HT Programmer's Guide

February 16, 2015

Version 1.4

900-00044-000

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

Xilinx, Virtex and ISE are registered trademarks of Xilinx in the United States and other countries.

Revisions

Version	Description
1.0	January 22, 2014. Initial Release
1.1	July 22, 2014. Updated API, clarified queuing between host HIF and unit modules, added clarification of global variable definition with memory read/write, added rspgrpID to ReadMemPoll, updated modlule message interface
1.11	August 6, 2014. Added note to include PersCommon.h when using primitives, corrected CNY_HT_SIM_ML description
1.2	August 12, 2014. Added streaming interface
1.3	November 4, 2014. Added barrier support, streaming interface example, CNY_HT_FREQ, read stream tag support, read stream last element support. Updated model development and input clocks for primitives.
1.4	February 16, 2015. Replaced zero parameter for staged variables with primOut parameter. Added examples of module messages with replicated modules. Added response groups to write streams, resulting in removal of WriteStreamPauseMask API. Updated message interconnect hti command. Modified for open source. Other misc corrections. Consolidated much of the reference Guide in this document. Updates for open sourcing.

Table of Contents

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

- Application Development Using HT
- Simulation Model Development Using HT
- Personality Development Using HT
- Examples
- Debugging a Personality
- HT Tool Flow

1.3 Related Documents

The Convey HT Reference Manual contains the architectural description of the HT infrastructure. It also contains a detailed reference for all ***htd*** commands and the routines generated by the HTL.

1.4 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++.

Users should be familiar with the Overview of an HT Design Section of the HT Reference Manual.

1.5 Required Software

The HT environment requires the following components:

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

2 HT Development Overview

This section contains a high level description of the step-by-step process for developing an application for the Convey coprocessor, using the HT Tools. The steps are illustrated in Figure 1.

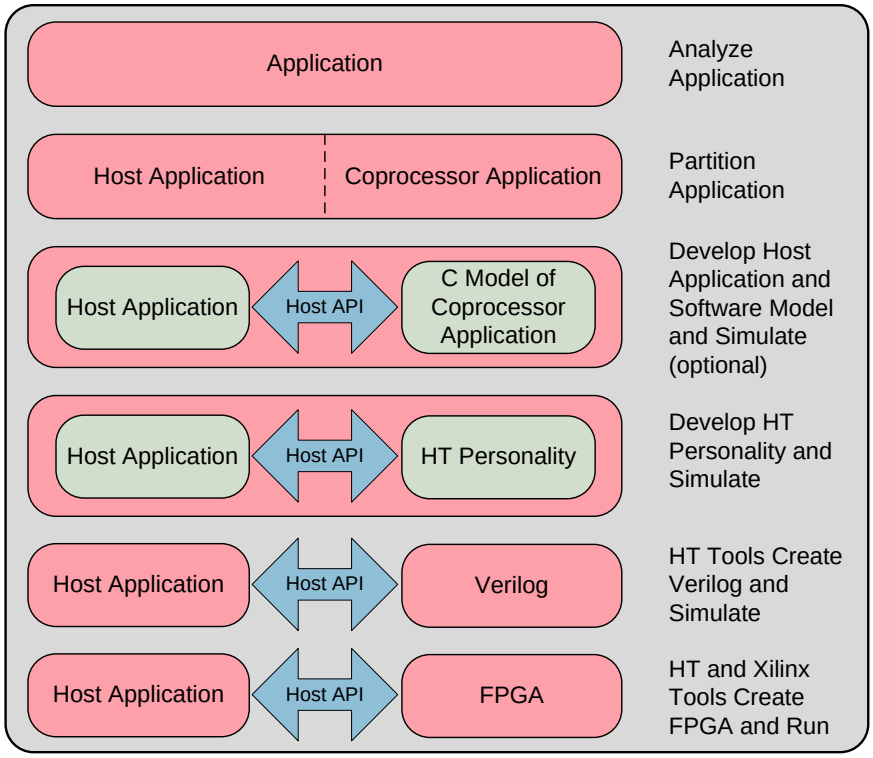


Figure 1 - HT Development Steps

Later sections of this document contain detailed information on the following

- Host application development (Section 3)
- C model development (Section 4)
- HT personality development (Section 5)

2.1 Analyze Application

The first step of personality development is to understand the problem to be solved.

- What bottlenecks are limiting the performance?
- What data structures are involved?
- How parallelizable is the application?

Answers to these questions provide the first insight into how the application can be accelerated.

2.2 Partition the Application

The application is divided into two components, the host application and the coprocessor application. At this point the interface between the host and coprocessor is being defined. The following should be identified

- Communication requirements between host and coprocessor
- Data movement requirements between host and coprocessor

2.3 Modify the Host Application and Test Host API

2.3.1 Modify HT Host Application

The Host application is modified replacing the kernel to be implemented on the coprocessor with calls utilizing the Host API to control and transfer data to and from the coprocessor application.

2.3.2 Develop Software Model of Custom Personality

A high level Model of the coprocessor application can be developed, so that the host application and coprocessor interface can be verified. The Model utilizes the Host API and implements the functionality in software, of the kernel to be implemented on the coprocessor. This is an optional step that can be useful for testing the host application prior to or concurrently with the development of the personality.

2.3.3 Simulate Host Application

Once the coprocessor software model is complete, the host application can be simulated. This allows the host application and the interface between the host and coprocessor to be verified, prior to the development of the coprocessor hardware.

2.4 Develop Coprocessor Personality

The coprocessor application or personality is divided into modules. The functionality of each module described in two parts, the Hybrid Thread Description (***htd***) and Module HT instructions.

The ***htd*** defines the Module's variables, interfaces and call and return capabilities. The Module HT instructions define the functionality of the Module.

2.5 Simulate Personality and Tune for Performance

The host application and the coprocessor personality are simulated. Reports are generated by the simulation, providing feedback for performance tuning.

2.6 Use HT Tools to Create Verilog

The Hybrid Thread Verilog translator (HTV) is used to generate verilog for the coprocessor personality. The personality includes the custom logic defined by the user and infrastructure provided by Convey.

2.7 Use HT Tools to Create Coprocessor FPGA

The HT tools interface with the Xilinx tools to create the coprocessor FPGA, using the verilog generated by the HT tools

3 HT Host Application Programming

The HT host application can be a standalone program or a library linked with a larger application. The application is a standard C/C++ application that runs on the host processor. The application controls, and transfers data to, and from the coprocessor, using routines from the CHtHIF class. The CHtHIF class contains a CHtAuUnit class for each unit in the design. The CHtAuUnit class contains routines generated from the unit design.

3.1 CHtHIF Class

Execution of an HT application starts with a program running on the host processor. The host processor starts the coprocessor by first constructing the Host Interface (HIF) class. This loads the personality into the FPGAs, allocates communication queues in memory and starts the HIF thread on the coprocessor. This thread then waits for call or messages to be sent from the host.

Once the HIF class has been constructed, the host can construct unit class(es) on the coprocessor, for the host application's use. The host then communicates with the units independently through call/return and messaging interfaces defined in the HT Host Application Programming Interface.

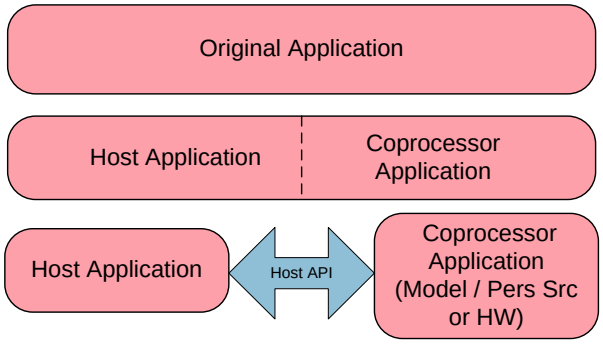


Figure 2 – Host Application

Two classes are provided by the HT infrastructure for use in Host Application: CHtHif and CHtAuUnit. CHtAuUnit is a member of the CHtHif class. Figure 3 shows the member functions in the CHtHif class.

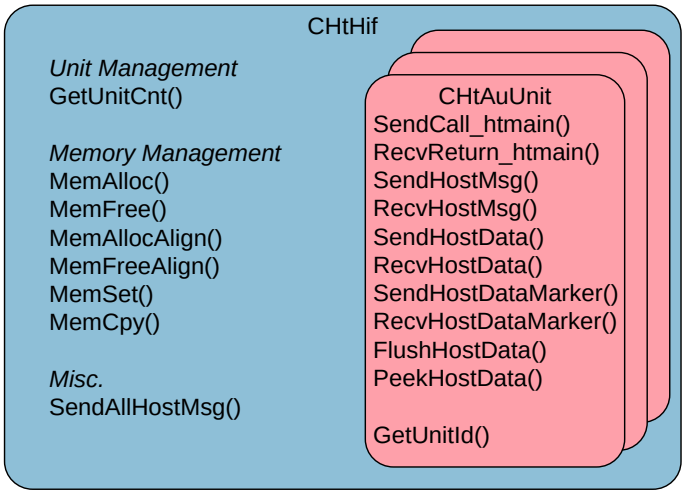


Figure 3 – CHtHif Class Definition

3.1.1 Unit Management

Units are prepared for use by constructing a HIF class for each Unit. This section describes functions provided in the *CHtHif* class to support constructing and or freeing (destructing) Units.

The *GetUnitCnt* function returns an integer containing the number of units in the personality..

```
int cnt = pHtHif->GetUnitCnt();
```

This function also allows the use of the same source code for various development targets (model, personality, verilog simulation), since the number of units implemented may differ.

A single units is constructed as shown below

```
CHtAuUnit *pAuUnit = new CHtAuUnit(pHtHif);
```

If multiple Units are needed, the code can loop n times using the value returned from *GetUnitCnt* as n.

```
CHtHif *pHtHif = new CHtHif();
int unitCnt = pHtHif->GetUnitCnt();
CHtAuUnit ** pAuUnits = new CHtAuUnit * [unitCnt];
for (int unit = 0; unit < unitCnt; unit++)
      pAuUnits[unit] = new CHtAuUnit(pHtHif);
```

Upon completion, units are deconstructed using

```
Delete pAuUnit;
```

3.1.2 HIF Parameters

3.1.3 Coprocessor Memory Management

Data locality is important in achieving maximal performance, when using the Convey Coprocessor. Memory can be allocated in host memory or coprocessor memory. The

host processor can access host memory much more efficiently than coprocessor memory, just as the coprocessor can access coprocessor memory much more efficiently than host memory. Data that is to be accessed by the coprocessor should reside in the coprocessor memory. After the coprocessor completes, the data may be moved back to host memory if it is to be accessed by the host. In this manner, it is important for the user to allocate regions where they are most advantageously utilized by the respective processing elements. If the majority of the work within a given kernel is done on the coprocessor, then the major data elements should be moved to coprocessor memory.

The standard Linux memory allocation routines are supported and are used for host memory.

The remainder of this subsection is split into three topics. The first section covers the ability to directly allocate/free memory coprocessor memory. The second section covers initialization of coprocessor memory and the third section covers copying techniques between host and coprocessor memory regions.

3.1.3.1 Memory Allocation

The HT infrastructure provides a series of functions that allow the user to explicitly allocate memory on the coprocessor. Any time a user allocates dynamic memory regions, they must use the same type of function to free the memory region. For example if a memory block is allocated using a coprocessor allocation function, the memory must be freed with the coprocessor free function.

It is important to note when allocating memory on the coprocessor, the default behavior of the coprocessor allocators is an implied page fault operation. All coprocessor pages are faulted in by default via a *touch* policy on every block.

Applications requiring memory allocated on the Convey coprocessor should use the following functions:

`void*  MemAlloc(size_t size)`

This routine allocates memory bound to the Convey coprocessor. A pointer to the allocated memory is returned or *NULL* if the allocation failed. This routine requires one parameter:

> **`size:`** size of the requested memory allocation in bytes

`void  MemFree(void *pMem)`

This routine frees memory previously allocated on the Convey coprocessor, using **`MemAlloc`**. There is no return value. This function requires one parameter:

> **`pMem:`** pointer to the memory block to be freed. This must be a value previously returned by an allocation function.

`void* MemAllocAlign(size_t align, size_t size)`

This routine allocates memory bound to the Convey coprocessor. The memory is aligned on a power of two boundary. A pointer to the allocated memory is returned or *NULL* if the allocation failed. This routine requires two parameters:

> **`align:`** memory is allocated on align boundary, which must be a power of 2
>
> **`size:`** size of the requested memory allocation in bytes

```
void  MemFreeAlign(void *pMem);
```

This routine frees memory previously allocated on the Convey coprocessor, using **MemAllocAlign**. There is no return value. This function requires one parameter:

pMem: pointer to the memory block to be freed. This must be a value previously returned by an allocation function.

3.1.3.2 Initialize Memory

Convey also provides the necessary facilities to accelerate memory initialization. There exists a Convey-accelerated version of *memset(…)* that utilize a hardware-based datamover operation that natively understands the interleave factors and page-table layouts of coprocessor memory. Traditional *memset(…)* operations can be performed on coprocessor memory, but the Convey equivalents are higher performance for large areas of memory. The calling semantics for the Convey equivalent functions are identical to the traditional versions.

```
void *  MemSet(void *pDst, int pattern, size_t len);
```

This routine fills the first *len* bytes of the memory area pointed to by *dst* with the 8 bit constant value *pattern*.

pDst: pointer to the target location

pattern: the 8 bit pattern to write (type converted from int to unsigned char)

len: the number of bytes to write

3.1.3.3 Memory Movement

Convey also provides the necessary facilities to accelerate memory copies between host and coprocessor memory regions. There exists a Convey accelerated version of *memcpy(…)* that utilizes a hardware-based datamover operation that natively understands the interleave factors and page-table layouts of coprocessor memory. Traditional *memcpy(..)* operations can be performed on coprocessor memory, but the Convey equivalents are higher performance for large amounts of data. The calling semantics for the Convey equivalent functions are identical to the traditional versions.

```
void *  MemCpy(void *pDst, const void *pSrc, size_t len);
```

This routine copies *len* bytes from memory area *src* to memory area *dest*.

pDst: pointer to the target location

pSrc: pointer to the original location

len: the number of bytes to move

3.1.3.4 Memory Management Usage

The following code allocates arrays in host memory (*a1, a2* and *a3*) and in coprocessor memory (cp_a1, cp_a2 and cp_a3), then copies host arrays (a1, and a2) from host memory to the arrays in coprocessor memory (cp_a1 and cpa2).

```
// Allocate host memory arrays
a1 = (uint64_t *)malloc(vecLen * sizeof(uint64_t));
```

```
a2 = (uint64_t *)malloc(vecLen * sizeof(uint64_t));
a3 = (uint64_t *)malloc(vecLen * sizeof(uint64_t));
memset(a3, 0, vecLen * sizeof(uint64_t));

//  Fill host memory arrays
for (i = 0; i < vecLen; i++) {
      a1[i] = i;
      a2[i] = 2 * i;
}
//  Construct the HIF and units
CHtHif *pHtHif = new CHtHif();
int unitCnt = pHtHif->GetUnitCnt();

CHtAuUnit ** pAuUnits = new CHtAuUnit * [unitCnt];
for (int unit = 0; unit < unitCnt; unit++)
      pAuUnits[unit] = new CHtAuUnit(pHtHif);

// Allocate coprocessor memory arrays
uint64_t *cp_a1 = (uint64_t*)pHtHif->MemAlloc(vecLen *
sizeof(uint64_t));
uint64_t *cp_a2 = (uint64_t*)pHtHif->MemAlloc(vecLen *
sizeof(uint64_t));
uint64_t *cp_a3 = (uint64_t*)pHtHif->MemAlloc(vecLen *
sizeof(uint64_t));

//  Copy host arrays to coprocessor memory
pHtHif->MemCpy(cp_a1, a1, vecLen * sizeof(uint64_t));
pHtHif->MemCpy(cp_a2, a2, vecLen * sizeof(uint64_t));
//  Initialize results array in coprocessor memory
pHtHif->MemSet(cp_a3, 0, vecLen * sizeof(uint64_t));
//
//  Initiate processing on coprocessor
//  Wait for coprocessor to complete
//
//  Copy results from coprocessor memory to host memory
pHtHif->MemCpy(a3, cp_a3, vecLen * sizeof(uint64_t));

//  Process results on host

//  Free memory
free(a1);
free(a2);
free(a3);
pHtHif->MemFree(cp_a1);
pHtHif->MemFree(cp_a2);
pHtHif->MemFree(cp_a3);

delete pHtHif;
```

Example 1 – Memory Management

3.1.4 Host Application API

The HT Host API utilizes the Host Interface (HIF) which consists of a set of queues for each unit on the coprocessor. Each unit has an inbound and an outbound control, message and data queue as shown in Figure 4.

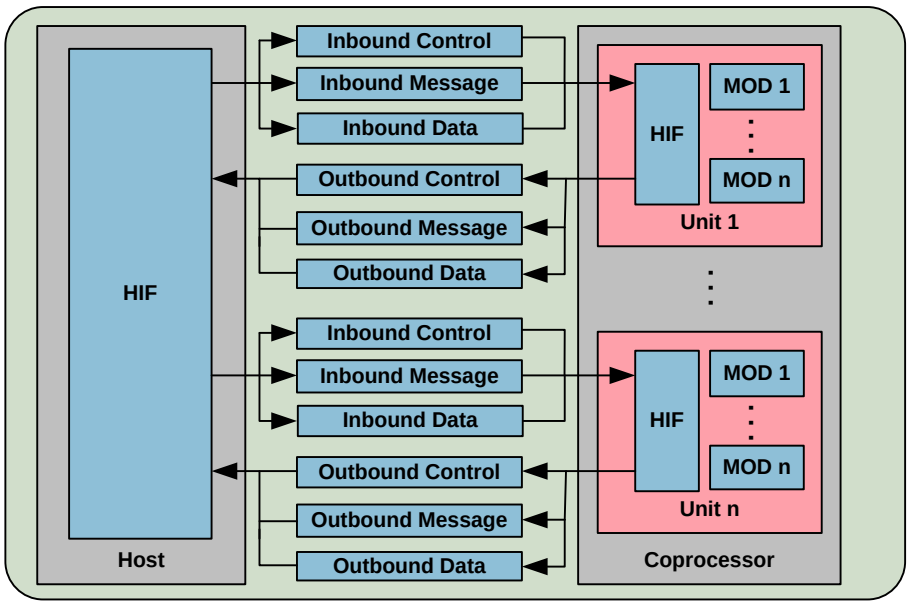


Figure 4 – Host Interface

Like types of transactions to or from the same unit is ordered, so messages are received in the order they are sent, data and data markers are received in the order sent and calls and returns are received in the order they are sent. The queues for different types of transactions are independent, so there is no relationship between different transaction types.

The queues for different units are independent, so there is no relationship between transactions to / from different units.

3.2 CHtAuUnit Class

The CHtAuUnit class is defined within the CHtHif class. The content of the CHtAuUnit class is design specific, based on ***htd*** commands. Table 1 shows CHtAuUnit routines and the ***htd*** commands that provide the routines.

CHtAuUnit Routine	***htd* Command**
bool SendCall_htmain(...)	<mod>.AddEntry(func=htmain, host=true)
bool RecvReturn_htmain(…)	<mod>.AddReturn(htmain)
bool SendHostMsg(…)	<mod>.AddHostMsg(dir=in, name=<name>) AddDst( )
bool RecvHostMsg(…)	<mod>.AddHostMsg(dir=out, name=<name>)

CHtAuUnit Routine	*htd* Command
int SendHostData(…) bool SendHostDataMarker() void FlushHostData()	<mod>.AddHostData(dir=in)
int RecvHostData(…) bool RecvHostDataMarker() bool PeekHostData(…)	<mod>.AddHostData(dir=out)

Table 1 – CHtAuUnit Routines

Details of the HIF routines and usages are provided in the following sections.

3.2.1 Call / Return

The *SendCall_htmain* routine begins a thread on the coprocessor. The thread in the host routine continues execution.

```
bool SendCall_htmain(<callParam1>, <callParam2>, …)
```

The entry point in the called module and call parameters are defined with the ***htd*** *AddEntry* command and the *AddParam* subcommand(s). The *host* parameter of the entry point is set to *true*, indicating the entry point is used by the host. The optional call parameters are written to the callee's private variables. The type of the private variables may differ from the type of the host variable, since the width of a variable in the personality is optimized to conserve hardware.

The *SendCall_htmain* routine will return a Boolean value *true* if the call was sent, *false* otherwise.

The *RecvReturn_htmain* routine is used to determine that a call was returned from a module.

```
bool RecvReturn_htmain(<rtnParam1>, <rtnParam2>, …)
```

The return parameters are defined with the ***htd*** *AddReturn* command and the *AddParam* subcommand(s) of the module. The type of the optional return parameters may be optimized in the personality to conserve hardware. This is done by assigning the *type* parameter to the actual width needed by the personality and the *hostType* parameter to a standard width.

The *RecvReturn_htmain* routine will return Boolean value *true* if a call was returned, *false* otherwise.

3.2.2 Host Message Interface

Using a Host Message Interface, the host can send or receive messages to or from one or more modules in a unit. Host messages contain 56 bits of data. Multiple messages can be used if more than 56 bits are needed. Modules that require sending or receiving host messages include the ***htd*** *AddHostMsg* command.

3.2.2.1 Inbound Host Message

An inbound host message is used to modify or initialize shared variable(s) in the receiving module(s) of the unit. Multiple modules in a unit may receive the same host message.

Host messages can be sent to modules in the personality at any time, since the shared variable(s) in the receiving module(s) are automatically updated with the message content.

The *SendHostMsg* routine sends an inbound message to one or more modules within a unit (or to the personality model).

```
bool SendHostMsg( uint8_t name, uint64_t data )
```

The *name* parameter specifies the name of the inbound host message. The *name* must be one of the specified message names in the ***htd*** *AddHostMsg* commands.

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

The *SendHostMsg* routine will return the value *true* if the message was sent, *false* otherwise. The send will fail if the queue to the coprocessor unit is full.

The *CHtHif* class contains a routine that can be used to send the same host message to all units.

```
bool SendAllHostMsg( uint8_t name, uint64_t data )
```

The content of the host message is defined in the ***htd*** command which defines the resources used by the module(s) to receive the message.

The *SendAllHostMsg* routine will return the value *true* if the messages were sent, *false* otherwise.

3.2.2.2 Outbound Host Message

An outbound host message is used to receive a message from a module (or the Model). An outbound message will only be received while a call is active, since the coprocessor initiates the message.

The *RecvHostMsg* Host routine receives an outbound message from one or more destination unit modules (or from the personality model).

```
bool RecvHostMsg( uint8_t & name, uint64_t & data )
```

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.

The *RecvHostMsg* routine will return the value *true* if a message was available, *false* otherwise.

3.2.3 Host Data Interface

Using the Host Data Interface, the host can send data to one module in a unit and/or receive data from one module in a unit. The host data interface is a streaming interface, modeled after the Linux socket send / receive interface, The host data interface can be used to send or receive a user defined number of 8 byte words. The module sending or receiving host data must have a host data interface specified in the ***htd*** file.

Inbound and outbound data is transferred to/from the coprocessor when anyone of the following occurs:

- The data buffer is full (64K buffer)
- 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 default timer value for both inbound and outbound data is 1000 usec. This value can be modified with a HIF parameter when constructing the HIF.

Host data is typically sent or received while a coprocessor call is active, since a module on the coprocessor must receive or initiate the message.

3.2.3.1 Inbound Host Data

Inbound host data is used to send a block of data to a module in the personality (or to the Model).

The *SendHostData* routine places inbound data in the queue to be sent to a module (or the personality Model). A single module within a unit can be designated to receive inbound host data. The amount of data placed in the queue is provided in the returned value. The amount of data will be at most ***size*** 8-byte words.

```
int SendHostData( int size, uint64_t * pData )
```

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.

The *SendHostData* routine will return the number of 8-byte words sent. Less than the full size is sent only if the send buffer becomes full. The remaining data can be sent by adjusting the *size* and *pData* and calling *SendHostData* again.

In latency sensitive situations, the application can force all data previously placed in the inbound data queue to be sent using the *FlushHostData* routine.

```
void FlushHostData()
```

3.2.3.2 Outbound Host Data

The *RecvHostData* routine allows the Host to receive outbound data from a module (or the personality Model). The number of 8-byte words written to the buffer will be returned.

```
int RecvHostData( int size, uint64_t * pData )
```

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.

The *RecvHostData* routine returns the number of 8-byte words written into the buffer by the personality module or the Model.

3.2.3.3 Host Data Marker

There are a variety of ways the application can utilize data markers. Some examples are listed below:

- Indicate start / end of a data set
- Terminate Execution

The *SendHostDataMarker* routine sends an inbound data maker to the personality (or to the Model). The routine returns *true* if the marker was sent and *false* otherwise.

```
bool SendHostDataMarker()
```

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

```
bool RecvHostDataMarker()
```

3.2.3.4 Host Data Usage

Inbound Host Data

When sending inbound host data, the host application must assure the entire data block is sent. Example 2 shows usage of the *SendHostData* routine.

```
int sentCnt = 0;
while (sentCnt < data_size)
      sentCnt += pAuUnit->SendHostData(data_size - sentCnt,
      data + sentCnt);
```

Example 2 – Inbound Host Data

Inbound Host data with Markers

When sending an inbound marker, the host must assure the marker is sent. An application sending inbound host data, preceded by a marker and followed by a marker is shown in Example 3.

```
int sentCnt = 0;
// send marker and assure it is sent
while (!pAuUnit->SendHostDataMarker() );
while (sentCnt < data_size)
      sentCnt += pAuUnit->SendHostData(data_size - sentCnt,
      data + sentCnt);
// send marker and assure it is sent
while (!pAuUnit->SendHostDataMarker() );
```

Example 3 Inbound Host Data with Marker

Outbound Host Data

Example 4 shows the host receiving a known amount of host data.

```
int recvCnt = 0;
while (recvCnt < data_size)
      recvCnt += pAuUnit->RecvHostData(data_size - recvCnt,
      data + recvCnt);
```

Example 4 – Outbound Host Data

Outbound Host Data with Markers

Example 5 shows an application receiving outbound host data, preceded by a marker and followed by a marker.

```
int recvCnt = 0;
//Receive start Marker
while ( pAuUnit->RecvHostDataMarker() )
      //Receive data until End Marker
      while (!pAuUnit->RecvHostDataMarker() )
            recvCnt += pAuUnit->RecvHostData(max_data_size
            - recvCnt, data + sentCnt);
```

Example 5 – Receive Host Data and Marker

In this example size of the host data buffer (pointed to by *data*) is *max_data_size*. This is the maximum number of 8 byte words that can be received.

4 Development of the C Model of the Coprocessor

A Model of the coprocessor application is developed to allow simulation of the host application prior to the development of the coprocessor personality. The Model is written in C/C++ and contains routines generated by the HTL for the host interface

The C model of the coprocessor contains the portion of the application targeted for implementation on the coprocessor. The coprocessor functionality is modeled in the CHtModelAuUnit class.

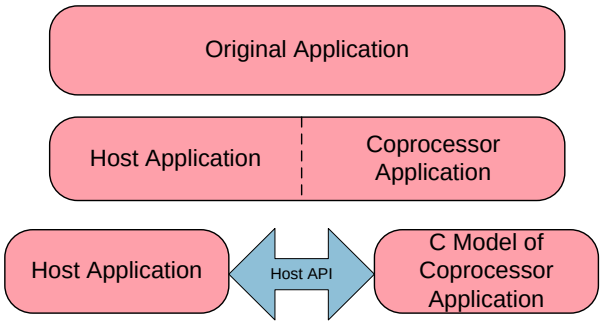


Figure 5 – Coprocessor Model

4.1.1 CHtModelAuUnit Routines

The functionality of a Model is defined in two components, the Hybrid Thread Description (***htd***) which generates the design specific Model HT API, found in the CHtAuUnit class and the C Coprocessor Model defined in the UnitThread function, which uses HT instructions from the CHtModelAuUnit class.

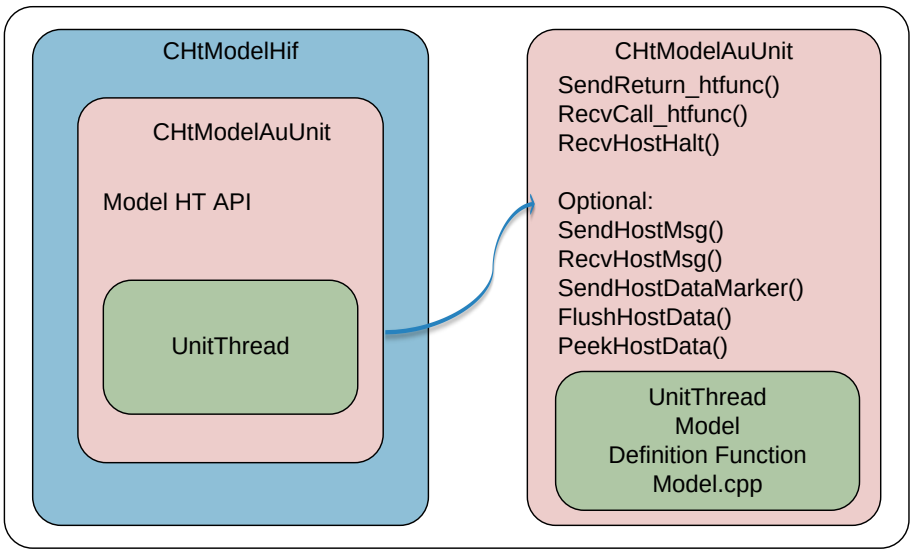


Figure 6 – CHtModelAuUnit Class

Table 2 shows the *CHtModelAuUnit* Model HT API routines and the ***htd*** commands that provide the routines.

CHtAuUnit Routine	***htd* Command**
bool RecvCall_htmain(…)	<mod>.AddEntry(func=htmain, host=true)
bool SendReturn_htmain*(* …)	<mod>.AddReturn(htmain)
bool SendHostMsg(…)	<mod>.AddHostMsg(dir=out, name=<name>) AddDst( )
bool RecvHostMsg( )	<mod>.AddHostMsg(dir=in, name=<name>)
int SendHostData( ) bool SendHostDataMarker( ) void FlushHostData( )	<mod>.AddHostData(dir=out)
int RecvHostData( ) bool RecvHostDataMarker( ) bool PeekHostData( )	<mod>.AddHostData(dir=in)

Table 2 – CHtModelAuUnit Routines

The model typically contains one module. The model ***htd*** is required to contain an *AddModule*, *AddEntry* and *AddReturn* command. Other ***htd*** commands are included only if needed.

4.2 Host Message Interface

Using a Host Message Interface, the model can send or receive messages to or from the host. Host messages contain 56 bits of data. Multiple messages can be used if more than 56 bits are needed. Modules that require sending or receiving host messages include the ***htd*** *AddHostMsg* command.

4.2.1 Inbound Host Message

An inbound host message is used to modify or initialize shared variable(s) in the model. The *RecvHostMsg* routine receives an inbound message from the host.

```
bool RecvHostMsg( uint8_t & name, uint64_t & data )
```

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 message data value will be written.

The *RecvHostMsg* routine will return the value *true* if a message was available, *false* otherwise.

4.2.2 Outbound Host Message

An outbound host message is used to send a message from a model

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

```
bool SendHostMsg( uint8_t name, uint64_t data )
```

The *name* parameter specifies the name of the outbound host message. The *name* must be one of the specified message names in the ***htd*** *AddHostMsg* commands.

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

The *SendHostMsg* routine will return the value *true* if the message was sent, *false* otherwise. The send will fail if the queue to the host is full.

4.3 Host Data Interface

Using the Host Data Interface, the model can send data to the host and/or receive data the host. The host data interface is a streaming interface which can be used to send or receive data user defined number of 8 byte words. The model must have a host data interface specified in the ***htd*** in order to send or receive a host data.

Inbound and outbound data is transferred to/from the model when anyone of the following occurs:

- A data block is full (64K block)
- 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 default timer value for both inbound and outbound data is 1000 used. This value can be modified with a HIF parameter when construction the HIF.

4.3.1 Inbound Host Data

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

```
int RecvHostData( int size, uint64_t * pData )
```

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.

The *RecvHostData* routine returns the number of 8-byte words written into the buffer by the host.

4.3.2 Outbound Host Data

Outbound host data is used to send a block of data from the model to the host.

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

```
int SendHostData( int size, uint64_t * pData )
```

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.

The *SendHostData* routine will return the number of 8-byte words sent. Less than the full size is sent only if the send buffer becomes full. The remaining data can be sent by adjusting the *size* and *pData* and calling *SendHostData* again.

In latency sensitive situations, the application can force all data previously placed in the outbound data queue to be sent using the *FlushHostData* routine.

```
void FlushHostData()
```

4.3.3 Host Data Marker

There are a variety of ways the application can utilize data markers. Some examples are listed below:

- Indicate start / end of a data set
- Used to instruct the model to terminate the thread

The *SendHostDataMarker* routine sends an outbound data maker to the host. The routine returns *true* if the marker was sent and *false* otherwise.

```
bool SendHostDataMarker()
```

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

```
bool RecvHostDataMarker()
```

5 Development of an HT Personality

An HT personality contains the portion of the user application to be implemented on the coprocessor. 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. A single coprocessor instruction may represent multiple lines of source code. A coprocessor instruction may perform its operation across multiple pipelined clocks.

5.1 CHtPersHIF Class

The CHtPersHIF class contains the functionality of the personality. It contains a CHtPers<Unitn>Unit class for each unit. If the design contains multiple units, multiple instances exist. Each module in the unit is defined in a CPers<*Modn*> class which is contained in the unit class.

The functionality of a Module is defined in two components.

- Hybrid Thread Description (***htd***), which generates the design specific Module HT API.
- Module Custom Instructions descriptions, which use Module HT API.

The Module HT API generated by the HTL is in the CPers<module_name> class. The custom instructions are defined in a function called Pers<module_name>, which is a member of the CPers<module_name> class as illustrated in Figure 7

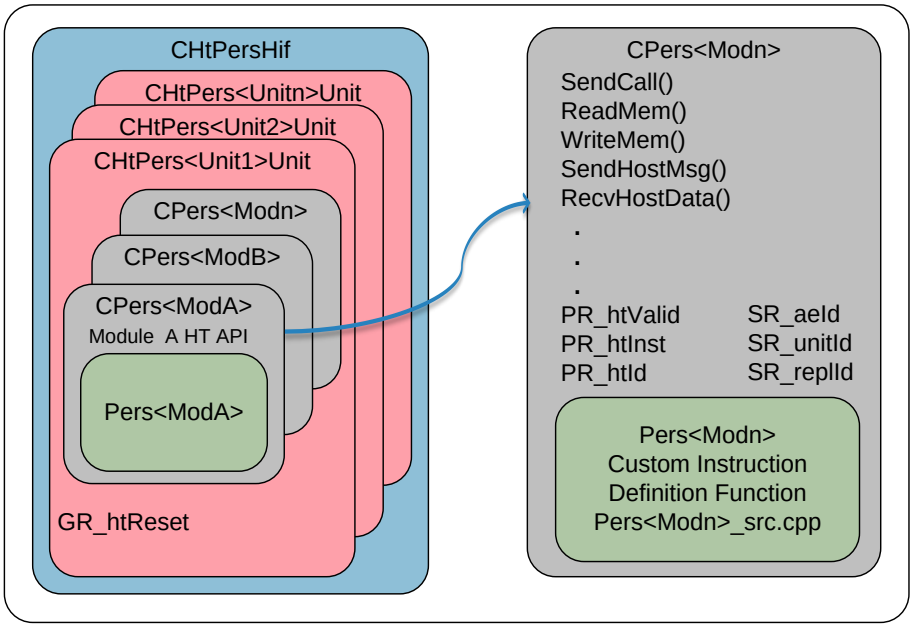


Figure 7 – ChtPersHif Class

5.2 CHtPers<Unitn>Unit Class

There is a CHtPers<Unitn>Unit class for each unit in the design. The CHtPers<Unitn>Unit class contains a class for each module in the unit (CPers<Modn>). The unit class also contains a global reset variable, GR_htReset.

5.3 CPers<Modn> Class

The CPers<Modn> class contains a set variables provided by the infrastructure for the module and the functionality of a user specified Module. The module functionality is defined in two components.

- The Hybrid Thread Description (***htd***) which generates the Module HT API. (*src_pers/*.htd*)
- The module custom instruction descriptions (HT instructions), which use HT generated API routines. (*src_pers/Pers<Modn>_src.cpp*)

5.3.1 CPers<*Mod*> Class Variables

The CPers<Modn> class contains a set of variables for use in the module custome instruction descriptions.

5.3.1.1 CPers<Mod> Class Shared Variables

Shared variables contain state that is share between all threads within a module.

The shared variables described below are provided in the CPers<*Modn*> class. They cannot be modified by the personality.

Private Variable	**Description**
SR_aeId	Identifies the AE
SR_unitId	Identifies the unit
SR_replId	Identifies the replication ID for replicated modules

Table 3 – CPers<*Modn*> Shared Variables

5.3.1.2 CPers<Mod> Class Private Variables

Private variables contain 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 variables described below are provided in the CPers<*Modn*> class. They cannot be modified by the personality.

Private Variable	**Description**
PR#_htValid	Indicates that the thread ID and HT instruction are valid
PR#_htInst	Contains the HT instruction to be executed
PR#_htId	Contains the thread ID

Table 4 – CPers<*Modn*> Private Variables

For multistage designs the # represents the stage number with the first stage being 1. This number is omitted for single stage designs. (See Section 5.4.4 for more information on multi-stage designs)

5.3.2 Module HT API

The Module HT API routines, generated from the ***htd*** are contained in the CPers<Modn> class. These routines provide

- Specification of variables (private, shared and global)
- Thread instruction scheduling
- Thread call, return, spawn and join
- Memory read, write and atomic operators
- Communication with Host
- Communication with other Modules

Figure 8 illustrates the interfaces that can be supported in the Module HT API.

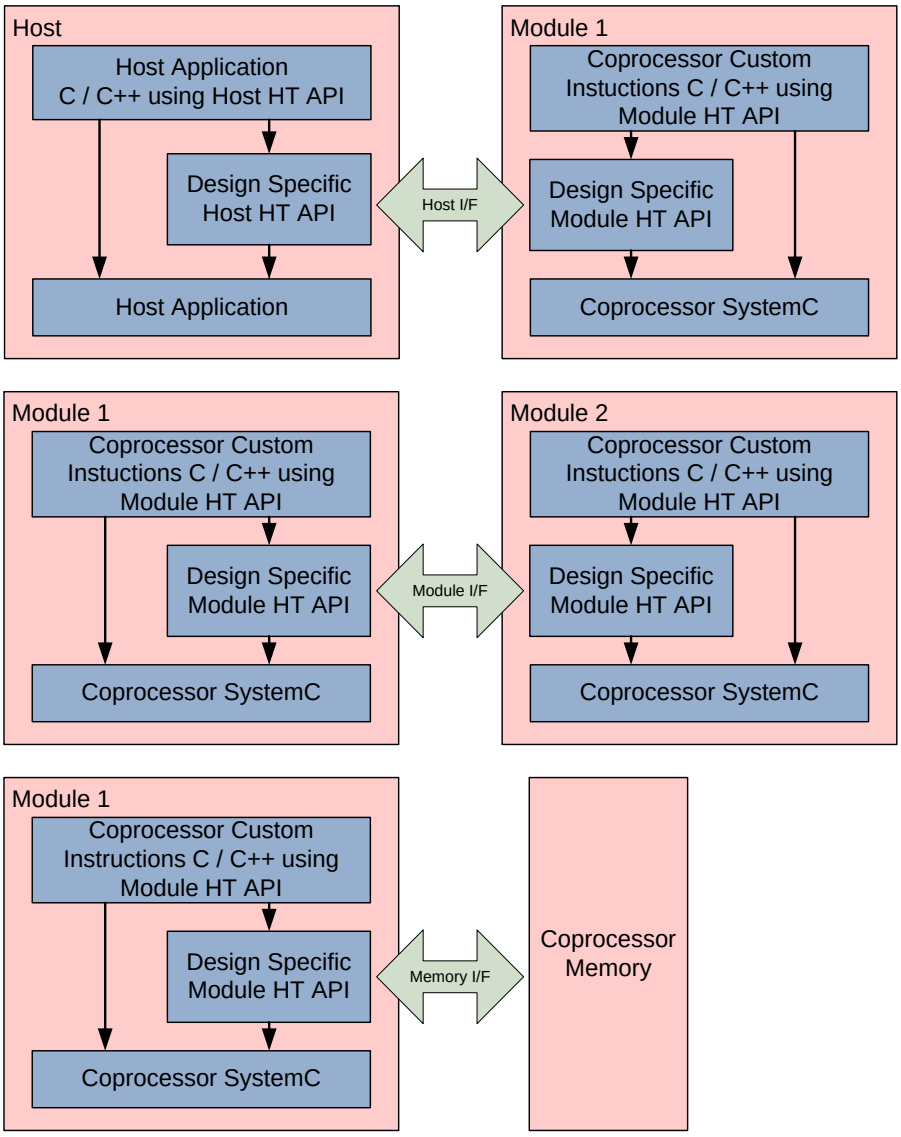


Figure 8 – Module Interfaces

The routines generated by the HTL are in the CPers<*Modn*> class. The HT instructions are written by the user in a function called Pers<*Modn*>, which is a member of the CPers<*Modn*> class as illustrated in Figure 7.

5.3.2.1 CPers<Modn> Class Routines

The following tables provide a summary of routines that may be available to the personality in the Module HT API, along with the ***htd*** commands that are used to include the routines. The HT instructions in the Pers<Modn> function utilize the routines in the API.

CPers<Modn> Routine	*htd* Command	Description
HtContinue(*inst*)	AddModule( )	Specifies the next instruction to execute.
HtRetry()	AddModule( )	Indicates the current instruction should be re-executed. Performance Monitor retry count is incremented.
HtTerminate( )	AddModule(reset=<inst>)	Terminates the currently executing thread. Only used to terminate reset thread.
HtPause(*inst*)	AddModule(pause=true)	Suspends the currently executing thread. Indicates the next instruction to execute when a *HtResume* is executed for the thread.
HtResume(*threadID*)	AddModule(pause=true)	Resumes the specified thread (that was previously suspended using *HtPause*). Note threadID is not specified if ***htd*** *htIdW* is not specified.

Table 5 – Threading and Instruction Scheduling Routines

CPers<*Modn*> Routine	*htd* Command	Description
SendCallBusy_*htfunc*( )	AddCall( )	Indicates if the interface has space to accept another call thread
SendCall_*htfunc*(…)	AddCall( )	Begins a thread in the called module. The calling module pauses until a corresponding return is received
SendCallFork_*htfunc*(…)	AddCall(…, fork=true,)	Begins a thread in the called module. The thread in the calling module continues execution
RecvReturnPause_*htfunc*(…)	AddCall( )	Pauses the current thread if any forked calls have not returned
RecvReturnJoin_*htfunc*( )	AddCall( )	Must be called in the first instruction by the returning forked call thread.

CPers<*Modn*> Routine	***htd* Command**	**Description**
SendTransferBusy_*htfunc*( )	AddTransfer( )	Indicates if the transfer interface has space to accept another transfer thread
SendTransfer_*htfunc*(…)	AddTransfer( )	Begins a thread in the destination module. The thread in the source module terminates
SendReturnBusy_*htfunc*()	AddReturn( )	Indicates whether the return interface has space to accept another return thread
SendReturn_*htfunc*(…)	AddReturn( )	Terminates the thread in the called module and resumes a thread in the original calling module

Table 6 – Call/Return and Join Routines

CPers<Modn> Routine	***htd* Command**	**Description**
SendMsgBusy_*name* ( )	AddMsgIntf(dir=out, …)	Indicates if the *name* message interface can accept a new message.
SendMsgFull_*name* ( )	AddMsgIntf(dir=out, …)	Indicates if the *name* message interface can accept a new message.
SendMsg_*name* (…)	AddMsgIntf(dir=out, …)	Send message data on the *name* message interface.
RecvMsgBusy_*name*(…)	AddMsgIntf(dir=in, …)	Indicates if the *name* message interface contains a received message.
RecvMsgReady_*name*(…)	AddMsgIntf(dir=in, …)	Indicates if the *name* message interface contains a received message.

CPers<Modn> Routine	*htd* Command	Description
RecvMsg_*name*(…)	AddMsgIntf(dir=in, …)	Access the received message from the *name* message interface and remove the message from the queue.
PeekMsg_*name*(…)	AddMsgIntf(dir=in, …)	Access the received message from the *name* message interface but do not remove the message from the queue.

Table 7 – Module Message Routines

CPers<Modn> Routine	*htd* Command	Description
SendHostMsgBusy(…)	AddHostMsg(dir=out, …)	Indicates if the host message interface can accept a new message.
SendHostMsg (…)	AddHostMsg(dir=out, …)	Send an outbound message to the host.

Table 8 – Host Message Routines *

Note: Receive host messages are written directly to the modules shared variable(s). If used, the ***htd*** *AddHostMsg,* with the *dir* parameter set to *in* must be used with an *AddDst* subcommand specifying the destination shared variable.

CPers<Modn> Routine	*htd* Command	Description
SendHostDataBusy ( )	AddHostData(dir=out)	Indicates if the outbound host data interface has space to accept data.
SendHostData(…)	AddHostData(dir=out)	Send outbound host data.
SendHostDataMarker()	AddHostData(dir=out)	Send outbound host data marker.
FlushHostData()	AddHostData(dir=out)	Flush all outbound host data.
RecvHostDataBusy ( )	AddHostData(dir=in)	Indicates if the outbound host data interface has space to accept data.
RecvHostData( )	AddHostData(dir=in)	Send outbound host data.
RecvHostDataMarker( )	AddHostData(dir=in)	Send outbound host data marker.
PeekHostData( )	AddHostData(dir=in)	Flush all outbound host data.

Table 9 – Host Data Routines

CPers<Modn> Routine	***htd* Command**	**Description**
ReadMemBusy ( )	AddReadMem(…)	Indicates if the memory interface can accept a read request.
ReadMem_*name* (…)	AddReadMem(…)	Initiates a memory read request.
ReadMemPause(…)	AddReadMem(…, pause=true)	Pauses thread until outstanding memory read request have completed.
ReadMemPoll( )	AddReadMem(…, poll=true)	Indicates if the thread (or defined group of responses (response group) has any outstanding read requests.
WriteMemBusy ( )	AddWriteMem(…)	Indicates if the memory interface can accept a write request.
WriteMem_*name* (…)	AddWriteMem(…)	Initiates a memory write request.
WriteMem_*type* (…)	AddWriteMem(…)	Initiates a sub 64 bit write request
WriteMemPause(…)	AddWriteMem(…, pause=true)	Pauses thread until outstanding memory write request have completed.
WriteReqPause(…)	AddWriteMem(…,reqPause=true)	Pauses thread until outstanding memory write requests have been initiated.
WriteMemPoll( )	AddWriteMem(…, poll=true)	Indicates if the thread (or defined group of responses (response group) has any outstanding write requests.

Table 10 – Memory Access Routines

CPers<Modn> Routine	***htd* Command**	**Description**
ReadStreamOpen<*_name*>( )	AddReadStream(…)	Open a read stream.
ReadStreamClose<*_name*>( )	AddReadStream(close=true…)	Close an open read stream
ReadStreamBusy<*_name*>( )	AddReadStream(…)	Indicates if a read stream is open or closed.
ReadStreamBusyMask<*_name*>( )	AddReadStream(strmCnt=<value>, …)	Indicates if a set of read streams are closed.
ReadStreamReady<*_name*>( )	AddReadStream(…)	Indicates if the read stream is able to provide the next stream element
ReadStreamReadyMask<*_name*>( )	AddReadStream(strmCnt=<value>, …)	Indicates if a set of read streams are able to provide the next stream elements.
ReadStreamPeek<*_name*>( )	AddReadStream(…)	Returns the next element of a read stream without advancing the stream.
ReadStream<*_name*>( )	AddReadStream(…)	Returns the next element of a read stream.
ReadStreamLast<*_name*>( )	AddReadStream(close=false, …)	Returns true if the last element of a read stream.
ReadStreamTag<*_name*>( )	AddReadStream(tag=<type>, …)	Returns the tag variable associated with the read stream.
WriteStreamOpen<*_name*>( )	AddWriteStream(…)	Open a write stream.
WriteStreamClose<*_name*>( )	AddWriteStream(close=true,…)	Close an open read stream
WriteStreamBusy<*_name*>( )	AddWriteStream(…)	Indicates if a write stream is open or closed.
WriteStreamBusyMask<*_name*>( )	AddWriteStream(stremCnt=<value>, …)	Indicates if a set of write streams are closed.

CPers<Modn> Routine	*htd* Command	Description
WriteStreamPause<_*name*>	AddWriteStream(…)	Pauses a thread until the write stream is closed and all writes to the stream are complete
WriteStreamReady<_*name*>( )	AddWriteStream(…)	Indicates if the write stream is ready to accept the next stream element
WriteStreamReadyMask<_*name*>( )	AddWriteStream(strmCnt=<value>, …)	Indicates if a set of write streams are ready to accept the next stream elements.
WriteStream<_*name*>( )	AddWriteStream(…)	Writes the next element to a read stream.

Table 11 – Streaming Memory Access Routines

CPers<Modn> Routine	*htd* Command	Description
HtBarrier ( )	AddBarrier(…)	Provides barrier synchronization for an unnamed barrier across threads in a module or replicated modules
HtBarrier_<name> ()	AddBarrier(name=<name>,…)	Provides barrier synchronization for a named barrier across threads in a module or replicated modules

Table 12 – Barrier Routines *

5.4 Pers<Modn> Function

The Pers<*Modn*> is a member function of CPers<*Modn*> class. This function is called every clock cycle. The module functionality is implemented in this function as a series of HT instructions written in C/C++, using the routines in the HTL generated API routines. The function is provided by the user and is contained in a file named *Pers<Modn>_src.cpp*, located in the *src_pers* directory.

5.4.1 HT Threads

A module typically is capable of handling multiple threads. An HT thread is selected for execution each clock cycle the module is called. The threads in a module are operated on in a round robin fashion, so that on each clock cycle, one HT instruction is executed by one of the threads. The execution of the HT instruction is pipelined over multiple

stages, allowing the HT infrastructure to prepare thread state before execution of the instruction. On the execute cycle, the HT instruction programmed by the user is executed. Following execution of the instruction the HT infrastructure saves private variables so they are available to subsequent HT instructions executed by the thread. Every HT instruction must schedule the thread's next instruction or action. Thread scheduling routines include the following:

- HtContinue – execute the indicated HT instruction
- HtRetry – execute the current HT instruction again and updates performance monitor counts
- HtPause – suspend the current thread
- HtResume – resume indicated suspended thread
- Call – begin a thread in another module
- Return – return the thread to the module that spawned the thread
- <action>Pause – wait for action indicated to complete

The HT instruction flow for a module is illustrated in Figure 9.

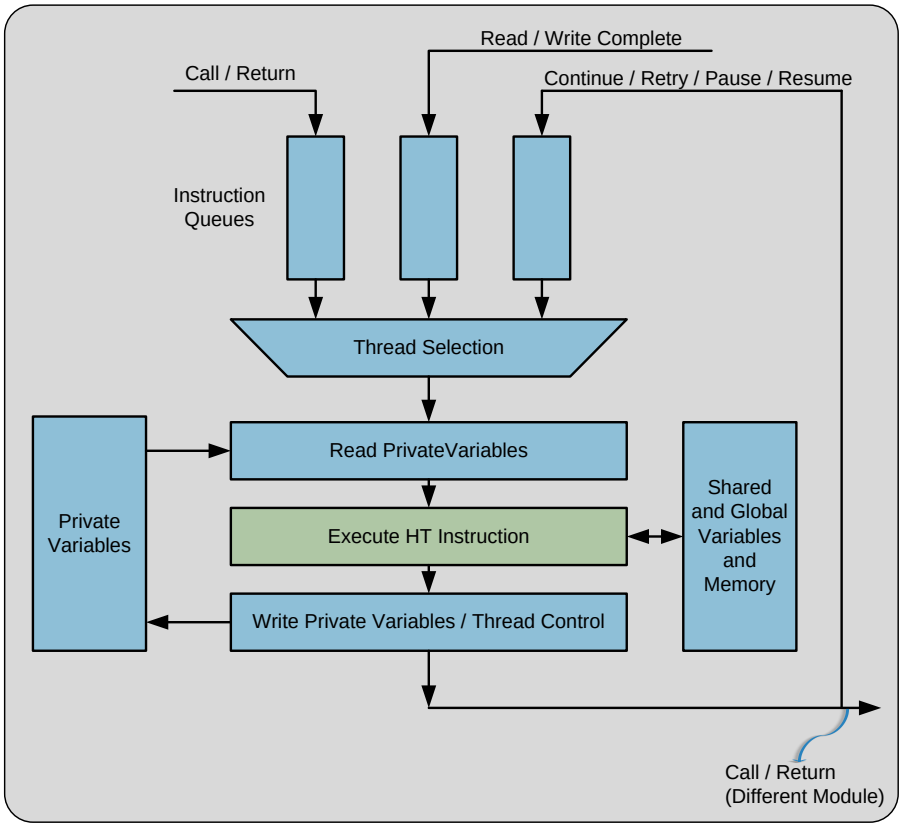


Figure 9 – HT Instruction Flow

5.4.2 HT Functions

An HT function is defined with an *AddEntry* and *AddReturn* ***htd*** command in the called module. The *AddEntry* command specifies the name of the HT function and the first HT

instruction executed when the HT function is called. The *AddReturn* command provides the return linkage for the HT function.

When an HT function is called a new thread is started. The thread executes a sequence of HT instructions. An HT instruction is executed when *PR_htValid* is true. When true, *PR_htInst* contains the HT instruction that will be executed and *PR_htId* will contain the thread number (if there is more than 1 thread). Example 6 shows an example of a HT function that would have an entry instruction called *INST1*.

A module often contains a single HT function so to make the code easier to follow the HT function is often given the same name as the module.

Modules may contain multiple HT functions. Each HT function is defined with an entry point and a return. If multiple HT functions contain the same HT instructions, the user must qualify the return functionality with the entry point.

5.4.3 HT Instructions

HT functions are written as a sequence of HT instructions. An HT instruction is one or more operations that will execute within a single clock cycle and typically consists of 10-50 lines of code along with calls to routines in the Module HT API. Before the instruction is executed, the HT infrastructure reads all private state variables so they can be used during the instruction. After the operation is complete updates to private variables are saved.

After the operation(s) to be performed in the instruction is complete, the instruction calls an API routine which indicates what the thread should do next.

Instructions are defined using the htd command

HT instructions are typically defined using a case statement for each HT instruction defined by the module as shown in Example 6.

```
void
CPersExMod::PersExMod()
{
      if (PR_htValid) {
            switch (PR_htInst) {
            case INST1: {

                  //Instruction 1 functionality
                  //After INST1 execute INST2
                  HtContinue(INST2)
            }
            break;
            case INST2: {

                  //Instruction 2 functionality
                  //Direct thread
                  HTContinue(INST3);
            }
            break;
            case INST3: {
                  //Instruction 3 functionality
                  HTContinue(RTNINST);
            }
            break;
            case RTNINST; {
                  //Return functionality
                  // Check if return interface can accept a
                  return
                  if (SendReturnBusy_<mod>()) {
                        htRetry();
                        break;
                  }
                  // Send return
                  SendReturn_<mod>();

            {
      }
}
```

Example 6 – Typical Custom Instruction Definition

Each HT instruction is executed in a single clock cycle. If the work to be done in a custom instruction cannot be completed in a single clock cycle the following actions can be taken:

- Divide the work into additional custom instructions
- Call another module to perform some of the work
- Add stages to the custom instruction and divide the work between the stages

See Appendix A for guidelines for determining functionality that can be completed in a single clock cycle.

5.4.4 Multi-Stage Instructions

Multi-stage instructions allow the execution of an instruction to be spread across multiple clock cycles. Multi-stage instructions are implemented in modules in the following situations:

- The module reads shared variables implemented as block RAMs
- An instruction contained in the module cannot be completed in a single clock cycle

The *AddStage* ***htd*** command is used to specify the module requires multiple stages to execute. Typically, the *execStg* parameter specifies the number of stages required by the module. In the *execStg* the following occur

- private variables are saved
- memory accesses are performed
- thread control operation is performed

All instructions in the module will be staged across *exeStg* stages. Private, shared and global variables can be accessed on any instruction stage, using the methods described in Section 5.5.5. Note that all global variable reads must be on the same instruction stage and all global variable writes must be on the same instruction cycle as specified in the *AddGlobal* ***htd*** command.

Intermediate data values may be stored between instruction stages in temporary variables. Temporary variables are declared using the *AddStage, AddVar* subcommand. Only scalar temporary variables are supported.

The temporary variable may be automatically initialized in the first stage from a private variable of the same name or initialized to zero. The subcommand can also specify that the temporary variable be connected in a pipeline in successive stages.

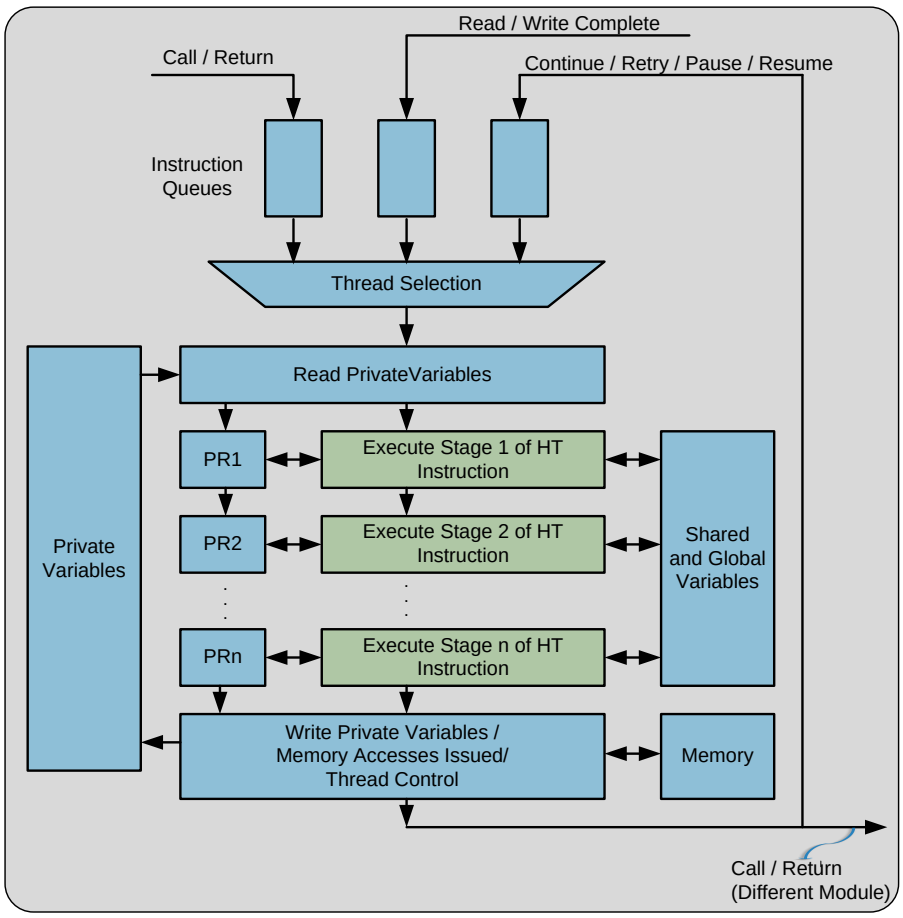


Figure 10 – Typical Staged HT Instruction Flow

5.4.4.1 Temporary Stage Variable Access

The *T#_name* macro is used to access temporary stage variables. The temporary variable may be scalar or an array of scalar variables. The type of the accessed data may be a simple type, a structure or a union.

Temporary stage variable read:

```
data = T#_name{ [idx1] { [idx2] }}
```

Temporary stage variable write:

```
T#_name{ [idx1] { [idx2] }} = data
```

5.4.4.2 Multi-Stage Usage

Pseudo code for a three stage pipeline is shown in Example 7. The module in this example has the following ***htd*** commands

```
mod_name.AddStage(execStg=3);
mod_name.AddGlobal(var=gvar, rdStg=1, wrStg=2)
```

All the instructions in the example are piped across three clocks. INST1 has functionality in each clock. INST2 has functionality in stages 1 and 2. INST3 only has functionality in stage 1. All instructions must direct the thread to the next instruction in the execute stage (stage 3).

```
void
CPersSmrd::PersSmrd()
{
      if (PR1_htValid){
            switch (PR1_HtInst){
            //  All global reads must be done in stage 1
            case INST1: {
                  //  Stage 1 functionality of INST1
            }
            break;
            case INST2: {
                  //  Stage 1 functionality of INST2
            }
            break;
            case INST3: {
                  //  Stage 1 functionality of INST3
            }
            break;
            }
      }
      if (PR2_htValid){
            switch (PR2_HtInst){
            //  All global writes must be done in stage 2
            case INST1: {
                  //  Stage 2 functionality of INST1
            }
            break;
            case INST2: {
                  //  Stage 2 functionality of INST2
            }
            break;
            }
      }
      if (PR3_htValid){
      //  All private variables will be save
            switch (PR3_HtInst){
            case INST1: {
                  //  Stage 3 functionality of INST1 with all memory
                  //  accesses
                  HtContinue(INST2);
            }
            case INST2: {
                  HtContinue(INST3)
            }
            break;
            case INST3: {
                  //  Return functionality
            }
            break;
            }
      }
}
```

Example 7 – Three Stage Pipeline

Shared memory reads from block RAM require multiple stages. A design containing a shared block RAM read is shown in Example 10.

5.4.5 Non-Instruction Functionality

The Pers<Modn> function may also contain functionality that is not included in a HT instruction. Examples of non-instruction functionality include

- Reset functionality
- Incorporation of other Verilog functionality (Section 5.14.4)
- Reading and writing memory streams (Section 5.12)

Non-instruction functionality is executed every clock cycle.

```
void
CPersExMod::PersExMod()
{
      if (PR_htValid) {
            switch (PR_htInst) {
            case INSTRUCTION1: {

                  //Instruction 1 functionality
                  //After INTSTRUCTION1 execute
                  INTSTRUCTION2
                  HtContinue(INSTRUCTION2)
            }
            break;
            case INSTRUCTION2: {

                  //Instruction 2 functionality
                  //Direct thread
            }
            break;
            }
      }
      if (GR_htReset) {

            // reset functionality
      }
{
```

Example 8 – Reset Non-Instruction Functionality

5.5 Variables

HT supports three categories of variables, private, shared and global. The widths of variables used in the personality are typically minimized to conserve resources in the FPGA. All signed and unsigned integers with widths between 1 and 64 are pre-defined as ht_uint1_t, ht_uint2_t, … ht_uit64_t and ht_int1_t, ht_int2_t, ...ht_int64_t.

5.5.1 Private

Private variables contain 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.

Private variables can be scalar or memory.

Private variables can be

- modified by the HT instructions
- passed as an argument of a call
- returned as parameter from a call

Private variables are added to a Module using the ***htd*** command / subcommand

```
<mod>.AddPrivate( … )
        AddVar( … )
        AddVar( … )
;
```

When an HT function is called the thread's private variables are initialized to zero or set to the value passed in the call.

5.5.2 Shared

Shared variables contain 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 instruction. Care must be taken when using pipelined instructions, since shared state must only be accessed by a single stage.

Shared state variables are added to a Module using the ***htd*** command / subcommand

```
<mod>.AddShared()
        AddVar( … )
        AddVar( … )
;
```

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

Shared variables can be

- modified by the HT instructions
- modified by non-instruction functionality
- modified by memory reads
- modified by host messages

Shared variables are stored in FPGA Black RAM or distributed RAM depending on the required dimensions.

Shared variables are not modified by call or return parameters.

Shared variables can be initialized.

5.5.3 Global

Global variables are accessible by all threads of all modules in a unit. They can be scalar or a memory. Global variables are added to a Module using the htd command

```
<mod>.AddGlobal()
        AddVar( … )
        AddVar( … )
;
```

Global variables can be modified by up to 2 writers. These can be

- HT instructions

- memory reads

Global variables are not modified by return parameters.

Global variables are not initialized.

5.5.4 Declaring Variables

Variables are defined in the ***htd*** description. The ***htd*** description specifies the variable scope, data type and in some cases the array dimensions or field description. Table 13 shows the ***htd*** command that is comparable to the C/C++ declaration.

C++ Declaration	*htd* Command
int variable	<mod>.AddPrivate() AddVar(type=int, name=variable); <mod>.AddShared() AddVar(type=int, name=variable); <mod>.AddGlobal(var=variable) AddField(type=int);
int variable[20]	<mod>.AddPrivate() AddVar(type=int, name=variable, dimen1=20); <mod>.AddShared() AddVar(type=int, name=variable, dimen1=20); <mod>.AddGlobal(var=variable, dimen1=20) AddField(type=int);
int variable[20][8]	<mod>.AddPrivate() AddVar(type=int, name=variable, dimen1=20, dimen2=8); <mod>.AddShared() AddVar(type=int, name=variable, dimen1=20, dimen2=8); <mod>.AddGlobal(var=variable, dimen1=20, dimen2=8) AddField(type=int);

Table 13 – Declaring Variables

5.5.5 Accessing Variables

The HTL application generates macros to simplify variable access. This section describes the macros the personality can use to access variables of each scope.

5.5.5.1 Private Variable Access

The *P#_name* macro is used to read or write private variables. The private variable can be scalar, an array of scalar variables or an array of memory. The type of the accessed data may be a simple type, a structure or a union.

Private variable read:

```
data = P#_name{ [varidx1] { [varidx2] }}
```

If the private variable has depth the *addr1* and *addr2* values are obtained from the private variable specified in the *AddVar* subcommand of the *AddPrivate* command. The private variable address must be valid on entry to the instruction.

Private variable write:

```
P#_name{ {[addr1] {, [addr2]}} {,[varidx1] {,[varidx2]}} } = data
```

For single stage designs the # is omitted.

When the *AddStage* ***htd*** command has been provided for a module, then multiple clock stages are used to execute instructions. In multistage designs the # is replaced with the instruction execution stage number the variable is to be accessed, with a '1' for the first stage. Private variables can be read or written in each instruction execution stage.

Registered private variables can be read using the *PR#_name* versions of the macros. Registered versions cannot be written. The registered versions are used similarly to the non-registered versions described above.

Registered private variable read:

```
data = PR#_name{ [varidx1] { [varidx2] }}
```

5.5.5.2 Shared Variable Access

Shared variables can be scalar, memory or queue. The HTL application generates macros for each of these.

5.5.5.2.1 Shared Scalar Variables:

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

Shared variable read:

```
data = S_name{ [varidx1] { [varidx2] }}
```

Shared variable write:

```
S_name{ [varidx1] { [varidx2] }} = data
```

Registered shared variables can be read using the *SR_name* versions of the macros. Registered versions cannot be written. The registered versions are used similarly to the non-registered versions described above.

Registered shared variable read:

```
data = SR_name{ [varidx1] { [varidx2] }}
```

5.5.5.2.2 Shared Memory Variable

Shared Memory Variable Read:

Two steps are required to read a shared memory variable. The first step is to specify the address of the memory to be read and the second step is to access the memory to obtain the data value.

The *.read_addr( )* method is used to specify the address of the memory and the *.read_mem( )* method is used to access the read data. The two steps can be separated within the instruction. The *.read_addr( )* call can be made out of a conditional statement and the *.read_mem( )* method can optionally be called based on conditional instruction execution. This approach is frequently advantageous from a logic timing perspective. A simple example is shown in Example 9.

```
void
CPersSrd::PersSrd()
{
       //Set up shared memory read address
       S_ptr1.read_addr(PR_htId);
       S_prt2.read_addr(PR_htId);
       if (PR_htValid){
              switch (PR_HtInst){
              case INST1: {
                     //  Instruction 1 functionality
                     HtContinue(INST2);
              }
              break;
              case INST2: {
                     //  Instruction 2 functionality
                     HtContinue(INST3);
              }
              break;
              case INST3: {
                     //Perform read
                     P_result = S_ptr1.read_mem() + S_ptr2.read_mem();
                     HTContinue(INST4)
              }
              case INST4: {
              //  Instruction 4 functionality
              break;
              }
       . . .
```

Example 9- Shared Memory Variable Read with Distributed RAM

Shared memories implemented as block rams must have the *.read_addr()* method called one clock cycle before the *.read_mem()* method is called as shown in Example 10. This requires a multi-stage instruction. Refer to Section 5.4.4 for more information on multi-stage design.

```
void
CPersSmrd::PersSmrd()
{
      if (PR1_htValid){
            switch (PR1_HtInst){
            case INST1: {
                  //  Instruction 1 functionality
            }
            break;
            case INST2: {
                  //  Instruction 2 functionality
                  //Set up read address
                  S_ptr1.read_addr(PR1_htId);
                  S_prt2.read_addr(PR1_htId);

            }
            break;
            case INST3: {
                  //  Instruction 3 functionality
            }
            break;
            }
      }

      if (PR2_htValid){
            switch (PR2_HtInst){
            case INST1: {
                  //  Instruction 1 functionality
                  HtContinue(INST2);
            }
            break;
            case INST2; {
                  // Read memory
                  P2_result = S_ptr1.read_mem() + S_ptr2.readmem();
                  HTContinue(INST3)
            }
            break;
            case INST3: {
                  //  Instruction 3 functionality
                  //  Return functionality
            }
            break;
      }
}
```

Example 10 – Shared Memory Read with Block RAM

Shared Memory Variable Write:

Two steps are required to write a shared memory variable. The first step is to specify the address of the memory to be written and the second step is write the data value.

The *.write_addr( )* method is used to specify the address of the memory and the *.write_mem( )* method is used to write the data. The two steps can be separated within

the instruction, but must be called in the same clock cycle (same instruction stage). A shared memory variable write is shown in Example 11

```
void
CPersSwr::PersSwr()
{
       //Set up write address
       S_ptr1.write_addr(PR_htId);
       S_prt2.write_addr(PR_htId);
       if (PR_htValid){
              switch (PR_HtInst){
              case INST1: {
                     //  Instruction 1 functionality
                     HtContinue(INST2);
              }
              break;
              case INST2: {
                     //  Instruction 2 functionality
                     HtContinue(INST3);
              }
              break;
              case INST3: {
                     //Write shared variable
                     S_ptr1.write_mem(data1);
                     S_ptr2.write_mem(data2);
                     HtContinue(INST4);
              }
              break;
              }
              case INST4: {
              //  Instruction 4 functionality
       . . .
```

Example 11 –Shared Memory Variable Write

5.5.5.2.3 Shared Queue Variables:

Data can be pushed onto a shared variable queue or popped from a shared variable queue. The HTL application provides API routines in support of both actions.

Shared Queue Variable Push:

Before a value can be pushed on the queue, a check is done to verity there is space available. If space is available the value is pushed on the queue.

There must be room in the queue for the personality to push data on the queue. The *.full()* method is used to determine if a value can be pushed on the queue.

```
bool S_name{ [varidx1] { [varidx2] }}.full();
```

The *.full()* routine returns a *true* if a value cannot be pushed on the queue.

The *.push* method is used to push a value on the queue.

```
void S_name { [varidx1]{ [varidx2] }}.push(data);
```

Example 12 shows a shared queue variable push.

```
void
CPersPush::PersPush()
{

      if (PR_htValid){
            switch (PR_htInst)
            case INST1:
            {
                  //Instruction functionality
                  //If queue is full, retry
                  if ( S_name.full(); )
                  {
                        HtRetry();
                        break;
                  }
                  //Queue is not full, push data
                  S_name.push(data_value);
                  HtContinue(INST2);
            }
            break;
            case INST2:
            {
                  //Instruction 2 functionality
      . . .
```

Example 12 – Shared Queue Variable Push

Shared Queue Variable Pop:

Three routines are provided to read data from a shared variable queue. A shared variable queue can be checked for data. If data is available, the value in the front of the queue can be accessed. After accessing the data, the data can be popped from the front of the queue.

The *.empty()* method is used to determine if a shared variable queue contains data.

```
bool S_name{ [varidx1] { [varidx2] }}.empty();
```

The *.empty()* routine returns a *true* if there is no data in the queue.

After verifying a data is available in the queue, the personality uses the *.front()* method to access the value at front of the queue.

```
data = S_name{ [varidx1] { [varidx2] }}.front();
```

The *.pop()* method is used to advance or remove the data from the front of the queue.

```
S_name{ [varidx1] { {varidx2] }}.pop();
```

Example 13 shows usage of a shared queue variable pop.

```
void
CPersPop::PersPop()
{
       if (PR_htValid){
              switch (PR_htInst)
              case GET_DATA:
              {
                     //Instruction functionality
                     //If queue is full, retry
                     if ( S_name.empty(); )
                     {
                            HtRetry();
                            break;
                     }
                     //Queue contains data
                     data=S_name.front();
                     S_name.pop();
                     HtContinue(NEXT_INST);
              }
              break;
              case NEXT_INST: {

              . . .
```

Example 13 – Shared Queue Variable Pop

5.5.5.3 Global Variable Access

Global variable read:

The *GR#_var_field()* macro is used to read global scalar and memory variables. The global variable may be scalar, an array of scalar variables or an array of memory. The type of the accessed data may be a simple type, a structure or a union.

The address for global memory reads is obtained from the private or temporary stage variable specified in the *AddGlobal* ***htd*** command. If a private variable is used, it must be valid on entry to the instruction, so it is assigned in a previous instruction for the thread. If a temporary stage variable is used, it must be valid on the previous cycle.

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

When the *AddStage* command has been provided for a module, then multiple clock stages may be required to execute instructions. In multistage designs the # is replaced with the instruction execution stage number the variable is to be accessed, with a '1' for the first stage. Global variables can be read during the instruction stage specified by the *rdStg* parameter of the *AddGlobal* command.

The # value is omitted for single stage designs.

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. In multistage designs the # is replaced with the instruction execution stage number the variable is to be accessed, with a '1' for the first stage. Global variables can be written during the instruction stage specified by the *wrStg* parameter of the *AddGlobal* command.

The # value is omitted for single stage designs.

5.6 Creating Threads

A thread can only be created by another module. There are two means to initiate a thread in another module.

- A *Call* begins a thread in the called module, while the thread in the calling module pauses execution until a return is executed.
- A *CallFork* begins a thread in the called module, while the thread in the calling module continues execution.

Both *Call* and *CallFork* begin a thread in an HT function in different module. The following sections provide further information each means to initiate a thread in another module.

5.6.1 Call Functionality

A Call begins a thread in the called module at the entry point defined by the HT function called. The execution of the thread in the calling module is paused, until a return is executed. A return is required for each call

The linkage from the caller's module to the called module is provided with the *AddCall* ***htd*** command in the calling module's description. The *AddCall* command specifies the name of the HT function to be called. The HT function is defined in the called module with the *AddEntry* ***htd*** command. The function called must reside in a different module than the module executing the call.

Before a call can be issued, the personality must assure the call interface can accept a call. The *SendCallBusy* routine is used to determine if a call can be accepted by the target module.

```
SendCallBusy_htfunc()
```

The *SendCallBusy* routine returns a *true* if a new call cannot be sent. If a call cannot be sent, the instruction should use *HtRetry* to retry the instruction

After verifying a call can be sent, the personality uses the *SendCall* routine to send the call.

```
SendCall_htfunc(rsmInst, <callParam1>, <callParam2>, …)
```

The *rsmInst* parameter indicates the instruction executed when the call returns.

The optional call parameters are written to the callee's thread private variables. The call parameters correspond to the *AddParam* subcommands for the called entry point.

5.6.2 CallFork / Join Functionality

A call fork begins a thread in the called module, while the thread in the calling module continues execution. When a return from an call fork is performed, special care must be taken. Since the original thread is still operating, there will be two threads with the same thread ID. A return is required for each call.

The linkage from the caller's module to the called module is provided with the *AddCall* ***htd*** command in the calling module's description. The *AddCall* command specifies the name of the HT function to be called and the *fork* parameter is set to *true* for calls that require the calling thread to continue execution. The HT function is defined in the called module with the *AddEntry* ***htd*** command. The function called must reside in a different module than the module executing the call.

Before a call fork can be issued, the personality must assure the call interface for the indicated function can accept a call. The *SendCallBusy* routine is used to determine if a call can be sent.

```
SendCallBusy_htfunc()
```

The *SendCallBusy* routine returns a *true* if a new call cannot be sent. If a call cannot be sent, the instruction should use *HtRetry* to retry the instruction

After verifying a call can be sent, the personality uses the *SendCallFork* routine to send the call.

```
SendCallFork_htfunc(rsmInst, <callParam1>, <callParam2>, …)
```

The *rsmInst* parameter indicates the instruction to execute when the call returns.

The optional call parameters are written to the callee's thread private variables. The parameters correspond to the *AddParam* subcommands for the called module's entry point.

The *SendCallFork* routing begins a thread in the called module. The thread in the calling module continues execution, often times executing additional *SendCallFork* routines resulting in multiple threads in the called module.

When an call fork returns the first instruction executed (*rsmInst* from the *SendCallFork*) must include a call to the *RecvReturnJoin* routine, which ends the thread.

```
void RecvReturnJoin_htfunc ( )
```

An instruction may only contain a *RecvReturnJoin* for a single *htfunc*.

If there are return parameters they are stored in the calling modules private variables and are valid on entry to the *rsmInst* instruction of the *SendCallFork* routine that initiated the call. Once an HT instruction is executed the private variables containing the return parameters are no longer valid.

The calling module executes the *RecvReturnPause* routine after all call forks to a function have been performed.

```
void RecvReturnPause_htfunc ( rsmInst )
```

The *rsmInst* parameter indicates the instruction to execute when the all calls to *func* have returned.

5.7 Terminating Threads

A return terminates the thread in the called module and resumes the thread in the calling module.

5.7.1 Return Functionality

A return returns control to the calling module and terminates the thread in the called module. The return linkage is provided with the *AddReturn* ***htd*** command. Optional return parameters are designated with the *AddParam* subcommand.

Every ***htd*** *AddEntry* command has a corresponding ***htd*** *AddReturn* command. Likewise each *Call* or *CallFork* routine executed has a corresponding *Return*. The return functionality is independent of how the thread to be returned was created. In other words the function is not aware if the function was entered as a result of a *Call* or *CallFork* from the calling module.

Before a return can be issued, the personality must assure the return interface for the function can accept another return. The *SendReturnBusy* routine is used to determine if a return can be sent.

```
bool SendReturnBusy_htfunc()
```

The *SendReturnBusy* routine returns a *true* if a return cannot be sent. If a return cannot be sent, the instruction should use *HtRetry* to retry the instruction

After verifying a return can be sent, the personality uses the *SendReturn* routine to send the return.

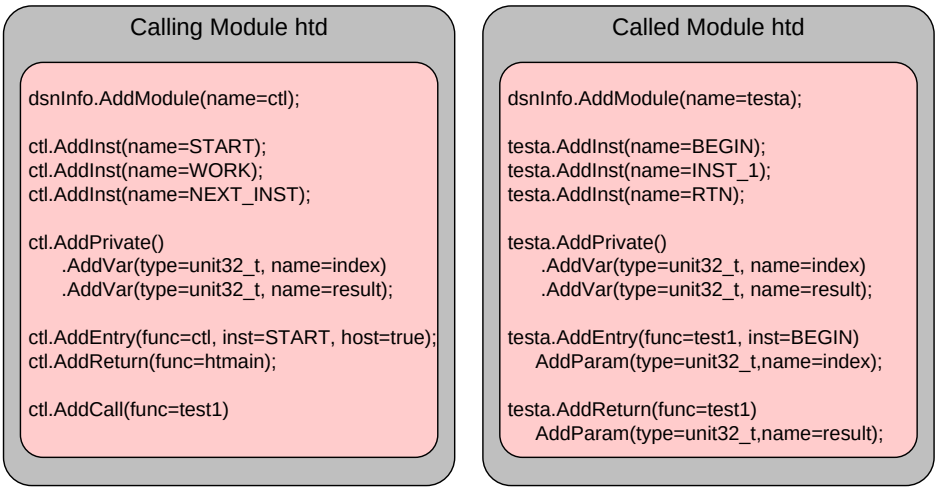
```
void SendReturn_htfunc(<rtnParam1>, <rtnParam2>, …)
```

The SendReturn routing resumes a thread in the original calling module, executing the *rsmInst* indicated in the call. Optional return parameters are written to the caller's thread private variables.

Note: Returns from within switch statements are not supported. Returns from if/else statement are supported as long as the if/else statement is not in a switch statement.

5.7.2 Call / Fork / Return Usage

This section provides pseudo code examples for modules executing Call, CallFork and Return functionality. The ***htd*** for both the calling and the called modules is shown below:



5.7.2.1 Call Usage

Example 14 shows a module named *ctl* that makes a call to the *test1* function of the *testa* module, passing a private variable *P_Index*. The *testa* module has a return parameter named *result*. When a return from the *testa* module is received the *NEXT_INST* instruction is executed.

```
void
CPersCtl::PersCtl()
{
       if (PR_htValid) {
              switch (PR_htInst) {
              case START: {
                     // Entry Point - Initial processing

                     // Execute WORK instruction next
                     HtContinue(WORK);
              }
              break;
              case WORK: {
                     // Check if call interface can accept a new call
                     if (SendCallBusy_test1()) {
                            HtRetry();
                     break;
                     }

                     // Call test1, passing private
                     // variable P_Index
                     // Execute NEXT_INST on return.
                     SendCall_test1(NEXT_INST, P_Index);
              }
              break;
              case NEXT_INST: {

                     // P_result contains return value from test
                     // Whatever you want to happen next
              }
              break;

              . . .
```

Example 14 – Call Functionality

5.7.2.2 CallFork / Join Usage

Example 15 shows the instruction definition for the *ctl* module. The *ctl* module makes multiple call forks to the *test1* function of the *testa* module, passing a private variable *P_Index*. The *test1* function has a return parameter named *result*. When a return from the *test1* function is received, the *JOIN* instruction is executed, terminating the returned thread. When all returns are received the *NEXT_INST* instruction is executed.

The ***htd*** for the modules is the same as the example in Section 5.7.2, with the addition of the *fork* parameter being set *true* in the ***htd*** *AddCall* command of the *ctl* module.

```
void
CPersCtl::PersCtl(){
      if (PR_htValid) {
            switch (PR_htInst) {
            case START: {
                  // Entry Point - Initial processing

                  // Execute WORK instruction next
                  HtContinue(WORK);
            }
            break;
            case WORK: {
                  // Check if call interface can accept a new call
                  if (SendCallBusy_test1()) {
                  HtRetry();
                  break;
                  }
                  if (work to be done) {
                        // Call fork to test1, passing private
                        // variable P_Index
                        // Execute JOIN on return.
                        SendCallFork_test1(JOIN, P_Index);

                        // Processing to determine if further work to
                        be done

                        // Execute WORK instruction next
                        HtContinue(WORK);
                  }
                  else {
                        // No more work to do
                        // Wait for all outstanding calls to return
                        // then execute NEXT_INST instruction
                        RecvReturnPause_test1(NEXT_INST);
                  }
            }
            break;
            case JOIN: {
                  // Return from test1 received
                  //Processing using return parameter P_result
                  RecvReturnJoin_test1();
                  // Return thread is terminated.
                  // Return thread can only be active for one
                  instruction
            }
            break;
            case NEXT_INST: {
                  // Whatever you want to happen next
            }
            break;

            . . .
```

Example 15 – Call Fork Functionality

5.7.2.3 Return Usage

Example 16 shows the instruction definition for the *testa* module. The entry point for the *test1* function is the *BEGIN* instruction. The function has a return parameter called *result*.

```
void
CPersTesta::PersTesta()
{
       if (PR_htValid) {
              switch (PR_htInst) {
              case BEGIN: {
                     // Entry Point - Initial processing

                     // Execute INST_1 instruction next
                     HtContinue(INST_1);
              }
              break;
              case INST_1: {
                     // Instruction 1 processing

                     // Execute RTN instruction next
                     HtContinue(RTN);
              }
              break;
              case RTN: {
                     // Check if return interface can accept a return
                     if (SendReturnBusy_test()) {
                            htRetry();
                            break;
                     }

                     // Send return with return parameter P_result)
                     SendReturn_test(P_result);
              }
              break;

       }
}
```

Example 16 – Return Functionality

5.8 Module Message Interface

Messages are sent and received from other modules, using named message interfaces. A module can send outbound messages to and/or receive inbound messages from other modules. Message interfaces are defined with the *AddMsgIntf* ***htd*** command. The ***htd*** command assigns a *name* to each message. Module to module messages can be simple types, structures or unions.

Messages between replicated modules, between modules in different units and other less common module message scenarios are addressed under advance topics in Section 5.14.

5.8.1 Outbound Module Messages

Outbound module messages are used to send a message to another module in the unit. Before sending an outbound message, the personality must assure the message

interface can accept a new message. The *SendMsgBusy* routine is used to determine if a message can be sent.

```
bool SendMsgBusy_<name>()
```

The *SendMsgBusy* routine returns a *true* if a new message cannot be sent. If a new message cannot be sent, the instruction should use *HtRetry* to retry the instruction

After verifying a message can be sent, the personality uses the *SendMsg* routine to send the message.

```
void SendMsg_<name>(<msg>)
```

The *msg* parameter indicates the data to be sent.

5.8.1.1 Outbound Message Usage

Example 17 shows a module executing the instruction *SEND_MSG* which sends an outbound message named *testmsg* to another module. The next instruction the thread will execute is *NEXT_INST*.

```
void
CPersOmsg::PersOmsg()
{
	if (PR_htValid){
		switch (PR_htInst)
		case SEND_MSG:
		{
			//If send message interface is not available, retry
			if ( SendMsgBusy_testmsg() )
			{
				HtRetry();
				break;
			}
			//Send message interface is available, send message
			SendMsg_testmsg(data);
			HtContinue(NEXT_INST);
		}
		break;
		case NEXT_INST;
		{

		. . .
```

Example 17 – Outbound Message Functionality

5.8.2 Inbound Module Messages

Inbound module messages are used to receive a message from another module in the unit. The *RecvMsgReady* routine is used to determine if the *name* message interface contains a received message.

```
bool RecvMsgReady_<name>()
```

The *RecvMsgBusy* routine returns a *true* if a new message is available.

After verifying a message is available, the personality uses the *RecvMsg* routine to access the message data.

```
<type> RecvMsg_<name>()
```

5.8.2.1 Inbound Message Usage

Example 18 shows a module executing the instruction *RECV_MSG* which receives an inbound message named *testmsg* from another module. The next instruction the thread will execute is *NEXT_INST*.

```
void
CPersImsg::PersImsg()
{
       if (PR_htValid){
              switch (PR_htInst)
              case RECV_MSG:
              {
                     // If testmsg available push on the share queue
                     //  queMsg
                     if (RecvMsgReady_testmsg() )
                     {
                            S_queMsg.push(RecvMsg_testmsg() );
                            HtContinue(NEXT_INST);
                     }

              . . .
```

Example 18 – Inbound Message Functionality

5.9 Host Message Interface

A module can send outbound messages to and/or receive inbound messages from the host. Each host message is defined with the *.AddHostMsg* ***htd*** command. The ***htd*** command specifies a *name* for each message.

Inbound host messages are assigned a destination using the ***htd*** *AddDst* subcommand. The destination is the shared variable(s) that receive the data.

5.9.1 Outbound Host Messages

Outbound host messages are used to send a message to the host. Before sending an outbound message, the personality must assure the message interface can accept a new message. The *SendHostMsgBus*y routine is used to determine if a message can be sent.

```
bool SendHostMsgBusy()
```

The *SendHostMsgBusy* routine returns a *true* if a new message cannot be sent and the instruction should use HtRetry to retry the instruction.

After verifying a message can be sent, the personality uses the *SendHostMsg* routine to send the message.

```
void SendHostMsg(<msg_name>, <msg_data>)
```

The *msg_name* parameter indicates the name of the message to be sent.

The *msg_data* parameter contains the data to be sent. Message data is a maximum of 56 bits.

5.9.1.1 Outbound Host Message Usage

Example 19 shows a module executing the instruction *SEND_HOST_MSG* which sends an outbound message named *TEST_MSG* to the host. The next instruction the thread will execute is *NEXT_INST*.

```
void
CPersOhostm::PersOhostm()
{
       if (PR_htValid){
              switch (PR_htInst)
              case SEND_HOST_MSG:
              {
                     //If send message interface is not available, retry
                     if ( SendHostMsgBusy() )
                     {
                            HtRetry();
                            break;
                     }
                     //Send message interface is available, send message
                     SendHostMsg(TEST_MSG,0x123456789abcdeULL);
                     HtContinue(NEXT_INST);
              }

              . . .
```

Example 19 – Outbound Host Message Functionality

5.9.2 Inbound Host Messages

Inbound host messages are written directly to the shared variable(s) indicated in the ***htd*** *AddDst* subcommand. No action is required by the HT instructions.

Inbound host messages are often used to initialize shared variables prior to calling the coprocessor routine. Typically inbound host messages are not used when a call is active.

5.10 Host Data Interface

The host data interface is a streaming interface which can be used to send or receive blocks of data. A single module within a unit can send outbound data to and/or receive inbound data from the host. A module requires a host data interface to receive inbound host data or send outbound host data. Host data interfaces are defined with the *AddHostData* ***htd*** command. The ***htd*** command specifies if the interface is an inbound or outbound interface.

Inbound and outbound data is sent when anyone of the following occurs:

- A data block is full (64K block)
- 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 default timer value for both inbound and outbound data is 1000 used. This value can be modified with a HIF parameter when constructing the HIF (see Section 3.1.1).

Markers can be used in the streaming interface to delineate start or end of data blocks.

When inbound host data is transferred outside of a call, the data must be serviced or calls can be blocked.

5.10.1 Outbound Host Data

Outbound host data is used to send a block of data to the host. Before sending outbound host data, the personality must assure the data interface has space to accept data. The *SendHostDataBusy* routine is used to determine if a data or a data marker can be sent.

```
bool SendHostDataBusy()
```

The *SendHostDataBusy* routine returns a *true* if a new message cannot be sent. The instruction uses *HtRetry* to retry the instruction if the routine evaluates to *true*.

After verifying the interface can accept data, the personality uses the *SendHostData* routine to send the data.

```
void SendHostData( unit64_t data)
```

The *data* parameter specifies the 64 bit data value to be sent.

If outbound data markers are used the *SendHostDataMarker* routine is used.

```
void SendHostDataMarker()
```

Blocks of data are sent using multiple *SendHostData* routine calls.

5.10.1.1 Outbound Host Data Usage

The examples in this section show instructions sending outbound host data. The first example sends just host data and the second example sends data, preceded by a data marker.

Example 20 shows a module executing the instruction *SEND_DATA* which sends an outbound data to the host. The next instruction the thread will execute is *NEXT_INST*.

```
void
CPersOhostd::PersOhostd()
{
       if (PR_htValid){
              switch (PR_htInst)
              case SEND_DATA:
              {
                     //If send data interface is not available, retry
                     if SendHostDataBusy()
                     {
                            HtRetry();
                            break;
                     }
                     //Send data interface is available, send data
                     SendHostData(data);
                     HtContinue(NEXT_INST);
              }

              . . .
```

Example 20 – Outbound Host Data Functionality

Example 21 shows a module executing the instruction *SEND_MARKER* which sends outbound host data marker. The next instruction executed is *SEND_DATA*, which sends outbound host data. The next instruction the thread will execute is *NEXT_INST*.

```
void
CPersOhostdm::PersOhostdm()
{
       if (PR_htValid){
              switch (PR_htInst)
              case SEND_MARKER:
              {
                     //If send data interface is busy, retry
                     if ( SendHostDataBusy() )
                     {
                            HtRetry();
                            break;
                     }
                     //Send data interface is available, send marker
                     SendHostDataMarker();
                     HtContinue(SEND_DATA);
              }
              case SEND_DATA
              {
                     //If send data interface is busy, retry
                     if ( SendHostDataBusy() )
                     {
                            HtRetry();
                            break;
                     }
                     //Send data interface is available, send data
                     SendHostData(data);
                     HtContinue(NEXT_INST);
              }
              case NEXT_INST

              . . .
```

Example 21 – Outbound Host Data / Marker Functionality

Since the outbound marker and the outbound host data both use the outbound host interface they must be sent in separate instructions.

5.10.2 Inbound Host Data

Inbound host data is used to receive a block of data from the host. The *RecvMsgBusy* routine is used to determine if host data or a data marker is available.

```
bool RecvHostDataBusy()
```

The *RecvHostDataBusy* routine returns a *true* if a data or a marker is available. The instruction uses *HtRetry* to retry the instruction if data is not available.

After verifying host data or a host data marker is available, the personality uses the *RecvHostData* routine to access the data.

```
unit64_t RecvHostData()
```

The returned value is the 64 bit received data.

If using data markers, the *RecvHostDataMarker* routine must be called prior to calling *RecvHostData.*

```
bool RecvHostDataMarker()
```

The *RecvHostDataMarker* routine returns true if a data marker is present.

5.10.2.1 Inbound Host Data Usage

The examples in this section show instructions receiving inbound host data. The first example uses host data without markers and the second example receives data, preceded by a data marker.

Example 22 shows a module executing the instruction *RECV_DATA* which receives inbound host data. The next instruction the thread will execute is *NEXT_INST*.

```
void
CPersInhostd::PersInhostd()
{
       if (PR_htValid){
              switch (PR_htInst)
              case RECV_DATA:
              {
                     //If receive data is not available, retry
                     if (!RecvHostDataBusy () )
                     {
                            HtRetry();
                            break;
                     }
                     //Receive data is available, receive data
                     unit64_t data = RecvHostData();
                     HtContinue(NEXT_INST);
              }
              break;
              case NEXT_INST:

              . . .
```

Example 22 – Inbound Host Data Functionality

Example 23 shows a module looking for host data, preceded by a marker. The module executes the instruction *RECV_MARKER* which receives an inbound host data marker. The next instruction executed is *RECV_DATA*, which receives inbound host data. The *RecvHostDataMarker* routine must be called prior to calling *RecvHostData* and *RecvHostData* should only be called if *RecvHostDataMarker* returns false. The next instruction the thread will execute is *NEXT_INST*.

```
void
CPersHostdm::PersHostdm()
{
       if (PR_htValid){
              switch (PR_htInst)
              case RECV_MARKER:
              {
                     //If receive data is not available, retry
                     if ( !RecvHostDataBusy() )
                     {
                            HtRetry();
                            break;
                     }
                     //Receive data is available, receive marker
                     if RecvHostDataMarker()
                            HtContinue(RECV_DATA);
              }
              case RECV_DATA
              {
                     //If receive data is not available, retry
                     if ( !RecvHostDataBusy() )
                     {
                            HtRetry();
                            break;
                     }
                     //If marker, retry
                     if RecvHostDataMarker();
                            HtRetry();
                            break;
                     //Receive data available and not marker, receive data
                     unit64_t data = RecvHostData();
                     HtContinue(NEXT_INST);
              }
              case NEXT_INST:

              . . .
```

Example 23 – Inbound Host Data Preceded by a Marker

Since the inbound marker and the inbound host data both use the inbound host interface they must be received in separate instructions.

5.11 Memory Access

A module can read or write host or coprocessor memory. The memory subsystems on some platforms are optimized for 64 byte accesses. The HT tools will convert memory accesses to the type of access supported by the platform used.

In order to optimize performance and develop personalities that perform across platforms, it is recommended that data be arranged on 64 byte boundaries when possible and accessed in 64 byte accesses.

5.11.1 Read Memory

A read memory interface is added to a module using the *AddReadMem* ***htd*** command. The destination of the read data is specified using the *AddDst* subcommand.

If the memory interface for a destination requires multiple 8 byte reads with a single read request, the *multiRd* parameter of the *AddDst* subcommand is set to *true* and the index of the destination variable is set using the *dstIdx* parameter.

If the memory interface for a destination requires support of quadword or sub quadword reads the *AddDst* subcommand enables the capability with the *rdType* parameter. This parameter is set to the required type. Valid values are uint8, uint16, uint32, uint64, int8, int16, int32 and int64.

Before sending a memory read request, the personality must assure the memory interface can accept the request. The *ReadMemBus*y routine is used to determine if a message can be sent.

```
ReadMemBusy()
```

The *ReadMemBusy* routine returns a *true* if a new message cannot be sent and the instruction should use *HtRetry* to retry the instruction.

After verifying the read memory interface is available, the personality uses the *ReadMem_<name>* routine to initiate a memory read. The memory read response is written to the destination variable specified in the *AddDst* ***htd*** command with the matching *name* parameter.

```
ReadMem_name(memAddr, varAddr1, varAddr2, varldx1, varldx2,
fldldx1, fldldx2, qwCnt)
```

The *memAddr* parameter indicates 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..

The *varAddr1* 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).

The *varAddr2* 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).

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

The *varIdx2* parameter 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.

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

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

The *qwCnt* parameter specifies the number of quadwords (8 byte quantities) that are accessed by the *ReadMem* routine.

If the read access is a 64 byte access, 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.

5.11.1.1 Handling Memory Read Latency

Two routines are provided to handle read memory latency, *ReadMemPause* and *ReadMemPoll*.

The *ReadMemPause* routine pauses the executing thread until all outstanding memory read requests for the thread or defined group of responses (response group) have completed.

```
ReadMemPause(rsmInst)
```

The *rsmInst* parameter specifies the resume instruction, which is the next instruction to execute once outstanding requests have completed.

Before executing a *ReadMemPause* routine, the personality must execute the *ReadMemBus*y routine.

Many threads can be paused at the same time, but each thread must be paused waiting on a different thread or defined group of responses (response group). A memory read request may be issued in the same instruction prior to call the *ReadMemPause* routine. In this case, the thread will wait for the just issued memory read request to complete along with all other outstanding reads from the response group. Memory read requests should not be called in the same instruction following a *ReadMemPause* call.

The *ReadMemPoll* routine indicates whether the indicated response group has any outstanding requests. A true response indicates that the memory read requests have not completed.

```
bool ReadMemPoll(rspGrpId)
```

The *rspGrpId* parameter specifies the read response group, defined by the rspGrpId parameter of the AddReadMem ***htd*** command. If *rspGrpI*d is not specified, htId is implied

The routine is primarily used when multiple threads must wait for all requests on a specific response group to complete. This routine may also be used when a thread has other work to do while waiting for a memory read to complete. Note that *ReadMemPause* is the default method. If *ReadMemPoll* enabled, the ***htd*** *pause* default is changed to false. If both ReadMemPause and ReadMemPoll are desired, both must be set to true.

The *ReadMemPoll* routine should not be called in the same instruction after issuing a memory read request. The routine does not have visibility of memory read requests issued in the same instruction.

5.11.2 Write Memory

A write memory interface is added to a module using the *AddWriteMem* ***htd*** command. Most platforms are optimized for quad word writes, so it is recommended to utilize quad

word or multi-quad word writes when possible. The HT Tools will implement the optimal type of access for the platform.

If multi-quadword writes are used, the source variable is defined with an *AddSrc* ***htd*** subcommand. The *multiWr* parameter must be set and the index of the source variable is set using the *srcIdx* parameter. Valid values for *srcidx* are *varAddr1, varAddr2, varIdx1, varIdx2, fldIdx1 or fldIdx2*.

The first write request is sourced by the source variable specified by the *WriteMem_name(…)* routine. Subsequent requests are source by the source variable with the specified *srcIdx* incremented by one for each 8-byte request.

Before sending a memory write request, the personality must assure the memory interface can accept a new write request. The *WriteMemBus*y routine is used to determine if a message can be sent.

```
bool WriteMemBusy()
```

The *WriteMemBusy* routine returns a *true* if a new message cannot be sent and the instruction should use *HtRetry* to retry the instruction.

5.11.2.1 64 Bit or Sub 64 Bit Writes

After verifying the write memory interface is available, the personality uses the *WriteMem* routine to initiate a single memory write request of size *type*.

```
void WriteMem_<type>(memAddr, memData)
```

Valid values for *type* are *uint8, uint16, uint32, uint64, int8, int16, int32* and *int64*.

The *memAddr* parameter indicates the 48-bit memory request address. The address must be 8-byte aligned

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

5.11.2.2 Multi-Quadword Writes

For multi-quadword writes, after verifying the write memory interface is available, the personality uses the *WriteMem_<name>* routine to initiate a memory write. The source of the data is specified in the *AddSrc* ***htd*** command with the matching *name* parameter.

```
void WriteMem_<name>(memAddr, varAddr1, varAddr2, varidx1,
varidx2, fldidx1, fldidx2, qwCnt)
```

The *memAddr* parameter indicates 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..

The *varAddr1* parameter specifies the entry within a source memory variable where the data is provided. 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).

The *varAddr2* parameter specifies the entry within a source memory variable where the data is provided. 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).

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

The *varIdx2* parameter 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.

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

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

The *qwCnt* parameter specifies the number of quadwords (8 byte quantities) that are accessed by the *WriteMem_<name>* routine.

5.11.2.3 Handling Memory Write Latency

Three routines are provided to handle write memory latency, *WriteMemPause*, *WriteReqPause* and *WriteMemPoll*. These routines are available when the corresponding *pause, reqPause* and *poll* parameters are set in the *AddWriteMem* ***htd*** command. *WriteMemBusy* is called before each of these routines. In some cases a memory write request occurs in the same cycle. Since a *WriteMemBusy* is required before the write request an additional *WriteMemBusy* is not needed.

The *WriteMemPause* routine pauses the executing thread until all outstanding memory write requests for the thread or defined group of responses (response group) have completed.

```
void WriteMemPause(rsmInst)
```

The *rsmInst* parameter specifies the resume instruction, which is the next instruction to execute once outstanding write requests have completed.

Many threads can be paused at the same time, but each thread must be paused waiting on a different response group. A memory write request may be issued in the same instruction prior to call the *WriteMemPause* routine. In this case, the thread will wait for the just issued memory write request to complete along with all other outstanding write requests from the response group. Memory write requests should not be called in the same instruction following a *WriteMemPause* call.

The *WriteReqPause* routine pauses the executing thread until all outstanding memory write requests for the response group have been initiated. This allows the user to update variables used by the write request in preparation for the next write, without waiting for the request to complete.

```
void WriteReqPause(rsmInst)
```

The *rsmInst* parameter specifies the resume instruction, which is the next instruction to execute once outstanding write requests have been initiated.

Many threads can be paused at the same time, but each thread must be paused waiting on a different response group. The *WriteReqPause* routine must be called in the same instruction that issues the last memory write for the response group. In this case, the thread will wait for the just issued memory write request to complete along with all other outstanding write requests from the response group. Memory write requests should not be called in the same instruction following a *WriteReqPause* call.

The *WriteMemPoll* routine indicates whether the response group has any outstanding requests. A true response indicates that the memory write requests have not completed.

```
bool WriteMemPoll()
```

The routine is primarily used when multiple threads must wait for all requests on a specific response group to complete. This routine may also be used when a thread has other work to do while waiting for a memory writes to complete. Note that *WriteMemPause* is the default method. If *WriteMemPoll* is preferred the ***htd*** *pause* parameter should be set to *false* and the ***htd*** *poll* parameter set to *true*.

The *WriteMemPoll* routine should not be called in the same instruction after issuing a memory write request, since the routine does not have visibility of memory write requests issued in the same instruction. *WriteMemBusy* must be called prior to calling *WriteMemPoll*.

5.11.3 Memory Read and Write Usage

Example 24 shows memory read and write functionality. The module executes the instructions *LD_DATA_1* and *LD_DATA_2,* which read data from memory. After the read requests are complete the thread executes the *WORK* instruction, which adds the data and writes the result to memory. When the write is complete the thread executes the *NEXT_INST* instruction.

```
void
CPersMemrw::PersMemrw()
{
       if (PR_htValid)
       {
              switch (PR_htInst)
              {
              case LD_DATA_1:{
                     //Can read memory interface accept a request?
                     if ReadMemBusy() ){
                            HtRetry();
                            break;
                     }
                     //Read data
                     ReadMem_data1(memRdAddr1, PRhtId);
                     HtContinue(LD_DATA_2);
              }
              break;
              case LD_DATA_2:{
                     //Can read memory interface accept a request?
                     if ReadMemBusy() ){
                            HtRetry();
                            break;
                     }
                     //Read data
                     ReadMem_data2(memRdAddr2, PRhtId);
                     //Pause until reads are complete then go to ADD
                     ReadMemPause(WORK);
              }
              break;
              case WORK:{
                     //Can write memory interface accept a request?
                     if (WriteMemBusy() )
                     {
                            HtRetry();
                            break;
                     }
                     //Calculate result from S_data1 and S_data2
                     //Write result to memory
                     WriteMem_int64(memWrAddr, P_sum);
                     //Pause until write complete then execute NEXT_INST
                     WriteMemPause(NEXT_INST);
              }
              break;
              case NEXT_INST:{
                     //
              {
              break;
              }

              . . .
```

Example 24 – Memory Read / Write Functionality

5.12 Streaming Memory Access

Streaming memory interfaces provide a more efficient means to implement a string of memory accesses than individual memory accesses. A module can read or write a stream to or from host or coprocessor memory.

A stream interface is typically controlled in the instruction functionality and data is read from or written to the stream in non-instruction functionality.

5.12.1 Read Stream

A read stream interface is defined using the *AddReadStream* ***htd*** command. The primary stream characteristics included in the ***htd*** command are

- Name of the stream or unnamed stream
- Type of stream elements
- Number of independent streams
- Maximum number of elements in the stream
- How the stream is closed
- Memory ports associated with the stream

Typically a read stream is controlled in the instruction functionality and accessed in non-instruction functionality.

5.12.1.1 Read Stream Control

Before opening a read stream, the personality assures the stream is not opened. One of the *ReadStreamBusy* routines is used to determine if the stream(s) can be opened. These routines return true if the read stream(s) is open and false if the read stream(s) is closed and can be re-opened. The *ReadStream Busy* routines are available for unnamed, named streams and for multiple streams.

`ReadStreamBusy({strmId});`	unnamed stream
`ReadStreamBusy<_`*`name`*`>({strmId});`	named streams
`ReadStreamBusyMask<_`*`name`*`>({strmMask});`	multiple streams

The *strmId* specifies the stream to check for open status. Valid values are dependent on the *strmCnt* parameter of the *AddReadStream* ***htd*** command and range from *0 to strmCnt-1*. The *strmId* is *log2(strmCnt)* bits in width. The *strmId* parameter is not present if the *AddReadStream strmCnt* ***htd*** parameter is not specified. If the *AddReadStream strmCnt* parameter is 1, the *strmId* is zero.

The *strmMask* parameter specifies the streams to check for open status. Bit zero of the mask is associated with *strmId* 0, etc. The *ReadStreamBusyMask* routine is only available if the *AddReadStream strmCnt* ***htd*** parameter is specified.

The *ReadStreamBusy* routines return a *true* if the stream(s) cannot be opened and the instruction should use *HtRetry* to retry the instruction.

After verifying the read stream (s) can be opened, the personality uses the appropriate *ReadStreamOpen* routine to open the read stream.

`ReadStreamOpen({strmId, }addr{, elemCnt}{ ,tag});`	unnamed

ReadStreamOpen<_*name*>({strmId, }addr{, elemCnt}{ ,tag}); named

The *strmId* specifies the stream to open. Valid values are dependent on the *strmCnt* parameter of the *AddReadStream* ***htd*** command and range from *0 to strmCnt-1*. The *strmId* is *log2(strmCnt)* bits in width. The *strmId* parameter is not present if the *AddReadStream strmCnt* ***htd*** parameter is not specified. If the *AddReadStream strmCnt* parameter is 1, the *strmId* is zero.

The *addr* parameter specifies the 48 bit starting address for the first element to be accessed.

The *elemCnt* parameter determines how many stream elements can be accessed. If the *close* parameter of the *AddlReadStream* ***htd*** command is *false* (default), the read stream is closed after *elemCnt* elements are accessed if *true*, *elemCnt* specifies the maximum number of elements that can be accessed, but the stream is not closed. Prefetching of elements stops once the *elemCnt* value is reached. The *elemCnt* parameter is *elemCntW* bits wide.

The *tag* parameter specifies the tag data associated with the read stream. The data is read with the *ReadStreamTag* routine.

The *AddReadStream* ***htd*** command *close* parameter determines how a read stream is closed. If this parameter is set to false, 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 after *elemCnt* elements, but the stream will stay open until the appropriate *ReadStreamClose* routine is executed.

ReadStreamClose ({strmId}); unnamed stream

ReadStreamClose<_*name*>({strmId}); named stream

The *strmId* specifies the stream to close. Valid values are dependent on the *strmCnt* parameter of the *AddReadStream* ***htd*** command and range from *0 to strmCnt-1*. The *strmId* is *log2(strmCnt)* bits in width. The *strmId* parameter is not present if the *AddReadStream strmCnt* ***htd*** parameter is not specified. If the *AddReadStream strmCnt* parameter is 1, the *strmId* is zero.

5.12.1.2 Accessing Read Streams

Once the stream is opened the stream is typically accessed in non-instruction functionality, which occurs every clock cycle.

The *ReadStreamReady* routines are used to determine if the read stream is able to provide the next stream element. These routines return true if the next element is available and false otherwise. *ReadStreamReady* routines are available for unnamed, named and multiple read streams.

ReadStreamReady({strmId}); unnamed stream

ReadStreamReady<_*name*>({strmId}); named stream

ReadStreamReadyMask<_*name*>({strmMask}); multiple streams

The *strmId* specifies the stream to check for the next element to be ready. Valid values are dependent on the *strmCnt* parameter of the *AddReadStream* ***htd*** command and range from *0 to strmCnt-1*. The *strmId* is *log2(strmCnt)* bits in

width. The *strmId* parameter is not present if the *AddReadStream strmCnt* ***htd*** parameter is not specified. If the *AddReadStream strmCnt* parameter is 1, the *strmId* is zero.

The *strmMask* parameter specifies the streams to check for open status. Bit zero of the mask is associated with *strmId* 0, etc. The *ReadStreamReadyMask* routine is only available if the *AddReadStream strmCnt* ***htd*** parameter is specified.

Once it is determined the read stream is ready to provide the next element, the application can return the next element with or without advancing the stream. The *ReadStream* routines are used to return the next element advancing the stream and the *ReadStreamPeek* routines are used to return the next element without advancing the stream. Both types of routines are available for unnamed and named read streams.

`ReadStream ({strmId});`	unnamed stream
`ReadStream<_name>({strmId});`	named stream
`ReadStreamPeek({strmId});`	unnamed stream
`ReadStreamPeek<_name>({strmId});`	named stream

The *strmId* specifies the stream to return the next element from. Valid values are dependent on the *strmCnt* parameter of the *AddReadStream* ***htd*** command and range from *0 to strmCnt-1*. The *strmId* is *log2(strmCnt)* bits in width. The *strmId* parameter is not present if the *AddReadStream strmCnt* ***htd*** parameter is not specified. If the *AddReadStream strmCnt* parameter is 1, the *strmId* is zero.

The *ReadStreamLast* routines are used to determine if the current read element is the last element in the read stream. These routines return true if the current element is the last element in the stream. *ReadStreamLast* routines are available for unnamed and named read streams.

`ReadStreamLast ({strmId});`	unnamed stream
`ReadStreamLast<_name>({strmId});`	named stream

The *strmId* specifies the stream to check for last element. Valid values are dependent on the *strmCnt* parameter of the *AddReadStream* ***htd*** command and range from *0 to strmCnt-1*. The *strmId* is *log2(strmCnt)* bits in width. The *strmId* parameter is not present if the *AddReadStream strmCnt* ***htd*** parameter is not specified. If the *AddReadStream strmCnt* parameter is 1, the *strmId* is zero.

The *ReadStreamTag* routines are used to read the tag associated with the stream. *ReadStreamTag* routines are available for unnamed and named read streams.

`ReadStreamTag ({strmId});`	unnamed stream
`ReadStreamTag<_name>({strmId});`	named stream

The *strmId* specifies the read the tag from. Valid values are dependent on the *strmCnt* parameter of the *AddReadStream* ***htd*** command and range from *0 to strmCnt-1*. The *strmId* is *log2(strmCnt)* bits in width. The *strmId* parameter is not present if the *AddReadStream strmCnt* ***htd*** parameter is not specified. If the *AddReadStream strmCnt* parameter is 1, the *strmId* is zero.

5.12.2 **Write Stream**

A write stream interface is defined using the *AddWriteStream* ***htd*** command. The primary stream characteristics included in the ***htd*** command are

- Name of the stream
- Type of stream elements
- Number of independent streams
- Maximum number of elements in the stream
- How the stream is closed
- Memory ports associated with the stream

Typically a write stream is controlled in the instruction functionality and written in non-instruction functionality.

5.12.2.1 Write Stream Control

Typically write streams are controlled in the instruction functionality of a module.

Before opening a write stream, the personality assures the stream is not opened. One of the *WriteStreamBusy* routines is used to determine if the stream(s) can be opened. These routines return true if the read stream(s) is open and false if the read stream(s) is closed and can be re-opened

```
WriteStreamBusy({strmId});                          unnamed stream

WriteStreamBusy<_name>({strmId});                   named streams

WriteStreamBusyMask<_name>({strmMask});             multiple streams
```

The *strmId* specifies the stream to check for open status. Valid values are dependent on the *strmCnt* parameter of the *AddWriteStream* ***htd*** command and range from *0 to strmCnt-1*. The *strmId* is *log2(strmCnt)* bits in width. The *strmId* parameter is not present if the *AddWriteStream strmCnt* ***htd*** parameter is not specified. If the *AddWriteStream strmCnt* parameter is 1, the *strmId* is zero.

The *strmMask* parameter specifies the streams to check for open status. Bit zero of the mask is associated with *strmId* 0, etc. The *WriteStreamBusyMask* routine is only available if the *AddWriteStream strmCnt* ***htd*** parameter is specified.

The *WriteStreamBusy* routines return a *true* if the stream(s) cannot be opened and the instruction should use *HtRetry* to retry the instruction.

After verifying the write stream(s) can be opened, the personality uses the appropriate *WriteStreamOpen* routine to open the read stream.

```
WriteStreamOpen({strmId, }addr {, elemCnt}{, rspGrpId});

WriteStreamOpen<_name>({strmId, }addr {, elemCnt}{, rspGrpId);
```

The *strmId* specifies the stream to open. Valid values are dependent on the *strmCnt* parameter of the *AddWriteStream* ***htd*** command and range from *0 to strmCnt-1*. The *strmId* is *log2(strmCnt)* bits in width. The *strmId* parameter is not present if the *AddWriteStream strmCnt* ***htd*** parameter is not specified. If the *AddWriteStream strmCnt* parameter is 1, the *strmId* is zero.

The *addr* parameter specifies the 48 bit starting address for the first element to be written.

The *elemCnt* parameter determines how many stream elements can be written. If the *close* parameter of the *AddlWriteStream* ***htd*** command is *false* (default), the write stream is closed after *elemCnt* elements are written if *true*, the *elemCnt* parameter is not present. The *elemCnt* parameter is *elemCntW* bits wide.

The *rspGrpId* parameter specifies the response group. If *the AddWriteStream rspGrpW* parameter is not specified the *rspGrpId* parameter is not present and the thread ID is used.

Once the stream is opened the stream is typically paused using the *WriteStreamPause* routines and the stream is written to in non-instruction functionality. Only one thread can pause on a specific response group at a time. Multiple threads can be pause each on a different response group. By default the response group is the thread ID.

```
WriteStreamPause({strmId, }nextInstr{, rspGrpId});

WriteStreamPause<_name>({strmId, }nextInstr{, rspGrpId});
```

The *strmId* specifies the stream to pause. Valid values are dependent on the *strmCnt* parameter of the *AddWriteStream* ***htd*** command and range from *0 to strmCnt-1*. The *strmId* is *log2(strmCnt)* bits in width. The *strmId* parameter is not present if the *AddWriteStream strmCnt* ***htd*** parameter is not specified. If the *AddWriteStream strmCnt* parameter is 1, the *strmId* is zero.

The *nextInstr* parameter specifies the next instruction for the thread to execute, once the specified stream has been completely written to memory.

The *rspGrpId* parameter specifies the response group. If *the AddWriteStream rspGrpW* parameter is not specified the *rspGrpId* parameter is not present and the thread ID is used.

The *AddWriteStream* ***htd*** command *close* parameter determines how a write stream is closed. If this parameter is set to false, the stream will automatically close after *elemCnt* elements are written. 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 appropriate *WriteStreamClose* routine is executed.

```
WriteStreamClose ({strmId});                    unnamed stream

WriteStreamClose<_name>({strmId});              named stream
```

The *strmId* specifies the stream to close. Valid values are dependent on the *strmCnt* parameter of the *AddWriteStream* ***htd*** command and range from *0 to strmCnt-1*. The *strmId* is *log2(strmCnt)* bits in width. The *strmId* parameter is not present if the *AddWriteStream strmCnt* ***htd*** parameter is not specified. If the *AddWriteStream strmCnt* parameter is 1, the *strmId* is zero.

5.12.2.2 Writing Write Streams

Once the stream is opened the stream is typically written in non-instruction functionality, which occurs every clock cycle.

The *WriteStreamReady* routines are used to determine if the write stream(s) is able to accept the next stream element. These routines return true if the next element can be accepted and false otherwise. WriteStreamReady routines are available for unnamed, named and multiple write streams.

`WriteStreamReady({strmId});`	unnamed stream
`WriteStreamReady<_name>({strmId});`	named stream
`WriteStreamReadyMask<_name>({strmMask});`	multiple streams

The *strmId* specifies the stream to check for element ready. Valid values are dependent on the *strmCnt* parameter of the *AddWriteStream* ***htd*** command and range from *0 to strmCnt-1*. The *strmId* is *log2(strmCnt)* bits in width. The *strmId* parameter is not present if the *AddWriteStream strmCnt* ***htd*** parameter is not specified. If the *AddWriteStream strmCnt* parameter is 1, the *strmId* is zero.

The *strmMask* parameter specifies the streams to check for ready. Bit zero of the mask is associated with *strmId* 0, etc. The *WriteStreamReadyMask* routine is only available if the *AddWriteStream strmCnt* ***htd*** parameter is specified.

Once it is determined the write stream(s) is ready to accept the next element, the application can write the next element to the stream, using the appropriate *WriteStream* routine.

`WriteStream ({strmId, }elemData);`	unnamed stream
`WriteStream<_name>({strmId, }elemData);`	named stream

The *strmId* specifies the stream to write the next element to. Valid values are dependent on the *strmCnt* parameter of the *AddWriteStream* ***htd*** command and range from *0 to strmCnt-1*. The *strmId* is *log2(strmCnt)* bits in width. The *strmId* parameter is not present if the *AddWriteStream strmCnt* ***htd*** parameter is not specified. If the *AddWriteStream strmCnt* parameter is 1, the *strmId* is zero.

5.12.3 Read and Write Stream Usage

Example 25 shows read and write stream functionality. This example performs a vector add, using streams. The module checks to see if read streams A and B and write stream C are open. If they are not open the streams are opened. The write stream C is paused to wait for writes to the stream to complete.

In the non-instruction functionality if the read streams have an element ready and the write stream can accept an element, the elements from the read streams are read, the elements are added together and the sum is written to the write stream. When all elements are complete the *WriteStreamPause* routine provides the next instruction to execute (*VADD_RETURN*).

```
#include "Ht.h"
#include "PersVadd.h"

#define BUSY_RETRY(b) {if (b) {HtRetry(); break;}}

#ifndef min
#define min(a,b) (((a)<(b)) ? (a) : (b))
#endif

void
CPersVadd::PersVadd()
{
      if (PR_htValid) {
            switch (PR_htInst) {
            case VADD_ENTER: {
                  BUSY_RETRY(ReadStreamBusy_A());
                  BUSY_RETRY(ReadStreamBusy_B());
                  BUSY_RETRY(WriteStreamBusy_C());

                  S_sum = 0;

                  if (!PR_vecLen) {
                        HtContinue(VADD_RETURN);
                        break;
                  }

                  MemAddr_t addrA = SR_op1Addr + PR_offset *
                  sizeof(uint64_t);

                  MemAddr_t addrB = SR_op2Addr + PR_offset *
                  sizeof(uint64_t);

                  MemAddr_t addrC = SR_resAddr + PR_offset *
                  sizeof(uint64_t);

                  ReadStreamOpen_A(addrA, PR_vecLen);
                  ReadStreamOpen_B(addrB, PR_vecLen);
                  WriteStreamOpen_C(addrC, PR_vecLen);

                  WriteStreamPause_C(VADD_RETURN);
                  break;
            }
            case VADD_RETURN: {
                  BUSY_RETRY(SendReturnBusy_htmain());

                  SendReturn_htmain(S_sum);
            }
            break;
            default:
                  assert(0);
            }
      }
```

```
/*  Non-Instruction Functionality – executed every clock */

      if (ReadStreamReady_A() &&
            ReadStreamReady_B() &&
            WriteStreamReady_C())
      {
            uint64_t a = ReadStream_A();
            uint64_t b = ReadStream_B();
            uint64_t c = a + b;

            S_sum += c;
            WriteStream_C(c);
      }
}
```

Example 25 –Read / Write Stream Functionality

5.13 Miscellaneous Features

5.13.1 File Inclusion

Files containing defines, typdefs, structure and union declarations may be included in the personality ***htd*** file, the module instruction definition files (*Pers<mod>_src.cpp*) and the host application. Standard C/C++ *#include* control lines are supported.

5.13.2 Symbolic Constants

Symbolic constants are defined using standard C/C++ *#define* control lines. Symbolic constants can be contained in include files or inlined in the ***htd*** file, the module instruction definition files (*Pers<mod>_src.cpp*) and the host application.

5.13.3 Type Definitions

Data type names are created using standard C/C++ *typedef* declarations. Data type declarations can be contained in include files or inlined in the ***htd*** file, the module instruction definition files (*Pers<mod>_src.cpp*) and the host application.

Note: All signed and unsigned integers with widths between 1 and 64 are pre-defined. The names of these pre-defined types are ht_uint1_t, ht_uint2_t, … ht_uint64_t and ht_int1_t, ht_int2_t, … ht_int64_t. Applications can use these types without having to define them.

5.13.4 Structures and Unions

Structures and unions are declared using standard C/C++ declarations. These declarations can be contained in include files or inlined in the ***htd*** file, the module instruction definition files (*Pers<mod>_src.cpp*) and the host application.

Structures/Unions with bit field width specifications are frequently used in C/C++ programming to reduce the space needed to store a collection of data items in memory. Since it is necessary for a personality to read/write these structures, the HT tools generate the same compressed structure format. In addition, the HT tools also support arrays of structure members that use bit field widths, for use in the personality only, since C/C++ does not support arrays of structure members that use bit field widths.

5.14 Advanced Personality Features

5.14.1 Replication of Modules

In some situations, performance can be increased by replicating modules. Replicated modules have the same inputs and outputs. Threads are distributed across the replicated modules. Replicated modules are specified in the ***hti*** file. See Section 5.14.3 for information on specifying replication of modules in the ***hti*** file .

Some interfaces are restricted in replicated modules, since the module name is used as the source or destination. As a result the following functionality is not available for replicated modules:

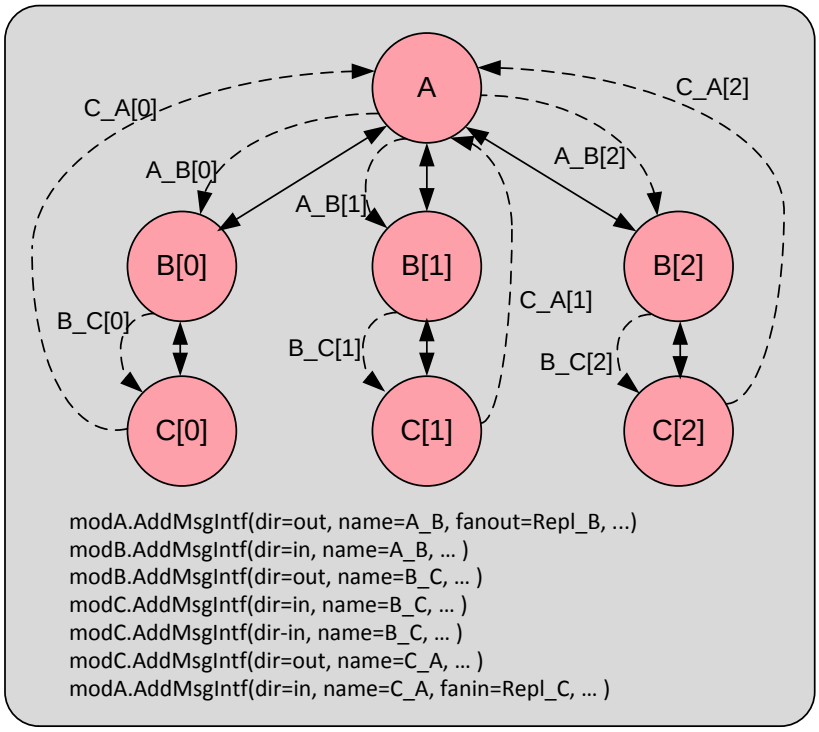
- Host Data
- Outbound Host Message
- Access to global variables

A replication ID (SR_replId) is available to each module instance.

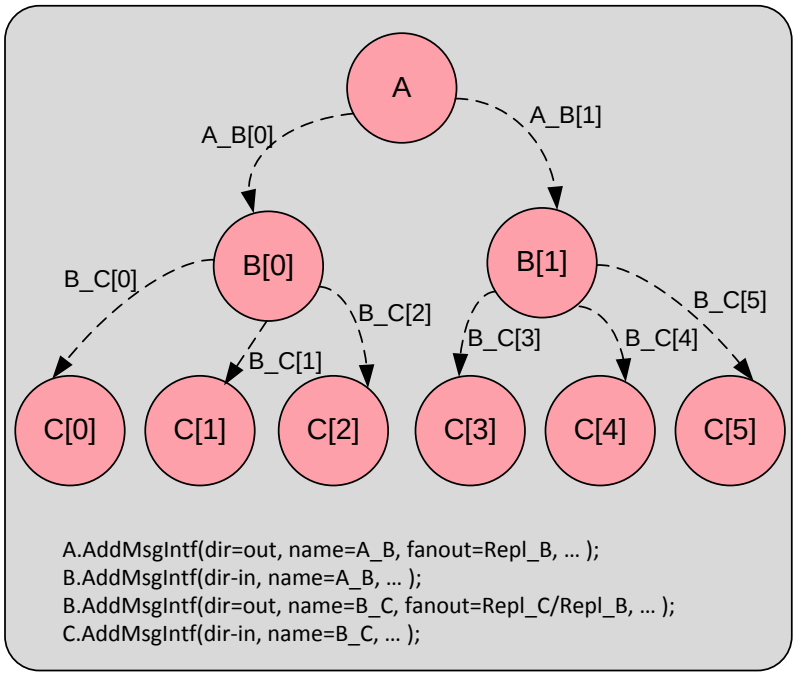
5.14.1.1 Module Messages and Replicated Modules

The connection of module messages is defined in the *AddMsgIntf* ***htd*** command. When module replication is used the *fanin* and *fanout* parameters are used to configure the message interfaces between replicated modules.

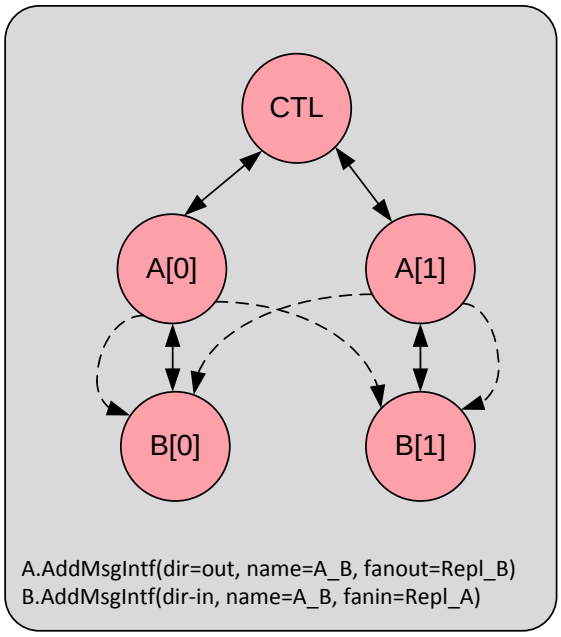
Examples of several message interface topologies with replicated modules are shown below.



Example 26- Message Interface with Replicated Modules



Example 27 – Message Interface with Replicated Modules



Example 28 – Message Interface with Replicated Modules

5.14.2 Assigning Memory Ports to Modules

By default each unit is assigned a single memory port. If additional memory bandwidth is required, memory ports can be assigned to modules (or replicated modules). Memory ports can be added for

- Memory read/write interfaces
- Read stream(s)
- Write streams(s)

The maximum number of memory ports a module can be assigned is 16. The module memory ports are mapped to the unit memory ports using the Hybrid Thread Instance (***hti***) file. A maximum of 16 unit memory ports are available. Units are assigned consecutive memory ports. The total number of memory ports in an AE is 16. This includes the memory ports assigned to each unit and any additional memory ports assigned to modules within the units.

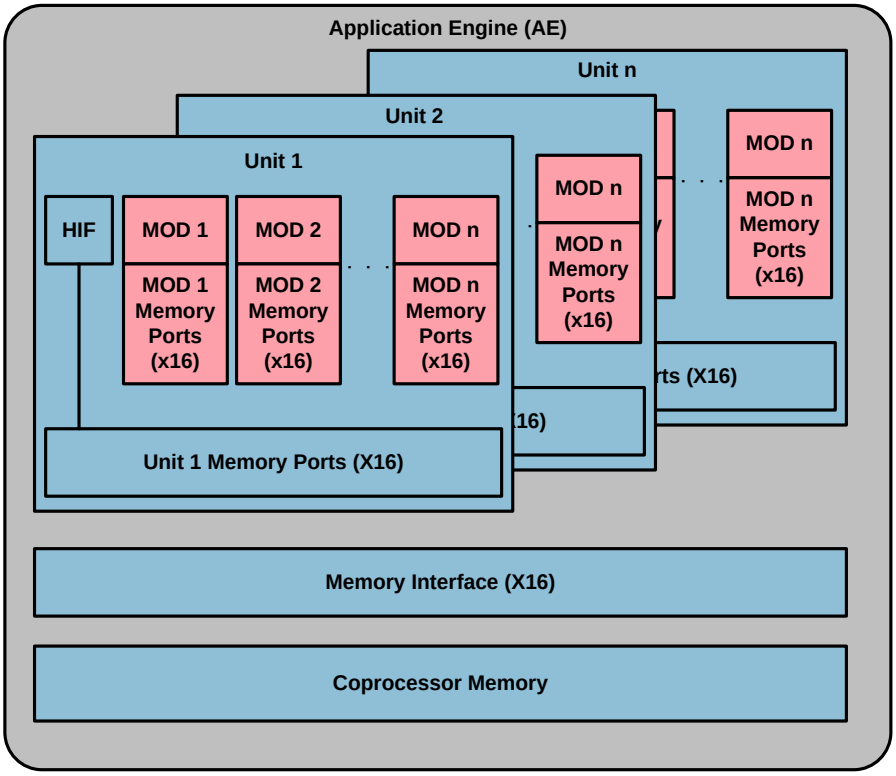


Figure 11- HT Memory Ports

5.14.2.1 Module Memory Ports

Each module supports a maximum of 16 module memory ports. Memory read and write interfaces are assigned to module memory port 0. Read and write streams default to module memory port 0, but must be assigned to other module memory ports if the module contains a memory read or write interface. Read and write streams are assigned module memory ports using the *memPort* parameter of the *AddReadStream* or *AddWriteStream* ***htd*** command. Module memory ports must be assigned consecutively.

5.14.2.2 Unit Memory Ports

By default all module memory ports are assigned to like numbered unit memory ports. Unit memory port assignments are selected using the *memPort* parameter of the *AddModInstParams* command in the Hybrid Thread Instance file (***hti***). See Section 5.14.3 for more information on ***hti*** commands.

The HIF for the unit is always assigned unit memory port 0. A module read / write interface and/or read / write streams can also be assigned to unit port 0. Unit memory ports must be assigned consecutively.

5.14.2.3 Memory Interfaces

There are a maximum of 16 memory interface ports per AE. Unit memory ports are assigned to memory interface ports, consecutively starting with unit 0.

5.14.3 Hybrid Thread Instance File

The Hybrid Thread Instance file (***hti***) is used to add parameters to modules and add message interfaces between modules in different units.

5.14.3.1 Module Instance Parameters

Parameters for module instances are set using the *AddModInstParams* ***hti*** command. The module instance is identified using the unit and module path. This command is used to replicate module instances and associate module memory ports to a unit memory ports.

```
AddModInstParams( unit=<unit_name>,
modPath="<module_path>", memPort=<port>,
replCnt=<repl_num>);
```

The *unit* parameter specifies the unit containing the module instance.

The *modPath* parameter specifies the path to the module instance. If multiple paths exist, one of the paths is used to identify the module instance.

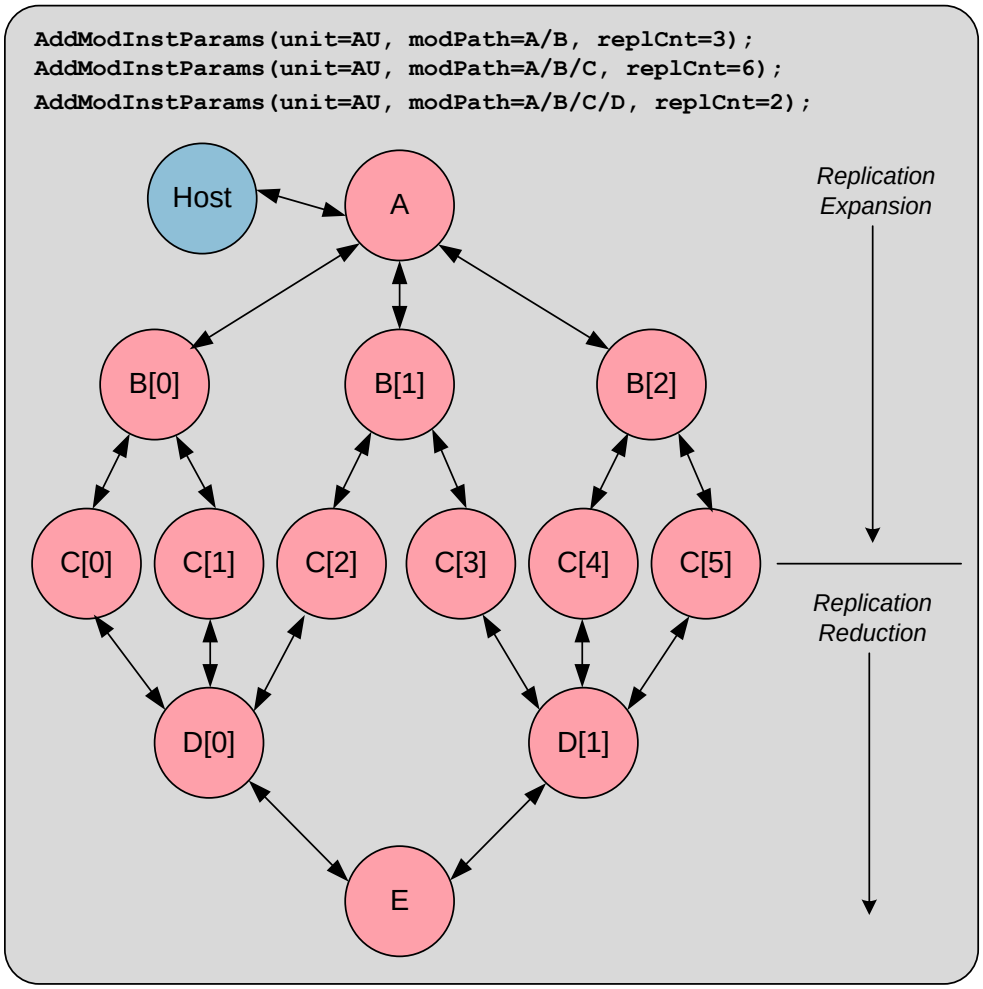
The *memPort* parameter is used to assign module memory ports to unit memory ports. Valid values are 0 – 15. Any combination of module memory ports can be mapped to a unit memory port, but the unit ports must be consecutive. The HIF is assigned to unit memory port 0.

The *replCnt* parameter specifies the number of times the module is replicated. Valid values are 1 – 8. Using and additional AddModInstParams command with the path to the module instance, memory ports can be assigned to each module instance.

5.14.3.1.1 Module Instance Parameter Usage

This section contains examples of several module instance usages.

Example 29 illustrates a call graph showing module replication expansion and reduction, along with the corresponding commands in the ***hti*** file.



Example 29 Replicated Modules Expansion and Reduction

Note: When expanding the replication count must be a multiple of the replication count of the preceding module in the path.

replCnt of B = 3 * replCnt of A = 3

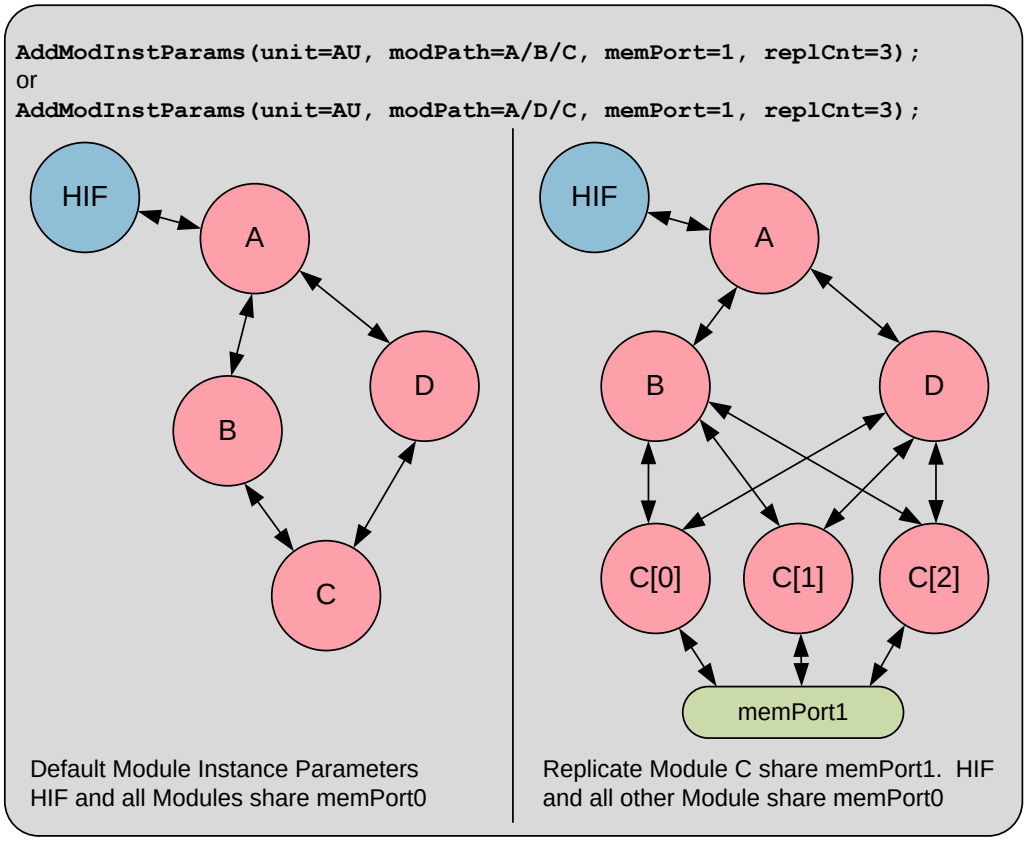
replCnt of C = 2 * replCnt of B = 6

When reducing the replication count of the preceding module in the path must be divisible by the replication count.

replCnt of D = replCnt of C / 3 = 2

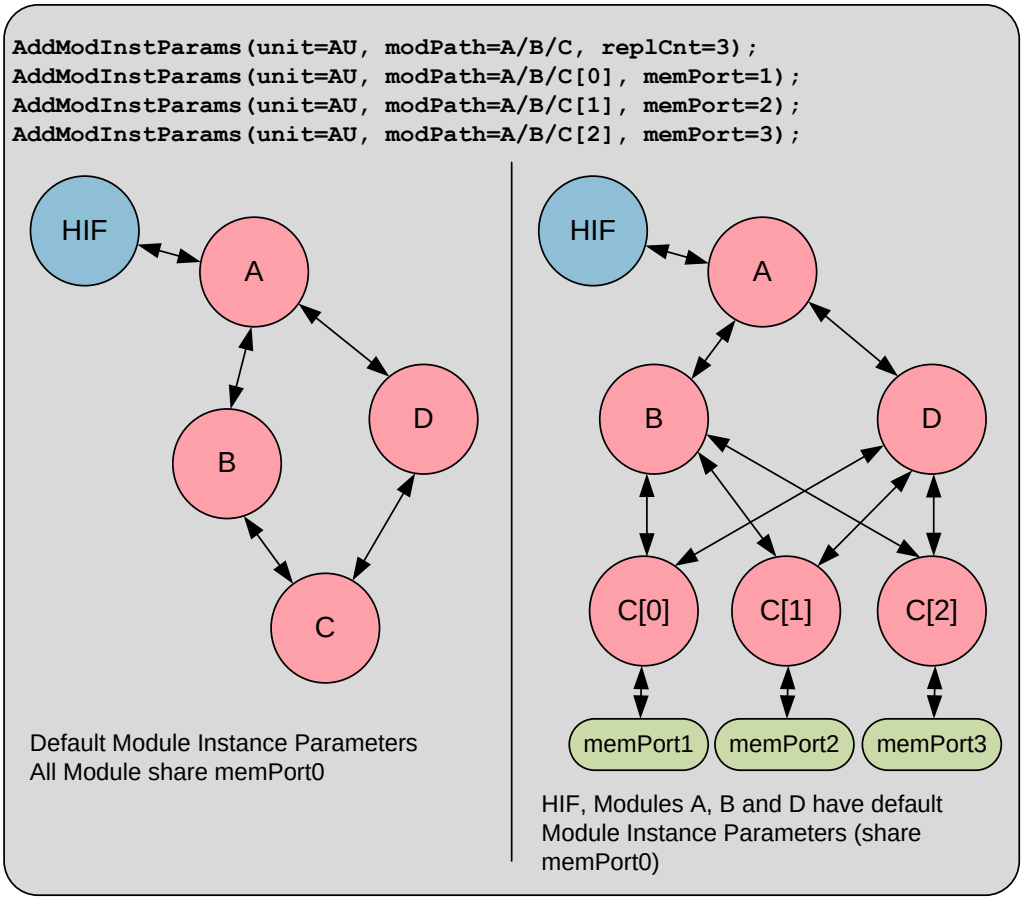
replCnt of E = replCnt of D / 2 = 1

Example 30 illustrates a call graph showing module replication with replicated modules sharing unit memory port 1, along with the corresponding commands from the ***hti*** file. Note there are two paths for module C. Either path is valid to specify module C replication.



Example 30 Replicated Modules Sharing a Memory Port

Example 31 illustrates a call graph showing module replication with the memory read and write interfaces of replications of module C each having a unique unit memory port, along with the corresponding commands from the ***hti*** file.



Example 31 – Replicated Modules with Unique Memory Ports

Each unit of design in Example 31 uses 4 memory ports, so the design can have up to 4 units.

5.14.3.2 Message Interface Between Units

Connections for messages between modules in different units are added using the *AddMsgIntfConn* ***hti*** command. The module instance is identified using the unit and module path. Note messages are only supported between units in the same AE.

```
AddMsgIntfConn( outUnit=<unit_name>, outPath="<module_path,
inUnit=<unit_name>, inPath="<module_path>",);
```

The *outUnit* parameter specifies the unit containing the module instance that is sending the message.

The *outPath* parameter specifies the path to the module instance that will send the message. If multiple paths exist, one of the paths is used to identify the module instance.

The *inUnit* parameter specifies the unit containing the module instance that is receiving the message.

The *inPath* parameter specifies the path to the module instance that will receive the message. If multiple paths exist, one of the paths is used to identify the module instance.

5.14.4 Incorporating Existing IP in an HT Design

The HT toolset allows the user to incorporate existing IP verilog or modules created from CORE Generator as a black box primitive in an HT design.

The black box primitive is called as a function in the non-instruction functionality of the module. The primitive is executed every clock cycle. The inputs to the primitive are the input verilog ports and the outputs are the output verilog ports.

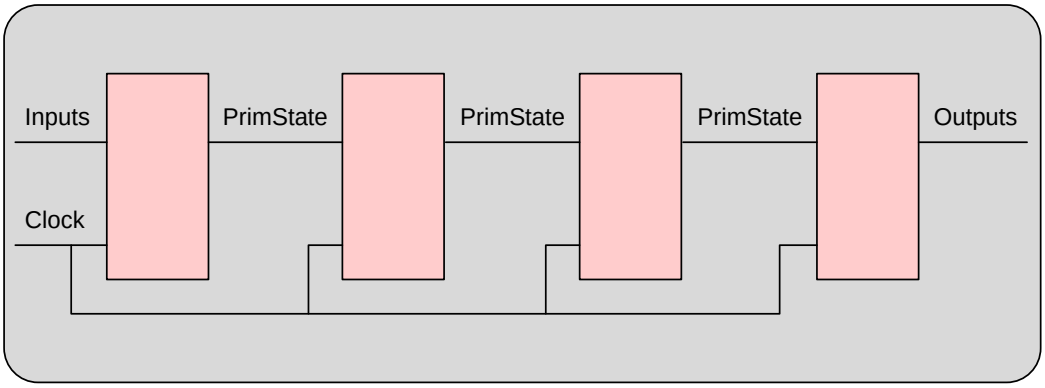


Figure 12 – Black Box Primitive

Multiple clock options are available for the primitive.

- Module clock
- 2x module clock
- 1x absolute clock.
- 2x absolute clock

The following steps are required to incorporate a black box in an HT design

- Define the internal state (PrimState) of the black box primitive
- Define the functionality of the primitive for simulation
- Declare the primitive as a function
- Call the function from the HT module

In some cases the designer may need to design a wrapper for the existing verilog module to ease integration into the HT design.

5.14.4.1 Define Primitive State

Primitive state is internal state in the black box primitive that is needed to simulate the HT design. The internal state includes all state that is required from one clock to the next in the model of the primitive. Primitive state is specified using the *AddPrimState* ***htd*** command.

The *AddPrimState* ***htd*** command specifies the type of the *ht_state* structure containing the state for the primitive and the name of the instance of primitive state, along with the name of the file that contains the structure and primitive definitions. The file is typically named *Pers<unit_name><mod_name>_prim.h.* This file is included in following files:

- Model of the verilog module (*Pers<unit><mod_name>_prim.cpp*)

- Module custom instruction definition (*Pers<unit><mod_name>.cpp*)

5.14.4.2 Declare the Primitive State Structure and Primitive Function

The primitive state structure and the primitive function are declared in the file specified in the *AddPrimState* ***htd*** command. The file is typically named *Pers<unit_name><mod_name>_prim.h*.

The primitive ht_state structure declaration is shown below:

```
ht_state struct <ht_prim_type> {
      <type>      <state_1>;
      <type>      <state_2>;
      …
      <type>      <state_n>;
}
```

The primitive function declaration associates the primitive module to the function and maps the clock to the primitive clock. Arguments of the function include I/O ports of the primitive and the *PrimState* as shown below:

```
ht_prim ht_clk("<prim_clk>") void <prim_mod_name> (
      <type> & <prim_port1_name>,
      <type> & <prim_port2_name>,
      …
      <type> & <prim_portn_name>.
      <type_prim_state> &s);
```

prim_mod_name is the primitive module name

prim_clk is the name of the clock signal in the primitive module. Valid names are

`intfClk1x` – module clock

`intfClk2x` – 2x module clock

`clk1x, ck or clk` – absolute clock

`clk2x` – 2x absolute clock

prim_port defines the input and output ports of the primitive module.

If the Verilog uses typedefs, defines, structures, etc defined in the ***htd***, *PersCommon.h* should be included in the file specified in the *AddPrimState* ***htd*** command.

5.14.4.3 Define Primitive Functionality

In order to run systemC simulation the functionality of the verilog module must be provided. The functionality of the verilog module is described in *Pers<unit><mod_name>_prim.cpp*.

The functionality does not have to be implemented with cycle by cycle accuracy, but the outputs must be valid on the same cycle as the outputs are valid in the verilog module.

The primitive state must be passed in, since all local state is lost after each cycle.

5.14.4.4 Call Primitive Function

Primitive functions are called as non-instruction functionality, so they are executed every clock cycle. Typically, arguments of the primitive function are set up in the instruction

functionality of the module. The thread is then paused using the *HTPause* routine. The *HtPause* routine specifies the instruction to execute when the thread is resumed.

The module calls the primitive function, every clock cycle from the non-instruction. Outputs from the primitive are stored in temporary C variables that are not saved functionality. Valid results from the primitive are stored and the thread is resumed, using the *HtResume* routine.

5.14.4.5 Primitive Usage

This section describes an example in which a 5 stage verilog module is incorporated in an HT module named *exmod.* The verilog module (*ex_v_mod*) has a clock, 3 inputs and 3 outputs as shown in the verilog port list in Example 32. The verilog module adds *i_a* and *i_b* to produce *o_res*.

```
module ex_v_mod (
input       ck,

input  [63:0]   i_a, i_b,
input   [6:0]   i_htId,
input           i_vld,

output [63:0]   o_res,
output  [6:0]   o_htId,
output          o_vld
);
```

Example 32 – Primitive Verilog Port List (*.v)

The *htd* command in Example 33 describes an instance of the primitive state called *state1* of type *prim_state*. The primitive state structure and primitive function are declared in *PersExmod_prim.h*.

```
exmod.AddPrimState(type=prim_state, name=state1, include =
"PersExmod_prim.h");
```

Example 33 – Primitive htd (*.htd)

The *PersExmod_prim.h* file contains the ht_state structure declaration and the primitive function declaration as shown in Example 34.

```
#pragma once

// state for exmod clocked primitive
ht_state struct prm_state {
      uint64_t    res[6];
      ht_uint7    htId[6];
      bool        vld[6];
};

// function definition for verilog module
ht_prim ht_clk("ck") void ex_v_mod (
      uint64_t const & i_a,
      uint64_t const & i_b,
```

```
        ht_uint7 const & i_htId,
        bool const & i_vld,
        uint64_t & o_res,
        ht_uint7 & o_htId,
        bool & o_vld,
        prm_state &s);
```

Example 34 – Primitive State Structure and Function Declarations

In order to perform systemC or verilog simulations the behavior of the verilog module must be provided in *PersExmod_prim.cpp.* The verilog output must be available on the same cycle as the actual verilog module, so in the example the results are staged for 5 cycles to match the verilog. Keep in mind that all state within the primitive is lost between stages. Example 35 contains the simulation model for the example.

```
#include "Ht.h"
#include "PersExmod_prim.h"

ht_prim ht_clk("ck") void ex_v_mod (
      uint64_t const &    i_a,
      uint64_t const &    i_b,
      ht_uint7 const &    i_htId,
      bool const &        i_vld,
      uint64_t &          o_res,
      ht_uint7 &          o_htId,
      bool &              o_vld,
      prm_state &         s)
{
#ifndef _HTV

      o_res  = s.res[5];
      o_htId = s.htId[5];
      o_vld  = s.vld[5];

// stage the results for 5 cycles to match the verilog
// state must be passed in because all local state is lost
// on each return
      int i;
      for (i = 5; i > 1; i--) {
            s.res[i]  = s.res[i - 1];
            s.htId[i] = s.htId[i - 1];
            s.vld[i]  = s.vld[i - 1];
      }

      s.res[1]  = i_a + i_b;
      s.htId[1] = i_htId;
      s.vld[1]  = i_vld;

#endif
}
```

Example 35 – Simulation Model of Primitive

The primitive is called in the non-instruction functionality, which is executed every clock cycle as shown in Example 36. The custom instructions *ADD_LD1* and *ADD_LD2* load operand 1 and 2. The *ADD_PAUSE* instruction prepares the inputs for the verilog module, storing the operands in private variables along with the thread ID and the input valid indication. The instruction then pauses the thread. In the non-instruction functionality the primitive module function is called (every clock cycle). Output valid from the primitive is checked. If valid, the result from the primitive is stored and the thread is resumed executing the *ADD_ST* instruction, as specified by the *HTPause* routine .

```
#include "Ht.h"
#include "PersAdd.h"

// Include file with primitives
#include "PersAdd_prim.h"

void
CPersAdd::PersAdd()
{
// Force "Inputs Valid" to default to false
      P_i_vld = false;

      if (PR_htValid) {
            switch (PR_htInst) {
            case ADD_LD1: {
                  //Load the first operand
                  HtContinue(ADD_LD2);
            }
            break;
            case ADD_LD2: {
                  //Load the second operand
                  //ReadMemPause(ADD_PAUSE);
            }
            break;
            case ADD_PAUSE: {
                  // Store op1 and op2 into private variables 'a'
                  // and 'b'.

                  // Mark inputs as valid, set htId
                  P_i_htId = PR_htId;
                  P_i_vld = true;

                  // Pause thread and wait for primitive to
                  // calculate the result...
                  // (will return to ADD_ST)
                  HtPause(ADD_ST);
            }
            break;
            case ADD_ST: {
                  //Store result
            }
            break;
            case ADD_RTN: {
                  // Return result
            }
            break;

            }
      }

      //Non Instruction functionality executed every cycle

      // Temporary variables to use as outputs to the primitive
      // (these are not saved between cycles)
```

```
      uint64_t o_res;
      ht_uint7 o_htId;
      bool o_vld;

      // use primitive function declared in Example 34
      ex_v_mod(P_a, P_b, P_i_htId, P_i_vld, o_res, o_htId, o_vld,
state1);

      // Check for valid outputs from the primitive
      if (o_vld) {
            // Store Result
            S_resMem.write_addr(o_htId);
            S_resMem.write_mem(o_res);
            // Wake up the thread (with corresponding htId)
            HtResume(o_htId);
      }

}
```

Example 36- Calling a Primitive Function

5.14.5 Thread Synchronization / Barriers

The barrier function provides barrier synchronization across the threads in a module, including the threads in all replications of the module, if replication is implemented.

The barrier is defined with the *AddBarrier* ***htd*** command in the module's description. The *AddBarrier* command specifies the name of the barrier and the width of the barrier index. If no name is specified the barrier is unnamed. Multiple *AddBarrier* ***htd*** commands can exist within a module, but all must have unique names, so only one unnamed barrier can exist.

The personality sets a barrier using the following routines:

`HtBarrier({barId,}nextInst, threadCnt)`
`HTBarrier_<name>({barId,}nextInst, threadCnt)`

The *barId* parameter specifies the index of the barrier to be used. Note that if the *barIdW* ***htd*** parameter is not specified, only one barrier is provided and the *barId* parameter is not present. If the *barIdW* ***htd*** parameter is specified as zero, the *barId* parameter is present, but the only valid value is zero.

The *nextInst* parameter specifies the next instruction that all threads will execute once the barrier thread count is reached. The next instruction is the same for all threads entering the barrier.

The *threadCnt* parameter specifies the number of threads that must enter the barrier before the threads are all released to execute the *nextInst*. The *threadCnt* value can range from 2 to the sum of the number of threads in the module or replicated modules.

Once the barrier is release, allowing threads to execute the *nextInst*, the barrier can be immediately reused.

5.14.6 Thread Transfer

A transfer begins a thread in another module at the entry point defined by the HT function indicated. The thread has the same thread ID as the source thread. The execution of

the thread in the source module is terminated. When a return from a transfer is performed the return goes to the module of the last caller.

The linkage from source module to the specified module is provided with the *AddTransfer* ***htd*** command in the source module's description. The *AddTransfer* command specifies the name of the destination HT function. The HT function is defined in the transferred module with the *AddEntry* ***htd*** command. The HT function must reside in a different module than the module executing the transfer.

Before a transfer can be issued, the personality must assure the transfer interface can accept a transfer. The *SendTransferBusy* routine is used to determine if a transfer can be performed.

```
SendTransferBusy_htfunc()
```

The *SendTransferBusy* routine returns a *true* if a new transfer cannot be performed. If a transfer cannot be performed, the instruction should use *HtRetry* to retry the instruction

After verifying a transfer can be performed, the personality uses the *SendTransfer* routine to transfer the thread to the destination module.

```
SendTransfer_htfunc(<xferParam1>, <xferParam2>, …)
```

The optional transfer parameters are written to the destination thread's private variables. The transfer parameters correspond to the *AddParam* subcommands for the called entry point.

5.14.6.1 Transfer Usage

This section provides a pseudo code example for modules executing the Transfer and Return functionality illustrated in Figure 13.

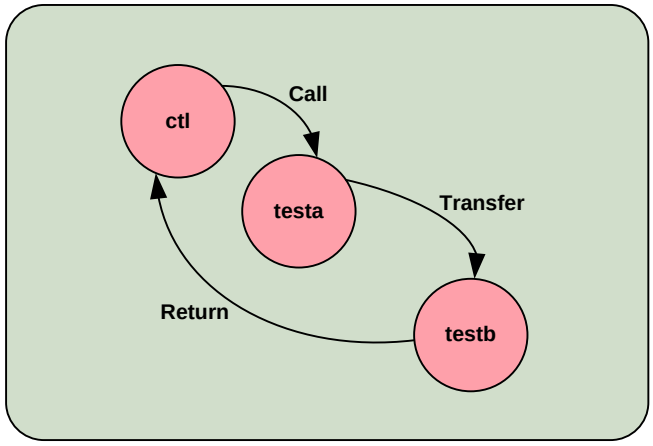


Figure 13 – Call Graph for Transfer Example

The ***htd*** for both the modules is shown in Figure 14.

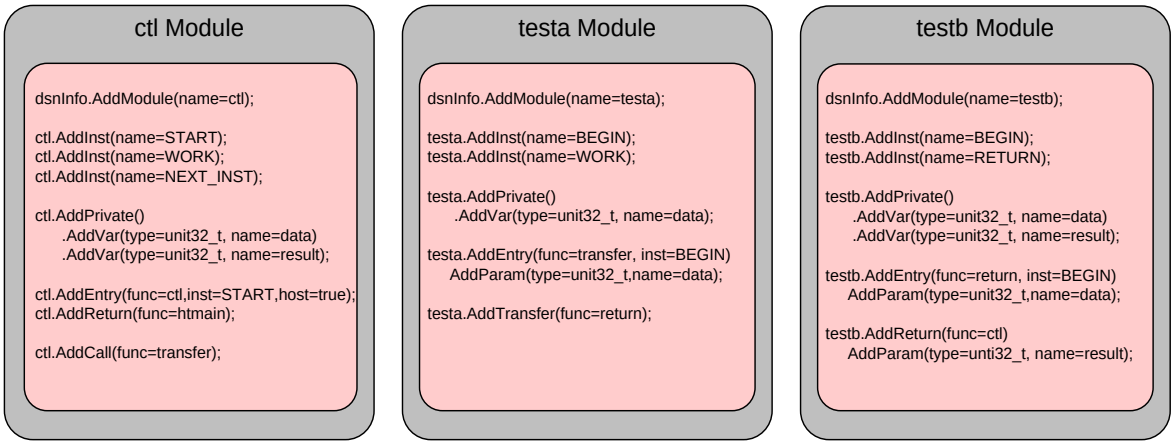


Figure 14 – htd for Transfer Example

Example 14 shows the instruction definitions for the transfer example. A module named *ctl* is called from the host. The *ctl* function makes a call to the *transfer* function in the *testa* module, passing a private variable *P_data*. The *testa* module transfers the thread to the *return* function of the *testb* module. The *return* function returns *result* back to *ctl*. Note the *return* is associated with the *call*, so the *call* in the *ctl* module specifies the next instruction to execute.

```
void
CPersCtl::PersCtl()
{
       if (PR_htValid) {
              switch (PR_htInst) {
              case START: {
                     // Entry Point - Initial processing

                     // Execute WORK instruction next
                     HtContinue(WORK);
              }
              break;
              case WORK: {
                     // Check if call interface can accept a new call
                     if (SendCallBusy_transfer()) {
                     HtRetry();
                     break;
                     }

                     // Call transfer, passing private
                     // variable P_data
                     // Execute NEXT_INST on return.
                     SendCall_transfer(NEXT_INST, P_data);
              }
              break;
              case NEXT_INST: {

                     // P_result contains return value from return
                     // Whatever you want to happen next
              }
              break;

              . . .
```

```
void
CPersTesta::PersTesta()
{
      if (PR_htValid) {
            switch (PR_htInst) {
            case BEGIN: {
                  // Entry Point - Initial processing

                  // Execute WORK instruction next
                  HtContinue(WORK);
            }
            break;
            case WORK: {
                  // Check if transfer interface can accept a transfer
                  if (SendTransferBusy_return()) {
                  HtRetry();
                  break;
                  }

                  // Transfer thread to transfer function of the test1,
                  // module, passing private variable P_data
                  SendTransfer_return (P_data);
            }
            break;
            }
      }
```

```
void
CPersTestb::PersTestb()
{
       if (PR_htValid) {
              switch (PR_htInst) {
              case BEGIN: {
                     // Entry Point - Initial processing

                     // Execute RETURN instruction next
                     HtContinue(RETURN);
              }
              break;
              case RETURN: {
                     // Check if return interface can accept a return
                     // Return thread to original call point
                     if (SendReturnBusy_transfer()) {
                     HtRetry();
                     break;
                     }

                     // Return, passing private variable P_result
                     SendReturn_transfer(P_result);
              }
              break;
              }
       }
}
```

Example 37 – Transfer Functionality

5.15 Personality Design Considerations

This section provides information to help the user in selecting the appropriate functionality to utilize in developing personalities using the HT tools.

5.15.1 Selecting Data Storage

Data can be stored

Storage Type	Primary Uses
Private Variables	• Module state • Data unique to each thread
Shared Variables	• Data from asynchronous events • Data shared by multiple threads
Global Variables	• Data shared between modules
Memory	• Data shared between units

Table 14 – Data Storage

Table 15 summarizes the characteristics of the different variable types.

Characteristic	Private	Shared	Global
Type			
HT Special (ht_int*t, ht_uint*t)	X	X	X
Scalar	X	X	X
Multi-dimensional (2 indexes)	X	X	X
Depth (2 indexes)	X		X
Field			X
Queue		X	
Structure	X	X	X
Union	X	X	X
Accessible by			
Thread in a module	X	X	X
All threads in the module		X	X
All modules in a Unit			X
Initialized	X		
Modified by			
HT Instruction	X	X	X
Argument of a call	X		
Return parameter	X		
Host message		X	
Memory access	X	X	X
Non-instruction Functionality		X	

Table 15 – Comparison of Variable Types

5.15.2 Using Arrays

The HT tools support two methods to define the number of elements in arrays, depth and index dimension as shown in Figure 15. Each method supports two components. The methods can be combined to define a two dimensional structure using one depth component and one index dimension component or any combination of components to specify up to a four dimensional data structure.

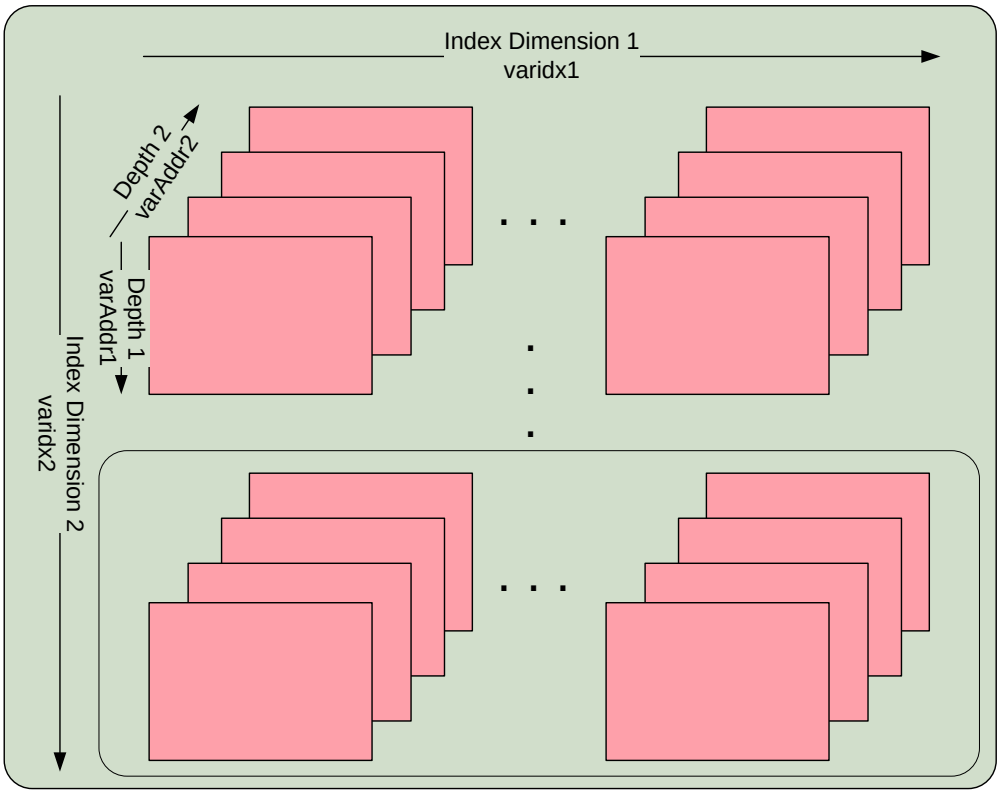


Figure 15 – Array Definition

Global variables may also consist of fields that can be one or two dimensional and are addressed by *fldidx1* and *fldidx*..

5.15.2.1 Variable Depth vs Index Dimension

Using the depth to define the number of elements in an array utilizes FPGA resources the most efficiently. However, when depth is used, only one element can be read and only one element can be written per HT instruction. The code accessing elements is also more complex, since the element location, varAddr1(/2) is specified in private or temporary variable(s), addr1 (and addr2).

All elements of an array defined with index dimensions can be accessed in the same HT instruction.

5.15.3 Accessing Data

The personality can access data using several mechanisms. Some of the mechanisms are supported from host or coprocessor memory. In order to pick the appropriate means access requirements should be considered. The requirements may include the following:

- Bandwidth needed
- Tolerance to latency
- Tolerance to variation in latency
- Sequential or random data
- How often the data is accessed
- Hardware utilization

5.15.3.1 Host Message Interface

The host message interface supports a 56 bit data field to send to or from the host. Multiple messages can be used to provide information that requires greater than 56 bits. Typically used to transfer small amounts of data. Applications include passing an address or reporting an asynchronous event. Messages from the host to the coprocessor are automatically stored in shared variables and are often used to initialize shared variables prior to calling the coprocessor.

5.15.3.2 Host Data Interface

The host data interface provides inbound and / or outbound queues used to send or receive n X 8 byte data blocks to or from the host.

5.15.3.3 Read / Write Host Memory

The coprocessor can read and write host memory. Host memory accesses occur across the PCIe bus, so they are lower bandwidth and have higher latency than coprocessor memory accesses. Host memory accesses are recommended for random data which the coprocessor does not access often.

5.15.3.4 Read / Write Coprocessor Memory

Coprocessor memory reads and writes are higher bandwidth and have lower latency than host memory accesses, but the data must be moved to coprocessor memory prior to the coprocessor call. Coprocessor memory accesses are recommended for large amounts of random data or random data that is accessed multiple times.

5.15.3.5 Streaming Data from Host Memory

The coprocessor may utilize a streaming data interface to access sequential data in host memory. The streaming interface provides high bandwidth, low latency access to host data. This interface is recommended for accessing sequential data that is not accessed repeatedly.

5.15.3.6 Streaming Data from Coprocessor Memory

The coprocessor may utilize a streaming data interface to access sequential data in coprocessor memory. Data must be moved to coprocessor memory prior to the coprocessor call. The streaming interface provides high bandwidth, low latency access to sequential data located in coprocessor memory. This interface is recommended for accessing sequential data that is accessed multiple times.

	High Bandwidth	Low Latency	Consistent Latency	Sequential Data	Random Data	Multiple Coprocessor Access	High Hardware Utilization
Host Message Interface				NA	NA		
Host Data Interface				NA	NA		
Read / Write Host Memory					X		
Read / Write Coprocessor Memory	X	X			X	X	X
Streaming from Host Memory		X	X	X			X
Streaming from Coprocessor Memory	X	X	X	X		X	X

Table 16 – Memory Access Comparison

5.15.4 Partitioning Considerations

Partitioning a coprocessor personality into modules, instruction and stages is design specific. This section outlines information to consider when partitioning a design.

- Within a module one thread is valid each cycle. Each cycle one instruction (or one stage of one instruction) is executed on the active thread.
- When using multiple stages, the same thread will be active for the length of the instruction pipeline.
- All instructions in a module have the same pipe length, dictated by the instruction with the largest number of stages.
- Calls and Returns have overhead in the call and return interface

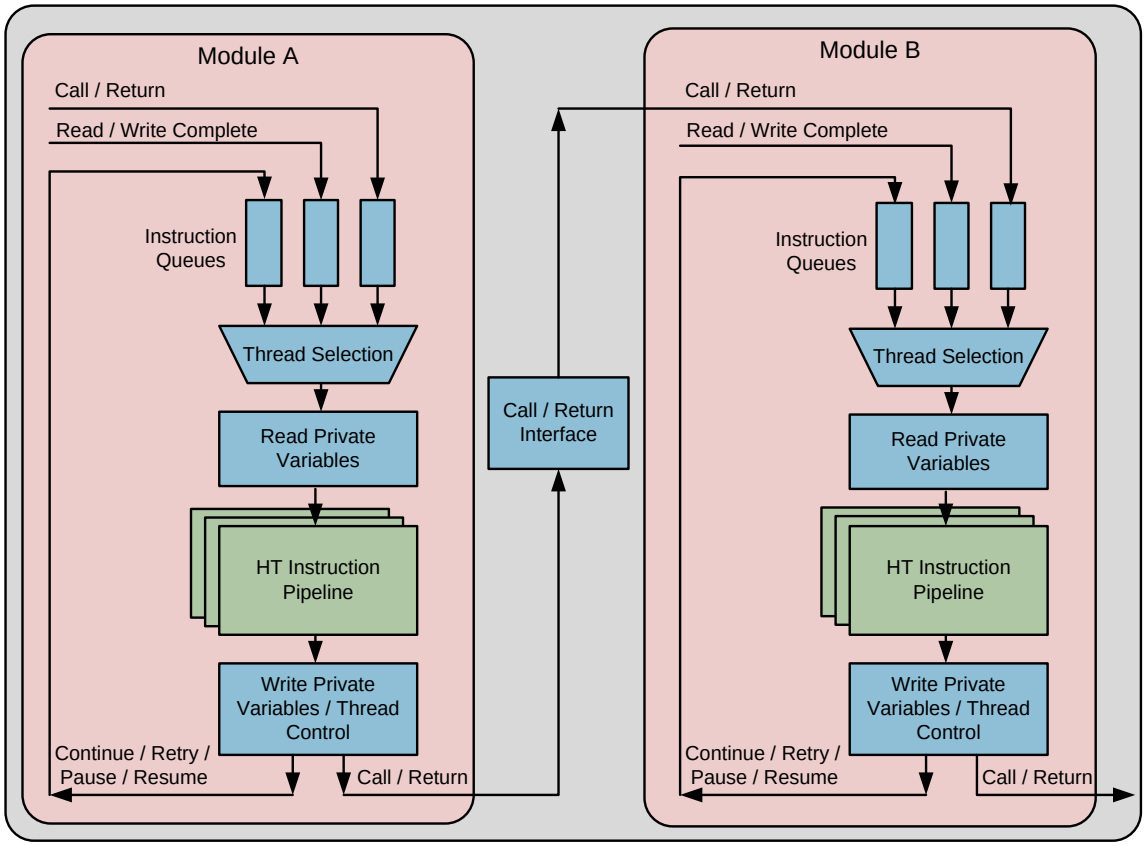


Figure 16 – Instruction Flow with Single or Multiple Modules

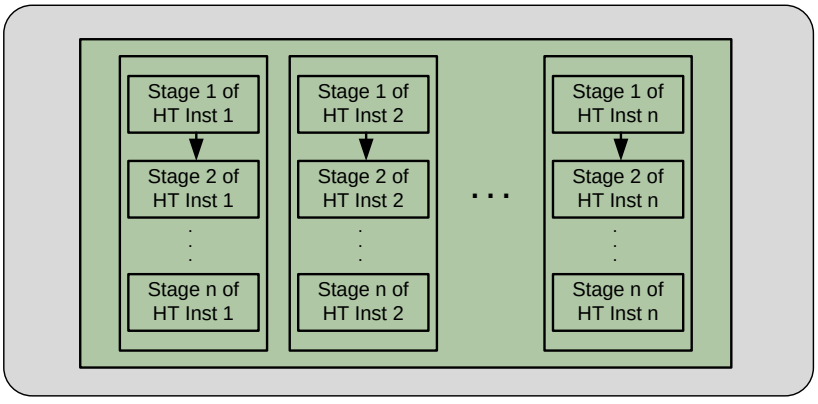


Figure 17 – HT Instruction Pipeline

6 Debugging and Optimizing HT Personalities

Standard debugging methods such as GDB and printf can be used to debug the host application and the model, along with the personality design during simulation. In addition, the HT tools provide an *HtAssert()* function, performance monitoring and a html design that are useful in developing, debugging and tuning performance.

6.1 Html Design Report

The HT tools produce an html design report (HtDsnRpt.html) from the ***htd*** input, which contains the following:

- Call graph of the design
- Module HT API for each module
- Information on the RAMs generated in the FPGA

The html design report can be a useful resource when coding the custom instructions, since it contains the Module HT API. Note: The report is generated from the ***htd*** input only, so the definition of the custom instructions does not have to exist or may be incomplete for the report to be generated.

6.2 HtAssert

The *HtAssert()* function acts as a standard *assert()* during simulation, which aborts the program if an expression fails. When verilog is generated for the personality, the HT tools also generate logic for the assertion, so the assertion exists in the actual hardware.

If an assertion fails, a message with the instruction source file and line number is sent to the host and printed to standard error.

6.3 Performance Monitor

The HT environment supports performance profiling of the units, including

- Memory latencies and utilization
- Thread utilization
- Module utilization
- Instruction tracing

The data supplied in the performance monitoring report aids the programmer in exploring architectural changes and identifying performance bottlenecks before running on the actual hardware.

The performance monitor report is found in a file named *HtMonRpt.txt* located in the directory from which the simulation was run.

A sample performance monitor report is shown below:

```
Platform              :   hc-2
Simulated Cycles      :  578000
#
# Memory Summary
```

```
#
Host Latency (cyc)    :    399 /     432 /     626  (Min / Avg / Max)
CP Latency (cyc)      :     68 /     131 /     706  (Min / Avg / Max)
Host Operations       :    392 /      39          (Read / Write)
CP Operations         :  200000 /  100000          (Read / Write)
Host Efficiency       :  12.72% /  12.50%          (Read / Write)
CP Efficiency         : 100.00% / 100.00%          (Read / Write)
Host Utiliztion       :   0.08% /   0.08%          (Req / Resp)
CP Utiliztion         :  53.82% /  53.82%          (Req / Resp)
#
# Memory Operations              Read     ReadMw     Write     WriteMw
#                      ---------- ---------- ---------- ----------
<total>                      200391          1     100039          0
HIF                             391          1         39          0
ctl/ADD                      200000          0     100000          0
#
# Thread Utilization           Avg. Run  Avg. Alloc Max. Alloc  Available
#                      ---------- ---------- ---------- ----------
ctl/ADD                           3         58         63        128
#
# Module Utilization             Valid     Retry   Active Cyc  Act. Util
#                      ---------- ---------- ---------- ----------
CTL                          278598      78595     557413     96.44%
 CTL_ENTRY                        1          0
 CTL_ADD                     178596      78595
 CTL_JOIN                    100000          0
 CTL_RTN                          1          0
ctl/ADD                      428892      28892     557403     96.44%
 ADD_LD1                     111810      11810
 ADD_LD2                     111630      11630
 ADD_ST                      105452       5452
 ADD_RTN                     100000          0
```

6.3.1 Simulated Cycles

The *Simulated Cycles* field provides the total number of cycles simulated, starting when the coprocessor is started to the return to the host application.

6.3.2 Memory summary

The *Memory Summary* provides information concerning accesses to coprocessor memory. The information includes latency, number of operations, efficiency and utilization data.

6.3.2.1 Host and CP Latency

Memory latency is provided for both host and coprocessor accesses to coprocessor memory. Minimum, average and maximum values are provided in clock cycles.

6.3.2.2 Host and CP Operations

The number of read and write accesses to coprocessor memory from the host and coprocessor are provided in the *Host* and *CP Operations* fields.

6.3.2.3 Host and CP Efficiency

The *Efficiency* fields of the performance monitor report provide information on the bandwidth of the memory operations. Memory accesses that do not use the full width of the memory access, such as multiword accesses that do not utilize the full memory width or sub 64 bit accesses reduce the efficiency. For example if all memory accesses are 32 byte accesses and the platform supports 64 byte accesses the efficiency will be 50%.

6.3.2.4 Host and CP Utilization

The *Utilization* fields of the performance monitor report provide information on the number of cycles used for memory accesses. The percent of cycles used for memory requests and responses are provided for both host and coprocessor accesses to coprocessor memory.

6.3.3 Memory Operations

The *Memory Operations* fields of the performance monitor report provide the total number of memory operations performed by the coprocessor. The numbers of memory operations performed by the HIF and each module in the design are also provided.

6.3.4 Thread Utilization

The *Thread Utilization* fields of the performance monitor report provide data on thread utilization for each module. The number of threads available (determined by the *htIdW* parameter of the *AddModule* ***htd*** command), the maximum number of threads allocated and the average number of threads allocated are also provided along with the average number of threads running. The average number of threads running reflects threads executing HT instructions.

6.3.5 Module Utilization

The *Module Utilization* fields of the performance monitor report provide activity data for the modules and the instructions in the module. The *Active Cycle* field contains the number of cycles each module has a thread active. This number includes all cycles from the time a thread is created in the module to time all threads in the module are terminated.

The *Valid* field contains the number of cycles the module has a valid instruction. A valid instruction count for each instruction in the module is also included. The *Retry* field contains a count of the number of time the *HTRetry* routines executes in the module or instruction.

7 HT Tools

7.1 HT Tool Flow

There are three steps involved in compiling an application with HT: The Hybrid Threading Linker (HTL), the Hybrid Threading Verilog translator (HTV), and the FPGA build. Figure 18 illustrates the HT tool flow.

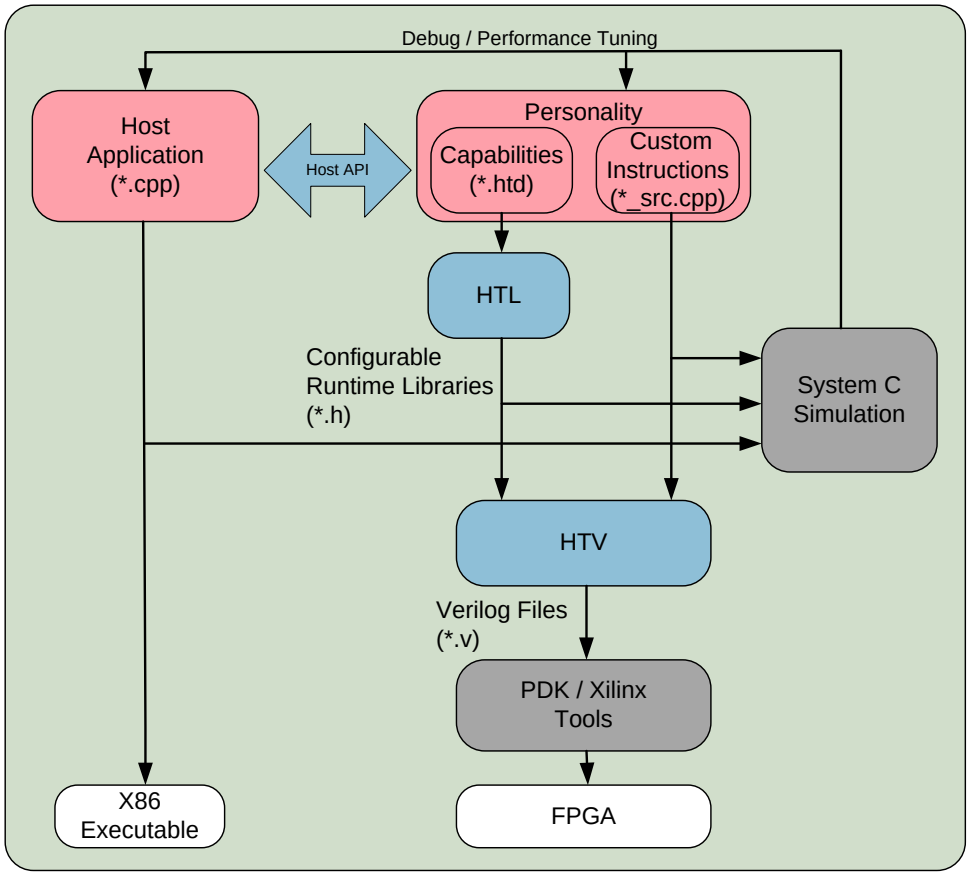


Figure 18 – HT Tool Flow

7.1.1 Hybrid Threading Linker (HTL)

The Hybrid Threading Linker (HTL) generates a framework of interconnected modules. The output generated by HTL is SystemC, which is directly simulated.

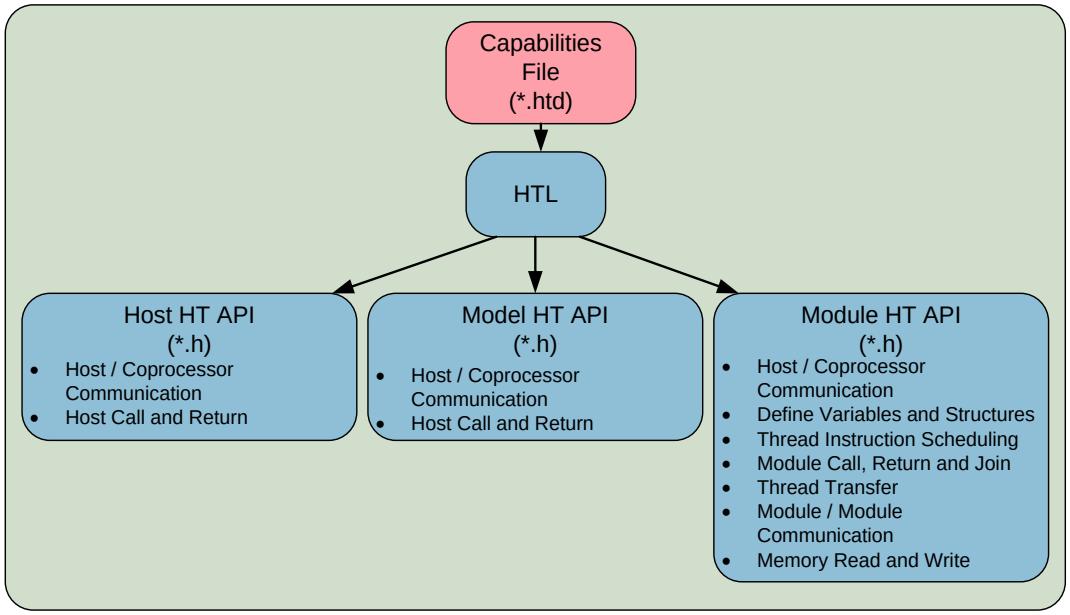
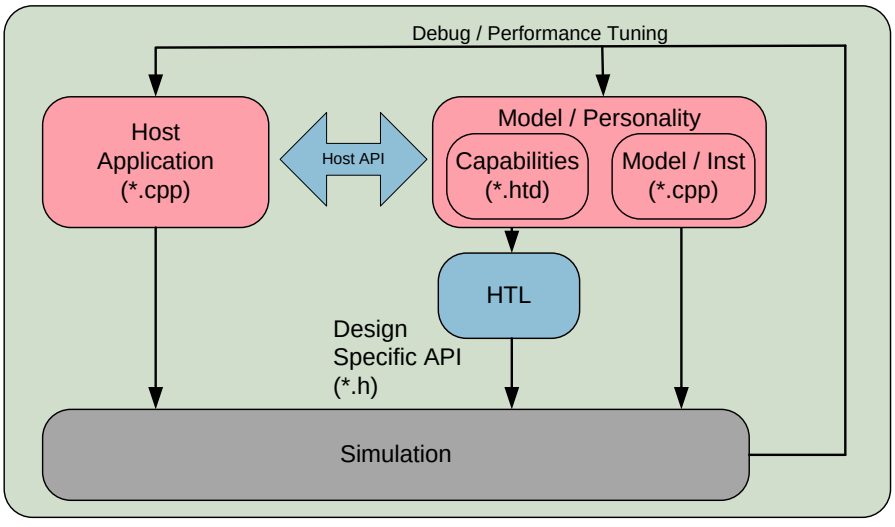


Figure 19- HTL Design Specific API Generation

7.1.2 System C Simulation

The host application can be simulated with the model or the personality. The output of HTL is SystemC. SystemC is a convenient intermediate language that allows the design to be simulated with cycle-by-cycle accuracy, yet is relatively fast compared to HDL simulators (used to simulate Verilog or VHDL). The generated SystemC is instrumented to collect debugging and performance information. An additional benefit of SystemC is that the simulation can be run on any platform with a C++ compiler; it does not require a Verilog simulator.



GDB as well as many Integrated Development Environments (IDEs) can be used in debugging of the SystemC, using the standard run, breakpoint and watch point commands.

There is also an option to produce a VCD file during SystemC simulation. The VCD file can be viewed using common open source viewers.

7.1.3 Hybrid Threading Verilog Translator (HTV)

The Hybrid Threading Verilog Translator (HTV) is the SystemC to Verilog translator. This tool generates Verilog code that will be compiled into the FPGA, along with the Convey PDK infrastructure.

7.1.4 Verilog Simulation

The Verilog generated by HTV can be simulated using ModelSim or VCS. Verilog simulation is limited to a single AE.

SystemC simulation provides the primary means of functional verification. The verilog simulation is much slower and may be difficult to interpret, since signal names may not be as the user defined. However there are circumstances in which the verilog simulation can provide useful insight, such as debugging user provided primitive functionality in verilog form.

7.1.5 Build FPGA

The final step uses the Convey Personality Development Kit (PDK) and the Xilinx ISE development suite to synthesize the generated Verilog into FPGA hardware. Convey provides a makefile environment to ease this process.

7.2 Using HT Tools

Convey provides a complete environment for personality development using the HT tool set. The environment includes Makefiles and scripts supporting all development phases.

7.2.1 Conditional Compilation

The HT toolset defines macros that can be used to optimize the host application for the phase of development. Conditional code may

- Include debug information, appropriate for the target
- Utilize smaller data sets to reduce the time required to simulate
- Build the appropriate number of Units for the target
- Include other target specific actions

The conditions defined by the HT tools are shown in Table 17.

Name	Description
HT_MODEL	Set as *true* for simulation of the host application with the Model Default = *false*
HT_SYSC	Set as *true* for Personality simulation Default = *false*

Name	Description
HT_VSIM	Set as *true* for verilog simulation Default = *false*

Table 17 – Conditional Compilation Options

Conditional compilation macros can be used in both the .htd and the instruction definition (.cpp).

7.2.2 HT Project

A HT project contains all personality specific components. A typical new project will have the following directory structure and components:

```
<project_name>/
      Makefile                  (may be updated by user)
      src/                      (host application source)
            Main.cpp            (Host application)
      src_model/                (model source)
            <name>.htd          (capabilities for model)
            Model.cpp           (Software Model of coprocessor
                                application)
      src_pers                  (personality source)
            Pers<mod1>_src.cpp  (custom instructions for <mod1>)
            Pers<mod2>_src.cpp  (custom instructions for <mod2>)
            Pers<modn>_src.cpp  (custom instructions for <modn>)
            *.htd               (module declarations / capabilities)
            *.hti               (optional advanced features)
            Pers<modn>_prim.cpp (optional modn bbox)
            Pers<modn>_prim.h   (optional modn bbox)
            *.v                 (optional verilog for bbox)
            *.xco               (optional CORE Gen for bbox)
```

Additional directories and files are created by the tools.

7.2.3 HT Makefiles

HT uses makefile templates for all stages of development. This process allows the user to override all or part of the makefile provided by Convey. It also allows the user to switch between revisions of HT with minimal effort, since the makefiles will be updated along with the HT and PDK libraries.

7.2.3.1 HT / PDK Variables

Some variables are used to configure the HT and PDK, while others impact the custom personality and simulation options. These variables are updated for the personality in the project Makefile. Defaults will be used if the variable is not specified.

Variable	*Description*
CNY_PDK	Points to PDK installation, typically /opt/convey/pdk
CNY_PDK_REV	PDK Revision. Default is the revision of the PDK that the version of HT is supplied with
CNY_PDK_PLATFORM	Selects the target platform (hc-1, hc-1ex, hc-2, hc-2ex, wx-690 or wx-2000) Required
CNY_HT_SIG	The personality signature assigned to the custom personality Default = 65000
CNY_HT_NICK	The personality nickname of the signature Default = CNY_HT_SIG
CNY_HT_AEUCNT	Number of unit instances per AE (1-16). Note: There are 4 AEs, so total number of units is 4 * CNY_HT_AEUCNT Default = 1
CNY_HT_ASSERT	Enable hardware assertions Default = false
CNY_HT_FREQ	Personality frequency. Can be set between 50 and 250 (50 – 250 MHz) that is divisible by 25. Default is platform dependent 150 MHz for HC 167 MHz for WX
CNY_HT_SIM_RR	Enables random retry. The SystemC simulation randomly generates busy for all module interfaces. This mode is used to test rarely executed code paths Default = off
CNY_HT_SIM_NRI	No random initialization. The SystemC simulation initialize state variables using a random number generator to assist determining state initialization problem Default = false

Variable	*Description*
CNY_HT_SIM_ML	System C memory latency scaling. Scales the typical memory latency by 0.0 to 10.0. Useful to expose memory dependent race conditions and for analyzing performance under different memory contention conditions. Default = 1.0
CNY_HT_SIM_VCD	Enables value change dump during SystemC simulation. Default = false

Table 18 – HT Variables

7.2.4 Make Options

From the **<project_name>** directory various make options are available to select the target. The options are shown in Table 19.

Option	*Description*
help	Display help for make options
make report	Generate design report (**HtDsnRpt.html**)
make app	Builds the coprocessor application (**app**)
make model	Builds the functional model application (**app_model**)
make sysc	Builds SystemC simulation application (**app_sysc**) and creates the html design report.
make vfiles	Run htv generating verilog
make vsim	Run htv generating verilog Build the verilog simulation application. Create the **personalities/<personality_number>/** directory, containing files needed to install the personality on the system.
make pers	Run Xilinx tools to build the AE FPGA including the Convey supplied infrastructure and the user generated personality Create the **ht.released** directory containing the bit file and other files that may be used when releasing the personality. Puts a pointer to the bit file in the **personalities/<pers_number>/** directory.

Option	*Description*
make libmodel	Builds the functional model interface library (**`libhtmodel.a`**)
make libsysc	Builds the SystemC interface library (**`libhtsysc.a`**)
make libapp	Builds the coprocessor interface library (**`libhtapp.a`**)
make clean	Removes application specific generated files. Should be run between design iterations
make distclean	Removes all generated files. Should be run when changing platforms or when HT Tools are updated.

Table 19 – Project Makefile Options

7.2.5 Running the Application

The host application can be run with the indicated target as shown in Table 20.

Target	**Description**
./app	Run the host application with the target hardware
./app_model	Run the host application with the software model as the target
./app_sysc	Run the host application with the SystemC as the target (SystemC simulation)
./app_vsim	Run the host application with the verilog as the target (verilog simulation)

Table 20 – Application Target Options

7.3 Completed Project

When all development steps are complete the project will have the following directory structure and components:

```
<project_name>/
      app           (Script to run application - created by tools)
      app_model     (Model simulation script – created by tools)
      app_sysc      (System C simulation script – created by tools)
      app_vsim      (Verilog simulation script – created by tools)
      ht/           (Contain personality information created by HT)
      HtMonRpt.txt        (performance monitoring report)
      HtDsnRpt.html       (Design report contains API description,
                          generated by HT)
      Makefile            (may be updated by user)
```

```
personalities/      (created by tools)
      <pers_number>
            ae_fpga.tgz (pointer to the bit file in ht.released)
            context_size
            PersDesc.dat
            rest.0
            save.o
            zero.o
      customdb
src/                (host application source)
      Main.cpp            (application)
src_model/          (model source)
      <name>.htd    (capabilities for model)
      Model.cpp     (Software Model of coprocessor application)
src_pers            (personality source)
      Pers<mod1>_src.cpp  (custom instructions for <mod1>)
      Pers<mod2>_src.cpp  (custom instructions for <mod2>)
      Pers<modn>_src.cpp  (custom instructions for <modn>)
      *.htd               (module declarations / capabilities)
      *.htl               (htl - generated by tools)
      *.hti               (optional advanced - instance def.)
      Pers<modn>_prim.cpp (optional modn bbox model)
      Pers<modn>_prim.h         (optional modn bbox)
      *.v                       (optional verilog for bbox)
      *.xco                     (optional CORE Gen for bbox)
```

8 Customer Support Procedures

Email	support@conveycomputer.com
Web	From www.conveycomputer.com click on **Customer Support**. .

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).
Host Interface	The host interface is a module within each Unit that provides the communication paths between the host application and the Unit.

Term	Definition
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 – Instruction Timing Guidelines

With experience the designer will better be able to determine the functionality that can be performed in a clock cycle. Several examples are provided in Table 21 to help guide the designer.

The default clock rate is 150 MHz, resulting in a 6 ns cycle for HC and .

Operation Description	Example	Time
32 bit non-dependent bitwise operation	ht_int32 a = b / c	.4 ns
32 bit carry chain	ht_int32 b,c ht_int32 a = b + c	2.5 ns
look up table (Xilinx uses 6 input)	ht_unit1 a = b / c & d ^ e / f & g	.4 ns
Block RAM read		3 ns

Table 21 – Example Delays

Loading also impacts delay as shown in Table 22.

Loads	Delay
1	0.1 ns
10	.25 ns
100	1.0 ns

Table 22 – Load Delays