Hybrid Threading Reference Manual

February 16, 2015

Version 1.4

901-000013-000

Trademarks

The following are trademarks of Convey Computer Corporation:

The Convey Computer Logo: 

Convey Computer

HC-1

HC-1^{ex}

HC-2

HC-2^{ex}

Trademarks of other companies

Intel is a registered trademark of Intel Corporation

Adobe and Adobe Reader are registered trademarks of Adobe Systems Incorporated

Linux is a registered trademark of Linus Torvalds

Revisions

Version	Description
1.0	January 22, 2014. Initial Release
1.01	February 10, 2014. Updated customer support link
1.1	July 22, 2014. Updated API, added clarification of global variable definition with memory read/write, added rspgrpId to ReadMemPoll, updated module message interface. Corrected default rspCntW in read memory interface.
1.2	August 12, 2014. Added streaming interface
1.3	November 5, 2014. Added barrier functionality, reset functionality of shared variables, reserve parameter of AddWriteStream, read stream tag support, read stream last element support. Updated model development.
1.4	January 12, 2015. Replaced zero parameter for staged variables with primOut parameter. Added readOnly parameter to host message destination. Added response groups to write streams, resulting in removal of WriteStreamPauseMask API. Updated message interconnect hti command. Other misc corrections. Moved content to Programmers Guide

Table of Contents

1 Overview

1.1 Introduction

The Convey Hybrid Threading (HT) development environment enables developers of Convey personalities to compile applications written in C/C++ to target both the host processor (x86) and the Convey coprocessor (FPGA). By allowing the developer to focus on the core functionality of the application instead of low-level hardware details, it provides a significant increase in developer productivity.

1.2 Document Content

- Overview of HT Architecture
- Description of HT Tools
- Command Reference

1.3 Related Documents

The Convey HT Programmers Guide contains detailed information on developing applications with the HT Tools. It is recommended that users have an understanding of the HT Architecture found in Section 2 before using the Programmers Guide.

1.3.1 Intended Audience

This document is intended for users interested in developing custom personalities for the Convey family of products. While it does raise the level of abstraction involved in programming the personality, it is recommended that the user have some experience with FPGA development. The user should also be capable of writing and debugging applications in C/C++.

1.3.2 Required Software

The HT environment requires the following components:

- Convey Personality Development Kit (PDK)
- SystemC libraries
- C/C++ compiler
- Xilinx ISE Design Suite14
- An HDL simulator for Verilog/VHDL simulation.
    - Mentor ModelSim
    - VCS

2 Overview of an HT Design

2.1 Overview of HT

The Hybrid Threading approach involves two sections of code to be written in C/C++: The main portion of the application, running on the x86 host processor, and the code to be compiled to the FPGA-based coprocessor.

The term “Hybrid Threading” was chosen because a thread on the host processor is migrated to the coprocessor and continues execution. Coprocessor operations are performed by compute machines, which act as functions in C/C++, each handling one piece of the algorithm.

Figure 1 shows a call graph of an application developed with HT.

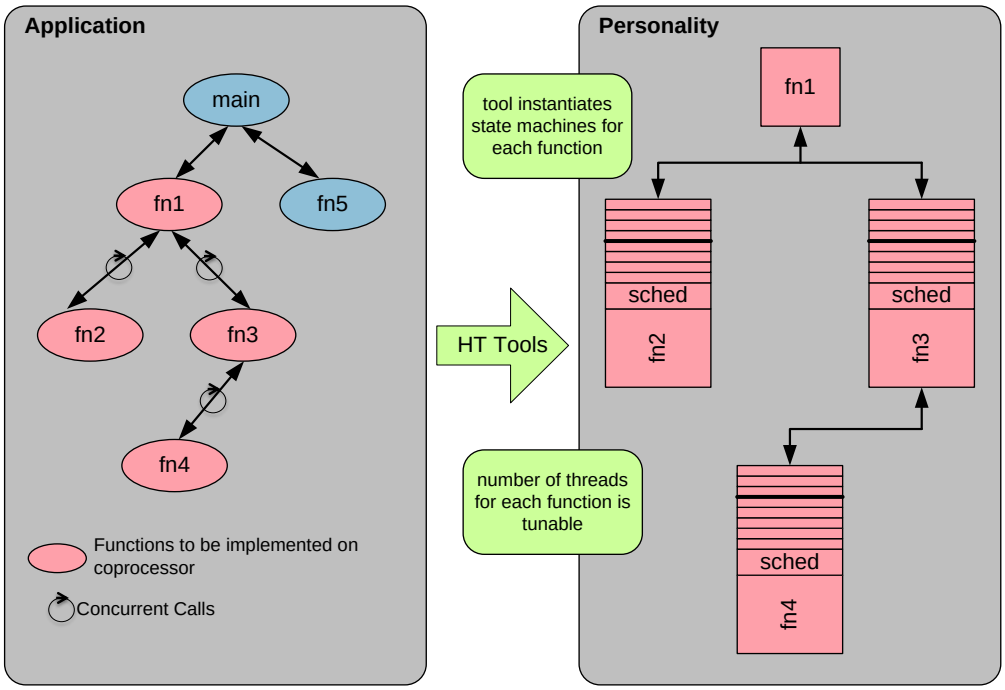

Figure 1 – Application Call Graph

The box on the left shows the original application running on the x86 host processor. The application main function calls two functions, fn1 and fn5. The fn1 function calls fn2 and fn3, and fn3 calls fn4. As this application is ported to the Convey coprocessor using HT, the developer chooses to move fn1, fn2, fn3 and fn4 to the coprocessor by implementing them in a Convey personality.

Executing as a compute machine and implemented as a finite state machine, each of these functions performs a piece of the overall application. Each compute machine has some number (typically 3-10) application-specific instructions. An instruction is generally 10-50 lines of C/C++ code allowing considerable work to be performed in a single clock cycle in each compute machine. The compute machines are multithreaded and the

threads are time division multiplexed so that on each clock cycle, one instruction is executed from one of the threads. If a thread is stalled waiting for data from memory, the thread can be paused until the memory response is received. Paused threads do not consume clock cycles in the compute machine.

The term “thread” is used to describe these units of execution because they are analogous to threads running on the x86. They execute as a subset of the overall application, and many threads can execute on a single piece of hardware. Each thread, within a compute machine can have its own private variables, and shared variables can be accessed by all threads. One major difference between the host instructions and the compute machine instructions is that the host has a fixed set of general purpose registers, and a fixed set of general purpose instructions. The compute machines, however, operate on the specific variables of a program. If a variable can be represented by 3 bits of data, then only 3 bits of state are used to store the value of the variable.

A collection of compute engines or modules makes up a unit, and a unit can be replicated up to 16 times per coprocessor Application Engine (AE).

A typical application starts the Units on the coprocessor then migrates threads to and from the coprocessor through memory-based queues. As a result, the units on the coprocessor are continuously processing, and doing so in parallel with the x86 processing.

The Hybrid Threading approach was designed to mimic the familiar software debugging environment of a standard processor with a fixed instruction set. A design can be compiled for debugging or for performance. When compiled for debugging, the unit is instrumented with functionality that provides visibility into a compute machine’s state. Asserts can be used to enable runtime checking, so that when an assertion fails, state can be saved for examination. When compiling for performance, all debugging and profiling capabilities are removed, freeing space in the FPGA to be used by compute logic.

2.2 HT Units

A unit contains all functions necessary to perform a coprocessor operation. Multiple instances of a unit are replicated on each Application Engine to increase performance. Units are typically self-contained and do not interact with other units.

A Unit contains several types of memory. Memory is defined as storage for an application referenced by its execution threads. Memory can be physically located on the FPGA in the form of Block Rams (in 36K increments) or Distributed Rams (in 32-bit increments). Memory is also used to reference main memory which is not located in the Unit, but off the FPGA. Memory that resides on the FPGA is limited in size (about 2MB) but has very low latency (6 nano-seconds) and high bandwidth (tera-bytes/sec). Memory that resides off the FPGA is large in size (16-64GB), longer latency (about 700 nano-seconds) and limited bandwidth (HC – 76.8GB/sec, WX – 42.6GB/sec).

The units that reside in a single AE share 16 coprocessor memory interfaces, where each memory interface can accept a single read or write request per clock. Within a Unit, all Modules that need to make memory requests must share these memory interfaces.

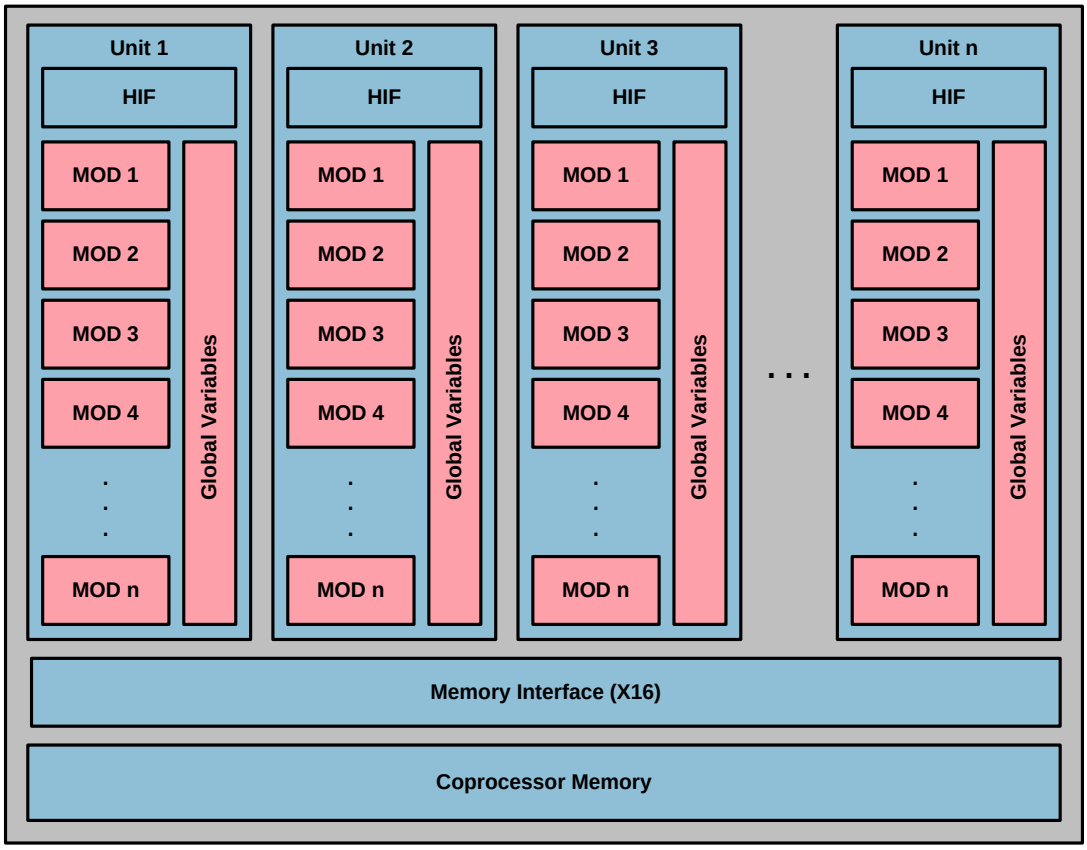

Figure 2 – HT Units

2.3 HT Modules

A Unit consists of multiple Modules. Modules perform a specific function for the unit. Each Module can have zero or one instruction execution state machines and zero or more internal variables. Modules can access global variables in other Modules. Modules can pass a thread to another Module within the same Unit using a thread interface between the two modules.

Modules can utilize three types of variables, private, shared and global. Private variables are available to a single thread in a Module. Shared variables are available to all threads in a Module and global variables are accessible by all Modules in a Unit.

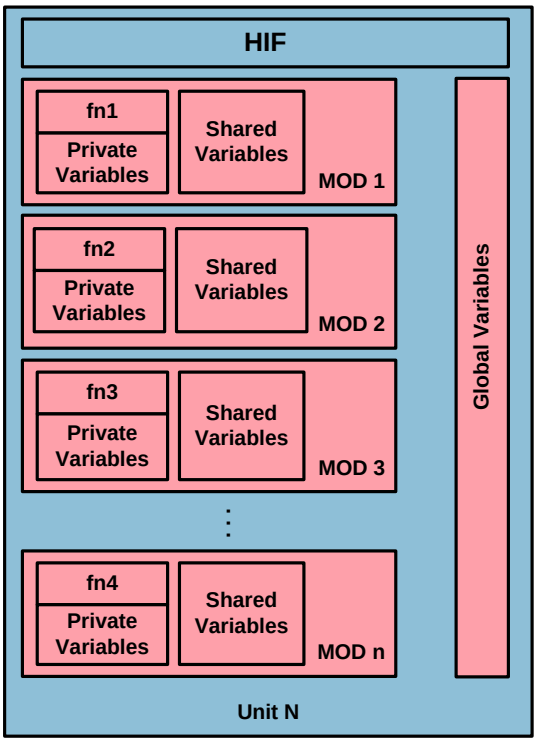

Figure 3 – HT Modules

2.4 Host Interface

The host application communicates with the HT Units through the Host Interface (HIF). The interface consists of a set of queues for each Unit on the coprocessor. Each Unit has a set of queues consisting of an inbound and outbound message queue and an inbound and outbound data queue, as shown in Figure 4. The queues are located in host memory.

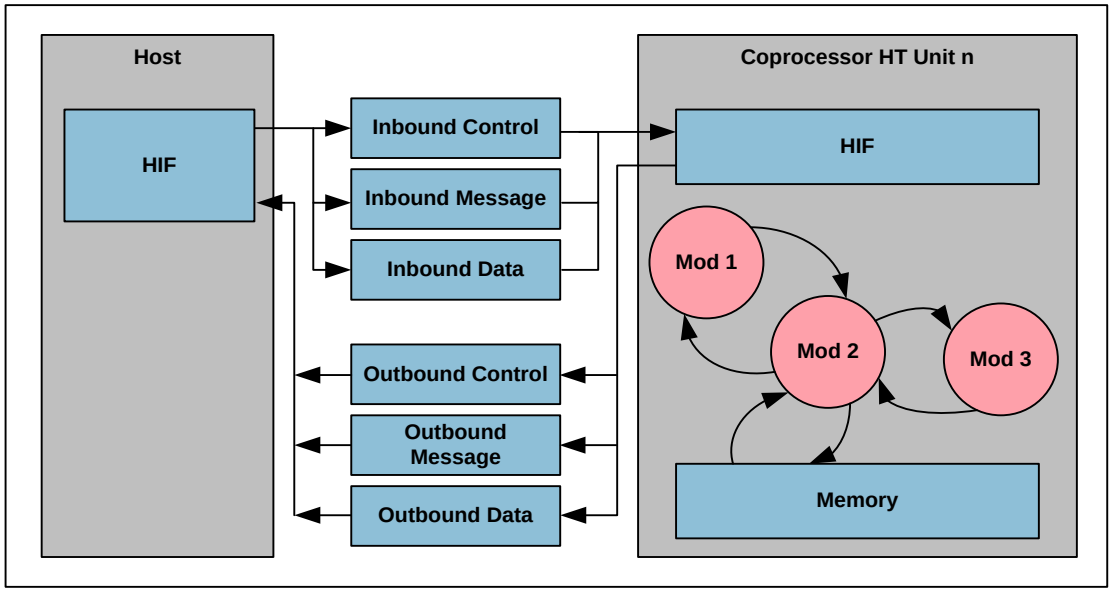

Figure 4 – Host Interface

The order of transactions sent may be different than the order that the transactions are received due to queuing and staging delays. The following ordering rules apply to the host interface, both inbound and outbound.

- Each HT Unit has a separate and independent host interface. The order of transactions on one unit is completely independent of the order of transactions on a different unit.
- Within an HT Unit, transactions are ordered within each transaction type. Specifically, messages are received in the order they are sent, data and data markers are received in the order sent, and calls/returns are received in the order they are sent.
- No other ordering rules should be assumed.

2.4.1 Host Message Interface

The host message interface supports messages from the Host to one or more Modules in a Unit (inbound) or from a Module to the Host (outbound). The inbound and outbound message queues hold 8 byte messages consisting of a single bit valid indication, a 7 bit type field and a 56 bit data field, as shown in Figure 5.

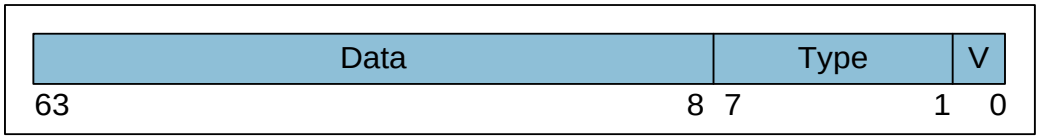

Figure 5 – Inbound / Outbound Message

Multiple messages can be used to provide information that requires greater than 56-bits. The host can provide parameters to a Unit by sending a Host Message to the Unit. Typically the host message interface is used to transfer relatively small amounts of data.

2.4.2 Host Data Interface

The host data interface is a queued interface. The inbound and outbound data queues are comprised by multiple variable length data blocks.

Data is sent when any one of the following occurs:

- A data block is full
- Under timer control (timer expired)
- A host message is sent
- A call or return is sent
- An explicit instruction to flush the queue is executed.

The size of the data block is n X 8 bytes, where n is specified in the routine used to send or receive host data.

The default timer value for both inbound and outbound data is 1000 usec. This value can be changed when the host interface is constructed.

2.5 Variables

Three types of variables are supported, private, shared and global. This section describes the different types of variables available in HT personalities.

2.5.1 Private Variables

A single thread in a Module can access private variables. Private variables can be scalar or memory.

Private variables can be

- modified by the Module state machine
- passed as an argument of a call
- returned as parameter from a call

Private variables are initialized to zero or set to the value passed in the call.

2.5.2 Shared Variables

All threads of a Module can access shared variables. They can be scalar, memory or queue. Shared variables are stored on chip with a user option to utilize either distributed or block RAM.

Shared variables can be modified by memory reads, host messages or the state machine. Shared variables cannot be passed as arguments for a call or be modified on a return.

Shared variables are not initialized.

2.5.3 Global Variables

Global variables are accessible by all threads of all Modules in a Unit. They can be scalar or a memory.

Global variables can be modified by memory reads or the state machines contained in the Unit. Global variables cannot be passed as arguments for a call or be modified on a return.

Global variables are not initialized.

2.6 HT Tools

There are three steps required in compiling an application with HT: The Hybrid Threading Linker (HTL), and the Hybrid Threading Verilog translator (HTV), and the FPGA build. Figure 6 illustrates the HT tool flow.

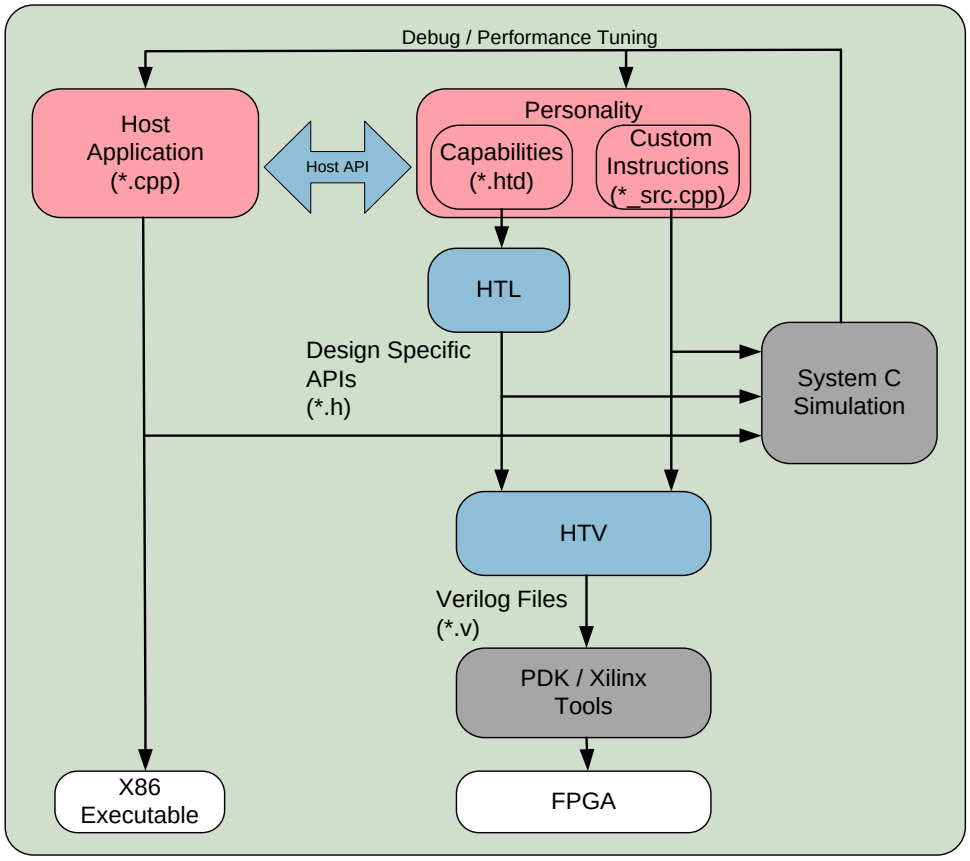

Figure 6 – HT Tool Flow

The coprocessor application can be implemented in a high level model to allow simulation of the host application, and the host / coprocessor interface, prior to development of the hardware, as shown in Figure 7.

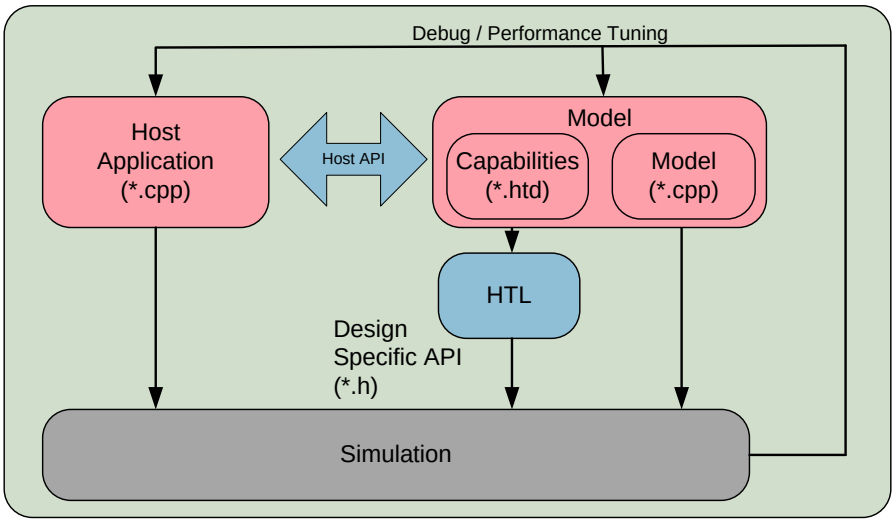

Figure 7 – HT Model Flow

2.6.1 Hybrid Threading Linker - HTL

HTL is a configurable API generator. It takes the capabilities defined in the ***htd*** file and the API used by the host application, the model and the custom instructions as shown in Figure 8.

Figure 8- HTL Design Specific API Generation

2.6.2 Hybrid Thread Verilog Translator - HTV

The Hybrid Threading Verilog Translator (HTV) translates HT coprocessor application files (Pers*_src.cpp) written in C++ into Verilog. This tool generates Verilog code that will be compiled into the FPGA, along with the Convey PDK infrastructure.

HTV supports a subset of C++ language. The supported language subset is shown in Appendix B.

A Appendix – Definitions

Term	Definition
Thread	A thread within the hardware threading model has the same definition as a thread in a software model. The only difference is that threads on the coprocessor execute application specific instructions. A single instruction can represent 10's of lines of high level source code.
Instruction	On the host processor, an instruction is defined by the x86 instruction set architecture (ISA). On the coprocessor, an instruction is the operation performed by a thread making a single pass through a module's state machine. Note that an instruction may perform its operation across multiple pipelined clocks.
Unit	A unit contains all functions necessary to perform a coprocessor operation. Note that multiple instances of a unit are replicated on each Application Engine to increase performance. Units are typically self-contained and do not interact with other units. One exception to this is when reduction operations must be performed across all units.
Module	A unit consists of multiple modules. Modules perform a specific function for the unit. Each module can have zero or one instruction execution state machines and zero or more internal memories. Modules can access memories in other modules. Modules can pass a thread to another module within the same unit using a thread interface between the two modules.
Memory	Memory is defined as storage for an application referenced by its execution threads. Memory can be physically located on the FPGA in the form of Block Rams (in 36K increments) or Distributed Rams (in 32-bit increments). Memory is also used to reference main memory which is off the FPGA. Memory that resides on the FPGA is limited in size (about 2MB) but has very low latency (6 nano-seconds) and high bandwidth (tera-bytes/sec). Memory that resides off the FPGA is large in size (16-64GB), longer latency (about one micro second) and limited bandwidth (80GB/s shared by all Units).

Term	Definition
Host Interface	The host interface is a module within each Unit that provides the communication paths between the host application and the Unit.
Memory Interface	The Units that reside on a single AE FPGA share 16 memory interfaces, where each memory interface can accept a single read or write request per clock. Within a Unit, all Modules that need to make memory requests must share a small number of these memory interfaces. As an example, if an AE has eight Units, then each Unit can have two dedicated memory interfaces. The ***htl*** software generates a module for each memory interface that is used by a Unit.
Host Message	The host can provide parameters to a Unit by sending a Host Message to the Unit. A single Host Message is 64-bits in size. Each message has a one bit message valid bit, a 7-bit message type field, and 56-bits of message data. Multiple messages can be used to provide information that requires greater than 56-bits.
Hardware Thread Shared State	State that is shared by all threads within a Module. A single copy of shared state exists. Shared state is often accessed in critical regions of an application. Care must be taken that shared state is accessed by a single stage in a state machines instruction pipeline.
Thread Private State	State that is private to a thread within a Module. Each thread executing within the Module will have its own copy of thread private state. The private state is indexed by the thread's Id (*HtId*).

B Appendix – Acronyms

Term	Definition
AE	Application Engine – FPGAs on the coprocessor that contain the custom personality
API	Application programming interface
FPGA	Field programmable gate array
HC-1/1EX/2/2EX	Hybrid Core platform
HDL	Hardware description language
HIF	Host interface
HT	Hybrid threading
HTD / htd	Hybrid threading description
HTL / htl	Hybrid threading linker
HTV	Hybrid threading verilog
PDK	Personality Development Kit – Provides interface between the HT personality and the coprocessor
VHDL	VHSIC Hardware Description Language. VHSIC is an abbreviation for Very High Speed Integrated Circuit
WX-690/2000	Wolverine Platform

C Appendix – Hybrid Thread Description

This Appendix lists and describes the Hybrid Thread Description (htd) file commands supported by the htl software application. Each command description provides a brief description of the command, the available parameters, any subcommands, an example and the generated interface.

The generated interface may be the interface to a routine or a macro definition. There are three interfaces that are generated.

- Host interface
- Personality interface
- High level model interface

The host interface is used by the application running on the host. The personality interface is used by the HT instructions of the HT modules. And the model interface is used by a C/C++ model of the personality.

AddModule(…)

AddModule(name=<name> {, clock=<clock_rate>} {, htIdW=<width>} {, reset=<inst>} {, pause==<true or false>})

Description:

Specifies that a module with the specified name exists in the design. Each module can be single threaded (htIdW=0) or multi-threaded. A reset sequence of instructions can be performed when a unit initially begins execution from a host dispatch. The reset sequence starts at the instruction provided by the reset parameter. The reset thread can execute a sequence of instructions and then terminate (using HtTerminate), or can continue to execute instructions for the duration of the dispatch.

Many AddModule subcommands exist. These subcommands are used by specifying the <module name> followed by the subcommand.

Parameters:

AddModule

Parameter Name	Parameter Type	Required?
name	Identifier	Yes
clock	Identifier	No (default=1x)
htIdW	Integer	No (default=0)
reset	Identifier	No (default=””)
pause	Boolean	No (default=false)

The **name** parameter specifies the name of the module. Module names must be unique.

The **clock** parameter specifies the clock rate for the module. Valid options are 1x and 2x.

The **htIdW** parameter specifies the number of threads (log2) in the module.

The **reset** parameter specifies that a reset sequence of instructions is to be initiated when the unit is dispatched from the host. The reset entry instruction is specified as the argument to the reset parameter. The execution of the reset sequence is stopped with *HtTerminate()*.

The **pause** parameter specifies the module requires pause and resume functionality.

Example:

```
dsnInfo. AddModule( name=inc );
```

Generated Personality Interface:

The ***htl*** application generates routines to simplify next instruction processing. The following routines are provided:

HtContinue(inst)

The *HtContinue* routine is used to set the next instruction to execute.

HtRetry()

The *HtRetry* routine is used to indicate that the current instruction should be re-executed as the next instruction. The performance monitor counts the number of times the HtRetry routine is executed to provide visibility of busy resources.

HtTerminate()

The *HtTerminate* routine is used to terminate the currently executing thread. This routine should only be used to terminate a module's reset thread (a thread that begins executing out of reset).

HtPause(inst)

The *HtPause* routine is used to pause the currently executing thread. The thread is paused until the *HtResume* routine is executed, to resume executing the thread at the instruction specified.

HtResume(thread id)

The *HtResume* routine is used to resume execution of the indicated thread, paused with the *HtPause* routine. If the ***htd*** *htIdW* is not specified *HtResume* has no arguments.

Generated Host or Model Interface:

No host or model interface routines are generated.

<mod>.AddInst (…)

<mod>.AddInst (name=<name>)

Description:

Specifies that the module with <mod> name has an instruction with the specified name. The instructions are defined with #ifdef's in the Pers***Unit***Common.h file. The **htl** tool determines the maximum value for all instructions within a module and generates a #ifdef in the Pers***Unit****Module*.h file that specifies the width of variables that hold the instruction. An example would be: #define CTL_INST_W 4. This implies there are less than 2**4 or 16 instructions in module ctl.

Parameters:

AddInst(…)

Parameter Name	Parameter Type	Required?
name	Identifier	Yes

The **name** parameter specifies the name of an instruction for module < mod>.

Example:

```
ctl.AddInst( name=CS_PARSE );
```

Generated Interface:

No routines are generated for the *AddInst* command.

<mod>.AddEntry(…)

<mod>.AddEntry(func=<name>, inst=<inst> {, reserve=<number>} {, host=<true or false>})

Description:

Adds a function call entry point for a module. Optionally, the entry point can be designated as the entry point for a host call routine. Typically a module has a single entry point, but a module may have multiple entry points or no entry points. A module containing no entry points, also contains no instructions and has no threads allocated to it. An example application of a module with no entry points is a module containing a global RAM to be used by other modules.

Parameters:

AddEntry(…)

Parameter Name	Parameter Type	Required?
func	Identifier	Yes
inst	Identifier	Yes
reserve	Integer	No (default=0)
host	Boolean	No (default=false)

The **func** parameter specifies name of the HT function. The function name is used by callers to specify the HT function being called. If *host* is true (host entry point) *func* must be *htmain*.

The **inst** parameter specifies the entry point instruction. The named instruction will be the first instruction executed for a function call.

The **reserve** parameter specifies the number of threads reserved for the HT function. This parameter is used to avoid lockouts, in cases where a module has multiple entry points. The reserved threads are taken from the threads allocated to the module (htIdW in the AddModule command.

The **host** parameter specifies that the entry point is for a host entry point. A single entry point for a unit can be designated as the host entry point. The *func* parameter for a host entry point is *htmain*.

AddParam subcommand:

The *AddParam* subcommand is used to specify an entry point parameter. Entry point parameters must have a corresponding private variable with the same type. Entry parameters are written to the corresponding private variable prior to executing the first instruction of the function.

Parameter Name	Parameter Type	Required?
hostType	Identifier	No (default=<type>)
type	Identifier	Yes
name	Identifier	Yes
unused	Boolean	No (default=false)

The **hostType** parameter specifies the host type of the entry parameter. The *hostType* is used when generating the call/return interfaces used by the host code. If the *hostType* is not specified then the host code will use the *type* parameter as the parameter type.

The **type** parameter specifies the type of the entry parameter. The function must have a private variable with the same *type*, unless the *unused* parameter is *true*. The **name** parameter specifies the name of the entry parameter. The function must have a private variable with the same *name*, unless the *unused* parameter is *true*. Note: If the reentry parameter is a structure, the structure must have no unused bytes within the structure caused by variable alignment.

The **unused** parameter specifies that the *AddParam* parameter is not read by the called function. This occurs for parameters passed by reference to allow the function to pass back values on the associated return. The *unused* parameter is used to inform the htl application that the calling interface does not need to provide data path for the variable that is not read.

Example:

```
prs.AddEntry( func=parse, inst=PRS_INIT )
        .AddParam( type=unit32_t, name=arg1 )
        .AddParam( type=int5_t, name=arg2 )
        :
prs.AddEntry( func=free, inst=PRS_FREE, reserve=8);
```

The examples provide two entry points into module *prs*. The first entry point is named *parse* and begins execution with instruction *PRS_INIT*. The entry point has two associated parameters from the calling function, *arg1* and *arg2*.

The second example provides an entry point named *free* that begins execution at instruction *PRS_FREE*. Eight threads module's threads are reserved for the HT function *free*.

Generated Personality Interface:

No personality routines are generated for the *AddEntry* command. Refer to the *AddCall* and *AddReturn* commands for personality interface routines.

Generated Host Interface:

The *AddEntry* command generates the following Host interface routine when the *host* parameter is set to true.

bool SendCall_*htmain* (<callParam1>, <callParam2>, …)

The *SendCall* routine begins a thread in the called module. The thread in the host routine continues execution. The call parameters are written to the callee's thread private variables. The *SendCall* routine parameters correspond to the *AddParam* subcommands for the called entry point. The type for the call parameters will be from the *AddParam*'s *hostType* parameter if present; otherwise it will be from the *type* parameter.

Generated High Level Model Interface:

The *AddEntry* command generates the following high level model interface routine when the *host* parameter is set to true.

bool RecvCall_*htmain* (<callParam1>, <callParam2>, …)

The *RecvCall* routine allows the high level model to determine that a call was issued by the host with the provided parameters. The *RecvCall* routine parameters correspond to the *AddParam* subcommands for the called entry point. The type for the call parameters will be from the *AddParam*'s *hostType* parameter if present; otherwise it will be from the *type* parameter.

<mod>.AddReturn(…)

<mod>.AddReturn(func=<name>)

Description:

Adds return linkage for the entry point with function name ***func***.

Parameters:

AddReturn(…)

Parameter Name	Parameter Type	Required?
func	Identifier	Yes

The **func** parameter specifies name of the HT function associated with the return. When the return is to the host the *func* parameter is *htmain*.

AddParam subcommand:

The *AddParam* subcommand is used to specify a return parameter.

Parameter Name	Parameter Type	Required?
hostType	Identifier	No (default=<type>)
type	Identifier	Yes
name	Identifier	Yes
unused	Boolean	No (default=false)

The **hostType** parameter specifies the host type of the return parameter. The *hostType* is used when generating the call/return interfaces used by the host code. This parameter is required when the return is to the host and the *type* parameter is a non-standard type.

The **type** parameter specifies the type of the return parameter. The calling function must have a private variable with the same *name* and *type*. Note: If the return parameter is a structure, the structure must have no unused bytes within the structure caused by variable alignment.

The **name** parameter specifies the name of the return parameter. The calling function must have a private variable with the same *name* and *type*.

The **unused** parameter specifies that the *AddParam* parameter is not written by the called function. This occurs for parameters passed by reference where the parameter is read but not written. The *unused* parameter is used to inform the ***htl*** application that the returning interface does not need to provide data path for the variable that is not written.

Example:

```
prs.AddReturn( func=parse );
prs.AddReturn( func=free )
        .AddParam( type=bool, name=failure )
        ;
```

The examples provide two returns from module *prs*. The first return is named *parse*, the second is from *free*. A *bool* named *failure* is returned from *free* to a private variable of the same name (*failure*) in the calling function.
An entry point can have at most one associated return within a single module.

Generated Personality Interface:

The ***htl*** application generates routines to simplify issuing function returns. The following routines are provided:

SendReturnBusy_*htfunc* ()

The *SendReturnBusy* routine indicates whether the return interface has space to accept another return thread. The *SendReturnBusy* routine should be followed with a conditional call to the *HtRetry()* routine.

Example usage:

```
If (SendReturnBusy_free() ) {
        HtRetry();
        Break;
}
```

SendReturn_*htfunc* (<rtnParam1>, <rtnParam2>, …)

The *SendReturn* routine resumes a thread in the original calling module. Return parameters are written to the caller's thread private variables. Note that a *SendReturnBusy* routine must be called prior to executing the *SendReturn* routine (a runtime SystemC check ensures that the *SendReturnBusy* routine has been called). The private thread state associated with the thread issuing the return is released (and available to future calls to the module).

Example usage:

```
SendReturn_free ( false );        // the caller's private variable failure will have the value false
```

Note: Returns are not supported within a switch statement.

Generated Host Interface:

The *AddReturn* command generates the following Host interface routine when the name of the function (i.e. *func* parameter value) matches an *AddEntry* function name that also has the *AddEntry's host* parameter is set to true.

bool RecvReturn_*htmain*(<rtnParam1>, <rtnParam2>, …)

The *RecvReturn* routine allows the host to determine that a call was returned from a module with the provided parameters. The *RecvReturn* routine parameters correspond to the *AddParam* subcommands for the *AddReturn* command. The *type* for the return parameters will be from the *AddParam*'s *hostType* parameter if present; otherwise it will be from the *type* parameter.

Generated High Level Model Interface:

The *AddReturn* command generates the following high level model interface routine when the name of the function (i.e. *func* parameter value) matches an *AddEntry* function name that also has the *AddEntry*'s *host* parameter is set to true.

bool SendReturn_*htmain* (<rtnParam1>, <rtnParam2>, …)

The *SendReturn* routine allows the high level model to mimic the behavior of a personality module performing a *SendReturn* interface routine. The *SendReturn* routine parameters correspond to the *AddParam* subcommands for the *AddReturn* command. The *type* for the return parameters will be from the *AddParam*'s *hostType* parameter if present; otherwise it will be from the *type* parameter.

<mod>.AddCall(…)

<mod>.AddCall(func=<name> {, fork=<boolean>} {, queueW=<depth>})

Description:

Adds call linkage from the caller's module to the module with the specified function. The called module has a queue of depth *queueW* (log2) entries to accept calls even when the called module has no available threads.

Parameters:

AddCall(…)

Parameter Name	Parameter Type	Required?
func	Identifier	Yes
fork	Boolean	No (default=false)
queueW	Integer	No (default=5)

The **func** parameter specifies name of the HT function to be called.

The **fork** parameter specifies that the call will be issued asynchronously. The calling thread will continue execution and a new thread will begin execution in the destination module.

The **queueW** parameter specifies the depth of the queue in the caller's module to accept calls even when no caller module threads are available.

Example:

```
ctl.AddCall( func=parse );
ctl.AddCall( func=free, fork=true, queueW=0);
```

The examples provide two calls from module *ctl*. The first call is to function named *parse,* the second is to a function named *free*. The second call specifies that the calling function does not require a queue when the caller is unavailable (i.e. the caller will wait until the caller can start the thread).

Generated Personality Interface:

The **htl** application generates routines to simplify issuing function calls. The following routines are provided:

SendCallBusy_*htfunc* ()

The *SendCallBusy* routine returns a boolean value indicating if the call interface has space to accept another call thread. A call to the *SendCallBusy* routine should be followed by a conditional call to the *HtRetry()* routine.

Example usage:

```
If (SendCallBusy_free () ) {
        HtRetry();
        Break;
}
```

SendCall_*htfunc* (rsmInst, <callParam1>, <callParam2>, …)

The *SendCall* routine begins a thread in the called module. The thread in the calling module pauses execution until a corresponding return is executed. The *rsmInst* parameter specifies the instruction to be executed after the call returns. The call parameters are written to the callee's thread private variables. Note that a *SendCallBusy* routine must be executed (with *HtRetry()*) prior to executing the *SendCall* routine (a runtime SystemC check ensures that it has been called). The *SendCall* routine parameters correspond to the *AddParam* subcommands for the called entry point.

Example usage:
SendCall_free (head, tail); // the callee's private variables will have the values for head and tail

SendCallFork_*htfunc* (rsmInst, < callParam1>, <callParam2>, …)

The *SendCallFork* routine begins a thread in the called module. Note that the thread in the calling module continues execution. The *rsmInst* parameter specifies the instruction to be executed after the call returns. This instruction must execute the *RecvReturnJoin* routine. The call parameters are written to the callee's thread private variables. Note that a *SendCallBusy* routine must be executed (with *HtRetry()*) prior to executing the *SendCallFork* routine (a runtime SystemC check ensure that it has been called). The *SendCallFork* routine call parameters correspond to the *AddParam* subcommands for the called entry point. The *SendCallFork* routine is only provided if the *AddCall*'s *fork* parameter is specified.

Example usage:
SendCallFork_free (JOIN, head, tail); // the callee's private variables will be set with head and tail. When the thread returns the JOIN instruction will be executed. The JOIN instruction must execute the *RecvReturnJoin* routine.

RecvReturnPause_*htfunc* (rsmInst)

The *RecvReturnPause* routine pauses the current thread if any forked calls have not returned. The *rsmInst* parameter specifies the instruction to be executed after all asynchronous calls have returned.

RecvReturnJoin_*htfunc* ()

The *RecvReturnJoin* routine must be executed by the very first instruction for the returning asynchronous call threads in order for the pause mechanism to work properly.

Generated Host or Model Interface:

No host or model interface is generated.

<mod>.AddTransfer(…)

<mod>.AddTransfer(func=<ht_func> {, queueW=<depth>})

Description:

Adds transfer linkage from the source module to the module with the specified HT function. The destination module has a queue of depth *queueW* (log2) entries to accept transfers even when the destination module has no available threads.

Parameters:

AddTransfer(…)

Parameter Name	Parameter Type	Required?
func	Identifier	Yes
queueW	Integer	No (default=5)

The **func** parameter specifies name of the target HT function.

The **queueW** parameter specifies the depth of the queue in the destination module to accept transfers even when no destination module threads are available (log2).

Example:

```
ctl.AddTransfer( func=parse );
ctl.AddTransfer( func=free, queueW=0);
```

The examples provide two transfers from module *ctl*. The first transfer is to an entry named *parse,* the second is to an entry named *free*. The second transfer specifies that the destination HT function does not require a queue when the destination is unavailable (i.e. the transfer will wait until the destination can start the thread).

Generated Personality Interface:

The **htl** application generates routines to simplify issuing transfers. The following routines are provided:

SendTransferBusy_*htfunc* ()

The *SendTransferBusy* routine returns a boolean value indicating if the transfer interface has space to accept another transfer thread. The *SendTransferBusy* routine is followed by a conditional call to the *HtRetry*()routine.

Example usage:

```
If (SendTransferBusy_free () ) {
```

```
            HtRetry();
            Break;
    }
```

SendTransfer_*htfunc* (<xferParam1>, <xferParam2>, …)

The *SendTransfer* routine begins a thread in the destination module. The thread in the source module terminates and releases its thread state. The transfer parameters are written to the destination thread's private variables. Note that a *SendTransferBusy* routine must be called (with conditional *HtRetry()*) prior to executing the *SendTransfer* routine (a runtime SystemC check ensure that it has been called). The *SendTransfer* routine parameters correspond to the *AddParam* subcommands for the destination module's entry point.

Example usage:

```
SendTransfer_free ( head, tail );
```

Generated Host or Model Interface:

No host or model interface is generated.

<mod>.AddPrivate( )

<mod>.AddPrivate( )

Description:

Specifies that module <*mod*> has one or more thread private variables. Private variables can be scalar or memory variables.

Subcommand AddVar(…)

Parameter Name	Parameter Type	Required?
type	Identifier	Yes
name	Identifier	Yes
dimen1	Integer	No (default=””)
dimen2	Integer	No (default=””)
addr1W	Integer	No(default=””)
addr2W	Integer	No (default=””)
addr1	Identifier	No (default=””)
addr2	Identifier	No (default=””)

The **type** parameter specifies the type for the private variable.

The **name** parameter specifies the name of the private variable.

The **dimen1** parameter specifies the size of the first dimension for the private variable.

The **dimen2** parameter specifies the size of the second dimension for the private variable.

The **addr1W** parameter specifies the width of the first address for a private memory variable.

The **addr2W** parameter specifies the width of the second address for a private memory variable.

The **addr1** parameter specifies a private scalar variable within the local module that is to be used as the first address for reading the private memory variable.

The **addr2** parameter specifies a private scalar variable within the local module that is to be used as the second address for reading the private memory variable.

Example:

```
idf.AddPrivate( )
        .AddVar( type=uint32_t, name=qid0 )
        .AddVar( type=MchQid_t, name=mchQid )
```

;

Generated Personality Interface:

The **htl** application generates macros to simplify private variable access. The following may be arguments in the private variable macros:

varIdx1 – The parameter *varIdx1* specifies the index for the destination variable's first dimension. This parameter is only present if the destination variable has a first dimension.

varIdx2 – The parameter *varIdx2* specifies the index for the destination variable's second dimension. This parameter is only present if the destination variable has both a first and second dimension.

varaddr1 – The parameter *varaddr1* specifies the first address for writing the private memory variable.

varaddr2 - The parameter *varaddr2* the second address for writing the private memory variable.

The following macros are provided:

If the *AddStage()* command has not been provided for a module then a single clock stage for instruction execution is generated. Within the single stage, private variables can be accessed by the following macros:

Private variable read:
data = P_*name* { [varidx1] { [varidx2] }};

Private variable write:
P_*name*{ { [varaddr1] {,[varaddr2] }} {, [varidx1] {, [varidx2] }} }= data;

The *P_name* macro is used to access private variables. The type of the accessed data may be a simple type, a structure or union.

An additional macro is provided to allow direct access to the private variable registered value. Access to the registered version of the variable is sometimes required to allow a Xilinx synthesis directive to be attached to a register. The macro is: *PR_name{[varidx1] {[varidx2] }}*. Note that the *PR_name* version of the variable should only be read.

When the *AddStage()* command has been provided for a module then multiple clock stages may be required to execute instructions. Within each instruction execution stage, the private variables can be read or written. The provided macros include a stage number as part of the name to indicate which macro should be used.

Private variable read:

data = P#_*name* { [varidx1] { [varidx2] }};

Private variable write:

P#_*name*{ { [varaddr1] {,[varaddr2] }} { [varidx1] {, [varidx2] }} }= data;

The # symbol in the macro names is replaced with the instruction execution stage number starting with a '1' for the first stage. Macros are generated for all stages starting with the first stage to the last stage specified by the *privWrStg* parameter of the *AddStage()* command.

The registered version of the macros are provided as PR#_*name* and are used similarly to the PR_*name* versions described in the non-staged section above.

Note: For reads, the private variables specified in *addr1* and *addr2* of the *AddVar()* subcommand contain the *varaddr1* and *varaddr2*. The private variable address must be valid on entry to the instruction (i.e. assigned in a previous instruction for the thread or parameters assigned on entry by a call or return).

Generated Host or Model Interface:

No host or model interface is generated.

<mod>.AddShared()

<mod>.AddShared()

Description:

Specifies that the module <mod> has shared variables. All threads in the module can access the shared variables. Shared variables can be scalar, memory or queue.

Parameters:

AddShared() – No parameters are required for the command.

Subcommand AddVar(…)

Parameter Name	Parameter Type	Required?
type	Identifier	Yes
name	Identifier	Yes
addr1W	Integer	No(default="")
addr2W	Integer	No (default="")
queueW	Integer	No (default="")
dimen1	Integer	No (default="")
dimen2	Integer	No (default="")
blockRam	Boolean	No (default=false)
reset	Boolean	No (default=false)

The **type** parameter specifies the type for the shared variable. The *type* can be a simple type, a structure or a union.

The **name** parameter specifies the name of the shared variable.

The **addr1W** parameter specifies the width of the first address for a shared memory variable.

The **addr2W** parameter specifies the width of the second address for a shared memory variable.

The **queueW** parameter specifies the number of elements for a shared queue variable (log2).

The **dimen1** parameter specifies the size of the first dimension for the shared variable.

The **dimen2** parameter specifies the size of the second dimension for the shared variable.

The **blockRam** parameter specifies the type of Xilinx FPGA memory to be used for a shared memory variable. The two types are Block Ram, and Distributed Ram. Block rams use

dedicated FPGA resources that function as 36Kbit rams. These resources can only be used as memory, but there are a limited number of them on an FPGA. The second type of memory, called Distributed Ram, leverages the logic lookup table resources as small 64-bit rams. These logic lookup tables are distributed across the entire FPGA. Typically, a shared memory variable can fit in a single or a small number of block rams. Conversely, a shared memory variable would typically require many distributed rams to provided enough memory bits for the variable.

The **reset** parameter specifies that the shared variable is reset on a hardware reset. Note: The reset parameter can be specified as a parameter of the AddShare command, resulting in all shared variables being reset.

Note that the *addr1W* and *addr2W* parameters are specified for memory variables, and the *queueW* parameter is specified for queue variables. The *dimen1* and *dimen2* parameters can be specified for all shared variables.

Example:

```
idf.AddShared()
        .AddVar( type=uint32_t, name=qid0 )
        .AddVar( type=MchQid_t, name=mchQid )
        ;
```

Generated Personality Interface:

The **htl** application generates macros to simplify shared variable access. The following may be arguments in the shared variable macros:

varIdx1 – The parameter *varIdx1* specifies the index for the destination variable's first dimension. This parameter is only present if the destination variable has a first dimension.

varIdx2 – The parameter *varIdx2* specifies the index for the destination variable's second dimension. This parameter is only present if the destination variable has both a first and second dimension.

varaddr1 – The parameter *varaddr1* specifies the first address for writing the shared memory variable.

varaddr2 - The parameter *varaddr2* the second address for writing the shared memory variable.

The following macros are provided:

Shared Scalar Variables:

Shared variable read:

data = S_*name* { [varidx1] { [varidx2] }};

Shared variable write:

S_*name*{ [varidx1] { [varidx2] }} = data;

The S_*name* macro is used to access a shared scalar and arrays of scalar variables. The type of the accessed data may be a simple type, a structure or union.

An additional macro is provided to allow direct access to the shared variable registered value. Access to the registered version of the variable is sometimes required to allow a Xilinx synthesis directive to be attached to a register. The macro is: SR_name{[idx1] {[idx2] }}. Note that the SR_name version of the variable should only be read.

Shared Memory Variables:

Shared variable read:

S_*name* { [varidx1] { [varidx2] }}.read_addr(varaddr1 {, varaddr2 });
data = S_*name* { [varidx1] { [varidx2] }}.read_mem();

Reading a shared memory variable requires a two-step sequence. The first step is to specify the address of the memory to be read using the *.read_addr( )* method. The second step is to access the memory to obtain the data value. Note that the two steps can be separated at different locations within the instruction being executed. A run time check is performed by the *.read_mem()* method to ensure that a *.read_addr()* method was previously called.

Note: Calling the *.read_addr()* method only sets the address of the memory where a read may happen. The *.read_mem()* method can be optionally called based on conditional instruction execution. It is frequently advantageous from a logic timing perspective to hoist the *.read_addr()* call out of conditional statements.

Note: Shared memories implemented as block rams must have the *.read_addr()* method called on the previous instruction stage as the *.read_mem()* (i.e. previous clock cycle) because block rams register the input address. Shared memories implemented as distributed rams must have the *.read_addr()* method called on the same instruction stage as the *.read_mem()* method because the input address is not registered.

Shared variable write:

S_*name* { [varidx1] { [varidx2] }}.write_addr(varaddr1 {, varaddr2 }); // block or distributed ram

S_*name*{ [varidx1] { [varidx2] }}.write_mem(data); // block or distributed ram
S_*name*{ [varidx1] { [varidx2] }}.write_mem( ) = data; // distributed ram only
S_*name*{ [varidx1] { [varidx2] }}.write_mem( ).m_*field* = data; // distributed ram only

Writing a shared memory variable requires a two-step sequence. The first step is to specify the address of the memory to be written using the *.write_addr( )* method. The second step is to write the data value to the memory variable. Note that the two steps can be separated at different locations within the instruction being executed. A run time check is performed by the *.write_mem()* method to ensure that a *.write_addr()* method was previously called.

Note: The *.write_addr()* method must be called on the same instruction stage (same clock cycle) as the call to *.write_mem()* for both distributed and block rams.

Shared Queue Variables:

Shared variable push:

bool full = S_*name* { [varidx1] { [varidx2] }}.full();
S_*name* { [varidx1] { [varidx2] }}.push(data);

Two routines are provided for pushing a value onto a shared variable queue. The first routine checks for available space in the queue, returning true if the queue is full. The second method pushes data on the queue. A runtime check verifies that space is available when a .push() routine is executed.

Shared variable pop:

bool empty = S_ *name* { [varidx1] { [varidx2] }}.empty();
data = S_*name* { [varidx1] { [varidx2] }}.front();
S_*name* { [varidx1] { [varidx2] }}.pop();

Three routines are provided for popping a value from a shared variable queue. The first routine checks whether the queue is empty and return true if it is empty. The second routine accesses the data at the front of the queue. The third routine pops the front item off the queue.

Note that it is okay to access the front of the queue even when empty. However, calling .pop() when the queue is empty will result in a run time error.

Generated Host or Model Interface:

No host or model interface is generated.

<mod>.AddGlobal(…)

<mod>.AddGlobal(var=<name> {, addr1W=<width> {, addr2W=<width>}} {, addr1=<name> {, addr2=<name>}}{, dimen1=<size> {, dimen2=<size> }} {, extern=<bool> }{, rdStg=<num>} {, wrStg=<num>})

Description:

Specifies that the module <mod> requires a global variable. All threads within module *<mod>* have access to the global variable. Additionally, other modules can access the global variable by using the *AddGlobal()* command for the other module(s) with the extern=true parameter. Global variables can be a scalar or a memory. The parameters *addr1* and *addr2* are the names of private or temporary stage variables that are to be used as the address variable for global memory variable reads. A global variable can be modified by up to 2 writers.

Parameters:

AddGlobal(…)

Parameter Name	Parameter Type	Required?
var	Identifier	Yes
addr1W	Integer	No(default="")
addr2W	Integer	No (default="")
dimen1	Integer	No (default="")
dimen2	Integer	No (default="")
addr1	Identifier	No (default="")
addr2	Identifier	No (default="")
extern	Boolean	No (default=false)
rdStg	Integer	No (default=1)
wrStg	Integer	No (default=see below)

The **var** parameter specifies the name of the global variable.

The **addr1W** parameter specifies the width of the first address for a global memory variable.

The **addr2W** parameter specifies the width of the second address for a global memory variable.

The **dimen1** parameter specifies the size of the first dimension for the global variable.

The **dimen2** parameter specifies the size of the second dimension for the global variable.

The **addr1** parameter specifies a private or stage variable within the local module that is to be used as the first address for reading the global memory variable. Note: The variable used must have a unique name. Note: This parameter is only needed when the **read** parameter in the **AddField** subcommand is **true.**

The **addr2** parameter specifies a private stage variable within the local module that is to be used as the second address for reading the global memory variable. Note: The variable used must have a unique name. . Note: This parameter is only needed when the **read** parameter in the **AddField** subcommand is **true.**

The **extern** parameter specifies that the named global variable is located external to the local module.

The **rdStg** parameter specifies the stage within the instruction pipeline that the global variable read value will be used.

The **wrStg** parameter specifies the stage within the instruction pipeline that the global variable write is executed. The default is the **privWrStg** parameter specified in the **AddStage** command.

Subcommand AddField(…)

Parameter Name	Parameter Type	Required?
type	Identifier	Yes
name	Identifier	No (default="")
dimen1	Integer	No (default="")
dimen2	Integer	No (default="")
read	Boolean	No (default=false)
write	Boolean	No (default=false)

The **type** parameter specifies the type for the field.

The **name** parameter specifies the name of the field.

The **dimen1** parameter specifies the size of the first dimension for the field.

The **dimen2** parameter specifies the size of the second dimension for the field.

The **read** parameter specifies that the current module requires reading the field. Note: This parameter results in the generation of the **GR*_var_field()*** macro. If the global variable is the source of data for a memory write, this parameter is false.

The **write** parameter specifies that the current module requires writing the field. This parameter results in the generation of the **GW*_var_field()*** macro. If the global variable is the destination of data from a memory read, this parameter is false.

Example:

```
idf.AddGlobal( var=strSetL, addr1W=9, vdimen1=LINE_STR_CNT_L, addr1=strSetL_addr )
        .AddField( type=uint32_t, name=qid0, read=true, write=false  )
        . AddField( type=MchQid_t, name=mchQid, read=true, write=true )
        ;
```

Generated Personality Interface:

The **htl** application generates macros to simplify global variables. The following are arguments in the global variable macros.

varIdx1 – The parameter *varIdx1* specifies the index for the variable's first dimension. This parameter is only present if the variable has a first dimension.

varIdx2 – The parameter *varIdx2* specifies the index for the variable's second dimension. This parameter is only present if the variable has both a first and second dimension.

fldIdx1 – The parameter *fldIdx1* specifies the first index for the field. This parameter is only present if the field has a first dimension.

fldIdx2 – The parameter *fldIdx2* specifies the second index for the field. This parameter is only present if the field has both first and second dimensions.

varaddr1 – The parameter *addr1* specifies a first address for writing the global memory variable.

varaddr2 - The parameter *addr2* specifies the second address for writing the global memory variable.

Note: For reads the private variable address (*addr1* and *addr2* in the ***htd*** *AddVar* subcommand) must be valid on entry to the instruction (i.e. assigned in a previous instruction for the thread or parameters assigned on entry by a call or return). If a temporary variable is used it must be valid on the previous cycle.

The following macros are provided:

If the *AddStage()* command has not been provided for a module then a single clock stage for instruction execution is generated. Within the single stage, global variables can be accessed by the following macros:

Global variable read:

data = GR_*var_field(* { varidx1 {, varidx2 }} {, fldidx1 {, fldidx2 }});

The **GR_*var_field*** macro is used to read a global scalar and memory variables. The type of the accessed data may be a simple type, a structure or union. The address

for global memory reads is obtained from the private variable specified *AddGlobal()* command. Note that the private variable address must be valid on entry to the instruction (i.e. assigned in the previous instruction for the thread).

Global variable write:

GW*_var_field* ({, varaddr1 {, varaddr2 }} {, varidx1 {, varidx2 }} {, fldidx1 {, fldidx2 }}, data);

When the *AddStage()* command has been provided for a module then multiple clock stages may be required to execute instructions. The *AddGlobal()* command's *rdStg* parameter specifies the specific instruction stage where the global variable will be read.

Global variable read:

data = GR#*_var_field(* { varidx1 {, varidx2 }} {, fldidx1 {, fldidx2 }});

The # symbol in the macro names is replaced with the instruction execution stage number starting with a '1' for the first stage. The macro is generated for the specific stage specified by the *rdStg* parameter of the *AddGlobal()* command.

Global variable write:

GW#*_var_field* ({, varaddr1 {, varaddr2 }} {, varidx1 {, varidx2 }} {, fldidx1 {, fldidx2 }}, data);

The # symbol in the macro names is replaced with the instruction execution stage number starting with a '1' for the first stage. The macro is generated for the specific stage specified by the *wrStg* parameter of the *AddGlobal()* command.

Note: For reads, the *varaddr1* and *varaddr2* values are obtained from the private or temporary stage variable specified in the *AddVar()* ***htd*** subcommand. The private variable address must be valid on entry to the instruction (i.e. assigned in a previous instruction for the thread or parameters assigned on entry by a call or return). If a temporary variable is used it must be valid on the previous cycle.

Generated Host or Model Interface:

No host or model interface is generated.

<mod>.AddStage(…)

<mod>.AddStage({privWrStg=<stage>} {, execStg=<stage>})

Description:

Specifies that the instructions within the module require greater than a single register stage to execute. The command specifies the stage at which private variables are saved and memory and thread control operations execute. The length of the instruction pipeline is dictated by the greater of the private write stage and the execution stage.

Parameters:

AddStage (…)

Parameter Name	Parameter Type	Required?
privWrStg	Integer	No (default =execStg)
execStg	Integer	No (default=1)

The **privWrStg** parameter specifies the instruction execution stage that private variables are saved.

The **execStg** parameter specifies the instruction execution stage that memory operations are issued and thread control operations execute. All memory read and write interface routines are executed using variables associated with the specified instruction stage Thread execution operations include *HtContinue*, *HtRetry*, call and return.

Subcommand AddVar(…)

Typically intermediate data values must be staged between instruction execution stages when additional instruction execution stages are required. The *AddVar* command is used to declare these temporary variables. Only scalar temporary variables are supported.

Parameter Name	Parameter Type	Required?
type	Identifier	Yes
name	Identifier	Yes
range	Integer Range	Yes
dimen1	Integer	No (default=””)
dimen2	Integer	No (default=””)
init	boolean	No (default=false)
primOut	boolean	No (default=false)
conn	boolean	No (default=true)
reset	boolean	No (default=false)

The **type** parameter specifies the type for the stage variable.

The **name** parameter specifies the name of the stage variable.

The **range** parameter specifies the instruction stages where the staging variables are required.

The **dimen1** parameter specifies the size of the first dimension for the stage variable.

The **dimen2** parameter specifies the size of the second dimension for the stage variable.

The **init** parameter specifies that the first stage variable should be initialized from a private variable with the same name.

The **primOut** parameter specifies that the variable is an output of a primitive.

The **conn** parameter specifies that the successive stage variables should be connected as a pipelined.

The **reset** parameter specifies that the stage variables are reset on a hardware reset.

Example:

```
inc.AddStage(execStg=6)
        .AddVar( type=LineStrCntR_t, name=lineStrCntR, range=1-6, init=true )
        .AddVar( type=ht_uint4_t, name=strLenRa, dimen1=STR_CNT_L,
                dimen2=STR_CNT_R, range=3-6, conn=false )
;
```

The example specifies that two strings of staging registers are needed. The first string of registers is named lineStrCntR with type LineStrCntR_t. The string of staging register consists of six registers. The first register of the string is initialized from a private variable with the same name from the first instruction stage. The second stage variable gets its value from the first stage variable, the third from the second, etc.

The second string of staging registers is named strLenRa with type ht_uint4_t. The string of staging registers consists of four registers. The instruction provides the input to each stage register.

Private variables are updated and thread control operations are executed in stage 6.

Generated Personality Interface:

The **htl** application generates macros to simplify stage variable access. The following macros are provided:

Temporary stage variable read:

data = T#_*name* { [idx1] { [idx2] }};

Temporary stage variable write:

T#_*name*{ [idx1] { [idx2] }} = data;

The T#_*name* macro is used to access a shared scalar and arrays of scalar variables. The type of the accessed data may be a simple type, a structure or union.

An additional macro is provided to allow direct access to the stage variable register. Access to the registered version of the variable is sometimes required to allow a Xilinx synthesis directive to be attached to a register. The macro is: TR#_name{[idx1] {[idx2] }}. Note that the TR#_name version of the variable should only be read.

Generated Host or Model Interface:

No host or model interface is generated.

<mod>.AddReadMem(…)

<mod>.AddReadMem ({ queueW=<depth> }{, rspGrpId=<name>} {, rspCntW=<width> } {, maxBw=<bool>} {, pause=<bool>} {, poll=<bool>})

Description:

Adds a memory read interface to the associated module. The destination for a memory read must be a variable that is up to 8 bytes wide. The destination variable may be indexed (array or field indexing), allowing up to eight indexed locations to be read from memory with a single read operation.

Parameters:

AddMemRead(…)

Parameter Name	Parameter Type	Required?
queueW	Integer	No (default=5)
rspGrpId	Identifier	No (default="")
rspCntW	Integer	No (default=6, 9-htIdW))
maxBw	Boolean	No (default=false)
pause	Boolean	No (default=true)
poll	Boolean	No (default=false)

The **queueW** parameter specifies the depth (log2) of the queue in which memory request are written. Memory requests are subsequently read from the queue and sent to memory.

The **rspGrpId** parameter specifies whether response groups are associated with the individual threads, or explicitly specified via a private or shared variable. If the *rspGrpId* parameter is not specified then the response groups are associated with individual threads, otherwise the specified variable (shared or private) is used to identify the response group.

The **rspCntW** parameter specifies the number of outstanding requests that can be active for each response group. The *ReadMemBusy()* routine will return true if called when the maximum number of outstanding requests for the specified response group is reached. The default is 6 or based on the number of threads (9-HtIdW), whichever is higher.

The **maxBw** parameter specifies that the maximum amount of memory read bandwidth is required. If a read request is attempted every instruction and *maxBw* is set to false, then 25% of the instructions will result in *ReadMemBusy()* returning true. However, for the same scenario, if *maxBw* is true then only 6% of instructions will result in *ReadMemBusy()* returning true. The *ReadMemBusy()* being true is the result of a conflict between instruction issued memory requests and memory responses.

The ***pause*** parameter specifies that the module requires the capability to pause a thread or thread group, when waiting for read requests to be completed. When the *pause* parameter is true, then the *ReadMemPause()* routine is generated. The default value for

the *pause* parameter is *true*, if the *poll* parameter is *false*. If the *poll* parameter is *true* the *pause* parameter defaults to *false*.

The ***poll*** parameter specifies that the module requires the capability for a thread or thread group to wait for reads to complete by polling. When the *poll* parameter is true, then the ReadMemPoll() routine is generated. The default value for the *poll* parameter is false.

Subcommand AddDst(…)

Parameter Name	Parameter Type	Required?
var	Identifier	Yes
name	Identifier	No (default=<var>)
field	Identifier	No (default="")
dataLsb	Integer	No (default=0)
multiRd	Boolean	No (default=false)
dstIdx	Identifier	No (default="")
rdType	Identifier	No (default="")
memSrc	Identifier	No (default=coproc)
atomic	Identifier	No (default=read)

The **name** parameter specifies the name used when constructing the *ReadMem_name* function. Note that if the *name* parameter is not provided then the *var* parameter is used for the function name.

The **var** parameter specifies the name of a shared or global variable that is to be the destination of the memory read.

The **field** parameter specifies the field of the destination variable where the read data will be written. Global variables require the field parameter to be present. Shared variables only require the parameter if the shared variable is a structure or union.

The **dataLsb** parameter specifies the bit within the returning data that is associated with the least significant bit of the shared or global variable.

The **multiRd** parameter specifies that the ability to request multiple 8-byte quantities with a single read request is required. The *dstIdx* parameter is a required parameter when *multiRd* is specified. Multi-word reads are more efficient for platforms that support accesses to memory that are larger than 64-bits. For platforms that do not support the multi-word write then individual requests are issued for each 64-bit word.

The **dstIdx** parameter specifies the index of the destination variable that is used for storing the response data of multi-quadword memory reads. The accepted values for the parameter are: *varAddr1, varAddr2, varIdx1, varIdx2, fldIdx1 or fldIdx2*. The first read response is written to the destination variable specified by the *ReadMem_name(…)* routine. Subsequent responses are written to the destination variable with the specified *dstIdx* incremented by one for each 8-byte response. The *dstIdx* will wrap back to zero if insufficient elements exist in the specified dimension.

The **rdType** parameter specifies that the ability to request sub 64 bit request is required. Valid values are uint8, uint16, uint32, uint64, int8, int16, int32 and int64.

The **memSrc** parameter specifies the possible sources for the memory read operation. The supported sources are *host* or *coproc*. The parameter is used to optimize the memory interface by using memory operations with different read sizes (8B versus 64B).

The **atomic** parameter specifies the required operation for the destination variable. The possible values depend on the target Convey platform. Refer to the table below for the supported atomic parameter values.

Example:

```
inc. AddMemRead ( queueW=5 )
        .AddDst( name= strLenRam1, var=strLenRam, field=data1 )
        .AddDst( name= strLenRam2, var=strLenRam, field=data2 )
        ;
```

This example provides a read interface with a 32 entry memory interface queue. Two read destinations are to variable *strLenRam*, fields *data1* and *data2*.

Generated Personality Interface:

The **htl** application generates interface routines to simplify issuing read requests. The following routines are provided:

ReadMemBusy()

The routine evaluates to a boolean value indicating whether the memory interface can accept a new read request. The ReadMemBusy() routine must be evaluated prior to issuing a memory read request (a run time check is made to ensure that the routine is called). Failure to check for memory interface busy would result in memory interface queue overflow and will typically hang the personality.

ReadMem_*name* (memAddr, varAddr, varIdx1, varIdx2, fldIdx1, fldIdx2, qwCnt)

The *ReadMem* routine executes as a statement initiating a memory read request. The memory read response is written to the destination variable specified in the *AddDst* command with the matching *AddDst name* parameter. The routine takes the following arguments:

memAddr – The 48-bit memory request address. The address must be 8-byte aligned. For multi-quadword requests, the address specified must allow the entire multi-quadword access to reside in a 64-byte aligned location.

varAddr1 – The variable address parameter specifies the entry within a destination memory variable where the response data is to be written. This parameter is only present if the destination variable is declared with a non-zero depth (addr1W of a shared or global variable is not zero).

varAddr2 – The variable address parameter specifies the entry within a destination memory variable where the response data is to be written. This parameter is only present if the destination variable is declared with a non-zero depth (addr2W of a shared or global variable is not zero).

varIdx1 – The parameter *varIdx1* specifies the index for the destination variable's first dimension. This parameter is only present if the destination variable has a first dimension.

varIdx2 – The parameter *varIdx2* specifies the index for the destination variable's second dimension. This parameter is only present if the destination variable has both a first and second dimension.

fldIdx1 – The parameter *fldIdx1* specifies the first index for the destination field. This parameter is only present if the destination field has a first dimension.

fldIdx2 – The parameter *fldIdx2* specifies the second index for the destination field. This parameter is only present if the destination field has both first and second dimensions.

qwCnt – The parameter specifies the number of quadwords (8 byte quantities) that are accessed by the ReadMem routine. This parameter is only present for multi quadword reads.

ReadMemPause (rsmInst)

The *ReadMemPause* routine pauses the executing thread until all outstanding memory read requests for the specified response group have completed. The response group is either the current htId or defined by the *respGrpId* parameter of *the AddReadMem* ***htd*** command. Note that many threads can be paused at the same time, but each thread must be paused waiting on a different response group. Note that a memory read request may be issued in the same instruction prior to calling the *ReadMemPause()* routine. In this case, the *ReadMemPause()* will cause the issuing thread to wait for the just issued memory read request to complete along with previous memory read requests from the response group. Note that a memory read request should not be called in the same instruction after the *ReadMemPause()* is called. The routine takes the following argument:

rsmInst – The resume instruction parameter specifies the next instruction to execute once all outstanding requests have completed.

ReadMemPoll(rspGrpId)

The *ReadMemPoll* routine evaluates to a Boolean value indicating whether the response group has any outstanding memory read requests. A true response from the routine indicates that the memory read requests have not completed. The routine is primarily used when multiple threads must wait for all requests on a specific response group to complete. Poll waiting may also be used when a thread has other work to do while waiting for a memory read to complete. Note that *ReadMemPoll()* should not be called in the same instruction after issuing a memory read request (i.e. the *ReadMemPoll()* routine does not have visibility of memory read requests issued in the same instruction). The routine takes the following argument:

rspGrpId – The read response group is either the current htId or defined by the respGrpId parameter of the AddReadMem htd command. If rspGrpId is not specified, htId is implied. .

***Atomic*Mem_*name* (memAddr, varAddr, varIdx1, varIdx2, fldIdx1, fldIdx2**)
***Atomic*Mem_*name* (memAddr, varAddr, varIdx1, varIdx2, fldIdx1, fldIdx2, data**)

The *AtomicMem* routines execute as a statement initiating an atomic memory request. The atomic memory response is written to the destination variable specified in the *AddDst* command with the matching *AddDst name* parameter.

The routine takes the same arguments as the *ReadMem_var()* routine. Some atomic memory operations also require a 64-bit data value:

data – The parameter *data* provides the value sent to memory to be used as an input to the atomic memory operation. Not all atomic operations require the *data* parameter.

Convey platforms have varying support for atomic memory operations. The following table lists the atomic memory routines available for each platform.

Atomic Parameter	Atomic Operator Routine	Data Required	HC-2/ HC-2ex	WX
setBit63	AtomicSetBit63Mem	no	yes	yes

Generated Host or Model Interface:

No host or model interface is generated.

<mod>.AddWriteMem(…)

<mod>.AddWriteMem ({ queueW=<depth> } {, rspGrpId=<name>} {, rspCntW=<width>} {, pause=<bool>} {, poll=<bool>} {,maxBw=<true/false>})

Description:

Adds a memory write interface to the associated module.

Parameters:

AddMemWrite(…)

Parameter Name	Parameter Type	Required?
queueW	Integer	No (default=5)
rspGrpId	Identifier	No (default="")
rspCntW	Integer	No (default=4)
pause	Boolean	No (default=true)
reqPause	Boolean	No (default=false)
poll	Boolean	No (default=false)
maxBw	Boolean	No (default=false)

The **queueW** parameter specifies the depth (log2) of the queue in which memory request are written. Memory requests are subsequently read from the queue and sent to memory.

The **rspGrpId** parameter specifies whether response groups are associated with the individual threads, or explicitly specified via a private or shared variable. If the *rspGrpId* parameter is not specified then the response groups are associated with individual threads, otherwise the specified variable (shared or private) is used to identify the response group.

The **rspCntW** parameter specifies the maximum number of outstanding write responses (log2) per write response group that can be active for each response group. The default is 6 or based on the number of threads (9-HtIdW), whichever is higher.

The ***pause*** parameter specifies that the module requires the capability to pause a thread when waiting for write to memory write to be completed. When the *pause* parameter is true, then the *WriteMemPause()* routine is generated. The default value for the *pause* parameter is true. Note that if *poll* is used, then FPGA resources are saved by setting the *pause* parameter to false.

The ***reqPause*** parameter specifies that the module requires the capability to pause a thread when waiting for the write request to memory to be initiated. This allows the user to set up the next memory write, without waiting for the current write to complete. When the *reqPause* parameter is true, then the *WriteReqPause()* routine is generated. The default value for the *reqPause* parameter is false. Note that if pausing a thread is not required, then FPGA resources are saved by setting the *reqPause* parameter to false.

The ***poll*** parameter specifies that the module requires the capability for a thread to wait for writes to complete by polling. When the *poll* parameter is true, then the *WriteMemPoll()* routine is generated. The default value for the *poll* parameter is false.

The **maxBw** parameter specifies that the maximum amount of memory write bandwidth is required. If a write request is attempted every instruction and *maxBw* is set to false, then 25% of the instructions will result in *WriteMemBusy()* returning true. However, for the same scenario, if *maxBw* is true then only 6% of instructions will result in *WriteMemBusy()* returning true. The *WriteMemBusy()* being true is the result of a conflict between instruction issued memory requests and memory responses.

Subcommand AddSrc(…)

Parameter Name	Parameter Type	Required?
var	Identifier	Yes
name	Identifier	No (default=<var>)
field	Identifier	No (default="")
multiWr	Boolean	No (default=false)
srcIdx	Identifier	No (default="")
memDst	Identifier	No (default=coproc)

The **name** parameter specifies the name used when constructing the *WriteMem_<name>* function. Note that if the *name* parameter is not provided then the *var* parameter is used for the function name.

The **var** parameter specifies the name of a shared or global variable that is to be the source of the memory write.

The **field** parameter specifies the field of the source variable that sources the write data. Global variables require the field parameter to be present. Shared variables only require the parameter if the shared variable is a structure or union.

The **multiWr** parameter specifies that the ability to write multiple 8-byte quantities with a single write request is required. The *srcIdx* parameter is a required parameter when *multiWr* is specified. Multi-word writes are more efficient for platforms that support accesses to memory that are larger than 64-bits. For platforms that do not support the multi-word write then individual requests are issued for each 64-bit word.

The **srcIdx** parameter specifies the index of the source variable that provides the data for multi-quadword memory write. The accepted values for the parameter are: *varAddr1, varAddr2, varIdx1, varIdx2, fldIdx1 or fldIdx2*. The *srcIdx* will wrap back to zero if insufficient elements exist in the specified dimension.

The **memDst** parameter specifies the possible destinations for the memory write operation. The supported destinations are *host* or *coproc*. The parameter is used to simulate the correct memory interface for the platform (8B versus 64B).

Example:

```
inc. AddWriteMem ( queueW=0, rspGrpId=5, rspGrpCntW=0 );
```

The example provides a memory write interface without request queuing, with 32 response groups that each allow a single response to be outstanding.

Generated Personality Interface:

The **htl** application generates routines to simplify issuing write requests. The following routines are provided:

WriteMemBusy()

The routine evaluates to a boolean value indicating whether the memory interface can accept a new write request. The *WriteMemBusy* routine must be evaluated prior to issuing a memory write request (a run time check is made to ensure that the routine is called). Failure to check for memory interface busy would result in memory interface queue overflow and will typically hang the personality.

WriteMem_*name* (memAddr, varAddr1, varAddr2, varIdx1, varIdx2, fldIdx1, fldIdx2, qwCnt)

The *WriteMem* routine executes as a statement initiating a memory write request. The data is sourced by the variable specified in the *AddSrc* command with the matching *AddSrc name* parameter. The routine takes the following arguments:

memAddr – The 48-bit memory request address. The address must be 8-byte aligned. For multi-quadword requests, the address specified must allow the entire multi-quadword access to reside in a 64-byte aligned location.

varAddr1 – The variable address parameter specifies the entry within a source memory variable. This parameter is only present if the source variable is declared with a non-zero depth (addr1W of a shared or global variable is not zero).

varAddr2 – The variable address parameter specifies the entry within a source memory variable. This parameter is only present if the source variable is declared with a non-zero depth (addr2W of a shared or global variable is not zero).

varIdx1 – The parameter *varIdx1* specifies the index for the source variable's first dimension. This parameter is only present if the source variable has a first dimension.

varIdx2 – The parameter *varIdx2* specifies the index for the source variable's second dimension. This parameter is only present if the source variable has both a first and second dimension.

fldIdx1 – The parameter *fldIdx1* specifies the first index for the source field. This parameter is only present if the source field has a first dimension.

fldIdx2 – The parameter *fldIdx2* specifies the second index for the source field. This parameter is only present if the source field has both first and second dimensions.

qwCnt – The parameter specifies the number of quadwords (8 byte quantities) that are written by the WriteMem routine.

WriteMem_*type*(memAddr, memData)

The *WriteMem* routine initiates a memory write request of size *type*. This routine is used for sub 64 bit writes. Valid values for type are uint8, uint16, uint32, uint64, int8, int16, int32 and int64.

memAddr – The 48-bit virtual memory request address. The address must be 8-byte aligned.

memData – The value that is to be written to memory. The value is of the same size as *type*.

WriteMemPause(rsmInst)

The routine causes the execution of instructions to be paused for the current thread until all write responses have been received for the specified response group. Note that a memory write request may be issued in the same instruction prior to calling the *WriteMemPause()* routine. In this case, the *WriteMemPause()* will cause the issuing thread to wait for the just issued memory write request to complete along with all other outstanding write requests from the write response group. Note that a memory write request should not be called in the same instruction after the *WriteMemPause()* is called.

rsmInst – The instruction to execute once the group pending response counter reaches zero.

WriteReqPause(rsmInst)

The routine causes the execution of instructions to be paused for the current thread until all write requests have been initiated for the specified response group. Note that a memory write request must be issued in the same instruction prior to calling the *WriteReqPause()* routine. In this case, the *WriteReqPause()* will cause the issuing thread to wait for the just issued memory write request to be initiated along with all other outstanding write requests from the write response group.

rsmInst – The instruction to execute once the group pending response counter reaches zero.

WriteMemPoll()

The *WriteMemPoll* routine evaluates to a boolean value indicating whether the specified response group has any outstanding requests. The routine is primarily used when multiple threads must wait for all requests on a specific response group to complete. Poll waiting may also be used when a thread has other work to do while waiting for a memory write to complete. Note that *WriteMemPoll()* should not be called in the same instruction after issuing a memory write request (i.e. the *WriteMemPoll()* does not have visibility of memory write requests issued in the same instruction).

Generated Host or Model Interface:

No host or model interface is generated.

<mod>.AddReadStream(…)

<mod>.AddReadStream({name=<streamName>, } type=<typeName> {,strmCnt=<strmCnt>}{, elemCntW=<width>} {, close=<bool>}{, memSrc=<src>}{, memPort=<port>}{, strmBW=<bandwidth>})

Description:

Adds a streaming memory read interface to the associated module. The destination for a memory read must be a variable that is up to 8 bytes wide. The destination variable may be indexed (array or field indexing), allowing up to eight indexed locations to be read from memory with a single read operation.

Parameters:

AddMsgIntf(…)

Parameter Name	Parameter Type	Required?
name	Identifier	No (default is unnamed)
type	Identifier	Yes
strmCnt	Integer	No (default=1)
elemCntW	Integer	No (default=32)
close	bool	No (default=false)
memSrc	Identifier	No (default=coproc)
memPort	Integer	No (default=0)
strmBW	Integer	No (default=10)
tag	Type Identifier	No (default=no tag)

The **name** parameter specifies the name of the read stream. The default stream is unnamed. Only one unnamed stream (read or write) can be specified within a module.

The **type** parameter specifies the type of the stream element that is to be read. The type can have the value of any supported native C++ type, structure or union. The width of the type must be 8, 16, 32 or 64 bits. The type parameter is required.

The **strmCnt** parameter specifies the number of independent read streams being defined. If *strmCnt* is not specified the personality interface for the stream does not have a *strmId* parameter. If *strmCnt* is specified with the value of one, a single stream is provided with the only legal value of *strmId* being zero.

The **elemCntW** parameter specifies the maximum number elements that will be transferred using a single stream open set of accesses. The default is 32, allowing a read stream count parameter of up to 2^{32}-1 elements.

The **close** parameter specifies that the *ReadStreamClose* interface routine is to be used to close an open stream. The default parameter value is false. If this parameter is set to false, the close routine is not provided and the stream will automatically close after *elemCnt* elements are accessed. If the close parameter is set to true, the read stream will stop prefetching elements from memory after elemCnt elements, but the stream will stay open until the close routine is called.

The **memSrc** parameter specifies the stream source is located on the host or coprocessor. The accepted values are 'host' or ' coproc'. The default is 'coproc'. This parameter is used to generate appropriate code for coprocessors with different memory access capabilities (i.e. 8-byte accesses versus 64-byte accesses). The parameter is also used for simulation purposes to properly model the latency and bandwidth to host versus coprocessor memory.

The **memPort** parameter specifies the memory port within the module that is to be used for the stream interface. The *memPort* parameter can be a single value, resulting in all *strmCnt* streams using the same module memory port. Alternatively, the *memPort* parameter can be a list of values (i.e. memPort={0,1,2}), resulting in the ability to specify the same or a different module memory port for each of the strmCnt streams. The module must use consecutive memory ports starting with 0. The *AddReadMem()* and *AddWriteMem()* commands use module memory port 0. If either of these commands is specified for a module then any stream for the module must not specify memory port 0.

The **strmBW** parameter specifies the target bandwidth for read streams. The parameter can have a value in range of 1-10, where 1 is the smallest specifiable bandwidth and 10 is the largest. A value of 10 will result in sufficient prefetching to cover approximately 3 microseconds of latency. The values 1-9 linearly cover the range from 0.2 microseconds to 1.5 microseconds. A value of 1 is appropriate when an element will be accessed every 12 clocks from the stream. A value of 9 will normally cover the latency to memory for accessing an element every clock, depending on the total amount of memory bandwidth being used by the personality.

The **tag** parameter specifies the type of the tag stream element. This parameter is only specified if the user desires a variable associated with read stream to be read when accessing the read stream. The type can have the value of any supported native C++ type, structure or union. The width of the type must be 8, 16, 32 or 64 bits

Example:

```
vadd.AddReadStream(name=A, type=uint64_t, memPort=0);
vadd.AddReadStream(name=B, type=uint64_t, memPort=1);
```

The example provides a two read streams, named A and B. Each stream has a unique module memory port.

Generated Personality Interface:

The **htl** application generates routines to simplify managing the message interface. The following routines are provided:

void ReadStreamOpen<*_name*>({strmId, }addr {, elemCnt}{ ,tag});

The *ReadStreamOpen* routine is used to open a read stream for subsequent element access.

strmId – Specifies the stream to open Valid values are *0 – strmCnt-1*. The *strmId* is *log2(strmCnt)* bits in width. The *strmId* parameter is not present if the *AddReadStream strmCnt* parameter is not specified. If the *AddReadStream strmCnt* parameter is 1, the *strmId* is zero.

addr – Specifies the 48 bit starting address for the first element to be accessed.

elemCnt – . If the *AddReadStream close* parameter is set to false (the default), the read stream is closed after exactly *elemCnt* elements are accessed. If the *AddReadStream close* parameter is set to true, the *elemCnt* parameter specifies the maximum number of elements that can be accessed from the read stream. Prefetching of elements stops once the *elemCnt* value is reached. The *elemCnt* parameter is *elemCntW* bits wide.

tag – Specifies the tag data to associated with the read stream. The data is read with the *ReadStreamTag* routine.

void ReadStreamClose<*_name*>({strmId});

The *ReadStreamClose* routine is used to close an open read stream. The *ReadStreamClose* routine is only present when the *AddReadStream close* parameter is set to true.

strmId – Specifies the stream to close. Valid values are *0 – strmCnt-1*. The *strmId* is *log2(strmCnt)* bits in width. The *strmId* parameter is not present if the *AddReadStream strmCnt* parameter is not specified. If the *AddReadStream strmCnt* parameter is 1, the *strmId* is zero.

bool ReadStreamBusy<*_name*>({strmId});

The *ReadStreamBusy* routine checks the open/closed state of the read stream. The *ReadStreamBusy* routine is normally used to determine if the read stream can be re-opened. The routine returns true if the read stream is open, false if the read stream is closed and can be re-opened.

strmId – Specifies the stream to check for open status. Valid values are *0 – strmCnt-1*. The *strmId* is *log2(strmCnt)* bits in width. The *strmId* parameter is not present if the

AddReadStream strmCnt parameter is not specified. If the *AddReadStream strmCnt* parameter is 1, the *strmId* is zero.

bool ReadStreamBusyMask<*_name*>({strmMask});

The *ReadStreamBusyMask* routine checks the open/closed state of multiple read streams. The *ReadStreamBusyMask* routine is normally used to determine if a set of read stream can be re-opened. The routine returns true if any read streams associated with bits set in the mask are open, false if all read streams are closed and can be re-opened. The *ReadStreamBusyMask* routine is only available if the *AddReadStream strmCnt* ***htd*** parameter is specified.

strmMask – Specifies the streams to check for open status. Bit zero of the mask is associated with *strmId* 0, etc.

bool ReadStreamReady<*_name*>({strmId});

The *ReadStreamReady* routine checks the ready status of the read stream. The *ReadStreamReady* routine is normally used to determine if the ReadStream routine is able to provide the next stream element. The routine returns true if the next element is available, false otherwise.

strmId – Specifies the stream to check for element ready. Valid values are *0 – strmCnt-1*. The *strmId* is *log2(strmCnt)* bits in width. The *strmId* parameter is not present if the *AddReadStream strmCnt* parameter is not specified. If the *AddReadStream strmCnt* parameter is 1, the *strmId* is zero.

bool ReadStreamReadyMask<*_name*>({strmMask});

The *ReadStreamBusyMask* routine checks the ready status of multiple read streams. The *ReadStreamBusyMask* routine is normally used to determine a set of read streams are able to provide the next stream element. The routine returns true if the next element is available for every stream associated with a set bit in the mask. The *ReadStreamReadyMask* routine is only available if the *AddReadStream strmCnt* parameter is specified.

strmMask – Specifies the streams to check for element ready. . Bit zero of the mask is associated with *strmId* 0, etc.

type ReadStreamPeek<*_name*>({strmId});

The *ReadStreamPeek* routine returns next element without advancing the read stream. Note that the*ReadStreamReady* routine should be called prior to calling the *ReadStreamPeek* routine to check that the next element for the stream is ready.

strmId – Specifies the stream to check for element ready. Valid values are *0 – strmCnt-1*. The *strmId* is *log2(strmCnt)* bits in width. The *strmId* parameter is not present if the

AddReadStream strmCnt parameter is not specified. If the *AddReadStream strmCnt* parameter is 1, the *strmId* is zero.

type ReadStream<*_name*>({strmId});

The *ReadStream* routine returns next element and advances the read stream. Note that the*ReadStreamReady* routine should be called prior to calling the *ReadStream* routine to check that the next element for the stream is ready.

strmId – Specifies the stream to check for element ready. Valid values are *0 – strmCnt-1*. The *strmId* is *log2(strmCnt)* bits in width. The *strmId* parameter is not present if the *AddReadStream strmCnt* parameter is not specified. If the *AddReadStream strmCnt* parameter is 1, the *strmId* is zero.

bool ReadStreamLast<*_name*>({strmId});

The *ReadStreamLast* routine returns true if the element available is the last element in the stream. The *ReadStreamLast<_name>* routine is not generated if the *AddReadStream close* parameter is true.

strmId – Specifies the stream to check for last element ready. Valid values are *0 – strmCnt-1*. The *strmId* is *log2(strmCnt)* bits in width. The *strmId* parameter is not present if the *AddReadStream strmCnt* parameter is not specified. If the *AddReadStream strmCnt* parameter is 1, the *strmId* is zero.

bool ReadStreamTag<*_name*>({strmId});

The *ReadStreamTag* routine returns the tag variable associated with the read stream. The *ReadStreamTag* routine is not generated if the *AddReadStream tag* parameter is not specified.

strmId – Specifies the stream to check for last element ready. Valid values are *0 – strmCnt-1*. The *strmId* is *log2(strmCnt)* bits in width. The *strmId* parameter is not present if the *AddReadStream strmCnt* parameter is not specified. If the *AddReadStream strmCnt* parameter is 1, the *strmId* is zero.

Generated Host or Model Interface:

No host or model interface is generated.

<mod>.AddWriteStream(…)

<mod>.AddWriteStream({name=<streamName>, } type=<typeName> {,strmCnt=<strmCnt>}{, elemCntW=<width>} {, rspGrpW=<width>}{, close=<bool>}{, memDst=<dst>}{, memPort=<port>}{, pipLen=<length>})

Description:

Adds a streaming memory write interface to the associated module.

Parameters:

AddMsgIntf(…)

Parameter Name	Parameter Type	Required?
name	Identifier	No (default is unnamed)
type	Identifier	Yes
strmCnt	Integer	No (default=1)
elemCntW	Integer	No (default=32)
rspGrpW	Integer	No (default=
close	bool	No (default=false)
memDst	Identifier	No (default=coproc)
memPort	Integer	No (default=0)
reserve	Integer	No (default=0)

The **name** parameter specifies the name of the write stream. The default stream is unnamed. Only one unnamed stream (read or write) can be specified within a module.

The **type** parameter specifies the type of the stream element that is to be written. The type can have the value of any supported native C++ type, structure or union. The width of the type must be 8, 16, 32 or 64 bits. The type parameter is required.

The **strmCnt** parameter specifies the number of independent write streams being defined. If *strmCnt* is not specified the personality interface for the stream does not have a *strmId* parameter. If *strmCnt* is specified with the value of one, a single stream is provided with the only legal value of *strmId* being zero.

The **elemCntW** parameter specifies the maximum number elements that will be transferred using a single stream open set of accesses. The default is 32 allowing a write stream count parameter of up to 2^{32}-1 elements.

The **rspGrpW** parameter specifies the maximum number of response groups (log2) available for the write stream. The default is the number of threads supported by the

module (htIdW from the AddModule htd command), with each thread being a response group.

The **close** parameter specifies that the *WriteStreamClose* interface routine is to be used to close an open stream. The default parameter value is false. If this parameter is set to false, the close routine is not provided and the stream will automatically close after *elemCnt* elements are transferred. If the close parameter is set to true, the write stream will stop writing elements to memory after elemCnt elements, but the stream will stay open until the close routine is called.

The **memDst** parameter specifies the stream destination is located on the host or coprocessor. The accepted values are 'host' or ' coproc'. The default is 'coproc'. This parameter is used to generate appropriate code for coprocessors with different memory access capabilities (i.e. 8-byte accesses versus 64-byte accesses). The parameter is also used for simulation purposes to properly model the latency and bandwidth to hos versus coprocessor memory.

The **memPort** parameter specifies the memory port within the module that is to be used for the stream interface. The *memPort* parameter can be a single value, resulting in all *strmCnt* streams using the same module memory port. Alternatively, the *memPort* parameter can be a list of values (i.e. memPort={0,1,2}), resulting in the ability to specify the same or a different module memory port for each of the strmCnt streams. The module must use consecutive memory ports starting with 0. The *AddReadMem()* and *AddWriteMem()* commands use module memory port 0. If either of these commands is specified for a module then any stream for the module must not specify memory port 0.

The **reserve** parameter specifies the depth of the pipelined data path in register stages. The *reserve* parameter can be used to ensure that sufficient space exists in the write stream to write all stages of the pipelined data path from the point where the write stream is checked to be ready and when the data is actually written to the stream

Example:

```
vadd.AddWriteStream(name=C, type=uint64_t, memPort=2);
```

The example provides a write stream, named C using module memory port 2.

Generated Personality Interface:

The **htl** application generates routines to simplify managing the message interface. The following routines are provided:

void WriteStreamOpen_*name*({strmId, }addr {, elemCnt}{, rspGrpId});

The *WriteStreamOpen* routine is used to open a write stream for subsequent element access.

strmId – Specifies the stream to open Valid values are *0 – strmCnt-1*. The *strmId* is *log2(strmCnt)* bits in width. The *strmId* parameter is not present if the *AddWriteStream strmCnt* parameter is not specified. If the *AddWriteStream strmCnt* parameter is 1, the *strmId* is zero.

addr – Specifies the 48 bit starting address for the first element to be written.

elemCnt – . If the *AddWriteStream close* parameter is set to false (the default), the write stream is closed after exactly *elemCnt* elements are written. If the *AddWriteStream close* parameter is set to true, the *elemCnt* parameter is not present. The *elemCnt* parameter is *elemCntW* bits wide.

rspGrpId – specifies the response group for the write stream. If the *AddWriteStream respGrpW* parameter is not specified the *respGrpId* parameter is not present and the thread ID is used.

void WriteStreamClose_*name*({strmId});

The *WriteStreamClose* routine is used to close an open write stream. The *WriteStreamClose* routine is only present when the *AddWriteStream close* parameter is set to true.

strmId – Specifies the stream to close. Valid values are *0 – strmCnt-1*. The *strmId* is *log2(strmCnt)* bits in width. The *strmId* parameter is not present if the *AddWriteStream strmCnt* parameter is not specified. If the *AddWriteStream strmCnt* parameter is 1, the *strmId* is zero.

bool WriteStreamBusy_*name*({strmId});

The *WriteStreamBusy* routine checks the open/closed state of the write stream. The *WriteStreamBusy* routine is normally used to determine if the write stream can be re-opened. The routine returns true if the write stream is open, false if the write stream is closed and can be re-opened.

strmId – Specifies the stream to check for open status. Valid values are *0 – strmCnt-1*. The *strmId* is *log2(strmCnt)* bits in width. The *strmId* parameter is not present if the *AddWriteStream strmCnt* parameter is not specified. If the *AddWriteStream strmCnt* parameter is 1, the *strmId* is zero.

bool WriteStreamBusyMask_*name*({strmMask});

The *WriteStreamBusyMask* routine checks the open/closed state of multiple write streams. The *WriteStreamBusyMask* routine is normally used to determine if a set of write streams can be re-opened. The routine returns true if any write stream associated with bits set in the mask are open, false if all write streams are closed and can be re-opened. The *WriteStreamBusyMask* routine is only available if the *AddWriteStream strmCnt* parameter is specified.

strmMask – Specifies the streams to check for open status. . Bit zero of the mask is associated with *strmId* 0, etc.

void WriteStreamPause_*name*({strmId, } nextInstr, {rspGrpId});

The *WriteStreamPause* routine pauses the thread until the specified write stream is closed and all writes to the stream have been written to memory

strmId – Specifies the stream to check for open status. Valid values are *0 – strmCnt-1*. The *strmId* is *log2(strmCnt)* bits in width. The *strmId* parameter is not present if the *AddWriteStream strmCnt* parameter is not specified. If the *AddWriteStream strmCnt* parameter is 1, the *strmId* is zero.

nextInstr – Specifies the next instruction for the thread to execute, once the specified stream has been completely written to memory.

rspGrpId – specifies the response group for the write stream. If the *AddWriteStream respGrpW* parameter is not specified the *rspGrpId* parameter is not present and the thread ID is used.

bool WriteStreamReady_*name*({strmId});

The *WriteStreamReady* routine checks the ready status of the write stream. The *WriteStreamReady* routine is normally used to determine if the *WriteStream* routine is able to accept the next stream element. The routine returns true if the next element can be accepted.

strmId – Specifies the stream to check for element ready. Valid values are *0 – strmCnt-1*. The *strmId* is *log2(strmCnt)* bits in width. The *strmId* parameter is not present if the *AddWriteStream strmCnt* parameter is not specified. If the *AddWriteStream strmCnt* parameter is 1, the *strmId* is zero.

bool WriteStreamReadyMask_*name*({strmMask});

The *WriteStreamBusyMask* routine checks the ready status of multiple write streams. The *WriteStreamBusyMask* routine is normally used to determine if a set of write streams are ready to accept the next element. The routine returns true if the next element can be accepted for every stream associated with a set bit in the mask. The *WriteStreamWriteMask* routine is only available if the *AddWriteStream strmCnt* parameter is specified..

strmMask – Specifies the streams to check for element ready. Bit zero of the mask is associated with *strmid* 0, etc.

type WriteStream_*name*({strmId, } elemData);

The *WriteStream* routine writes the next element to the write stream. Note that the*WriteStreamReady* routine should be called prior to calling the *WriteStream* routine to check that the next element for the stream can be accepted.

strmId – Specifies the stream to check for element ready. Valid values are *0 – strmCnt-1*. The *strmId* is *log2(strmCnt)* bits in width. The *strmId* parameter is not present if the *AddWriteStream strmCnt* parameter is not specified. If the *AddWriteStream strmCnt* parameter is 1, the *strmId* is zero.

Generated Host or Model Interface:

No host or model interface is generated.

<mod>.AddMsgIntf(…)

<mod>.AddMsgIntf(dir=<direction>, name=<name>, type=<type>)

Description:

Specifies that the module design requires a message interface with another module. The message is identified as <name>. The *dir* parameter is specified as *in* if the module receives the message and out if the module sends the message. The message is of type *<type>,* which can be a basic type, a structure or union.

A single module can specify the direction as out (i.e. the originator of a message) for each named message interface. Multiple modules can specify the direction as in.

The message is staged with one or more register stages from the source module to the destination module(s).

Parameters:

AddMsgIntf(…)

Parameter Name	Parameter Type	Required?
dir	Identifier	Yes
name	Identifier	Yes
type	Identifier	Yes
dimen	Integer	No (default=0)
queueW	Integer	No (default=0)
fanin	Integer	No (default=0)
fanout	Integer	No (default=0)
autoConn	Boolean	No (default = true)

The **dir** parameter specifies the direction the message travels. Messages that originate from a modules are considered outbound and are specified as dir=out. Messages that are from another module are considered inbound and are specified as dir=in.

The **name** parameter specifies the name of the message interface.

The **type** parameter specifies the type of the message data. The type can be a basic type, a structure or union.

The **dimen** parameter specifies the dimension of the message data. The dimension parameter must be set the same in both the sending and receiving message interface.

The **queueW** parameter specifies the depth of the queue for an inbound message.

The **fanin** parameter is used when adding a message interface to receive messages from replicated modules. It specifies the number of replications of the sending module.

The **fanout** parameter is used when adding a message interface to send messages to replicated modules. It specifies the number of replications of the receiving module.

The **autoConn** parameter is used to disable the connection of message interface ports described by the *AddMsgIntf* ***htd*** command. When used the message interface connections are described using the *AddMsgIntfConn* ***hti*** command.

Example:

```
send. AddMsgIntf( dir=out, name=msgIntf1, type=bool );
recv. AddMsgIntf( dir=in, name=msgIntf1, type=bool );
```

The example provides a messaging interface between modules send and recv. The message is of type Boolean allowing a true or false message value. The message interface is named *msgIntf1*.

Generated Personality Interface:

The **htl** application generates routines to simplify managing the message interface. The following routines are provided:

bool SendMsgBusy_*name*(dimenIdx, replIdx)

The *SendMsgBusy* routine is used to determine if the *name* messaging interface can accept a new message. The instruction should use *HtRetry()* to retry the instruction if the routine evaluates to true.

bool SendMsgFull_*name*(dimenIdx, replIdx)

The *SendMsgFull* routine is used to determine if the *name* messaging interface can accept a new message. The instruction should use *HtRetry()* to retry the instruction if the routine evaluates to true.

void SendMsg_*name*(dimenIdx, replIdx, msg)

The *SendMsg* routine sends the provided message data, msg.

bool RecvMsgBusy_*name*(dimenIdx, replIdx)

The RecvMsgBusy routine is used to determine if the *name* message interface contains a received message. A false is returned if a message is available.

bool RecvMsgReady_*name*(dimenIdx, replIdx)

The RecvMsgReady routine is used to determine if the *name* message interface contains a received message. A true is returned if a message is available to be received. The RecvMsg or PeekMsg routines are used to access the message.

<type> RecvMsg_*name*(dimenIdx, respIdx)

The RecvMsg routine is used to access the received message. The message is popped from the queue.

<type> PeekMsg_*name*(dimenIdx, respIdx)

The *PeekMsg* routine is used to access the received message. While leaving the message in the queue.

Generated Host or Model Interface:

No host or model interface is generated.

<mod>.AddHostMsg(…)

<mod>.AddHostMsg(dir=<direction>, name=<name>)

Description:

Specifies that the module requires a host message with specified message name. The *dir* parameter is specified as Inbound if the message source is the host and Outbound is the source is the coprocessor.

Parameters:

AddHostMsg(…)

Parameter Name	Parameter Type	Required?
dir	Identifier	Yes
name	Identifier	Yes

The **dir** parameter specifies the direction the message travels. Messages from the host are considered Inbound, and messages from the coprocessor are Outbound.

The **name** parameter specifies the name of the message.

Subcommand AddDst(…)

The AddDst subcommand specifies the destination for the data of an inbound host message.

Parameter Name	Parameter Type	Required?
var	Identifier	Yes
dataLsb	Integer	No (default=0)
addr1Lsb	Integer	No (default=0)
addr2Lsb	Integer	No (default=0)
idx1Lsb	Integer	No (default=0)
idx2Lsb	Integer	No (default=0)
field	Identifier	No (default=””)
fldIdx1Lsb	Integer	No (default= 0)
FldIdx2Lsb	Integer	No (default=0)
readOnly	Boolean	No (default=true)

The **var** parameter specifies the name of the shared variable that is to receive the data. The shared variable can be a scalar value, a memory, or a queue. Note: The destination must be sized appropriately to accept the message

The **dataLsb** parameter specifies the least significant bit location of the *var* data within the 56-bit message data value. The width of the shared variable type is used to determine the width of the *var* data within the message data.

The **addr1Lsb** parameter specifies the LSB location of a shared memory variable's first address. The width of the address is determined by the width of the shared memory variable's declared first depth dimension.
The **addr2Lsb** parameter specifies the LSB location of a shared memory variable's second address. The width of the address is determined by the width of the shared memory variables declared second depth dimension.

The **idx1Lsb** parameter specifies the LSB location of a shared variable's first dimension index. The width of the index is determined by the width of the shared variable's declared first dimension.

The **idx2Lsb** parameter specifies the LSB location of a shared variable's second dimension index. The width of the index is determined by the width of the shared variable's declared second dimension.

The **fldIdx1Lsb** parameter specifies the LSB location of a shared variable field's first dimension index. The width of the index is determined by the width of the shared variable field's declared first dimension.

The **fldIdx2Lsb** parameter specifies the LSB location of a shared variable field's second dimension index. The width of the index is determined by the width of the shared variable field's declared second dimension.

The **readOnly** parameter specifies that the shared variable that an inbound host message is written to is read only. The default keeps the user from inadvertently writing to a shared variable that is written be a host message.

Example:

```
inc.AddHostMsg( dir=out, name=OHM_MEM_ALLOCED );
inc.AddHostMsg(dir=in, name=IHM_MEM_ALLOC_RDY  )
        .AddDst( var=allocSize, dataLsb=0 )
        .AddDst( var=allocArray, dataLsb=20, idx1Lsb=40 )
        ;
```

Generated Personality Interface:

The **htl** application generates routines to simplify sending outbound host messages. The following Personality routines are provided:

bool SendHostMsgBusy()

The *SendHostMsgBusy* routine evaluates to a boolean value indicating that the outbound host message interface has space to accept a new message. The instruction should use *HtRetry()* to retry the instruction if the routine evaluates to true.

void SendHostMsg(name, data)

The *SendHostMsg* routine sends an outbound message to the host.

The **name** parameter specifies the name of the outbound host message.

The **data** parameter specifies the data value to be sent. The data value can be up to 56 bits in width.

Generated Host Interface:

bool SendHostMsg(uint8_t name, uint64_t data)

The *SendHostMsg* Host routine sends an inbound message to the one or more destination unit modules (or to the personality model). The *name* must be one of the specified messages in the AddHostMsg by the name parameter. The routine will return the value true if the message was sent, false otherwise.

The **name** parameter specifies the name of the inbound host message.

The **data** parameter specifies the data value to be sent. The data value can be up to 56 bits in width.

bool RecvHostMsg(uint8_t & name, uint64_t & data)

The *RecvHostMsg* Host routine receives an outbound message from one or more destination unit modules (or from the personality model). The routine will return the value true if a message was available, false otherwise.

The **name** parameter specifies the variable in which the name of the outbound personality message will be written.

The **data** parameter specifies the variable in which the data value will be written.

Generated High Level Model Interface:

bool SendHostMsg(uint8_t name, uint64_t data)

The *SendHostMsg* Model routine sends an outbound message to the host. The *name* must be one of the specified messages in the AddHostMsg by the name parameter. The routine will return the value true if the message was sent, false otherwise.

The **name** parameter specifies the name of the outbound message.

The **data** parameter specifies the data value to be sent. The data value can be up to 56 bits in width.

bool RecvHostMsg(uint8_t & name, uint64_t & data)

The *RecvHostMsg* Model routine receives an inbound message from the host. The routine will return the value true if a message was available, false otherwise.

The **name** parameter specifies the variable in which the name of the inbound host message will be written.

The **data** parameter specifies the variable in which the data value will be written.

<mod>.AddHostData(…)

<mod>.AddHostData(dir=<in or out>)

Description:

Specifies that the design requires a host data interface (inbound or outbound). The *dir* parameter is specified as 'in' if the data source is the host and 'out' is the source is the coprocessor. A single module within a unit can be designated to receive inbound host data, and similarly, a single module within a unit can be the source of outbound host data.

Parameters:

AddHostData(…)

Parameter Name	Parameter Type	Required?
dir	Identifier	Yes
maxBw	Boolean	No

The **dir** parameter specifies the direction the data travels. Data from the host are considered Inbound, and data from the coprocessor are Outbound.

The **maxBW** parameter increases the available bandwidth at the expense of additional hardware. This parameter should be set true if greater than 50 megabytes per second is needed.

Example:

```
inc.AddHostData( dir=out );
inc.AddHostData( dir=in );
```

Generated Personality Interface:

The **htl** application generates routines to simplify sending and receiving host data. The following routines are provided:

bool SendHostDataBusy()

The *SendHostDataBusy* routine evaluates to a boolean value indicating whether the outbound host data interface has space to accept data. A true value indicates that the outbound host data interface is not available to send data or a data marker. The instruction should use HtRetry() to retry the instruction if the routine evaluates to true.

void SendHostData(uint64_t data)

The *SendHostData* routine sends outbound data to the host.

The **data** parameter specifies the data value to be sent. The data value is a 64 bit value. The *SendHostDataBusy* routine must be executed prior to calling the *SendHostData* routine.

void SendHostDataMarker()

The *SendHostDataMarker* routine sends an outbound data marker to the host. The *SendHostDataBusy* routine must be executed prior to calling the *SendHostDataMarker* routine.

void FlushHostData ()

The *FlushHostData* routine flushes all data previously sent to the host. Note that the outbound host data is automatically flushed if any other type of outbound host transfer is initiated (SendHostMsg() or SendReturn_*htfunc*()). The outbound data is periodically flushed if the m_oBlkTimerUSec value of the parameter structure used by the host interface constructor is non-zero.

bool RecvHostDataBusy()

The RecvHostDataBusy routine evaluates to a boolean value indicating whether the inbound host data interface has available data or a marker. The instruction should use HtRetry() to retry the instruction if the routine evaluates to true.

uint64_t RecvHostData()

The *RecvHostData* routine receives inbound data from the host. The returned value is the 64-bit received data. The *RecvHostDataBusy* routine must be called prior to calling the *RecvHostData* routine. If data markers are used by the application's host interface then the *RecvHostDataMarker* routine must be called prior to calling *RecvHostData* (and *RecvHostData* should only be called if *RecvHostDataMarker* returns false).

uint64_t PeekHostData()

The *PeekHostData* routine allows access to the next inbound data word without advancing the read position. The *RecvHostDataBusy* routine must be called prior to calling the *PeekHostData* routine. If data markers are used by the application's host interface then the *RecvHostDataMarker* routine must be called prior to calling *PeekHostData.*

bool RecvHostDataMarker()

The *RecvHostDataMarker* routine receives an inbound data marker from the host. The returned Boolean value true if a marker is the next item received. When a marker is

received, then no data is available. The *RecvHostDataBusy* routine must be called prior to calling the *RecvHostDataMarker* routine.

Generated Host Interface:

The **htl** application generates routines to simplify sending and receiving host data to/from the Personality Ht modules or from the high level model. The following routines are provided:

int SendHostData(int size, uint64_t * pData)

The *SendHostData* routine sends inbound data to the A Personality HT Module or high level model. The amount of data sent is provided in the returned value. The amount of data sent will be at most *size* 8-byte words.

The **size** parameter indicates the maximum number of 8-byte words that are to be sent.

The **pData** parameter specifies the address of the data buffer that contains the 8-byte words to be sent.

bool SendHostDataMarker()

The *SendHostDataMarker* routine sends an inbound data marker to a Personality Ht module or the high level model. The routine returns true if the marker was sent, false otherwise.

void FlushHostData ()

The *FlushHostData* routine flushes all data previously sent to the Personality or high level model.

int RecvHostData(int size, uint64_t * pData)

The *RecvHostData* routine receives outbound data from a Personality Ht Module or the high level model. The number of 8-byte words written to the buffer will be returned.

The **size** parameter indicates the size of the data buffer in 8-byte words. At most *size* 8-byte words will be received.

The **pData** parameter specifies the data buffer that will be written with the data.

bool PeekHostData(uint64_t & data)

The *PeekHostData* routine allows access to the next outbound data word without advancing the read position. The returned value indicates if the data value is valid.

The **data** parameter specifies the variable that will be written with the data.

bool RecvHostDataMarker()

The *RecvHostDataMarker* routine returns true if the next received item is a data marker.

Generated High Level Model Interface:

The **htl** application generates routines to simplify sending and receiving host data to/from the high level model. The following routines are provided:

int SendHostData(int size, uint64_t * pData)

The *SendHostData* routine sends outbound data to the host. The amount of data sent is provided in the returned value. The amount of data sent will be at most *size* 8-byte words.

The **size** parameter indicates the maximum number of 8-byte words that are to be sent.

The **pData** parameter specifies the address of the data buffer that contains the 8-byte words to be sent.

bool SendHostDataMarker()

The *SendHostDatMarker* routine sends an outbound data marker to the host. The routine returns true if the marker was sent, false otherwise.

void FlushHostData ()

The *FlushHostData* routine flushes all data previously sent to the host.

int RecvHostData(int size, uint64_t * pData)

The *RecvHostData* routine receives inbound data from the host. The number of 8-byte words written to the buffer will be returned.

The **size** parameter indicates the size of the data buffer in 8-byte words. At most size 8-byte words will be received.

The **pData** parameter specifies the data buffer that will be written with the data.

bool PeekHostData(uint64_t & data)

The *PeekHostData* routine allows access to the next inbound data word without advancing the read position. The returned value indicates if the data value is valid.

The **data** parameter specifies the variable that will be written with the data.

bool RecvHostDataMarker()

The *RecvHostDataMarker* routine returns true if the next received item is a data marker.

<mod>.AddBarrier(…)

<mod>.AddBarrier({name=<barrierName>,} {barIdW=<0>})

Description:

Provides barrier synchronization across the threads within a module or replicated modules. Multiple barriers can exist within a module, but all must have unique names. .

Parameters:

AddBarrier(…)

Parameter Name	Parameter Type	Required?
name	Identifier	No (default is unnamed)
barIdW	Boolean	No (default = 0)

The **name** parameter specifies the user provided name for the barrier. If name is not specified then an unnamed barrier is defined. Only one unnamed barrier is allowed.

The **barIdW** parameter specifies the width of the barrier index for the barrier being defined. If not specified or a value of zero is specified, a single barrier is provided. Values over 7 are not recommended (i.e. 128 index-able barriers).

Example:

```
inc.AddBarrier();
inc.AddBarrier(name=barrier, barIdW=2 );
```

The first example creates a single unnamed barrier. The second example creates 4 barriers named barrier

Generated Personality Interface:

The **htl** application generates routines to simplify the implementation of barriers. The following routines are provided:

void HtBarrier({barId,} nextInst, threadCnt);
void HtBarrier_<name>({barId,} nextInst, threadCnt);

An unnamed barrier uses the *HtBarrier* routine and a named barrier uses the *HtBarrier_<name>* routine.

barId – The index of the barrier to be used. If the *barIdW* parameter in the ***htd*** is not specified then only one barrier is provided and the *barId* parameter is not present in the parameter list. If the *barIdW* is specified as zero, then the parameter is present and must be set to zero.

nextInst – Specifies the next instruction for all of the threads to execute, once the barrier thread count is reached.

threadCnt – Specifies the number of threads that must enter the barrier before the threads are released to execute the next instruction. The threadCnt value can range from 1 to the sum of the number of threads in the replicated module.

<mod>.AddFunction(…)

<mod>.AddFunction(type=<type>, name=<name>)

Description:

Specifies that the module requires a local function with name <name>. The function returns a value of type <type>. The declaration of the function is placed in the file named Pers<mod>.h by the *htl* application. The definition of the function is placed in the file Pers<mod>_src.cpp by the user.

Parameters:

AddFunction(…)

Parameter Name	Parameter Type	Required?
type	Identifier	Yes
name	Identifier	Yes

The **name** parameter specified the name of the function.

The **type** parameter specifies the return type for the function

Subcommand AddParam(…)

The AddParam subcommand specifies the parameters of the function.

Parameter Name	Parameter Type	Required?
dir	Identifier	Yes
type	Identifier	Yes
name	Identifier	Yes

The **dir** parameter specifies the direction the parameter is passed. The accepted values are input, output or inout. The parameter is used to determine how function parameters are declared. Parameters defined with dir=input are declared as “const <type> &<name>”. Parameters specified with dir=output or inout are declared as “<type> &<name>”.

The **type** parameter specifies the type for the function parameter.

The **name** parameter specifies name for the function parameter.

Example:

```
inc.AddFunction( type=uint32_t, name=TrailingZeroCnt)
        .AddParam( dir=input, type=uint64_t, name=in );
```

The above command would result in the following function declaration in the Pers*Module*.h file:

uint32_t TrailingZeroCnt(const uint64_t & in);

Generated Interface:

No routines are generated for the *AddFunction* command.

<mod>.AddPrimState(…)

<mod>.AddPrimState(type=<type>, name=<name>, include="<file_name>")

Description:

Adds state for a primitive. This command is used when the design incorporates a black box primitive.

Parameters:

AddPrimState(…)

Parameter Name	Parameter Type	Required?
type	Identifier	Yes
name	Identifier	Yes
include	Identifier	Yes

The **type** parameter specifies the name of the ht_state structure containing the state for the primitive. The structure is defined in the file specified by the **include** parameter.

The **name** parameter specifies the name of the primitive state.

The **include** parameter specifies file containing the structure and the primitive definitions.

Example:

```
pipe.AddPrimState( type=bbox_prim_state, name=bbox_prim_state1,
    include="PersPipe_prim.h");
```

The example defines the primitive state bbox_prim_state1, which is an ht_state structure bbox_prim_state, defined in PersPipe_prim.h.

Generated Personality Interface:

Generated Host or Model Interface:

No host or model interface is generated.

D Appendix – Hybrid Thread Instance File

This Appendix lists and describes the Hybrid Thread Instance (hti) file commands supported by the htl software application. These commands are used to add parameters to modules and add message interfaces between modules in different units. Each command description provides a brief description of the command and the available parameters.

AddModInstParams(…)

AddModInstParams(unit=<unit_name>, modPath=<module_path> {, memPort=<unit_port_list>} (, replCnt=<repl_num>});

Description:

Adds parameters to a module or module instance

Parameters:

AddPrimState(…)

Parameter Name	Parameter Type	Required?
unit	Identifier	Yes
modPath	Identifier	Yes
memPort	Integer(s)	No (Default=module memory port(s))
replCnt	Integer	No (Default=1)

The **unit** parameter specifies the name of the unit containing the module.

The **modPath** parameter specifies the path to the module instance. If multiple paths exist, any valid path can be used to identify the module instance. Replicated modules are specified by the module name and the instance indicator. For example modA[0], modA[1]…

The **memPort** parameter specifies the unit memory port used for each module memory port. Valid values are 0 – 15. Any combination of module memory ports can be mapped to a unit memory port, but consecutive unit ports must be used. The HIF is assigned to unit memory port 0. Module memory port 0 will be associated with the first unit port listed, module memory port 1 will be associated with the second unit port listed etc.

The **replCnt** parameter specifies the number of times the module is replicated. Valid values are 1 - 8.

Example 1:

In this example the add module contains a read memory interface and a write memory interface. These are defined in the htd file. Memory interfaces are assigned to module memory port 0.

```
AddModInstParams(unit=Au, modPath="ctl/add", replCnt=3);
AddModInstParams(unit=Au, modPath="ctl/add[0]", memPort=1);
AddModInstParams(unit=Au, modPath="ctl/add[1]", memPort=2);
AddModInstParams(unit=Au, modPath="ctl/add[2]", memPort=3);
```

The example replicates the add module 3 times. Note the add module is called by the ctl module. The HIF uses unit memory port 0. The memory interfaces for each instance of the add module are assigned a different unit memory port.
The unit defined in this example utilizes unit memory ports 0-3. There are 16 memory interfaces available, so 4 units can be implemented utilizing all 16 memory interfaces.

Example 2:

In this example the add htd defines the add module which contains a read and write memory interface and 2 read streams and a write stream. The memory interfaces are assigned to module memory port 0. The read streams are assigned to module memory ports 1 and 2 and the write stream is assigned to module memory port 3. The example hti file below replicates the add module and assigns unit memory ports:

```
AddModInstParams(unit=Au, modPath="ctl/add", replCnt=3);
AddModInstParams(unit=Au, modPath="ctl/add[0]", memPort=1,3,3,2);
AddModInstParams(unit=Au, modPath="ctl/add[1]", memPort=1,4,4,2);
AddModInstParams(unit=Au, modPath="ctl/add[2]", memPort=1,5,5,2);
```

The example replicates the add module 3 times. Note the add module is called by the ctl module. The HIF and the read and write memory interfaces from all instances of the add module are assigned to unit memory port 0. . The memory interfaces for each instance of the add module are assigned a different unit memory port 1. The read streams of the add modules are assigned to the same unit memory port as shown below

Add Module Instance	Read Stream Assignments
0	3
1	4
2	5

The write stream interfaces for all instances of the add module are assigned to unit memory port 5.
The unit defined in this example utilizes unit memory ports 0-5. There are 16 memory interfaces available, so 2 units can be implemented, leaving4 memory interfaces unused.

AddMsgIntfConn(…)

AddMsgIntfConn(outUnit=<unit_name>, outPath=<msg_path>, inUnit=<unit_name>, inPath=<msg_path>)

Description:

Connects message interfaces between modules. This command is only used when the htd AddMsgIntf command autoConn parameter is false, leaving the message interface unconnected

Parameters:

AddPrimState(…)

Parameter Name	Parameter Type	Required?
outUnit	Identifier	Yes
outPath	Identifier	Yes
inUnit	Identifier	Yes
inPath	Identifier	Yes

The **outUnit** parameter specifies the name of the unit containing the module that sends the message. If the message connections are identical for all units instance is not required.

The **outPath** parameter specifies the path to the message to be sent. The path includes the module path and instance and the message name and instance. Instances are not required when applicable to all instances.

The **inUnit** parameter specifies the name of the unit containing the module that receives the message. . If the message connections are identical for all modules the unit instance is not required.

The **inPath** parameter specifies the path to the message that receives the message. The path includes the module path and the message name and instance. Instances are not required when applicable to all instances.

Example:

This example implements a message ring. The *ctl* modules from 4 units are connected in a ring made up of message *A*. The ***htd*** commands creating the message interfaces and the ***hti*** commands connecting the message interfaces are shown below.

htd commands:

```
ctl.AddMsgIntf(dir=out, name=A, type=CCtlMsg, autoConn=false);
ctl.AddMsgIntf(dir=in, name=A, type=CCtlMsg, queueW=5,
autoConn=false);
```

hti commands:

```
AddMsgIntfConn(outUnit=Au[0], outPath=ctl/A, inUnit=Au[1],
inPath=ctl/A);
AddMsgIntfConn(outUnit=Au[1], outPath=ctl/A, inUnit=Au[2],
inPath=ctl/A);
AddMsgIntfConn(outUnit=Au[2], outPath=ctl/A, inUnit=Au[3],
inPath=ctl/A);
AddMsgIntfConn(outUnit=Au[3], outPath=ctl/A, inUnit=Au[0],
inPath=ctl/A);
```

This example implements messages between replicated modules, within a unit. The four replications of the *work* module are connected in a ring made up of message *A*. The ***htd*** commands creating the message interfaces and the ***hti*** commands connecting the message interfaces are shown below.

htd commands:

```
ctl.AddMsgIntf(dir=out, name=A, type=CCtlMsg, autoConn=false);
ctl.AddMsgIntf(dir=in, name=A, type=CCtlMsg, queueW=5,
autoConn=false);
```

hti commands:

```
AddModInstParams(unit=Au, modPath=ctl/work, replCnt=4);
AddMsgIntfConn(outUnit=Au, outPath=ctl/work[0]/A, inUnit=Au,
inPath=ctl/work[1]/A);
AddMsgIntfConn(outUnit=Au, outPath=ctl/work[1]/A, inUnit=Au,
inPath=ctl/work[2]/A);
AddMsgIntfConn(outUnit=Au, outPath=ctl/work[2]/A, inUnit=Au,
inPath=ctl/work[3]/A);
AddMsgIntfConn(outUnit=Au, outPath=ctl/work[3]/A, inUnit=Au,
inPath=ctl/work[0]/A);
```

E Appendix – HTV Language Reference

The HTV program translates HT application files written in C++ into Verilog appropriate for FPGA synthesis. A subset of the C++ language features is supported. The supported subset was largely determined by early users of the HT tool set. Over time, it is expected that more of the C++ language features will be supported. There are some features that are not supported today (exceptions, general pointer support) that will likely never be supported. It will be through the feedback from new and existing users that determine the new features and order for new features.

The following C++ language support list follows the order in which “The C++ Programming Language”, Third Edition book by Bjarne Stroustrup presented the material. Additional SystemC types and built-in functions are included in the list and can be the HT Programmers Guide.

Fundamental Types	Supported		
	Yes	No	Notes
Boolean (bool)	✓		
Character (char)	✓		
Integer			
C/C++ Integers			
char, unsigned char, uint8_t, int8_t	✓		
short, unsigned short, uint16_t, int16_t	✓		
int, unsigned int, uint32_t, int32_t	✓		
long long, unsigned long long, uint64_t, int64_t	✓		
SystemC Integers			
sc_uint<W>, sc_int<W>, sc_biguint<W> sc_bigint<W>	✓		
HT Integers			
ht_uint1 … ht_uint64, ht_int1 … ht_int64	✓		
Fixed Point		✗	
Floating Point		✗	
void (Function type only)	✓		
Enumerations			

Declarations	Support		
	Yes	No	Notes
Names			
Supported characters			
Letters (a … z, A … Z), Underscore (_)	✓		
Dollar Sign		×	1
Scope			
Global Variable Declarations		×	
Global Type Declarations	✓		
Function Variable Declarations	✓		
Function Type Declarations	✓		
Initialization			
No initializer (initialized to zero)	✓		
Array initializer (int a[] = { 1, 2 };	✓		
Function-style initializer (via constructor)		×	
Typedef	✓		

Notes:

1. The dollar sign is reserved for use by HT tools to provide unique names.

Pointers, Arrays and Structures	Support		
	Yes	No	Notes
Pointers		✗	
Arrays			
Multi-dimensional arrays	✓		
Array initialization	✓		
References			
Within general declaration		✗	
Within function parameter declaration	✓		
Structures			
Of fundamental types	✓		
Of nested struct / unions	✓		

Expressions		Support		
		Yes	No	Notes
C/C++ evaluation order		✓		
C/C++ operator precedence		✓		
C/C++ integer promotion		✓		
Scope resolution	class_name :: member		×	
Scope resolution	namespace_name :: member		×	
Global	:: name		×	
Member selection	object.member	✓		
Member selection	object->member		×	
Subscripting	name [expr]	✓		
Function call	name (expr_list)	✓		
Value construction	type (expr_list)		×	
Post increment	name ++	✓		
Post decrement	name --	✓		
Type identification	typeid (type)		×	
Sizeof type	sizeof(type)	✓		
Pre increment	++ name	✓		
Pre decrement	-- name	✓		
Complement	~ name	✓		
Not	! name	✓		
Unary minus	- name	✓		
Unary plus	+ name	✓		
Address of	& name		×	
Dereference	* expr		×	
Create	New type		×	
Destroy	Delete pointer		×	
Cast (type conversion)	(type) expr	✓		
Member selection	object.*pointer-to-member		×	
Member selection	pointer->*pointer-to-member		×	
Multiply	expr * expr	✓		
Divide	expr / expr	✓		1
Modulo (remainder)	expr % expr	✓		1
Add (plus)	expr + expr	✓		
Subtract (minus)	expr - expr	✓		
Shift left	expr << expr	✓		
Shift right	expr >> expr	✓		
Less than	Expr < expr	✓		
Less than or equal	Expr <= expr	✓		
Greater than	Expr > expr	✓		
Greater than or equal	Expr >= expr	✓		
Equal	Expr == expr	✓		
Not equal	Expr != expr	✓		
Bitwise AND	Expr & expr	✓		

Expressions		Support		
		Yes	No	Notes
Bitwise exclusive OR	Expr ^ expr	✓		
Bitwise inclusive OR	Expr | expr	✓		
Logical AND	Expr && expr	✓		
Logical inclusive OR	Expr || expr	✓		
Conditional expression	Expr ? expr : expr	✓		
Simple assignment	Lvalue = expr	✓		2
Multiply and assign	Lvalue *= expr	✓		2
Divide and assign	Lvalue /= expr	✓		1,2
Modulo and assign	Lvalue %= expr	✓		1,2
Add and assign	Lvalue += expr	✓		2
Subtract and assign	Lvalue -= expr	✓		2
Shift left and assign	Lvalue <<= expr	✓		2
Shift right and assign	Lvalue >>= expr	✓		2
AND and assign	Lvalue &= expr	✓		2
Inclusive OR and assign	Lvalue |= expr	✓		2
Exclusive OR and assign	Lvalue ^= expr	✓		2
Throw exception	Throw expr		✗	
Comma (sequencing)	Expr , expr	✓		
Comma (concatenation)	(expr_list)	✓		3

Notes:

1. Divide and Modulo operations are supported by the HTV tool. However, the Xilinx synthesis tool is only able to synthesize divide and modulo operations that can be performed as a table lookup. As such, these operations must keep the total width of input operands to 6 or so bits.
2. Cascaded assignment operators are not supported. (i.e. HTV does not support a = b = 3;)
3. Concatenation is a SystemC defined operator. The total width must be 64 bits or less.

Statements	Support		
	Yes	No	Notes
declaration	✓		
{ statement-list }	✓		
try *{ statement-list } handler-list*		×	
expression ;	✓		
if *(condition) statement*	✓		
if *(condition) statement* ***else*** *statement*	✓		
switch *(condition) statement*	✓		
while *(condition) statement*		×	
do *statement* ***while*** *(condition)*		×	
for (*for-init-statement ; condition ; expression) statement*	✓		1
case *constant-expression : statement*	✓		
default *: statement*	✓		
break ;	✓		
continue ;		×	
return *expression ;*	✓		
goto *identifier ;*		×	
identifier : statement		×	

Notes:

1. *for* statements are completely unrolled. The *for* statement control expressions must be specified as *for (int i = x; i < y; i += z)*, where *x*, *y* and *z* are constant integer expressions.

Functions	Support		
	Yes	No	Notes
Declaration			
Global scope	✓		
As member of struct/union/class	✓		
As constant (void func() const)		✗	
Overloading function names			
In same scope		✗	
In different scope	✓		
Default argument value (, arg=0)		✗	
Unspecified number of arguments (, …)		✗	
Pointer to Function		✗	
Argument passing			
Fundamental types	✓		
Struct/union/class	✓		
Arrays		✗	1
Array elements	✓		
As reference	✓		
As constant	✓		
Value return			
Void return	✓		
Fundamental type return	✓		
Struct/Union/Class	✓		
Array		✗	
As reference		✗	
As constant		✗	

Notes:

1. The current release of HTV does not support passing an array of elements to a function. However, HTV does support passing a structure that contains an array.

Namespaces and Exceptions	Support		
	Yes	No	Notes
Namespaces		x	
Exceptions		x	

Classes	Support		
	Yes	No	Notes
Member functions	✓		
Virtual member functions		x	
Access Control (public/private/protected -parsed but ignored)	✓		
Constructors		x	
Static members		x	
Constant member functions		x	
Copying class objects (a = b;)	✓		
Self-reference		x	
In class function definitions	✓		
Helper functions		x	
Overloaded operators		x	
Destructors		x	

Other Language Features	Support		
	Yes	No	Notes
Operator overloading		x	
Derived classes		x	
Templates		x	
Exception Handling		x	

3 Customer Support Procedures

E-mail	support@conveycomputer.com
Web	Go to Customer Support at www.conveycomputer.com