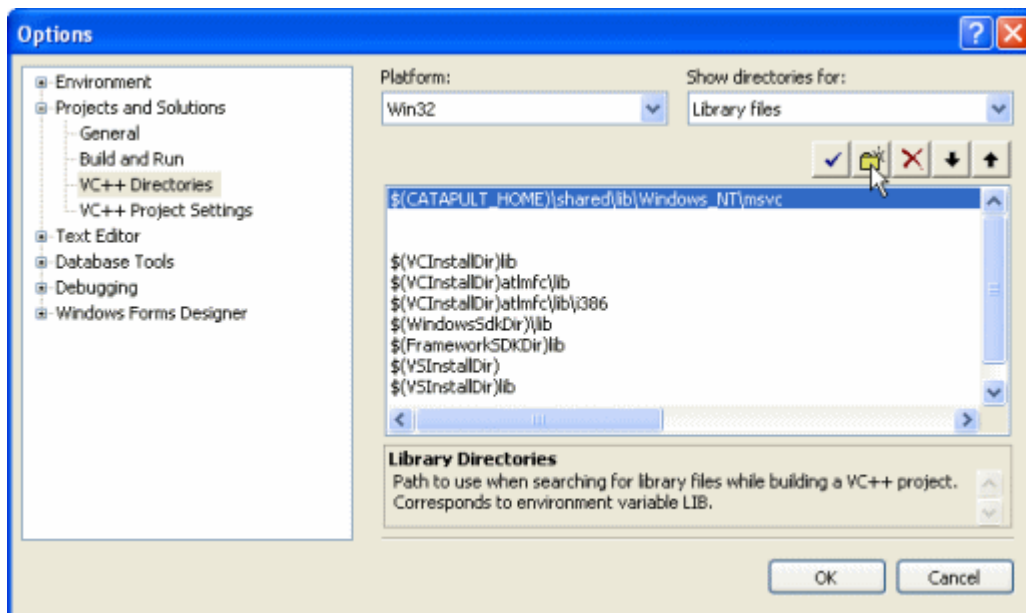
1. Select the **Tools -> Options...** menu item to open the Options window.

2. In the Options window, select **Projects and Solutions -> VC++ Directories**, then use the drop-down menu in the **Show directories for:** field to select **Library files**.



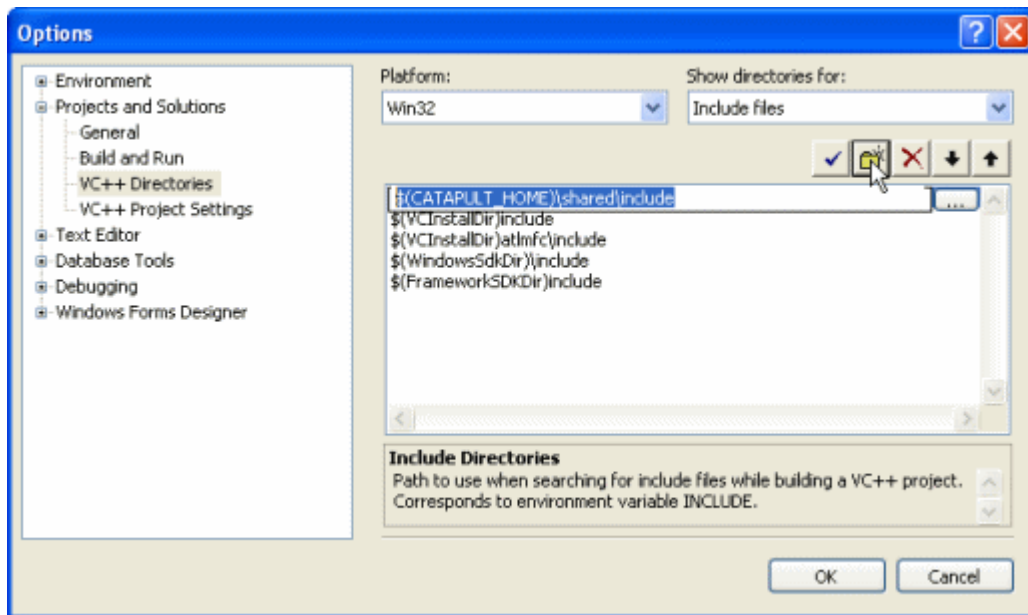
3. Click on the “New Line” icon and enter the string:

`$(CATAPULT_HOME)\shared\lib\Windows_NT\msvc`



4. Change the **Show directories for:** field to **Include files**:
5. Click on the “New Line” icon and enter the string:

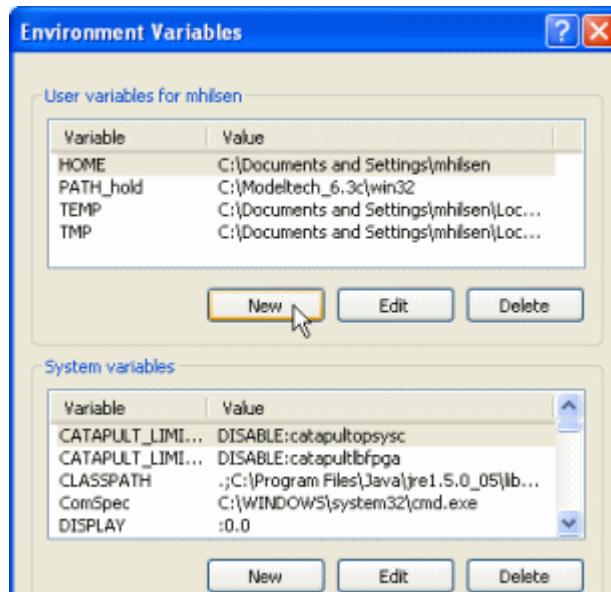
`$(CATAPULT_HOME)\shared\include`



6. Click “OK” in the Options window.

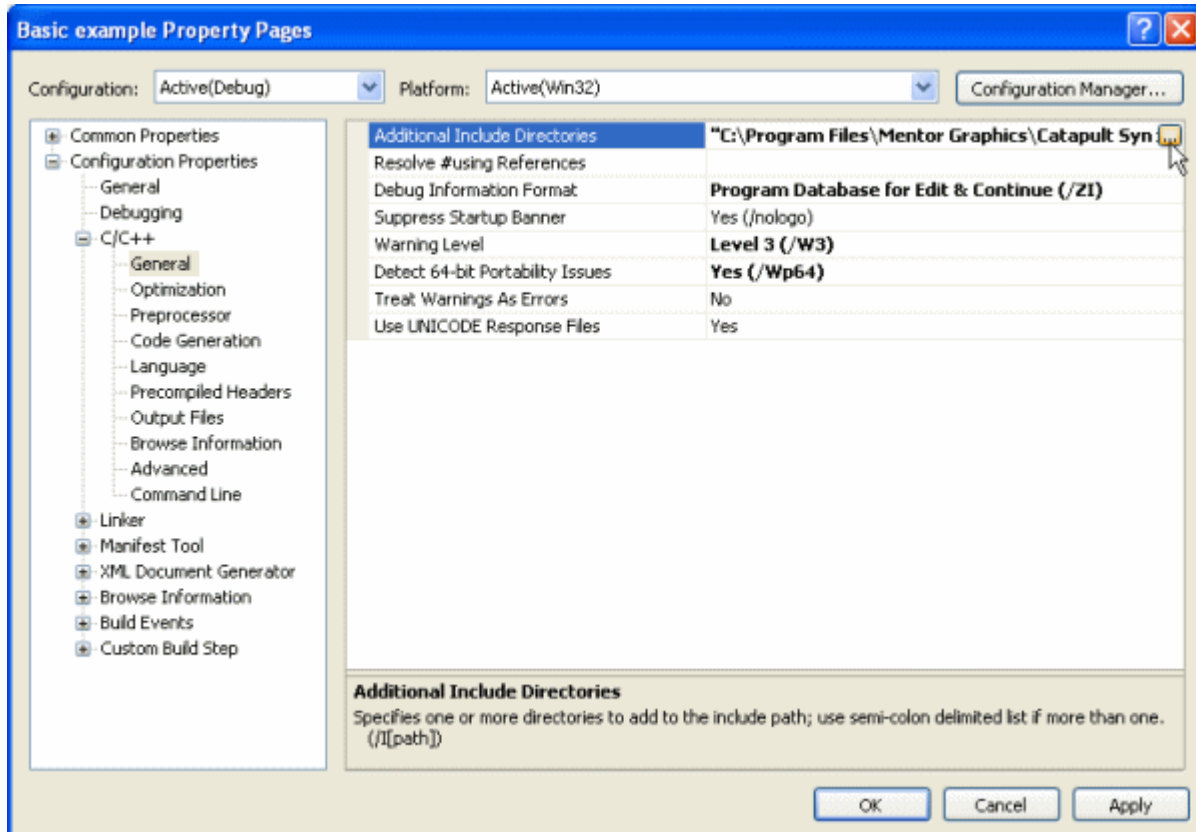
Set the CATAPULT_HOME environment variable (must restart MSVC++ for it to take effect).

- Open **My Computer -> Properties -> Advanced -> Environment Variables**. In the **User Variables** field click the **New** button and enter the following.
 - Variable name: **CATAPULT_HOME**
 - Variable value: **C:\Program Files\Calypto Design Systems\Catapult Synthesis 2010a.<ver> Release\Mgc_home**

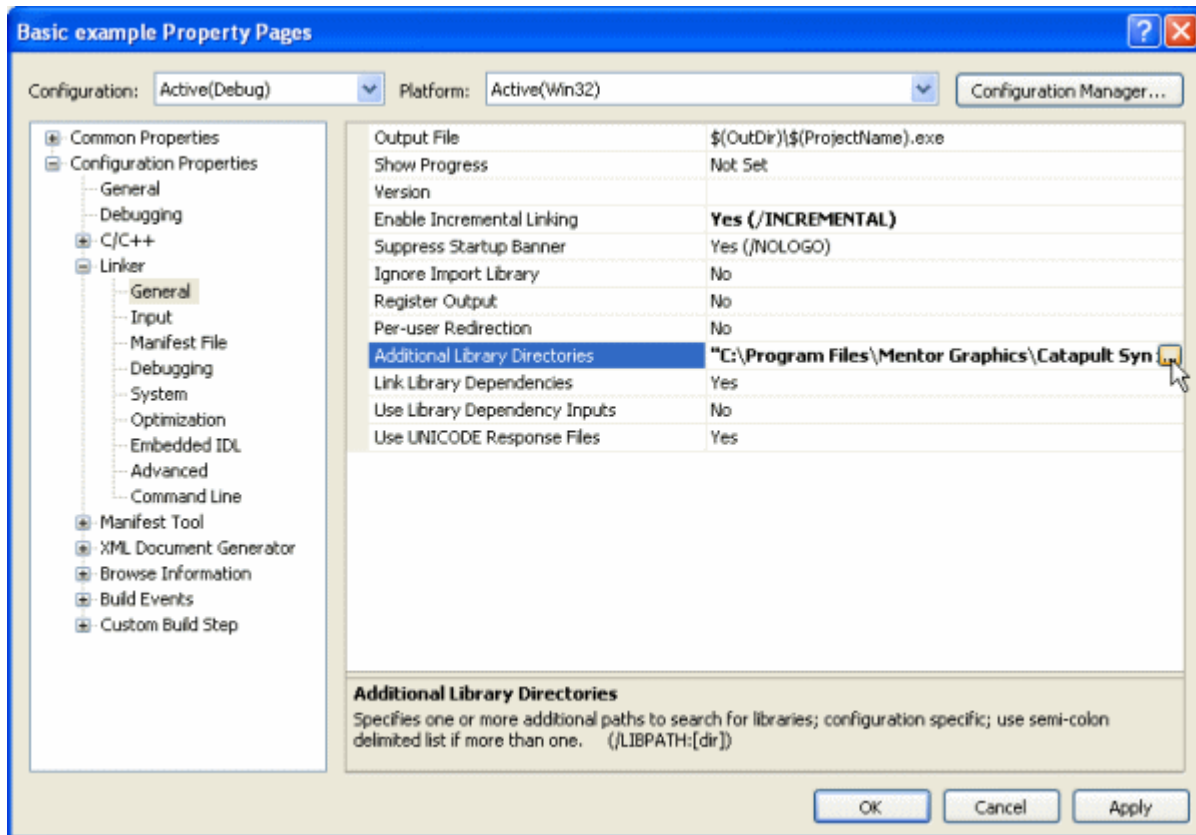


Add the include file and library directory search paths for the current project only.

1. In the MSVC window, open the project properties window by choosing the **Project -> Properties...** menu item.
2. Under **C/C++ -> General -> Additional Include Directories**, browse to the path to the Catapult include directory. (e.g. <Catapult install dir>\Mgc_home\shared\include).



3. Under **Linker -> General -> Additional Library Directories**, browse to the path to the SystemC library directory.
(e.g. <Catapult install dir>\Mgc_home\shared\lib\Windows_NT\msvc)



4. Click OK.

