Cadence® Virtual Component Co-Design Architecture Evaluation Guide

Product Version 2.1
March 2001

Contents

Preface

This manual describes how to customize output from the Cadence® Virtual Component Co-Design (VCC) environment for use with a software debugger or a co-verification tool. You customize output from the VCC environment by writing extensions to the Links to Implementation feature in VCC.

This manual shows how to extend Links to Implementation so that it generates

- Probes for software debuggers that the VCC environment does not already support
- Startup and configuration files for co-verification tools that the VCC environment does not already support

This manual is for design flow engineers who integrate third-party tools into the VCC design flow at the customer site for use by system designers and virtual component providers.

The preface discusses

- Related Documents on page 7
- Typographic and Syntax Conventions on page 8

Related Documents

The following documents give you more information about the VCC environment and related applications.

For information about…	**Refer to…**
How to accomplish specific tasks in the VCC environment and what each menu command does	*Help Topics: Cadence Virtual Component Co-Design* (to open VCC Help, start the Create editor and choose the *Help > VCC Help Topics* menu command)
Modeling behavior, architecture, and implementation; strategies for capturing and verifying behavior	VCC Modeling Guide

For information about…	**Refer to…**
The design libraries in the Project Folder	VCC Library Reference
Exporting hardware and software for co-verification; importing cycle-accurate simulation results	VCC Links to Implementation Design Guide
Customizing VCC output for use with a debugger or a co-verifier	VCC Links to Implementation Flow Customization Guide
Creating SPW models, which you can import into the VCC environment	Signal Processing WorkSystem (SPW) documentation library

Typographic and Syntax Conventions

This manual uses the following typographic and syntax conventions.

- Text that you type, such as commands, filenames, and values for dialog box fields, appears in Courier type.

  Example: Type `create` to start the Create editor from a UNIX window.

- Variables appear in Courier italic type.

  Example: *`lib.cell:view`* defines the library, cell, and view for each model in the Project Folder.

- User interface elements, such as field names, button names, menus, menu commands, and items in clickable list boxes, appear in Helvetica italic type.

  Example: From the *Edit* menu, choose the *Data Types* command.

- Menu commands use the > character to indicate menu levels.

  Example: Choose the *Edit > Data Types* command.

Caution

A Caution informs you about possible damage to equipment, data, or software.

Important

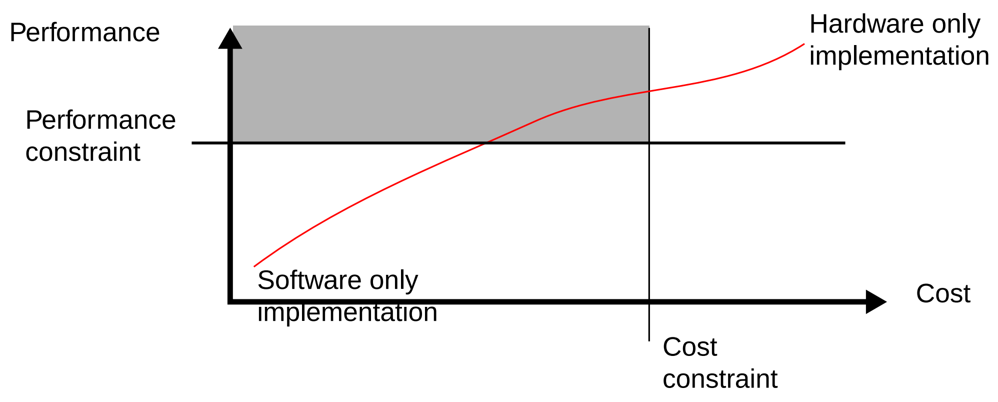
An Important emphasizes valuable information.

1 Overview

The Cadence® Virtual Component Co-Design (VCC) environment lets you integrate intellectual property models to simulate, evaluate, and select appropriate virtual components for digital communication, automotive, and multimedia system design.

The VCC environment differentiates between a behavior model, which determines *what* the system is supposed to do, and an architecture model, which determines *how* the system is supposed to perform.

Using the VCC environment, you can explore independent dimensions of behavior and architecture to reach optimal design performance within your given constraints, as shown in the following figure. The gray area of the graph represents the area where performance and cost constraints are met.



Virtual component model developers or vendors can use the VCC environment to accomplish the following objectives:

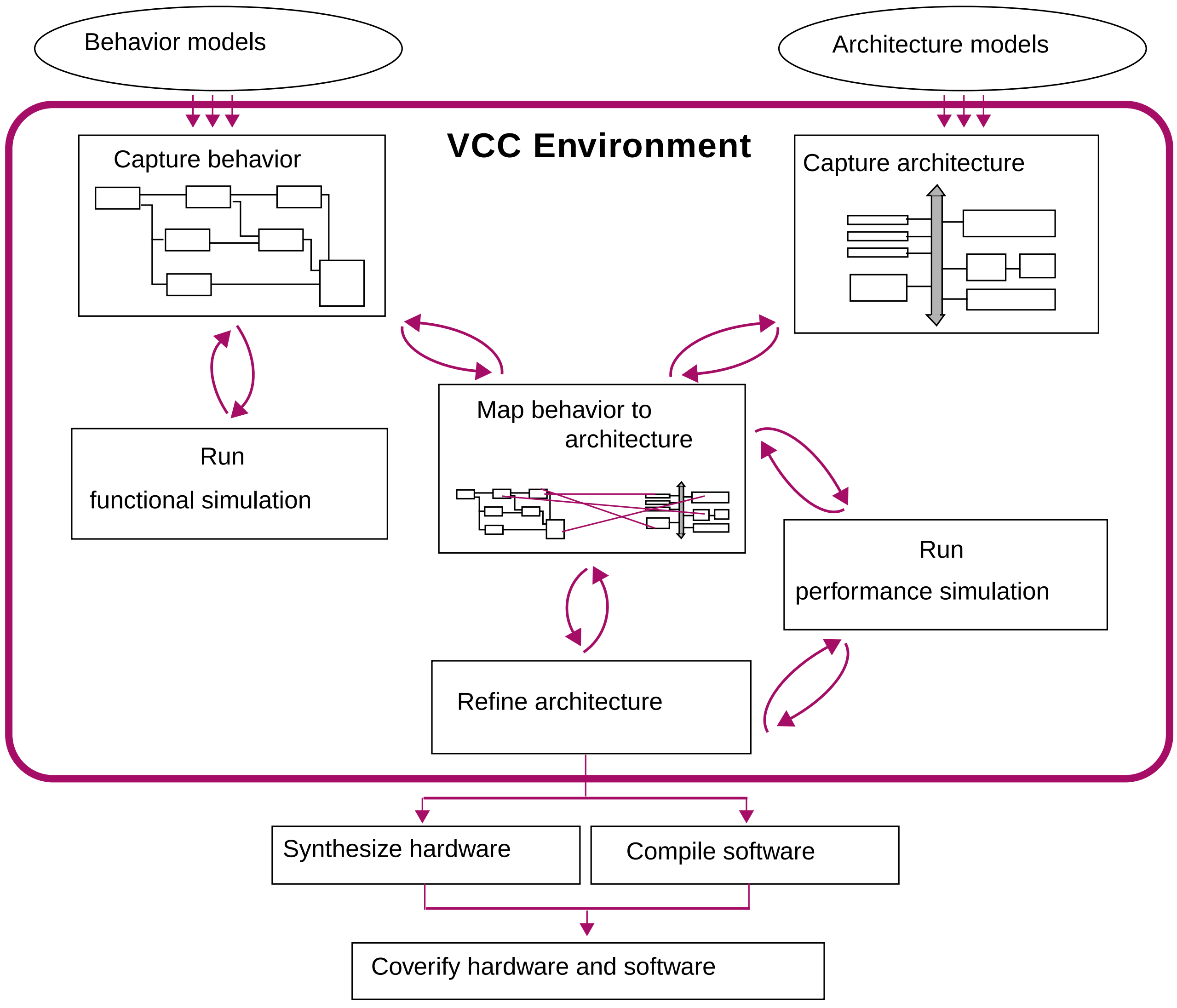
- To provide models of single-chip or chip-set systems for evaluation and selection by prospective customers
- To integrate virtual component models as building blocks into higher level models

Similarly, system designers and architects can use the VCC environment to accomplish related objectives:

- To evaluate performance and select models that meet product requirements
- To integrate internally developed or purchased models of hardware and software blocks from Signal Processing WorkSystem (SPW) and other design tools
- To verify that a given behavior can be performed on an existing architecture and still meet performance requirements
- To explore the feasibility of new behaviors on various architectures

Integration-Based Design Flow

The VCC design flow is illustrated in the following figure.



To begin the process, you capture the desired behavior and verify that the system works through functional simulation. Separately, you capture a potential system architecture using performance models of hardware and software components.

Next, you map the verified behavior diagram to the system architecture diagram. From this mapping diagram, you run a performance simulation to evaluate the performance of the behaviors on the selected architecture. You refine, remap, and reverify until your design meets your performance requirements.

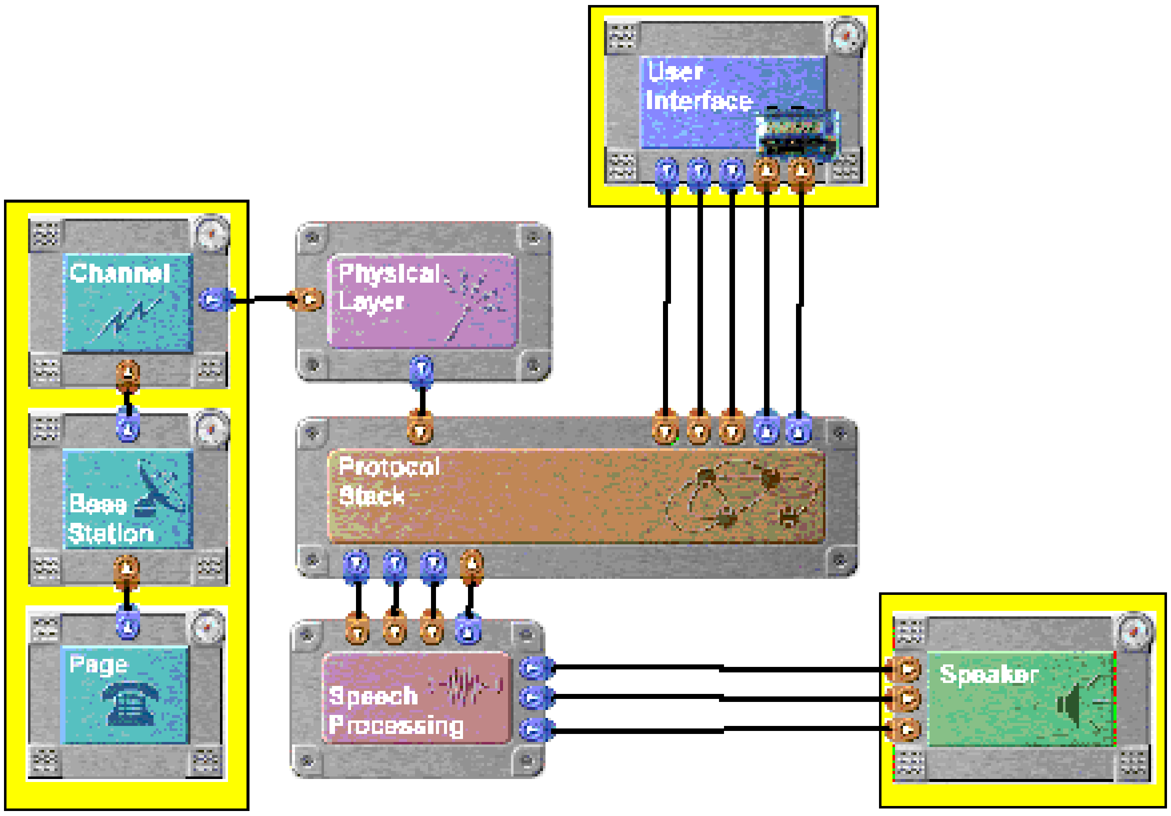
If a desired architecture is too expensive to implement, you can adjust constraints or functionality to accomplish your goal or declare the behavior unimplementable under the current performance constraints.

Capturing Behavior

Behavior models represent functional system blocks and testbenches. When capturing the behavior of a system design, you specify behavior as high-level behavior models without implying an eventual architecture implementation.

In the VCC environment, a behavior primitive is mapped to a single architecture primitive. It cannot be partitioned across several architecture models. If you want to explore mapping parts of a behavior on separate architecture models, you should compose it as a hierarchy of models. This approach lets you map all the segments on to one architecture model or individual segments on to separate architecture models.

You use the Behavior Diagram Editor to create your behavior diagram. For a new behavior diagram, you instantiate symbols that represent behavior models and set model properties to define model and port characteristics. You connect the input and output ports of the behavioral models to create a system diagram, as shown in the following figure.



Behavior Modeling Tools

You can use various tools to create libraries of behavior models depending on the role of the models in your diagram.

To import new or existing models, you can use the following tools:

- Use SPW to specify DSP-oriented applications and other synchronous data flow models.
- Use the Telelogic SDT tool to specify graphical SDL models of behavior. This tool is a standard in the telecommunications industry.

To create models in VCC, you can use the following tools:

- Use the STD Editor to specify graphically communication refinement and similar co-design finite state machine (CFSM) models.
- Use the text editor in the VCC environment to specify textual SDL models for communication refinements and for CFSM models.
- Use a C software development tool or a text editor in the VCC environment to specify Whitebox C models for embedded software.
- Use a C++ software development tool or a text editor in the VCC environment to specify C++ models for modeling general functional blocks.

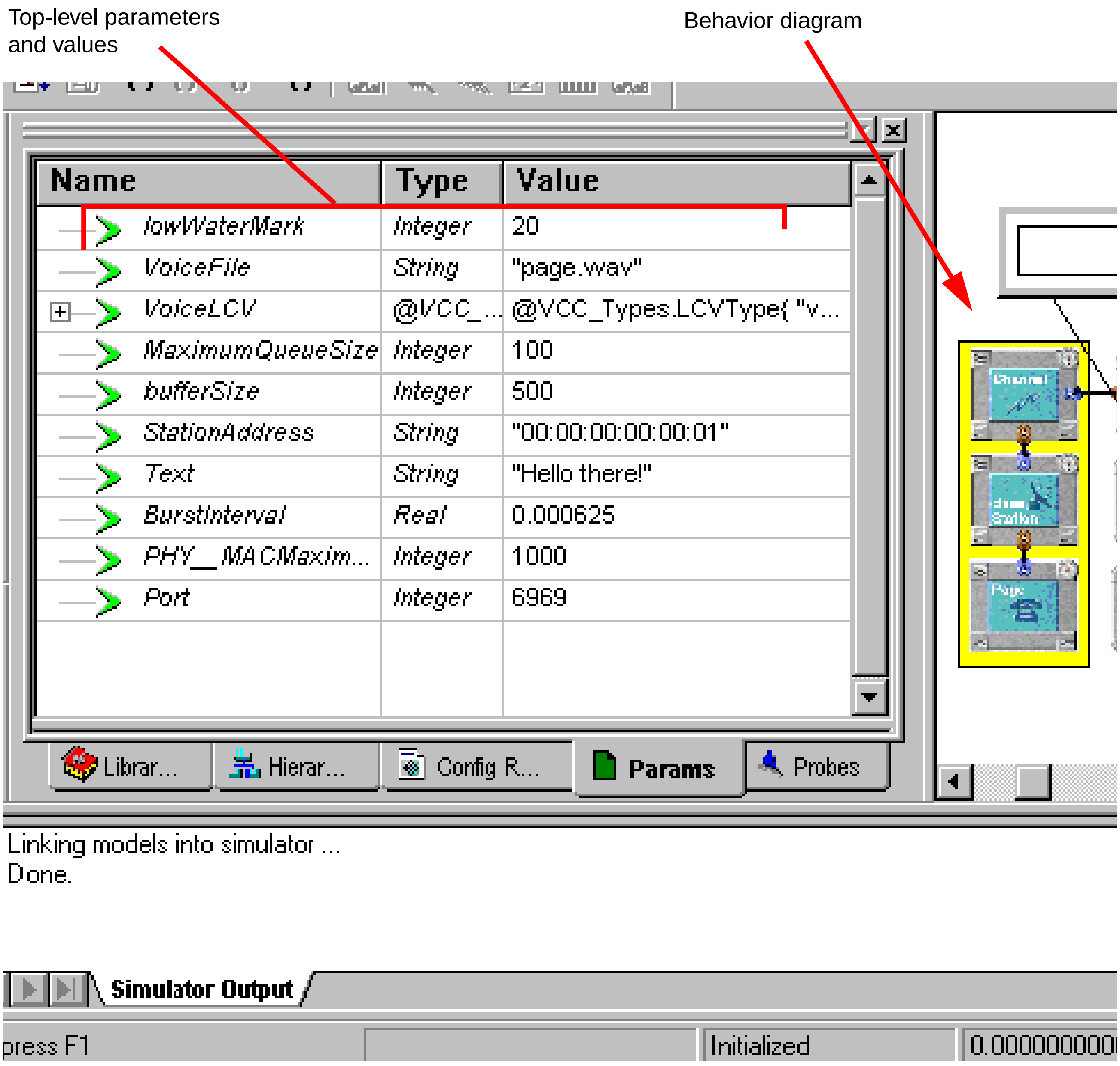
Running a Functional Simulation

You should verify the integrity of your behavior design through functional simulation before evaluating the performance of the system. Turnaround time in functional simulation is shorter than in a performance simulation because execution of the blocks is instantaneous. Functional simulation, therefore, is preferable for integrating and debugging behavior models.

In functional simulation, you

- Create an analysis session for your behavior diagram.
- Set design parameters and sweep parameters.
- Place probes and display objects in your design so you can check results.

- Use the Visualize tool to postprocess simulation results.

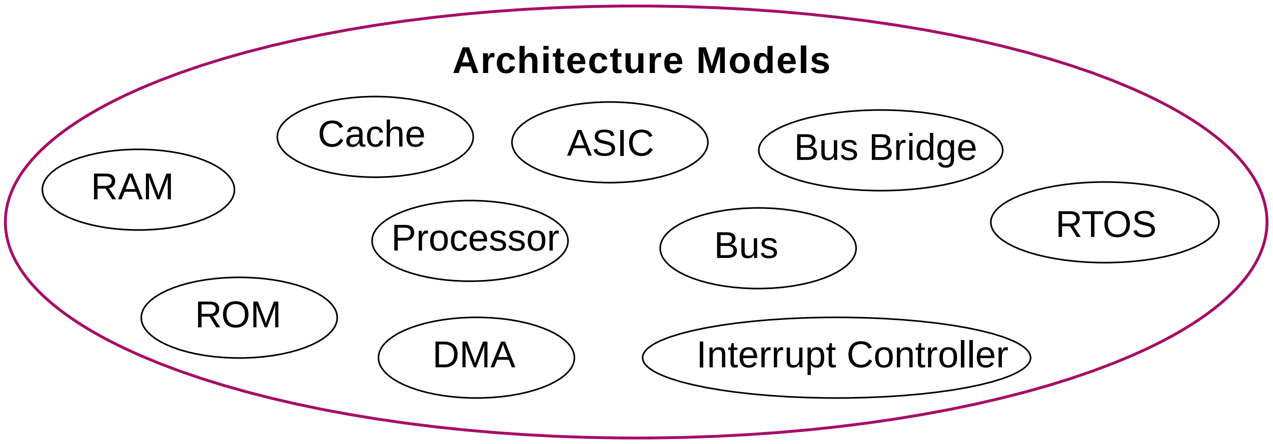


Name	Type	Value
lowWaterMark	Integer	20
VoiceFile	String	"page.wav"
VoiceLCV	@VCC_...	@VCC_Types.LCVType("v...
MaximumQueueSize	Integer	100
bufferSize	Integer	500
StationAddress	String	"00:00:00:00:00:01"
Text	String	"Hello there!"
BurstInterval	Real	0.000625
PHY__MACMaxim...	Integer	1000
Port	Integer	6969

```
Linking models into simulator ...
Done.
```

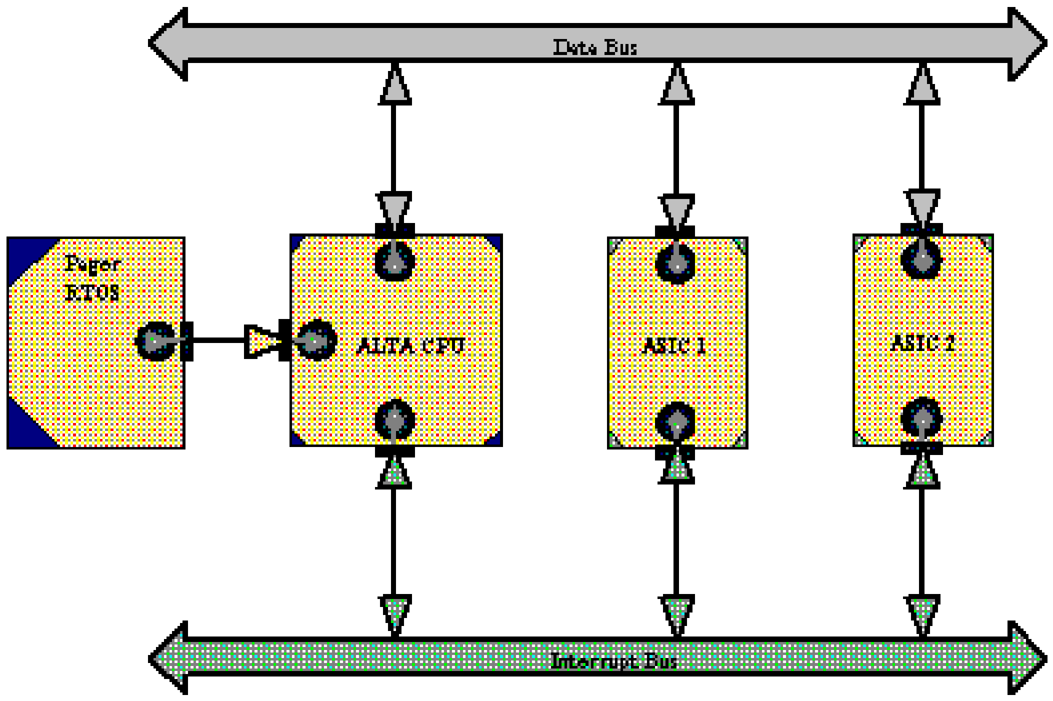
Capturing Architecture

In the VCC environment, potential architectures consist of the models shown in the following figure.



Initially, to capture your architecture diagram, you use the Architecture Diagram Editor. This tool lets you quickly enter a relaxed architecture that represents a target implementation architecture. In a relaxed architecture diagram, you do not need to define every port and signal—you need only define the basic topology.

In an architecture diagram, you instantiate symbols representing various architecture models, such as a bus, RTOS, and processor. You add ports and connect the models to appropriate buses, as shown in the following figure. Architecture properties define performance and implementation characteristics.

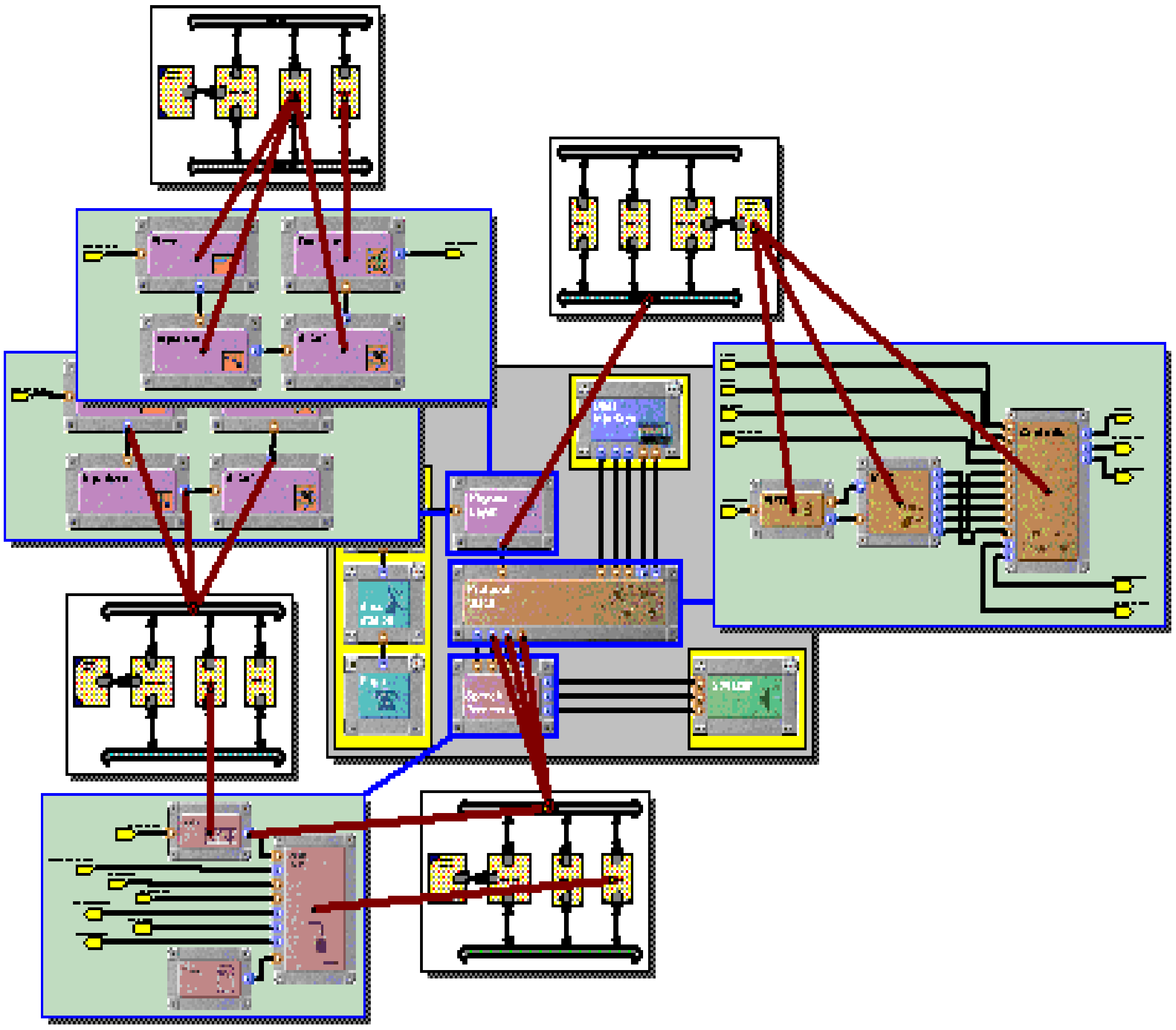


A performance model must be specified for each architecture primitive model. For example, a processor is characterized by delays of executing instructions.

Mapping Behavior to Architecture

Mapping makes the connection between behavior models in a behavior diagram and architecture models in an architecture diagram.

The Mapping Diagram Editor lets you map the behavior to the relaxed architecture, as shown in the following figure.



Mapping defines the hardware and software partitions. For example, behaviors mapped to a scheduler associated with a processor are implemented in software.

Mapping also identifies a performance model that provides the basis for assessing realistic delays incurred if the behavior is run on the mapped architecture.

Architecture models represent performance models that can be used to analyze an implementation of a behavior. Performance models can be precharacterized from an actual implementation or can be user-defined to characterize expected performance based on projected budgets or constraints.

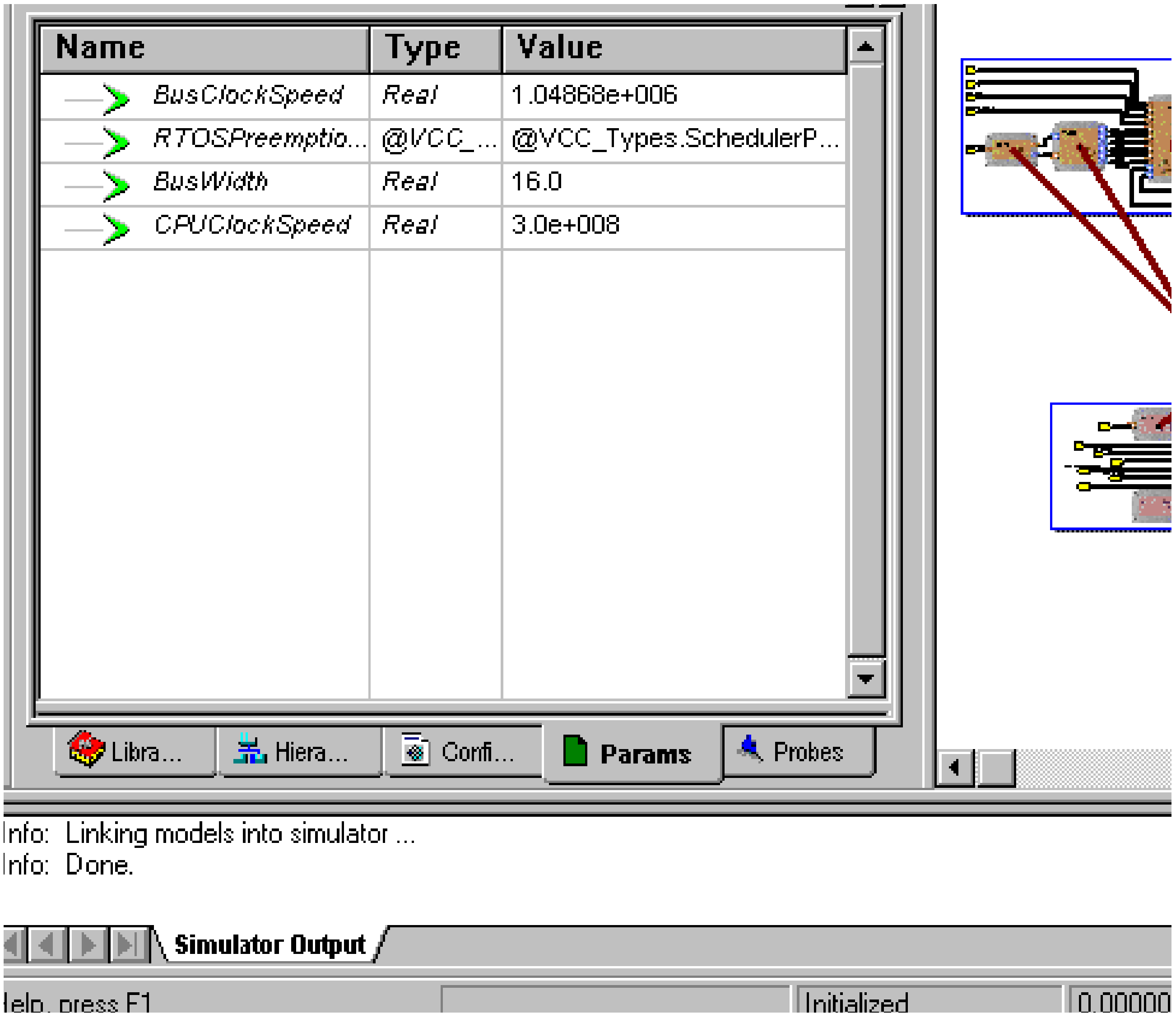
Running a Performance Simulation

The performance simulator evaluates the performance of the mapped diagram. Performance simulation can

- Identify missed events
- Estimate system-level performance
- Provide data on the usage of processors, buses, and other shared devices

By inserting probes in your mapping diagram, you can collect and graphically display data to assess and compare system function.

The following figure shows the performance parameters and their settings for a simulation run.



Name	Type	Value
BusClockSpeed	*Real*	1.04868e+006
RTOSPreemptio...	@VCC_...	@VCC_Types.SchedulerP...
BusWidth	*Real*	16.0
CPUClockSpeed	*Real*	3.0e+008

```
Info:  Linking models into simulator ...
Info:  Done.
```

You can remap behaviors to different architecture models, then rerun the simulation to analyze the effects on performance. You can sweep simulation parameter values to compare the performance using different characteristics.

Refining Mapping Diagrams

You can refine your mapped diagram to include more design decisions by

- Analyzing behavior delay

- Analyzing bus traffic
- Refining communication patterns
- Analyzing memory access

In addition, behavior refinements can include refining communication, specifying refined control protocols, or changing abstract data types to base data types that more closely represent the actual implementation.

Exploration

You can use the VCC environment to explore various possibilities for new product or feature design. The following sections offer several ways to use the features of the VCC environment depending on the goals of your exploration.

Derivative Design

In the VCC environment, you can explore the effects of adding new functionality to existing products.

- Create behavior models for additional features to enhance your original product design.
- Using the mapped diagram from your original product, create multiple mapping diagrams that combine some or all of the new features.
- Simulate and analyze the results.
- Experiment with remapping and resimulating, adding and dropping new behavior models, to obtain combinations that maintain the performance of the original product.

Hardware/Software Partitioning Changes

To explore cost-effectiveness and high performance, you can evaluate the benefits and drawbacks of partitioning various behavior models in hardware or software.

- Use the *Save As* option to save your current mapping diagram.
- Change the mapping of various behavior models to assess the benefits or drawbacks in implementing certain behaviors in hardware versus software.
- Sweep parameters in all mapping diagrams to compare statistics about each method.

“What If” Scenario

The VCC environment can be used effectively to explore potential architecture design combinations. In this type of exploration, you create architecture models and vary performance property values until a design meets your budget or constraint requirements. Once you have determined a suitable architecture, you can search vendor libraries to find virtual component models that meet your model specifications.

- Create and debug a mapping diagram that instantiates your behavior diagram and an architecture diagram that seems appropriate for your needs.
- Export architecture model performance properties that affect your budget or constraints.
- Simulate, analyze, and debug results of the first design’s performance.
- Use the Save As option to adjust your initial architecture diagram, using a naming convention that emphasizes the differences in your diagrams. Create, debug, and simulate a mapping diagram for each modified architecture.
- Adjust simulation parameter values and resimulate.
- Sweep parameters in several of your mapping diagrams to analyze multiple options.
- Use Visualize to display multiple windows for statistics and Gantt chart comparisons.

2

Performance Models

Because the performance of a behavior model is dependent on the architecture, the behavior performance model is described in terms of the services supported by the architecture. For example, the performance model of a behavior model mapped to an RTOS does not know the execution time for the behavior model—the execution time is dependent on the clock speed of the processor and the instruction set supported by the processor. The behavior model only contains the number of instructions to be executed. The architecture service contains the information about the clock speed and instruction set of the processor.

As another example, a behavior model accesses memory for instruction and data fetching. The performance impact of a memory access is dependent on the load of the bus, the transfer rate on the bus, and the memory latency. To model this, VCC supports behavior models that reference services supported by the architecture.

In the first example, the behavior model specifies the number and type of instructions in the behavior model, whereas the performance model of the processor models the cost of an instruction. In the second example, the performance model of the behavior specifies the type of memory access (read/write) and the data size of the request. The performance model of the processor converts the single read/write request into the appropriate number of bus requests, the bus adapter requests ownership of the bus, and the bus model accounts for the performance impact of arbitration. Finally, the appropriate RAM or ROM accounts for the read/write latency. Thus the performance impact is distributed to each of the architecture elements involved in the memory access.

Performance Models of Architecture Components

The performance model of an architecture component is defined by a set of architecture services. These services can be associated with the component itself or with a port on the component. Each service models the performance of a particular function supported by that architecture component. For example, an RTOS might define the following services:

- Scheduler service for arbitrating multiple tasks running on the same CPU
- Standard C library service for modeling standard C functions like *memset*, *memcpy*, and *sizeOf*

- SWInterrupts service for modeling software interrupts
- SWMutexes service for modeling mutual exclusion

Each of these services supports a number of functions. The implementation of a particular function specifies the performance impact of executing that function. For example, the standard C library service models the performance impact of executing the *memset*, *memcpy*, and *sizeOf* functions on a particular RTOS. These services do not implement the functions, rather, they specify the performance impact of calling these functions. In this example, the *memcpy* function blocks for the amount of time needed to write the data to memory—it does not make a copy of the data.

In VCC, the set of functions supported by a service is modeled as a Blackbox service declaration using a C++ header file. The implementation of these functions is modeled as a blackbox service definition using C++. VCC supplies a set of service declarations in the VCC_ServiceDeclarations library and a set of matching service definitions in the VCC_ArchitectureServices library. In the case of the RTOS example, the *VCC_ServiceDeclarations.LibCDeclaration:blk_serviceDecl* declares the *memset*, *memcpy*, and *sizeOf* functions, and the *VCC_ArchitectureServices.StandardCLibrary:blk_service* provides the performance impact of each these functions.

The following is the *blk_serviceDecl.h* file from the *VCC_ServiceDeclarations.LibCDeclaration:blk_serviceDecl* cellview:

```
#ifndef VCC_ServiceDeclarations_LibCDeclaration_blk_serviceDecl_blk_protoface_h
#define VCC_ServiceDeclarations_LibCDeclaration_blk_serviceDecl_blk_protoface_h

#include <sim/ModelSupport.h>

template<class portType>
class simulateDLLExport VCC_ServiceDeclarations_LibCDeclaration_blk_serviceDecl :
public serviceDeclaration
{
protected:
     VCC_ServiceDeclarations_LibCDeclaration_blk_serviceDecl(const ModuleProto&
     a, InstanceInit& b)
       : serviceDeclaration(a,b) {};
     VCC_ServiceDeclarations_LibCDeclaration_blk_serviceDecl()
       : serviceDeclaration() {};

public:
     virtual vccAddress* memcpy(vccAddress* s1, const vccAddress* s2, size_t n,
          vccInstance*) = 0;
     virtual vccAddress* memset(vccAddress* s, int c, size_t n, vccInstance*) = 0;
     virtual size_t sizeOf(const typeDefinition& type,vccInstance*) = 0;
     virtual size_t sizeOf(const typeObject& data,vccInstance*) = 0;

     void* preCast() {return this;};
};

#endif
```

The following is the *blk_service.cpp* file from the *VCC_ArchitectureServices.StandardCLibrary:blk_service* cellview:

```
#include "blk_service.h"

#include <sim/ModelSupport.h>

static const
VCC_ServiceDeclarations_MemoryAccessDeclaration_blk_serviceDecl<typeObject>::
     readWrite rwWrite =
VCC_ServiceDeclarations_MemoryAccessDeclaration_blk_serviceDecl<typeObject>::
     write;
static const
VCC_ServiceDeclarations_MemoryAccessDeclaration_blk_serviceDecl<typeObject>::
     readWrite rwRead =
VCC_ServiceDeclarations_MemoryAccessDeclaration_blk_serviceDecl<typeObject>::
     read;

CPP_MODEL_IMPLEMENTATION::CPP_MODEL_IMPLEMENTATION(const ModuleProto &proto,
     InstanceInit &inst)
: CPP_MODEL_INTERFACE(proto, inst), initialized_(false)
{
}

void CPP_MODEL_IMPLEMENTATION::Initialize(const char* name, vccInstance* block)
{
}

void CPP_MODEL_IMPLEMENTATION::Init()
{
     if (initialized_)
          return;
     initialized_ = true;

     bytesPerWord_ = typeInfo.getSizeOfWord();
}

/*******************************************************************
Name:          memcpy
Description:   Copies n bytes from memory address s2 to s1.  This
               implementation assumes that addresses lies on memory
               word boundaries.  No actual data is designed to be
               stored or read.  Memory transactions are generated for
               performance reasons only.
Return:        Always returns s1.
*******************************************************************/
vccAddress* CPP_MODEL_IMPLEMENTATION::memcpy(vccAddress* to, const vccAddress*
from, size_t n, vccInstance* inst)
{
     MS_DEBUG(2) InfoOut << Start << "memcpy()" << End;

     if (n == 0)
          return to;

     Init();

     unsigned reqTrans = n / bytesPerWord_;
     reqTrans = ((reqTrans==0) ? 1 : reqTrans);
     unsigned remTrans = reqTrans;
theBegin:
     if (remTrans == 0) goto theEnd;
     {
          if (from)
               memAccess.reference(*from, bytesPerWord_, rwRead,inst,true);
     else {
          // tbd
     }
     if (to)
          memAccess.reference(*to, bytesPerWord_, rwWrite,inst,true);
     else {
          // tbd
          }
     }

     remTrans--;
     goto theBegin;

theEnd:
     return to;
}


/*******************************************************************
Name:          memset
Description:   Sets the first n bytes of memory at address s to the value
               of c (converted to an unsigned char).  Memory transactions
               are generated for performance reasons only.
Return:        Always returns s;
*******************************************************************/
vccAddress* CPP_MODEL_IMPLEMENTATION::memset(vccAddress* s, int c, size_t n,
     vccInstance* inst)
{
     MS_DEBUG(2) InfoOut << Start << "memset()" << End;

     if (n == 0)
          return s;


     Init();

     vccAddress* to = s;

     unsigned remTrans = n;

theBegin:
     if (remTrans == 0) goto theEnd;
     {
          if (to)
               memAccess.reference(*to, bytesPerWord_, rwWrite,inst,true);
          else {
               // tdb
          }
     }

     remTrans--;
     goto theBegin;

theEnd:
     return s;
}
```

```
size_t CPP_MODEL_IMPLEMENTATION::sizeOf(const typeDefinition& type,
     vccInstance* srcBehInst)
{
     return typeInfo.getTypeSize(&type);
}

size_t CPP_MODEL_IMPLEMENTATION::sizeOf(const typeObject& data,
     vccInstance* srcBehInst)
{
     return typeInfo.getTypeSize(data.typeOf());
}
```

In VCC, the service declarations are associated with the symbol view of the architecture component, and the service definitions for each service declaration are specified in the performance view of an architecture component. For example, a symbol of the RTOS declares that the RTOS supports the *VCC_ServicesDeclaration.LibCDeclaration:blk_service*. Then, in the performance view of the RTOS, the performance impact of these functions is defined in the *VCC_ArchitectureServices.StandardCLibrary:blk_service*.

The separation of the declaration and the definition makes it easier for you to change the level of accuracy for the performance analysis. You can also choose a different performance type (such as power versus area) by binding a different performance model of the architecture component.

If the performance impact of a service depends on its interaction with another service (either on a different architecture component or its own), the service declares that it uses another service. Specifically it uses another service declaration because it calls a specific C++ function with specific arguments. The service makes a direct call to that function using the *use* handle name, function name, and function arguments.

For example, the *memcpy* function uses the *memoryAccess* service on the CPU to specify the delay for writing the data into memory. The model of the standard C library service calls the *reference* function from the *memAccess* service:

```
memAccess.reference(*from, bytesPerWord_, rwWrite, inst, true);
```

Where *memAccess* is the handle declared for the *memoryAccessDeclaration* in the use clause and *reference* is declared as a function that takes five arguments in the *VCC_ServiceDeclarations.memoryAccessDeclaration:blk_serviceDecl* cellview. In the architecture diagram, the CPU assigned by the RTOS must support the *memoryAccessDeclaration* for this call to be successful. Distributing the performance models among the architecture components provides the ability to explore different architectures quickly. You can easily change architecture components when the components support the same service declarations.

The service definitions can be parameterized. These parameters can be set inside the performance model of the architecture component, or they can be exported to the architecture instance in the architecture diagram. Parameters make the service definition reusable in different architecture components and architecture diagrams.

Typical Architecture Services

Service declarations and service definitions can be associated with any architecture resource, and you can also design your own services. The purpose of this section is to provide information about a typical set of services that are applicable to each of the architecture resources.

Architecture Resource	Applicable Services
Processor	Delays
	CPUInstructionCost
	CPUMemoryAccess
	CPUInterruptController
	CPUSingleStack
	CPUTypeSize
	FCFSBusAdapter
RTOS	SWInterrupts
	StandardCLibrary
	SWMutexes
	RoundRobinScheduler
	SWTimers
ASIC	AsynchronousDelays
	StaticPriorityHWScheduler
	FCFSSlaveAdapter
	FCFSBusAdapter
	InterruptBusAdapter
	Memory
	ASICTypeSize

Architecture Resource	Applicable Services
Data Bus	FCFSArbiter
Interrupt Bus	InterruptBus
Memory	FCFSSlaveAdapter Memory

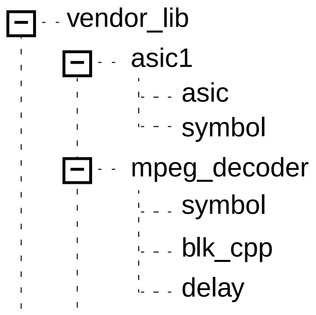
Processor Models

A performance model for a processor should account for the cost of executing instructions. This facilitates writing behavior performance models that are specified in terms of instructions. The processor should also model the performance impact of hardware scheduling and interrupt requests. In addition, the processor should model memory requests (including stack management) and interaction with the bus. These services in turn are dependent on services supported by the connected data bus and the interrupt controller.

Predefined ASIC Models

Predefined ASICs are modeled by virtual component providers or library developers by creating an architecture model representing the actual hardware device, one or more behavior models representing the behaviors associated with it, and at least one performance model to support delays, memory access, and bus adapters.

For example, an ASIC provider might supply a library with an MPEG decoder model that consists of an ASIC architecture model, a behavior model providing the algorithm, and one or more performance models.



Using this example, you would instantiate the `asic1` symbol in your architecture diagram and choose `asic` as your performance model. The `asic` performance model provides services

- For scheduling behaviors mapped to it

- For interactions with the databus and interrupt bus

In your behavior diagram, you would instantiate the `mpeg_decoder` symbol and make sure `blk_cpp` contains the behavior that models the functionality of an MPEG decoder. In your mapping diagram, you would map these two models to each other.

The behavior performance model, `delay`, describes the performance impact of running the decoder on the asic. You do not need to choose the behavior performance view because the performance model is preset by the `BehaviorPerfViewName` property in the ASIC model properties.

Refer to your vendor library documentation to determine any link parameter values required by the device.

Custom ASIC Models

A custom ASIC acts as a placeholder for a hardware model that is in development. You need to create an architecture model for the ASIC, one or more behavior models that model the ASIC function, and performance models that characterize its expected performance based on projected budgets or constraints. Refer to the *VCC Modeling Guide* for details about creating a custom ASIC and a DSL performance model.

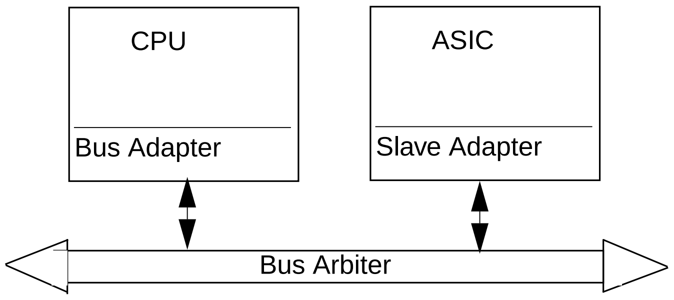
You are prompted for exported parameters for every performance model. You need to set only those parameters that are used in the configuration you have selected.

A scheduler model must be included internally as part of the model. If the ASIC includes a scheduler that is parameterized, you might need to assign link parameters.

Data Bus

The performance model for a data bus should model the arbitration for bus ownership. Examples of different bus arbiters are first-come-first served, time-sliced, and broadcast. The arbitration of the bus is typically requested by the bus adapter service on the bus master (such as the processor).

Once the adapter has ownership of the bus, the adapter starts transferring the data to the slave adapter (such as to the memory or ASIC). If the bus transaction is addressed to a particular slave, the appropriate slave adapter must first be identified by looking up the symbolic address in the bus registry. Therefore, each slave adapter must register itself with the bus registry.



Interrupt Bus

The performance model for an interrupt bus should model the contention caused by multiple simultaneous interrupt requests. The interrupt bus should decide the winner and send the request to the interrupt handler (on the processor).

The processor supports an interrupt controller service, which manages the interrupt service routines (ISRs). The ISRs are registered with the interrupt controller. When the interrupt controller gets an interrupt request, the appropriate ISR is scheduled based on the interrupt priority and the currently running task.

Memory

The performance model of a memory should model the read/write latency. A *slaveAdapter* service on the port models the interaction with the bus. The memory typically does not manage the actual data being read/written— it only models the performance impact of completing the read or write.

The memory can also be supported by a cache or DMA on the architecture diagram.

Bus Bridge

The performance model of a bus bridge should model the transfer of data from one bus to another. The bridge typically has a slave adapter on the primary bus and a *busAdapter* (master) on the external bus. When a communication is requested from one architecture component to another, VCC automatically determines the set of buses and bridges needed to complete the path through the architecture diagram.

Because bus bridges can support multiple buses, the legal paths through the bridge must be specified by setting the *vccOutputPortName* parameter on the port.

Additional Architecture Services

Each architecture component in the architecture diagram supports a set of services. Additional services might be appropriate based on the communication paths between these architecture components. For example, a device driver might be useful for the software to communicate with a particular ASIC. However, that device driver is dependent on the processor and possibly the RTOS used in the architecture.

VCC lets you add services to an RTOS in the architecture diagram. These services are defined by a handle name, a reference to a service definition (cellview name), and parameter values required by the service definition. This service definition can use services provided by the RTOS or the processor.

Behavior Performance on Software Architectures

In order to analyze the performance of your system on a specific architecture, you need to specify performance models for each behavior. These performance models can be port delays with parameterization for data type sizes. This approach might be sufficient for hardware, but for software, the performance model is also affected by the following:

- Execution Delays
- Delays Based on the Processor
- Delays Based on Memory Accesses
- Performance Modeling Styles
- Scheduling Shared Resources

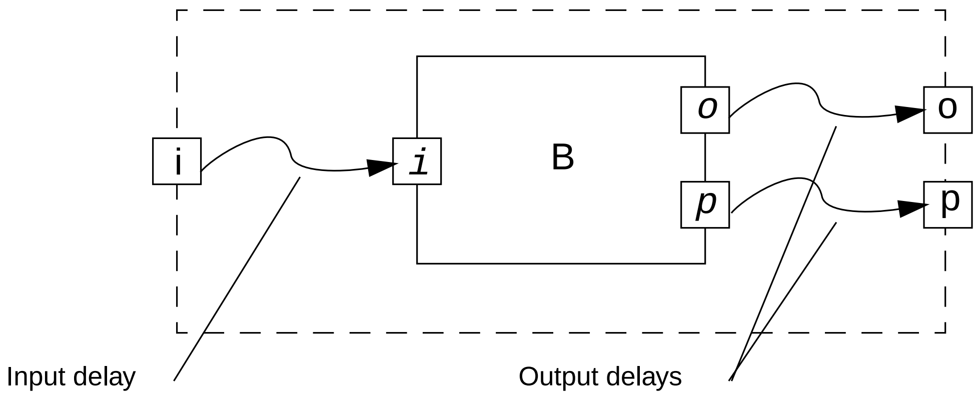
Execution Delays

A functional model executes with discrete event semantics. An execution is called a reaction. The reaction samples inputs, computes responses, and emits outputs, all in the same instant.

To make a functional model useful for performance simulation, it must be associated with a temporal model that specifies the execution delays of the reaction. In the VCC environment, you associate a performance model with the functional model.

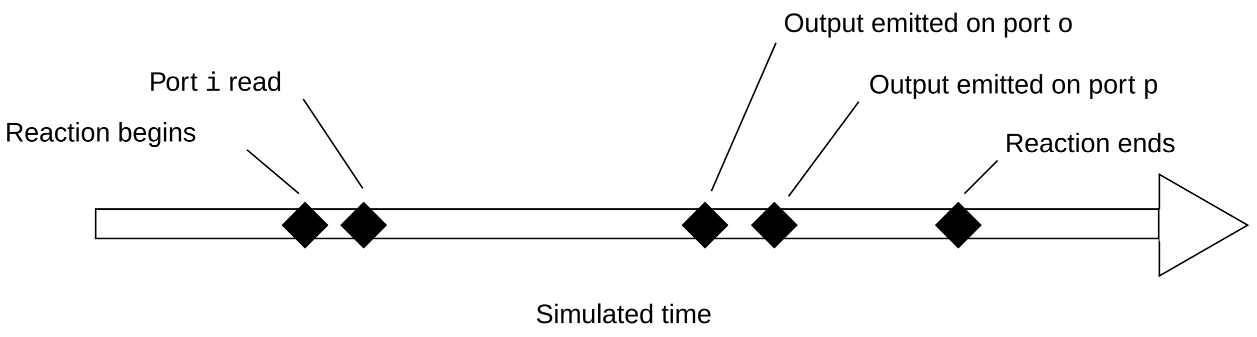
To model the performance of a behavior on some target architecture, you need to specify the instants at which values on input ports are read, the instants at which values on output ports are posted, and the instant at which the reaction ends.

The performance model can be represented as a wrapper around the block, as shown in the following figure.



The performance model causes the reaction to take time. If new values arrive on the input ports faster than the reaction time of the model, events could be lost or missed.

The following figure provides an example of a performance model timeline.



A performance model represents the temporal model of a reaction. The VCC environment defines three types of performance models: annotated C performance models, annotated C++ performance models, and delay scripting language (DSL) performance models.

Delays Based on the Processor

The performance of a behavior running on software is affected by the clock speed of the processor. It is advantageous to parameterize the performance model based on the clock speed of the processor in the mapped design. You can accomplish this in the VCC environment by referencing the CPS parameter on the processor from within your performance model. (For information about the CPS parameter, see the “Architecture Models” chapter of the *VCC Modeling Guide*.)

Similarly, the performance of a behavior running on software is affected by the instruction set supported by the processor. For example, a branch or load instruction on a particular processor might require cycle counts different from those of another processor. In order to explore different instruction sets, it is useful to parameterize the performance model based on the processor chosen in the mapped design. You can accomplish this by referencing the processor basis associated with the processor. (For information about the processor basis, see the “Architecture Models” chapter of the *VCC Modeling Guide*.)

The number of registers on the processor as well as the technique for register allocation also affects the performance of behaviors running on the processor. It is advantageous for the performance model of the behavior to account for usage of registers. In the VCC environment, the annotated C modeling style supports different register allocation techniques for more accurately modeling this impact. (For information about the annotated C modeling style, see the “Architecture Models” chapter of the *VCC Modeling Guide*.)

Delays Based on Memory Accesses

Execution of a behavior on a processor requires memory accesses for instructions and data that are assigned to specific segments in an architecture memory. The processor must issue a request to access the memory and transfer the memory segment across the bus. In addition, there is a performance impact of accessing data from a processor stack. This can be modeled in the VCC environment by declaring behavior memory segments and mapping them to architecture memories.

In the performance model, the memory segments are declared by name and size. The size specification is of type *String*, which can be parameterized by the size of the instruction. The following example shows a performance model that has two memory segments:

```
segmentName = "data" with size = "1024"
segmentName = "text" with size = "OP.i + LD + OP.i + ST + OP.i + IF"
```

During mapping, these memory segments are collected and presented as a single, uniquely named list of memory segments per processor. These memory segments are used in the linker/loader stage when building the software image.

The complete list of memory segments are gathered from

- Performance views of all behaviors mapped to the RTOS or nested scheduler.
- Services associated with the memory references and timer references of the behaviors mapped to the RTOS or nested schedulers.

- Sender services of patterns associated with the output ports of the behaviors mapped to the RTOS or nested schedulers.
- Receiver services associated with the input ports of the behaviors mapped to the RTOS or nested schedulers.
- Services supported by the RTOS or any nested scheduler.
- *Instantiate* clauses in services. For example, timerTickISR, ISRs for interrupt patterns, and polltaker behavior.
- Software explicitly added to the RTOS using the *Add Software* command in the Mapping Diagram Editor.

Example

Behavior instance “b1” has the following memory segments:

```
"data" of size "256"
"d1" of size "256"
```

Behavior instance “b2” has the following memory segments:

```
"data" of size "256"
"d2" of size "512"
```

As a result, the processor has the following linker segments:

```
"data" (calculated size of 512)
"d1" (calculated size of 256)
"d2" (calculated size of 512)
```

In this example, the processor linker segment “data” has a calculated size of 512 because the “data” memory segments from instances “b1” and “b2” are combined.

You map each linker segment to an architecture memory by specifying “Memory Participant” and “offset”. You also specify the type of the segment, which can be “CODE”, “DATA”, “STACK”, or “HEAP”.

The VCC software can automatically calculate the “size” field for each processor segment by adding the sizes of the individual segments of the behaviors mapped to the processor. You can also override this value. The “size” field is represented by a composite type with a field for “size” and a field for “Fixed” or “Floating”. If “Fixed,” you specify an integer for the size. If "Floating," the VCC software gathers the sizes and uses the *processBasis* file to convert strings to unsigned.

VCC uses the *directMemoryRead* pattern to model the communication of the processor to the specified architecture memory for instruction and data fetches.

Performance Modeling Styles

There are three types of performance models:

- Delay scripting language (DSL) performance models
- Annotated C performance models
- Annotated CPP performance models

DSL Performance Model

The DSL performance model is a method of associating a delay model with a functional model using the delay scripting language (DSL). A DSL performance model can be used in the following situations.

- When the code is not yet fully developed
- When using a blackbox behavior model in which the code is not accessible
- When a processor basis file does not exist
- For hardware

Under certain circumstances, VCC supports backward compatibility for DSL semantics. There is a DSL parameter that causes a DSL script to be interpreted in a way that is compatible with previous releases of VCC. The DSL semantics used prior to VCC 2.0 are still available if you specify the VCC1XMode mode parameter (of type Boolean) on the DSL view of the model and set the value to True. If you use this setting, make sure the pre-VCC 2.0 compatible services are used by the behavior for all port connections.

DSL primitives can sample inputs, perform delays, and post outputs. The delays can be constants, or they can be parameterized by the behavior model, the architecture model to which the behavior is mapped, or the state of a viewport in the behavior model.

The following table describes the semantics of some key DSL primitives.

`input(`*`port`*`)`	Defines the point in the reaction at which the read of the input port occurs.
`output(`*`port`*`)`	Defines the point in the reaction at which the posting to the output port occurs.
`run()`	This does not have any effect in VCC 2.0, but as in earlier versions of VCC, it is a required separator between the uses of the input primitives and the uses of the output primitives in the script.

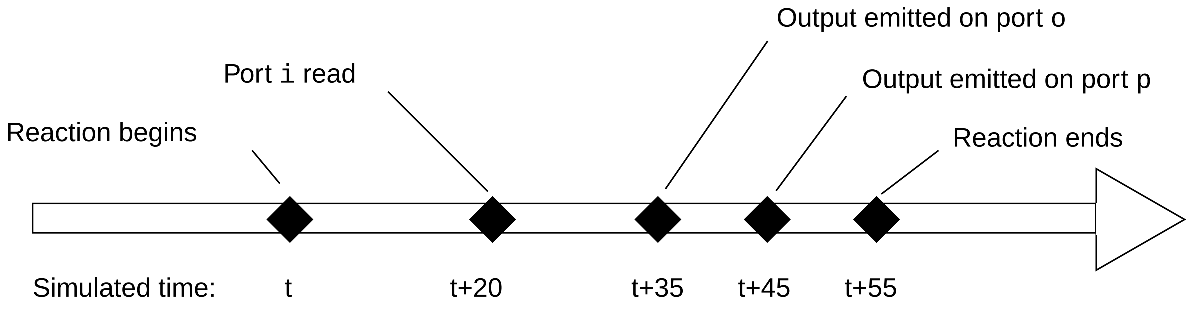
`delay(`*`expr`*`)`	Delays the simulation time from the current instant by the number of seconds specified by the expression.

Other commands are available in the DSL language. For more information, see the *VCC Modeling Guide*.

The following code segment shows a DSL script that models a behavior implemented on a target architecture.

```
#pragma "DELAY_MODEL_VERSION" "1.0"
delay_model( )
{
     /* Wait before reading input. */
     delay('20e-6');
     /* Now read the input. */
     input(inA);
     run( );
     /* Wait before posting outputs. */
     delay('10e-6');
     /* Post the output */
     output(outX);
     /* Wait between outputs, if needed. */
     delay('10e-6');
     output(outY);
     /* Wait for internal state to finish. */
     delay('10e-6');
}
```

This DSL example generates a temporal behavior similar to the timeline in the following figure.



Note: The output port statements must be in the same order as the execution of the functional model.

If a viewport is available on a functional model, an internal state variable can be referenced in a `delay` expression or in the conditional clause of an `if` statement. See VCC Help for details about using viewports.

The following DSL model is dependent on the input data and the attributes of the behavior model and architecture model.

```
     #pragma “DELAY_MODEL_VERSION” “1.0”
     attribute integer ntaps;
     attribute real cps;
     delay_model()
     {
          //First, read the input value.
          input(iblock);
          run();
          //Now, delay according to how long it takes to do an FIR.
          delay(‘((real) ((5 + iblock.npts) * (5 + ntaps))) / cps’);
          //For each output port, add ouput statement.
          output(oblock);
     }
```

In this script, there is an input port named *iblock* of composite type with an integer field called *npts*. The behavior model has an integer property called *ntaps* that is declared as an attribute in this model. The architecture model has a real performance property called *cps* that represents the clockspeed as cycles per second. The *cps* performance property is also declared as an attribute of this model. The casting operator *real* converts the integer expression to a real value because the *delay* expression must evaluate to a real expression.

Restrictions in DSL Semantics to Support Services

In previous VCC releases, you could use DSL to control both the order in which the outputs propagate from a block as well as the time delays between these output propagations. With the use of services in the current release, it can be the behavior, not the DSL script, that controls the order in which the outputs propagate from a block. You can use DSL to control the time delays between the output propagations. However, when a non-zero delay separates two output propagations (that is, a use of the ‘delay’ primitive with a non-zero parameter separates two uses of the ‘output’ primitive), the output port ordering in the DSL script must be consistent with the ordering from the execution of the behavior.

Furthermore, the behavior may execute, at most, one *Post* operation on a given output port in any reaction. The following example illustrates these points.

Example

Consider a behavior model with two output ports, o1 and o2, as follows:

```
void CPP_MODEL_IMPLEMENTATION::Run()
{
     ...
     o1.Post(...);
     o2.Post(...);
     ...
}
```

The corresponding DSL script is incorrect if it looks like this:

```
delay_model()
{
     ...
     output(o2);          // This is incorrect because
     delay(‘1.0’);        // the outputs were generated
     output(o1);          // in the reverse order.
     ...
}
```

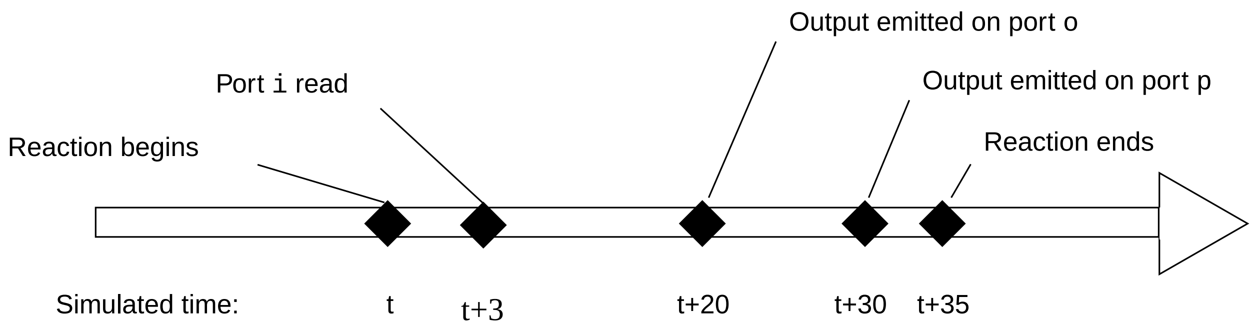
Annotated C Performance Model

The annotated C performance model is derived from a clearbox or whitebox model. The original model is annotated with inline delay calls and memory accesses to create the annotated C performance model. The VCC software estimator automatically inserts the annotations. Note that this technique is only applicable when you are mapping your behavior to a processor.

The VCC software estimator reads a restricted form of C code and estimates the cost of execution of each basic block. These estimates are parameterized by the cycle counts specified in the processor basis file. In addition, the software estimator inserts memory accesses for instructions and data fetching and estimates the size of the memory segments. The performance model is also impacted by the register allocation technique chosen for the model.

You can control many features of the software estimator. For more information, see the *VCC Modeling Guide*.

This performance model generates a temporal behavior similar to the following timeline:



Register Allocation

Without register allocation, VCC assumes that all variables are allocated to memory. The software estimator attributes a delay for loading variables from memory into registers prior to each use of the variable, and for storing variables in memory after each use. Using this method, the delay incurred by memory accesses might be pessimistic.

To improve this estimate, you can use register allocation for variables in a Whitebox C model in order to more closely estimate the actual processing cost of accessing variables. Register allocation assumes that as many registers as necessary are available and will be used. This implies that every use of a given variable does *not* result in a load and that every definition of the variable does *not* result in a store.

You can use register allocation for local scalars, parameters, or both. When you select one of these options, SUB and RET instructions in the processor basis file provide estimates for the cost of saving and restoring registers. Without register allocation, LD and ST instructions estimate the cost of memory load and store operations, which incur larger delays.

Note that register spills are not estimated. If the number of local scalar variables might exceed the number of general purpose registers, such as in data path code, this estimation might be optimistic. For most control path code, however, this estimation should be acceptable.

Annotated C++ Performance Model

The annotated C++ performance model is specified by inserting performance calls directly into your Blackbox C++ behavior model or service definition. This is the most general modeling style, which lets you model delays caused by the control structure of the model. You also have more freedom to model the performance at different levels of accuracy.

Performance calls in a behavior or service definition are made to routines specified in service declarations. The following four service declarations are useful in annotated C++ performance modeling:

- DelayDeclaration
- MemoryAccessDeclaration
- InstructionCostDeclaration
- ImplemenationSizeDeclaration

The details on the routines available in these services declarations can be found in the *VCC Library Reference*.

Before one of these service declarations can be used by the model, you must place it in the *Uses* tab, which is available when you are editing the properties of the model. (To access the *Uses* tab, open a .cpp file from Create, right click inside the text editor window, and choose *Properties*.)

In general, performance calls in the C++ model have no effect if the model is used in a functional simulation.

In a performance simulation, you must be able to bind the service declarations used for the performance calls to service definitions based on the mapping to architecture design. For example, the *InstructionCostDeclaration* service declaration is typically supported only by a processor, and if the behavior is mapped to a RTOS running on that processor, the implementation of *InstructionCostDeclaration* associated with that processor will be used. If, on the other hand, the behavior is mapped to an ASIC, it is unlikely that there will be an implementation of *InstructionCostDeclaration* available to satisfy binding.

Using DelayDeclaration

You use this service declaration to specify delays during a performance simulation. Delays can be expressed in terms of clock cycles or absolute time. See the *VCC Library Reference* for details of this service declaration.

The following example illustrates how you can use the *DelayDeclaration*.

Example

```
void CPP_MODEL_IMPLEMENTATION::Run()
{
     ...
     // Delays this behavior for two clock cycles. Assumes the
     // DelayDeclaration "Handle Name" on the "Uses" properties
     // tab for the model is "delayInterface".

     delayInterface.delayCycle(this, 2.0);
     ...
}
```

Using InstructionCostDeclaration

You use this service declaration for delay modeling based on a set of architecture-independent virtual instructions. For more details on this service declaration, see the *VCC Library Reference*, and for more details on these virtual instructions, see the information about annotated C delay models in the “Architecture Models” chapter of the *VCC Modeling Guide*.

One way in which you can use the *InstructionCostDeclaration* in a C++ model is by combining it with the *DelayDeclaration* to specify delay in terms of a set of virtual instructions. The actual delay will vary depending on the mapping behavior and the costs assigned to those instructions on the target processor.

The following example illustrates how you can use the *InstructionCostDeclaration*.

Example

```
void CPP_MODEL_IMPLEMENTATION::Run()
{
     ...
     // Delays this behavior for the execution of these 3 virtual
     // instructions. Assumes the "Handle Name" on the "Uses"
     // properties tab for InstructionCostDeclaration is
     // "instructionInterface". The call to getInstructionDelay
     // could be done once in the initialization
     // routine instead.
     double d = instructionInterface.getInstructionDelay (this,
          "ST + GOTO + OP.i");
     delayInterface.delayCycle(this, d);
     ...
     }
```

Using MemoryAccessDeclaration

This service declaration provides routines for modeling memory references. References may be made with respect to particular memory segments of the caller, or with respect to a particular value of the *vccAddress* type. Both reading and writing references are supported and the length of the reference is specified as an integer number of bytes. See the *VCC Library Reference* for details of this service declaration.

If you use memory segments, you should define the memory segments for the behavior or service definition model. Do this in the *Memory Segments* section of the *Parameters* tab when you are editing the properties of the model from a behavior diagram. Also, you should map the memory segments on the corresponding processor on the mapping diagram. Do this on the *Link Memory Segments* command on the processor.

Annotating Code/Data Memory References in Blackbox C++ Annotated Models

After you have specified the code/data segments as a property of your model, you can annotate your model with memory accesses to these code/data segments.

Typically, in the *init()* function, you get an index to the memory segment address for faster access

```
unsigned module, codeSeg, dataSeg;
void Init(){
     ...
     module = memDeclaration.registerModule(this);
     codeSeg = memDeclaration.registerModuleSegment(module, "code");
     dataSeg = memDeclaration.registerModuleSegment(module, "data");
}
```

Then, in your Run() (or other service functions), you can write:

```
void Run(){
     ...
     // Read 20 bytes from text segment offset 50
     memDeclaration.reference(module, codeSeg, 20, 50, memDeclaration.read)
     ...
}
```

Note: The offset specified for code/data memory reference is written to the segment of this service definition. This segment is then mapped to a specific architecture memory. Thus, the behavioral service definition can be annotated independent of the other software blocks and the linking/relocation information.

Using ImplementationSizeDeclaration

This service declaration provides routines for extracting architecture-specific information such as the sizes of various data types on the target platform. For more information see the *VCC Library Reference*.

The following example illustrates how you can use the *ImplementationSizeDeclaration*.

Example

```
void CPP_MODEL_IMPLEMENATATION::Init()
{
     ...
     // Obtain and store the size of a 'double' on the target
     // platform. Assumes the "Handle Name" on the "Uses"
     // properties tab for ImplementationSizeDeclaration is
     // "sizeofInterface".
     int sizeofDoubleInBytes=sizeofInterface.getSizeOfDouble();
     ...
}
```

Scheduling Shared Resources

When multiple behaviors are mapped to the same architecture model, these behaviors might share the same resources. For example, if multiple behaviors are mapped to an RTOS, they contend for execution time on the processor. In this example, the RTOS arbitrates between the behaviors by implementing a scheduling policy (such as round robin scheduling, priority-based scheduling, and so forth). In the VCC environment, a scheduling policy is modeled by an architecture service on the RTOS or ASIC architecture model.

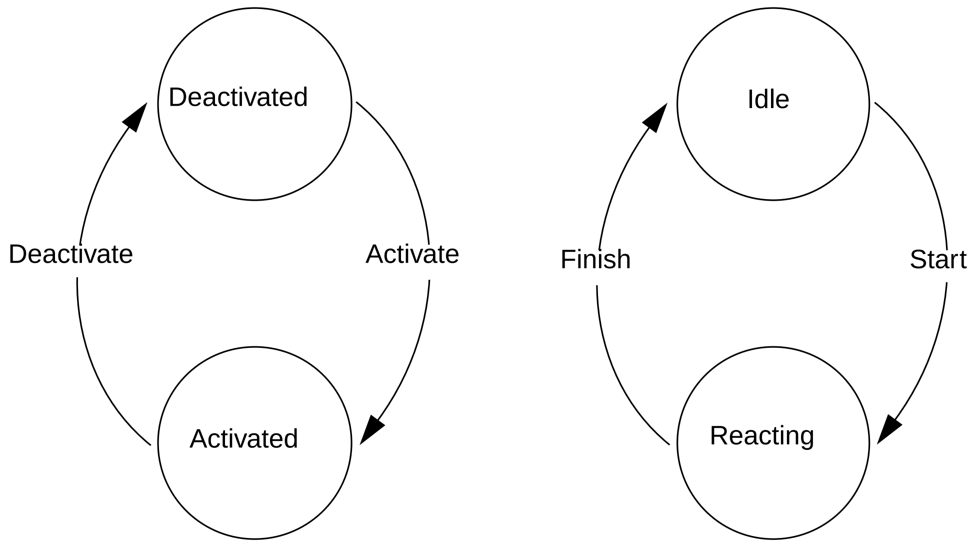
Performance Impact of Scheduling

Scheduling involves two fundamental concepts

- Activation–a state of readiness
- Reaction–the run state of a task from the time it starts to the time it finishes.

A task is activated whenever an event is received on any input port. An activated task waits for its assigned scheduler to respond with a start message. Once started, if the task is not suspended, it reacts until its performance model signals that the reaction is complete.

The activity of a task can be represented as a pair of concurrent finite state machines, or the equivalent product machine, as shown in the following figure.



The following transitions signal the conditions for state changes. They are issued by the simulator, scheduler, or a delay model associated with the task, as noted.

Transition	**Sender**	**Meaning**
Activate	Sent by the simulator as a result of other behaviors sending input to the behavior represented by the task	The task is ready to run.
Deactivate	Sent to the simulator as a result of receiving a start message from the assigned scheduler	The task no longer needs to be activated. (It is already running, or it has no input.)

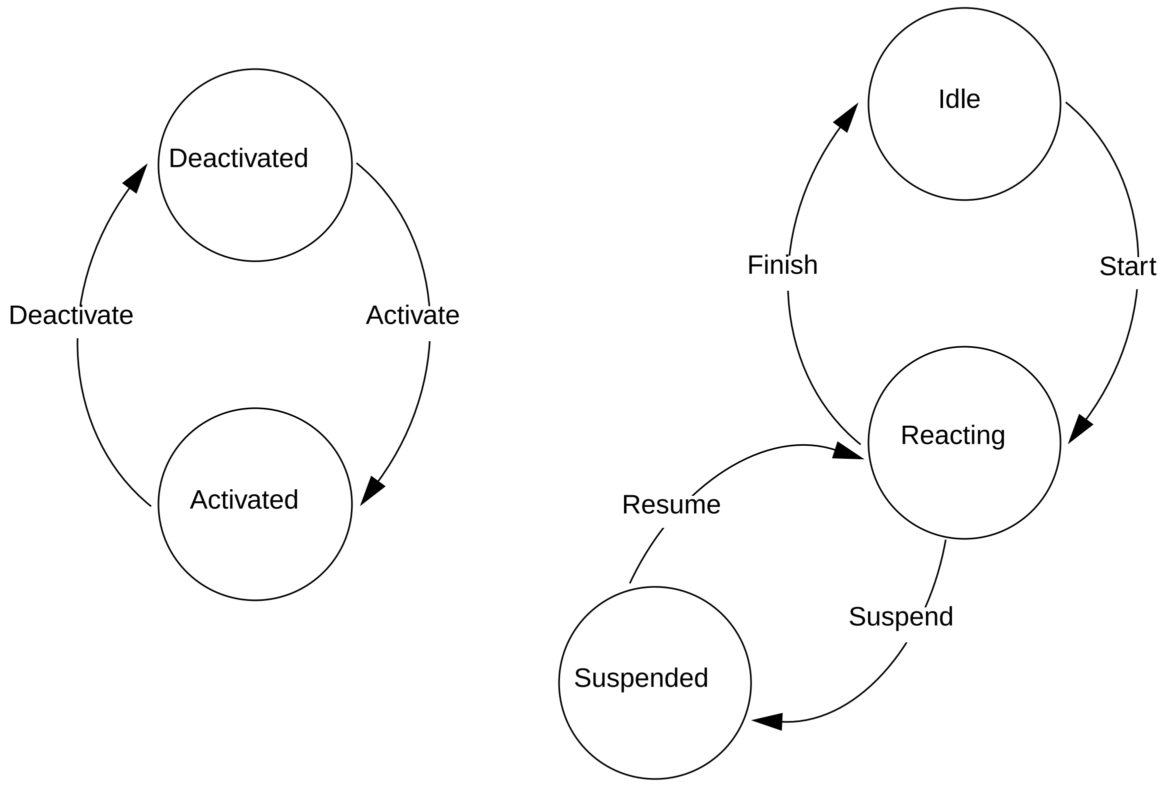
Transition	Sender	Meaning
Start	Sent by the scheduler to which the task is assigned	The reaction has started.
Finish	Sent by the delay model that is associated with the task	The reaction has ended.

To properly model software, both DSL and annotated C and C++ performance models can be suspended and resumed by an arbitrated scheduler. Suspending a reaction makes time stop for the performance model. Once the reaction is resumed, time begins again. This preemption capability can be fully integrated with the VCC scheduling facility without requiring changes to the functional model.

To support preemption, the scheduler provides an additional scheduler transition:

Transition	Sender	Meaning
Suspend	Sent by the scheduler to which the task is assigned	Stop processing the current task to activate a higher priority task.
Resume	Sent by the scheduler to which the task is assigned	Continue running the suspended task.

The following figure illustrates the addition of the suspended state in the task model.



The resume/suspend transition controls changes between the reacting and suspended states.

If preemption occurs after the input sampling phase, it delays the outputs by the duration of the preemption.

If preemption occurs during input sampling, computation might be affected. Input values might be overwritten before they are sampled, or new events might arrive that were not available originally. It is precisely these effects that need to be understood through simulation.

Scheduler Services

In the VCC environment, scheduling is modeled as an architecture service. This service is responsible for managing the state for each of the tasks mapped to it. The scheduler service determines what happens when two tasks request to run at the same time, and it defines the policy for preemption.

Scheduling policies can vary broadly. For example, a scheduler modeling a clock for digital hardware can disregard the notion of activation and start the reactions of all its tasks at a fixed frequency.

In contrast, a scheduler for a real-time operating system (RTOS) tracks activations of tasks as requests for service. In this case, the relation between activation and reaction is a complex function involving activated tasks and a set of priorities associated with them.

Most common scheduling policies are modeled as architecture services in the VCC_ArchitectureServices Library. The VCC simulator also provides support for modeling single-threaded and parallel-threaded schedulers.

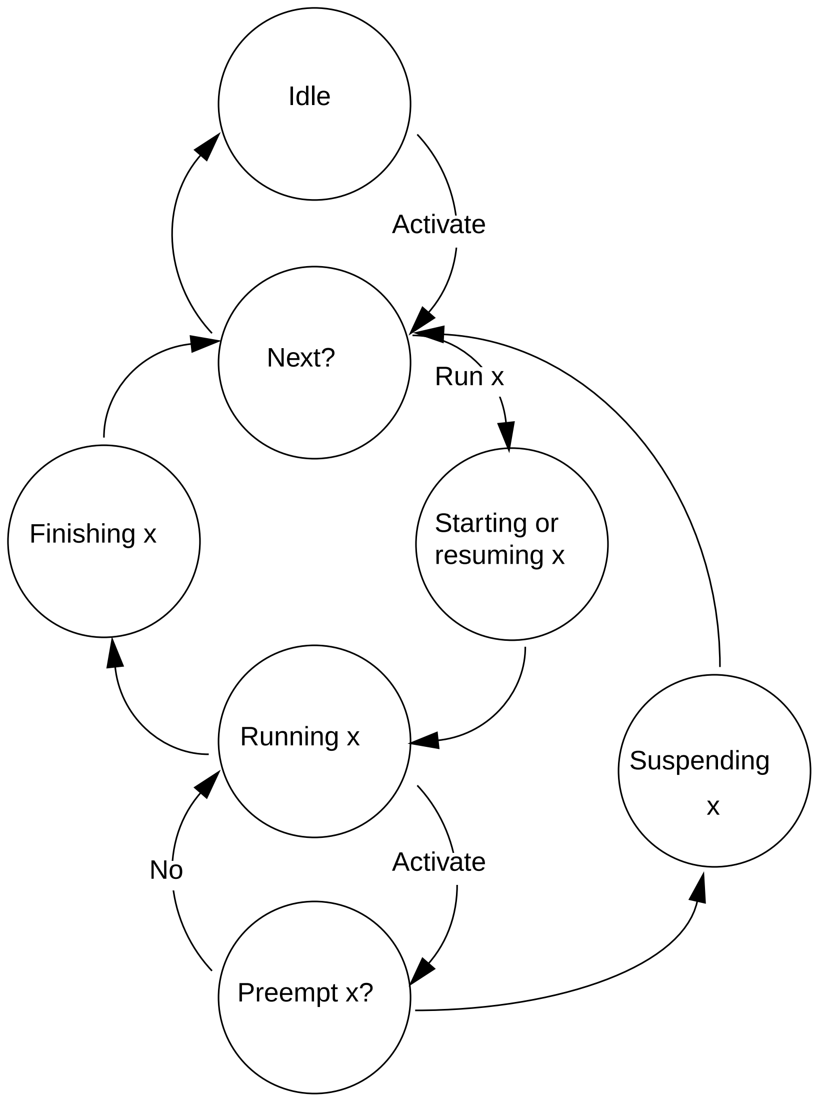
Single-Threaded Scheduling Model

The single-threaded scheduling model transitions a task from one state to another. Receiving an activation notice for one of its assigned tasks triggers one of the following changes.

- If the scheduler is idle, it determines the next task to run based on its scheduling policy.
- If a task is currently running and preemption is allowed, the scheduler determines if the new task preempts the current task. If so, the current task is suspended, and the scheduler determines the next task to run based on its scheduling policy.

  If the new task does not preempt the current task, the new task is maintained in the queue until the scheduler determines that it is next.

The following figure shows how the scheduler transitions through each state.



Modeling the overhead of scheduling uses delays associated with state transitions for the starting, resuming, finishing, and suspending states of the behavior.

RTOS Scheduler Services

Most application scheduling can use the following standard, single-threaded schedulers supplied in the VCC_ArchitectureService library.

- Cyclo-static scheduler
- FCFS scheduler

- Static priority scheduler
- Round robin scheduler

To configure these standard schedulers, specify the overheads for transitions in the scheduler states. Refer to VCC Help and the *VCC Library Reference* for details about setting these overheads when using a scheduler.

Cyclo-Static Scheduler

The cyclo-static scheduler associates an assigned task_order with each task and executes its assigned tasks on a fixed schedule. The scheduler needs at least one activation request before it can execute. When the scheduler receives an activation, it cycles through its schedule running each task in the order specified, then returns to the idle state. If a block is not runnable when its turn arrives, the scheduler sends an error message and continues with the next task.

Tasks are run from lowest to highest task order.

This type of scheduler is for behaviors that can be statically scheduled, such as a static dataflow.

First-Come-First-Served (FCFS) Scheduler

The first-come-first-served (FCFS) scheduler runs its tasks in a strict first-come-first-served activation order. When this scheduler receives an activation request, it runs its assigned tasks based on their activation timestamps.

Static Priority Scheduler

The static priority scheduler associates an assigned priority with each task. The priority is fixed for the duration of the simulation. The scheduler runs the task with the highest priority first. By default, the highest priority is defined as the largest `task_priority` value of its assigned tasks. This definition can be changed by changing the value of the `LargestPriorityIsHigher` performance property for this scheduler.

Activated tasks at equal priority are scheduled on a first-come-first-served basis.

This scheduler is parameterized to provide both preemptive or nonpreemptive modes.

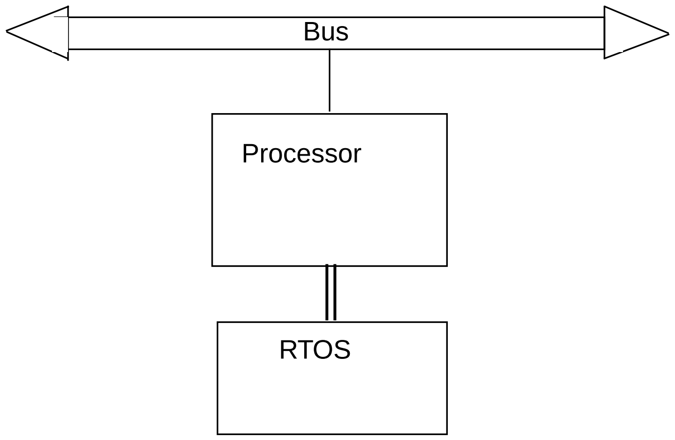
Round Robin Scheduler

The round robin scheduler associates a static priority with each task. The scheduler runs the task with the highest priority. By default, the highest priority is defined as the largest `task_priority` value of its assigned tasks. This definition can be changed by changing the value of the `LargestPriorityIsHigher` performance property for this scheduler.

Activated tasks at the same priority share the scheduler in a round-robin fashion using a specified time slice. The size of the time slice is configurable by the `Quantum` performance property.

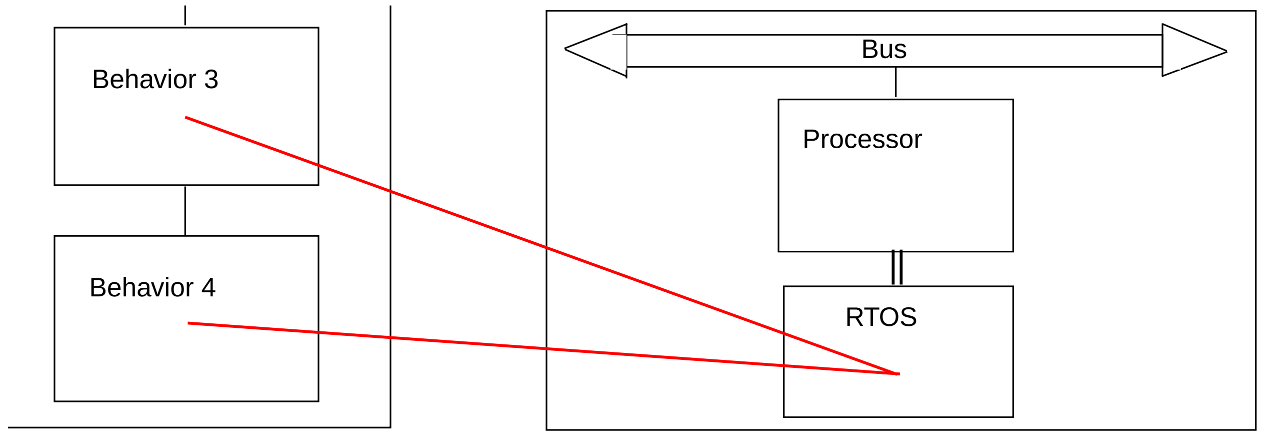
RTOS Models

If multiple behaviors are running on the processor model, it requires a scheduler. In the VCC environment, the scheduler for a processor is represented in the architecture diagram as a separate model, as shown in the following figure.



The processor model is attached directly to the bus model. The scheduler model is assigned to the processor model using the *Architecture > Scheduler Assignment* command. (The previous figure shows this connection with a double line.)

All behavior models accessing the processor are mapped to the scheduler model, as shown in the following figure.



The scheduler model can represent an RTOS or a single task. Both of these representations support a scheduler service to arbitrate the tasks. The RTOS typically provides additional services for standard C library calls, virtual timers, mutexes, and so forth.

Depending on the scheduler you select, you might need to specify one of the following link parameters (on the mapping link) to sequence tasks.

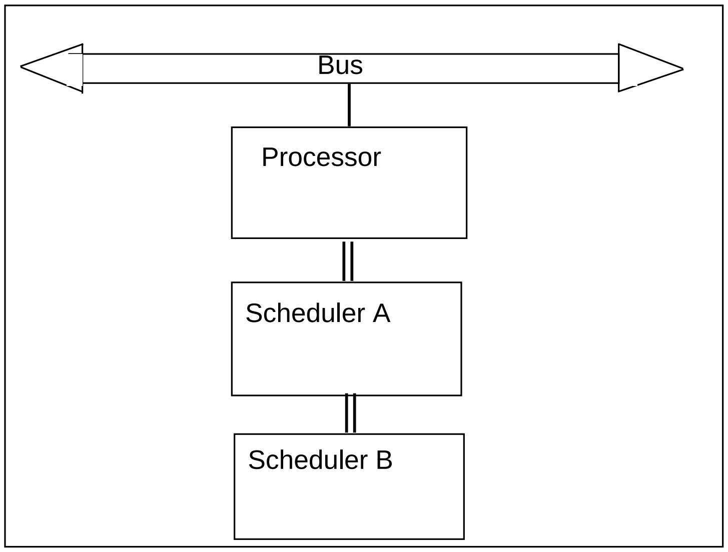
`task_priority`	Priority of the task associated with a specific mapping connection. Depending on how the scheduler model is defined, a higher value can mean either a high or low priority.
`task_order`	Order to execute a task on CycloStaticScheduler.

Note: On the mapping link, you are prompted for all link parameters of all performance views of the scheduler. You need only set those parameters that are used in the configuration you have selected.

Refer to Exporting Parameters on page 123 for information about exported parameters.

Nested Schedulers

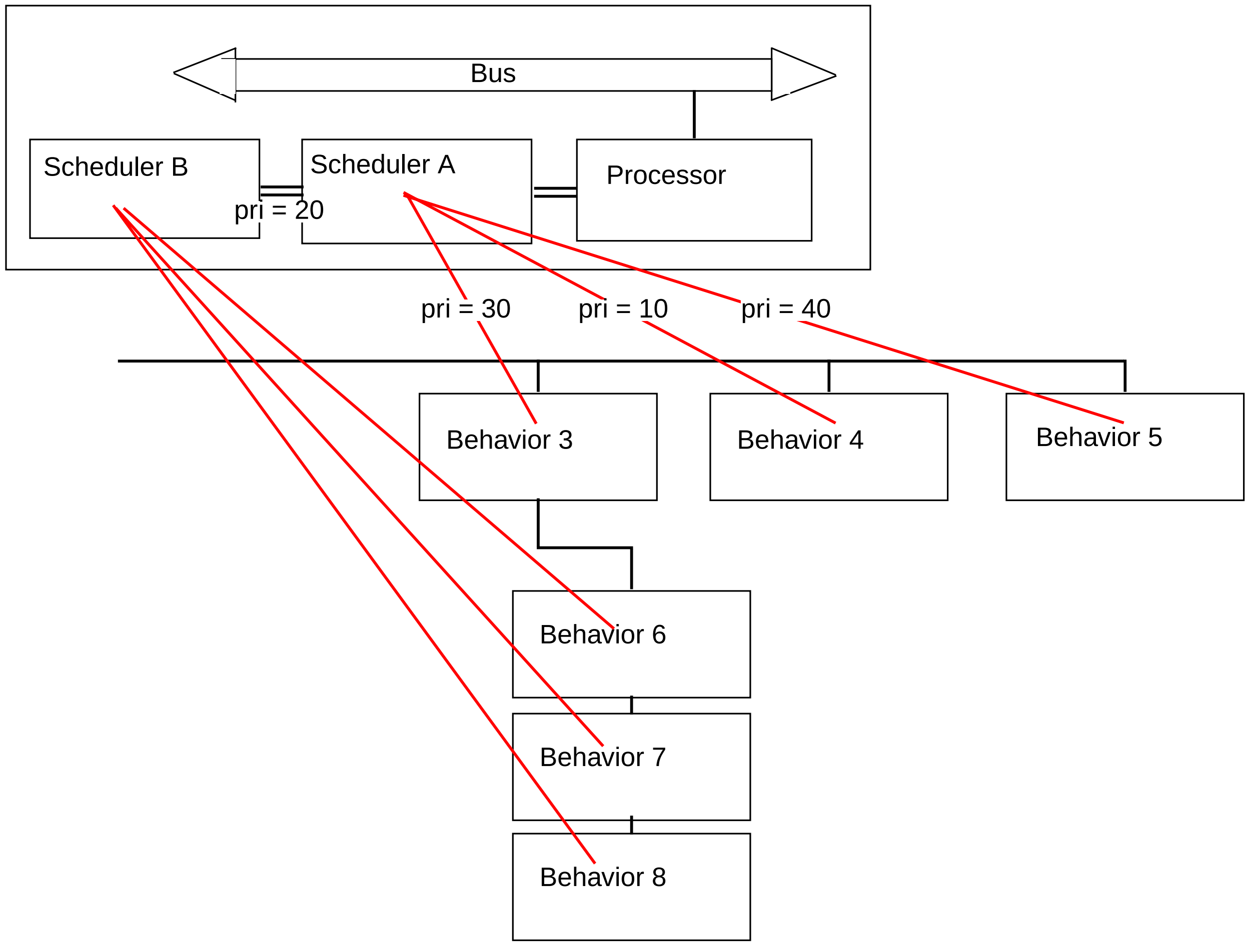
VCC also supports nested schedulers to model more complicated software architectures such as the grouping of multiple behaviors onto a single task that runs its behaviors sequentially. The single task is modeled as a secondary scheduler and is assigned to the primary scheduler using the *Architecture > Scheduler Assignment* command.



In this illustration, scheduler A is assigned directly to the processor. Scheduler B represents a second tier of scheduling and is assigned to scheduler A, not to the processor.

Scheduler B is considered a task assigned to scheduler A. If scheduler A requires link parameters, these parameters need to be set for the assignment between schedulers A and B. For example, if scheduler A has a prioritized scheduling policy, a `task_priority` link parameter needs to be set for scheduler B.

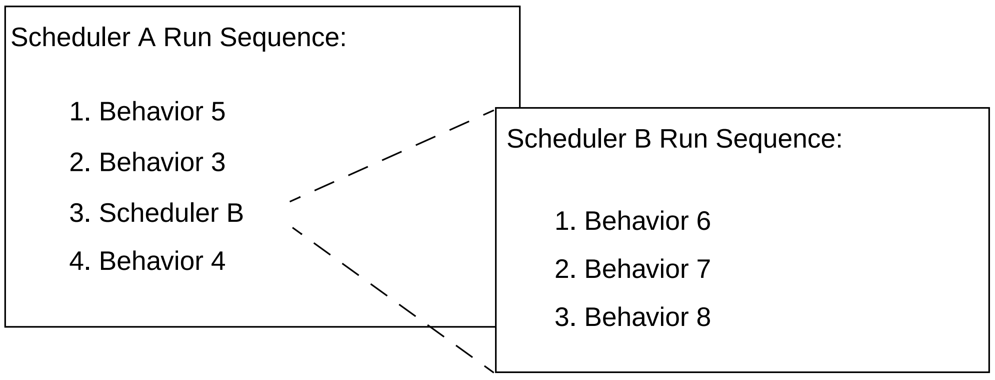
Behaviors can be mapped to both schedulers, as shown in the following figure.



Each mapping represents a task assigned to one of the schedulers. In this example, the mapped behavior models 3, 4, and 5, as well as scheduler B, are assigned to scheduler A. Behavior models 6, 7, and 8 are assigned to scheduler B.

How each behavior is mapped to a scheduler affects the run sequence of the behavior. Scheduler A handles all tasks assigned to it based on its scheduling policy. Scheduler B sequences its assigned tasks based on its own scheduling policy. When scheduler B is ready to run, all behaviors assigned to it are processed as a task flow on scheduler A, but in the sequence determined by the policy of scheduler B.

For example, a static priority scheduling policy can be used to sequence the tasks on scheduler A, while a first-come-first-served scheduling policy can be used to sequence the tasks on scheduler B. The following illustration provides a possible sequence of events for this example.



Behaviors 5 and 3 run first. Once they complete, scheduler B is ready to run. All of the behaviors assigned to scheduler B run in the order in which they were received by scheduler B. Once these three tasks complete, behavior 4 runs.

3 Communication Between Behaviors

Paths of communication exist in the architecture. For example, buses communicate with hardware, and interprocess communication, semaphores, and mailboxes handle communication between software tasks.

This chapter provides details about

Bus Arbitration

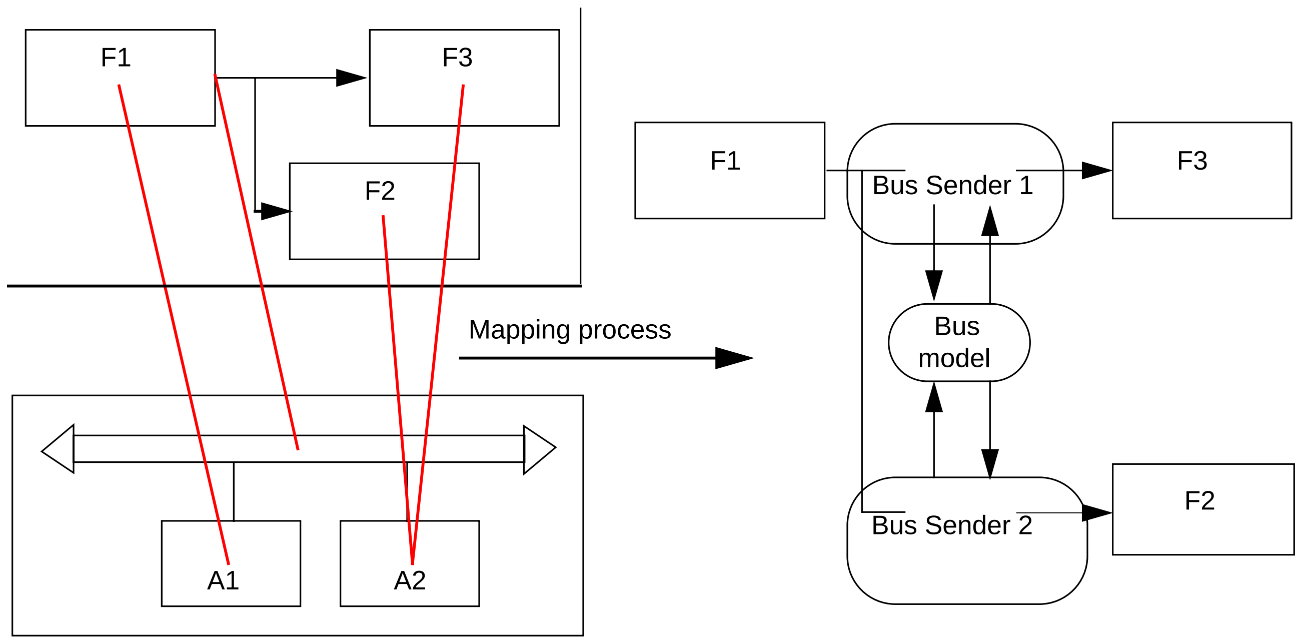
In a behavior diagram, communication paths are defined as wires between one source function and one or more destination functions. If the source and destination functions are to be partitioned onto separate pieces of hardware, the communication path usually occurs over a bus.

Typically, a communication path can be viewed as a dedicated or shared resource, such as a bus.

- When a communication path is dedicated, a single path exists between a source and destination, and delays incurred by tokens traversing the path are caused by propagation and signaling.
- If the path is shared, there are multiple behavior wires mapped to one architecture communication path. In this case, contention for access to the shared resource and a need for arbitration create additional delays.

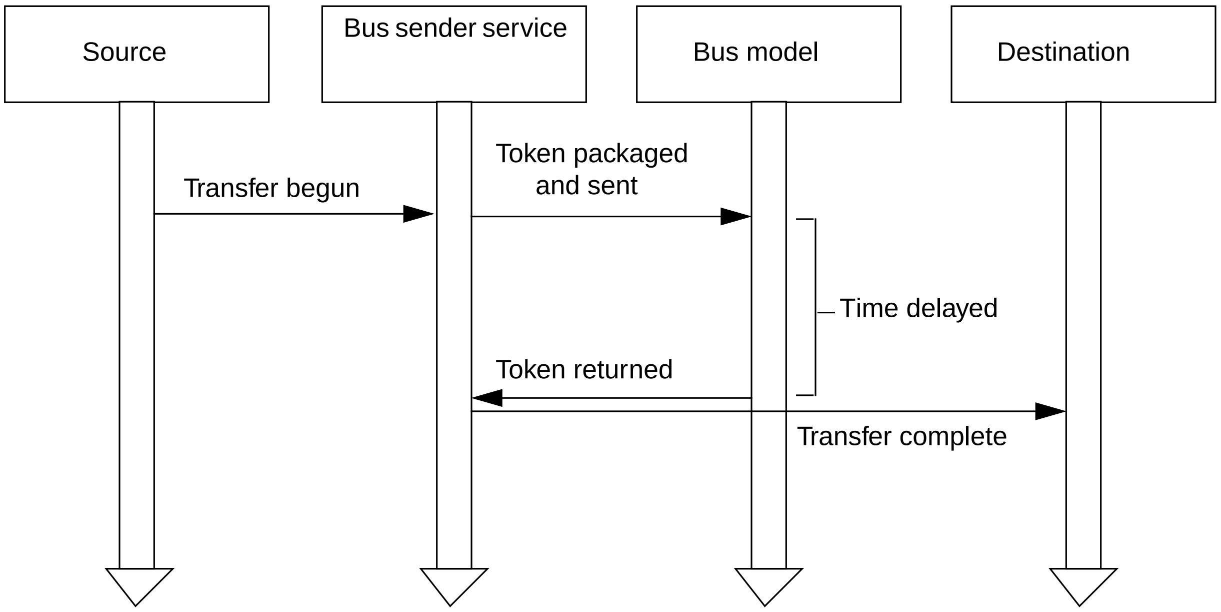
To model communication between different architecture models, some type of bus resource must be specified in the architecture diagram. VCC supports data, interrupt, and general bus resources.

The simple performance model for bus loading is produced by mapping each wire that shares a path of communication in the behavior model to a bus in the architecture. As a result of this mapping, the original behavior netlist is modified so that a bus sender service is placed between each source and destination, as shown in the following figure.



The bus sender service packages the token being sent and relays it to the bus model that represents the contention and delay of the shared bus. Each bus sender service can have attributes associated with it that convey information about the source and destination functions, such as a bus master’s ID and priority or a slave’s required wait cycles. The bus model obtains this information from the token relayed by the bus sender service.

The protocol for communication modeling can use fire-and-forget messaging (also known as blocking). When a source sends a token, the bus sender service intercepts the token, packages it, and submits it to the bus model. The bus model delays tokens appropriately based on the bus arbitration policy being used. Once the delay is complete, the bus model relays the token back to the bus sender service, which sends it to the destination, as shown in the following figure:



The bus sender and arbiter services are specified on the bus model. If necessary, you can write your own bus sender and arbiter services. The bus sender service must implement the *BusSenderDeclaration*, and the arbiter service must implement the *BusArbiterDeclaration*.

Arbitration Models

Arbitration models delay transfers in two ways. The queuing mechanism employed by the arbitration model itself delays bus access. Additional transfer delays are then calculated using various characteristics of the bus model. The following table defines characteristics that contribute to typical transfer delays.

Arbitration Cycles	Number of overhead cycles that occur between the completion of one data transfer and the start of the next Specified as a bus performance property value. (In the VCC buses library, this property is named `ArbCycles`.)

Data Transfer	Number of cycles required to transfer the data from the source to the destination with no contention The data transfer rate depends on the width of the data bus, the bus clock speed, and the size of the transaction. The width of the data bus and the bus clock speed are specified in bus performance property values. The size of a transaction is determined by the data type definition of the transaction.
Wait Cycles	Number of additional cycles required to compensate for slaves operating at lower clock frequencies than the masters that are accessing them By default, one cycle is required to write to or read from a slave. Additional cycles are specified in a bus performance property value.

The following arbitration models are provided in the VCC_ArchitectureServices library. For detailed descriptions of these arbitration models, refer to the *VCC Library Reference*. VCC also supplies parameterized buses for each of these arbitration models. You can create your own bus model, which references a VCC arbitration model, or you can create a customized arbitration model. Third-party bus models might use other arbitration models. Refer to your library documentation for details about third-party arbitration models and bus models.

FIFO Arbitration

When a transaction is received, the FIFO arbitration model places it in a first-in-first-out queue. No preemption or priority placement is allowed. The time it takes the transaction to reach the head of the queue accounts for the resource contention delay.

The cycles required for the transfer delay are calculated using the following equation:

```
[ArbCycles + WaitCycles * (bit_size + BusWidth - 1) / BusWidth]
    / BusClock
```

When the transaction reaches the head of the queue, its byte size is determined and the transfer delay is calculated. The bus model then waits the number of cycles calculated before returning the token to the bus sender service.

Preemptive FIFO Arbitration

The preemptive FIFO arbitration model places transactions in a first-in-first-out queue as they are received. A priority is specified when the communication is mapped. The time it takes the transaction to reach the head of the queue accounts for the resource contention delay.

When the transaction reaches the head of the queue, its byte size is determined and the transfer delay is calculated using the same equation as the FIFO arbitration model:

```
[ArbCycles + WaitCycles + (bit_size + BusWidth - 1) / BusWidth]
    / BusClock
```

With this model, however, a transaction with a higher priority than the currently running transaction preempts the running transaction. If a transaction is preempted, its transfer delay is recalculated using the remaining bytes left to transfer at the time the preemption occurred. The same equation is used to calculate the delay for resuming the transfer, but the remaining bytes to transfer is used for the `byte_size`.

Round Robin Arbitration

The round robin arbitration model places received transactions in a round robin queue. A separate queue exists for each priority. The queue containing the highest priority transactions is processed first. The time it takes the transaction to reach the head of the queue accounts for the resource contention delay.

The BusOwnerCycles bus performance property defines the maximum number of cycles that a master (usually a task) can control the bus. Large transfers represented by a single transaction are split up into one or more periods of ownership. Each of these periods of ownership is delayed based on the following equation:

```
{min [ArbCycles + BusOwnerCycles, ArbCycles + WaitCycles * (bit_size
    + BusWidth -1) / BusWidth]} / BusClock
```

When the remaining data to transfer requires less than a full period of ownership to transfer or if the transaction is small enough to complete within one period of ownership, the following equation is used:

```
[ArbCycles + WaitCycles + (RemainingBits + BusWidth - 1) / BusWidth]
    / BusClock
```

When enough time slices have occurred to satisfy the number of cycles needed to transfer all of the data, the token is returned to the bus sender service.

Simple Bus Arbitration

The simple arbitration model uses FIFO arbitration and models the delay based on the bus bandwidth (bps). No preemption or priority is allowed. The time it takes the transaction to reach the head of the queue accounts for the resource contention delay.

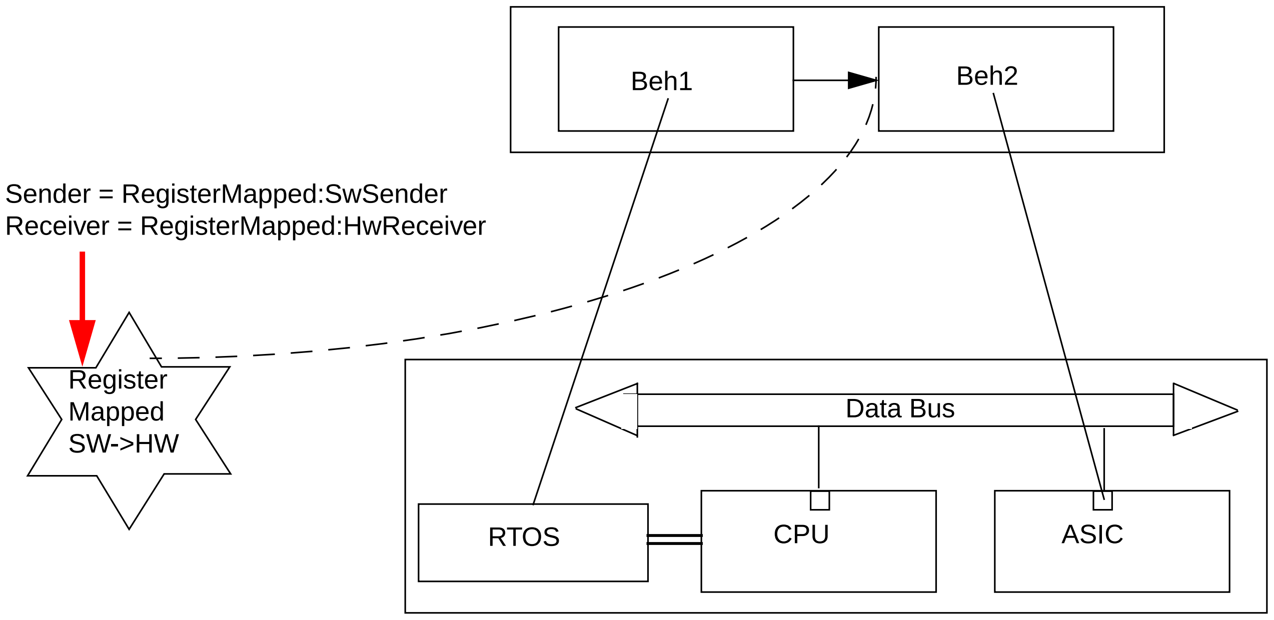
When the transaction is at the head of the queue, its byte size is determined, and its delay is calculated using the following equation:

```
(byte_size * 8) / BusBandwidth
```

This arbitration model can be used for shared or dedicated buses. When used for a dedicated bus, no queuing delay exists. The delay reflects only the transfer delay.

Communication Patterns

The previous section explained how to map a communication wire to a bus, which causes bus loading. For a more refined simulation, you can map a communication wire to a communication pattern. The following figure shows a communication wire mapped to a SW->HW RegisterMapped pattern.


Beh1
Beh2
Sender = RegisterMapped:SwSender
Receiver = RegisterMapped:HwReceiver
Register
Mapped
SW->HW
Data Bus
RTOS
CPU
ASIC

A pattern models specific protocols between the sender and receiver. A pattern also models interaction with other architecture components.

For example, to transfer a large data token from software to hardware, you might choose a shared memory pattern. This shared memory pattern models

- Writing the data to memory
- Sending the event to the ASIC
- The reading of the data from memory by the ASIC

This new model accounts for the delays of the three bus transactions as well as the memory delay for reading and writing data.

To transfer a smaller data token, you might choose a Register Mapped pattern in which the data is transferred directly to a register on the ASIC.

VCC provides a library of patterns for each fixed pattern supported by the VCC Links to Implementation Flow. These patterns are located in the VCC_Patterns library. Each cell has a symbol view, which is instantiated in the mapping diagram. The *proto* and *fixed_pattern* views are the performance views, which you need to bind in the mapping configuration. The *proto* view models the refined simulation of the protocol, while the *fixed_pattern* view models only the simple bus loading.

Note: When exporting your design, you need to bind the *protoImpl* view if you chose the *proto* performance view because additional parameters are needed for export. If you chose the *fixed_pattern* view, you do not need to specify additional parameters. The remaining views in the pattern cell are used for the Links To Implementation flow and can be ignored.

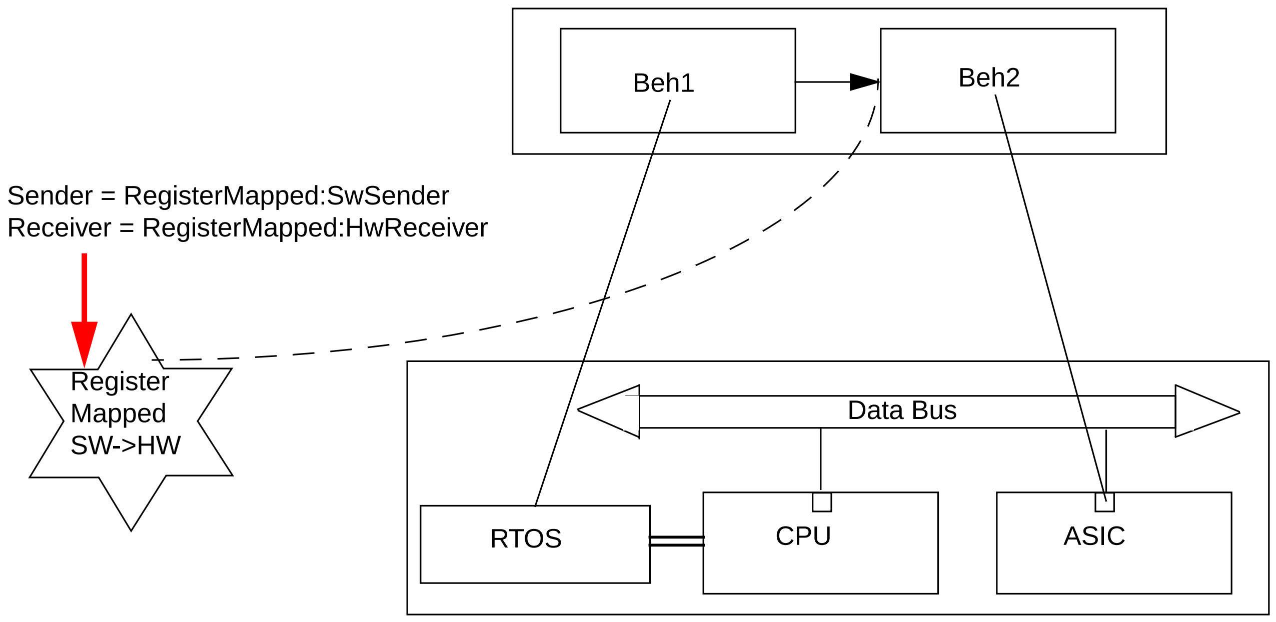
For more information on Links to Implementation, see the *VCC Links to Implementation Design Guide*.

Performance Model of a Pattern

Each pattern contains an arrangement of architecture services that model the path from sender to receiver. The architecture services are distributed on the architecture components in the architecture diagram. The performance view of a pattern binds the appropriate sender/receiver service definitions, which start and finish the particular communication protocol.

The sender architecture service for the pattern defines the *OutputPort* declaration for posting values to ports. The receiver architecture service defines the *InputPort* declaration for reading ports. VCC automatically inserts the sender service on the output port and the receiver service on the input port of the behavior communication wire.

In the following example, the SW->HW RegisterMapped pattern binds the RegisterMapped:SWsender and the RegisterMapped:HWreceiver services. VCC provides sender and receiver services for all the fixed patterns in the VCC_PatternServices library.

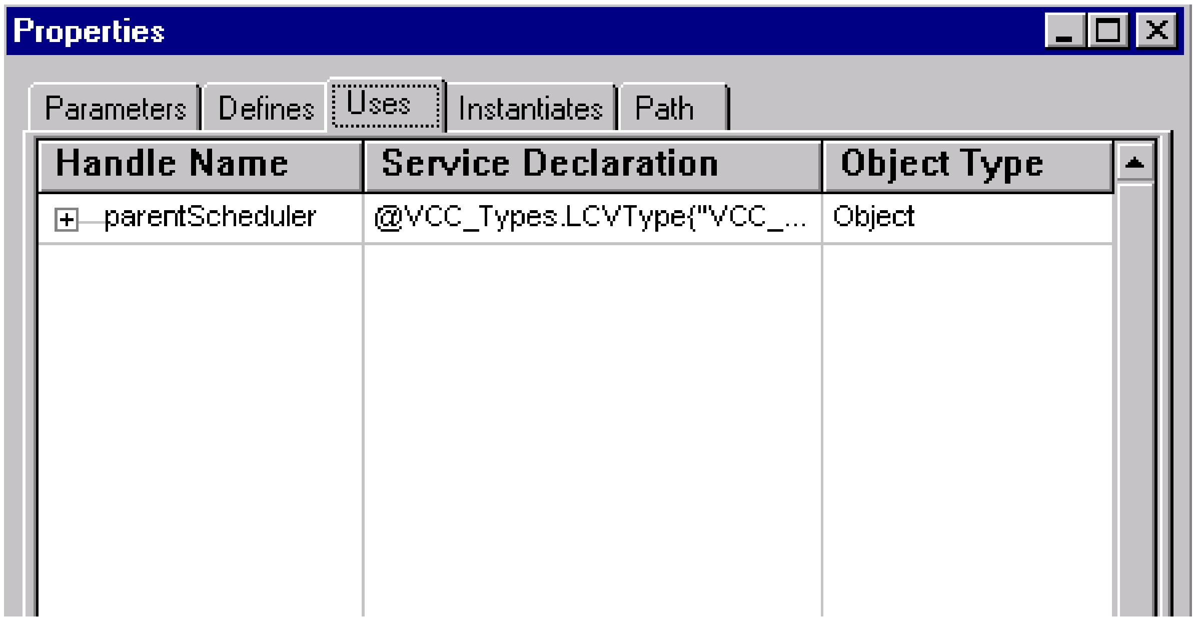

Beh1
Beh2
Sender = RegisterMapped:SwSender
Receiver = RegisterMapped:HwReceiver
Register
Mapped
SW->HW
Data Bus
RTOS
CPU
ASIC

There are architecture services between the read and write services that provide

- Communication from sender to receiver
- Effects of using shared memories, caches, DMAs, bus bridges, and so forth
- Effects of instruction fetches or data reads and writes from software tasks

The sender service can use services supported by the architecture component to which the source behavior is mapped. The receiver service can use services supported by the architecture component to which the destination behavior is mapped. In previous example, the sender service can use services supported by the RTOS and by association the CPU as well. The receiver service can use services supported by the ASIC.

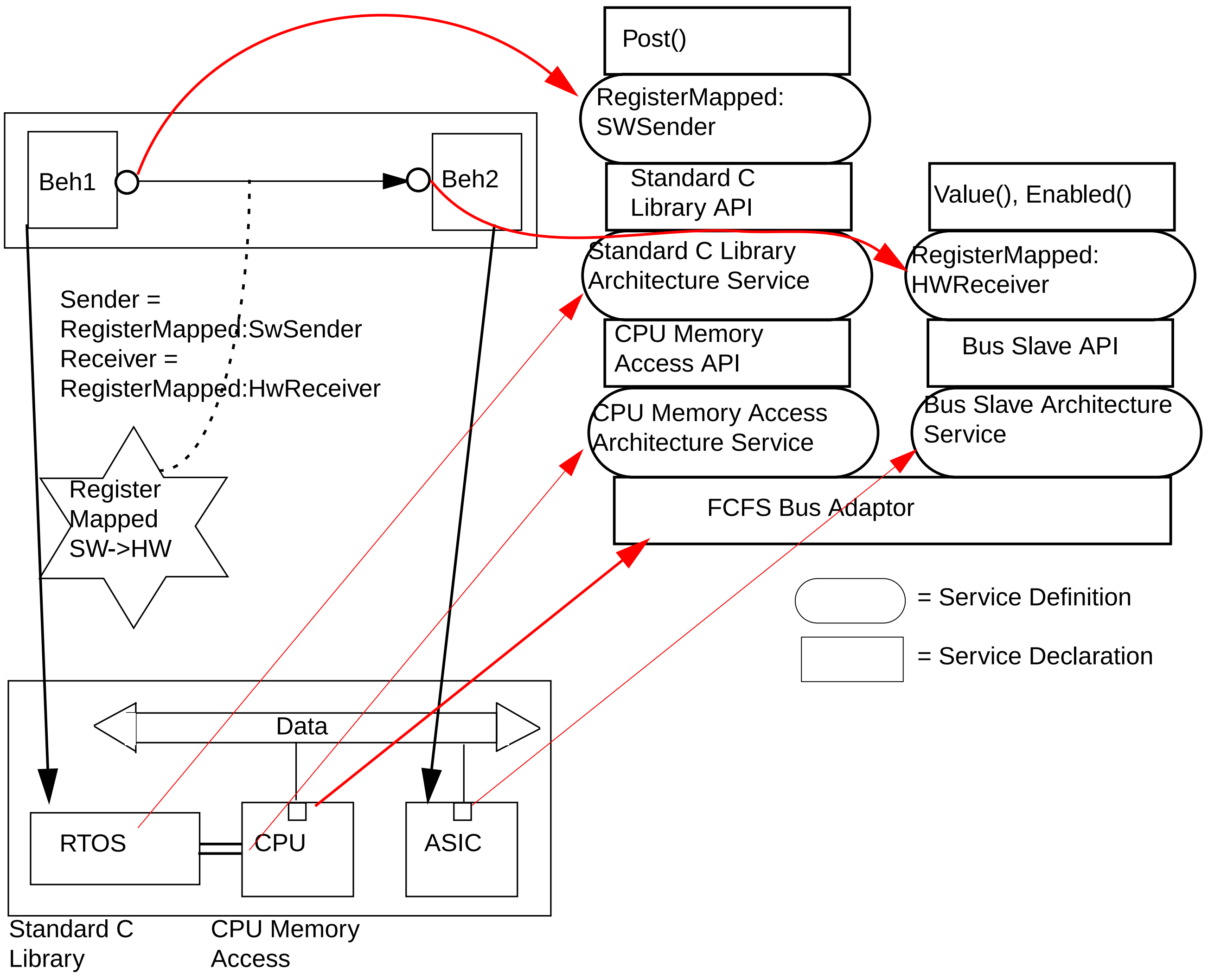
Distributing the services on each architecture component modularizes the performance impact to each component. If the architecture service interacts with other components, it declares the use of another service. Then during mapping, VCC binds the used service by following the architecture topology in the architecture diagram. To declare the use of another service in an architecture service, open the textual representation of the service in VCC, right click in the text window, and choose Properties from the pop-up menu. Click on the Uses tab and add the service. (See the VCC Help for more information.)



Handle Name	Service Declaration	Object Type
parentScheduler	@VCC_Types.LCVType{"VCC_...	Object

A service on an architecture component can use services on the same architecture component, on the port of the same architecture component, and on the architecture component to which it is connected. For example the CPU and ASIC can use services supported by the bus. The RTOS can use services supported by the CPU.

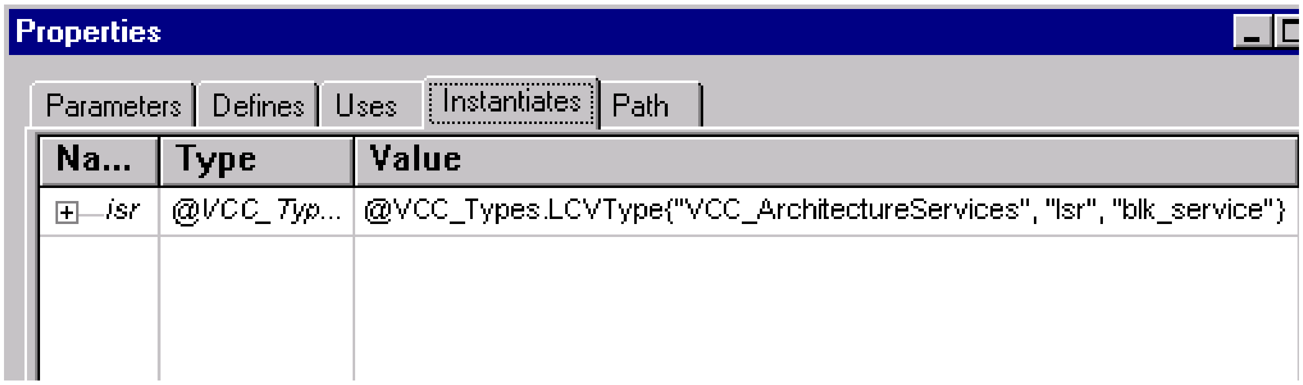
The following figure shows the bindings of services to complete a SW->HW Register Mapped pattern for the previous mapping diagram:



Additional Services

A pattern might require services other than those provided by the components of an architecture diagram. For example, a pattern might require a specific interrupt service routine for the communication to be complete (as in the case of an InterruptPattern). VCC supports *instantiates* clauses on services. An *instantiates* clause specifies another service that must be instantiated for this pattern to work properly. The *instantiates* clause is defined by a handle name, a reference to another service definition (by cellview name), and values for all parameters of that service. The instantiated service definition can use the same set of services that can be used by the original service.

For example, the *InterruptRegisterMapped:receiver* service instantiates the *ISR.blk_service* with the handle name *isr*. It specifies the parameter values for the *InterruptNumber* and *ISROverhead* as declared by the ISR service. Because the receiver can use the CPU (destination behavior is mapped to software) the instantiated ISR can also use services provided by the CPU. In this example, the ISR uses the *InterruptController* service of the CPU.



Na...	Type	Value
isr	@VCC_Typ...	@VCC_Types.LCVType{"VCC_ArchitectureServices", "Isr", "blk_service"}

Reusability of Patterns

To support reuse, most of the patterns are parameterized. The parameters can be specified on

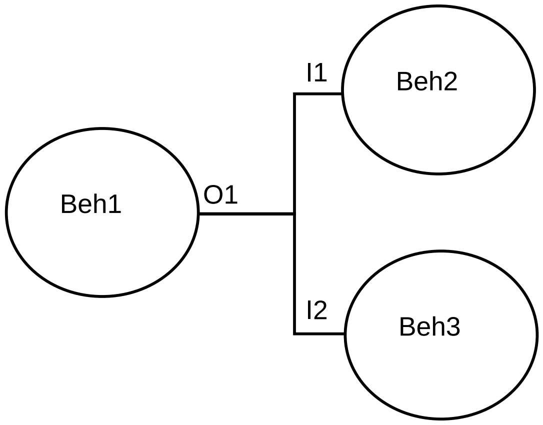
- The pattern instance
- The mapping link from the communication arc to the pattern (if a different value can be applied to each communication wire)

Some parameters reference other architecture components in the architecture diagram and are declared as participants of the pattern. You must specify the name of an architecture instance in the diagram as the value of this parameter. For example, the *SharedMemory* pattern sends messages to a memory component that is neither the sender nor the receiver architecture component. You need to specify the instance name of an architecture memory in the architecture diagram as the *MemoryParticipant* of the *SharedMemory* pattern.

Pattern Support for Fanouts

If a pattern instance is associated with an output port, then, by default, it covers all wires from that output port to the connected input ports. If the pattern instance is associated with an input port, then it covers only the single wire from the source output port to the input port.

Consider the following figure, which shows a behavior that is wired to two other behaviors with a fanout. The communication from Beh1 to Beh2 might be different than the communication from Beh1 to Beh3. If you map Beh1.O1, the same pattern applies. However, you can map Beh2.I1 and Beh3.I1 to different patterns.



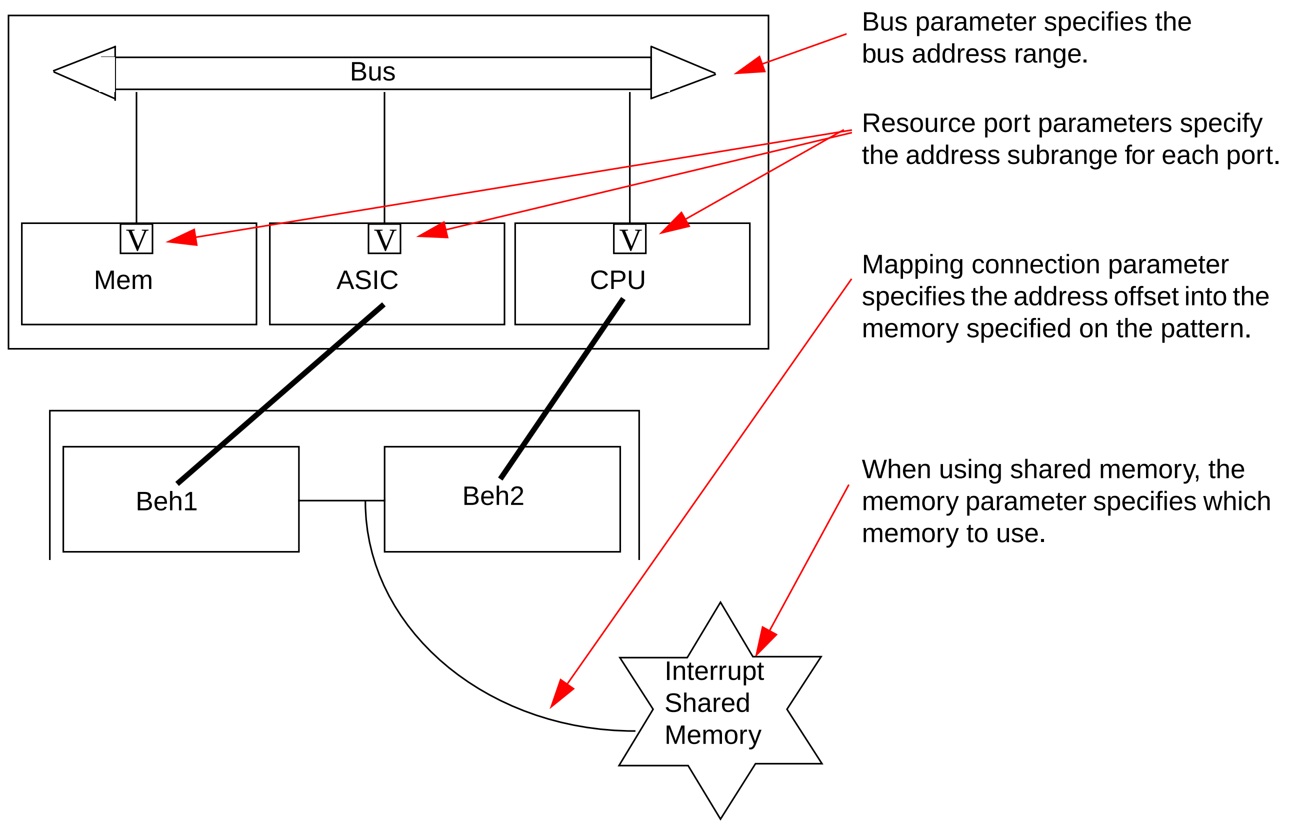
Addressing

Communication between behaviors requires that data or an event is sent to a specific memory register of an architecture instance. If the communication is over a bus, the transfer is directed by specifying a symbolic address (the instance name and an offset into that particular architecture instance). VCC converts the symbolic address into a port ID and offset by searching the architecture for the appropriate port on the specified instance.

For implementation, the symbolic address must be converted to a physical address. The physical address is determined by assigning each architecture instance a subrange into the bus address range.

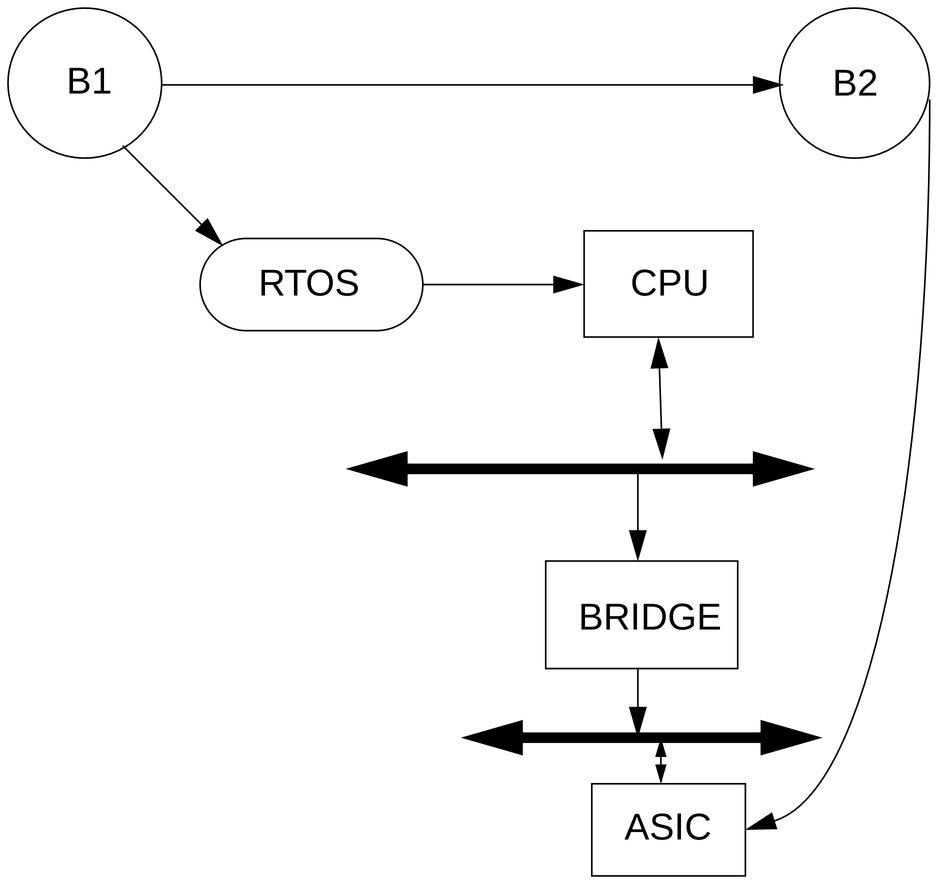
Therefore, the designer must set the bus address range on each bus, and subranges on each architecture instance port connected to the bus. Finally, the offset must be specified on the mapping link for each communication. The designer can specify the offset explicitly, or VCC can allocate the offset of a bus address using an algorithm that assigns non-conflicting addresses to each mapping connection.

The following figure identifies the locations of the parameters that determine bus addresses.


Bus
V
V
V
Mem
ASIC
CPU
Beh1
Beh2
Interrupt Shared Memory
Bus parameter specifies the bus address range.
Resource port parameters specify the address subrange for each port.
Mapping connection parameter specifies the address offset into the memory specified on the pattern.
When using shared memory, the memory parameter specifies which memory to use.

Bus Bridges

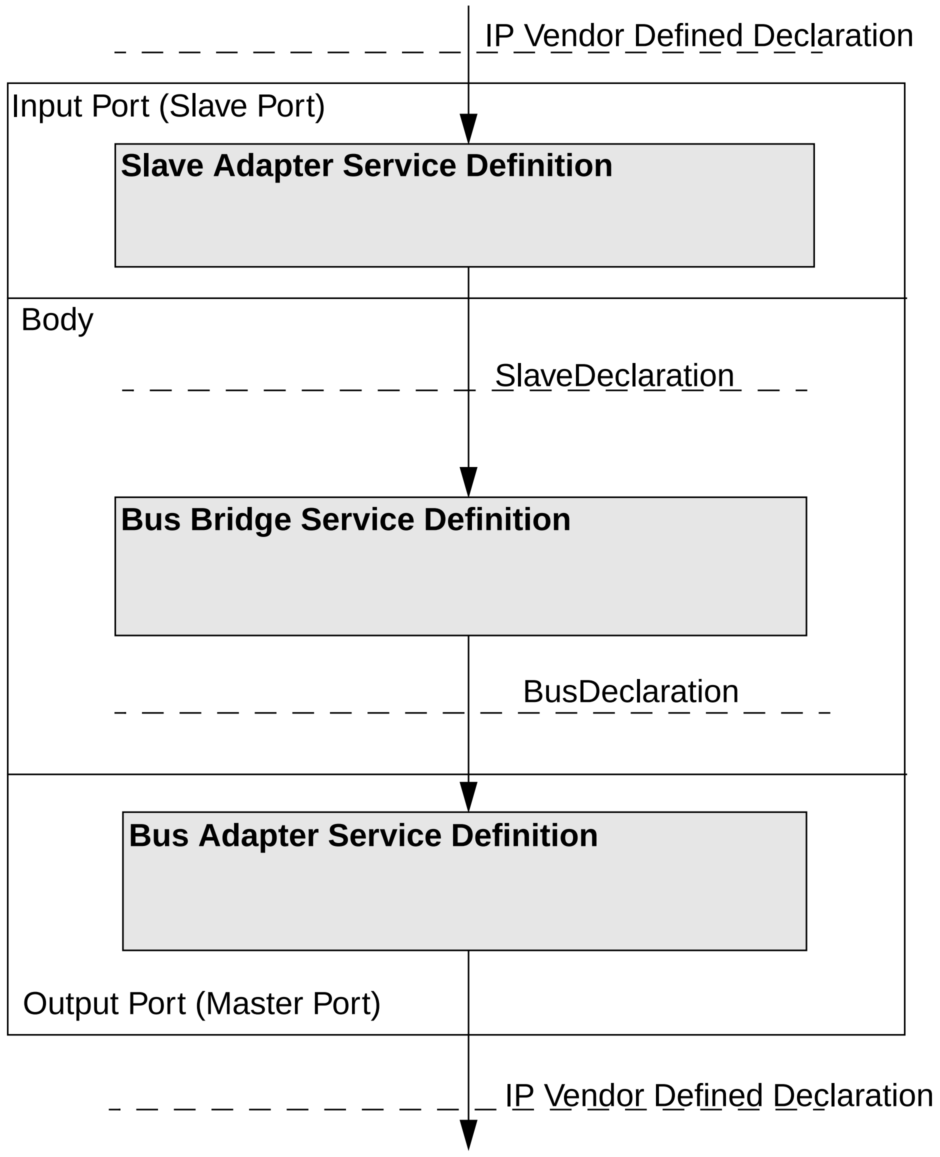
A bus bridge acts as a bridge between two buses. You can use a bus bridge to perform any address and service definition translations that you require. A bus bridge allows an element on one bus to write to and read from an element on another bus. A bus bridge can hold each bus for the duration of a bus transaction. The following figure illustrates how a bus bridge operates within the context of a simple architecture.



The source of the bus transaction (an annotated behavior, an output port service definition, or a memory reference service definition) knows the final destination address of a bus transaction in terms of a port id and an offset. The mapping editor derives this information based on the architecture topology and mapping, and places the information on the source instance as an attribute. Because the final address is known, for the purposes of simulation, no address translation is required in the bus bridge. Assuming the bus bridge has an input and output port, a slave adapter service definition is placed on the input port, a bus adapter service definition is placed on the output port, and a service definition is placed on the body of the bus bridge resource to propagate the data from the slave adapter to the bus adapter.

The service definition on the body of the bus bridge resource is called the bus bridge service definition. The bus bridge service definition implements the *SlaveDeclaration* and uses the *BusDeclaration*, and these declarations are implemented by the slave adapter service definition and the bus adapter definition, respectively. Typically, the bus bridge service definition is the only service definition that a bus bridge IP provider needs to write. Bus IP providers typically provide the bus adapter and slave adapter service definitions that go on the bus bridge ports.

The following figure shows the service definitions on the bus bridge resource and the declarations that they use for a simple two port bus bridge.


IP Vendor Defined Declaration
Input Port (Slave Port)
Slave Adapter Service Definition
Body
SlaveDeclaration
Bus Bridge Service Definition
BusDeclaration
Bus Adapter Service Definition
Output Port (Master Port)
IP Vendor Defined Declaration

You can construct the bus bridge with any number of service definitions, but the service definition that is used by the service definition external to the bus bridge must implement the *SlaveDeclaration*, and the service definition that uses a service definition external to the body must use the *BusDeclaration*. This ensures that your bus bridge model is interoperable with different buses.

The VCC software provides a bus bridge service definition in the *VCC_ArchitectureServices* library. It implements the *SlaveDeclaration* and uses the *BusDeclaration*. You should place this bus bridge service definition on the body of a unidirectional, 2-port bus bridge.

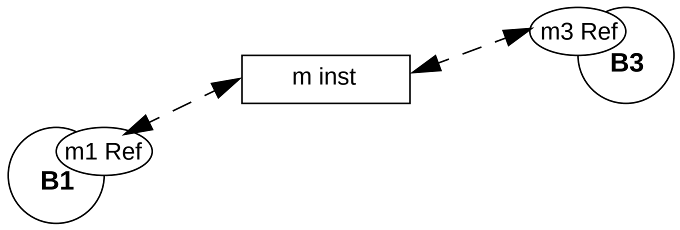
Because bus bridges can have multiple ports, the bus bridge must declare the legal direction of communication across the bridge. On each port, declare the output port on which the data is propagated using the *vccOutputPortName* parameter.

Behavior Memories

Behavior memories model the storage and retrieval of data in a behavior diagram. This communication takes place through strongly typed declarations that are defined in the behavior diagram.

The following example illustrates how behavior memories are used in the VCC environment.

Behavior Memories Example



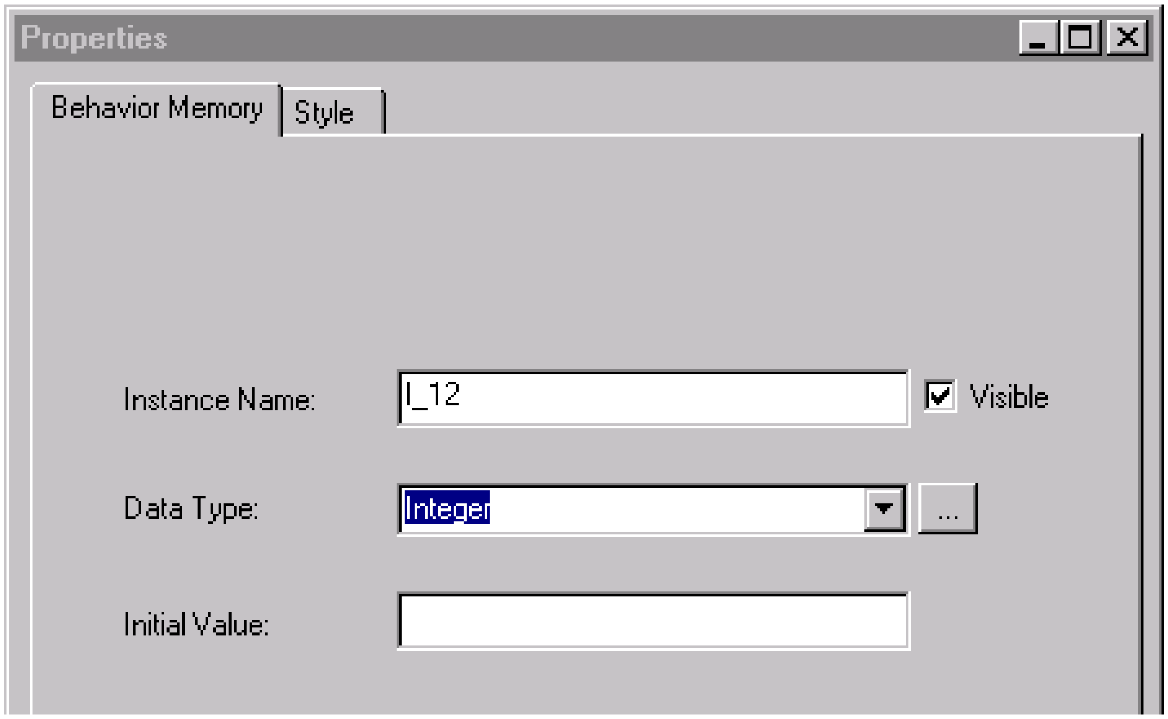
Using Behavior Memories

The process for using behavior memories is as follows:

- Place the memory instance in your behavior diagram.
- For each behavior that reads or writes to the behavior memory, declare a memory reference on the behavior model.
- Associate each memory reference to a memory instance.
- In the behavior model, write the code to access the memory.

Place the Memory Instance

When you instantiate a memory instance in a behavior diagram, you need to specify the name, type, and an initial value in the Properties dialog box. When you select and highlight the memory instance, you can attach probes, displays, and breakpoints.



Instance Name: I_12 [✓ Visible]
Data Type: Integer
Initial Value:

Declare the Memory Reference

In the symbol view of the behavior model, use the Memory Reference tab on the Properties dialog to declare the reference.

In the Behavior Memories Example on page 70, m1 is declared as a memory reference with the following parameters:

Name: m1
Datatype: Integer
Access mode: read

m1 is the *memRef* with *accessMode* (read) and the variable *arg* must have been declared of type Integer.

Associating a Memory Reference with a Memory Instance

Associate the memory reference on the behavior instance with the memory instance using the Memory Reference tab of the Properties dialog for the behavior.

In the Behavior Memories Example on page 70, the m1 reference on the B1 instance is associated with the *m* memory instance through the Memory Reference tab of the Properties dialog for B1.

Accessing Memory from within Your Model

Write your model code to access memory.

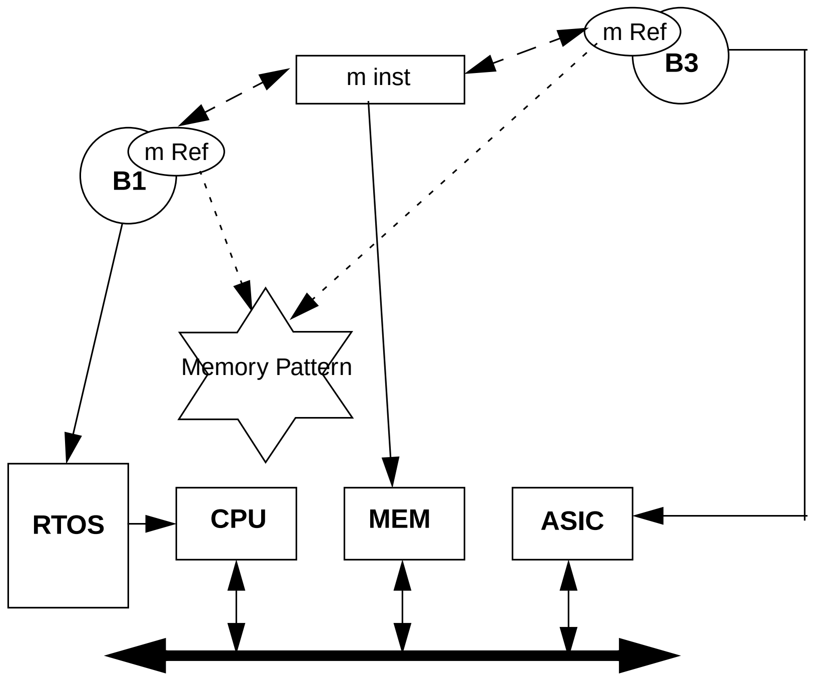
In the Behavior Memories Example, B1 is a Blackbox C++ behavior model. The code for the B1 model contains the following line:

```
m1.read(&arg);
```

This line assigns *arg* the integer value currently stored in the memory associated with reference m1.

Mapping the Memory Instance

The following figure illustrates behavior memories in a mapping diagram.



Each of the arrows in the diagram represents a function call. The dashed lines represent the associations between memory reference and memory instance. The dotted lines represent the communication pattern chosen for that memory reference.

Note: The dashed and dotted lines are not displayed in the VCC diagram, but there is highlighting to indicate these relationships.

Mapping the Memory Instance

The behavioral memory is mapped to an architecture memory. In the Performance tab for the link, there is a parameter for the offset. You can set the offset or you can leave it to be assigned by the address allocator.

Mapping the Memory Reference

To model the communication between the memory reference and the architecture memory, you map each *memRef* to a *Memory* pattern instance. The *Memory* pattern should have service handles for *read*, *write,* and *readWrite*. The appropriate service is associated based on the *accessMode* of the memory reference.

The *read*, *write*, and *readWrite* services use the *MemoryAccess* declaration supported by the architecture to which the behavior model is mapped. For example, the RTOS or ASIC. The

MemoryAccess declaration is located in the *VCC_PatternServices* library and is described in more detail in the *VCC Library Reference*.

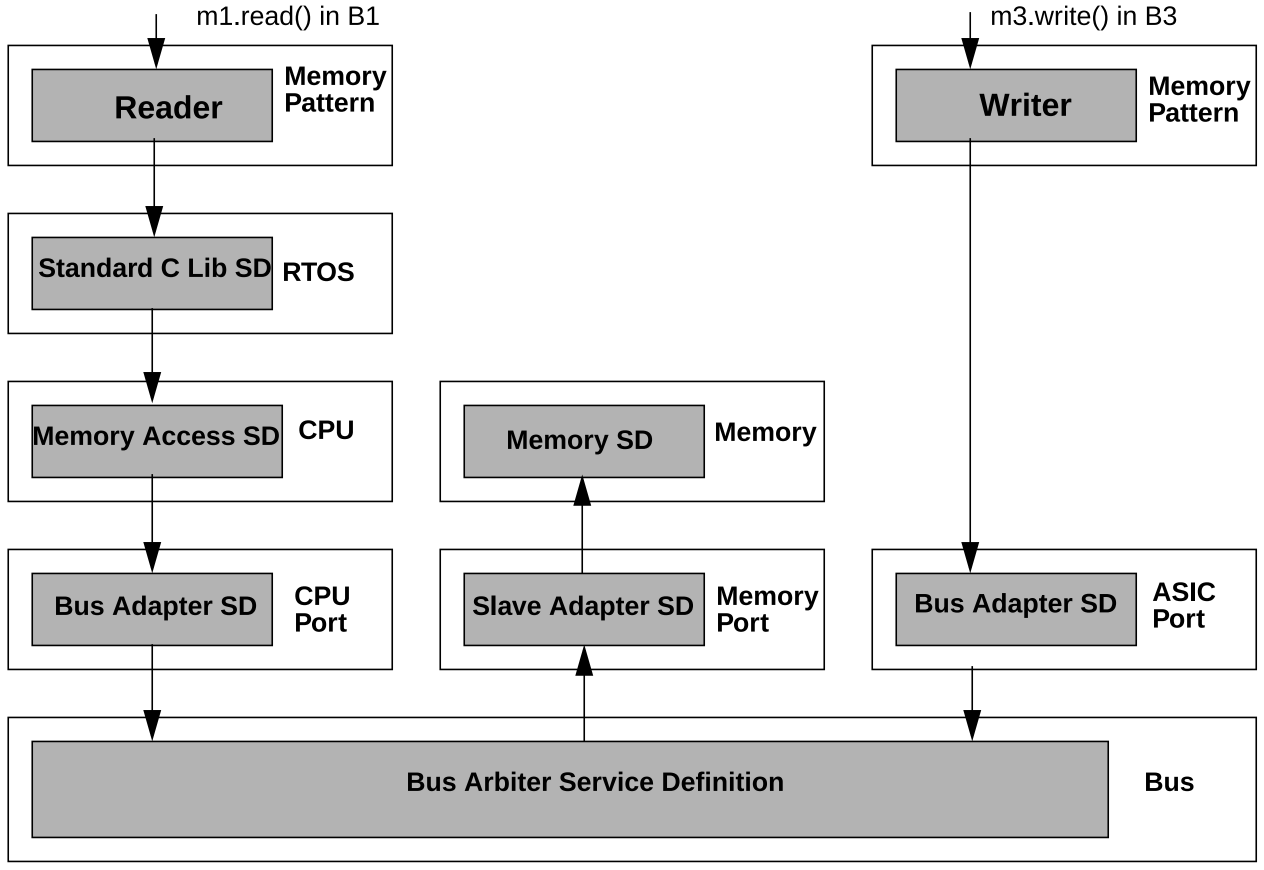
Additionally, these services can access parameters on the memory reference and the memory instance.

There are two pattern categories for *SW->Memory* and *HW->Memory*. You can set the default pattern for the *SW->Memory* pattern category as *SWDirectMemoryAccess* and the default pattern for the *HW->Memory* as *HWDirectMemoryAccess*. Both of these patterns are located in the *VCC_Patterns* library with the other provided VCC patterns.

Alternatively, you can use the *SWDMAMemoryAccess* pattern.

The pattern instances are very reusable, which means that you can *link* multiple memory references to the same pattern instance. For example, multiple memory references of the same behavior can *link* to the same pattern instance because the memory addresses are specified on the memory instance. Additionally, memory references from different behaviors can share the same pattern instance as long as both behaviors are mapped to software and communicate directly to the memory. If no DMA is involved, the number of pattern instances in a design might be two: one for HW and one for SW. You require additional pattern instances only if DMA is involved.

The following figure illustrates how service definitions with access to the behavioral memory are arranged based on the Behavior Memories Example on page 70.



The above diagram also shows the architecture resources with which each service definition is associated. Each of these service definitions is described in the *VCC Library Reference*.

DMA Modeling

A *DMA controller* (the DMA) lets you transfer data between two regions in the memory map without CPU intervention (other than the tasks the CPU needs to perform to start or finish a DMA request). The DMA usually contains several memory mapped registers that set up a context (a DMA channel) for the current transfer. For example, registers are normally used to indicate the source and destination addresses.

A transfer count register indicates how many bytes or words of storage will be transferred. A control register is used to set up some characteristics of the transfer, or to examine certain properties of the DMA, such as, priority, interruptabililty, and status. DMA controllers usually support several DMA channels, but data is only transferred one channel at a time unless the DMA controller is connected to more than one bus.

A DMA can be configured in a number of different ways:

- Some DMAs are tightly coupled with the bus architecture with which they are connected.
- DMAs can be on-chip or off-chip devices.
- DMAs can vary the number of channels supported.
- Some DMAs have FIFOs for buffering data.
- Some DMA transfers can be interrupted.
- Some channels have priorities over others.

Implementing a DMA Device Using Service Definitions

The following example shows how to implement a simple single-channel DMA device. The device is controlled solely by writing to memory mapped registers on the device. That is, there are no specific pins for requesting or acknowledging bus requests or grants. In this example, the DMA closely models the DMA controller found on some popular commercial DSPs.

The DMA controller acts as both a bus slave and a bus master. The DMA also raises an interrupt when the DMA transfer has completed. The service definition is located in *VCC_ArchitectureServices.SimpleSingleChannelDMA*. The DMA is a slave device because other devices control it by writing into its memory mapped registers.

The DMA implements the *ServiceDeclarations.SlaveDeclaration*. To issue bus requests to the bus, the DMA uses the *ServiceDeclarations.BusDeclaration*. To raise an interrupt, the DMA uses the declaration *ServiceDeclaration.InterruptDeclaration*.

This DMA example does not have its own service declaration. Devices use the DMA to write into its memory mapped registers—they do not communicate to it through a dedicated service declaration.

Memory Mapped Registers

A DMA contains four memory mapped registers that control it. These registers are mapped at offsets of 0 through 3, from the base address of the DMA.

- Source and destination address registers: The source address is the address from which the DMA device reads. The destination address is the address to which the DMA writes. Both the source address and the destination address registers are of type *vccAddress* in the DMA service definition.

- Transfer count register: The transfer count register is the number of sequential words, not bytes, that the DMA transfers.
- Control register: Writing to the control register starts the DMA transfer. The control register is not sensitive to any particular values. Writing any value to it initiates the DMA transfer.

Other Properties of the DMA Service Definition

Cycle Stealing

Often a DMA transfer has priority over the CPU when a bus is shared. This “cycle stealing” capability is implemented by ensuring that the *Priority* parameter on the bus adapter that the DMA is using is higher than that of the CPU.

Concurrency

When a DMA request begins, the DMA transfer occurs concurrently with the task that initiated the transfer. That is, control returns to the “caller” immediately and simultaneously with the DMA performing the data transfer. If the caller must block until the transfer is complete, it should block on a semaphore that is released when the DMA’s interrupt service routine runs. The DMA’s interrupt service routine is in *VCC_ServiceDeclarations.DMADoneISR*.

Setting up a DMA Transfer

The DMA declaration is defined and documented in *VCC_ServiceDeclarations.SimpleSingleChannelDMA:blk_serviceDecl*. The following example is a portion of a design that sets up the DMA device for a transfer. This example uses memory segments and declares a data memory segment named “dma” so that memory references can be issued to the DMA without using specific addresses. This example is part of a software block that is mapped to the CPU, which uses the CPU’s memory access declaration, *VCC_ServiceDeclarations.MemoryAccessDeclaration*. This could be part of an RTOS service routine, such as the *memcpy* function. You could also import it into a sender or receiver service definition.

Example

```
// Header file of a service definition using the DMA
class CPP_IMPLEMENTATION:public CPP_MODEL_DECLARATION{
public:
     CPP_MODEL_IMPLEMENTATION(const ModuleProto &, InstanceInit &);
     void Init();
     void Run():
```

```
     // Declare a segment ID for the memory segment that maps to
     // the DMA’s memory mapped registers. Also, set up a memory
     // segment for data in the main memory.
     MemoryAccessDeclaration<typeObject>::segmentId dmaSeg;
     MemoryAccessDeclaration<typeObject>::segmentId dataSeg;

     // The next four addresses correspond to the addresses of the
     // DMA’s memory mapped registers.
     vccAddress dmaSrcReg_;
     vccAddress dmaDestReg_;
     vccAddress dmaCountReg_;
     vccAddress dmaControlReg_;

     // These correspond to the actual source and destination
     // addresses. They are contained in the DMA’s address
     // registers.
     vccAddress srcAddressOfDMATransfer_;
     vccAddress destAddressOfDMATransfer_;

     // When real data is passed on the bus, it must be passed as
     // a VCC type and an object of a VCC type. The proper VCC
     // typeDefinition needs to be set up, so that the appropriate
     // typeObject can be passed.
     typeDefinition *addrType_;
};
```

The following portion of the model communicates with the DMA.

```
// Implementation of the above header file.
void CPP_MODEL_IMPLEMENTATION::Init() {

     // Some initialization needed to use memory segments.
     module = memDeclaration.registerModule(this);
     dataSeg = memDeclaration.registerModuleSegment(module, “data”);
     dmaSeg = memDeclaration.registerModuleSegment(module, “dma”);

     // Set up the addresses of each register on the DMA. These
     // addresses are used to issue memory write requests onto the
     // DMA’s address space.
     dmaSrcReg_=memDeclaration.allocate(“dma”, this);
     dmaSrcReg_.setArchOffset(0);

     dmaDestReg_memDeclaration.allocate(“dma”, this);
     dmaDestReg_.setArchOffset(1);

     dmaCountReg_=memDeclarartion.allocate( “dma”, this);
     dmaCountReg_.setArchOffset(2);

     dmaControlReg_=memDeclaration.allocate(“dma”, this);
     dmaControlReg_.setArchOffset(3);

     // Set up the memory areas where you want the DMA to start
     // reading from. This example reads to and writes from the
     // main memory.
     srcAddressOfDMATransfer_=memDeclaration.allocate(“data”, this);
     srcAddressOfDMATransfer_.setArchOffset(10);

     destAddressOfDMATransfer_=memDeclaration.allocate(“data”,
          this);
     destAddressOfDMATransfer_.setArchOffset(20);
     // When you write transfer addresses into the DMA, you need
     // to pass them on the bus as objects of type VCC.
     addrType_=typeDefinition::loadStandardType(
          "@VCC_Types.PortAddress");
}

void CPP_MODEL_IMPLEMENTATION::Run() {
// Code for the rest of the model functionality ...
...
// Start driving the DMA. You need to convert the vccAddress
// to an object of type VCC_Type.PortAddress.
typeObjectComposite source_addr(addrType_);
     typeObjectInteger
          source_offset(srcAddressOfDMATransfer_.getArchOffset());
     typeObjectInteger
          source_port_id(srcAddressOfDMATransfer_.getPortInstId());

     // Prepare the address to send to the DMA, and send it.
     source_addr.setField("Offset", &source_offset);
     source_addr.setField("PortInstID", &source_port_id);

     memDeclaration.writeData(dmaSrcReg_, &source_addr, 4);

     // Do the same for the destination register.
     typeObjectComposite source_addr(addrType_);
     typeObjectInteger
          source_offset(destAddressOfDMATransfer_.getArchOffset());
     typeObjectInteger
          source_port_id(destAddressOfDMATransfer_.getPortInstId());

     dest_addr.setField("Offset", &dest_offset);
     dest_addr.setField("PortInstID, &dest_port_id);

     memDeclaration.writeData(dmaDestReg_, &dest_addr, 4);

     // Set up the DMA's transfer count register with the number
     // of words to transfer.
     typeObjectInteger cnt(10);
     memDeclaration.writeData(dmaCountReg_, &cnt, 4);

     // Finally, tell the DMA to start the transfer. This call
     // returns immediately and the DMA transfer starts
     // concurrently.
     memDeclaration.reference(dmaControlReg_, 4,
          memDeclaration.write);

     // Code for the rest of the model functionality ...

     }// CPP_MODEL_IMPLEMENTATION::Run
```

A DMA performs two operations when it does a transfer. During a DMA read transfer, the device is reading data from the source memory. During a DMA write, the device is transferring the data it just read into the destination memory. For simulation efficiency this device performs all the reads in one large transfer, and all the writes in a second large transfer. This is more efficient in simulation because fewer bus events are generated. However, this approach has the disadvantage that reads and writes are not properly interleaved, resulting in a loss of fidelity regarding what is happening on the bus. You can modify the DMA so that it performs separate read/write requests for each word transferred.

Note: Performing the reads and writes in large transfers is different than a burst mode or block transfer because the transfer can be interrupted. You can modify the DMA example to make use of the memory subsystem's burst capability (if it has one).

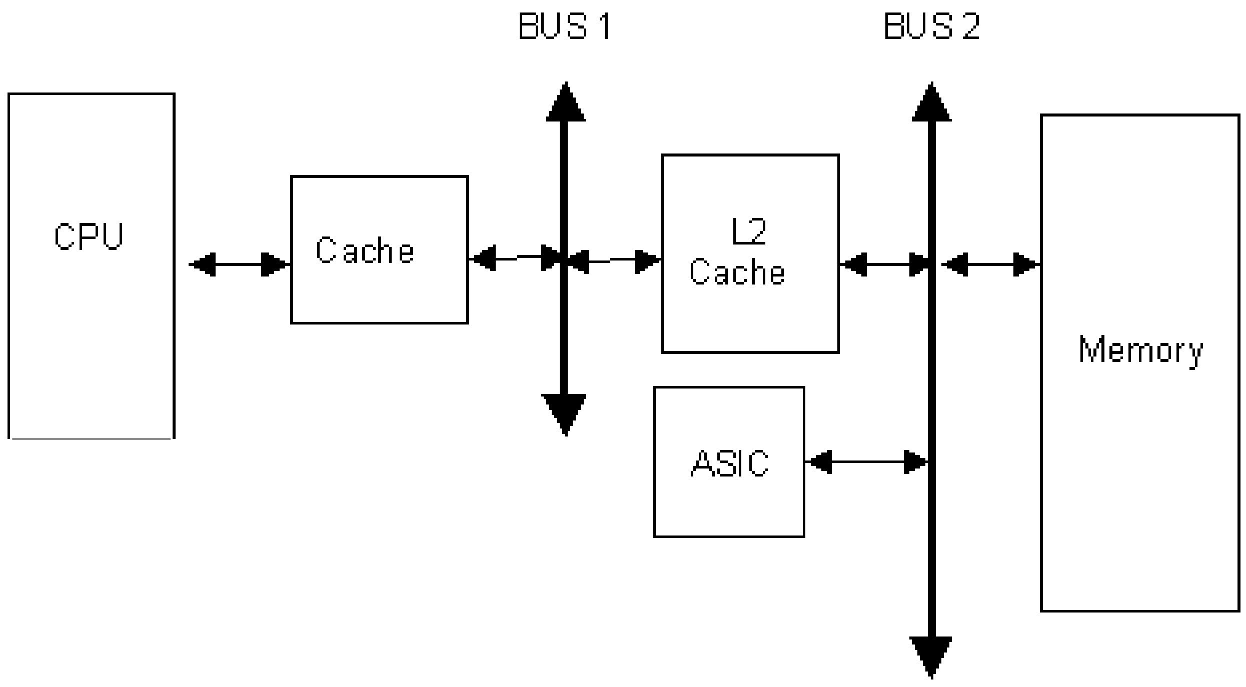
Parameters

The DMA service definition contains two parameters. The first parameter, *DMAInterruptNumber*, is an integer set to the interrupt number of the interrupt service routine that the interrupt controller is expecting to service the DMA's "done" interrupt.

The second parameter, *SizeOfSingleDMATransfer*, is the size, in bits, of each read and write transfer the DMA makes. You typically set this parameter to the natural word size of the system, which is the size of the data bus connected to the DMA (and the size of a word in memory).

Cache Modeling

You use a cache in the memory hierarchy between the CPU and main memory to balance the cost and performance of the system:



The CPU always reads from cache. Sometimes, depending on the write policy, the CPU writes indirectly to memory.

On a memory read, when the memory reference is not in cache, it is called a *cache miss*, otherwise it is called a *cache hit*. On a cache miss, the entire line of memory that contains the address is copied to cache first, then that reference is read from cache. If your program exhibits locality of reference, on your next request, you get a hit. A good example of this is code instructions, which exhibit locality of reference.

Usually, you use separate caches for data and instructions. Data cache modeling is more difficult, because it depends on the memory allocation/freeing algorithm and the dynamic nature of data allocation. Therefore, cache performance depends on the cache hit ratio—the higher it is, the more references are accessed from cache at a faster speed.

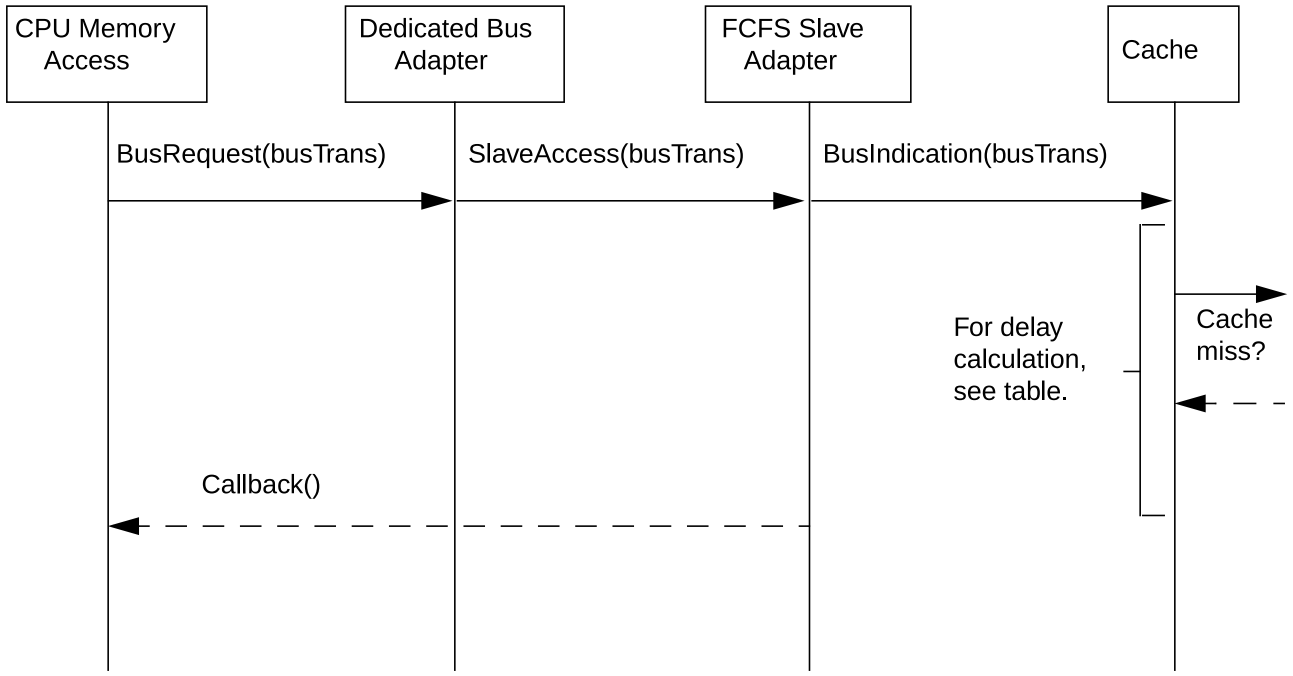
Two factors affect the cache hit ratio: the locality of reference of the program and the cache configuration. Because you cannot control the locality of reference, this section describes only the cache configuration (the parameters).

How VCC Models a Cache

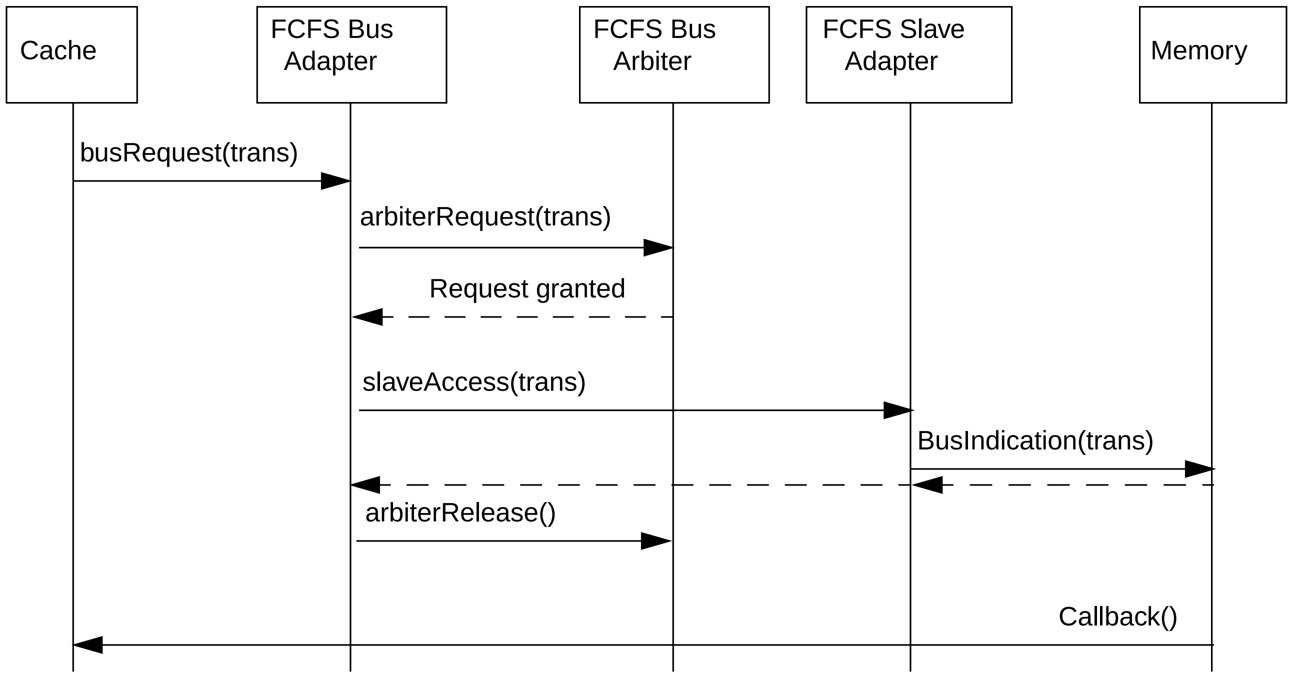
VCC models a cache by full simulation and records every memory reference. As the program executes, the CPU sequentially issues instruction fetches and memory reads and writes to the cache. The addresses are of the form `[MemoryPortId, Offset]`. The cache uses the offset to determine whether it is a cache miss or a cache hit.

For a cache hit, the cache returns control to the CPU, and simulation time is advanced by an amount equal to the cache latency. For a cache miss, the cache issues a request on the bus and waits for the memory to process the request.

When the memory receives a reply, the cache returns control to the CPU, and simulation time is advanced by the cache plus memory latencies. The following message sequence charts provide more detail about the delays encountered with a cache hit and cache miss.



If a cache miss is detected, a further delay is assessed, as shown in the following message sequence chart.



The following table shows the delays produced when the CPU issues a memory request. The delays depend on the write policy of the cache and whether the transaction is a read or a write and whether it causes a hit or a miss. The table assumes that BUS_1 connects the CPU to the cache, and that BUS_2 connects the cache to memory.

Read/ write	Hit/miss	Write policy	Transaction size		Delay (Excluding Bus Delay)
			Bus 1	Bus 2	
Read	Hit	n/a	Word	n/a	Cache read latency for one word
Read	Miss	n/a	Word	Lind	Memory read latency for one line, plus cache read latency for one word, plus memory write latency for one line if hit write policy is write-back, cache is full, and replaced line is dirty*
Write	Hit	Write back	Word	n/a	Cache write latency for one word

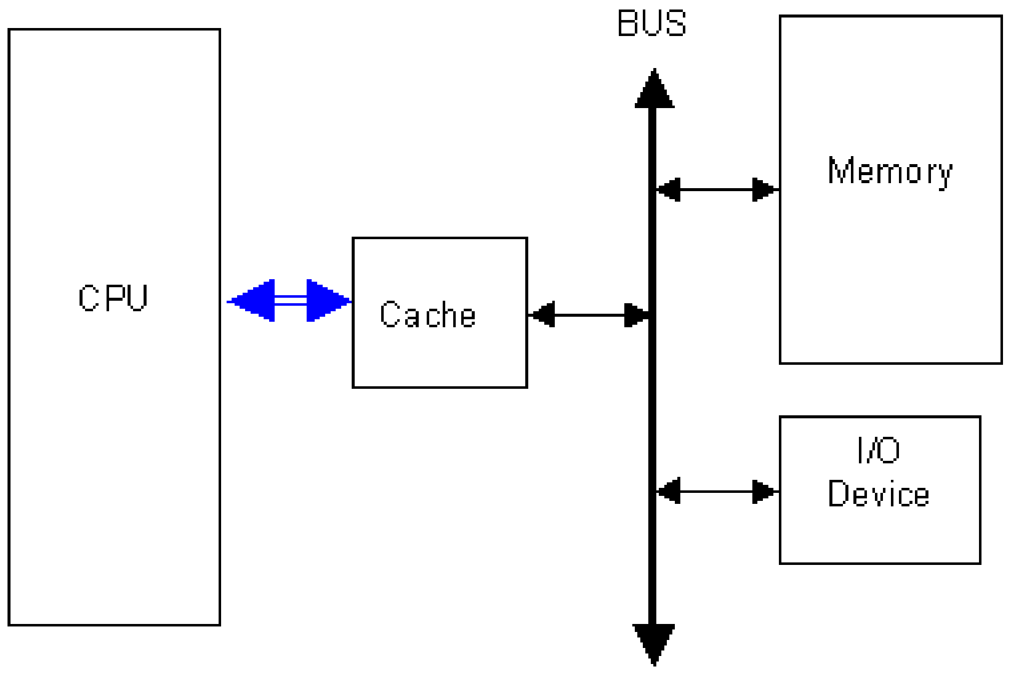
Read/ write	**Hit/miss**	**Write policy**	**Transaction size**		**Delay (Excluding Bus Delay)**
			Bus 1	**Bus 2**	
Write	Hit	Write through	Word	Word	Cache write latency for one word, plus memory latency for one word
Write	Miss	Write allocate	Word	Line	Memory read latency for one line, plus cache write latency for one word, plus memory write latency for one line if hit/write policy is write back, cache is full, and replaced line is dirty*
Write	Miss	Write around	Word	Word	Memory write latency for one word

* A “dirty” line is one that is the most up to date such that its copy in memory is invalid.

Note: The delays in the table depict the cache and memory delays only. They exclude the bus because the bus delays depend on other activities such as contention.

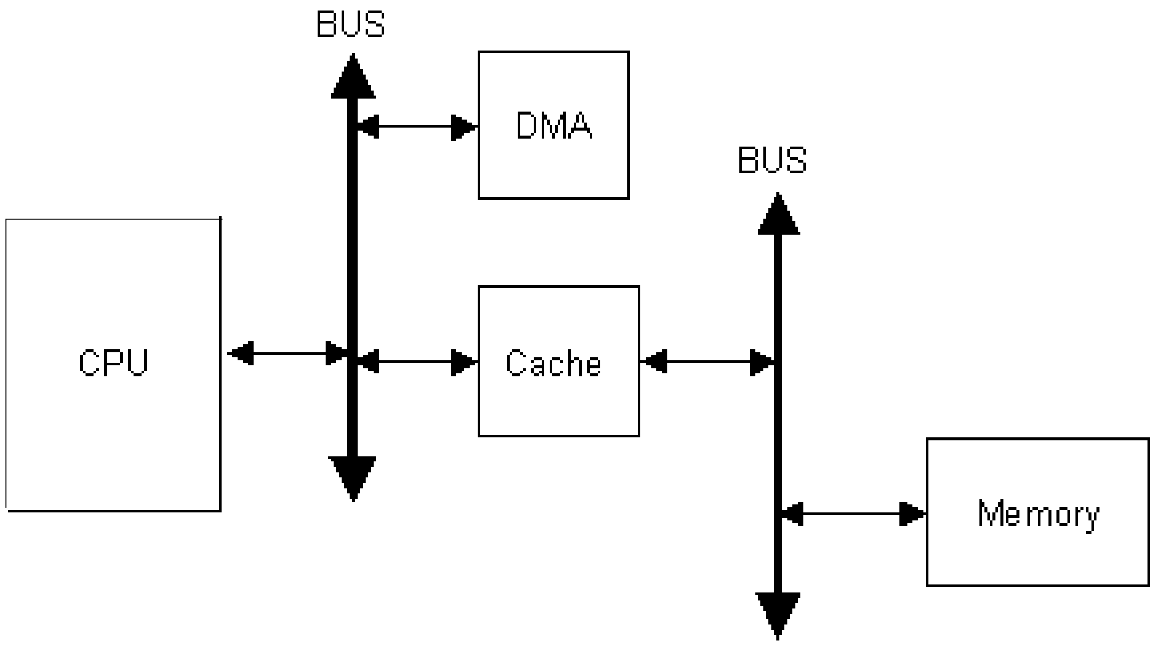
Cached Versus Non-Cached Data

When the CPU must read or write directly from a memory mapped I/O device without caching the result, the CPU tags the addresses (vccAddress class) of these requests with the *cacheable* flag. Therefore, the cache can differentiate between a cached and a non-cached request by testing that flag. If the flag is false, the cache passes the request (unchanged) to the memory. Otherwise, the cache checks for a hit or a miss.

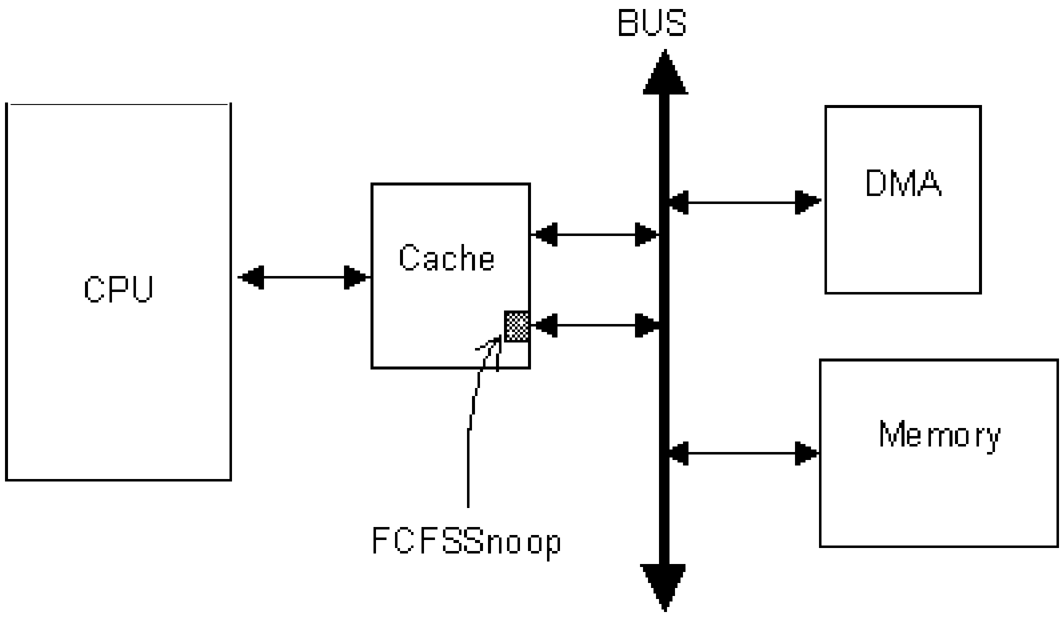


Cache Coherency

In an architecture with a cache and a device, such as a DMA or an ASIC, which is capable of reading or writing directly to memory, it is important to keep the memory and the cache coherent. For example, if a DMA writes to a memory line that is also in the cache, the line in the cache has to be invalidated. Although the actual data is not stored in the cache, VCC provides a mechanism that models the performance penalty that would occur if real coherency took place.



VCC provides a cache snooping service definition to solve the coherency problem. To accomplish this, VCC uses a cache snooping port, of type FCFSSnooper, which works with the FCFSBusAdapter, as shown in the following figure.



The snooping service definition works as follows:

1. The DMA places a request on the bus to read from memory.
2. The bus forwards the request to the memory and to every snooping port.
3. The cache evaluates whether the request invalidates the line in cache, updates the memory, or both.

   This decision depends on the type of the transaction, the state of the line, and the whether there is a hit or a miss.

If the cache line is dirty, the cache interrupts the DMA request from memory and writes the line back to memory. At this point, the DMA request to the memory can resume.

The following table shows the behavior of a cache when it receives a snoop transaction.

Cache snoop interface for a cache with write-back policy only		**Clean**	**Dirty (occurs only in write_back caches)**
From State			
Hit	Read	Clean	Write line back to memory Clean
	Write	1. Write_invalidate: Remove cache line 2. Write_broadcast: Write to cache line, (Clean)	Write line back to memory 1. Write_invalidate: Remove cache line 2. Write_broadcast: Write to cache line, (Clean)
Miss	Read	Do nothing	
	Write		

The cache filters all missed transactions because they do not need any action.

Also, the cache keeps a state of {clean, dirty} on each of its lines. *Clean* means it is identical to its copy in memory. *Dirty* means that the line is the most up-to-date and that its copy in memory is invalid.

Some cache systems have several states representing a cache line. One such state is Invalid, which means that the cache is older than memory. In VCC, because an invalid line must be brought back from memory anyway, VCC removes it from the cache. The two modes of *write_invalidate* and *write_broadcast* are supported.

The cache model does not store the data itself. It stores only a directory of references and uses this to compute hits and misses. Because behavioral memories are mapped to architectural memories and because sometimes architectural memories should be cached, the cache service component cannot handle data storage.

You can extend the cache service component to make it store the data, however, this causes significant degradation in memory usage and performance.

Behavior Timers

You use behavior timers to model the periodic or prescheduled activation of a behavioral model. You can program and service timers from separate behavior blocks from any level of the behavior hierarchy, but the timer references must be declared at all intervening levels of hierarchy. Timers are not global.

When a timer expires, any leaf-level block that has an event class timer reference is activated. The *TimerExpired()* function can be used to check the expiration of a timer. For more information on the *TimerExpired()* function, see the information about modeling timers in “Creating and Importing Blackbox C++ Models” in the *VCC Modeling Guide*. Alternatively, you can wait on timer expiration using the embedded wait capability.

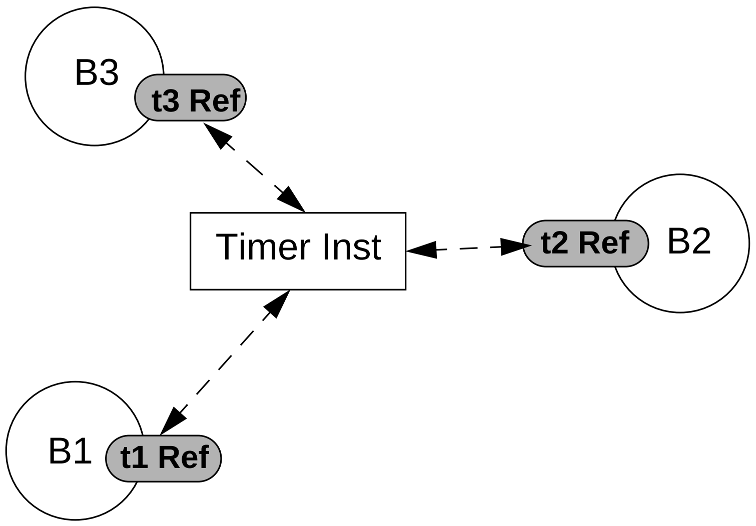
The VCC simulator defines three declarations for timers:

- Event declaration
- Program declaration
- Program and event declaration

When you do a performance simulation, the communication between behaviors that reference timers and the architecture implementation of a timer instance takes place through timer patterns.

The following figure illustrates how you use behavior timers in the VCC environment.

Behavior Timer Example

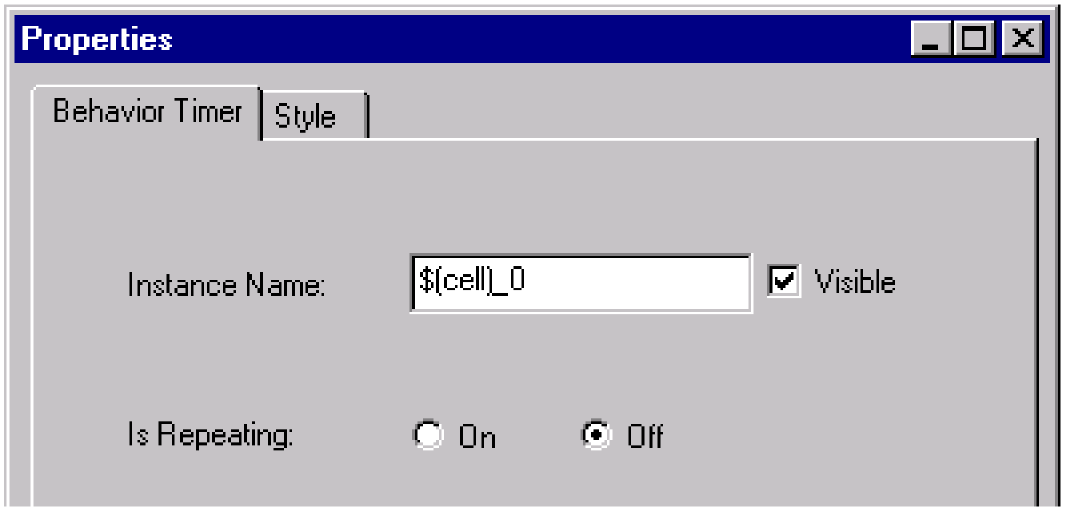


The process for using behavior timers is as follows:

- Place the timer instance in your behavior diagram.
- For each behavior to communicate with the timer, declare a timer reference on the behavior model.
- Associate each timer reference to a timer instance.
- In the behavior model, write code that starts or cancels the timer, or write code to be notified when a timer expires.

Place the Timer Instance

When you place a timer instance in a behavior diagram, you need to give the timer a name, and specify whether or not the timer is repeating.


Properties

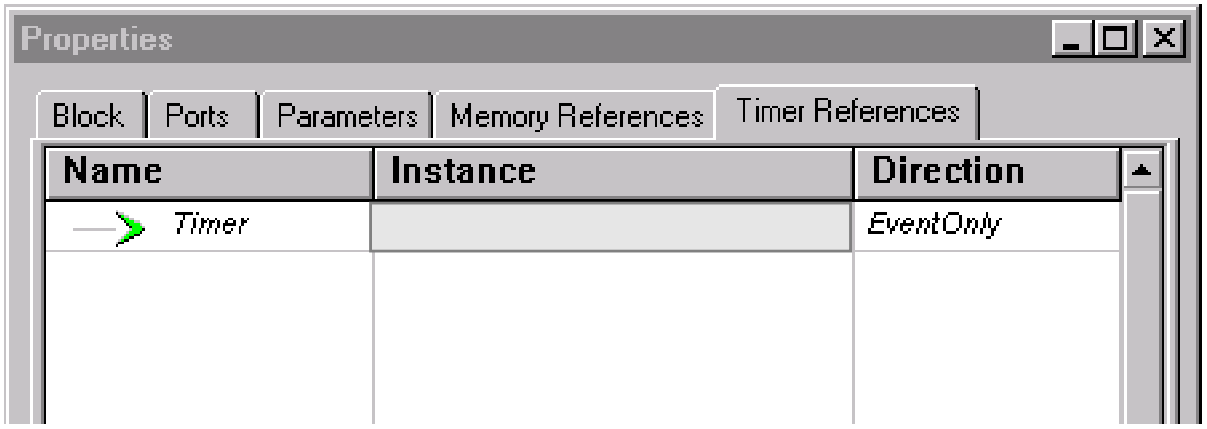
Behavior Timer | Style

Instance Name: $(cell)_0 [✓ Visible]

Is Repeating: On [✓ Off]

Declare the Timer Reference

In the symbol view of the behavior model, use the Timer References tab to specify the direction and the name of the timer reference.


Properties

Block | Ports | Parameters | Memory References | Timer References

Name	Instance	Direction
Timer		EventOnly

Associating a Timer Reference with a Timer Instance

Associate the timer reference on the behavior instance with the timer instance using the Timer References tab of the Properties dialog for the behavior.

Interacting with the Timer from within Your Model

In the behavior model, write code that starts or cancels the timer, or write code to be notified when a timer expires. The following sample code determines if the t1 timer reference has expired:

```
expired = t1.TimerExpired();
```

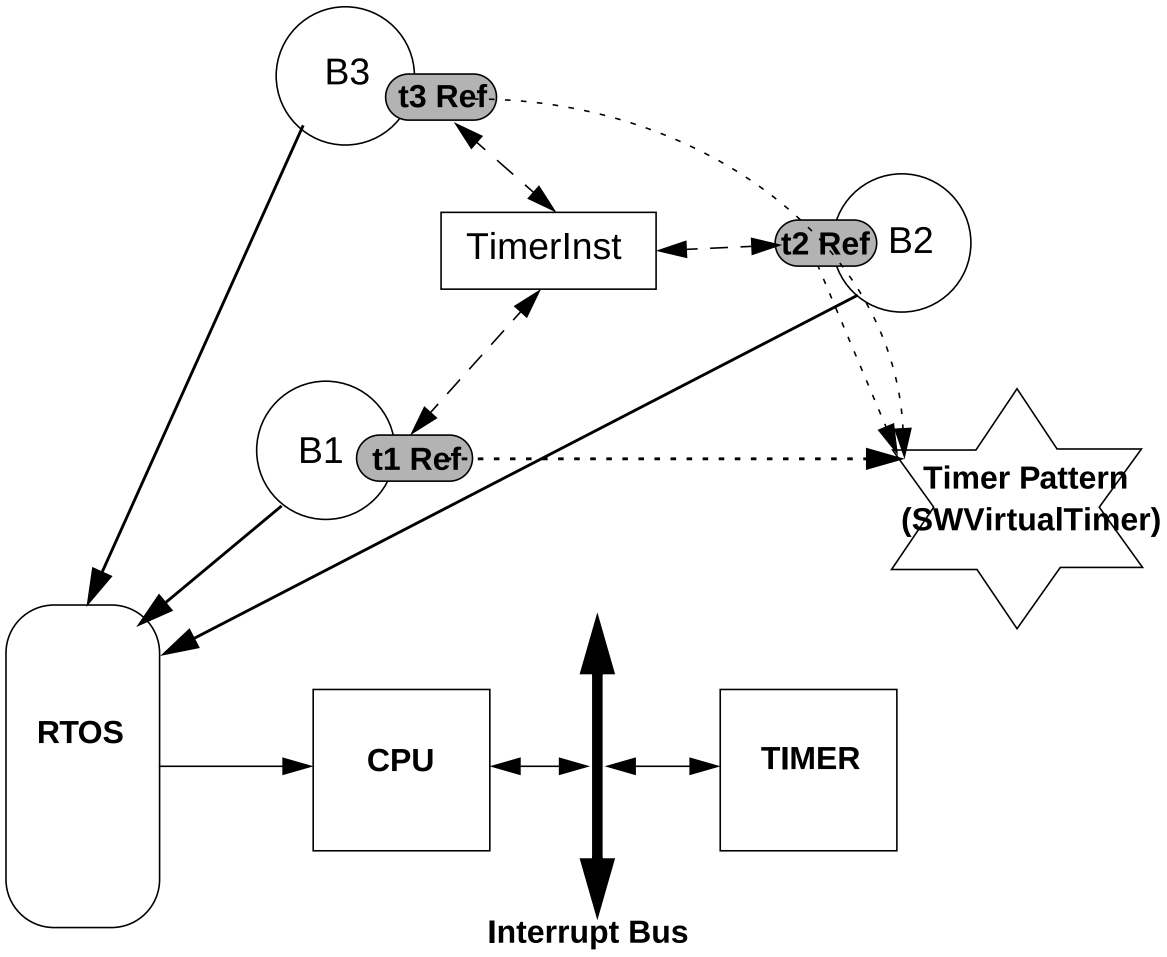
You can also use the following functions:

```
SetTimer()
StartTimer()
CancelTimer()
GetTimer()
```

Mapping the Timer

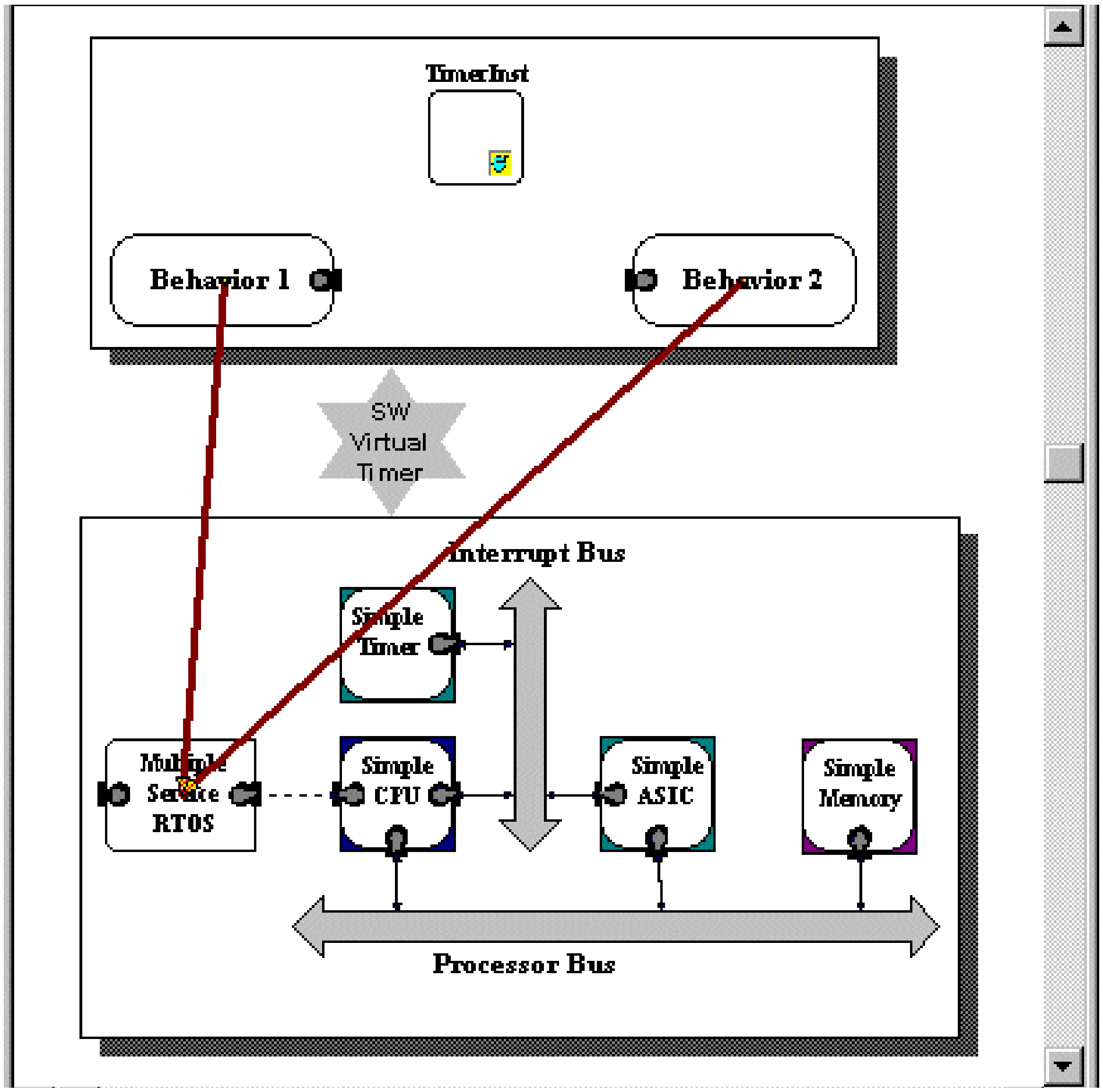
To map the timer instance, you need to map the timer references

Mapped Timer Example

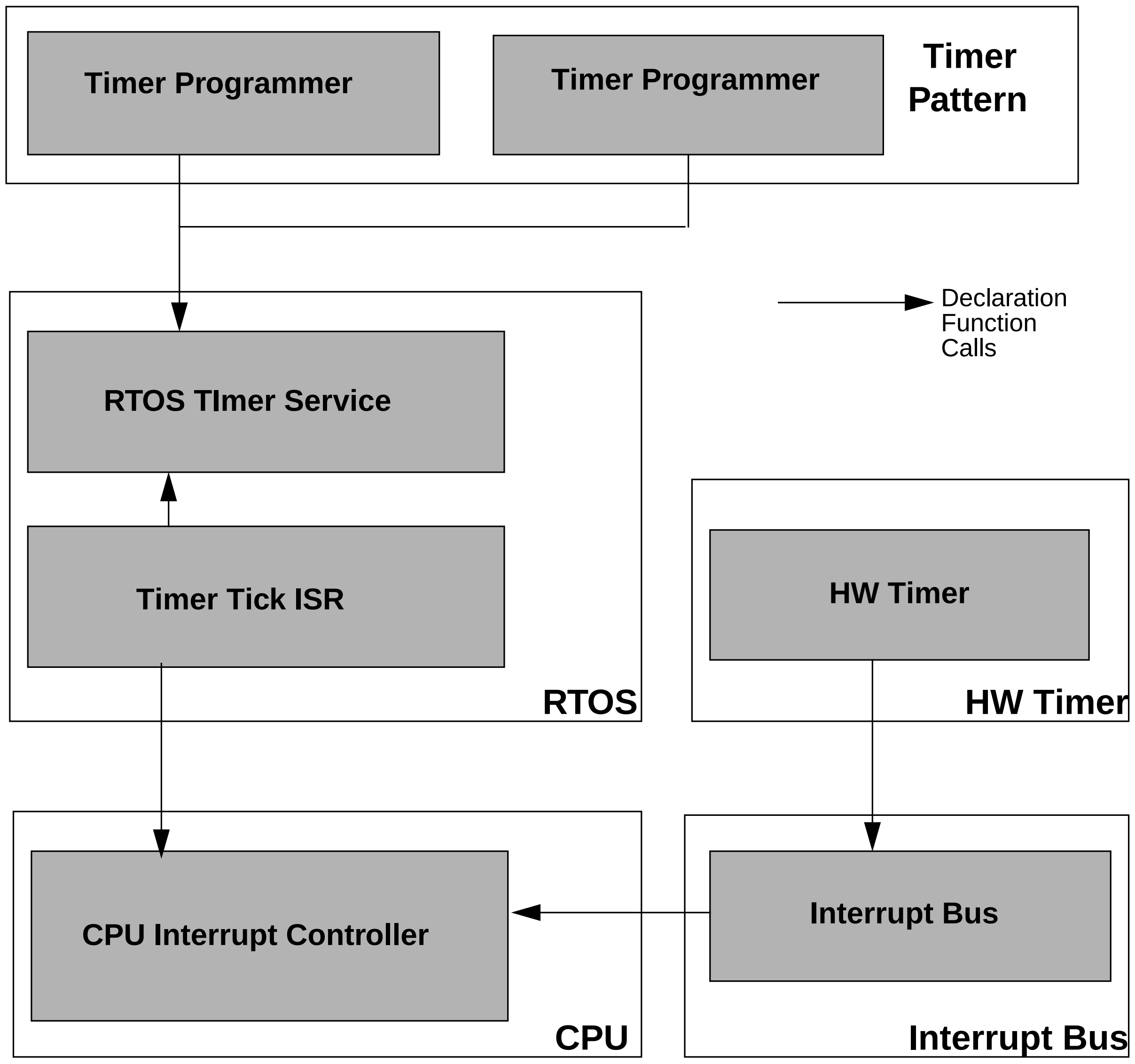


Each of the arrows in the diagram represents a function call. The dashed lines represent the associations between timer reference and timer instance. The dotted lines represent the communication pattern chosen for that timer reference.

The following figure also demonstrates behavior timers in a mapping diagram. The two behaviors share a timer instance, which is mapped to an RTOS.



The following figure illustrates how the service definitions associated with the patterns and architecture resources shown in the previous figure are arranged in the final simulated netlist.



When you select and highlight the timer instance, you can attach probes and breakpoints to it.

Mapping the Timer References

A timer reference is mapped to a timer pattern. The timer pattern defines services for the event, program, and program/event declarations. These services define the behavior timer API in terms of services provided by the architecture component to which the timer reference is mapped. In the Mapped Timer Example on page 92, the *t1 ref* is mapped to an RTOS, therefore, the RTOS must provide a service to support the timer pattern.

Typically, the use of a behavior timer implies the use of a software timer. However, a behavior timer might represent an internal timer on an ASIC. In both situations in the VCC environment, all behaviors referencing the same timer instant must be mapped to the same architecture resource.

See the *VCC Library Reference* for details on the timer services.

Software Timers

The VCC software provides a sample RTOS timer service. The implementation of the timer service consists of two service definitions:

- *SWTimers* service definition
- *TimerTickISR* service definition

The *SWTimers* service definition implements the *SWTimerDeclaration,* which is used by the patterns. The *TimerTickISR* loops through a list of active timers, decrements their values, and activates any tasks whose timer has just reached zero, whenever the ISR is activated via an interrupt. Thus, the *TimerTickISR* is closely tied to the *SWTimers* service definition. This provides a good example of a situation where the ability of one service definition to instantiate another service definition is required. The *SWTimers* service definition needs to declare that the *TimerTickISR* is instantiated during simulation. The concept of service definition instantiation is described in the *VCC Modeling Guide*. The *TimerTickISR* also needs to use the *InterruptDeclaration* that is assumed to be implemented by an interrupt controller. This allows the *TimerTickISR* to connect to an IRQ on the interrupt controller.

*See the VCC Library Reference for details about the SWTimer*s and *TimerTickISR* Service Definitions

Hardware Timers Service Definition

A RTOS that offers timing services is closely tied to a hardware timer/counter. Each microprocessor usually has its own timer that is uniquely configured. Therefore, the hardware timer cannot be independent from the RTOS. However, all hardware timer/counters usually share the same basic set of functions:

- Counting based on the system clock
- Counting based on level or edge-triggered events

- Generating interrupts on timer overruns

The RTOS generally uses the timer interrupts generated by the counter derived from the system clock. This gives it a periodic pulse with which it can keep track of time and virtual timers. To make the hardware as independent from the RTOS as possible, the hardware timer model provided by the VCC software only generates periodic pulses for the RTOS and can only be used by the RTOS.

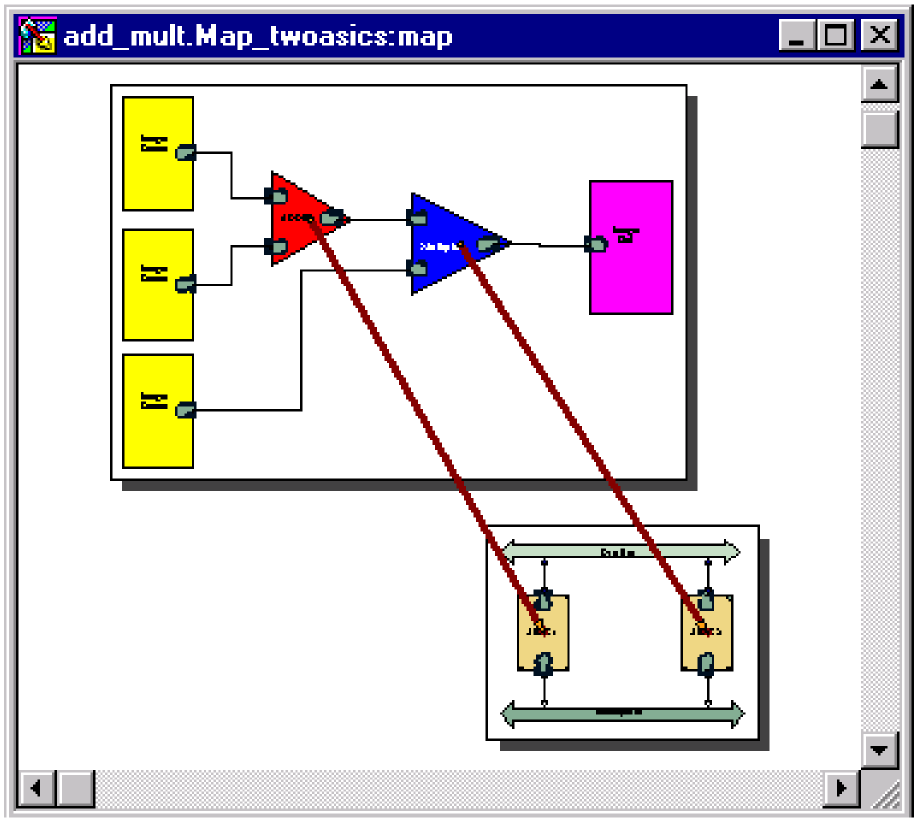
If you have platform specific-code and you use the hardware timer explicitly (not via RTOS calls), you can write your own timer models that can be configured using memory mapped registers.

If there are no hardware timers in the architecture, an ASIC resource can be used.

4

Analyzing Behavior Delay

After you create and functionally simulate a behavior diagram and decide on a target architecture, you are ready to map behavior models to architecture models. The following shows a mapping diagram for a simple mathematical system.



Mapping is a continual refinement process in which you make some design decisions, analyze the results, then repeat the process with different design decisions. You can work through this process to create a more accurate performance analysis.

For example, after you decide on the hardware/software partitioning for your design, you can do performance analyses to determine the delay of running the behavior in hardware versus software. (A performance analysis also factors in the contention and scheduling overhead for running behaviors in software.) Then, you might make design decisions about the communication between the behaviors. You can run more performance analyses based on these decisions— these analyses can factor in bus arbitration and transfer delays between hardware and software blocks. Based on these analyses, you might change your hardware/software partitioning or choose a different bus model.

To add another level of refinement, you can choose the specific pattern of communication between your behaviors. A hardware to software communication can use interrupts or polling communication. When you do a performance analysis based on this choice, the results are more accurate because the analysis models the correct control flow as well as shared memory accesses and interrupt architecture.

All accesses to memory must be assigned to architecture memories. For example, you need to decide whether the code and data sections are to reside in RAM or ROM. You also need to decide if cache or DMA is to be involved in the memory accesses. This level of refinement more accurately models the instruction and data fetches for software.

This chapter provides more details on how to refine your mapping diagram.

Creating the Mapping Diagram

You can create a mapping diagram using the *File -> New* command. Next, you instantiate your top-level behavior and architecture diagrams in the mapping diagram.

Mapping Configurations

A mapping configuration must be associated with a mapping diagram. The mapping configuration specifies the rules for binding

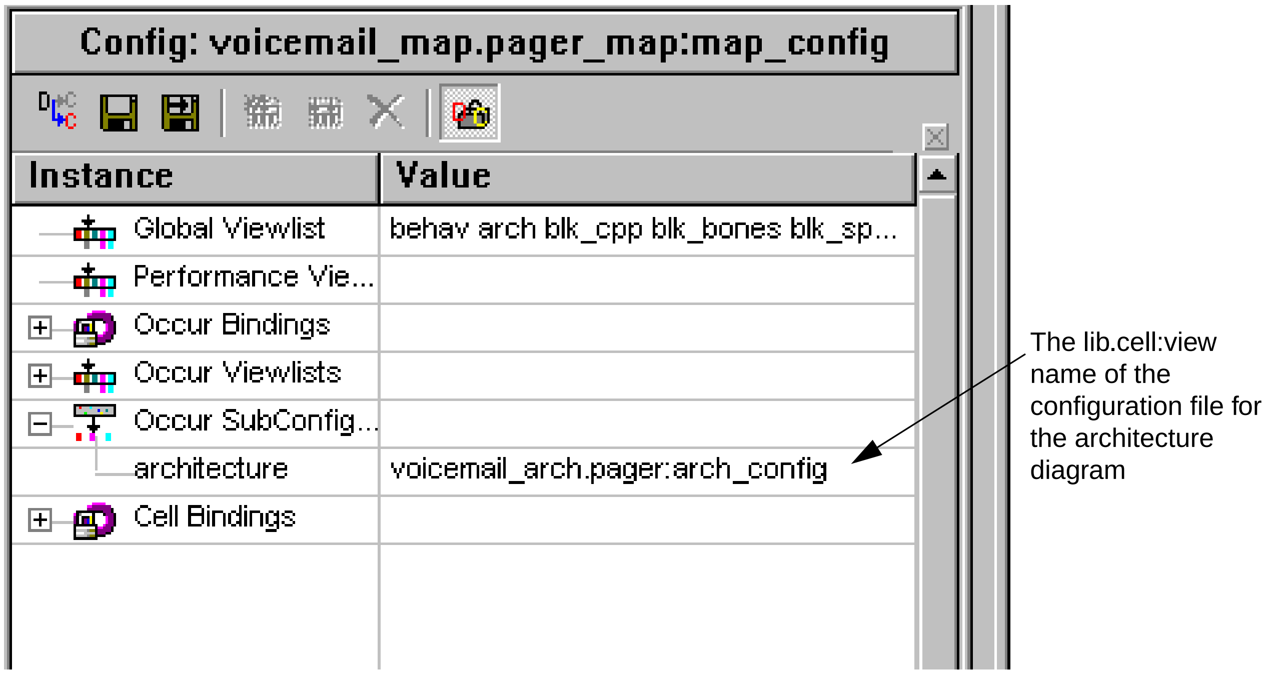
- The hierarchical behavior diagram
- The hierarchical architecture diagram

- The performance and implementation models for the mapping links

Your mapping diagram can reference an existing configuration or the default system configuration. For more information about working with configurations, see the VCC Help.

When you instantiate your top-level behavior and architecture diagrams in the mapping diagram, the most recently used configuration associated with each is included as a sub-configuration of your mapping configuration. Therefore, the rules for binding the hierarchical behavior design in the mapping diagram are the same as the rules used for the hierarchical behavior design you functionally simulated. If you change the hierarchical design of your behavior diagram, the changes are automatically reflected in the mapping diagram.

The following figure shows a sample sub-configuration rule in the mapping configuration.


Config: voicemail_map.pager_map:map_config

Instance	Value
Global Viewlist	behav arch blk_cpp blk_bones blk_sp...
Performance Vie...	
Occur Bindings	
Occur Viewlists	
Occur SubConfig...	
architecture	voicemail_arch.pager:arch_config
Cell Bindings	

The lib.cell:view name of the configuration file for the architecture diagram

Analyzing the Behavior Delay of a Hardware/Software Partition

You partition your design into hardware and software by mapping behaviors to architecture instances using the *Mapping Connection* command. You can map a leaf-level behavior to an RTOS or scheduler for software, or you can map the behavior to an ASIC for hardware. Typically, you do not map testbench models because they are not part of the design to be implemented.

General Mapping Guidelines

Map each behavior model to a single architecture model. If a behavior needs to map to more than one architecture model, separate the behavior into multiple behavior models so that each one can map to a single architecture model.

If your behavior design is hierarchical, use the *Mapping > Hierarchical Mapping* command to expand the next level of hierarchy for mapping.

To make your mapping diagram easier to read, use more than one copy of the same architecture diagram in your mapping diagram. Mapping to any architecture diagram has the same effect.

If the design is large, use the Mapping Table to assign mapping connections. See the VCC Help for more information about the Mapping Table.

After mapping your behaviors, you need to specify the performance model for each mapping link. VCC supports three types of behavior performance models:

- Delay scripting language (DSL) model
- Annotated C model
- Annotated C++ model

Because a behavior can have multiple performance models, you must choose which performance model is used for a particular performance analysis. The performance model bindings are stored in the mapping configuration. The mapping configuration has a performance viewlist, which lets you select performance views. Alternatively, you can explicitly bind the performance view for each mapping connection.

Note: When the architecture model specifies the performance model view, as in the case of the predefined ASIC, this specification overrides the binding in the mapping configuration.

Using the Performance Viewlist

The performance viewlist binds the performance view of each mapping connection that has not been explicitly bound on the mapping connection itself. The performance viewlist is an ordered list of view names specified in the mapping configuration. During performance simulation, VCC checks the first view name in the performance viewlist against the views provided by the behavior model. If there is a match, that performance view is used. If there is not a match, VCC continues down the performance viewlist until a match is found.

The performance viewlist works well when all behavior models in a design have a consistent set of view names for their performance models. If a behavior model has more than one performance model, the choice of performance view name can indicate different levels of performance accuracy or analysis types (delay, power). If you apply these choices consistently across the entire design, you can switch between levels of accuracy or analysis types by modifying the performance viewlist.

Binding Performance Models Explicitly

You can select or change a specific performance model view for any mapping connection in your mapping diagram. You can bind the view from the mapping diagram or from the mapping configuration.

- From the mapping diagram, right click the mapping connection and select *Bind View*.
- With the mapping diagram active, select the *Hierarchy* tab to display the current mapping configuration. Click on the mapping instance name, which displays *Simulation* and *Implementation* entries. Click on *Simulation*, then click on *Behavior*. This lists the behavior instances and the mapping connections in a hierarchical format. You can choose a mapping connection and right click to access the *Explain View Binding* or *Edit Occur Binding* commands.

  See the VCC Help for more information about these commands.

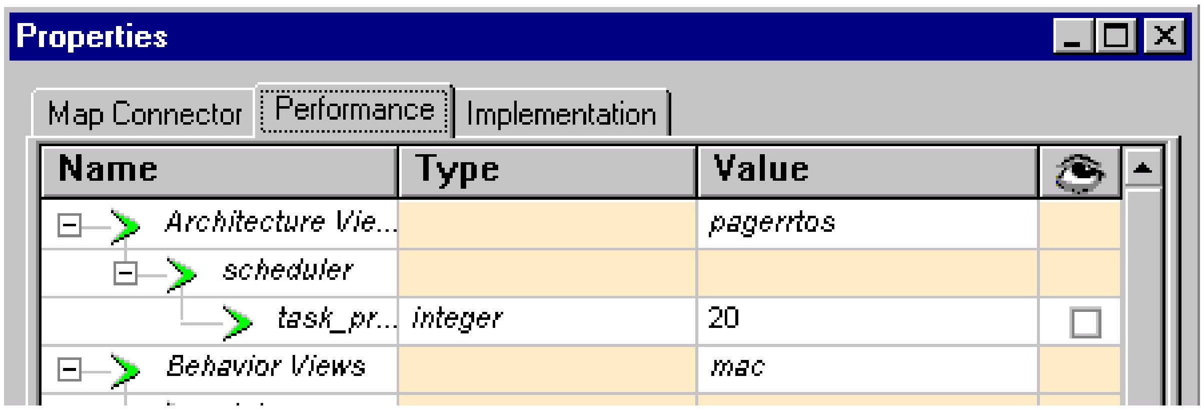
Mapping Parameters

Mapping parameters are unique to the mapping between behavior and architecture models. To display mapping parameters, right click on the appropriate mapping connection and select *Properties*. Alternatively, you can use the *Mapping Parameters* command to see all the parameters in the diagram.

Two types of properties are associated with mapping:

- Link parameters of the target architecture model
- Exported parameters of every performance model

Link parameters are retrieved from the architecture model definition. The following figure shows the link parameters for a behavior model mapped to an RTOS model. (The link parameters are extracted from the RTOS model definition.)


Properties

Map Connector | Performance | Implementation

Name	Type	Value	
⊟ Architecture Vie...		pagerrtos	
⊟ scheduler			
task_pr...	integer	20	☐
⊟ Behavior Views		mac	

In this example, the `task_priority` link parameter identifies the scheduling priority for the mapped behavior. Refer to the library documentation of your architecture model to determine the property settings that are appropriate for your design.

You are prompted for exported parameters for every performance view. You need to set only those parameters used in the configuration. In the Implementation tab, there are parameters for every implementation view. You do not need to set these implementation parameters for performance analysis, but you must set them before exporting your design through the Links to Implementation flow.

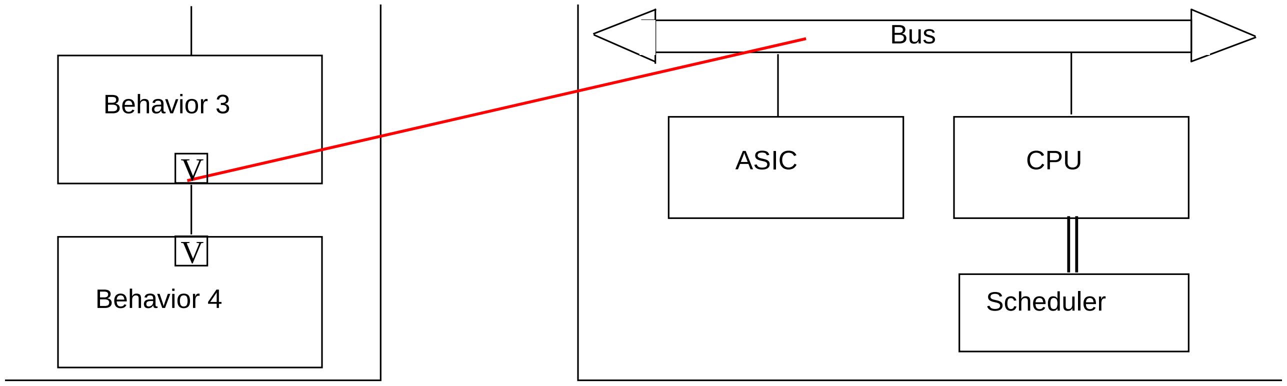
For information about creating link parameters and exporting parameters, refer to *VCC Help* and the *VCC Modeling Guide*.

You are now ready to run a performance simulation of your mapped design. See Chapter 5, “Performance Evaluation,” for more information.

Analyzing Bus Traffic

Mapping a communication wire or net in the behavior diagram to a bus in the architecture diagram associates a delay that represents the time required to transfer data. This delay also relates to the arbitration policy used for the bus.

A communication wire connecting behavior models represents data transmission between the models. Mapping connects the output port of the behavior to the bus that transmits the data.



Depending on the arbitration model used by the bus, you might need to specify values for link parameters. Refer to the *VCC Library Reference* for descriptions of link parameters required by the VCC_bus library models.

You can also map a net to a pattern. For more information, see "Refining Communication Patterns" on page 106.

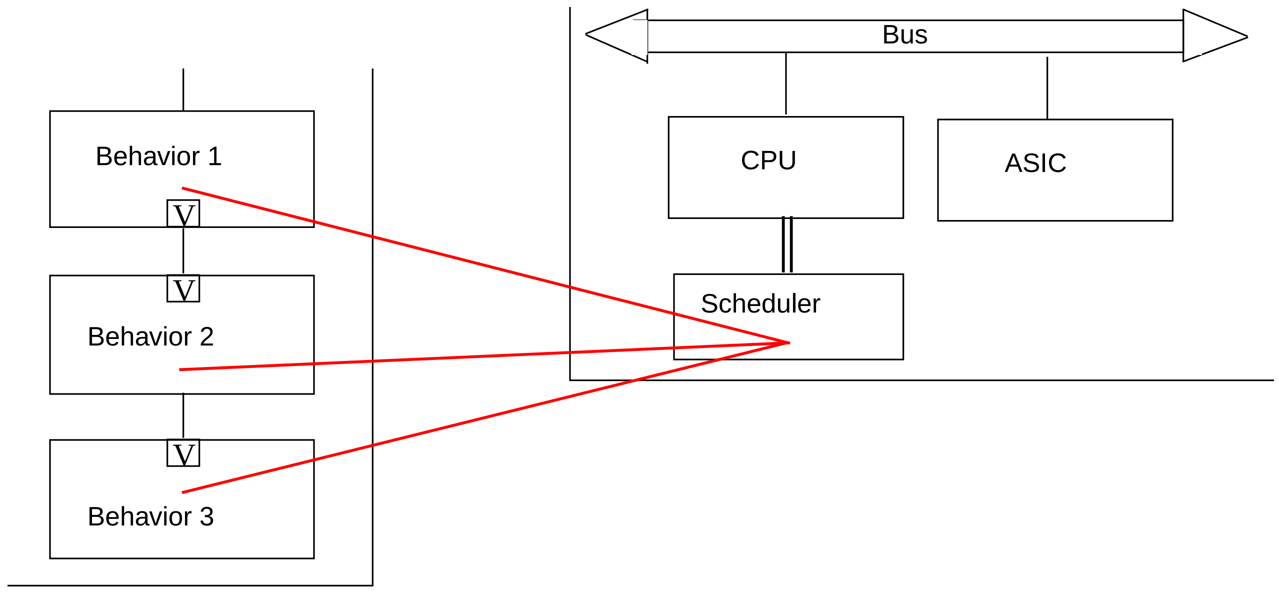
Bus Mapping Rules

You do not need to map all communication wires. Depending on the goal of your simulation, no mapping or implicit mapping might be acceptable.

No Mapping

By default, no communication delays are executed for unmapped communication wires. This default might be acceptable if the goal of your simulation is to estimate the delays of executing behaviors on various architectures without data communication delays being factored in. Usually, you do not need to map testbench nets.

No mapping might also be appropriate if the communication is between behaviors that are mapped to the same architecture model, as shown in the following figure. In these cases, there is no transmission delay to estimate.

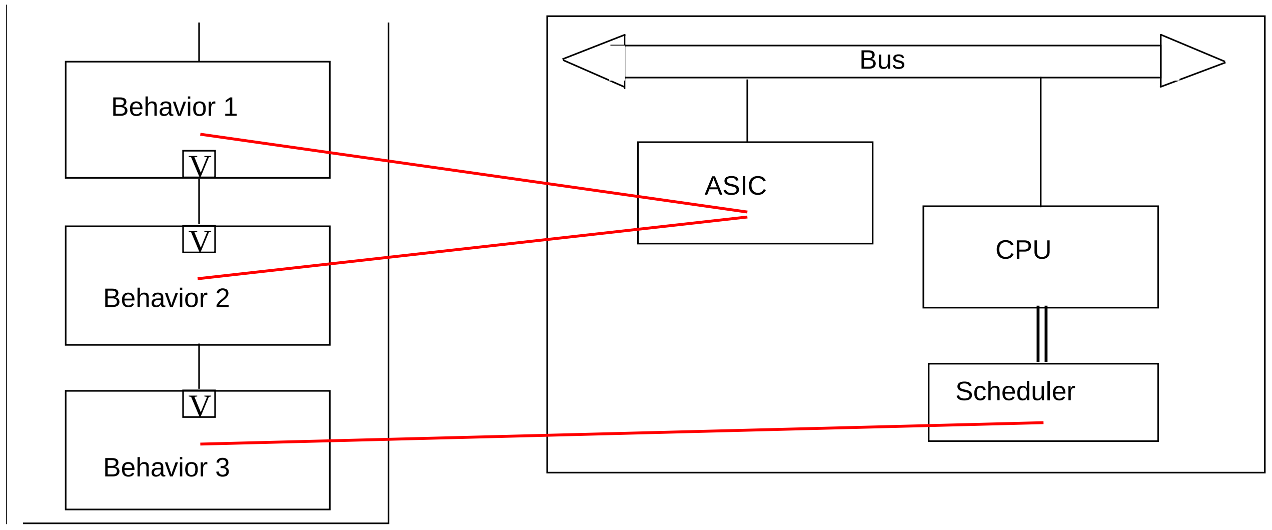


Implicit Mapping

At simulation time, you can specify implicit mapping for unmapped communication wires. Data transmission through an unmapped communication wire between two behaviors can be estimated if all of the following conditions are true:

- The sender and receiver behavior models are explicitly mapped.
- There is a unique, unambiguous bus connection between the two architecture models to which the two behaviors are mapped.
- If any link parameters exist for the bus model to which the communication wires are implicitly mapped, they must have a default value that can be used. (Without explicit mapping, there is no way for you to set link parameter values.)

The following figure shows a situation where the bus being used for transmission can be implied from the mapping of the behavior models themselves.



Implicit mapping is restrictive. You cannot specify different priority levels for communication wires that are implicitly mapped to the same bus.

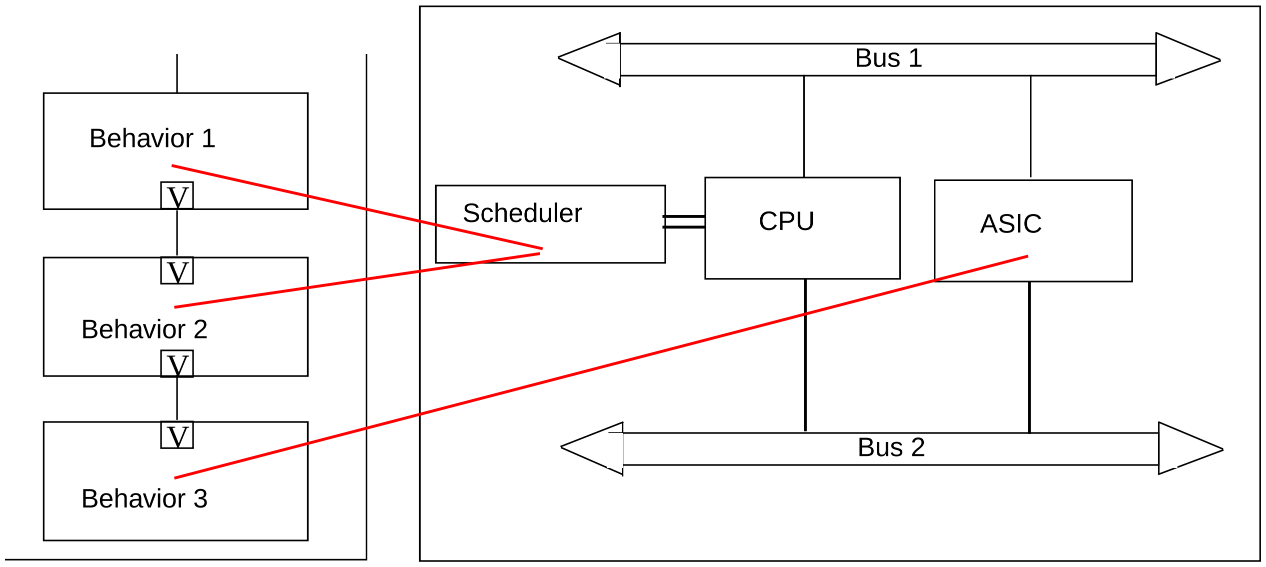
Explicit Mapping

In many situations, you must create mapping connections explicitly to accurately calculate the performance.

If the bus has a link parameter that requires a different value for each communication wire that is mapped to it, you must use explicit mapping.

If the communication wire between behaviors is mapped to architectures connected to more than one bus, and either bus can be used for the same data transfer, you must map the communication wire explicitly to determine which bus to use.

The following figure shows a situation where explicit mapping is needed to unambiguously represent the bus to use for the communication between Behavior 2 and Behavior 3.



Refining Communication Patterns

In the previous section, a communication wire is mapped to a bus, which causes bus loading. For a more refined simulation, you can map a communication wire to a communication pattern.

You can create the mapping explicitly by instantiating a pattern and binding the performance view of the pattern using the *Bind View* command on the pattern instance. Alternatively, you can bind a pattern by modifying the mapping configuration. You can modify the *Global Viewlist* in the Configuration tab or the *Edit Occur Bindings* in the Hierarchy tab of the mapping configuration.

You must bind the performance view before running performance analysis. You do not need to bind the implementation view until you are ready to export your design through the Links to Implementation flow. The pattern might have parameters, which you can set using the *Properties* command on the instance. You need only set the parameters for the performance view chosen.

Once the pattern instance is instantiated, you can map a behavior port to the pattern instance using the *Mapping Connection* or *Mapping Table* Command. The *Mapping Connection* command prompts you to graphically draw a link from the behavior port to the pattern instance.

For large designs, you can use the mapping table to map your communications. The mapping table lets you sort your mappings. In addition, this command supports an auto-mapping feature, which maps all unmapped connections based on the pattern category to which the communication belongs.

Pattern Categories

Communication patterns are categorized based on whether the start and end of the communication path is in hardware or in software, as listed below:

- Software-to-Hardware Communication
- Hardware-to-Hardware Communication
- Hardware-to-Software Communication
- Software-to-Software Communication
    - Intertask—Intratask communication is for two behaviors mapped to the same single task, where protection of overwriting variables is not needed.
    - Intratask—Intertask communication is for two behaviors mapped to the same RTOS, where protection using masked interrupts or semaphores might be appropriate.
- Software to Memory (SW->Memory)
- Hardware to Memory (HW->Memory)
- Software to Timer (SW->Timer)
- Hardware to Timer (HW->Timer)

VCC also supports separate categories based on the type of data being transferred. Typically, a trigger data type represents a simpler communication than the transfer of a large data token. For example, you might choose shared memory as the default pattern for all software-to-hardware communication. When communicating only an event, however, using shared memory might be inefficient. You can specify that register mapped communication be used for trigger-only port communication. VCC supports trigger and non-trigger versions of the pattern categories.

A default pattern is assigned to each of the pattern categories. The auto mapping feature (of the Mapping Table) maps the communication paths to the default pattern of the appropriate pattern category. The following table lists the patterns in each category with the system default shown in bold text. You can change the default pattern for a category using the Mapping Patterns tab, which is available from the *Tools > Options* command.

For more information about each pattern, see Appendix A, “Pattern Descriptions.”

Pattern Defaults

Pattern Category	Trigger	Non-Trigger
SW–>HW	**Register MappedSwHw**	**Register MappedSwHw** Shared Memory
HW–>HW	**Direct Connect** Register MappedHwHw	**Direct Connect** Register MappedHwHw Shared Memory
HW–>SW	**Interrupt** Polling Register Mapped Polling Shared Memory	**Interrupt Register Mapped**, Interrupt Shared Memory Polling Register Mapped Polling Shared Memory
SW–>SW (Intertask)	**Unprotected**	**Unprotected** Semaphore Protected Uninterruptable Protected
SW–>SW (Intratask)	**Unprotected**	**Unprotected**
HW->Memory		HWDirectMemoryAccess
SW->Memory		SWDirectMemoryAccess
SW->Timer		SWVirtualTimer

You can set the parameter values on the default pattern as follows.

1. Instantiate the pattern in the mapping diagram.

   For example, instantiate the *SharedMemorySwHw* pattern in your mapping diagram.

2. Set the parameter values on this instance.

   For example, set the *MemoryParticipant* to be *"Memory1"*.

3. Use the Properties dialog to identify this as the pattern instance to use in auto-mapping.

   With this flag set, the auto-mapper will map all the Sw->Hw (non trigger) communications to this pattern instance, which uses the "Memory1" to read and write the shared data value.

Mapping Parameters

Your mapping links might require some parameter values based on the pattern to which they are mapped. You can set these parameters with the *Properties* command, which appears when you right click on the mapping link. Alternatively, you can use the *Mapping > Mapping Parameters* command to see all the parameters in the diagram presented in a table format. You can sort the table by parameter value to determine which parameters are not set. You can also sort the table by parameter name, which is useful when setting interrupt numbers.

You must set the performance properties for the mapping link before running performance analysis. You do not need to set the implementation view parameters until you are ready to export your design through the Links to Implementation flow.

Performance Analysis of Patterns

The performance analysis of a pattern requires that appropriate architecture services are available in the architecture diagram. Therefore, you might need to refine your architecture diagram before running performance analysis. For example, if you use the *SharedMemory* pattern, you need to add the appropriate RAM and ROM to your architecture diagram. Another example is the interrupt pattern, which requires an appropriate interrupt bus and an interrupt controller.

Finally, patterns require addresses to transfer data across an arbitrated bus. Therefore, you need to assign the bus range on your data bus and address sub-ranges on the ports of each architecture instance connected to the bus. You can set the offsets for each communication path as a parameter on the mapping link, or you can use the *Address Allocator* command to automatically assign unique offsets.

Note: If you bind the *fixed_pattern* view for your pattern, VCC models only the loading of the bus specified by the simulation bus parameter—It does not use the architecture services.

Analyzing Memory Access

You can perform further mapping refinement to accurately model the memory accesses of your system. You can model

- Instruction and data fetches for behaviors mapped to software
- Memory accesses due to behavior memory references
- Memory transfers handled by caches and DMAs

Instruction and Data Fetch Analysis

In order to model instruction and data fetches, you need to map your memory sections to architecture memories. For details about mapping your memory sections, see "Delays Based on Memory Accesses" on page 34.

The accuracy of the simulation is controllable by the *DataReadMode*, *DataWriteMode*, and *InstructionFetchMode* parameters on the *CPUMemoryAccess* service of your processor.

- *DataReadMode* controls the modeling of data reads (load)
- *DataWriteMode* controls the modeling of data writes (store)
- *InstructionFetchMode* controls the modeling of instruction fetches

You can set these parameters to *None*, *BusTraffic*, or *BusTrafficAndTaskDelay*. If you specify *None*, this feature is turned off. *BusTraffic* indicates that the appropriate bus is loaded with traffic, which causes contention on the bus. *BusTrafficAndTaskDelay* indicates that the appropriate bus is loaded with traffic and the task is blocked until the memory access is complete.

The bus arbiter models the performance impact of the bus request. The bus adapter models the performance impact of transferring the data across the bus. The architecture memory models the performance impact of the memory read or write. For example, VCC provides a *VCC_Memory.SimpleMemory* cell, which has performance parameters for specifying the setup, read, and write latencies.

Software Estimation

Behavior models written in Whitebox C are automatically annotated with instruction delays and optional instruction and data fetches during performance simulation to model the execution time of software on a processor. When you turn on instruction and data fetching, the annotated behavioral model communicates directly with the Memory Access service. This in turn initiates bus transactions.

For more information about annotated C models, refer to the *VCC Modeling Guide*.

Performance Analysis of Behavior Memories

In order to model behavior memory references, you need to map your behavior memory to an architecture memory and map your memory references to memory patterns. The default pattern is *DirectMemoryAccess*, which reads and writes directly to the architecture memory.   See "Mapping the Memory Instance" on page 72 for more details.

You can control the accuracy of the simulation using the *CommunicationMode* parameter on the *CPUMemoryAccess* service of your processor. You can set this parameter value to *None*, *BusTraffic*, or *BusTrafficAndTaskDelay*. If you specify *None*, this feature is turned off. *BusTraffic* indicates that the appropriate bus is loaded with traffic, which causes contention on the bus. *BusTrafficAndTaskDelay* indicates that the appropriate bus is loaded with traffic and the task is blocked until the memory access is complete.

The bus arbiter models the performance impact of the bus request. The bus adapter models the performance impact of transferring the data across the bus. The architecture memory models the performance impact of the memory read or write. For example, VCC provides a *VCC_Memory.SimpleMemory* cell, which has performance parameters for specifying the setup, read, and write latencies.

Cache Analysis

You can refine your design to model the impact of caches. The sample cache service provided by VCC supports modeling the cache

- As a statistical hit or miss ratio
- By tracking the addresses referenced

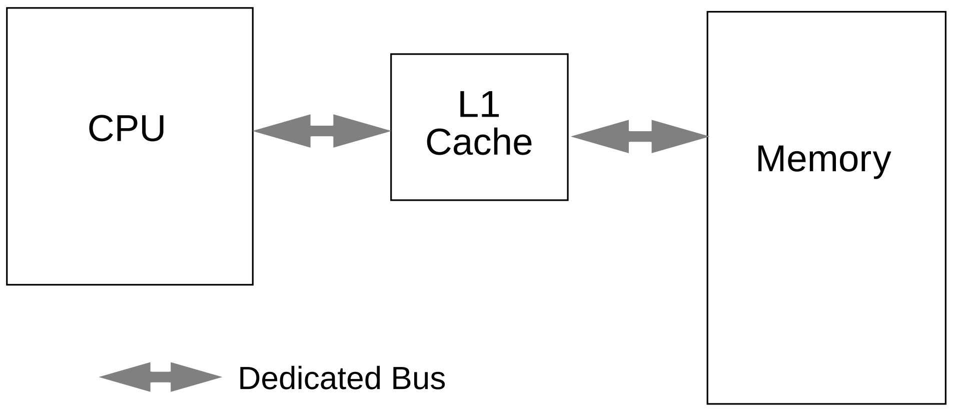
You can control this using the *fixHitRatio* boolean parameter on the cache service. In order to model the cache, you need to refine your architecture diagram to include the cache.

Architecture Diagram Refinements

The section describes several sample architecture diagram refinements using cache.

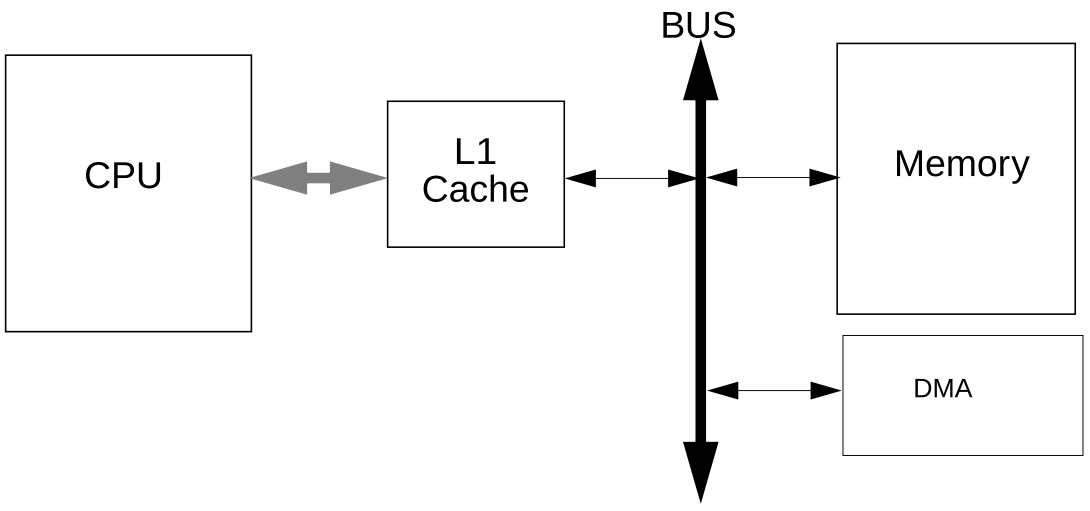
Cache on a Processor with Dedicated Memory

The following shows a cache on a processor with dedicated memory.



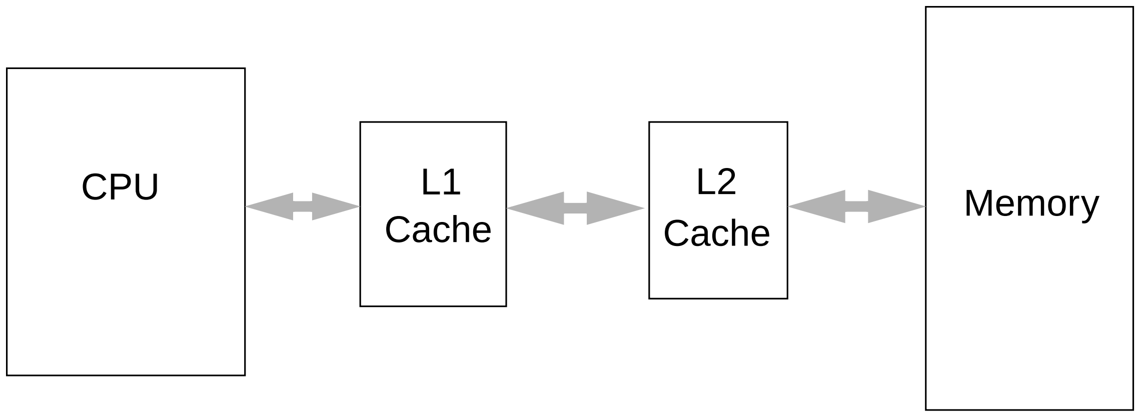
Cache on a Processor with External Memory

The following shows a cache on a processor with external memory. Because the DMA can write to the memory, the lines in the cache might need to be invalidated. In order to avoid this cache coherency, the cache must monitor the bus (or snoop) to get information about write transactions to the memory. You can model snooping by adding a snooper service to the bus port of the cache. Also, the databus must support a snooping cycle. For more details on snooping, see the information about the FCFSSnooper service in the *VCC Library Reference*.



Two-Level Cache

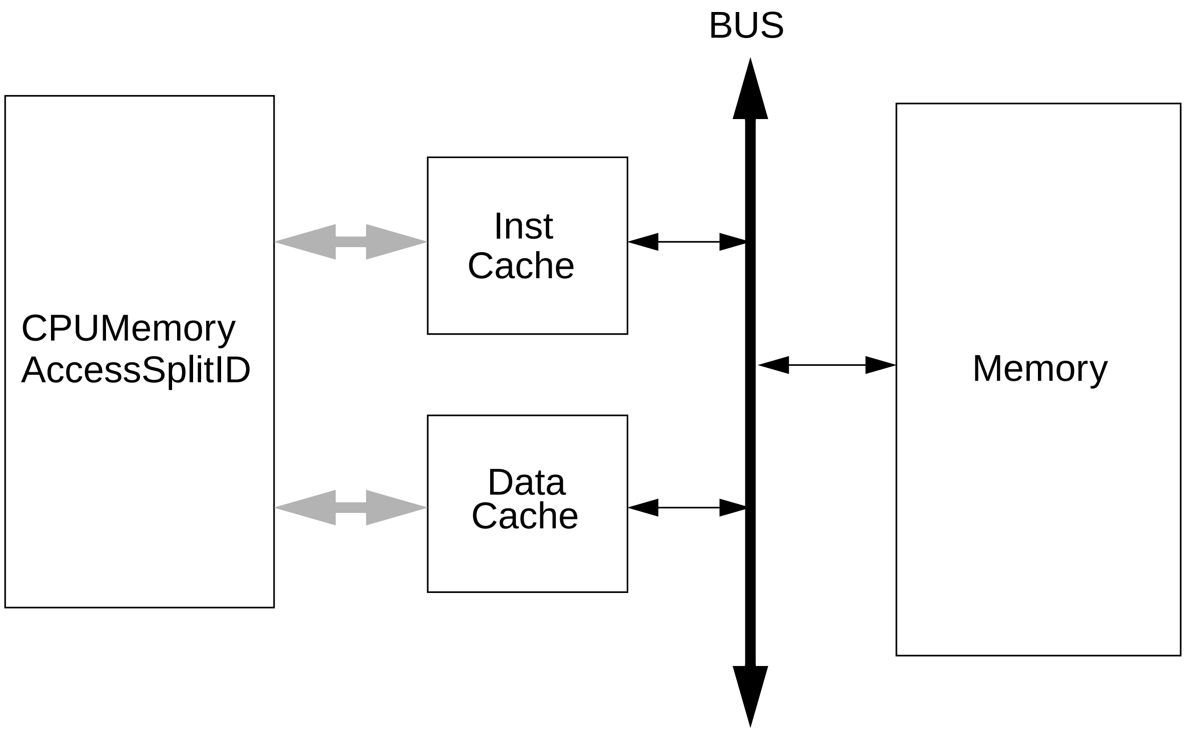
The following shows a two-level cache. VCC lets you model two- level caches by physically connecting the caches in a serial fashion. If the L1 cache has a miss, the request is handled by the L2 cache. If the L2 cache has a miss, the request is handled by the memory.



Separate Instruction/Data Caches

VCC supports separate data and instruction fetches when you use the *CPUMemoryAccessSplitId* service on your processor. This service sends address requests to the data or instruction port based on the type of the memory segment. If the code segment is of type *Data*, the service sends it to the port specified by the *DataBus* parameter. If the code segment is of type *Code*, the service sends it to the port specified by the

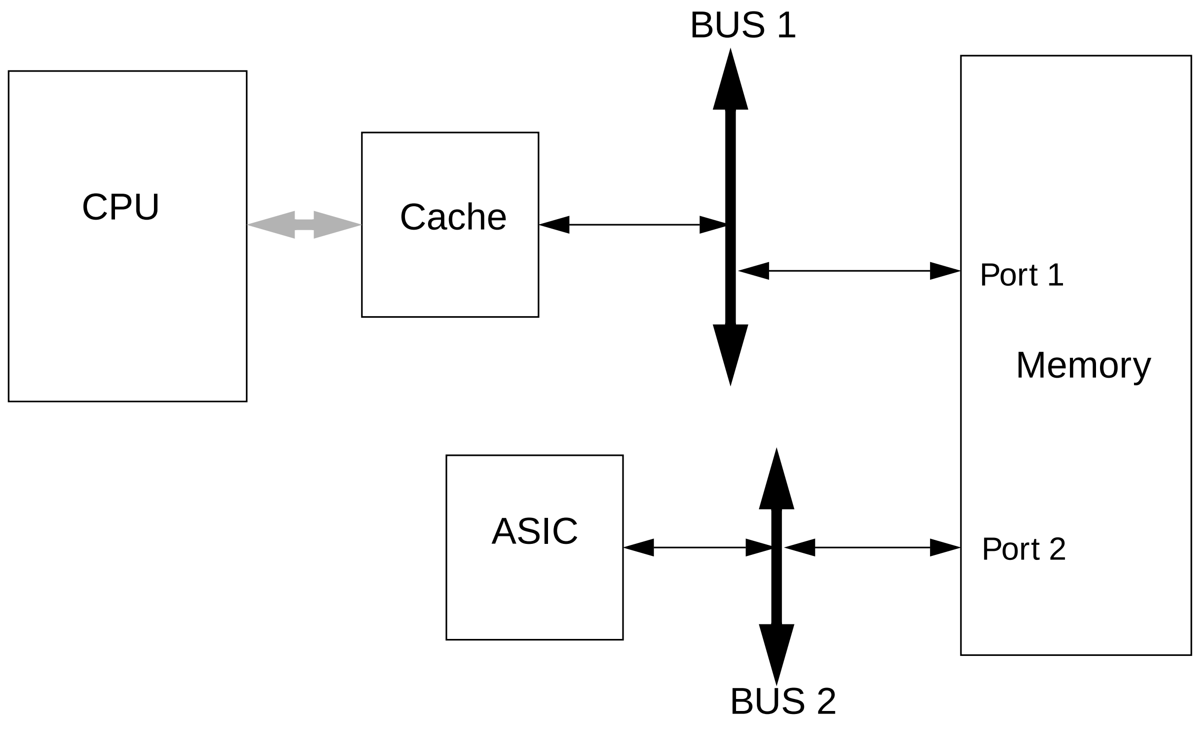
InstructionBus parameter. For more information on the *CPUMemoryAccessSplitID* service, see the *VCC Library Reference*.



Cache and Multi-Ported RAM

A multiported memory has multiple independent sets of address and data connections, allowing multiple independent memory accesses to proceed in parallel. The most common type of multiported memory is the dual-ported variety, which provides two simultaneous accesses. The address ranges of the ports of a multiported memory are the same, meaning that the same memory location can be accessed through either one of the ports.

The memory service definition provided by VCC is generic and can be bound to a memory architectural resource with more than one port. The following figure illustrates a dual-ported memory.



The ports in a VCC architecture are given unique IDs which are used to route transactions from source to destination through a network of buses, bridges, and caches.

The memory performance model does not take into account the effects of arbitration resulting from the case in which two requests on two ports are referencing the same memory location. This is modeled by delaying one request until the other request on the port is finished.

Cache Restrictions

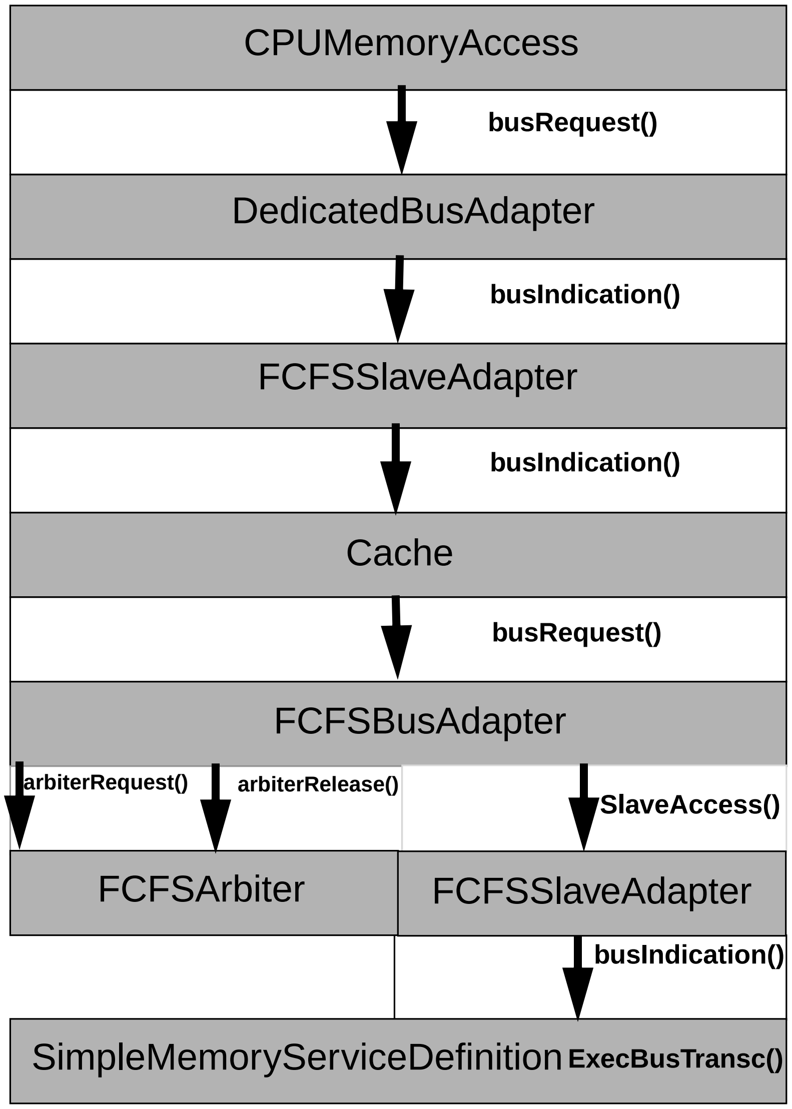
When you are modeling caches, you must follow these rules.

- A cache can only be divided into a number of lines that is a power of 2.
- All the memories that a cache services must also be divided into a number of lines that is a power of 2, where the line size of all the memories, and the cache that services them, is the same.

- A memory that is serviced by a cache will be cached in its entirety. No part of that memory may be marked as un-cachable. If this feature is needed, divide the memory into two memory categories, one that is cachable and one that is not.
- The cache model will not store the actual data. Only a directory of references that are presently in cache will be stored and used to compute hits and misses. Because behavioral memories will be mapped to architectural memories, and because there might be cases in which the latter should be cached, the present service definition cannot handle this. You can extend the cache service definition to make it store data. However, this causes significant degradation in memory usage and performance.
- Cache coherency in the presence of two-level caches is not supported.
- Cache coherency in the presence of multiple caches, where a cached line could exist in several caches, is not supported.

Definition-Declaration Interaction

The following diagram illustrates how the service definitions and service declarations used by a cache interact in the VCC environment.



Analyzing Timer Accesses

You can further refine your design to accurately model timer accesses. This includes modeling

- The overhead of managing the software timers

- The impact of SW timer interrupts on running tasks
- The impact of the HW timer generating the interrupt

In order to refine your design to model timer accesses, you need to map your behavior timer references to timer patterns, thus refining your architecture to support timers. For details about mapping behavior timer references, see "Mapping the Timer" on page 91.

To refine your architecture diagram, you need to add a SWTimer service to your RTOS to manage the software timers.   You can control the timer resolution and you can control whether or not to model the generation of the interrupt from a hardware timer. The *UseHWTimer* boolean parameter specifies whether the timer service schedules itself, or whether it will respond to an interrupt service routine that is triggered by a hardware timer. Setting this value will accurately model the traffic on the interrupt bus and the impact of calling the interrupt service routine for the timer tick.

If you set the *UseHWTimer* parameter on the RTOS to TRUE, you should also refine your architecture diagram to include a hardware timer connected to the interrupt bus. In this case, be sure that the *UseHWTimer* boolean on the hardware timer is also set to TRUE. You should set these two parameters consistently. It is recommended that you export these two parameters to the architecture diagram and set the parameter value once.

5 Performance Evaluation

Performance evaluation lets you analyze statistical data about the execution of your behavior on a specific architecture.

You can run two types of simulation in the VCC environment:

- Functional simulation lets you verify the integrity of your behavior design without the influence of architecture constraints or performance model delays.
- Performance simulation runs from the mapping diagram and lets you
  - Verify the performance of the behaviors on an architecture
  - Simulate processing and communication delays and resource contention through performance models

You can run simulations in three different ways:

- Interactive mode simulates in an active session and is used for debugging. Progress messages, error messages, and display object outputs are displayed as they occur.
- Background mode simulates in the background, allowing you to continue work in the active VCC session.
- Remote simulation mode lets you use a remote machine to process the simulation.

Simulating the Mapping Diagram

A performance simulation is run from your mapping diagram. It follows the same process as a functional simulation. This section provides details about setting up, initializing, running, and debugging a performance simulation.

Setting Up the Simulation

After opening a simulation session, you need to set the performance simulation option, make changes to the parameter values, set breakpoints for debugging, add probes to collect data for analysis, verify the simulation settings, and check the mapping configuration specified for this diagram.

- Open a new simulation session using your mapping diagram.
- Change parameter values

  To display the top-level design parameters, select the *Params* tab in the Simulation window. Make any parameter changes. These parameter values are valid for the simulation session only. The parameter values in the actual diagrams are not modified.

  Make sure that all performance properties on mapping connections are set to appropriate values. There are often no default values for these properties. Properties that are not set cause errors during the initialization phase of your simulation.

  Note: You only need to set performance properties. Implementation properties are not needed for simulation.

- Set breakpoints for debugging

  For debugging your mapping diagram, set breakpoints to halt the simulation, and use interactive mode to step through segments of your diagram. Refer to *VCC Help* for details on setting breakpoints in your behavior design.

- Set Probes and Display Objects

  In the behavior portion of your mapping diagram, you can set probes on ports, viewports, memory instances, and timer instances. Viewports on the models define the data to be collected and the presentation style of summaries and graphic presentations. Analysis data is collected and saved for only those viewports that have probes set for them. Refer to *VCC Help* for details about setting probes on behavior models.

  To compare the results of the functional simulation with the performance simulation, probe the same ports and viewports on behavior models as you probed in your functional simulation.

  Probes on architecture models provide a view of the processing activity. When you add a probe to an architecture primitive, you select the viewport from a list of available viewports on each of the services associated with the architecture primitive.

  Probes cannot be set for ports in an architecture diagram. However, probes can be set on viewports associated with any architectural services attached to those ports.

  You can also set display objects, such as displays, gauges, and charts, on viewports, to provide data while your simulation is in progress. Refer to *VCC Help* for details on setting probes and display objects on behavior models and architecture models.

- Check Simulation Settings

Simulation settings specify the type of simulation and various run options. Make sure that the simulation type is set to performance. Specify the end time for the simulation.

Once you have debugged your mapping diagram, you can use background mode for further simulation runs. In background mode, progress messages let you track the status of your simulation. Note that in background mode, any breakpoints and display objects are disabled.

By default, simulation results are saved as another view in the cell where your mapping diagram is located. You can specify the name of the results directory for each simulation and maintain results from several simulation runs for later comparison.

- Save the simulation session.

Initializing and Running the Simulation

1. In interactive mode, select *Analysis > Initialize* to start the simulation.

   During the initialization phase, a netlist is created that represents the combination of the mapped behavior, architecture and performance models specified in your mapping diagram. The netlist created in this phase provides the input for your simulation.

   The Simulator Output area displays progress and error messages.

2. When the initialization phase completes, select *Analysis > Continue* to simulate the mapping diagram.

   The simulation run time is updated in the status bar at the bottom right side of the simulation window.

   If no breakpoints are set, the simulation proceeds until the end time specified in your simulation settings. A message confirms that the simulation completed successfully.

   If you set breakpoints for debugging, simulation progresses to the first breakpoint encountered. In your diagram, the breakpoint where simulation paused is highlighted in red. Select *Analysis > Continue* to resume processing to the next breakpoint. Refer to *VCC Help* for detailed information about using breakpoints to step through a simulation.

Debugging Simulation Problems

Most errors occur during the initialization phase of the simulation. The following list describes common problems and suggested corrections.

- Null value for a parameter

Usually, a link parameter generates this error. Link parameters often do not have default values and must be set explicitly. The context of the message identifies which parameter has no value specified.

Note: You can use the *Mapping Parameters* command, then sort by value to bring all the null values to the top of the list.

- View does not exist
    - The mapping configuration specifies the performance views used by the mapping process during initialization. Verify that the Performance Viewlist or Global Viewlist contain the appropriate view names and that they are in the proper sequence.
    - If a diagram has been copied from another location, the current netlist might not have the appropriate compiled simulation objects. Update these objects by using the *Analysis > Settings* command and setting the Compile option to Recompile Design.
- Out-of-date netlist

  Any changes you have made to parameter values or to your mapping are not reflected in the netlist. Regenerate the code in your mapping diagram.
- Syntax error

  After mapping has been completed, individual functional models and delay script models are compiled into a simulation model, and C code in additional files is parsed. Syntax errors are displayed in the output area and written to the `error.txt` file. To limit the display in the output window, right click in the output area and choose *Output Filters* to select the types of errors, if any, you want displayed.

  Syntax error messages provide the filename and line number where the error occurs. During the initialization phase, syntax errors are usually in your scripted delay performance models. (Behavior model syntax is parsed before functional simulation. Architecture model syntax is parsed when the code is generated.)

  A syntax error can generate other errors in subsequent processes, so check the first set of errors displayed in the output window. After the syntax has been corrected, regenerate the code and rerun the initialization process to see if other errors recur. If you double click on the error, the location of the error is highlighted in the mapping diagram.
- Inconsistent interface between views of a cell

  All views of a cell must have a consistent set of ports, parameters, memory, and timer references. Verify that the behavior model (and symbol) have consistent interfaces.
- Service declaration is not bound to a service definition

This error message is typically generated by a performance view of an architecture instance. The performance view should specify a service definition for each of the service declarations in the architecture symbol. The context of the error message identifies the service declaration that is not bound and the architecture instance that is missing the binding. To fix this error, double click on the error message to highlight the instance. Use the *Edit View* command on the highlighted architecture instance and check the services in the *Service Bindings* tab of the Properties dialog.

■ Unable to resolve a service binding

This error message is typically generated by a performance view of an architecture instance. The performance view specifies a list of service definitions. Each of these service definitions uses other service declarations. These other service declarations must be supplied by the architecture instance itself or by the architecture instances to which it is connected. The context of the error message identifies the service declaration that is required by a specific service definition on a specific architecture instance. To fix this error, double click on the error message to highlight the instance that requires this service. First verify that this service declaration is used by the service definition, then verify that the architecture instance or the architecture models to which the instance is connected supports this service declaration. You can use the *Service Declarations* tab on the Properties command to see the services supported by this instance and the connected instances.

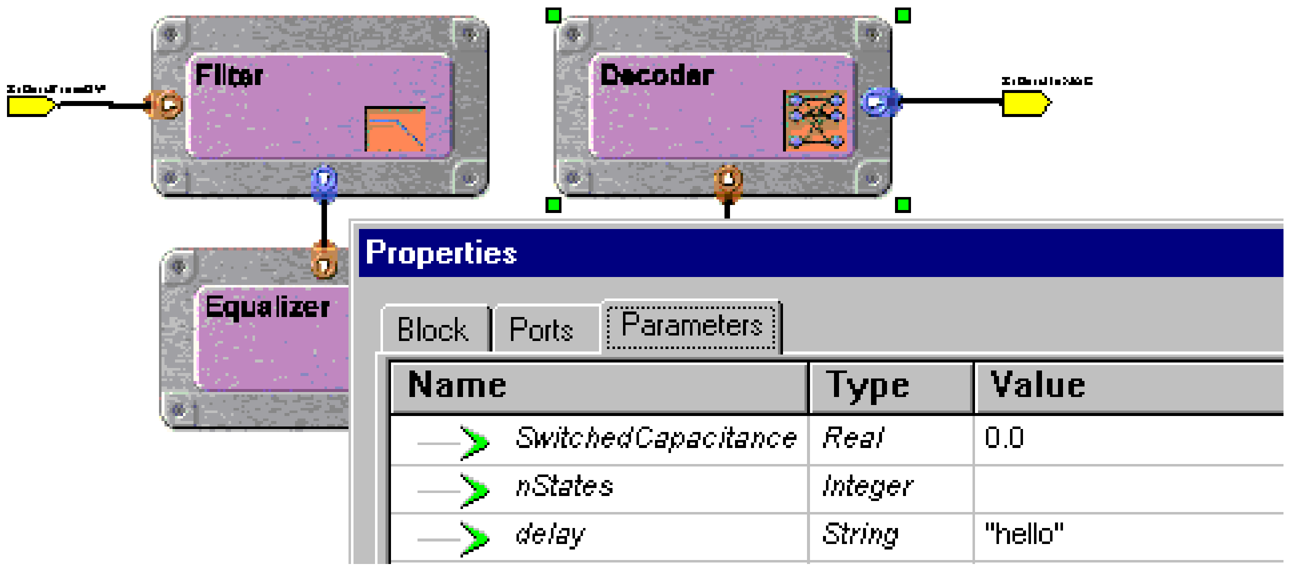
Exploring the Mapping Diagram

Once the mapping diagram has been debugged and you have analyzed your output and made corrections, you can run further simulations to sweep particular parameters with multiple values. In order to sweep parameters, you need to export parameters up to the mapping diagram, then set the parameter values in the simulation session.

Exporting Parameters

Behavior and architecture parameters can be exported to your mapping diagram so that you can modify or sweep parameter values at simulation time. Exporting parameters lets you explore variations in model characteristics without revising your original diagrams.

In the following figure, a behavior model (the Decoder) is selected, and the Parameters tab of the Properties dialog is displayed. You might want to export one of the parameters, for example, the *nStates* parameter, so that you can explore the impact of changing the number of states.

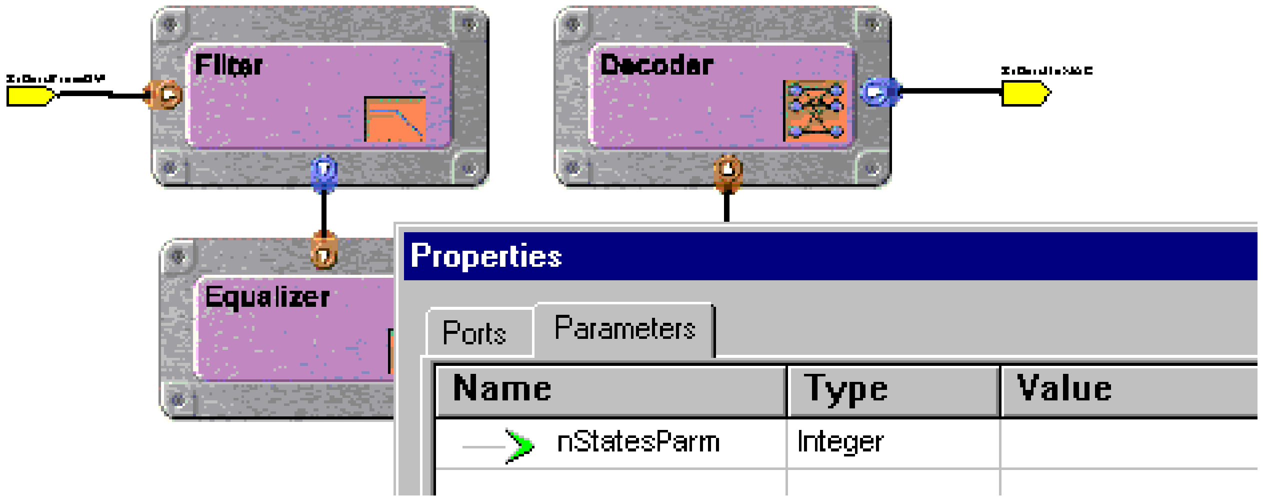


Name	Type	Value
SwitchedCapacitance	*Real*	0.0
nStates	*Integer*	
delay	*String*	"hello"

Existing parameters for the instantiated model are displayed here. If the value of a parameter can be modified, you can set the value to a constant, an expression, or a parameter name.

To set the value to a parameter, you first need to create a parameter in the next level of hierarchy. To create the export parameter, select the Properties menu from the background of the current display to access the parameters tab of the higher level of hierarchy. For example, right click in the background area of the behavior diagram in which the Decoder is instantiated.

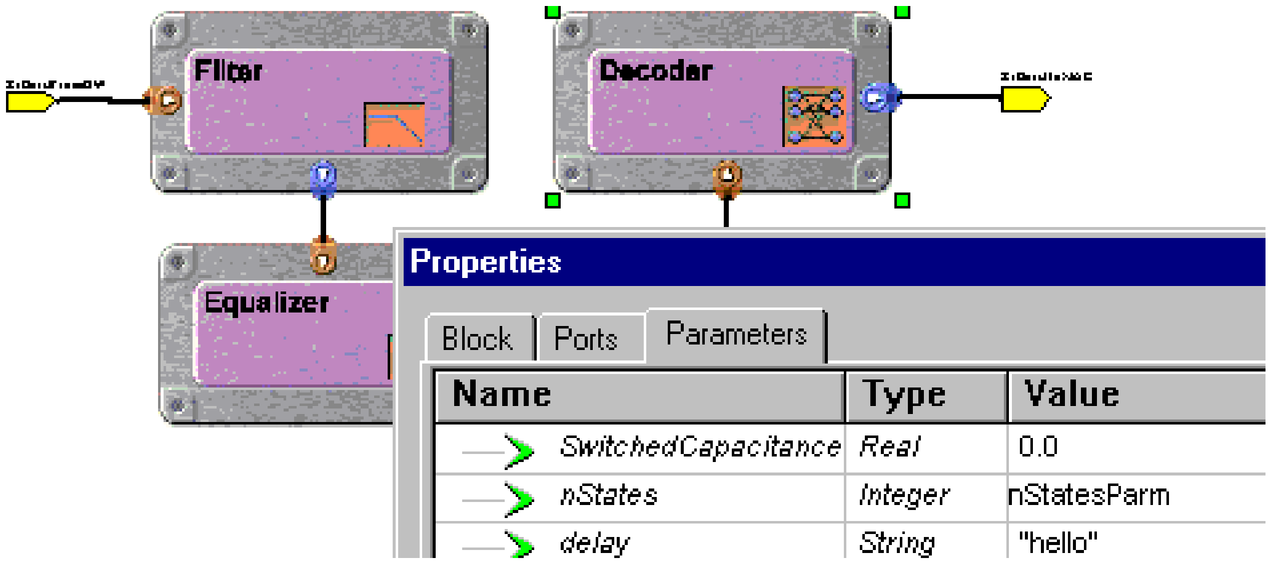
In the following figure, a parameter named `nStatesParm` of type integer is created in the behavior diagram containing the Decoder.



Name	Type	Value
nStatesParm	Integer	

You then set the value of the lower-level parameter to the new diagram parameter name.

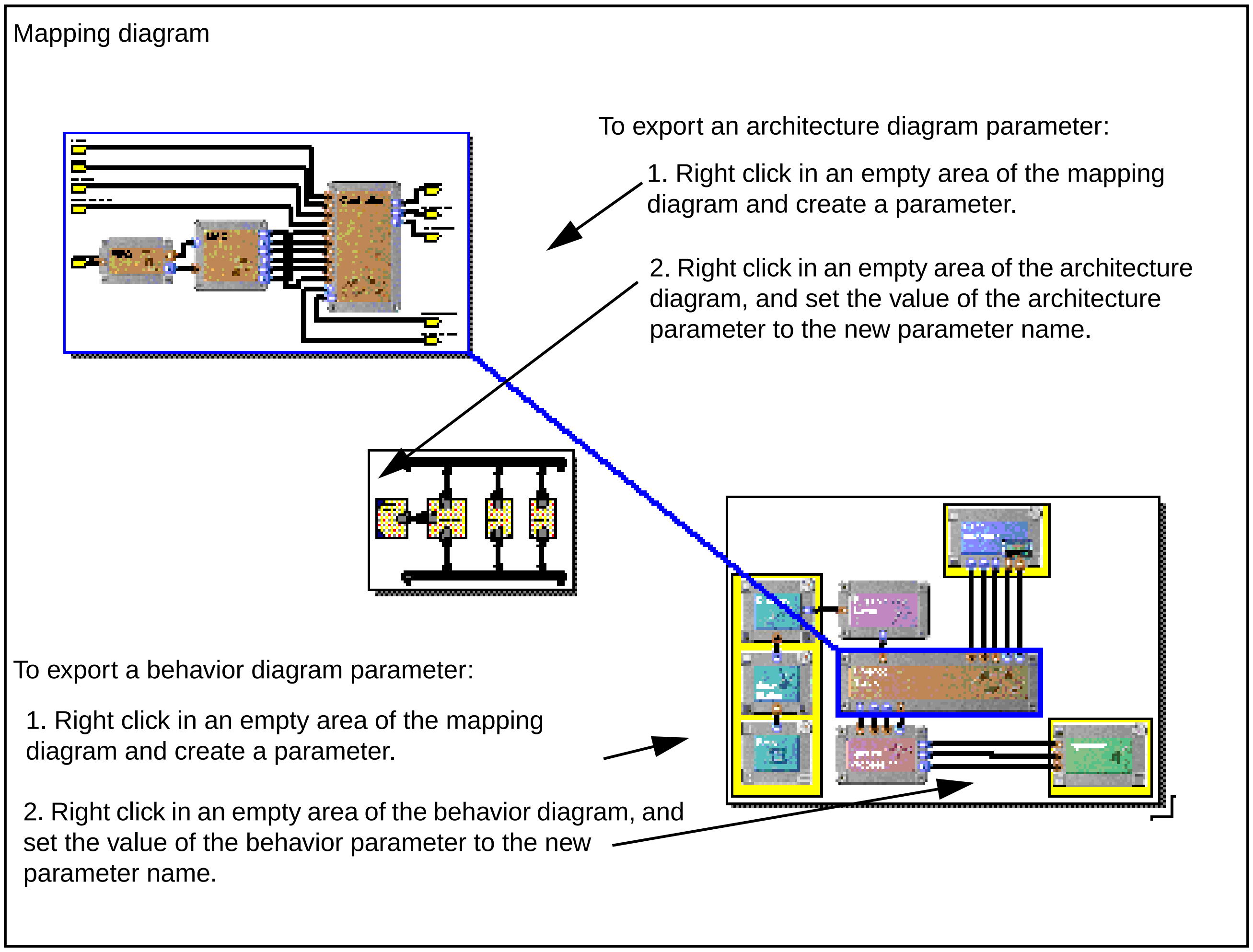
In this example, you would return to the Properties display of the Decoder behavior model and set the value of the `nstates` parameter to the newly created behavior diagram parameter `nStatesParm`.



Name	Type	Value
SwitchedCapacitance	*Real*	0.0
nStates	*Integer*	nStatesParm
delay	*String*	"hello"

Because all views of a cell must have matching interfaces, you need to add the `nStatesParm` parameter to the symbol view of the behavior diagram.

To export the parameter to the mapping diagram, you need to export it up through each level of the hierarchy to the top-level behavior diagram. The following figure of a hypothetical mapping diagram illustrates how to export parameters from the behavior and architecture diagrams to a mapping diagram.


Mapping diagram

To export an architecture diagram parameter:

1. Right click in an empty area of the mapping diagram and create a parameter.
2. Right click in an empty area of the architecture diagram, and set the value of the architecture parameter to the new parameter name.

To export a behavior diagram parameter:

1. Right click in an empty area of the mapping diagram and create a parameter.
2. Right click in an empty area of the behavior diagram, and set the value of the behavior parameter to the new parameter name.

All mapping diagram parameters are automatically displayed as simulation parameters during a performance simulation. By using simulation parameters, you can modify or sweep parameter values for different simulation runs without changing your original diagrams.

Sweeping Parameters

You can run simulations with different values for parameters by specifying sweep values for parameters in the simulation session. You use the Expression Builder to change the current value of a parameter to sweep across a list of values. For example, you might set the `nStatesParam` parameter to the values of 5 - 8. This will generate 4 simulations, with the `nStatesParam` set to 5 for the first iteration, 6 for the second iteration, and so on. Refer to

VCC Help for details on accessing and using the Expression Builder for specifying sweep values.

In the simulation settings form, select All Iterations. With this selection, the simulation automatically runs in background mode. You can also run the sweep simulation on a remote server. The simulator graphs the progress of each sweep. Refer to *VCC Help* for details on running in background mode or on a remote server.

Analyzing Simulation Results

How you analyze a performance simulation depends on the goal of the simulation, your mapping diagram, and any problems you are encountering.

Scheduling, prioritizing, and delaying functions introduce timing consequences into your design. Probing the appropriate viewports in your architecture models can capture data that identifies inadequacies in prioritizing functions and processor or bus speeds.

Use the Visualize analysis tool to build summary lists, Gantt charts, and 2D charts from your results file data. Refer to *Visualize Help* for details about displaying results.

Using Model Viewports

Viewports provide access to the internal state variables of a behavior primitive, behavior memory, or architecture service. Probes can be attached to these viewports to save data for later analysis. The statistics and data provided by viewports can be displayed in various formats through the Visualize tool.

This section provides details about the viewports associated with bus and scheduler models provided in the VCC environment. For details about viewports for other models, check the library documentation for your model libraries.

Probes on Services to Check Communication

You can attach probes to viewports of services associated with an architecture instance to analyze the activity of the architecture instance. Refer to the *VCC Library Reference* for descriptions of the viewports available on services in the VCC libraries. The following sections provide examples of viewports on buses and schedulers provided in the VCC libraries.

Bus Activity

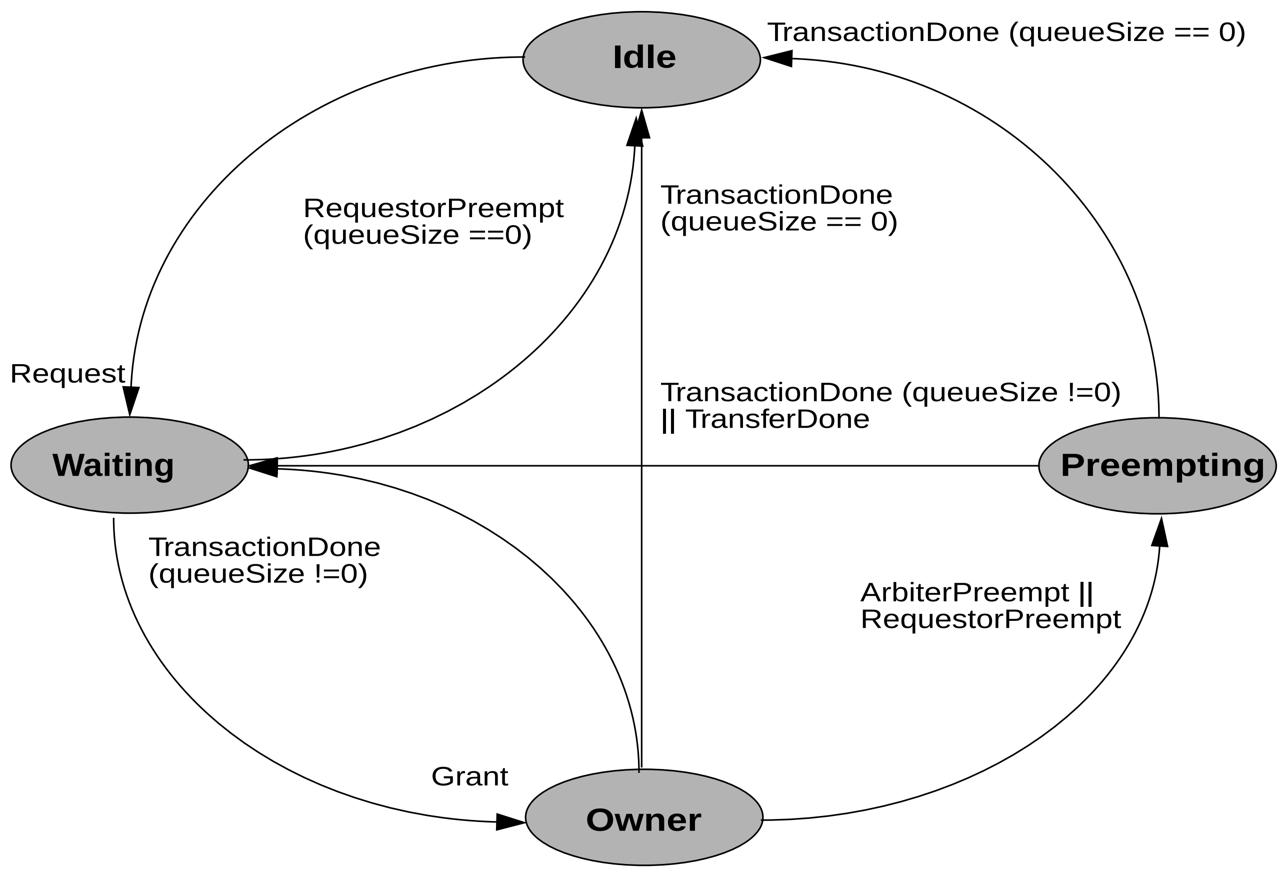
You can check the overall utilization of the FCFSBus in the VCC_Bus library by adding a probe to the *SummaryStats* viewport on the *busArbiter* service of the bus. You can retrieve overall utilization statistics or identify queue overruns by reviewing the results obtained from this viewport. From this data, you can determine if the bus speed is adequate, if a particular data path is controlling bus activity, or if token transfers are being lost.

To continue this example, you can attach a BusModelStatsProbe from the VCC_Test library to the *SummaryStats* viewport. This probe generates two tables of data that can be analyzed in a Visualize table chart. The first table displays activity for the entire bus, and the second table displays activity for each communication wire mapped to the bus. Refer to the "VCC_Architecture Services Library" chapter in the *VCC Library Reference Guide* for more information about BusModelSummaryStats.

Only one transaction per artifact (only one message between a particular source and destination) should be active at any time. If a source is sending transactions faster than transactions are arriving at a destination, the statistics will be inaccurate.

To analyze the activity of a particular bus master, you can probe the *busAdapter* service on your bus master. For example, the *busAdapter* service of the SimpleCPU in the VCC_CPU library has two viewports: summary statistics and Gantt chart.

You can attach a StatsCollectorProbe to the *summaryStats* viewport to collect the statistics of the bus transfers. You can also attach the FCFSBusAdapterGanttProbe from the VCC_Test library on the GanttChart viewport to see the timeline progression of the bus requests. The following figure shows the state model for the bus adapter:


Idle
Waiting
Preempting
Owner
TransactionDone (queueSize == 0)
RequestorPreempt (queueSize ==0)
TransactionDone (queueSize == 0)
Request
TransactionDone (queueSize !=0) || TransferDone
TransactionDone (queueSize !=0)
ArbiterPreempt || RequestorPreempt
Grant

Scheduler Activity

You can also analyze statistics on the activity of a scheduler. From this data, you can determine the efficiency in processing tasks, excessive suspensions or aborts of tasks, and excessive wait times.

For example, you can probe the *schedInit* service of the MultipleServiceRTOS in the VCC_RTOS library. This service has two viewports, SchedulerSummaryStats and GanttChart. You can add a StatsCollectorProbe to the summary statistics viewports to analyze the overall scheduler activity or the activity on each particular behavior model mapped to the RTOS.

The following table describes the different fields produced by the statistics viewports on this scheduler:

Scheduler Statistic	Description
name	Name of scheduler or task.
priority	Priority of a scheduler or a task.
number_activations	Number of times that a scheduler or task is activated.
number_deactivations	Number of times a scheduler is deactivated.
number_starts	Number of times a task or a scheduler enters the running state. A task or a scheduler can be in a running or suspended state during its reaction.
number_finishes	Number of times a task or scheduler finishes reaction.
number_suspends	Number of times a task or a scheduler enters a suspended state. A task or a scheduler can be in a running or suspended state during its reaction.
number_resumes	Number of times a task or a scheduler resumes running after being suspended.
mean_activation_time	Mean time a task or a scheduler spends waiting to run. More specifically, this is the mean time between a task or a scheduler's activation and deactivation.
mean_reaction_time	Mean time a task or a scheduler spends reacting. A task or a scheduler can be in a running state or a suspended state during its reaction. This does not include the start and finish overhead of the parent scheduler.
mean_run_time	Mean time a task or a scheduler spends in the run state. A task or a scheduler can be in a running or a suspended state during its reaction. This does not include the time spent in the suspended state during a reaction.
mean_utilization	Percentage of the total time that a task or scheduler spends in the run state.

You can also add a SchedulerGanttProbe to the GanttChart viewport to see the activations, deactivations, and the beginnings and ends of the reactions of each task assigned to the selected scheduler.

Simulation Messages

Simulation details, errors, and warnings are displayed in the output area of the simulation window. With the *Output > Filter* command, you can specify which category of messages you want displayed.

`debug`	Contains information about the progress of the simulation. It is a more verbose version of `info` and can be useful in tracking a problem.
`info`	Provides a textual description of the simulation process and outcome. It includes such information as the scheduler tree and scheduler statistics that can be useful in analyzing results. Messages printed from within whitebox C and blackbox C++ models are included in this category.
`warn`	Identifies warnings issued during simulation. Warnings do not have to be corrected to proceed with the simulation. You should review the problems noted to see if they affect your results.
`error`	Identifies more severe errors that must be corrected for the simulation to run successfully. For debugging, the file also provides the simulation time when the error occurred and the location of the problem.

This information is also written to files in the results view (`debug`*`N`*`.txt`, `info`*`N`*`.txt`, `warn`*`N`*`.txt`, and `error`*`N`*`.txt`). In these filminess, *`N`* represents the sweep number. For a single simulation run, the number is 0. For a sweep simulation, separate files are created for each sweep, and *`N`* represents the iteration number. You can access these files with a text editor from the results directory of your library cellview.

Comparing Functional and Performance Simulation Results

If one of the goals of your simulation is to maintain consistent results with your functional simulation, you can compare the output from your performance simulation to the output from your previous functional simulation. To make this comparison, set probes on the same inputs and outputs used in your functional simulation.

If results are not consistent, the delays incurred in the performance simulation might be changing the sequence of reactions or ports might be receiving input faster than the reaction can be activated. Compare results only at meaningful points for your design. Use the Visualize tool to review the processing details collected by your probes. You can also set display objects to watch values change while the simulation is in progress.

Analyzing Testbench Results

A watchdog timer in the testbench can help assess whether reactions were processed in a reasonable length of time. By setting up your own analysis mechanism in the testbench, you can collect statistics for your diagram. By probing results of input and output ports in the testbench, for example, you can calculate latency of each output received and graph the results. Refer to *Visualize Help* for details about displaying charts.

Analyzing Table Charts

You can quickly scan a table chart for expected results, errors, and inconsistencies. Depending on the goal of your simulation trials, certain fields of data or statistics should be of greater interest than others, and you can view a series of results by scanning a column in the display. You can also use table charts to provide summaries of statistics or utilization data.

For example, with bus models from the VCC_Bus library, you can retrieve overall utilization statistics or identify queuing overruns by reviewing the results in the Bus Model Summary Statistics chart. From this data, you can determine if the bus speed is adequate, if a particular data path is controlling bus activity, or if token transfers are being lost.

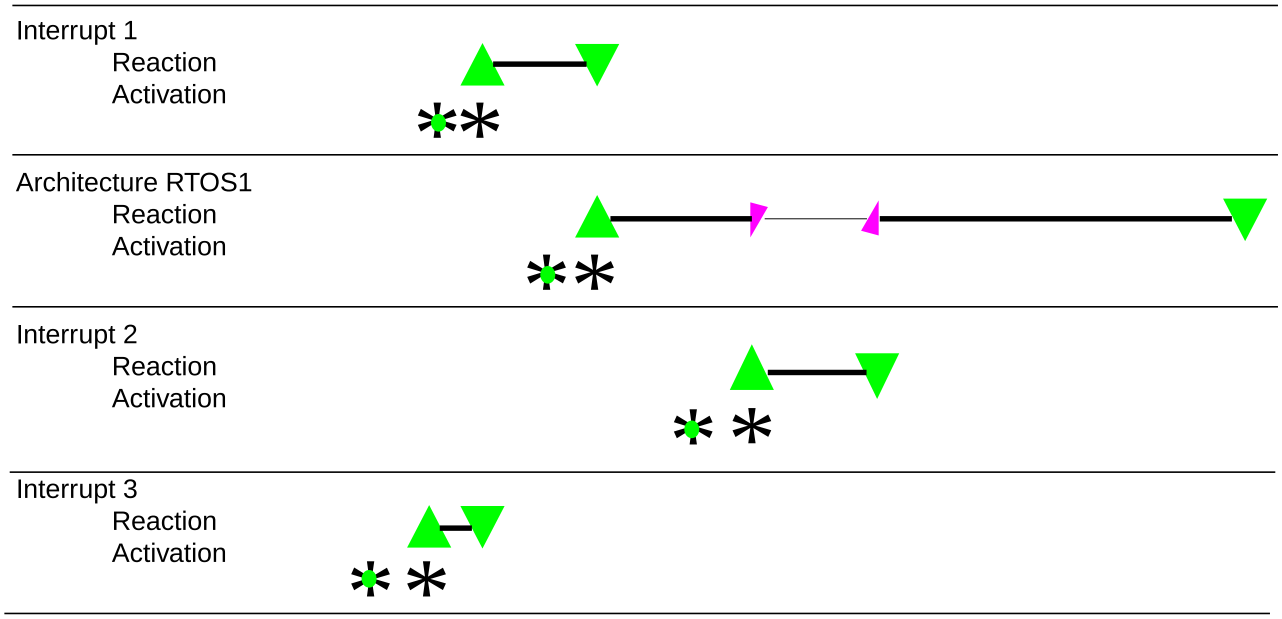
If you need more detail about a specific data path, you can review the Bus Artifact Summary Statistics chart for that data path. This chart provides details on the number of attempted and successful transfers, mean transfer rates, and a ratio of utilization of this data path versus the entire bus. From this data, you can determine if the bus can service this data path efficiently or if changing task priorities or remapping might improve transfer rates or reduce transfer failures.

Analyzing Gantt Charts

Gantt charts provide an in-depth analysis of the activation and reaction activity of a model. Gantt charts clearly represent the scheduling and processing sequences of multiple tasks. Table charts and Gantt charts used in combination let you identify a problem area from summary data and then zoom in on a display of the precise activities that are involved.

For example, the Gantt Chart viewport associated with the VCC scheduler and RTOS models in the VCC_RTOS library provides a graphic presentation of behavior activation, reaction, suspension, and completion for a particular scheduler.

The following figure represents a compressed illustration of the type of output this viewport provides. (Note that the color of the interrupt indicators in this and other figures has been changed for readability in this document.)

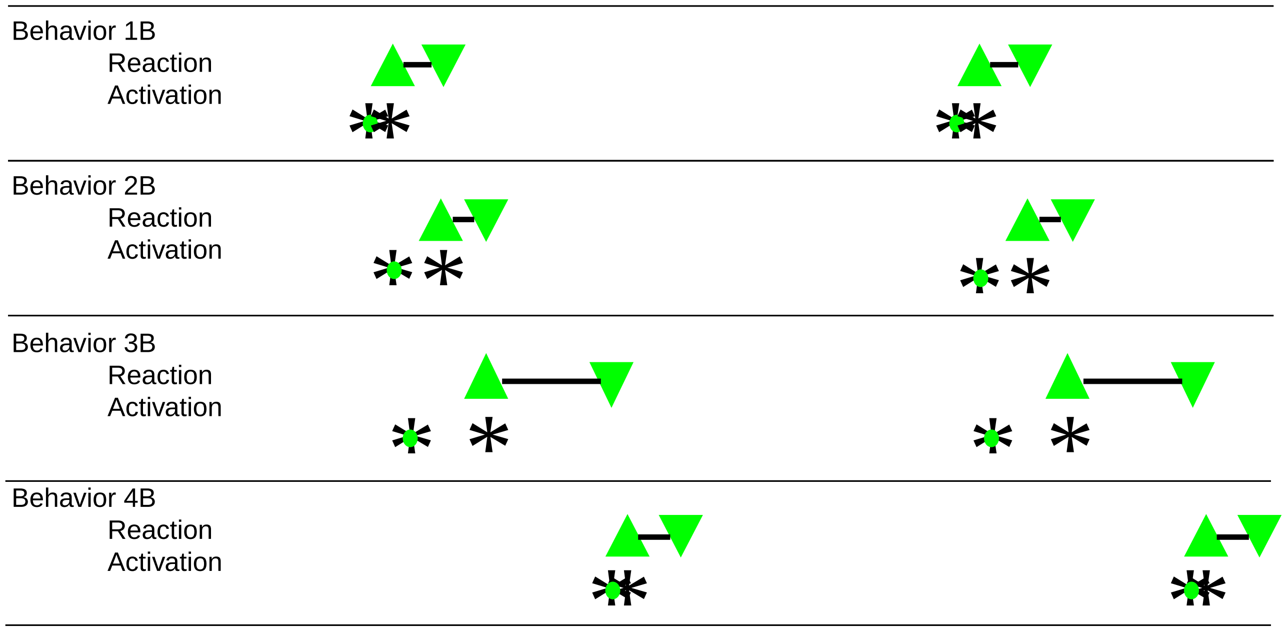


The activation row shows the activation and deactivation of each task. The reaction row identifies when each reaction starts and ends.

This output segment depicts the type of activity you see on a hardware scheduler that has three interrupt service routines and an RTOS assigned to it. The three service routines have higher priorities than the RTOS.

The activity shows that Interrupt 3 and Interrupt 1 have been processed. The reaction of the RTOS is suspended to accommodate processing Interrupt 2. Refer to *Visualize Help* for details about zooming in and maneuvering within the display window to view output at various levels of detail.

The RTOS in this example is a nested scheduler and is considered another task running on the hardware scheduler. For a similar display of the RTOS activity, you set a Gantt probe on the RTOS and chart activations and preemptions of the tasks assigned to it, as shown in the following figure.



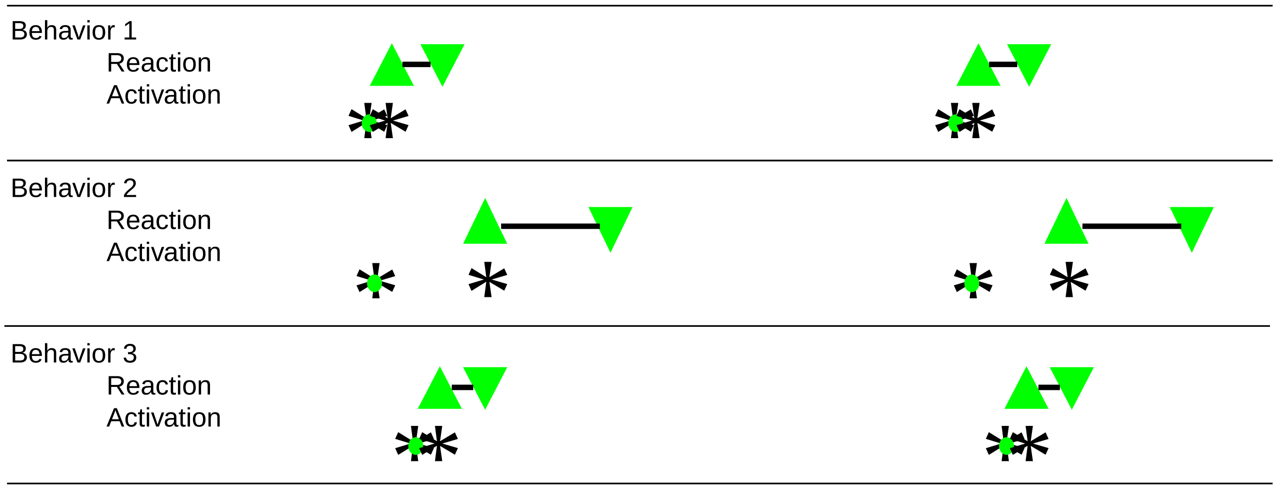
Comparing the timelines displayed at the top of two Gantt charts lets you coordinate multiple displays. For example, the display of a primary scheduler can be synchronized with the display of a nested scheduler to analyze the interactions between the two schedulers.

The following figures represent events that can be identified from a Gantt chart display.

- Inappropriate task priority

  The Gantt chart of scheduler activity clearly identifies the sequence in which behaviors are activated and when a reaction begins. From this information, you can identify behaviors that are processed in the wrong sequence.

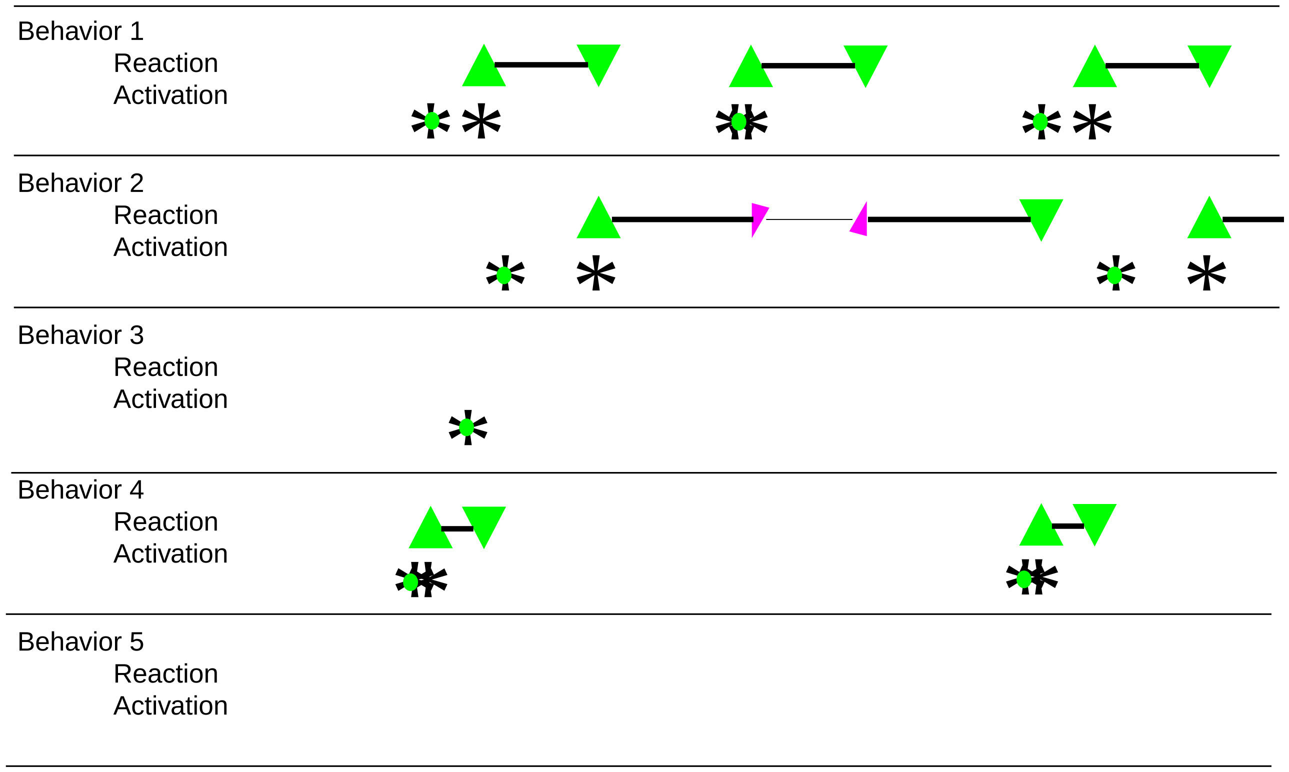
For example, in the following figure, Behavior 2 is activated while Behavior 1 is in progress, but Behavior 2 does not run until Behavior 3 has completed.



Depending on your design, this might indicate that the priorities of Behavior 2 and Behavior 3 are set inappropriately, and that these tasks are not being run in the appropriate sequence. If so, adjust priorities or use a first-come-first-served scheduler to maintain the proper sequence.

- Task starvation

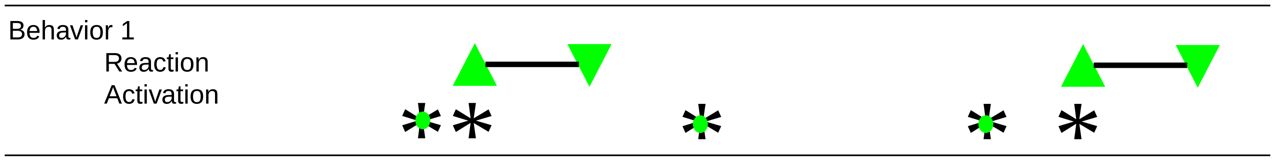
Throughout the processing cycle represented in the following figure, Behavior 5 is never activated. Also, Behavior 3 is activated but its reaction never begins. The relationship of these two events might be significant.



For example, if Behavior 3 feeds input to Behavior 5, the cause of the task starvation is that Behavior 3 has not processed the input for Behavior 5. Going further, the reason for this situation might be that the priority for Behavior 3 is set too low or that the CPU speed is too slow to allow reasonable throughput of higher priority tasks.

- Overwritten ports

  In the following example, three activations are received. The first reaction begins and uses the data from the first activation. Two more activations occur before the second reaction begins. When a task begins reacting, it reacts using the latest input. In this example, if the behavior has only one input port, the input from the second activation is being overwritten and is lost.



Solutions might be to change the priority of Behavior 1 (or of the other tasks assigned to the scheduler), to use a faster processor, or, if multiple processors are available, to change the mapping of various behaviors to balance the processing load.

If a behavior model has more than one input port, the second and third activations could be a result of data being received on these other ports. An activation occurs when one port receives data. In this case, multiple activations might not mean that data is being overwritten. You need to determine what ports are being activated and whether this is an acceptable situation.

6

Mapping Refinement

Once you have identified an architecture that meets your design needs, you need to refine communication and data transfer to accommodate the architecture. One example of a refinement is a data refinement.

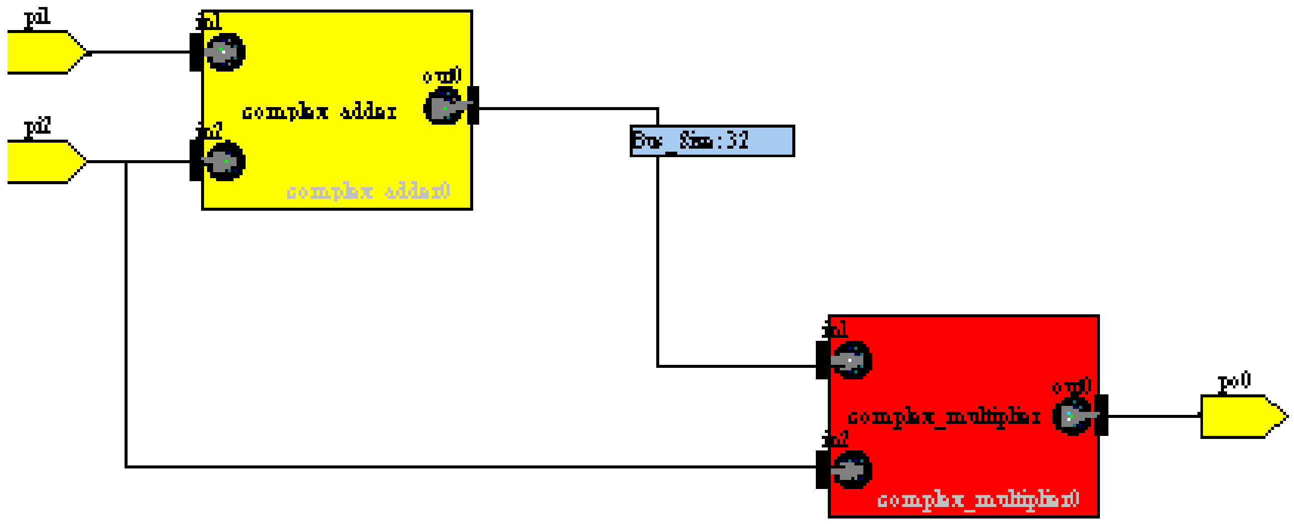
You add architecture-specific behavior to your mapping diagram through the refinement process. By doing the refinement in the mapping view, you preserve the original behavior model and keep it independent from the influence of a specific architecture. This way, you can use the behavior model in other designs.

This chapter demonstrates the refinement process using the example of a protocol down converter.

Protocol Down Converter Refinement

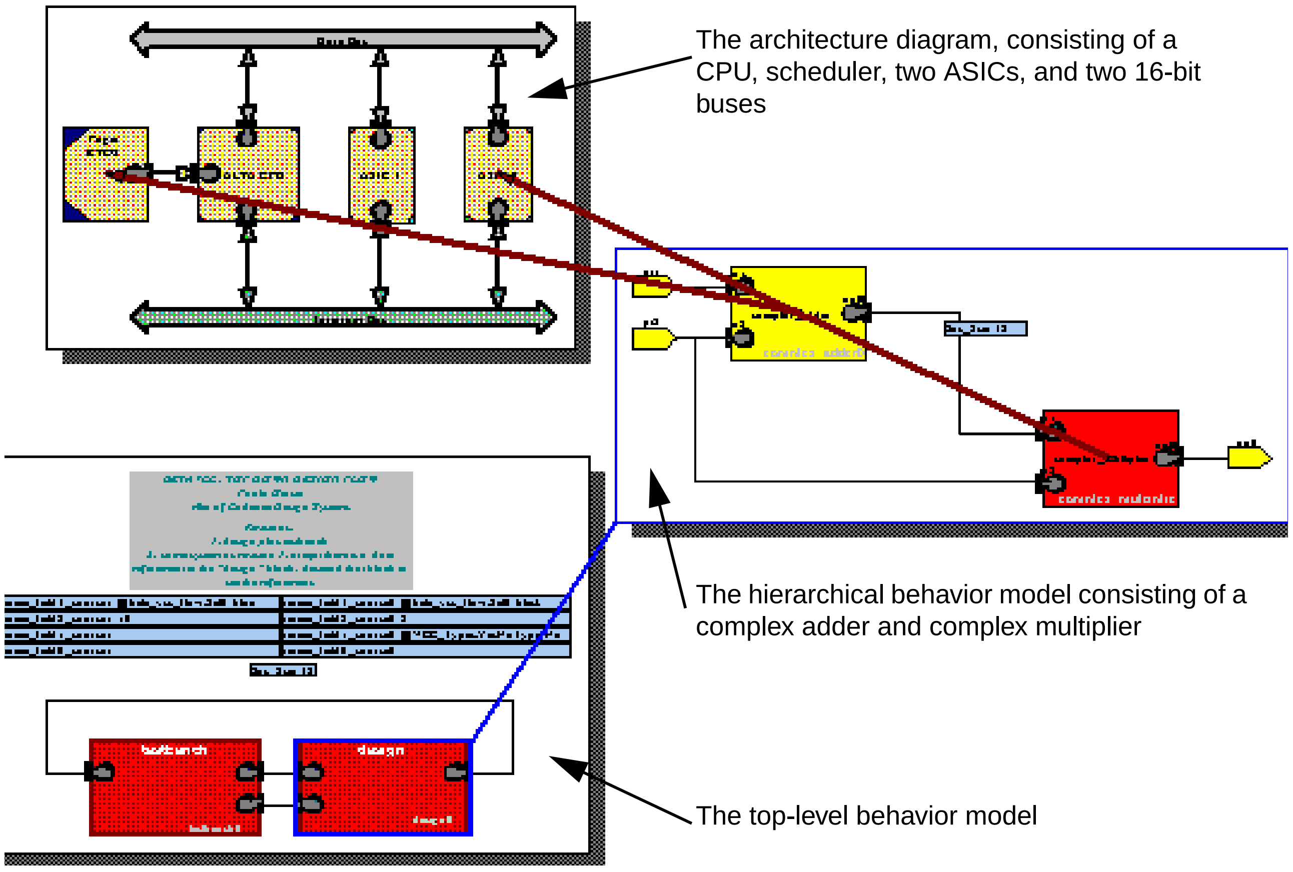
The refinement for the protocol down converter converts a high-level, abstract data structure to a set of tokens representing actual data on the bus.

The original behavior diagram processes input through a complex adder and multiplier, as shown in the following figure.



The two behaviors are implemented on different architecture models. The data, therefore, must be transferred from the output port of the adder to the input port of the multiplier across a bus.

The following figure shows the mapping diagram of the original behavior design implemented on the described architecture diagram.


The architecture diagram, consisting of a CPU, scheduler, two ASICs, and two 16-bit buses
The hierarchical behavior model consisting of a complex adder and complex multiplier
The top-level behavior model
complex_adder
complex_multiplier
testbench
design

The size of the data bus in the selected architecture is smaller than the token being transferred. A refinement is needed to convert the token to multiple 16-bit packets for transmission across the bus. The packets then need to be converted back to the original token.

Creating a Refinement Model

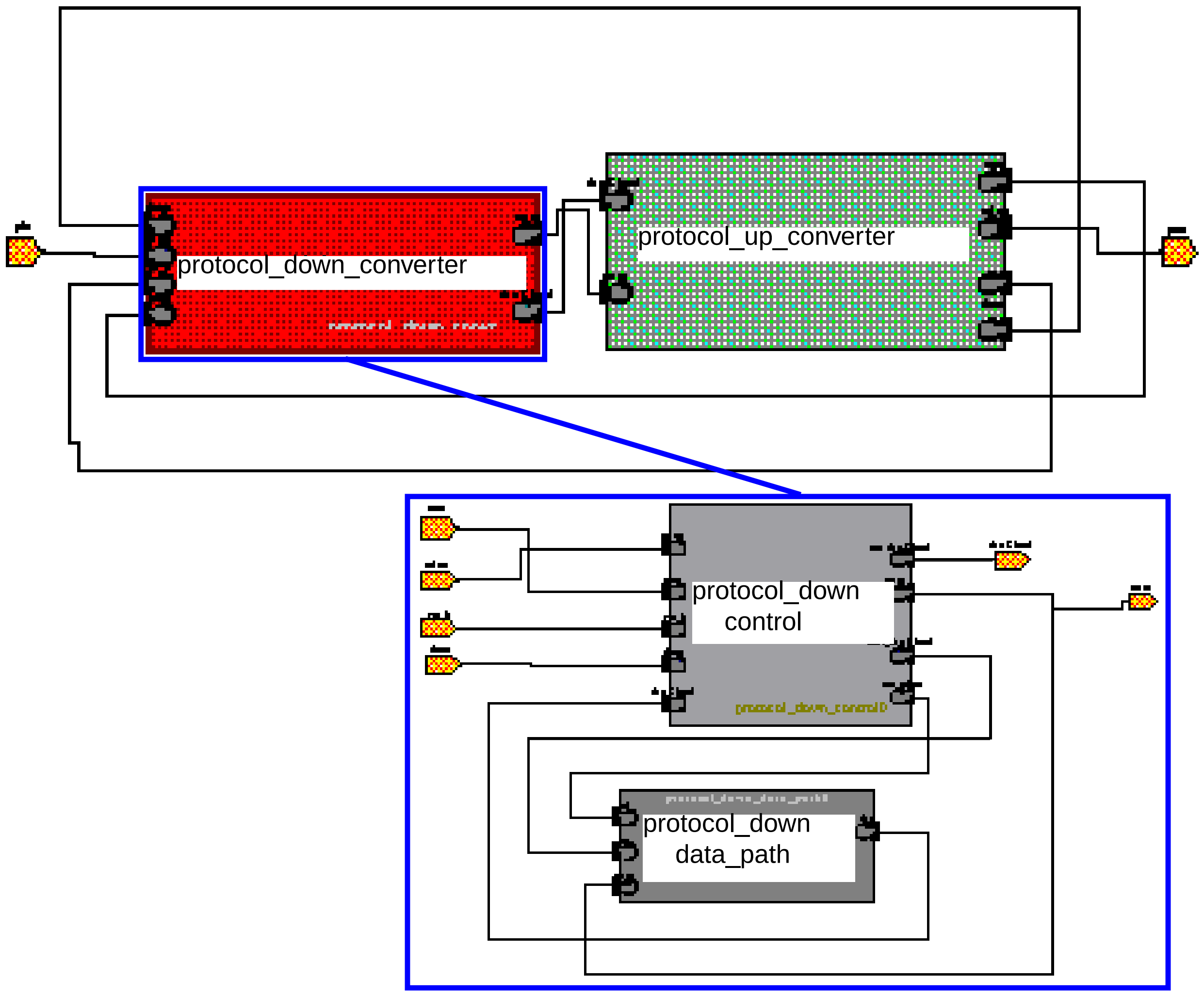
In VCC, you can add the additional behavior required by the mapping using the refinement options available through the Mapping Diagram Editor. This approach keeps the behavior diagram free of architectural influences.

You create the refinement behavior using the behavior diagram editor. You can instantiate one or more behavior models in the behavior diagram. The refinement must have a single input port and a single output port. The datatype of these ports must match the datatype of the port being refined. Many refinements involve control-dominated processing for which the State

Transition Diagram (STD) Editor is well-suited. You can also develop or import these behavior models using any of the methods available in VCC. For information about using the State Transition Diagram Editor, refer to VCC Help.

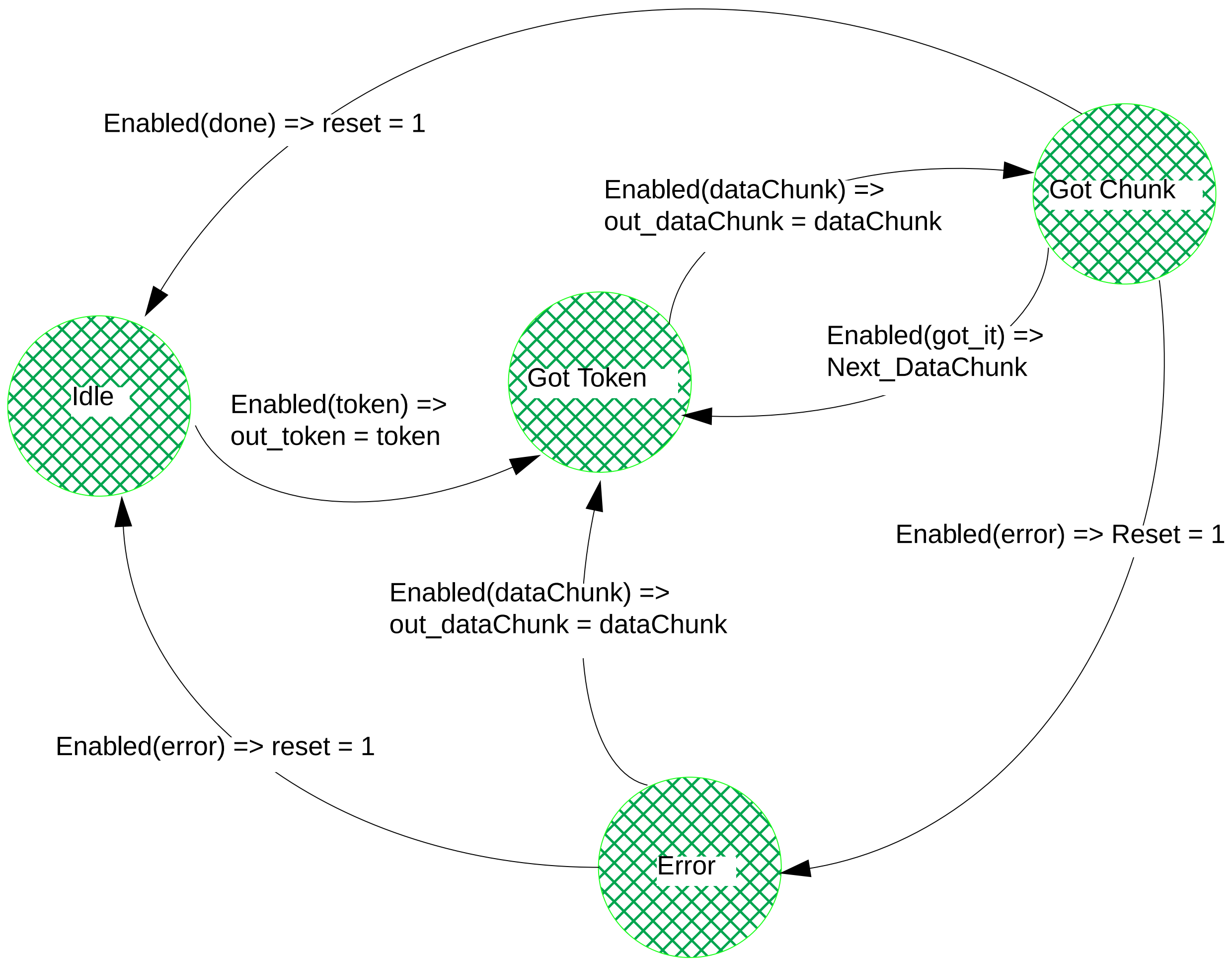
Performance models are needed for each of the behaviors in the refinement cellview. You create the performance model, as with other behavior models, by choosing the type of performance model and setting the appropriate parameters. For more information, see Chapter 2, "Performance Models."

The following figure shows the refinement of the protocol down converter.



The `protocol_down_converter` is a hierarchical behavior model. It consists of the `protocol_down_control` and the `protocol_down_data_path` primitives. The `protocol_down_control` model is implemented as a State Transition Diagram. The `protocol_down_data_path` and the `protocol_up_converter` models are implemented as Blackbox C++ behavior models.

The following figure shows the STD diagram for `protocol_down_control`. It provides an error routine that retries a packet transfer if an out-of-sequence data packet is received.



In this STD diagram, four states control and verify the down conversion:

Got Token	The Got Token state is activated when a token is received. The token is passed to the `protocol_down_data_path` function where the token is broken down into packets. Each packet is numbered so that its sequence can be monitored. As Got Token receives each packet from the data path, it sends the packet to the `protocol_up_converter` model and passes control to the Got Chunk state.

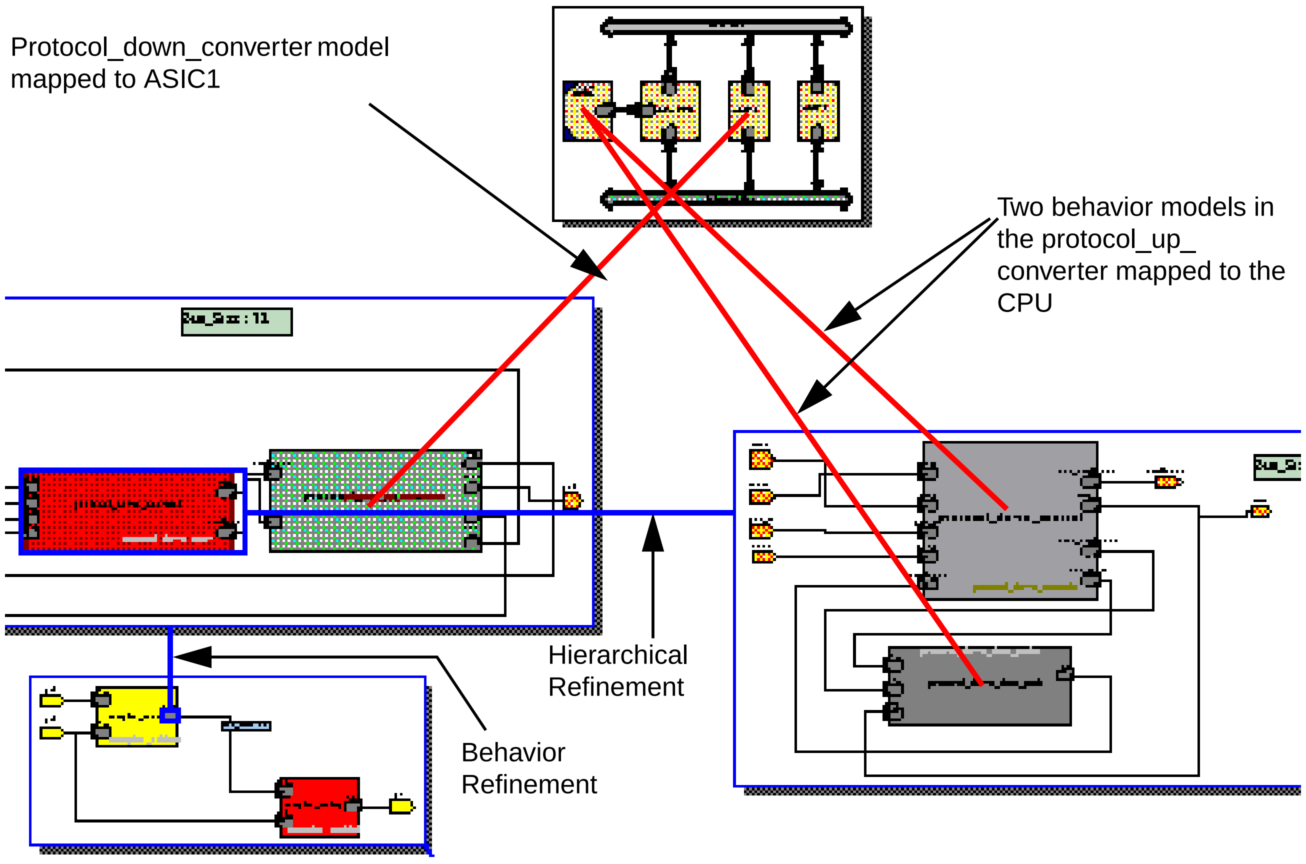
Got Chunk	The Got Chunk state is activated when a data packet from the data path is sent to the `protocol_up_converter`. The `protocol_up_converter` acknowledges receipt of the packet and confirms that it is in the proper sequence. If the `protocol_up_converter` confirms that the packet has been received and is in the correct sequence, control returns to the Got Token state to await the next packet from the data path. If an error message is received from the `protocol_up_converter`, control passes to the Error state.
Error	The Error state determines if this is the first retry for the packet. If so, control returns to the Got Token state for a retransmission of the same packet. If the retransmission is not successful, reconstructing the packet is considered impossible, and control passes to the Idle state.
Idle	The Idle state is activated when the token is reconstructed correctly and the `protocol_up_converter` sends a “done” signal, or if the Error state has determined that a token cannot be reconstructed. If another token is ready to be processed, control passes to the Got Token state to begin transferring a new token. If not, control remains in the Idle state until a new token is received.

Instantiating and Mapping a Refinement

To instantiate a behavior refinement, select the input port or output port of the communication wire where the refinement is to be placed. (To avoid problems with fan-out, use the output port of the communication wire.) Use the *Mapping > Mapping Refinement* command, and select your behavior refinement cellview. Then place the cellview in your mapping diagram.

Each behavior model in the refinement behavior diagram is then mapped to an appropriate architecture model, as shown in the following figure. If the refinement is hierarchical, use the

Mappping > Hierarchical Mapping command to expand the refinement, then map the models within it.


Protocol_down_converter model mapped to ASIC1
Two behavior models in the protocol_up_ converter mapped to the CPU
Hierarchical Refinement
Behavior Refinement
Original behavior

The `protocol_down_data_path` primitive and the `protocol_down_control` primitive are mapped to the CPU. The `protocol_up_converter` refinement model is mapped to ASIC1.

You need to bind a performance view for each of these mapping links. Refer to Chapter 4, “Analyzing Behavior Delay,” for more information. You also add mapping links to buses or patterns to model communication.

Simulating and Analyzing a Refinement

You can use breakpoints for debugging the refinement, select viewports for collecting data, and analyze results using Visualize. If you need to debug your refinement, you can run a functional simulation to test the functionality of the refinement behavior models.

Specifying a Sub-Configuration for a Refinement

When refining your design, you can insert new behaviors that further refine the communication or data transfer process. Refinement behaviors might require different bindings than used in the original mapping diagram. To use a different configuration for a refinement, add a sub-configuration to define the binding sequence.

Enter the new configuration cellview (the *lib.cell:view* of the configuration for the refinement behavior hierarchy) to use for this refinement.

Note: A sub-configuration applies only to the hierarchy below this occurrence. It does not apply to the occurrence where the configuration cellview is specified.

A

Pattern Descriptions

This appendix describes

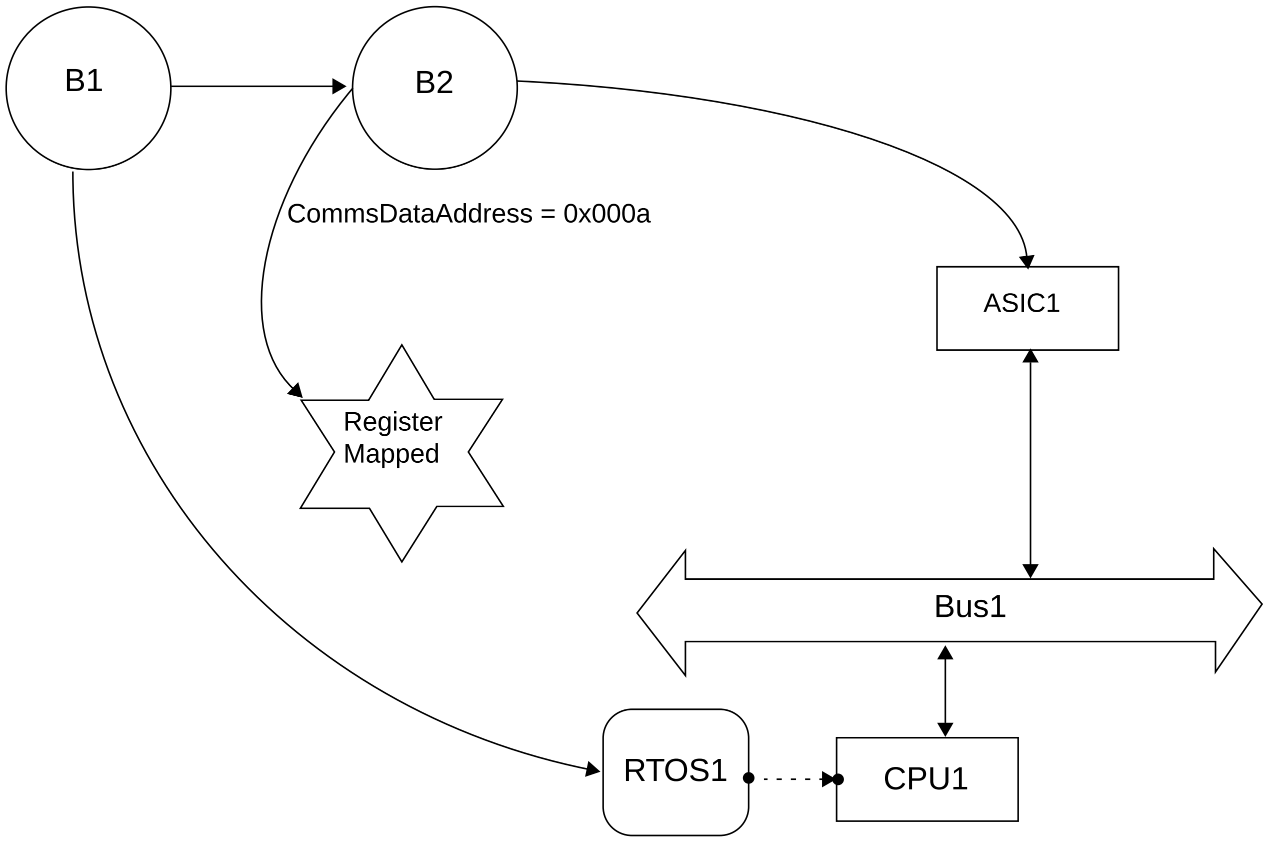
- Software-to-Hardware Communication
- Hardware-to-Hardware Communication
- Hardware-to-Software Communication
- Software-to-Software Communication

Software-to-Hardware Communication

You implement communication from software blocks to hardware blocks using the register mapped pattern or the shared memory pattern.

Using Register Mapped Pattern

The register mapped pattern is for transferring data from the CPU across the bus directly to a register on the ASIC resource.



The mapping of the behavior instances specifies RTOS1 for the source behavior and ASIC1 for the destination behavior. VCC uses the architecture topology to determine that the communication goes across the bus.

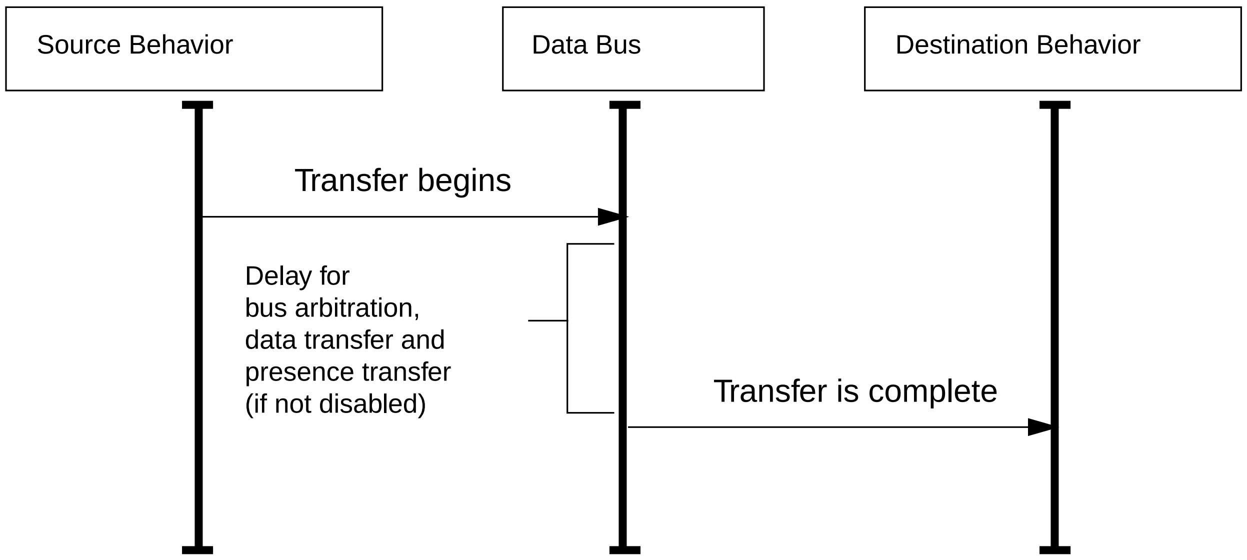
The register address is an offset of the starting address of the port connecting the ASIC to the bus. The register address is the value you set for the *CommsDataAddress* parameter on the mapping connection.

Set the following parameter on the instance of the pattern.

Parameter	Data Type	Default Value
CommsDataAddress	@VCC_Types.DataAddress	floating

The default value of `floating` for the CommsDataAddress means that if you do not specify a value, the Address Allocator assigns one for you.

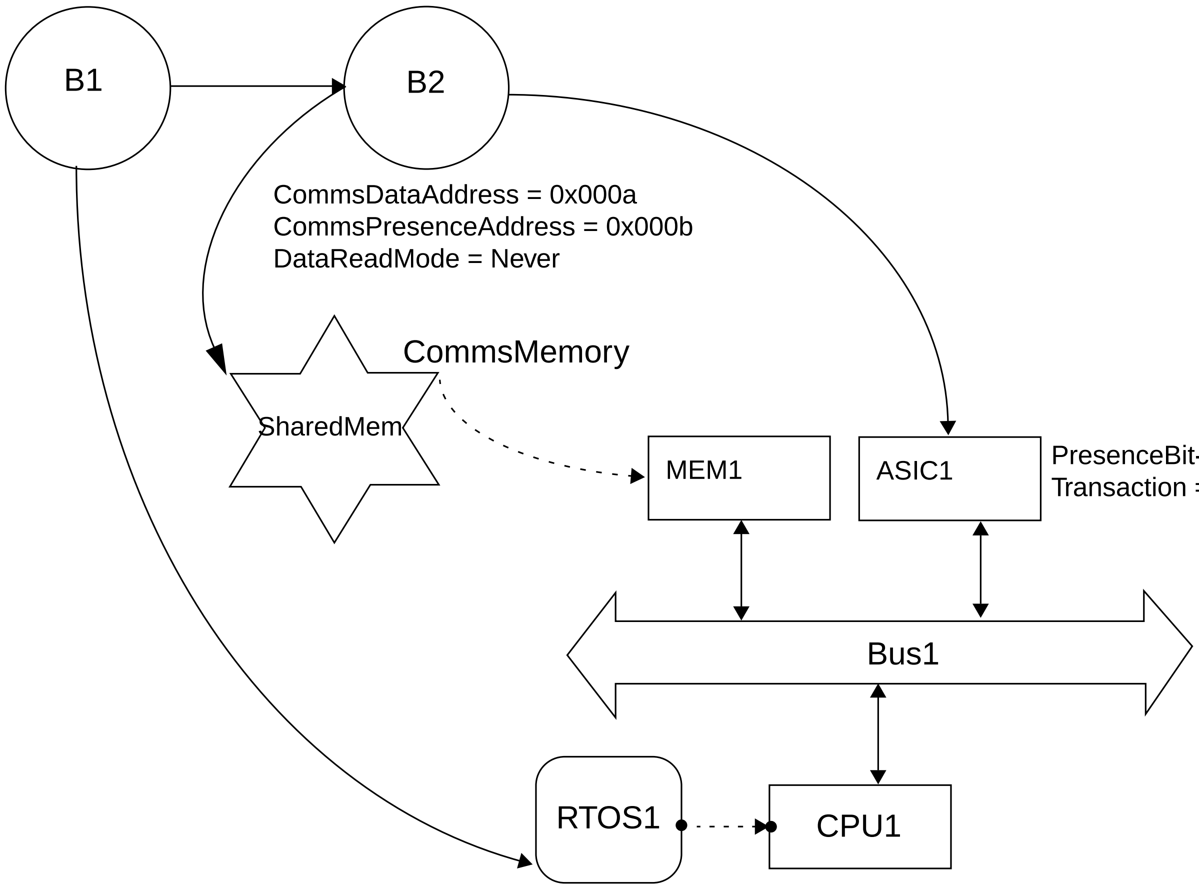
The following message sequence chart depicts token transfers and delays associated with the register mapped pattern.


Source Behavior
Data Bus
Destination Behavior
Transfer begins
Delay for bus arbitration, data transfer and presence transfer (if not disabled)
Transfer is complete

Using Shared Memory Pattern

The shared memory pattern is for transferring data to a memory on the bus. The data token is first written to the memory, then a control signal activates the ASIC. Once the ASIC is activated, it reads the data from the memory. This pattern is most appropriate for large data tokens.

You can set the *DataReadMode* to control data fetch modeling for the *Value*() call.


B1
B2
CommsDataAddress = 0x000a
CommsPresenceAddress = 0x000b
DataReadMode = Never
CommsMemory
SharedMem
MEM1
ASIC1
PresenceBit-Transaction = true
Bus1
RTOS1
CPU1

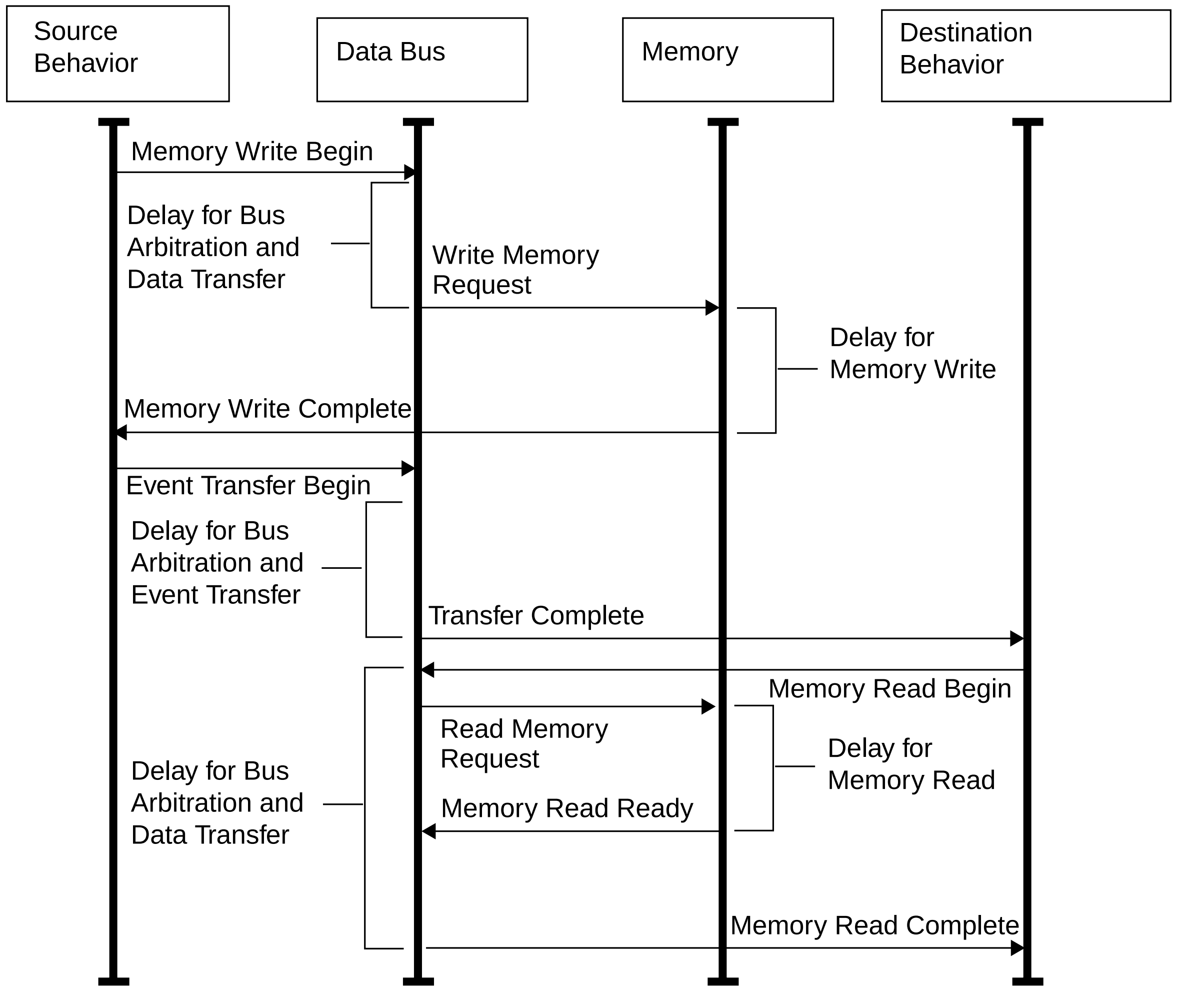
Set the following parameters on the pattern instance for shared memory.

Parameter	Data Type	Default Value
CommsMemory	@VCC_Types.MemoryParticipant	–
CommsDataAddress	@VCC_Types.DataAddress	floating
CommsPresenceAddress	@VCC_Types.TriggerAddress	floating
DataReadMode	@VCC_Types.DataReadMode	–

The following parameters are needed for the Links to Implementation flow.

Parameter	Data Type	Default Value
ReceiverMasterIndex	Integer	–
SenderMasterIndex	Integer	–

The following message sequence chart depicts token transfers and delays associated with the shared memory pattern.



Hardware-to-Hardware Communication

You can implement communication between hardware blocks using

- Direct connect pattern
- Register mapped pattern
- Shared memory pattern

Register mapped and shared memory patterns are described in “Software-to-Hardware Communication” on page 147. Direct connect is a pattern for communicating over dedicated wires.

Using Direct Connect Pattern

When there is no arbitration for the communication channel, you can configure the hardware resource to directly transfer data to another hardware resource over a dedicated wire. This pattern is commonly used for communicating between behaviors mapped to the same ASIC.

Direct connect communication requires no additional parameters.

Hardware-to-Software Communication

You can implement communication from hardware blocks to software blocks using

- Interrupt pattern
- Polling pattern

The interrupt pattern is appropriate for communication that is not regularly scheduled and needs real-time response from the software block. The polling pattern is useful when the hardware block sends data to the software block at regular intervals.

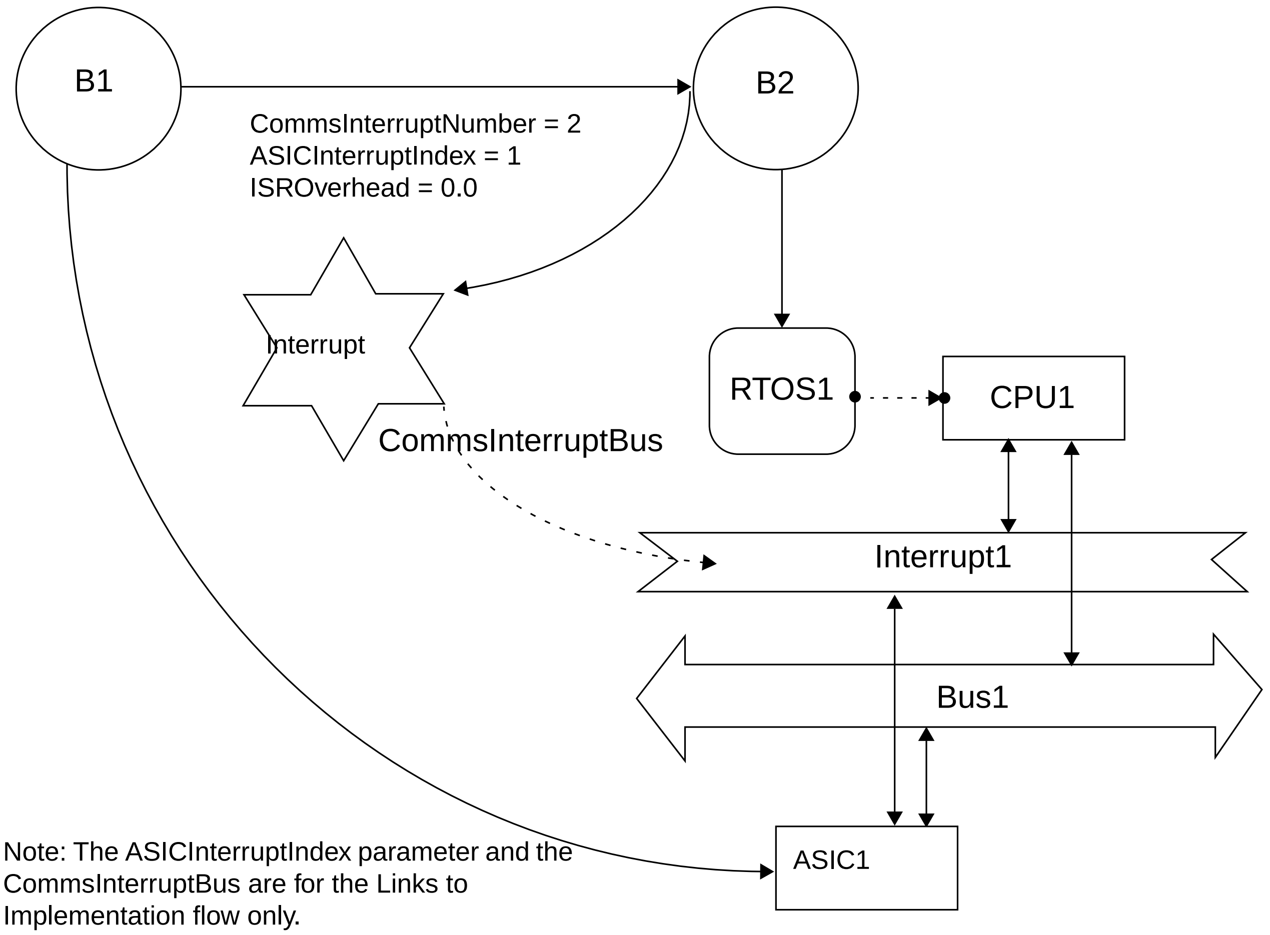
Using Interrupt Patterns

By using an interrupt pattern, the Post operation from the source behavior mapped to hardware issues an interrupt. VCC synthesizes an interrupt service routine (ISR) to handle this interrupt, which activates the destination behavior.

The ISROverhead is a link parameter that specifies the delay of executing the ISR (the default value of this parameter is zero).

Interrupt Pattern

Choose the interrupt pattern if no data is being transferred.



Note: The ASICInterruptIndex parameter and the CommsInterruptBus are for the Links to Implementation flow only.

When behavior B1 posts a value to B2, an interrupt occurs. The corresponding ISR activates behavior B2.

The interrupt bus is specified as a parameter on the pattern instance. The *CommsInterruptNumber* parameter on the mapping connection identifies the appropriate interrupt task.

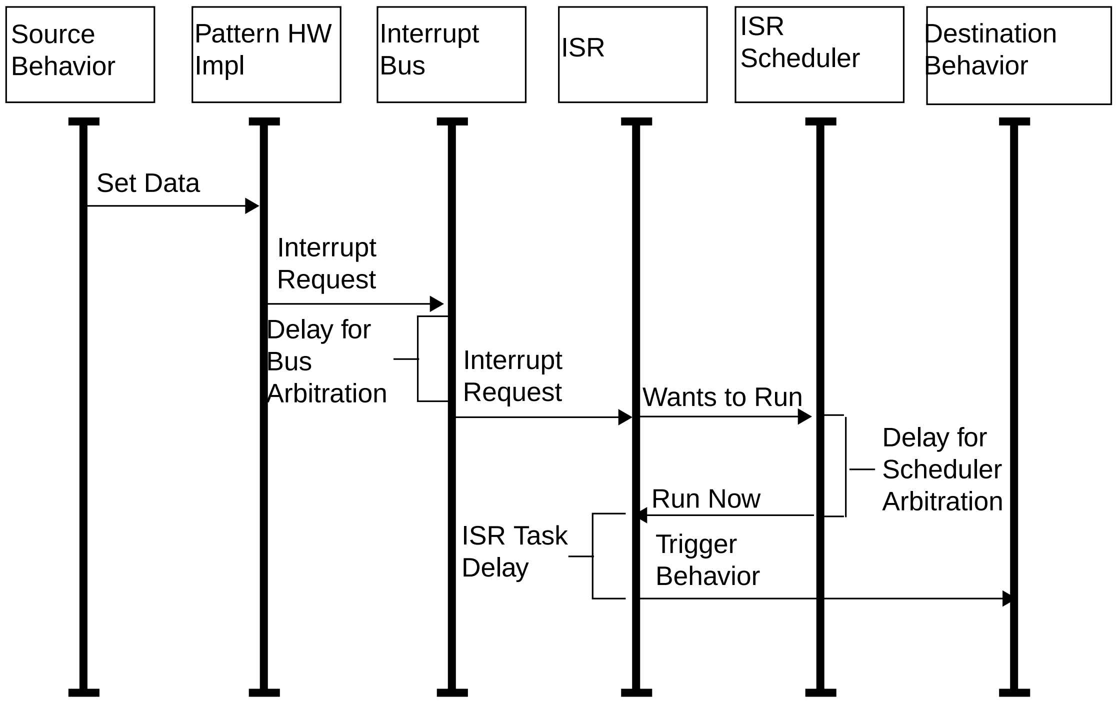
Set the following parameters on the pattern instance.

Parameter	Data Type	Default Value
CommsInterruptNumber	Integer Numbers must be assigned contiguously and must start at 1.	–
ISROverhead	Real	–

The following parameters are needed for the Links to Implementation flow.

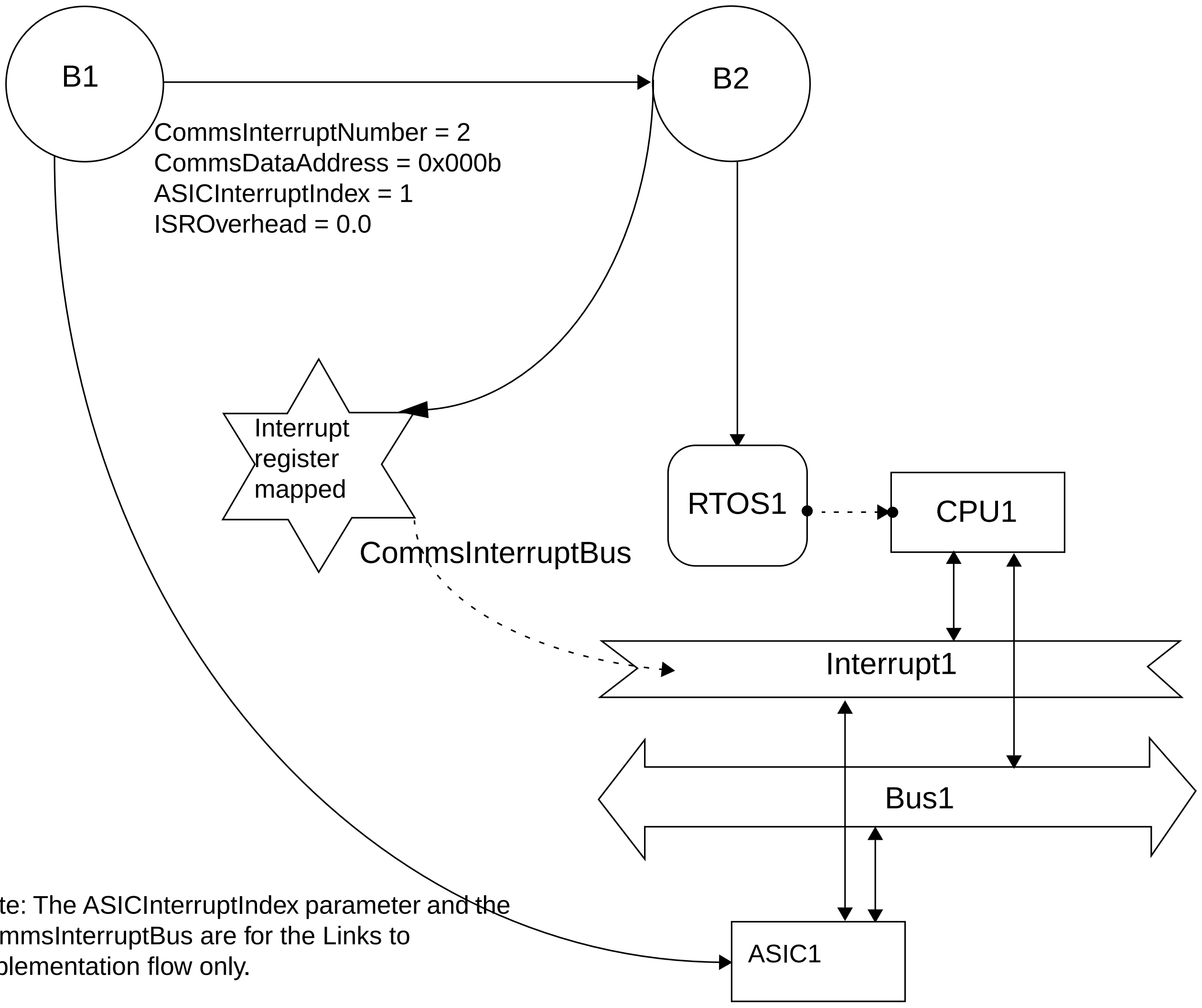
Parameter	Data Type	Default Value
CommsInterruptBus	@VCC_Types. InterruptBusParticipant	–
ASICInterruptIndex	Integer	–

The following message sequence chart depicts token transfers and delays associated with the interrupt pattern.



Interrupt Register Mapped

If data must be transferred, you can use the *interruptRegisterMapped* pattern. This pattern assumes the availability of data on a register of the ASIC. Once the ASIC behavior writes data into the register, an interrupt signal is sent by the ASIC over the interrupt bus. The interrupt task then reads the data from the register on the ASIC using the data bus.



Note: The ASICInterruptIndex parameter and the CommsInterruptBus are for the Links to Implementation flow only.

When B1 posts a value to B2, an interrupt is issued. When B2 is activated by the ISR, B2 reads from the address specified.

The data address is an offset to the starting address of the port connecting the bus to the ASIC. The data address is specified by the *CommsDataAddress* parameter on the mapping connection.

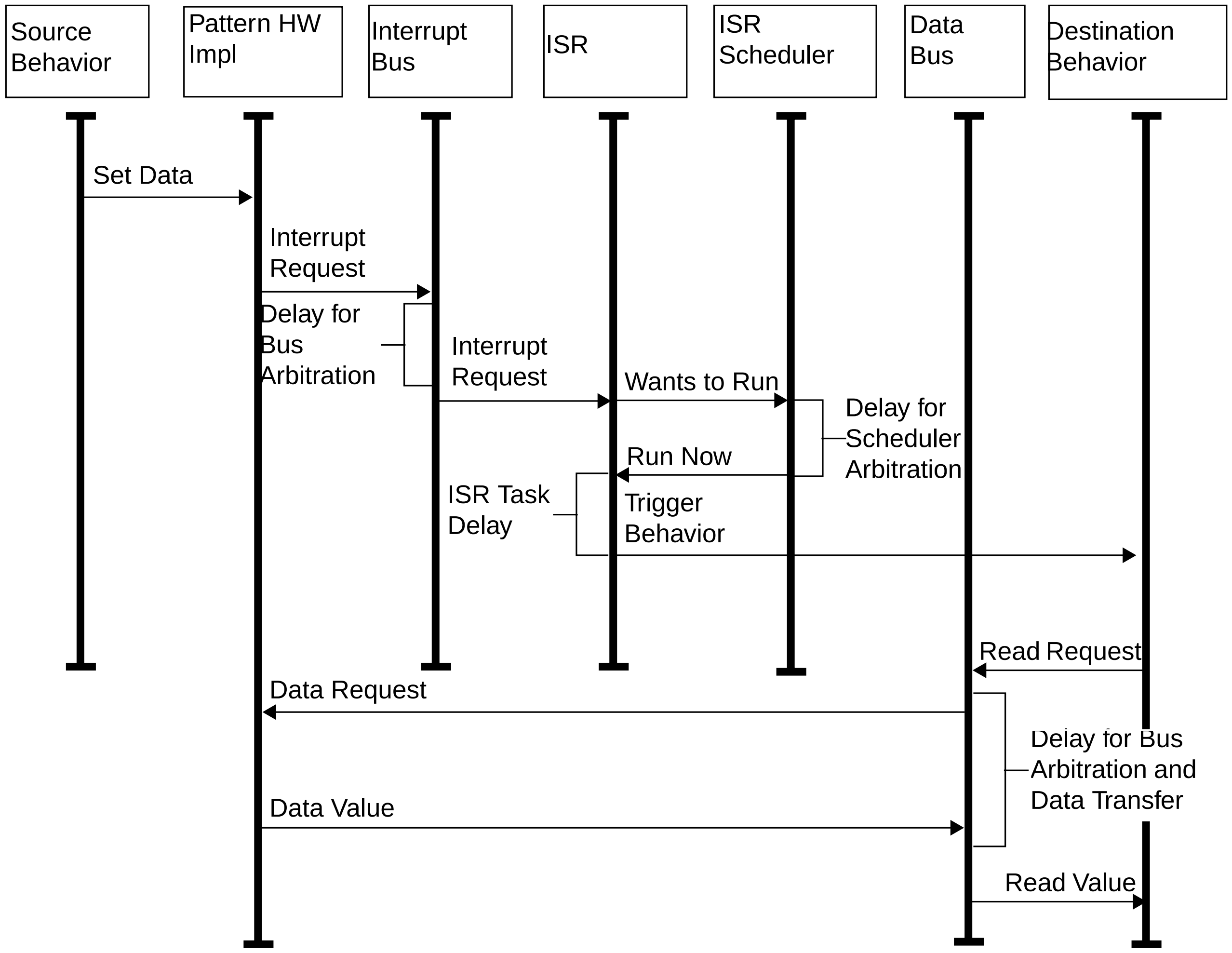
Set the following parameters on the pattern instance.

Parameter	**Data Type**	**Default Value**
CommsInterruptNumber	Integer Numbers must be assigned contiguously and must start at 1.	–
CommsDataAddress	@VCC_Types.DataAddress	floating
ISROverhead	Real	0.0

The following parameters are needed for the Links to Implementation flow.

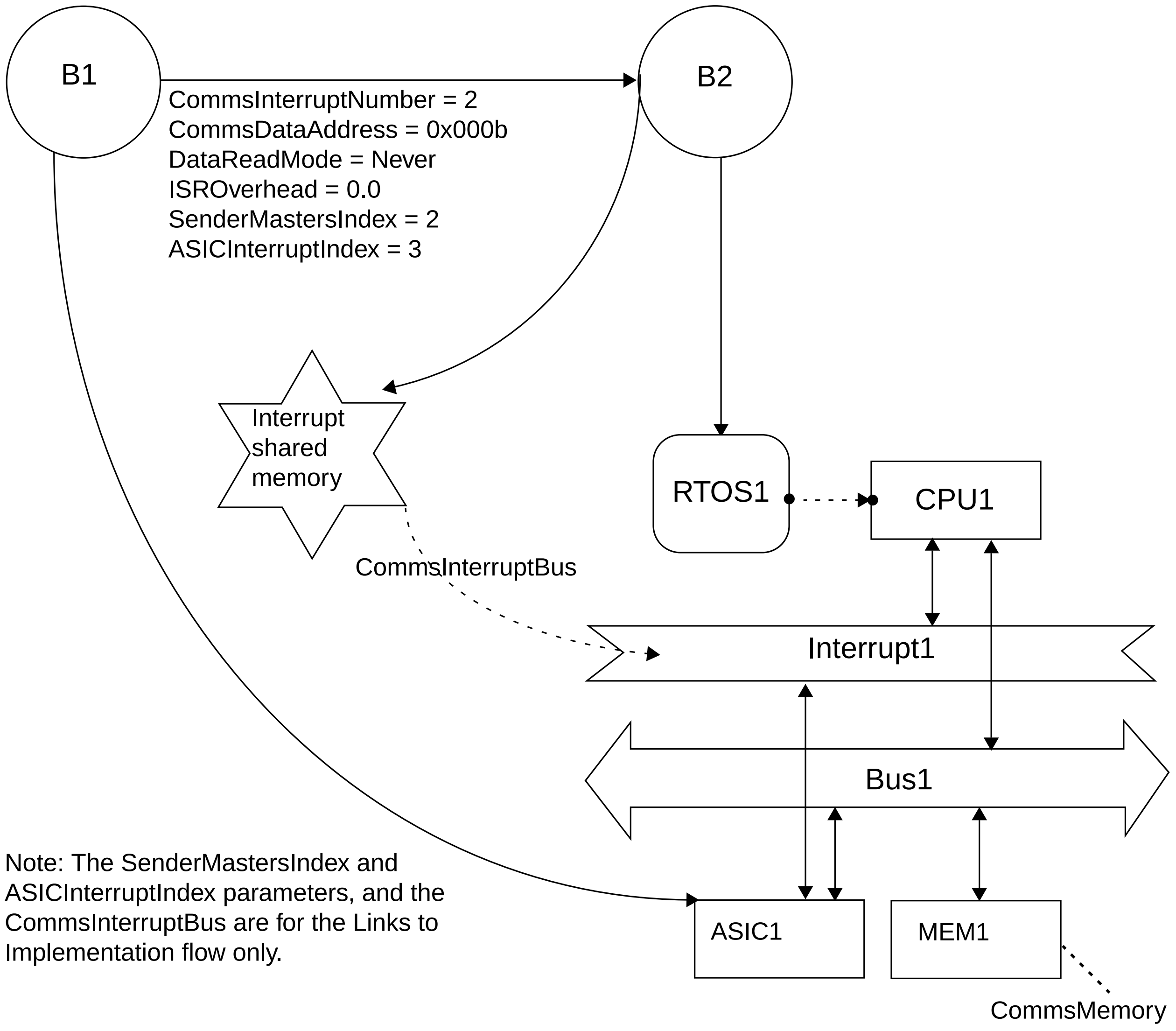
Parameter	**Data Type**	**Default Value**
CommsInterruptBus	@VCC_Types.InterruptBusParticipant	–
ASICInterruptIndex	Integer	–

The following message sequence chart depicts token transfers and delays associated with the interrupt register mapped pattern.



Interrupt Shared Memory Pattern

If data must be transferred, you can use the *interruptSharedMemory* pattern. This pattern makes the data available on a shared memory resource where the data is first written using the data bus. (The ASIC can write into shared memory only by acting as a bus master.) Then the interrupt signal is sent over the interrupt bus. The interrupt task reads data from the shared memory and activates the task.



Note: The SenderMastersIndex and ASICInterruptIndex parameters, and the CommsInterruptBus are for the Links to Implementation flow only.

The memory is specified as a parameter on the pattern instance. The data address is an offset to the starting address of the port connecting the bus to the memory. The data address is specified by the *CommsDataAddress* parameter on the mapping connection.

Set the following parameters on the pattern instance.

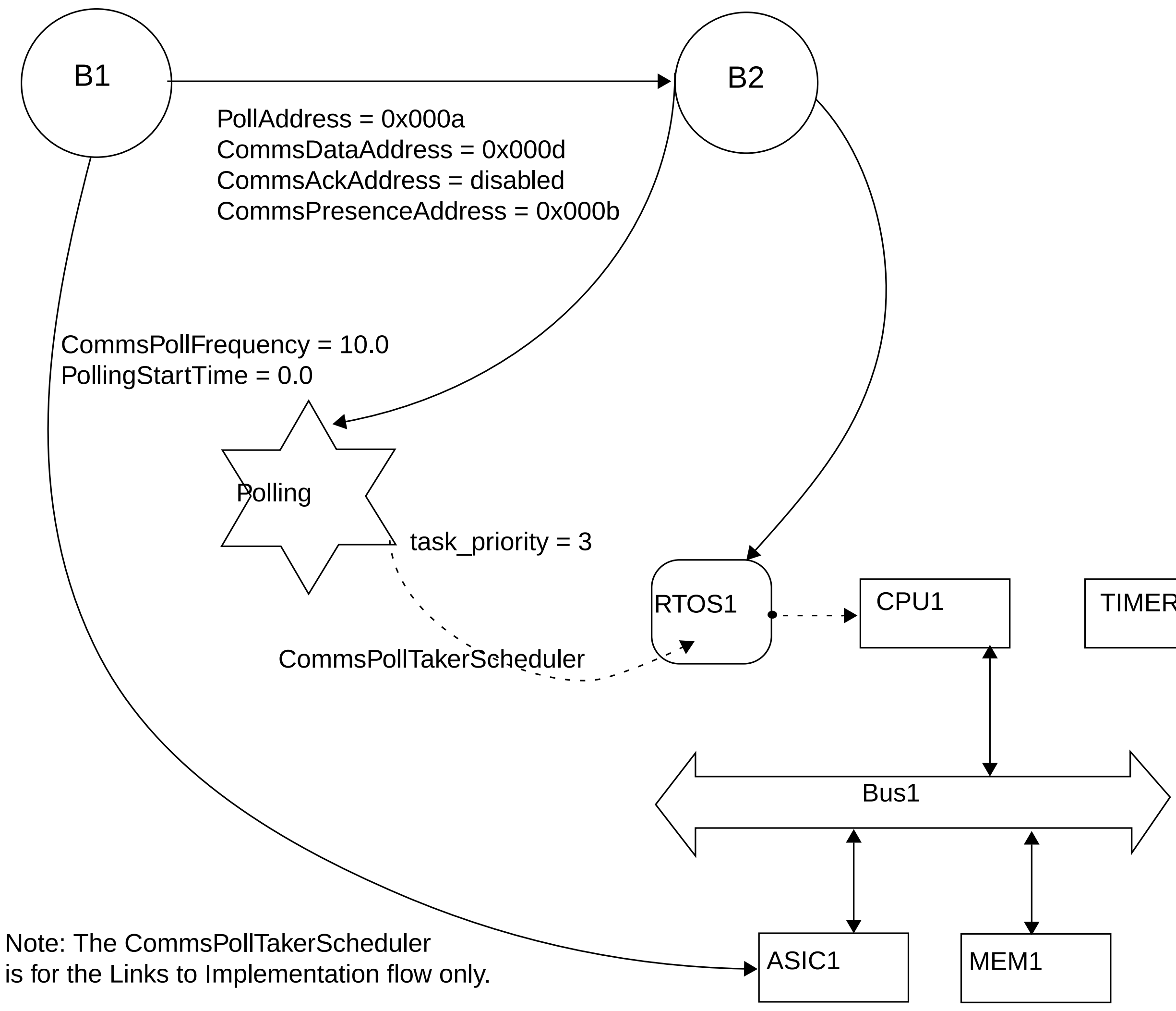
Parameter	**Data Type**	**Default Value**
CommsInterruptNumber	Integer Numbers must be assigned contiguously and must start at 1.	–
CommsMemory	@VCC_Types.MemoryParticipant	–
CommsDataAddress	@VCC_Types.DataAddress	floating
DataReadMode	@VCC_Types.DataReadMode	Never
ISROverhead	Real	0.0

The following parameters are needed for the Links to Implementation flow.

Parameter	**Data Type**	**Default Value**
CommsInterruptBus	@VCC_Types.InterruptBusParticipant	–
ASICInterruptIndex	integer	–
SenderMastersIndex	Integer	–

Using Polling Pattern

You can configure the software block to poll the hardware block for data at specified intervals. VCC synthesizes the polltaker task and schedules it on a scheduler resource that you specify. The polltaker continually checks if a specific address is ready.


B1
B2
PollAddress = 0x000a
CommsDataAddress = 0x000d
CommsAckAddress = disabled
CommsPresenceAddress = 0x000b
CommsPollFrequency = 10.0
PollingStartTime = 0.0
Polling
task_priority = 3
RTOS1
CPU1
TIMER1
CommsPollTakerScheduler
Bus1
Note: The CommsPollTakerScheduler
is for the Links to Implementation flow only.
ASIC1
MEM1

The polling task periodically reads from the poll address to see if any data is available. When Behavior B1 writes to this address and the address is polled, the polling task activates behavior B2. The polling task writes to the *CommsAckAddress* to acknowledge that the data has been detected. B2 then reads the data from the *CommsDataAddress*.

The *CommsPolltakerScheduler* parameter on the pattern instance identifies the scheduler resource on which the polltaker is scheduled. Any parameters required to schedule the polltaker on the scheduler are also specified on the pattern instance. For example, the scheduler might be a priority-based scheduler, which requires that you also configure the *task_priority* parameter.

The *CommsPollFrequency* parameter on the pattern instance specifies the frequency at which the polltaker task is activated.

If the data is large, you can configure the data to be written to and read from shared memory. To do this, you must configure an additional parameter, *CommsMemory*, on the pattern instance. In this scenario, the *CommsDataAddress* identifies the offset into the memory where the data resides.

If the hardware resource needs an acknowledgment that the communication completed, you can configure *CommsAcknowledgeAddress* as a mapping parameter. By default the value of this address is *invalid*, indicating an acknowledgment is not required. This identifies the register on the ASIC for acknowledgment.

Note: The Polltaker task is not simulated—therefore the *CommsAcknowledgeAddress* is not used in simulation.

Set the following parameters on the pattern instance.

Parameter	Data Type	Default Value
CommsPollFrequency	Real	–
CommsAcknowledgeAddress	@VCC_Types. TriggerAddress	disabled
CommsDataAddress	@VCC_Types.DataAddress	floating
CommsMemory *	@VCC_Types. MemoryParticipant	–
CommsPresenceAddress	@VCC_Types. TriggerAddress	floating
PollingStartTime	Real	–

***** The *PollingSharedMemory* pattern requires this additional parameter.

The following parameter is needed for the Links to Implementation flow.

Parameter	Data Type	Default Value
CommsPolltakerScheduler	@VCC_Types. SchedulerParticipant	–

Software-to-Software Communication

You can implement communication between software blocks using

- Unprotected pattern
- Interrupt protected pattern
- Semaphore protected pattern

Using Unprotected Pattern

You can configure the software blocks so that data integrity is not protected between data reads and writes. This configuration is efficient for variables where protection is not needed, such as when the variables are within the same software task. For example, for two behaviors mapped to the same single-task resource. In this case, VCC synthesizes a single static schedule of each behavior and the communication between these behaviors is implemented as local variables.

The unprotected communication pattern requires no additional parameters.

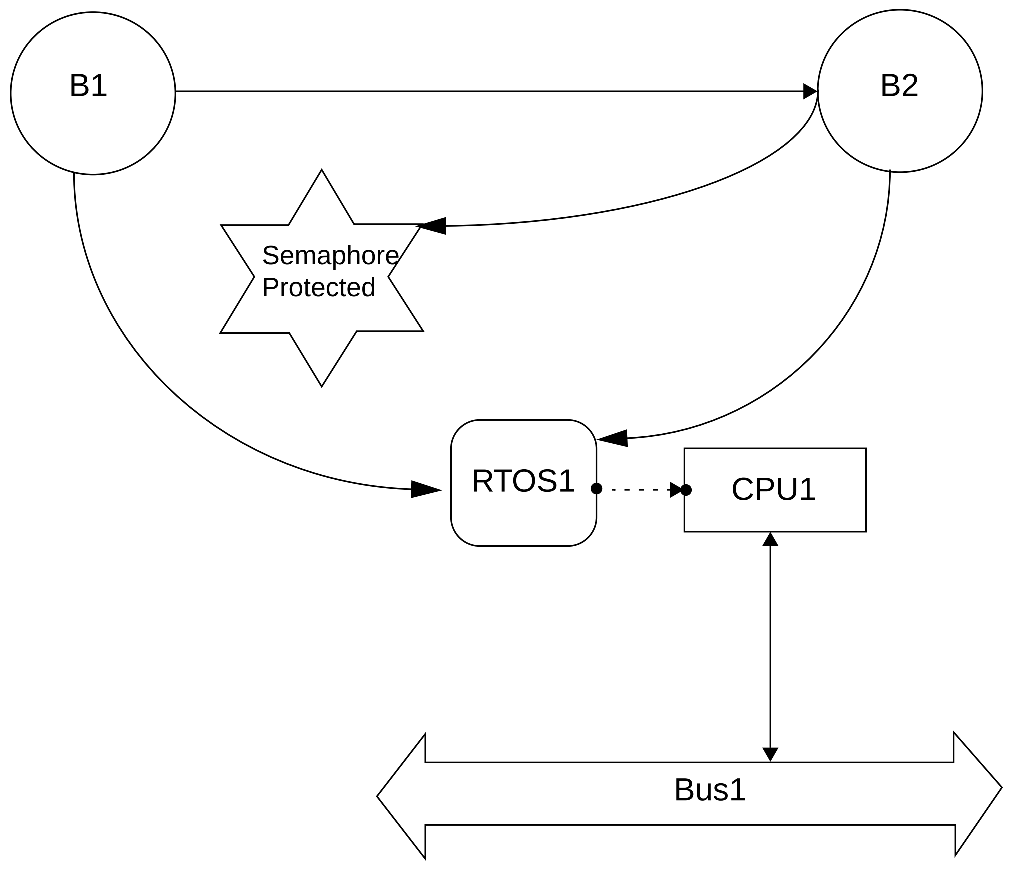
Using Uninterruptable Protected Pattern

By masking interrupts, you can configure the software blocks to protect the data token from being written while it is being read.

The uninterruptable protected communication pattern requires no additional parameters.

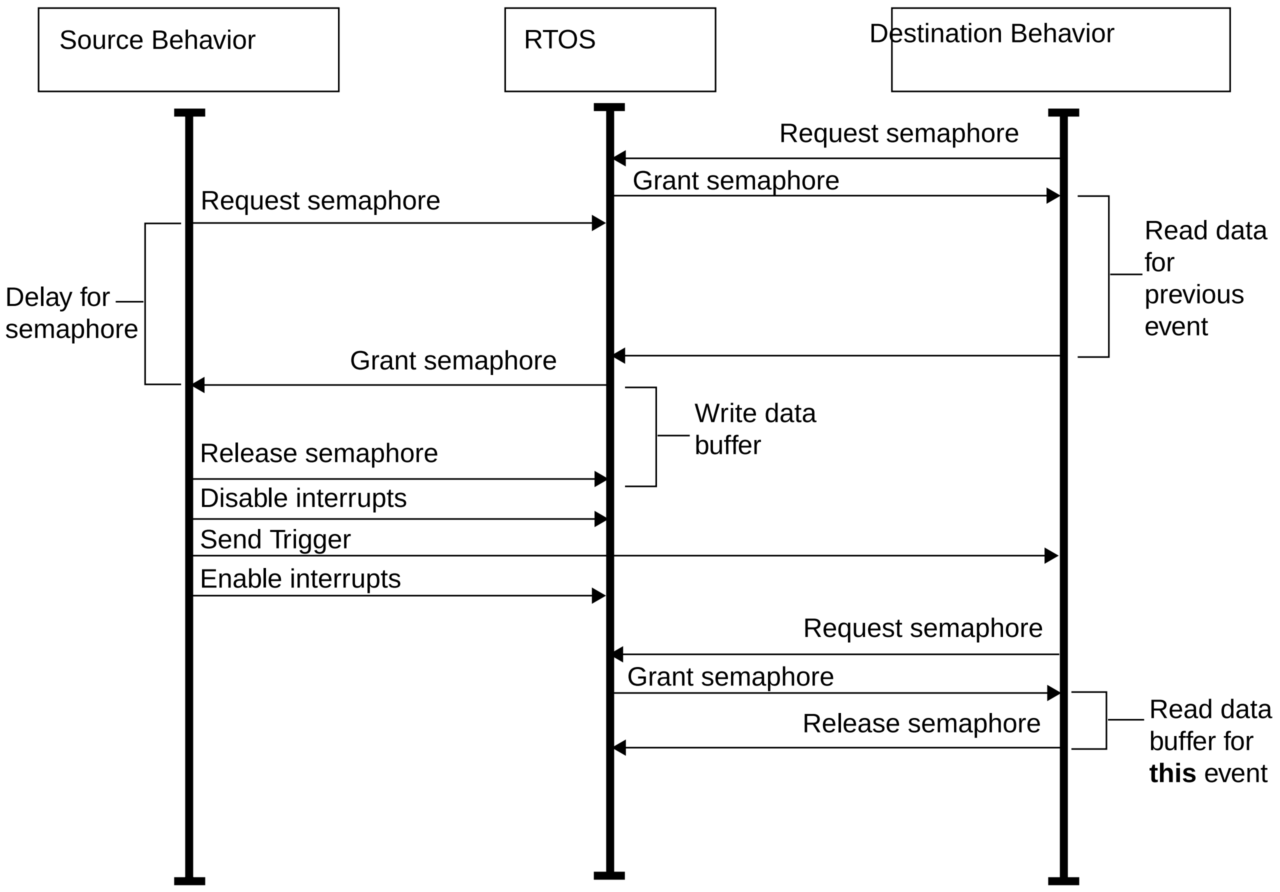
Using Semaphore Protected Pattern

Using RTOS services called “*semaphores*,” you can configure the software block to protect the data token from being written while it is being read.



The semaphore protected communication pattern requires no additional parameters.

The following message sequence chart depicts token transfers and delays associated with the semaphore protected pattern.


Source Behavior
RTOS
Destination Behavior
Request semaphore
Grant semaphore
Request semaphore
Read data for previous event
Delay for semaphore
Grant semaphore
Write data buffer
Release semaphore
Disable interrupts
Send Trigger
Enable interrupts
Request semaphore
Grant semaphore
Release semaphore
Read data buffer for **this** event